HAL open science

Statistics and Machine Learning in Python

Edouard Duchesnay, Tommy Lofstedt, Feki Younes

To cite this version:

Edouard Duchesnay, Tommy Lofstedt, Feki Younes. Statistics and Machine Learning in Python. Engineering school. France. 2021. hal-03038776v3

HAL Id: hal-03038776

https://hal.science/hal-03038776v3

Submitted on 2 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Statistics and Machine Learning in Python

Release 0.5

Edouard Duchesnay, Tommy Löfstedt, Feki Younes

Sep 02, 2021

CONTENTS

1 Introduction **1**
1.1 Python ecosystem for data-science . . . 1
1.2 Introduction to Machine Learning . . . 6
1.3 Data analysis methodology . . . 7

2 Python language **9**
2.1 Import libraries . . . 9
2.2 Basic operations . . . 9
2.3 Data types . . . 10
2.4 Execution control statements . . . 18
2.5 List comprehensions, iterators, etc. . . . 19
2.6 Functions . . . 22
2.7 Regular expression . . . 24
2.8 System programming . . . 25
2.9 Scripts and argument parsing . . . 32
2.10 Networking . . . 33
2.11 Modules and packages . . . 34
2.12 Object Oriented Programming (OOP) . . . 35
2.13 Style guide for Python programming . . . 36
2.14 Documenting . . . 36
2.15 Exercises . . . 38

3 Scientific Python **39**
3.1 Numpy: arrays and matrices . . . 39
3.2 Pandas: data manipulation . . . 50
3.3 Data visualization: matplotlib & seaborn . . . 65

4 Statistics **79**
4.1 Univariate statistics . . . 79
4.2 Lab: Brain volumes study . . . 119
4.3 Linear Mixed Models . . . 131
4.4 Multivariate statistics . . . 150
4.5 Time series in python . . . 162

5 Machine Learning **179**
5.1 Linear dimension reduction and feature extraction . . . 179
5.2 Manifold learning: non-linear dimension reduction . . . 192
5.3 Clustering . . . 198
5.4 Linear models for regression problems . . . 207

5.5 Linear models for classification problems 221
5.6 Non-linear models 240
5.7 Resampling methods 245
5.8 Ensemble learning: bagging, boosting and stacking 261
5.9 Gradient descent 276
5.10 Lab: Faces recognition using various learning models 285

6 Deep Learning **305**
6.1 Backpropagation 305
6.2 Multilayer Perceptron (MLP) 320
6.3 Convolutional neural network 342
6.4 Transfer Learning Tutorial 373

7 Indices and tables **383**

CHAPTER

ONE

INTRODUCTION

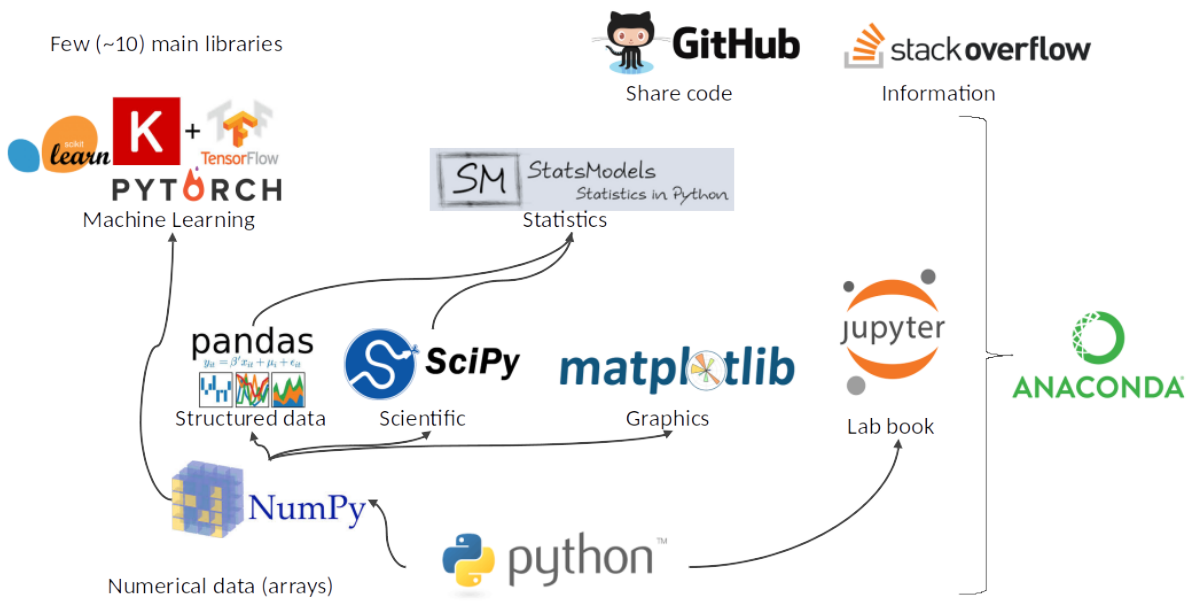
Important links:

- Web page
- Github
- Latest pdf
- Official deposit for citation.

This document describes statistics and machine learning in Python using:

- Scikit-learn for machine learning.
- Pytorch for deep learning.
- Statsmodels for statistics.

1.1 Python ecosystem for data-science



1.1.1 Python language

- Python is popular Google trends
- Interpreted: source file `.py` => interpretor => processor versus compiled, two steps (1) source file => compilaton => binary file. (2) Execution is simply binary file => processor.
- Garbage collector (do not prevent from memory leak)
- Dynamically-typed language (Java is statically typed)

1.1.2 Anaconda

Anaconda is a python distribution that ships most of python tools and libraries

Installation

1. Download anaconda (Python 3.x) http://continuum.io/downloads

2. Install it, on Linux

```
bash Anaconda3-2.4.1-Linux-x86_64.sh
```

3. Add anaconda path in your PATH variable in your `.bashrc` file:

```
export PATH="${HOME}/anaconda3/bin:$PATH"
```

Managing with ``conda``

Update conda package and environment manager to current version

```
conda update conda
```

Install additional packages. Those commands install qt back-end (Fix a temporary issue to run spyder)

```
conda install pyqt
conda install PyOpenGL
conda update --all
```

Install seaborn for graphics

```
conda install seaborn
# install a specific version from anaconda chanel
conda install -c anaconda pyqt=4.11.4
```

List installed packages

```
conda list
```

Search available packages

```
conda search pyqt
conda search scikit-learn
```

Environments

- A conda environment is a directory that contains a specific collection of conda packages that you have installed.
- Control packages environment for a specific purpose: collaborating with someone else, delivering an application to your client,
- Switch between environments

List of all environments

:: conda info –envs

1. Create new environment
2. Activate
3. Install new package

```
conda create --name test
# Or
conda env create -f environment.yml
source activate test
conda info --envs
conda list
conda search -f numpy
conda install numpy
```

Miniconda

Anaconda without the collection of (>700) packages. With Miniconda you download only the packages you want with the conda command: `conda install PACKAGENAME`

1. Download anaconda (Python 3.x) https://conda.io/miniconda.html
2. Install it, on Linux

```
bash Miniconda3-latest-Linux-x86_64.sh
```

3. Add anaconda path in your PATH variable in your `.bashrc` file:

```
export PATH=${HOME}/miniconda3/bin:$PATH
```

4. Install required packages

```
conda install -y scipy
conda install -y pandas
conda install -y matplotlib
conda install -y statsmodels
conda install -y scikit-learn
conda install -y sqlite
conda install -y spyder
conda install -y jupyter
```

1.1.3 Commands

python: python interpreter. On the dos/unix command line execute wholes file:

```
python file.py
```

Interactive mode:

```
python
```

Quite with `CTL-D`

ipython: advanced interactive python interpreter:

```
ipython
```

Quite with `CTL-D`

pip alternative for packages management (update `-U` in user directory `--user`):

```
pip install -U --user seaborn
```

For neuroimaging:

```
pip install -U --user nibabel
pip install -U --user nilearn
```

spyder: IDE (integrated development environment):

- Syntax highlighting.
- Code introspection for code completion (use `TAB`).
- Support for multiple Python consoles (including IPython).
- Explore and edit variables from a GUI.
- Debugging.
- Navigate in code (go to function definition) `CTL`.

3 or 4 panels:

text editor	help/variable explorer
	ipython interpreter

Shortcuts: - `F9` run line/selection

1.1.4 Libraries

scipy.org: https://www.scipy.org/docs.html

Numpy: Basic numerical operation. Matrix operation plus some basic solvers.:

```
import numpy as np
X = np.array([[1, 2], [3, 4]])
#v = np.array([1, 2]).reshape((2, 1))
v = np.array([1, 2])
np.dot(X, v) # no broadcasting
X * v # broadcasting
np.dot(v, X)
X - X.mean(axis=0)
```

Scipy: general scientific libraries with advanced solver:

```
import scipy
import scipy.linalg
scipy.linalg.svd(X, full_matrices=False)
```

Matplotlib: visualization:

```
import numpy as np
import matplotlib.pyplot as plt
#%matplotlib qt
x = np.linspace(0, 10, 50)
sinus = np.sin(x)
plt.plot(x, sinus)
plt.show()
```

Pandas: Manipulation of structured data (tables). input/output excel files, etc.

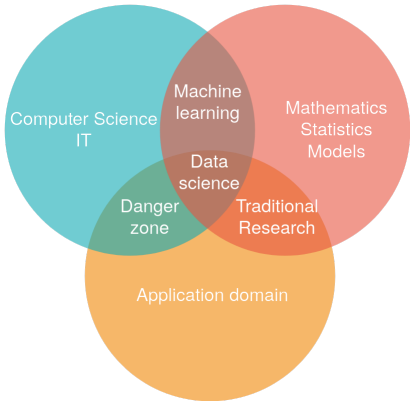
Statsmodel: Advanced statistics

Scikit-learn: Machine learning

library	Arrays data, Num. comp, I/O	Structured data, I/O	Solvers: basic	Solvers: advanced	Stats: basic	Stats: advanced	Machine learning
Numpy	X		X				
Scipy			X	X	X		
Pandas		X					
Statmodels					X	X	
Scikit-learn							X

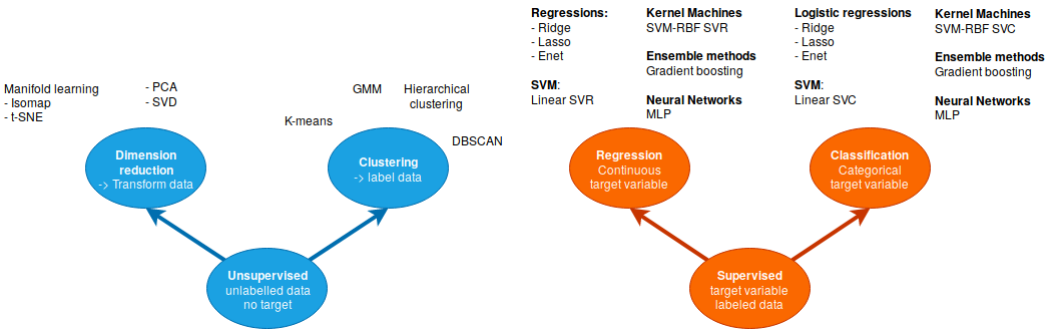
1.2 Introduction to Machine Learning

1.2.1 Machine learning within data science



Machine learning covers two main types of data analysis:

1. Exploratory analysis: **Unsupervised learning**. Discover the structure within the data. E.g.: Experience (in years in a company) and salary are correlated.
2. Predictive analysis: **Supervised learning**. This is sometimes described as **"learn from the past to predict the future"**. Scenario: a company wants to detect potential future clients among a base of prospects. Retrospective data analysis: we go through the data constituted of previous prospected companies, with their characteristics (size, domain, localization, etc. . .). Some of these companies became clients, others did not. The question is, can we possibly predict which of the new companies are more likely to become clients, based on their characteristics based on previous observations? In this example, the training data consists of a set of n training samples. Each sample, x_i, is a vector of p input features (company characteristics) and a target feature ($y_i \in \{Yes, No\}$ (whether they became a client or not).



1.2.2 IT/computing science tools

- High Performance Computing (HPC)
- Data flow, data base, file I/O, etc.
- Python: the programming language.
- Numpy: python library particularly useful for handling of raw numerical data (matrices, mathematical operations).

- Pandas: input/output, manipulation structured data (tables).

1.2.3 Statistics and applied mathematics

- Linear model.
- Non parametric statistics.
- Linear algebra: matrix operations, inversion, eigenvalues.

1.3 Data analysis methodology

1. **Formalize customer's needs into a learning problem:**
    - **A target variable: supervised problem.**
        - Target is qualitative: classification.
        - Target is quantitative: regression.
    - **No target variable: unsupervised problem**
        - Vizualisation of high-dimensional samples: PCA, manifolds learning, etc.
        - Finding groups of samples (hidden structure): clustering.
2. **Ask question about the datasets**
    - Number of samples
    - Number of variables, types of each variable.
3. **Define the sample**
    - For prospective study formalize the experimental design: inclusion/exlusion criteria. The conditions that define the acquisition of the dataset.
    - For retrospective study formalize the experimental design: inclusion/exlusion criteria. The conditions that define the selection of the dataset.
4. In a document formalize (i) the project objectives; (ii) the required learning dataset (more specifically the input data and the target variables); (iii) The conditions that define the acquisition of the dataset. In this document, warn the customer that the learned algorithms may not work on new data acquired under different condition.
5. Read the learning dataset.
6. (i) Sanity check (basic descriptive statistics); (ii) data cleaning (impute missing data, recoding); Final Quality Control (QC) perform descriptive statistics and think ! (remove possible confounding variable, etc.).
7. Explore data (visualization, PCA) and perform basic univariate statistics for association between the target an input variables.
8. Perform more complex multivariate-machine learning.
9. Model validation using a left-out-sample strategy (cross-validation, etc.).
10. Apply on new data.

CHAPTER

TWO

PYTHON LANGUAGE

Source Kevin Markham https://github.com/justmarkham/python-reference

2.1 Import libraries

```
# 'generic import' of math module
import math
math.sqrt(25)

# import a function
from math import sqrt
sqrt(25)    # no longer have to reference the module

# import multiple functions at once
from math import cos, floor

# import all functions in a module (generally discouraged)
# from os import *

# define an alias
import numpy as np

# show all functions in math module
content = dir(math)
```

2.2 Basic operations

```
# Numbers
10 + 4          # add (returns 14)
10 - 4          # subtract (returns 6)
10 * 4          # multiply (returns 40)
10 ** 4         # exponent (returns 10000)
10 / 4          # divide (returns 2 because both types are 'int')
10 / float(4)   # divide (returns 2.5)
5 % 4           # modulo (returns 1) - also known as the remainder
```

(continues on next page)

(continued from previous page)

```
10 / 4          # true division (returns 2.5)
10 // 4         # floor division (returns 2)


# Boolean operations
# comparisons (these return True)
5 > 3
5 >= 3
5 != 3
5 == 5

# boolean operations (these return True)
5 > 3 and 6 > 3
5 > 3 or 5 < 3
not False
False or not False and True     # evaluation order: not, and, or
```

Out:

```
True
```

2.3 Data types

```
# determine the type of an object
type(2)         # returns 'int'
type(2.0)       # returns 'float'
type('two')     # returns 'str'
type(True)      # returns 'bool'
type(None)      # returns 'NoneType'

# check if an object is of a given type
isinstance(2.0, int)            # returns False
isinstance(2.0, (int, float))   # returns True

# convert an object to a given type
float(2)
int(2.9)
str(2.9)

# zero, None, and empty containers are converted to False
bool(0)
bool(None)
bool('')    # empty string
bool([])    # empty list
bool({})    # empty dictionary

# non-empty containers and non-zeros are converted to True
```

(continues on next page)

(continued from previous page)

```
bool(2)
bool('two')
bool([2])
```

Out:

```
True
```

2.3.1 Lists

Different objects categorized along a certain ordered sequence, lists are ordered, iterable, mutable (adding or removing objects changes the list size), can contain multiple data types.

```
# create an empty list (two ways)
empty_list = []
empty_list = list()

# create a list
simpsons = ['homer', 'marge', 'bart']

# examine a list
simpsons[0]     # print element 0 ('homer')
len(simpsons)   # returns the length (3)

# modify a list (does not return the list)
simpsons.append('lisa')                 # append element to end
simpsons.extend(['itchy', 'scratchy'])  # append multiple elements to end
simpsons.insert(0, 'maggie')            # insert element at index 0 (shifts␣
↪everything right)
simpsons.remove('bart')                 # searches for first instance and removes␣
↪it
simpsons.pop(0)                         # removes element 0 and returns it
del simpsons[0]                         # removes element 0 (does not return it)
simpsons[0] = 'krusty'                  # replace element 0

# concatenate lists (slower than 'extend' method)
neighbors = simpsons + ['ned','rod','todd']

# find elements in a list
'lisa' in simpsons
simpsons.count('lisa')      # counts the number of instances
simpsons.index('itchy')     # returns index of first instance

# list slicing [start:end:stride]
weekdays = ['mon','tues','wed','thurs','fri']
weekdays[0]         # element 0
weekdays[0:3]       # elements 0, 1, 2
weekdays[:3]        # elements 0, 1, 2
weekdays[3:]        # elements 3, 4
```

(continues on next page)

(continued from previous page)

```
weekdays[-1]        # last element (element 4)
weekdays[::2]       # every 2nd element (0, 2, 4)
weekdays[::-1]      # backwards (4, 3, 2, 1, 0)

# alternative method for returning the list backwards
list(reversed(weekdays))

# sort a list in place (modifies but does not return the list)
simpsons.sort()
simpsons.sort(reverse=True)     # sort in reverse
simpsons.sort(key=len)          # sort by a key

# return a sorted list (but does not modify the original list)
sorted(simpsons)
sorted(simpsons, reverse=True)
sorted(simpsons, key=len)

# create a second reference to the same list
num = [1, 2, 3]
same_num = num
same_num[0] = 0         # modifies both 'num' and 'same_num'

# copy a list (three ways)
new_num = num.copy()
new_num = num[:]
new_num = list(num)

# examine objects
id(num) == id(same_num) # returns True
id(num) == id(new_num)  # returns False
num is same_num         # returns True
num is new_num          # returns False
num == same_num         # returns True
num == new_num          # returns True (their contents are equivalent)

# conatenate +, replicate *
[1, 2, 3] + [4, 5, 6]
["a"] * 2 + ["b"] * 3
```

Out:

```
['a', 'a', 'b', 'b', 'b']
```

2.3.2 Tuples

Like lists, but their size cannot change: ordered, iterable, immutable, can contain multiple data types

```
# create a tuple
digits = (0, 1, 'two')          # create a tuple directly
digits = tuple([0, 1, 'two'])   # create a tuple from a list
zero = (0,)                     # trailing comma is required to indicate it's a␣
↪tuple

# examine a tuple
digits[2]           # returns 'two'
len(digits)         # returns 3
digits.count(0)     # counts the number of instances of that value (1)
digits.index(1)     # returns the index of the first instance of that value (1)

# elements of a tuple cannot be modified
# digits[2] = 2       # throws an error

# concatenate tuples
digits = digits + (3, 4)

# create a single tuple with elements repeated (also works with lists)
(3, 4) * 2          # returns (3, 4, 3, 4)

# tuple unpacking
bart = ('male', 10, 'simpson')  # create a tuple
```

2.3.3 Strings

A sequence of characters, they are iterable, immutable

```
# create a string
s = str(42)         # convert another data type into a string
s = 'I like you'

# examine a string
s[0]                # returns 'I'
len(s)              # returns 10

# string slicing like lists
s[:6]               # returns 'I like'
s[7:]               # returns 'you'
s[-1]               # returns 'u'

# basic string methods (does not modify the original string)
s.lower()           # returns 'i like you'
s.upper()           # returns 'I LIKE YOU'
s.startswith('I')   # returns True
```

(continues on next page)

(continued from previous page)

```
s.endswith('you')   # returns True
s.isdigit()         # returns False (returns True if every character in the␣
↪string is a digit)
s.find('like')      # returns index of first occurrence (2), but doesn't support␣
↪regex
s.find('hate')      # returns -1 since not found
s.replace('like','love')    # replaces all instances of 'like' with 'love'

# split a string into a list of substrings separated by a delimiter
s.split(' ')        # returns ['I','like','you']
s.split()           # same thing
s2 = 'a, an, the'
s2.split(',')       # returns ['a',' an',' the']

# join a list of strings into one string using a delimiter
stooges = ['larry','curly','moe']
' '.join(stooges)   # returns 'larry curly moe'

# concatenate strings
s3 = 'The meaning of life is'
s4 = '42'
s3 + ' ' + s4       # returns 'The meaning of life is 42'
s3 + ' ' + str(42)  # same thing

# remove whitespace from start and end of a string
s5 = '  ham and cheese  '
s5.strip()          # returns 'ham and cheese'

# string substitutions: all of these return 'raining cats and dogs'
'raining %s and %s' % ('cats','dogs')                      # old way
'raining {} and {}'.format('cats','dogs')                  # new way
'raining {arg1} and {arg2}'.format(arg1='cats',arg2='dogs') # named arguments

# string formatting
# more examples: http://mkaz.com/2012/10/10/python-string-format/
'pi is {:.2f}'.format(3.14159)      # returns 'pi is 3.14'
```

Out:

```
'pi is 3.14'
```

2.3.4 Strings 2/2

Normal strings allow for escaped characters

```
print('first line\nsecond line')
```

Out:

```
first line
second line
```

raw strings treat backslashes as literal characters

```
print(r'first line\nfirst line')
```

Out:

```
first line\nfirst line
```

Sequence of bytes are not strings, should be decoded before some operations

```
s = b'first line\nsecond line'
print(s)

print(s.decode('utf-8').split())
```

Out:

```
b'first line\nsecond line'
['first', 'line', 'second', 'line']
```

2.3.5 Dictionaries

Dictionaries are structures which can contain multiple data types, and is ordered with key-value pairs: for each (unique) key, the dictionary outputs one value. Keys can be strings, numbers, or tuples, while the corresponding values can be any Python object. Dictionaries are: unordered, iterable, mutable

```
# create an empty dictionary (two ways)
empty_dict = {}
empty_dict = dict()

# create a dictionary (two ways)
family = {'dad':'homer', 'mom':'marge', 'size':6}
family = dict(dad='homer', mom='marge', size=6)

# convert a list of tuples into a dictionary
list_of_tuples = [('dad','homer'), ('mom','marge'), ('size', 6)]
family = dict(list_of_tuples)

# examine a dictionary
```

(continues on next page)

(continued from previous page)

```
family['dad']       # returns 'homer'
len(family)         # returns 3
family.keys()       # returns list: ['dad', 'mom', 'size']
family.values()     # returns list: ['homer', 'marge', 6]
family.items()      # returns list of tuples:
                    #   [('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
'mom' in family     # returns True
'marge' in family   # returns False (only checks keys)

# modify a dictionary (does not return the dictionary)
family['cat'] = 'snowball'              # add a new entry
family['cat'] = 'snowball ii'           # edit an existing entry
del family['cat']                       # delete an entry
family['kids'] = ['bart', 'lisa']       # value can be a list
family.pop('dad')                       # removes an entry and returns the value (
↪'homer')
family.update({'baby':'maggie', 'grandpa':'abe'})   # add multiple entries

# accessing values more safely with 'get'
family['mom']                       # returns 'marge'
family.get('mom')                   # same thing
try:
    family['grandma']               # throws an error
except  KeyError as e:
    print("Error", e)

family.get('grandma')               # returns None
family.get('grandma', 'not found')  # returns 'not found' (the default)

# accessing a list element within a dictionary
family['kids'][0]                   # returns 'bart'
family['kids'].remove('lisa')       # removes 'lisa'

# string substitution using a dictionary
'youngest child is %(baby)s' % family   # returns 'youngest child is maggie'
```

Out:

```
Error 'grandma'

'youngest child is maggie'
```

2.3.6 Sets

Like dictionaries, but with unique keys only (no corresponding values). They are: unordered, iterable, mutable, can contain multiple data types made up of unique elements (strings, numbers, or tuples)

```
# create an empty set
empty_set = set()

# create a set
languages = {'python', 'r', 'java'}         # create a set directly
snakes = set(['cobra', 'viper', 'python'])  # create a set from a list

# examine a set
len(languages)              # returns 3
'python' in languages       # returns True

# set operations
languages & snakes          # returns intersection: {'python'}
languages | snakes          # returns union: {'cobra', 'r', 'java', 'viper',
↪'python'}
languages - snakes          # returns set difference: {'r', 'java'}
snakes - languages          # returns set difference: {'cobra', 'viper'}

# modify a set (does not return the set)
languages.add('sql')        # add a new element
languages.add('r')          # try to add an existing element (ignored, no error)
languages.remove('java')    # remove an element

try:
    languages.remove('c')       # try to remove a non-existing element (throws an␣
↪error)
except  KeyError as e:
    print("Error", e)

languages.discard('c')      # removes an element if present, but ignored otherwise
languages.pop()             # removes and returns an arbitrary element
languages.clear()           # removes all elements
languages.update('go', 'spark') # add multiple elements (can also pass a list or␣
↪set)

# get a sorted list of unique elements from a list
sorted(set([9, 0, 2, 1, 0]))    # returns [0, 1, 2, 9]
```

Out:

```
Error 'c'

[0, 1, 2, 9]
```

2.3.7 Iterators

Cartesian product

```
import itertools

print([[x, y] for x, y in itertools.product(['a', 'b', 'c'], [1, 2])])
```

Out:

```
[['a', 1], ['a', 2], ['b', 1], ['b', 2], ['c', 1], ['c', 2]]
```

2.4 Execution control statements

2.4.1 Conditional statements

```
x = 3
# if statement
if x > 0:
    print('positive')

# if/else statement
if x > 0:
    print('positive')
else:
    print('zero or negative')

# if/elif/else statement
if x > 0:
    print('positive')
elif x == 0:
    print('zero')
else:
    print('negative')

# single-line if statement (sometimes discouraged)
if x > 0: print('positive')

# single-line if/else statement (sometimes discouraged)
# known as a 'ternary operator'
sign = 'positive' if x > 0 else 'zero or negative'
```

Out:

```
positive
positive
positive
positive
```

2.4.2 Loops

Loops are a set of instructions which repeat until termination conditions are met. This can include iterating through all values in an object, go through a range of values, etc

```
# range returns a list of integers
range(0, 3)     # returns [0, 1, 2]: includes first value but excludes second␣
↪value
range(3)        # same thing: starting at zero is the default
range(0, 5, 2)  # returns [0, 2, 4]: third argument specifies the 'stride'

# for loop
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
    print(fruits[i].upper())

# alternative for loop (recommended style)
for fruit in fruits:
    print(fruit.upper())

# use range when iterating over a large sequence to avoid actually creating the␣
↪integer list in memory
v = 0
for i in range(10 ** 6):
    v += 1
```

Out:

```
APPLE
BANANA
CHERRY
APPLE
BANANA
CHERRY
```

2.5 List comprehensions, iterators, etc.

2.5.1 List comprehensions

Process which affects whole lists without iterating through loops. For more: http://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html

```
# for loop to create a list of cubes
nums = [1, 2, 3, 4, 5]
cubes = []
for num in nums:
    cubes.append(num**3)

# equivalent list comprehension
```

(continues on next page)

(continued from previous page)

```
cubes = [num**3 for num in nums]    # [1, 8, 27, 64, 125]

# for loop to create a list of cubes of even numbers
cubes_of_even = []
for num in nums:
    if num % 2 == 0:
        cubes_of_even.append(num**3)

# equivalent list comprehension
# syntax: [expression for variable in iterable if condition]
cubes_of_even = [num**3 for num in nums if num % 2 == 0]    # [8, 64]

# for loop to cube even numbers and square odd numbers
cubes_and_squares = []
for num in nums:
    if num % 2 == 0:
        cubes_and_squares.append(num**3)
    else:
        cubes_and_squares.append(num**2)

# equivalent list comprehension (using a ternary expression)
# syntax: [true_condition if condition else false_condition for variable in␣
↪iterable]
cubes_and_squares = [num**3 if num % 2 == 0 else num**2 for num in nums]    # [1,␣
↪8, 9, 64, 25]

# for loop to flatten a 2d-matrix
matrix = [[1, 2], [3, 4]]
items = []
for row in matrix:
    for item in row:
        items.append(item)

# equivalent list comprehension
items = [item for row in matrix
              for item in row] # [1, 2, 3, 4]

# set comprehension
fruits = ['apple', 'banana', 'cherry']
unique_lengths = {len(fruit) for fruit in fruits}   # {5, 6}

# dictionary comprehension
fruit_lengths = {fruit:len(fruit) for fruit in fruits} # {'apple': 5, 'banana': 6,
↪ 'cherry': 6}
```

Exercise: upper-case names and add 1 year to all simpsons

```
simpsons = {'Homer': 45, 'Marge': 45, 'Bart': 10, 'Lisa': 10}
```

(continues on next page)

(continued from previous page)

```
simpsons_older = {k.upper(): v + 1 for k, v in simpsons.items()}
print(simpsons_older)
```

Out:

```
{'HOMER': 46, 'MARGE': 46, 'BART': 11, 'LISA': 11}
```

2.5.2 Exercice: count words in a sentence

```
quote = """Tick-tow
our incomes are like our shoes; if too small they gall and pinch us
but if too large they cause us to stumble and to trip
"""

count = {word: 0 for word in set(quote.split())}
for word in quote.split():
    count[word] += 1

# iterate through two things at once (using tuple unpacking)
family = {'dad': 'homer', 'mom': 'marge', 'size': 6}
for key, value in family.items():
    print(key, value)

# use enumerate if you need to access the index value within the loop
for index, fruit in enumerate(fruits):
    print(index, fruit)

# for/else loop
for fruit in fruits:
    if fruit == 'banana':
        print("Found the banana!")
        break   # exit the loop and skip the 'else' block
    else:
        # this block executes ONLY if the for loop completes without hitting
        # 'break'
        print("Can't find the banana")

# while loop
count = 0
while count < 5:
    print("This will print 5 times")
    count += 1      # equivalent to 'count = count + 1'
```

Out:

```
dad homer
mom marge
size 6
```

(continues on next page)

(continued from previous page)

```
0 apple
1 banana
2 cherry
Can't find the banana
Found the banana!
This will print 5 times
This will print 5 times
This will print 5 times
This will print 5 times
This will print 5 times
```

2.5.3 Exceptions handling

```
dct = dict(a=[1, 2], b=[4, 5])

key = 'c'
try:
    dct[key]
except:
    print("Key %s is missing. Add it with empty value" % key)
    dct['c'] = []

print(dct)
```

Out:

```
Key c is missing. Add it with empty value
{'a': [1, 2], 'b': [4, 5], 'c': []}
```

2.6 Functions

Functions are sets of instructions launched when called upon, they can have multiple input values and a return value

```
# define a function with no arguments and no return values
def print_text():
    print('this is text')

# call the function
print_text()

# define a function with one argument and no return values
def print_this(x):
    print(x)

# call the function
```

(continues on next page)

(continued from previous page)

```
print_this(3)       # prints 3
n = print_this(3)   # prints 3, but doesn't assign 3 to n
                    # because the function has no return statement

def add(a, b):
    return a + b

add(2, 3)

add("deux", "trois")

add(["deux", "trois"], [2, 3])

# define a function with one argument and one return value
def square_this(x):
    return x ** 2

# include an optional docstring to describe the effect of a function
def square_this(x):
    """Return the square of a number."""
    return x ** 2

# call the function
square_this(3)          # prints 9
var = square_this(3)    # assigns 9 to var, but does not print 9

# default arguments
def power_this(x, power=2):
    return x ** power

power_this(2)    # 4
power_this(2, 3) # 8

# use 'pass' as a placeholder if you haven't written the function body
def stub():
    pass

# return two values from a single function
def min_max(nums):
    return min(nums), max(nums)

# return values can be assigned to a single variable as a tuple
nums = [1, 2, 3]
min_max_num = min_max(nums)         # min_max_num = (1, 3)

# return values can be assigned into multiple variables using tuple unpacking
min_num, max_num = min_max(nums)    # min_num = 1, max_num = 3
```

Out:

```
this is text
3
3
```

2.7 Regular expression

```
import re

# 1. Compile regular expression with a patetrn
regex = re.compile("^.+(sub-.+)_(ses-.+)_(mod-.+)")
```

2. Match compiled RE on string

Capture the pattern `anyprefixsub-<subj id>_ses-<session id>_<modality>`

```
strings = ["abcsub-033_ses-01_mod-mri", "defsub-044_ses-01_mod-mri", "ghisub-055_
↪ses-02_mod-ctscan"]
print([regex.findall(s)[0] for s in strings])
```

Out:

```
[('sub-033', 'ses-01', 'mod-mri'), ('sub-044', 'ses-01', 'mod-mri'), ('sub-055',
↪'ses-02', 'mod-ctscan')]
```

Match methods on compiled regular expression

Method/Attribute	Purpose
match(string)	Determine if the RE matches at the beginning of the string.
search(string)	Scan through a string, looking for any location where this RE matches.
findall(string)	Find all substrings where the RE matches, and returns them as a list.
finditer(string)	Find all substrings where the RE matches, and returns them as an iterator.

2. Replace compiled RE on string

```
regex = re.compile("(sub-[^_]+)") # match (sub-...)_
print([regex.sub("SUB-", s) for s in strings])

regex.sub("SUB-", "toto")
```

Out:

```
['abcSUB-_ses-01_mod-mri', 'defSUB-_ses-01_mod-mri', 'ghiSUB-_ses-02_mod-ctscan']

'toto'
```

Remove all non-alphanumeric characters in a string

```
re.sub('[^0-9a-zA-Z]+', '', 'h^&ell`.,|o w]{+orld')
```

Out:

```
'helloworld'
```

2.8 System programming

2.8.1 Operating system interfaces (os)

```
import os
```

Current working directory

```
# Get the current working directory
cwd = os.getcwd()
print(cwd)

# Set the current working directory
os.chdir(cwd)
```

Out:

```
/home/ed203246/git/pystatsml/python_lang
```

Temporary directory

```
import tempfile

tmpdir = tempfile.gettempdir()
```

Join paths

```
mytmpdir = os.path.join(tmpdir, "foobar")
```

Create a directory

```
os.makedirs(os.path.join(tmpdir, "foobar", "plop", "toto"), exist_ok=True)

# list containing the names of the entries in the directory given by path.
os.listdir(mytmpdir)
```

Out:

```
['plop']
```

2.8.2 File input/output

```
filename = os.path.join(mytmpdir, "myfile.txt")
print(filename)

# Write
lines = ["Dans python tout est bon", "Enfin, presque"]

## write line by line
fd = open(filename, "w")
fd.write(lines[0] + "\n")
fd.write(lines[1]+ "\n")
fd.close()

## use a context manager to automatically close your file
with open(filename, 'w') as f:
    for line in lines:
        f.write(line + '\n')

# Read
## read one line at a time (entire file does not have to fit into memory)
f = open(filename, "r")
f.readline()    # one string per line (including newlines)
f.readline()    # next line
f.close()

## read one line at a time (entire file does not have to fit into memory)
f = open(filename, 'r')
f.readline()    # one string per line (including newlines)
f.readline()    # next line
f.close()

## read the whole file at once, return a list of lines
f = open(filename, 'r')
f.readlines()   # one list, each line is one string
f.close()

## use list comprehension to duplicate readlines without reading entire file at␣
↪once
f = open(filename, 'r')
[line for line in f]
f.close()

## use a context manager to automatically close your file
with open(filename, 'r') as f:
    lines = [line for line in f]
```

Out:

```
/tmp/foobar/myfile.txt
```

2.8.3 Explore, list directories

Walk

```
import os

WD = os.path.join(tmpdir, "foobar")

for dirpath, dirnames, filenames in os.walk(WD):
    print(dirpath, dirnames, filenames)
```

Out:

```
/tmp/foobar ['plop'] ['myfile.txt']
/tmp/foobar/plop ['toto'] []
/tmp/foobar/plop/toto [] []
```

glob, basename and file extension

```
import tempfile
import glob

tmpdir = tempfile.gettempdir()

filenames = glob.glob(os.path.join(tmpdir, "*", "*.txt"))
print(filenames)

# take basename then remove extension
basenames = [os.path.splitext(os.path.basename(f))[0] for f in filenames]
print(basenames)
```

Out:

```
['/tmp/foobar/myfile.txt']
['myfile']
```

shutil - High-level file operations

```
import shutil

src = os.path.join(tmpdir, "foobar",  "myfile.txt")
dst = os.path.join(tmpdir, "foobar",  "plop", "myfile.txt")
print("copy %s to %s" % (src, dst))

shutil.copy(src, dst)

print("File %s exists ?" % dst, os.path.exists(dst))

src = os.path.join(tmpdir, "foobar",  "plop")
dst = os.path.join(tmpdir, "plop2")
print("copy tree %s under %s" % (src, dst))
```

(continues on next page)

(continued from previous page)

```
try:
    shutil.copytree(src, dst)

    shutil.rmtree(dst)

    shutil.move(src, dst)
except (FileExistsError, FileNotFoundError) as e:
    pass
```

Out:

```
copy /tmp/foobar/myfile.txt to /tmp/foobar/plop/myfile.txt
File /tmp/foobar/plop/myfile.txt exists ? True
copy tree /tmp/foobar/plop under /tmp/plop2
```

2.8.4 Command execution with subprocess

- For more advanced use cases, the underlying Popen interface can be used directly.
- Run the command described by args.
- Wait for command to complete
- return a CompletedProcess instance.
- Does not capture stdout or stderr by default. To do so, pass PIPE for the stdout and/or stderr arguments.

```
import subprocess

# doesn't capture output
p = subprocess.run(["ls", "-l"])
print(p.returncode)

# Run through the shell.
subprocess.run("ls -l", shell=True)

# Capture output
out = subprocess.run(["ls", "-a", "/"], stdout=subprocess.PIPE, stderr=subprocess.
↪STDOUT)
# out.stdout is a sequence of bytes that should be decoded into a utf-8 string
print(out.stdout.decode('utf-8').split("\n")[:5])
```

Out:

```
0
['.', '..', 'bin', 'boot', 'cdrom']
```

2.8.5 Multiprocessing and multithreading

Process

A process is a name given to a program instance that has been loaded into memory and managed by the operating system.

Process = address space + execution context (thread of control)

Process address space (segments):

- Code.
- Data (static/global).
- Heap (dynamic memory allocation).
- Stack.

Execution context:

- Data registers.
- Stack pointer (SP).
- Program counter (PC).
- Working Registers.

OS Scheduling of processes: context switching (ie. save/load Execution context)

Pros/cons

- Context switching expensive.
- (potentially) complex data sharing (not necessary true).
- Cooperating processes - no need for memory protection (separate address spaces).
- Relevant for parrallel computation with memory allocation.

Threads

- Threads share the same address space (Data registers): access to code, heap and (global) data.
- Separate execution stack, PC and Working Registers.

Pros/cons

- Faster context switching only SP, PC and Working Registers.
- Can exploit fine-grain concurrency
- Simple data sharing through the shared address space.
- Precautions have to be taken or two threads will write to the same memory at the same time. This is what the **global interpreter lock (GIL)** is for.
- Relevant for GUI, I/O (Network, disk) concurrent operation

In Python

- The `threading` module uses threads.

- The `multiprocessing` module uses processes.

Multithreading

```
import time
import threading

def list_append(count, sign=1, out_list=None):
    if out_list is None:
        out_list = list()
    for i in range(count):
        out_list.append(sign * i)
        sum(out_list) # do some computation
    return out_list

size = 10000   # Number of numbers to add

out_list = list() # result is a simple list
thread1 = threading.Thread(target=list_append, args=(size, 1, out_list, ))
thread2 = threading.Thread(target=list_append, args=(size, -1, out_list, ))

startime = time.time()
# Will execute both in parallel
thread1.start()
thread2.start()
# Joins threads back to the parent process
thread1.join()
thread2.join()
print("Threading ellapsed time ", time.time() - startime)

print(out_list[:10])
```

Out:

```
Threading ellapsed time  0.6789593696594238
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Multiprocessing

```
import multiprocessing

# Sharing requires specific mecanism
out_list1 = multiprocessing.Manager().list()
p1 = multiprocessing.Process(target=list_append, args=(size, 1, None))
out_list2 = multiprocessing.Manager().list()
p2 = multiprocessing.Process(target=list_append, args=(size, -1, None))

startime = time.time()
p1.start()
p2.start()
p1.join()
p2.join()
```

(continues on next page)

(continued from previous page)

```
print("Multiprocessing ellapsed time ", time.time() - startime)

# print(out_list[:10]) is not availlable
```

Out:

```
Multiprocessing ellapsed time  0.19075894355773926
```

Sharing object between process with Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages shared objects.

```
import multiprocessing
import time

size = int(size / 100)   # Number of numbers to add

# Sharing requires specific mecanism
out_list = multiprocessing.Manager().list()
p1 = multiprocessing.Process(target=list_append, args=(size, 1, out_list))
p2 = multiprocessing.Process(target=list_append, args=(size, -1, out_list))

startime = time.time()

p1.start()
p2.start()

p1.join()
p2.join()

print(out_list[:10])

print("Multiprocessing with shared object ellapsed time ", time.time() - startime)
```

Out:

```
[0, 1, 2, 3, 4, 5, 0, 6, -1, 7]
Multiprocessing with shared object ellapsed time  0.3832252025604248
```

2.9 Scripts and argument parsing

Example, the word count script

```
import os
import os.path
import argparse
import re
import pandas as pd

if __name__ == "__main__":
    # parse command line options
    output = "word_count.csv"
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input',
                        help='list of input files.',
                        nargs='+', type=str)
    parser.add_argument('-o', '--output',
                        help='output csv file (default %s)' % output,
                        type=str, default=output)
    options = parser.parse_args()

    if options.input is None :
        parser.print_help()
        raise SystemExit("Error: input files are missing")
    else:
        filenames = [f for f in options.input if os.path.isfile(f)]

    # Match words
    regex = re.compile("[a-zA-Z]+")

    count = dict()
    for filename in filenames:
        fd = open(filename, "r")
        for line in fd:
            for word in regex.findall(line.lower()):
                if not word in count:
                    count[word] = 1
                else:
                    count[word] += 1

    fd = open(options.output, "w")

    # Pandas
    df = pd.DataFrame([[k, count[k]] for k in count], columns=["word", "count"])
    df.to_csv(options.output, index=False)
```

2.10 Networking

```
# TODO
```

2.10.1 FTP

```
# Full FTP features with ftplib
import ftplib
ftp = ftplib.FTP("ftp.cea.fr")
ftp.login()
ftp.cwd('/pub/unati/people/educhesnay/pystatml')
ftp.retrlines('LIST')

fd = open(os.path.join(tmpdir, "README.md"), "wb")
ftp.retrbinary('RETR README.md', fd.write)
fd.close()
ftp.quit()

# File download urllib
import urllib.request
ftp_url = 'ftp://ftp.cea.fr/pub/unati/people/educhesnay/pystatml/README.md'
urllib.request.urlretrieve(ftp_url, os.path.join(tmpdir, "README2.md"))
```

Out:

```
-rw-r--r--    1 ftp      ftp          3019 Oct 16  2019 README.md
-rw-r--r--    1 ftp      ftp      10672252 Dec 18 10:13␣
↪StatisticsMachineLearningPython.pdf
-rw-r--r--    1 ftp      ftp       9676120 Nov 12  2020␣
↪StatisticsMachineLearningPythonDraft.pdf
-rw-r--r--    1 ftp      ftp       9798485 Jul 08  2020␣
↪StatisticsMachineLearningPythonDraft_202007.pdf

('/tmp/README2.md', <email.message.Message object at 0x7f88eefd5580>)
```

2.10.2 HTTP

```
# TODO
```

2.10.3 Sockets

```
# TODO
```

2.10.4 xmlrpc

```
# TODO
```

2.11 Modules and packages

A module is a Python file. A package is a directory which MUST contain a special file called `__init__.py`

To import, extend variable *PYTHONPATH*:

```
export PYTHONPATH=path_to_parent_python_module:${PYTHONPATH}
```

Or

```
import sys
sys.path.append("path_to_parent_python_module")
```

The `__init__.py` file can be empty. But you can set which modules the package exports as the API, while keeping other modules internal, by overriding the __all__ variable, like so:

`parentmodule/__init__.py` file:

```
from . import submodule1
from . import submodule2

from .submodule3 import function1
from .submodule3 import function2

__all__ = ["submodule1", "submodule2",
           "function1", "function2"]
```

User can import:

```
import parentmodule.submodule1
import parentmodule.function1
```

Python Unit Testing

TODO

2.12 Object Oriented Programming (OOP)

Sources

- http://python-textbok.readthedocs.org/en/latest/Object_Oriented_Programming.html

Principles

- **Encapsulate** data (attributes) and code (methods) into objects.
- **Class** = template or blueprint that can be used to create objects.
- An **object** is a specific instance of a class.
- **Inheritance**: OOP allows classes to inherit commonly used state and behaviour from other classes. Reduce code duplication
- **Polymorphism**: (usually obtained through polymorphism) calling code is agnostic as to whether an object belongs to a parent class or one of its descendants (abstraction, modularity). The same method called on 2 objects of 2 different classes will behave differently.

```
import math


class Shape2D:
    def area(self):
        raise NotImplementedError()

# __init__ is a special method called the constructor


# Inheritance + Encapsulation
class Square(Shape2D):
    def __init__(self, width):
        self.width = width

    def area(self):
        return self.width ** 2


class Disk(Shape2D):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2


shapes = [Square(2), Disk(3)]

# Polymorphism
print([s.area() for s in shapes])

s = Shape2D()
```

(continues on next page)

(continued from previous page)

```
try:
    s.area()
except NotImplementedError as e:
    print("NotImplementedError", e)
```

Out:

```
[4, 28.274333882308138]
NotImplementedError
```

2.13 Style guide for Python programming

See PEP 8

- Spaces (four) are the preferred indentation method.
- Two blank lines for top level function or classes definition.
- One blank line to indicate logical sections.
- Never use: `from lib import *`
- Bad: `Capitalized_Words_With_Underscores`
- Function and Variable Names: `lower_case_with_underscores`
- Class Names: `CapitalizedWords` (aka: `CamelCase`)

2.14 Documenting

See Documenting Python Documenting = comments + docstrings (Python documentation string)

- Docstrings are use as documentation for the class, module, and packages. See it as "living documentation".
- Comments are used to explain non-obvious portions of the code. "Dead documentation".

Docstrings for functions (same for classes and methods):

```
def my_function(a, b=2):
    """
    This function ...

    Parameters
    ----------
    a : float
        First operand.
    b : float, optional
        Second operand. The default is 2.
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    Sum of operands.

    Example
    -------
    >>> my_function(3)
    5
    """
    # Add a with b (this is a comment)
    return a + b

print(help(my_function))
```

Out:

```
Help on function my_function in module __main__:

my_function(a, b=2)
    This function ...

    Parameters
    ----------
    a : float
        First operand.
    b : float, optional
        Second operand. The default is 2.

    Returns
    -------
    Sum of operands.

    Example
    -------
    >>> my_function(3)
    5

None
```

Docstrings for scripts:

At the begining of a script add a pream:

```
"""
Created on Thu Nov 14 12:08:41 CET 2019

@author: firstname.lastname@email.com

Some description
"""
```

2.15 Exercises

2.15.1 Exercise 1: functions

Create a function that acts as a simple calulator If the operation is not specified, default to addition If the operation is misspecified, return an prompt message Ex: `calc(4,5,"multiply")` returns 20 Ex: `calc(3,5)` returns 8 Ex: `calc(1, 2, "something")` returns error message

2.15.2 Exercise 2: functions + list + loop

Given a list of numbers, return a list where all adjacent duplicate elements have been reduced to a single element. Ex: `[1, 2, 2, 3, 2]` returns `[1, 2, 3, 2]`. You may create a new list or modify the passed in list.

Remove all duplicate values (adjacent or not) Ex: `[1, 2, 2, 3, 2]` returns `[1, 2, 3]`

2.15.3 Exercise 3: File I/O

1. Copy/paste the BSD 4 clause license (https://en.wikipedia.org/wiki/BSD_licenses) into a text file. Read, the file and count the occurrences of each word within the file. Store the words' occurrence number in a dictionary.

2. Write an executable python command `count_words.py` that parse a list of input files provided after `--input` parameter. The dictionary of occurrence is save in a csv file provides by `--output`. with default value word_count.csv. Use: - open - regular expression - argparse (https://docs.python.org/3/howto/argparse.html)

2.15.4 Exercise 4: OOP

1. Create a class `Employee` with 2 attributes provided in the constructor: `name`, `years_of_service`. With one method `salary` with is obtained by `1500 + 100 * years_of_service`.
2. Create a subclass `Manager` which redefine `salary` method `2500 + 120 * years_of_service`.
3. Create a small dictionary-nosed database where the key is the employee's name. Populate the database with: samples = Employee('lucy', 3), Employee('john', 1), Manager('julie', 10), Manager('paul', 3)
4. Return a table of made name, salary rows, i.e. a list of list [[name, salary]]
5. Compute the average salary

Total running time of the script: (0 minutes 2.741 seconds)

CHAPTER

THREE

SCIENTIFIC PYTHON

3.1 Numpy: arrays and matrices

NumPy is an extension to the Python programming language, adding support for large, multi-dimensional (numerical) arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

Sources:

- Kevin Markham: https://github.com/justmarkham

Computation time:

import numpy as np

l = [v for v in range(10 ** 8)] s = 0 %time for v in l: s += v

arr = np.arange(10 ** 8) %time arr.sum()

3.1.1 Create arrays

Create ndarrays from lists. note: every element must be the same type (will be converted if possible)

```
import numpy as np

data1 = [1, 2, 3, 4, 5]             # list
arr1 = np.array(data1)              # 1d array
data2 = [range(1, 5), range(5, 9)]  # list of lists
arr2 = np.array(data2)              # 2d array
arr2.tolist()                       # convert array back to list
```

Out:

```
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

create special arrays

```
np.zeros(10)
np.zeros((3, 6))
np.ones(10)
```

(continues on next page)

(continued from previous page)

```
np.linspace(0, 1, 5)            # 0 to 1 (inclusive) with 5 points
np.logspace(0, 3, 4)            # 10^0 to 10^3 (inclusive) with 4 points
```

Out:

```
array([   1.,   10.,  100., 1000.])
```

arange is like range, except it returns an array (not a list)

```
int_array = np.arange(5)
float_array = int_array.astype(float)
```

3.1.2 Examining arrays

```
arr1.dtype      # float64
arr2.ndim       # 2
arr2.shape      # (2, 4) - axis 0 is rows, axis 1 is columns
arr2.size       # 8 - total number of elements
len(arr2)       # 2 - size of first dimension (aka axis)
```

Out:

```
2
```

3.1.3 Reshaping

```
arr = np.arange(10, dtype=float).reshape((2, 5))
print(arr.shape)
print(arr.reshape(5, 2))
```

Out:

```
(2, 5)
[[0. 1.]
 [2. 3.]
 [4. 5.]
 [6. 7.]
 [8. 9.]]
```

Add an axis

```
a = np.array([0, 1])
a_col = a[:, np.newaxis]
print(a_col)
#or
a_col = a[:, None]
```

Out:

```
[[0]
 [1]]
```

Transpose

```
print(a_col.T)
```

Out:

```
[[0 1]]
```

Flatten: always returns a flat copy of the orriginal array

```
arr_flt = arr.flatten()
arr_flt[0] = 33
print(arr_flt)
print(arr)
```

Out:

```
[33.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
[[0. 1. 2. 3. 4.]
 [5. 6. 7. 8. 9.]]
```

Ravel: returns a view of the original array whenever possible.

```
arr_flt = arr.ravel()
arr_flt[0] = 33
print(arr_flt)
print(arr)
```

Out:

```
[33.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
[[33.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

3.1.4 Summary on axis, reshaping/flattening and selection

Numpy internals: By default Numpy use C convention, ie, Row-major language: The matrix is stored by rows. In C, the last index changes most rapidly as one moves through the array as stored in memory.

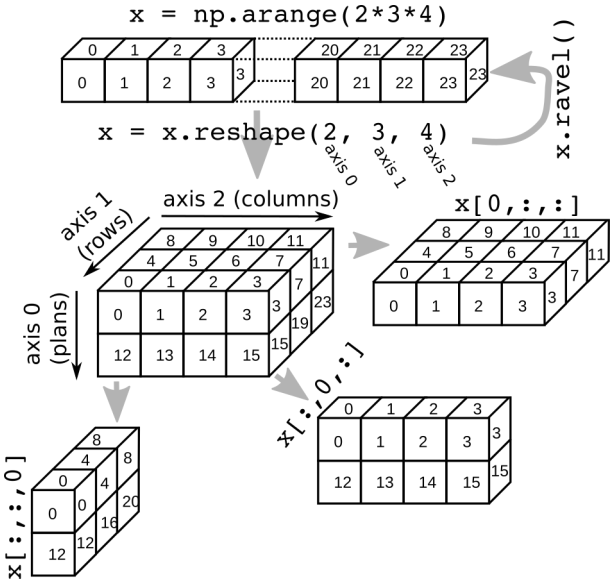
For 2D arrays, sequential move in the memory will:

- **iterate over rows (axis 0)**
  - iterate over columns (axis 1)

For 3D arrays, sequential move in the memory will:

- **iterate over plans (axis 0)**
  - **iterate over rows (axis 1)**

* iterate over columns (axis 2)




```
x = np.arange(2 * 3 * 4)
print(x)
```

Out:

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

Reshape into 3D (axis 0, axis 1, axis 2)

```
x = x.reshape(2, 3, 4)
print(x)
```

Out:

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Selection get first plan

```
print(x[0, :, :])
```

Out:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Selection get first rows

```
print(x[:, 0, :])
```

Out:

```
[[ 0  1  2  3]
 [12 13 14 15]]
```

Selection get first columns

```
print(x[:, :, 0])
```

Out:

```
[[ 0  4  8]
 [12 16 20]]
```

Simple example with 2 array

Exercise:

- Get second line
- Get third column

```
arr = np.arange(10, dtype=float).reshape((2, 5))
print(arr)

arr[1, :]
arr[:, 2]
```

Out:

```
[[0. 1. 2. 3. 4.]
 [5. 6. 7. 8. 9.]]

array([2., 7.])
```

Ravel

```
print(x.ravel())
```

Out:

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

3.1.5 Stack arrays

```
a = np.array([0, 1])
b = np.array([2, 3])
```

Horizontal stacking

```
np.hstack([a, b])
```

Out:

```
array([0, 1, 2, 3])
```

Vertical stacking

```
np.vstack([a, b])
```

Out:

```
array([[0, 1],
       [2, 3]])
```

Default Vertical

```
np.stack([a, b])
```

Out:

```
array([[0, 1],
       [2, 3]])
```

3.1.6 Selection

Single item

```
arr = np.arange(10, dtype=float).reshape((2, 5))

arr[0]          # 0th element (slices like a list)
arr[0, 3]       # row 0, column 3: returns 4
arr[0][3]       # alternative syntax
```

Out:

```
3.0
```

Slicing

Syntax: `start:stop:step` with `start` *(default 0)* `stop` *(default last)* `step` *(default 1)*

```
arr[0, :]      # row 0: returns 1d array ([1, 2, 3, 4])
arr[:, 0]      # column 0: returns 1d array ([1, 5])
arr[:, :2]     # columns strictly before index 2 (2 first columns)
arr[:, 2:]     # columns after index 2 included
arr2 = arr[:, 1:4]  # columns between index 1 (included) and 4 (excluded)
print(arr2)
```

Out:

```
[[1. 2. 3.]
 [6. 7. 8.]]
```

Slicing returns a view (not a copy) Modification

```
arr2[0, 0] = 33
print(arr2)
print(arr)
```

Out:

```
[[33.  2.  3.]
 [ 6.  7.  8.]]
[[ 0. 33.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

Row 0: reverse order

```
print(arr[0, ::-1])

# The rule of thumb here can be: in the context of lvalue indexing (i.e. the␣
↪indices are placed in the left hand side value of an assignment), no view or␣
↪copy of the array is created (because there is no need to). However, with␣
↪regular values, the above rules for creating views does apply.
```

Out:

```
[ 4.  3.  2. 33.  0.]
```

Fancy indexing: Integer or boolean array indexing

Fancy indexing returns a copy not a view.

Integer array indexing

```
arr2 = arr[:, [1, 2, 3]]  # return a copy
print(arr2)
arr2[0, 0] = 44
```

(continues on next page)

(continued from previous page)

```
print(arr2)
print(arr)
```

Out:

```
[[33.  2.  3.]
 [ 6.  7.  8.]]
[[44.  2.  3.]
 [ 6.  7.  8.]]
[[ 0. 33.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

Boolean arrays indexing

```
arr2 = arr[arr > 5]  # return a copy

print(arr2)
arr2[0] = 44
print(arr2)
print(arr)
```

Out:

```
[33.  6.  7.  8.  9.]
[44.  6.  7.  8.  9.]
[[ 0. 33.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

However, In the context of lvalue indexing (left hand side value of an assignment) Fancy authorizes the modification of the original array

```
arr[arr > 5] = 0
print(arr)
```

Out:

```
[[0. 0. 2. 3. 4.]
 [5. 0. 0. 0. 0.]]
```

Boolean arrays indexing continues

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob'])
names == 'Bob'                          # returns a boolean array
names[names != 'Bob']                   # logical selection
(names == 'Bob') | (names == 'Will')    # keywords "and/or" don't work with␣
↪boolean arrays
names[names != 'Bob'] = 'Joe'           # assign based on a logical selection
np.unique(names)                        # set function
```

Out:

```
array(['Bob', 'Joe'], dtype='<U4')
```

3.1.7 Vectorized operations

```
nums = np.arange(5)
nums * 10                          # multiply each element by 10
nums = np.sqrt(nums)               # square root of each element
np.ceil(nums)                      # also floor, rint (round to nearest int)
np.isnan(nums)                     # checks for NaN
nums + np.arange(5)                # add element-wise
np.maximum(nums, np.array([1, -2, 3, -4, 5]))  # compare element-wise

# Compute Euclidean distance between 2 vectors
vec1 = np.random.randn(10)
vec2 = np.random.randn(10)
dist = np.sqrt(np.sum((vec1 - vec2) ** 2))

# math and stats
rnd = np.random.randn(4, 2) # random normals in 4x2 array
rnd.mean()
rnd.std()
rnd.argmin()                # index of minimum element
rnd.sum()
rnd.sum(axis=0)             # sum of columns
rnd.sum(axis=1)             # sum of rows

# methods for boolean arrays
(rnd > 0).sum()             # counts number of positive values
(rnd > 0).any()             # checks if any value is True
(rnd > 0).all()             # checks if all values are True

# random numbers
np.random.seed(12234)       # Set the seed
np.random.rand(2, 3)        # 2 x 3 matrix in [0, 1]
np.random.randn(10)         # random normals (mean 0, sd 1)
np.random.randint(0, 2, 10) # 10 randomly picked 0 or 1
```

Out:

```
array([0, 0, 0, 1, 1, 0, 1, 1, 1, 1])
```

3.1.8 Broadcasting

Sources: https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html Implicit conversion to allow operations on arrays of different sizes. - The smaller array is stretched or "broadcasted" across the larger array so that they have compatible shapes. - Fast vectorized operation in C instead of Python. - No needless copies.

Rules

Starting with the trailing axis and working backward, Numpy compares arrays dimensions.

- If two dimensions are equal then continues
- If one of the operand has dimension 1 stretches it to match the largest one
- When one of the shapes runs out of dimensions (because it has less dimensions than the other shape), Numpy will use 1 in the comparison process until the other shape's dimensions run out as well.

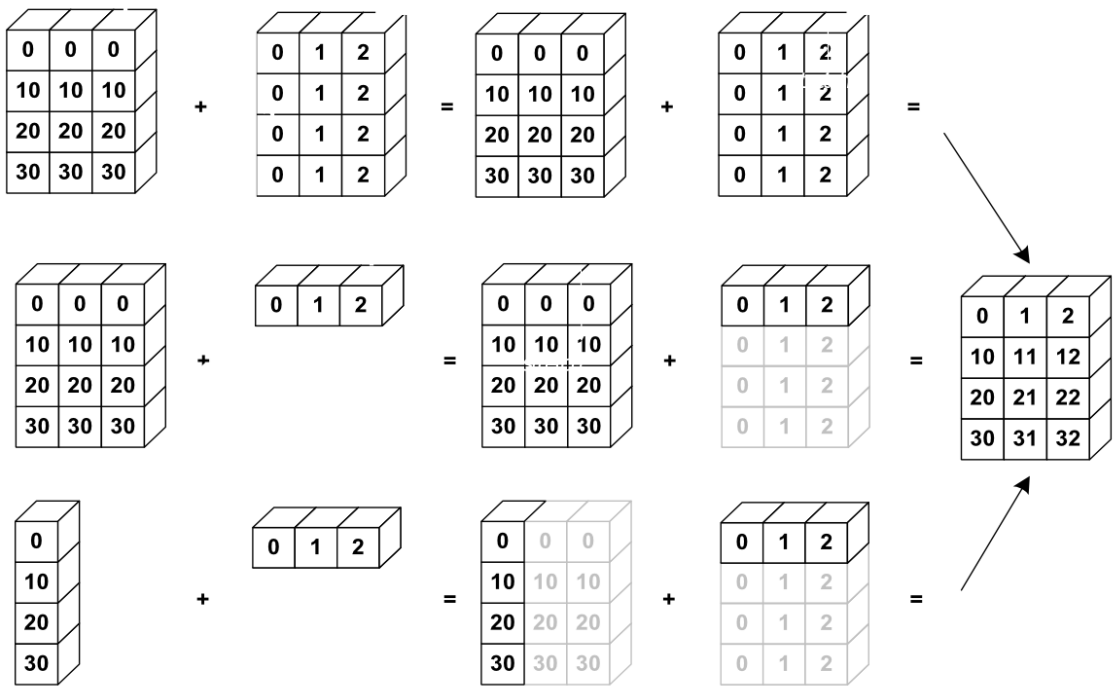


Fig. 1: Source: http://www.scipy-lectures.org

```
a = np.array([[ 0,  0,  0],
              [10, 10, 10],
              [20, 20, 20],
              [30, 30, 30]])
```

(continues on next page)

(continued from previous page)

```
b = np.array([0, 1, 2])

print(a + b)
```

Out:

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Center data column-wise

```
a - a.mean(axis=0)
```

Out:

```
array([[-15., -15., -15.],
       [ -5.,  -5.,  -5.],
       [  5.,   5.,   5.],
       [ 15.,  15.,  15.]])
```

Scale (center, normalise) data column-wise

```
(a - a.mean(axis=0)) / a.std(axis=0)
```

Out:

```
array([[-1.34164079, -1.34164079, -1.34164079],
       [-0.4472136 , -0.4472136 , -0.4472136 ],
       [ 0.4472136 ,  0.4472136 ,  0.4472136 ],
       [ 1.34164079,  1.34164079,  1.34164079]])
```

Examples

Shapes of operands A, B and result:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
```

(continues on next page)

(continued from previous page)

```
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

3.1.9 Exercises

Given the array:

```
X = np.random.randn(4, 2) # random normals in 4x2 array
```

- For each column find the row index of the minimum value.
- Write a function `standardize(X)` that return an array whose columns are centered and scaled (by std-dev).

Total running time of the script: (0 minutes 0.012 seconds)

3.2 Pandas: data manipulation

It is often said that 80% of data analysis is spent on the cleaning and small, but important, aspect of data manipulation and cleaning with Pandas.

Sources:

- Kevin Markham: https://github.com/justmarkham
- Pandas doc: http://pandas.pydata.org/pandas-docs/stable/index.html

Data structures

- **Series** is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call *pd.Series([1,3,5,np.nan,6,8])*
- **DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It stems from the *R data.frame()* object.

```
import pandas as pd
import numpy as np
```

3.2.1 Create DataFrame

```
columns = ['name', 'age', 'gender', 'job']

user1 = pd.DataFrame([['alice', 19, "F", "student"],
                      ['john', 26, "M", "student"]],
                     columns=columns)

user2 = pd.DataFrame([['eric', 22, "M", "student"],
                      ['paul', 58, "F", "manager"]],
                     columns=columns)

user3 = pd.DataFrame(dict(name=['peter', 'julie'],
                          age=[33, 44], gender=['M', 'F'],
                          job=['engineer', 'scientist']))

print(user3)
```

Out:

```
    name  age gender        job
0  peter   33      M   engineer
1  julie   44      F  scientist
```

3.2.2 Combining DataFrames

Concatenate DataFrame

Concatenate columns (axis = 1).

```
height = pd.DataFrame(dict(height=[1.65, 1.8]))
print(user1, "\n", height)

print(pd.concat([user1, height], axis=1))
```

Out:

```
    name  age gender      job
0  alice   19      F  student
1   john   26      M  student
    height
0    1.65
1    1.80
    name  age gender      job  height
0  alice   19      F  student    1.65
1   john   26      M  student    1.80
```

Concatenate rows (default: axis = 0)

```
users = pd.concat([user1, user2, user3])
print(users)
```

Out:

```
    name  age gender        job
0  alice   19      F    student
1   john   26      M    student
0   eric   22      M    student
1   paul   58      F    manager
0  peter   33      M   engineer
1  julie   44      F  scientist
```

Concatenate rows: append

```
user1.append(user2)
```

Join DataFrame

```
user4 = pd.DataFrame(dict(name=['alice', 'john', 'eric', 'julie'],
                          height=[165, 180, 175, 171]))
print(user4)
```

Out:

```
    name  height
0  alice     165
1   john     180
2   eric     175
3  julie     171
```

Use intersection of keys from both frames

```
merge_inter = pd.merge(users, user4)

print(merge_inter)
```

Out:

```
    name  age gender        job  height
0  alice   19      F    student     165
1   john   26      M    student     180
2   eric   22      M    student     175
3  julie   44      F  scientist     171
```

Use union of keys from both frames

```
users = pd.merge(users, user4, on="name", how='outer')
print(users)
```

Out:

```
    name  age gender        job  height
0  alice   19      F    student   165.0
1   john   26      M    student   180.0
2   eric   22      M    student   175.0
3   paul   58      F    manager     NaN
4  peter   33      M   engineer     NaN
5  julie   44      F  scientist   171.0
```

Reshaping by pivoting

"Unpivots" a DataFrame from wide format to long (stacked) format,

```
staked = pd.melt(users, id_vars="name", var_name="variable", value_name="value")
print(staked)
```

Out:

```
     name variable      value
0   alice      age         19
1    john      age         26
2    eric      age         22
3    paul      age         58
4   peter      age         33
5   julie      age         44
6   alice   gender          F
7    john   gender          M
8    eric   gender          M
9    paul   gender          F
10  peter   gender          M
11  julie   gender          F
12  alice      job    student
13   john      job    student
14   eric      job    student
15   paul      job    manager
16  peter      job   engineer
17  julie      job  scientist
18  alice   height        165
19   john   height        180
20   eric   height        175
21   paul   height        NaN
22  peter   height        NaN
23  julie   height        171
```

"pivots" a DataFrame from long (stacked) format to wide format,

```
print(staked.pivot(index='name', columns='variable', values='value'))
```

Out:

```
variable age gender height        job
name
alice     19      F    165    student
eric      22      M    175    student
john      26      M    180    student
julie     44      F    171  scientist
paul      58      F    NaN    manager
peter     33      M    NaN   engineer
```

3.2.3 Summarizing

```
users                   # print the first 30 and last 30 rows
type(users)             # DataFrame
users.head()            # print the first 5 rows
users.tail()            # print the last 5 rows
```

Descriptive statistics

```
users.describe(include="all")
```

Meta-information

```
users.index             # "Row names"
users.columns           # column names
users.dtypes            # data types of each column
users.values            # underlying numpy array
users.shape             # number of rows and columns
```

Out:

```
(6, 5)
```

3.2.4 Columns selection

```
users['gender']         # select one column
type(users['gender'])   # Series
users.gender            # select one column using the DataFrame

# select multiple columns
users[['age', 'gender']]        # select two columns
my_cols = ['age', 'gender']     # or, create a list...
users[my_cols]                  # ...and use that list to select columns
type(users[my_cols])            # DataFrame
```

3.2.5 Rows selection (basic)

iloc is strictly integer position based

```
df = users.copy()
df.iloc[0]     # first row
df.iloc[0, :]  # first row
df.iloc[0, 0]  # first item of first row
df.iloc[0, 0] = 55
```

loc supports mixed integer and label based access.

```
df.loc[0]         # first row
df.loc[0, :]      # first row
df.loc[0, "age"]  # age item of first row
df.loc[0, "age"] = 55
```

Selection and index

Select females into a new DataFrame

```
df = users[users.gender == "F"]
print(df)
```

Out:

```
    name  age gender        job  height
0  alice   19      F    student   165.0
3   paul   58      F    manager     NaN
5  julie   44      F  scientist   171.0
```

Get the two first rows using *iloc* (strictly integer position)

```
df.iloc[[0, 1], :]  # Ok, but watch the index: 0, 3
```

Use *loc*

```
try:
    df.loc[[0, 1], :]  # Failed
except KeyError as err:
    print(err)
```

Out:

```
"Passing list-likes to .loc or [] with any missing labels is no longer supported.␣
↪The following labels were missing: Int64Index([1], dtype='int64'). See https://
↪pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#deprecate-loc-
↪reindex-listlike"
```

Reset index

```
df = df.reset_index(drop=True)  # Watch the index
print(df)
print(df.loc[[0, 1], :])
```

Out:

```
    name  age gender        job  height
0  alice   19      F    student   165.0
1   paul   58      F    manager     NaN
2  julie   44      F  scientist   171.0
    name  age gender      job  height
0  alice   19      F  student   165.0
1   paul   58      F  manager     NaN
```

3.2.6 Sorting

3.2.7 Rows iteration

```
df = users[:2].copy()
```

iterrows(): slow, get series, **read-only**

- Returns (index, Series) pairs.
- Slow because iterrows boxes the data into a Series.
- Retrieve fields with column name
- **Don't modify something you are iterating over**. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

```
for idx, row in df.iterrows():
    print(row["name"], row["age"])
```

Out:

```
alice 19
john 26
```

itertuples(): fast, get namedtuples, **read-only**

- Returns namedtuples of the values and which is generally faster than iterrows.
- Fast, because itertuples does not box the data into a Series.
- Retrieve fields with integer index starting from 0.
- Names will be renamed to positional names if they are invalid Python

identifier

```
for tup in df.itertuples():
    print(tup[1], tup[2])
```

Out:

```
alice 19
john 26
```

iter using *loc[i, . . .]*: read and **write**

```
for i in range(df.shape[0]):
    df.loc[i, "age"] *= 10  # df is modified
```

3.2.8 Rows selection (filtering)

simple logical filtering on numerical values

```
users[users.age < 20]        # only show users with age < 20
young_bool = users.age < 20  # or, create a Series of booleans...
young = users[young_bool]            # ...and use that Series to filter rows
users[users.age < 20].job    # select one column from the filtered results
print(young)
```

Out:

```
    name  age gender      job  height
0  alice   19      F  student   165.0
```

simple logical filtering on categorial values

```
users[users.job == 'student']
users[users.job.isin(['student', 'engineer'])]
users[users['job'].str.contains("stu|scient")]
```

Advanced logical filtering

```
users[users.age < 20][['age', 'job']]            # select multiple columns
users[(users.age > 20) & (users.gender == 'M')]  # use multiple conditions
```

3.2.9 Sorting

```
df = users.copy()

df.age.sort_values()                      # only works for a Series
df.sort_values(by='age')                  # sort rows by a specific column
df.sort_values(by='age', ascending=False) # use descending order instead
df.sort_values(by=['job', 'age'])         # sort by multiple columns
df.sort_values(by=['job', 'age'], inplace=True) # modify df

print(df)
```

Out:

```
    name  age gender        job  height
4  peter   33      M   engineer     NaN
3   paul   58      F    manager     NaN
5  julie   44      F  scientist   171.0
0  alice   19      F    student   165.0
2   eric   22      M    student   175.0
1   john   26      M    student   180.0
```

3.2.10 Descriptive statistics

Summarize all numeric columns

```
print(df.describe())
```

Out:

```
             age      height
count   6.000000    4.000000
mean   33.666667  172.750000
std    14.895189    6.344289
min    19.000000  165.000000
25%    23.000000  169.500000
50%    29.500000  173.000000
75%    41.250000  176.250000
max    58.000000  180.000000
```

Summarize all columns

```
print(df.describe(include='all'))
print(df.describe(include=['object']))  # limit to one (or more) types
```

Out:

```
         name        age gender      job      height
count       6   6.000000      6        6    4.000000
unique      6        NaN      2        4         NaN
top     peter        NaN      F  student         NaN
freq        1        NaN      3        3         NaN
mean      NaN  33.666667    NaN      NaN  172.750000
std       NaN  14.895189    NaN      NaN    6.344289
min       NaN  19.000000    NaN      NaN  165.000000
25%       NaN  23.000000    NaN      NaN  169.500000
50%       NaN  29.500000    NaN      NaN  173.000000
75%       NaN  41.250000    NaN      NaN  176.250000
max       NaN  58.000000    NaN      NaN  180.000000
         name gender      job
count       6      6        6
unique      6      2        4
top     peter      F  student
freq        1      3        3
```

Statistics per group (groupby)

```
print(df.groupby("job").mean())

print(df.groupby("job")["age"].mean())

print(df.groupby("job").describe(include='all'))
```

Out:

```
                 age      height
job
engineer   33.000000         NaN
manager    58.000000         NaN
scientist  44.000000  171.000000
student    22.333333  173.333333
job
engineer     33.000000
manager      58.000000
scientist    44.000000
student      22.333333
Name: age, dtype: float64
           name                                ...    height
          count unique    top freq mean  std  ...       std    min    25%    50% ␣
↪  75%    max
job                                           ...
engineer      1      1  peter    1  NaN  NaN  ...       NaN    NaN    NaN    NaN ␣
↪  NaN    NaN
manager       1      1   paul    1  NaN  NaN  ...       NaN    NaN    NaN    NaN ␣
↪  NaN    NaN
scientist     1      1  julie    1  NaN  NaN  ...       NaN  171.0  171.0  171.0 ␣
↪171.0  171.0
student       3      3   eric    1  NaN  NaN  ...  7.637626  165.0  170.0  175.0 ␣
↪177.5  180.0

[4 rows x 44 columns]
```

Groupby in a loop

```
for grp, data in df.groupby("job"):
    print(grp, data)
```

Out:

```
engineer     name  age gender       job  height
4  peter   33      M  engineer     NaN
manager    name  age gender      job  height
3  paul   58      F  manager     NaN
scientist     name  age gender        job  height
5  julie   44      F  scientist   171.0
student     name  age gender      job  height
0  alice   19      F  student   165.0
```

(continues on next page)

(continued from previous page)

```
2   eric   22      M  student   175.0
1   john   26      M  student   180.0
```

3.2.11 Quality check

Remove duplicate data

```
df = users.append(users.iloc[0], ignore_index=True)

print(df.duplicated())                  # Series of booleans
# (True if a row is identical to a previous row)
df.duplicated().sum()                   # count of duplicates
df[df.duplicated()]                     # only show duplicates
df.age.duplicated()                     # check a single column for duplicates
df.duplicated(['age', 'gender']).sum() # specify columns for finding duplicates
df = df.drop_duplicates()               # drop duplicate rows
```

Out:

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Missing data

```
# Missing values are often just excluded
df = users.copy()

df.describe(include='all')

# find missing values in a Series
df.height.isnull()           # True if NaN, False otherwise
df.height.notnull()          # False if NaN, True otherwise
df[df.height.notnull()]      # only show rows where age is not NaN
df.height.isnull().sum()     # count the missing values

# find missing values in a DataFrame
df.isnull()             # DataFrame of booleans
df.isnull().sum()       # calculate the sum of each column
```

Out:

```
name      0
age       0
gender    0
job       0
height    2
dtype: int64
```

Strategy 1: drop missing values

```
df.dropna()             # drop a row if ANY values are missing
df.dropna(how='all')    # drop a row only if ALL values are missing
```

Strategy 2: fill in missing values

```
df.height.mean()
df = users.copy()
df.loc[df.height.isnull(), "height"] = df["height"].mean()

print(df)
```

Out:

```
    name  age gender        job  height
0  alice   19      F    student  165.00
1   john   26      M    student  180.00
2   eric   22      M    student  175.00
3   paul   58      F    manager  172.75
4  peter   33      M   engineer  172.75
5  julie   44      F  scientist  171.00
```

3.2.12 Operation: multiplication

Multiplication of dataframe and other, element-wise

```
df = users.dropna()
df.insert(0, 'random', np.arange(df.shape[0]))
print(df)
df[["age", "height"]].multiply(df["random"], axis="index")
```

Out:

```
   random   name  age gender        job  height
0       0  alice   19      F    student   165.0
1       1   john   26      M    student   180.0
2       2   eric   22      M    student   175.0
5       3  julie   44      F  scientist   171.0
```

3.2.13 Renaming

Rename columns

```
df = users.copy()
df.rename(columns={'name': 'NAME'})
```

Rename values

```
df.job = df.job.map({'student': 'etudiant', 'manager': 'manager',
                     'engineer': 'ingenieur', 'scientist': 'scientific'})
```

3.2.14 Dealing with outliers

```
size = pd.Series(np.random.normal(loc=175, size=20, scale=10))
# Corrupt the first 3 measures
size[:3] += 500
```

Based on parametric statistics: use the mean

Assume random variable follows the normal distribution Exclude data outside 3 standard-deviations: - Probability that a sample lies within 1 sd: 68.27% - Probability that a sample lies within 3 sd: 99.73% (68.27 + 2 * 15.73)

```
size_outlr_mean = size.copy()
size_outlr_mean[((size - size.mean()).abs() > 3 * size.std())] = size.mean()
print(size_outlr_mean.mean())
```

Out:

```
248.48963819938044
```

Based on non-parametric statistics: use the median

Median absolute deviation (MAD), based on the median, is a robust non-parametric statistics. https://en.wikipedia.org/wiki/Median_absolute_deviation

```
mad = 1.4826 * np.median(np.abs(size - size.median()))
size_outlr_mad = size.copy()

size_outlr_mad[((size - size.median()).abs() > 3 * mad)] = size.median()
print(size_outlr_mad.mean(), size_outlr_mad.median())
```

Out:

```
173.80000467192673 178.7023568870694
```

3.2.15 File I/O

csv

```
import tempfile, os.path

tmpdir = tempfile.gettempdir()
csv_filename = os.path.join(tmpdir, "users.csv")
users.to_csv(csv_filename, index=False)
other = pd.read_csv(csv_filename)
```

Read csv from url

```
url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_table.csv
↪'
salary = pd.read_csv(url)
```

Excel

```
xls_filename = os.path.join(tmpdir, "users.xlsx")
users.to_excel(xls_filename, sheet_name='users', index=False)

pd.read_excel(xls_filename, sheet_name='users')

# Multiple sheets
with pd.ExcelWriter(xls_filename) as writer:
    users.to_excel(writer, sheet_name='users', index=False)
    df.to_excel(writer, sheet_name='salary', index=False)

pd.read_excel(xls_filename, sheet_name='users')
pd.read_excel(xls_filename, sheet_name='salary')
```

SQL (SQLite)

```
import pandas as pd
import sqlite3

db_filename = os.path.join(tmpdir, "users.db")
```

Connect

```
conn = sqlite3.connect(db_filename)
```

Creating tables with pandas

```
url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_table.csv
↪'
salary = pd.read_csv(url)

salary.to_sql("salary", conn, if_exists="replace")
```

Push modifications

```
cur = conn.cursor()
values = (100, 14000, 5,  'Bachelor', 'N')
cur.execute("insert into salary values (?, ?, ?, ?, ?)", values)
conn.commit()
```

Reading results into a pandas DataFrame

```
salary_sql = pd.read_sql_query("select * from salary;", conn)
print(salary_sql.head())

pd.read_sql_query("select * from salary;", conn).tail()
pd.read_sql_query('select * from salary where salary>25000;', conn)
pd.read_sql_query('select * from salary where experience=16;', conn)
pd.read_sql_query('select * from salary where education="Master";', conn)
```

Out:

```
   index  salary  experience education management
0      0   13876           1  Bachelor          Y
1      1   11608           1      Ph.D          N
2      2   18701           1      Ph.D          Y
3      3   11283           1    Master          N
4      4   11767           1      Ph.D          N
```

3.2.16 Exercises

Data Frame

1. Read the iris dataset at 'https://github.com/neurospin/pystatsml/tree/master/datasets/iris.csv'
2. Print column names
3. Get numerical columns
4. For each species compute the mean of numerical columns and store it in a `stats` table like:

```
      species  sepal_length  sepal_width  petal_length  petal_width
0      setosa         5.006        3.428         1.462        0.246
1  versicolor         5.936        2.770         4.260        1.326
2   virginica         6.588        2.974         5.552        2.026
```

Missing data

Add some missing data to the previous table `users`:

```
df = users.copy()
df.loc[[0, 2], "age"] = None
df.loc[[1, 3], "gender"] = None
```

1. Write a function `fillmissing_with_mean(df)` that fill all missing value of numerical column with the mean of the current columns.

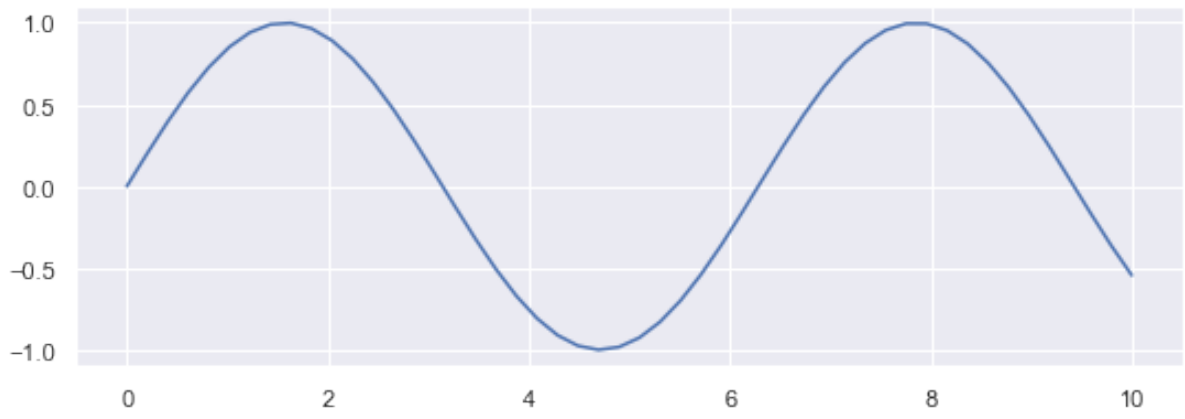
2. Save the original users and "imputed" frame in a single excel file "users.xlsx" with 2 sheets: original, imputed.

Total running time of the script: (0 minutes 1.125 seconds)

3.3 Data visualization: matplotlib & seaborn

3.3.1 Basic plots

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# inline plot (for jupyter)
%matplotlib inline

plt.figure(figsize=(9, 3))
x = np.linspace(0, 10, 50)
sinus = np.sin(x)

plt.plot(x, sinus)
plt.show()
```

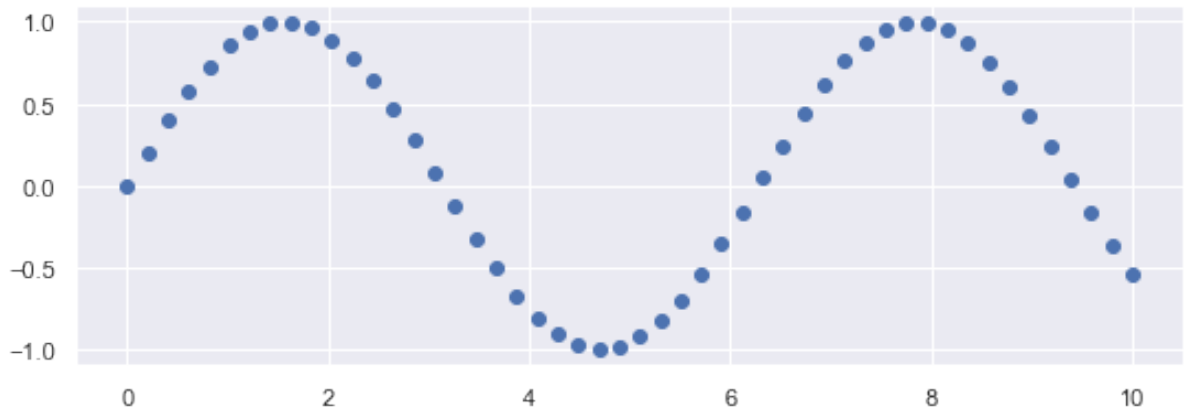



```
plt.figure(figsize=(9, 3))

plt.plot(x, sinus, "o")
```

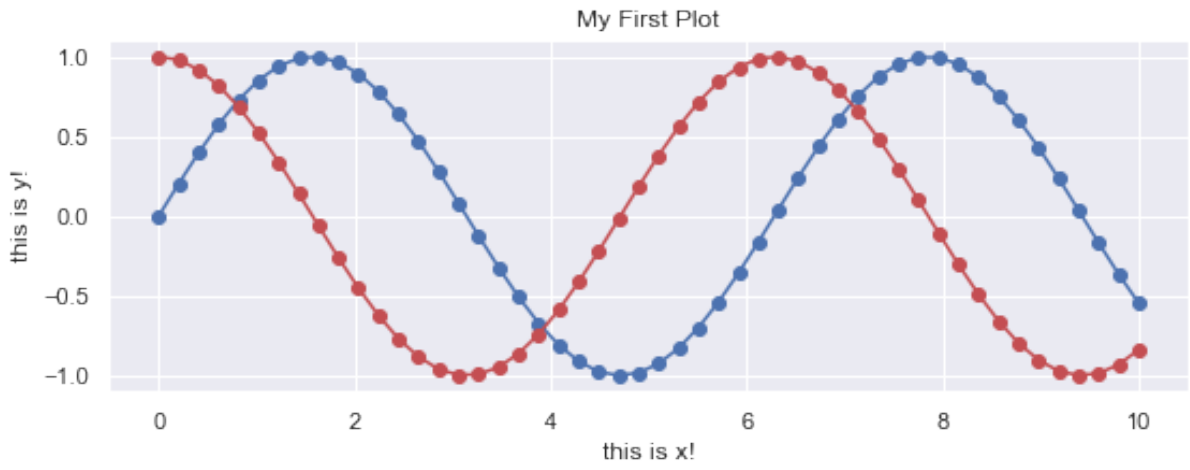
(continues on next page)

(continued from previous page)

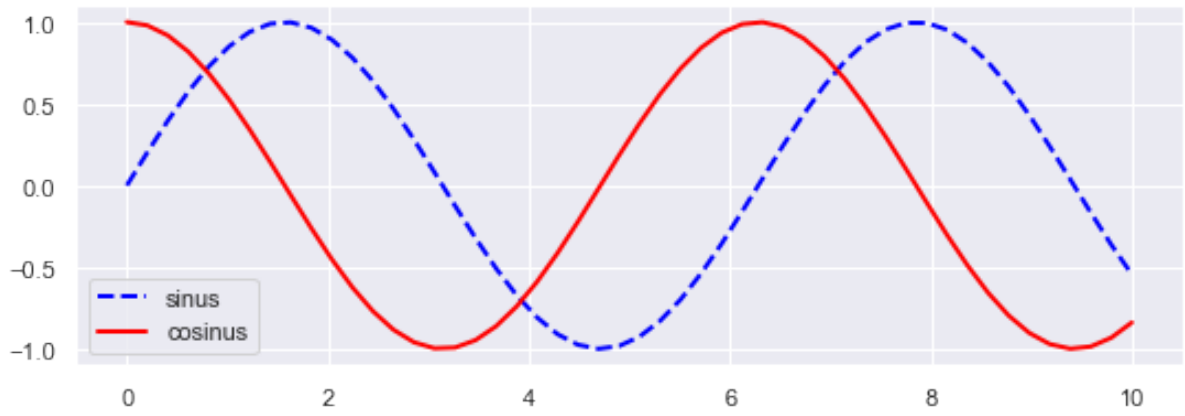
```
plt.show()
# use plt.plot to get color / marker abbreviations
```




```
# Rapid multiplot

plt.figure(figsize=(9, 3))
cosinus = np.cos(x)
plt.plot(x, sinus, "-b", x, sinus, "ob", x, cosinus, "-r", x, cosinus, "or")
plt.xlabel('this is x!')
plt.ylabel('this is y!')
plt.title('My First Plot')
plt.show()
```




```
# Step by step

plt.figure(figsize=(9, 3))
plt.plot(x, sinus, label='sinus', color='blue', linestyle='--', linewidth=2)
plt.plot(x, cosinus, label='cosinus', color='red', linestyle='-', linewidth=2)
plt.legend()
plt.show()
```



3.3.2 Scatter (2D) plots

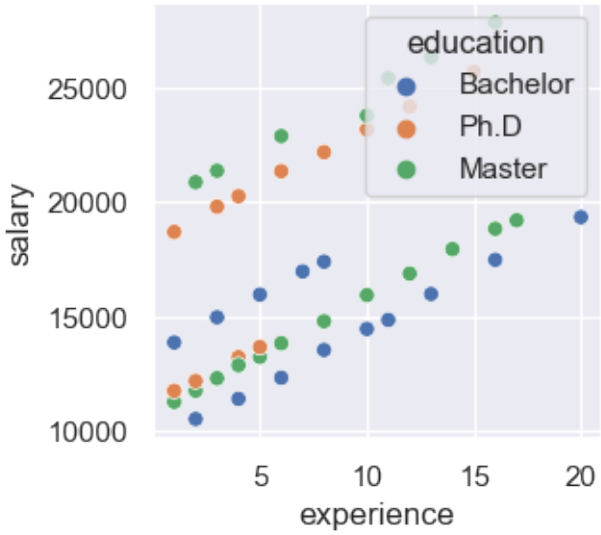
Load dataset

```
import pandas as pd
try:
    salary = pd.read_csv("../datasets/salary_table.csv")
except:
    url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_
↪table.csv'
    salary = pd.read_csv(url)

df = salary
print(df.head())
```

```
   salary  experience education management
0   13876           1  Bachelor          Y
1   11608           1      Ph.D          N
2   18701           1      Ph.D          Y
3   11283           1    Master          N
4   11767           1      Ph.D          N
```

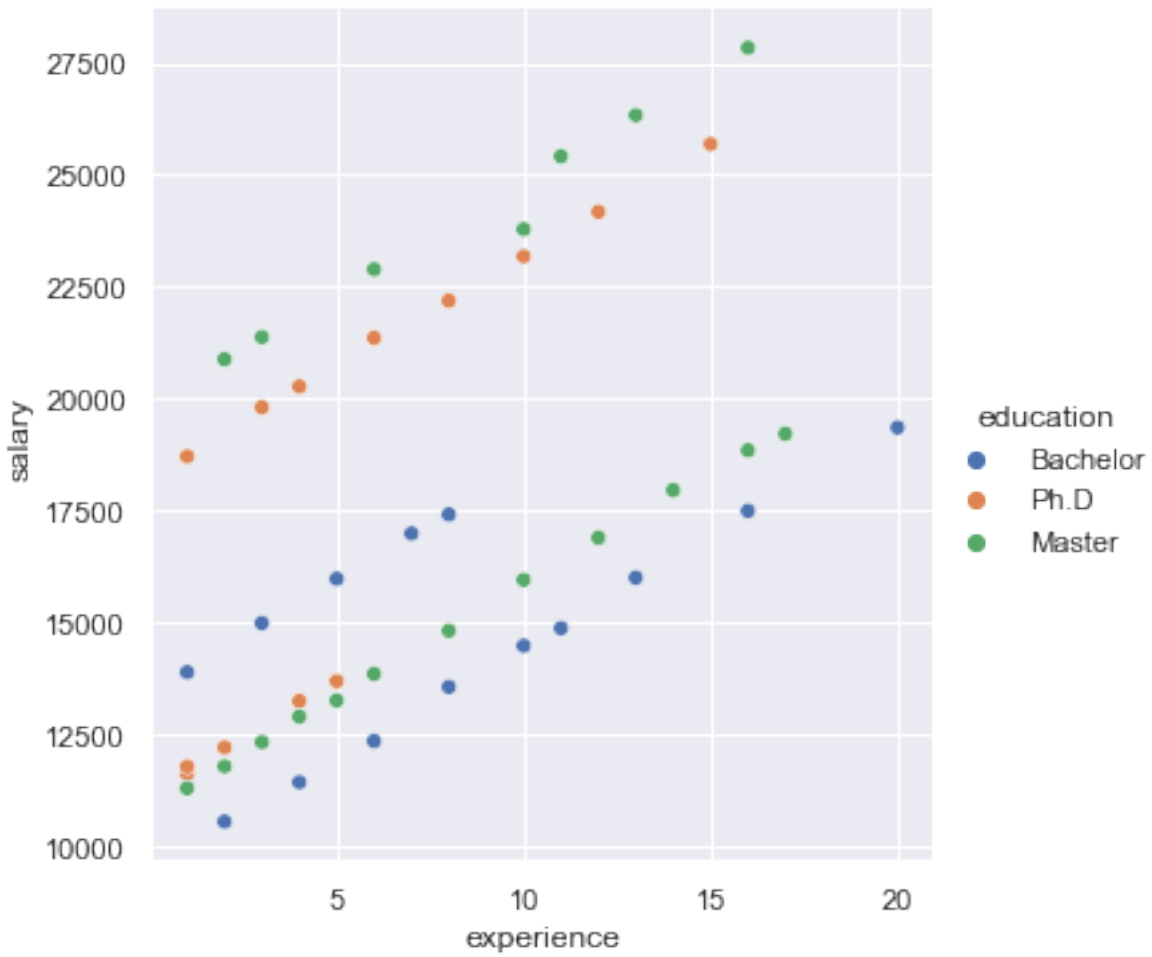
Simple scatter with colors

```
plt.figure(figsize=(3, 3), dpi=100)
_ = sns.scatterplot(x="experience", y="salary", hue="education", data=salary)
```



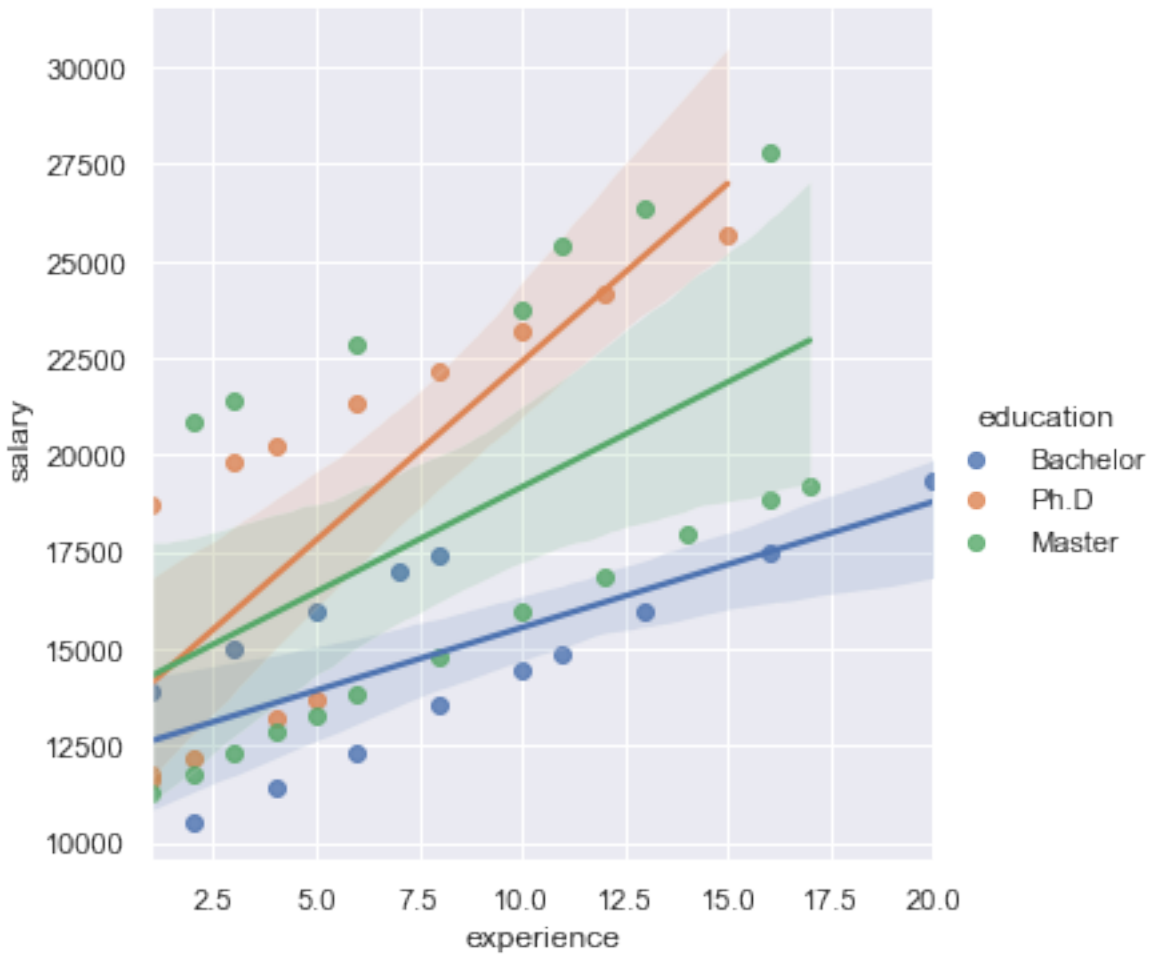
Legend outside

```
ax = sns.relplot(x="experience", y="salary", hue="education", data=salary)
```



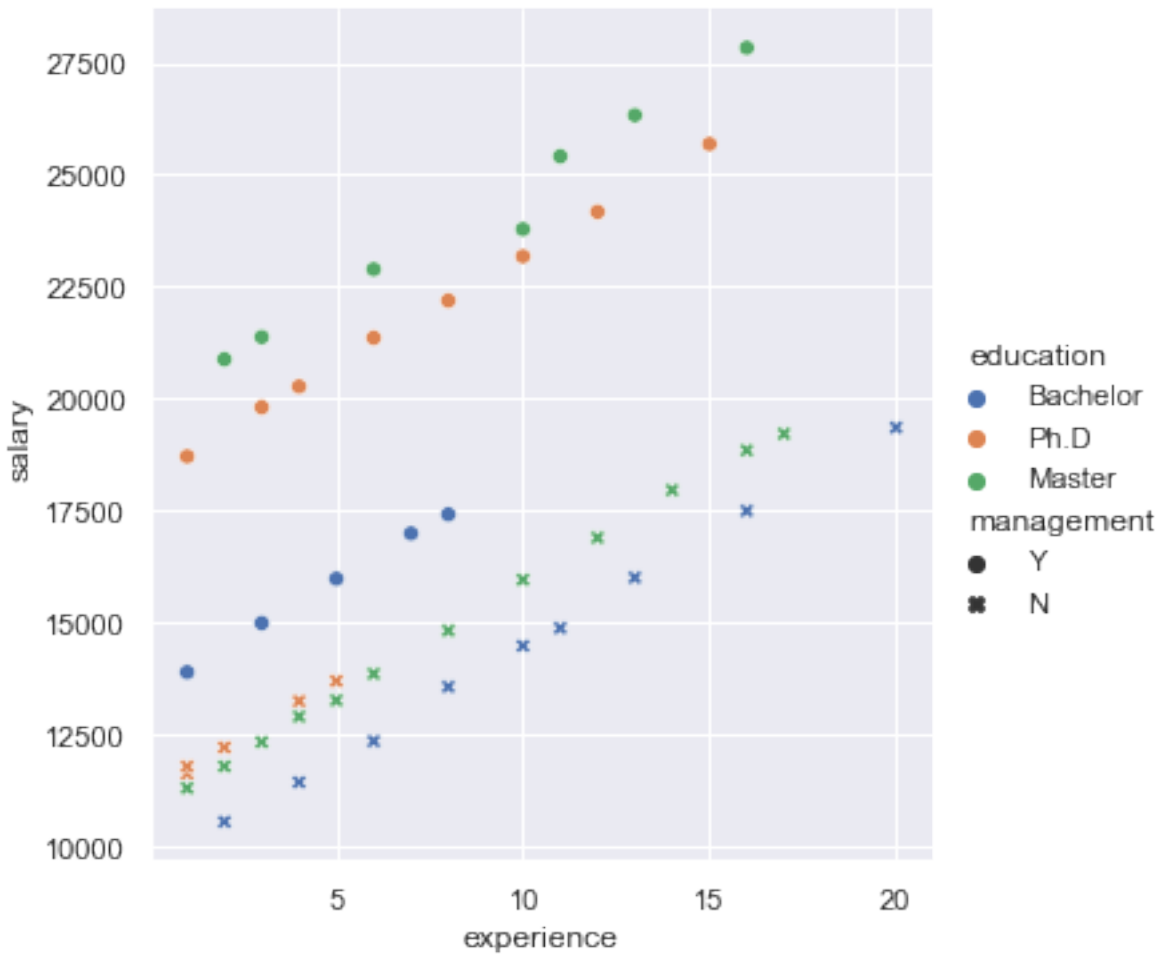
Linear model

```
ax = sns.lmplot(x="experience", y="salary", hue="education", data=salary)
```



Scatter plot with colors and symbols

```
ax = sns.relplot(x="experience", y="salary", hue="education", style='management',␣
↪data=salary)
```



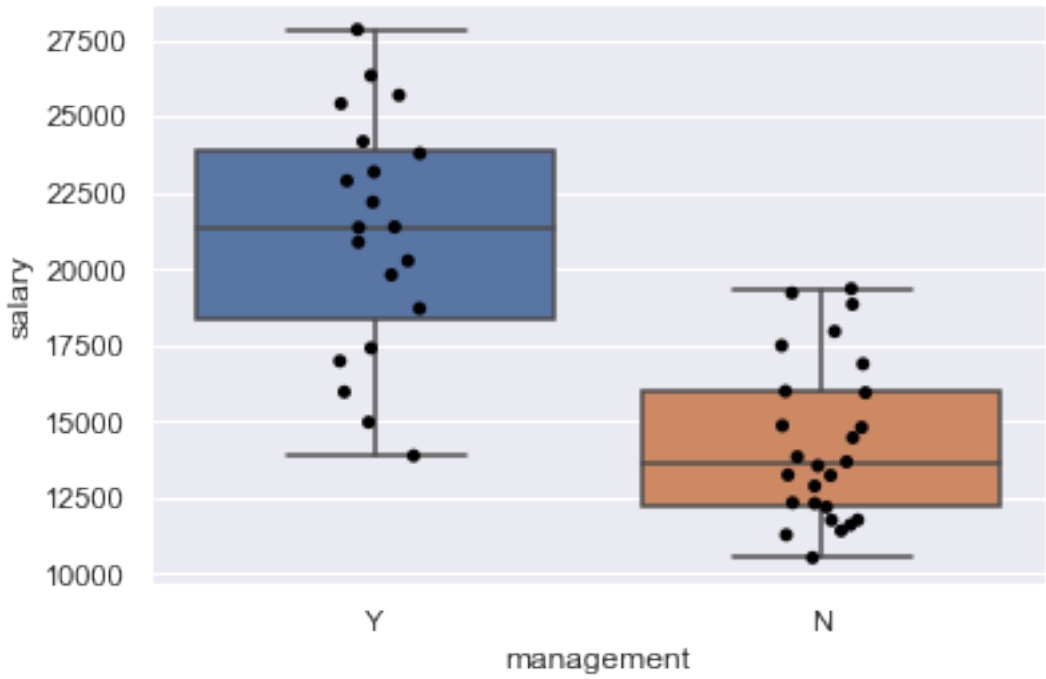
3.3.3 Saving Figures

```
### bitmap format
plt.plot(x, sinus)
plt.savefig("sinus.png")
plt.close()

# Prefer vectorial format (SVG: Scalable Vector Graphics) can be edited with
# Inkscape, Adobe Illustrator, Blender, etc.
plt.plot(x, sinus)
plt.savefig("sinus.svg")
plt.close()

# Or pdf
plt.plot(x, sinus)
plt.savefig("sinus.pdf")
plt.close()
```

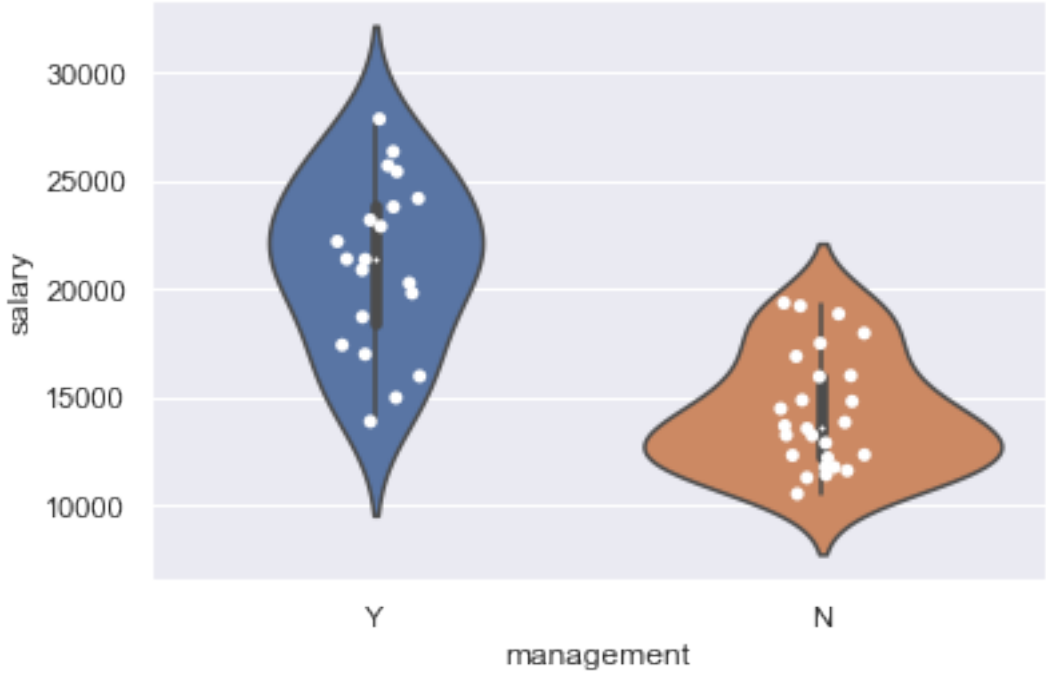
Boxplot and violin plot: one factor

Box plots are non-parametric: they display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution.

```
ax = sns.boxplot(x="management", y="salary", data=salary)
ax = sns.stripplot(x="management", y="salary", data=salary, jitter=True, color=
↪"black")
```

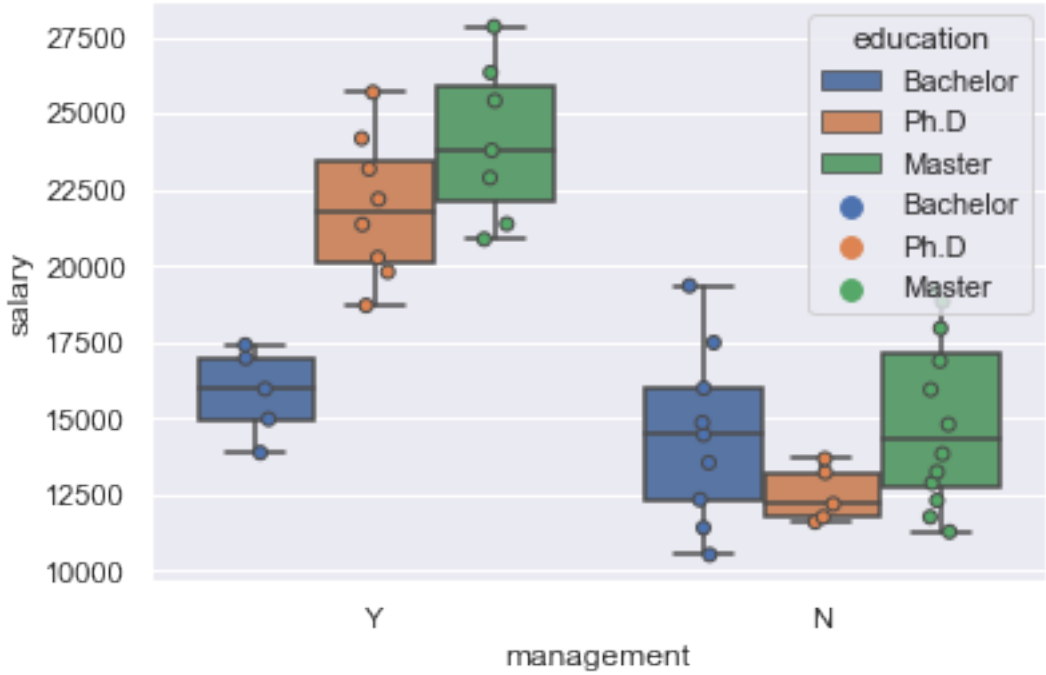



```
ax = sns.violinplot(x="management", y="salary", data=salary)
ax = sns.stripplot(x="management", y="salary", data=salary, jitter=True, color=
↪"white")
```

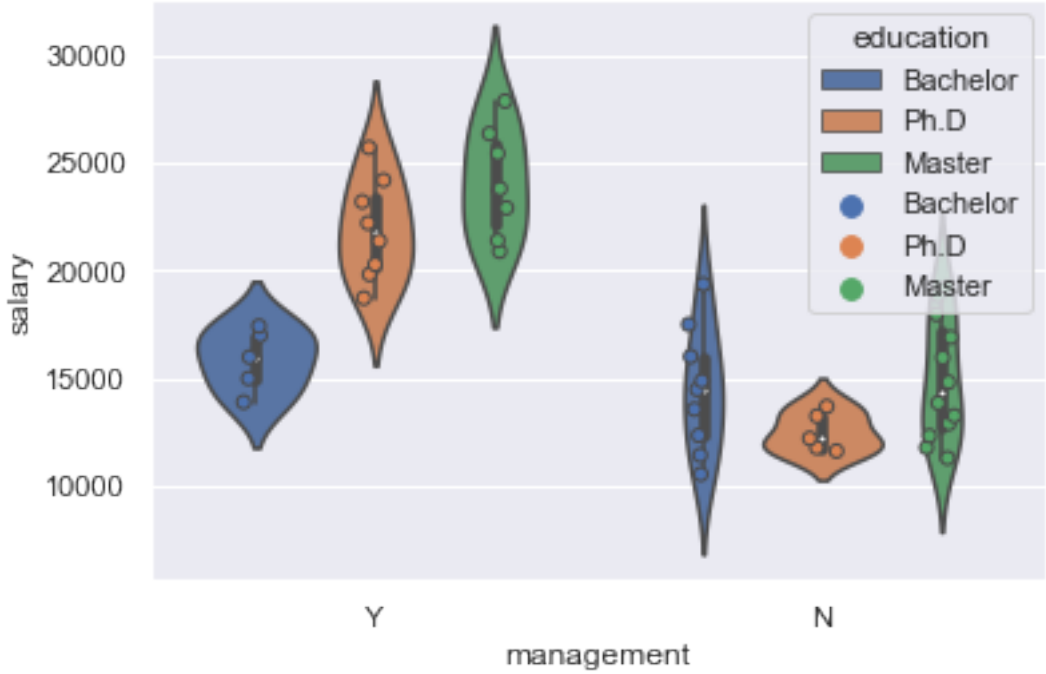


Boxplot and violin plot: two factors

```
ax = sns.boxplot(x="management", y="salary", hue="education", data=salary)
ax = sns.stripplot(x="management", y="salary", hue="education", data=salary,␣
↪jitter=True, dodge=True, linewidth=1)
```



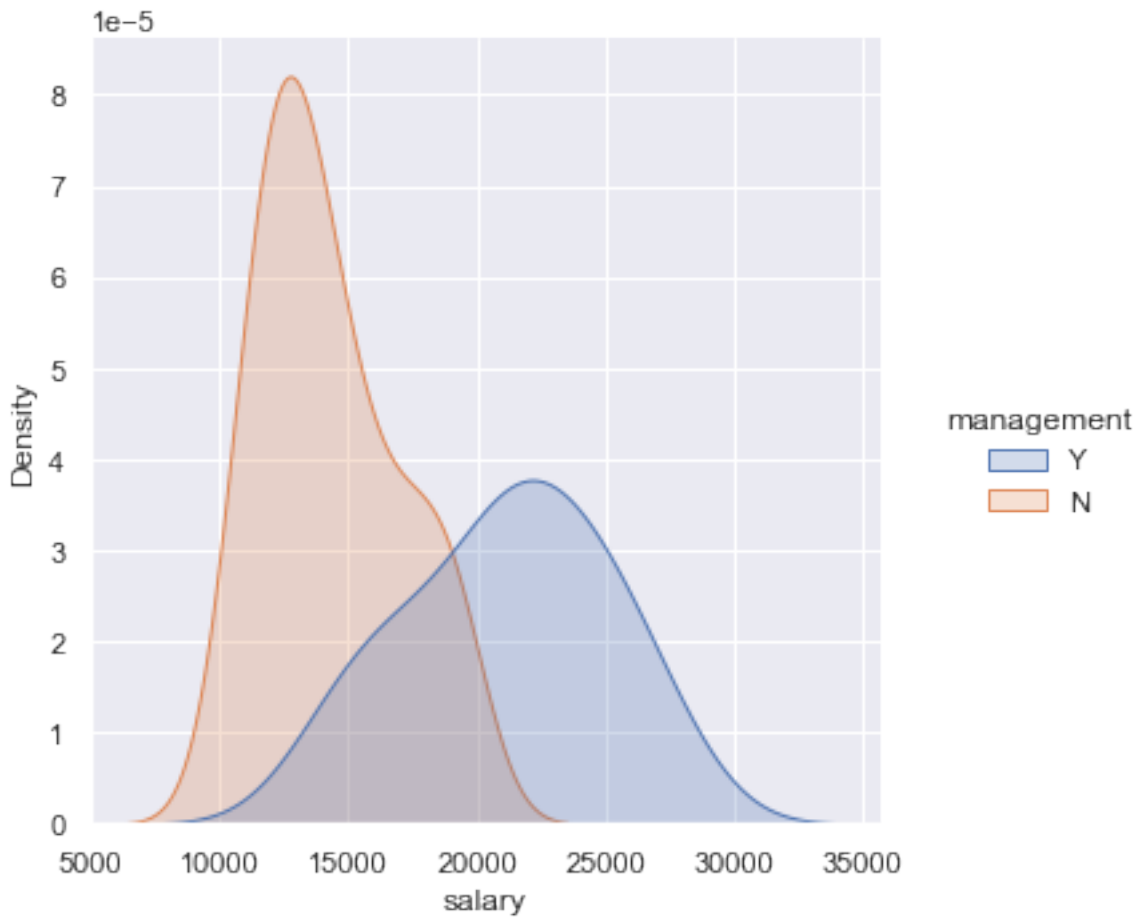

```
ax = sns.violinplot(x="management", y="salary", hue="education", data=salary)
ax = sns.stripplot(x="management", y="salary", hue="education", data=salary,␣
↪jitter=True, dodge=True, linewidth=1)
```



Distributions and density plot

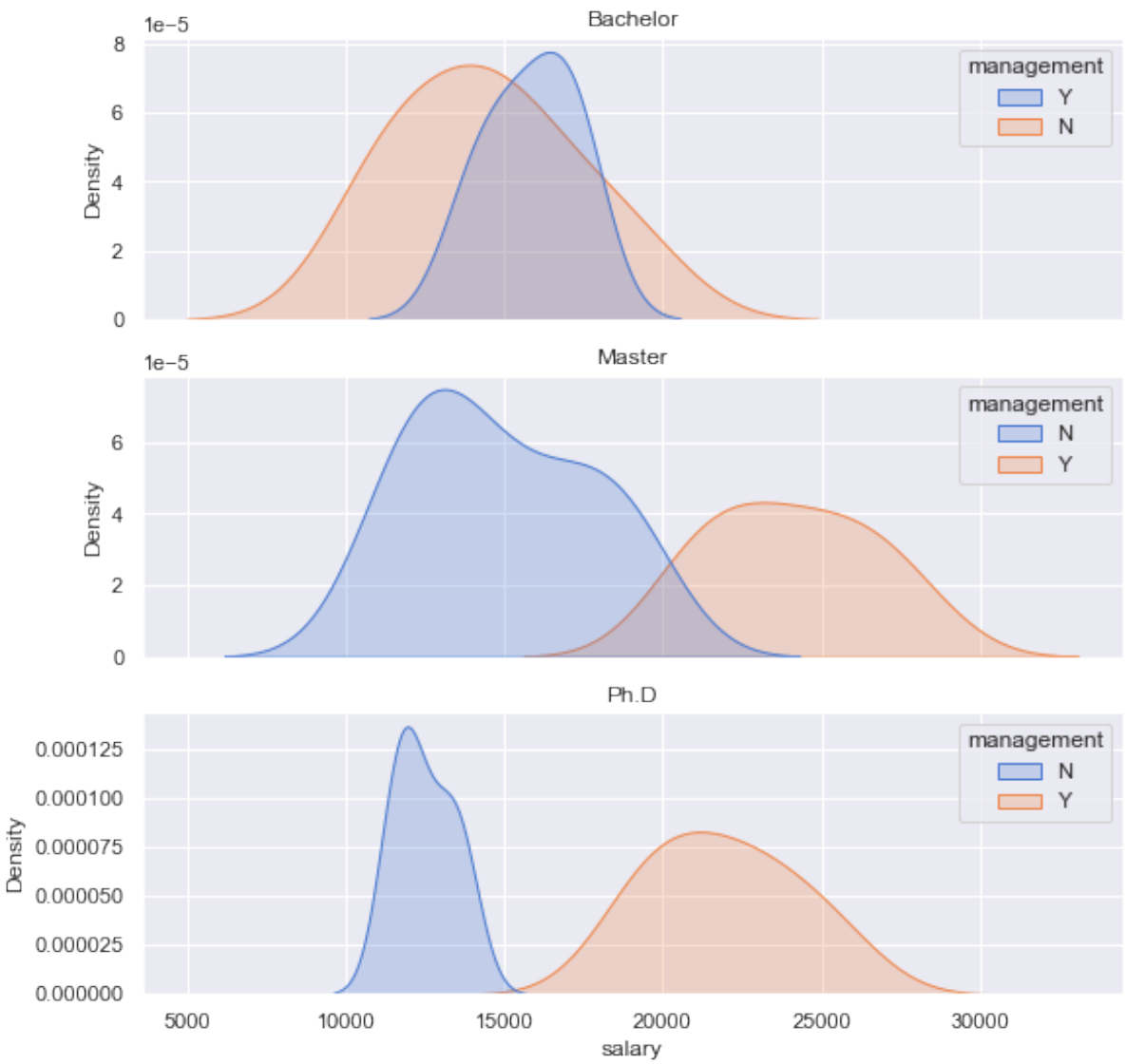
Distributions with seaborn

```
ax = sns.displot(x="salary", hue="management", kind="kde", data=salary, fill=True)
```



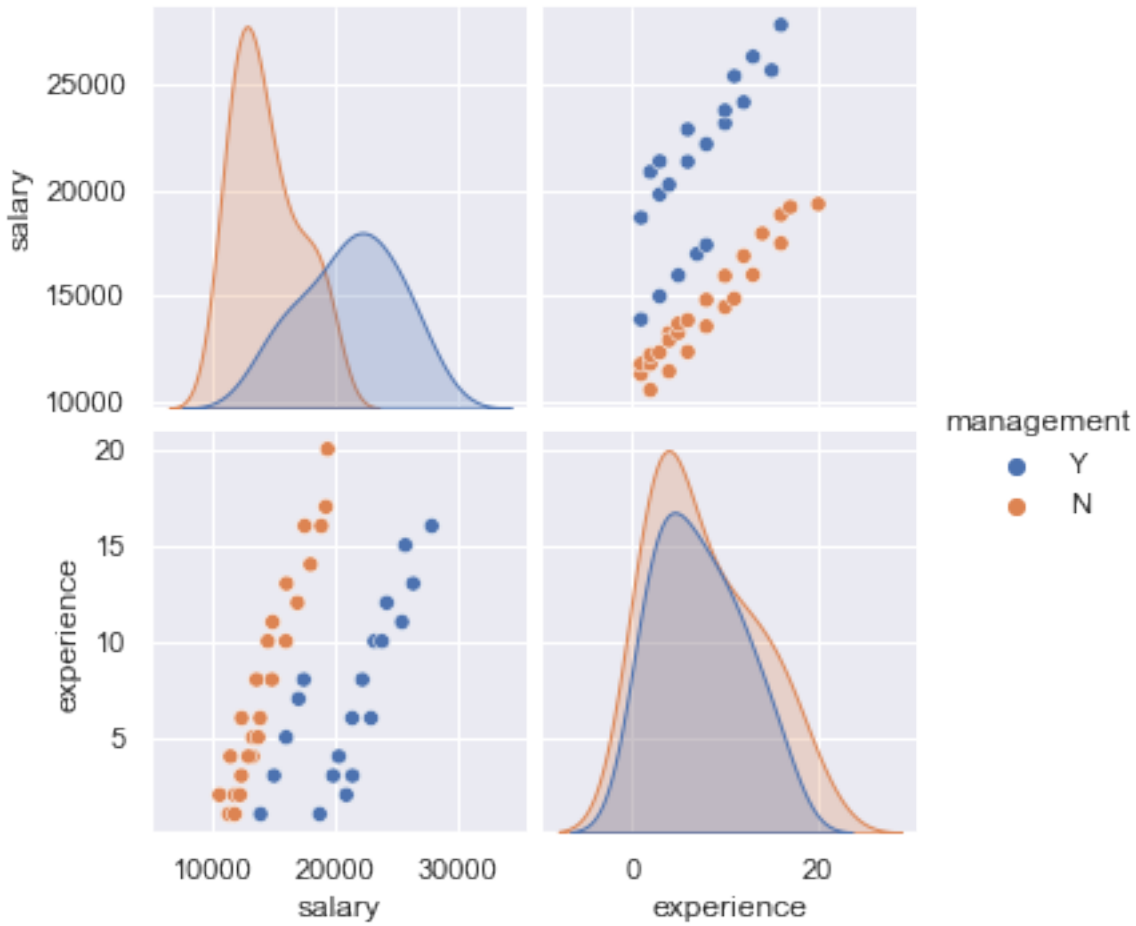
3.3.4 Multiple axis

```
fig, axes = plt.subplots(3, 1, figsize=(9, 9), sharex=True)

i = 0
for edu, d in salary.groupby(['education']):
    sns.kdeplot(x="salary", hue="management", data=d, fill=True, ax=axes[i],␣
↪palette="muted")
    axes[i].set_title(edu)
    i += 1
```



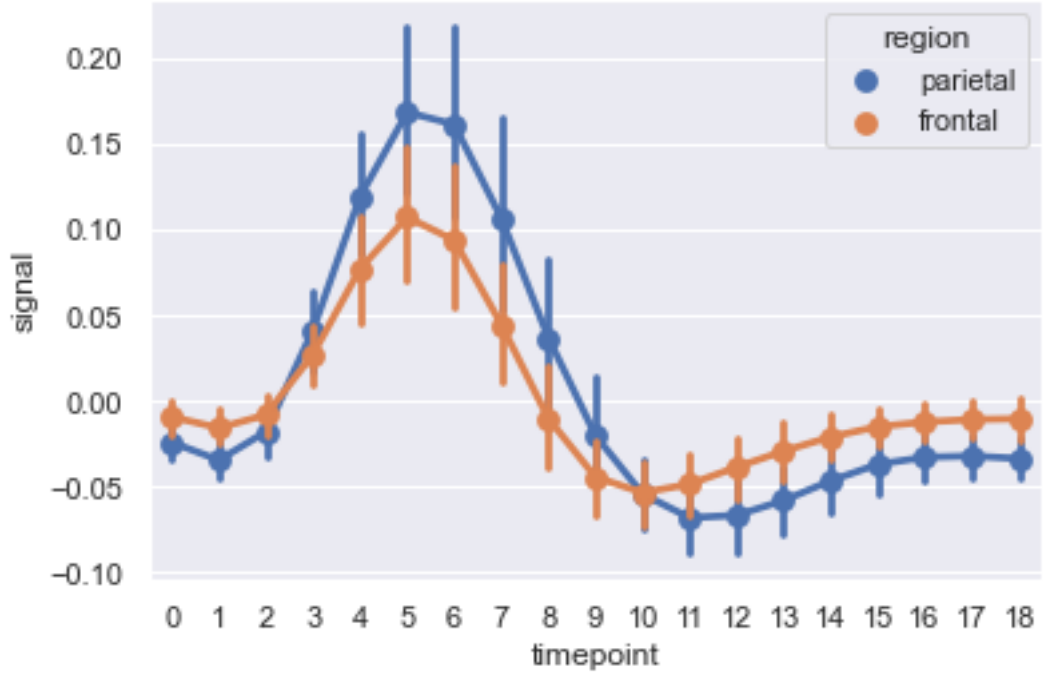
3.3.5 Pairwise scatter plots

```
ax = sns.pairplot(salary, hue="management")
```



3.3.6 Time series

```
import seaborn as sns
sns.set(style="darkgrid")

# Load an example dataset with long-form data
fmri = sns.load_dataset("fmri")

# Plot the responses for different events and regions
ax = sns.pointplot(x="timepoint", y="signal",
             hue="region", style="event",
             data=fmri)
```


region
parietal
frontal
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
signal
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
timepoint

CHAPTER

FOUR

STATISTICS

4.1 Univariate statistics

Basics univariate statistics are required to explore dataset:

- Discover associations between a variable of interest and potential predictors. It is strongly recommended to start with simple univariate methods before moving to complex multivariate predictors.
- Assess the prediction performances of machine learning predictors.
- Most of the univariate statistics are based on the linear model which is one of the main model in machine learning.

4.1.1 Libraries

Data

```
import numpy as np
import pandas as pd
```

Plots

```
import matplotlib.pyplot as plt
import seaborn as sns
```

Statistics

- Basic: scipy.stats
- Advanced: statsmodels. statsmodels API:
  - `statsmodels.api`: Cross-sectional models and methods. Canonically imported using `import statsmodels.api as sm`.
  - `statsmodels.formula.api`: A convenience interface for specifying models using formula strings and DataFrames. Canonically imported using `import statsmodels.formula.api as smf`
  - `statsmodels.tsa.api`: Time-series models and methods. Canonically imported using `import statsmodels.tsa.api as tsa`.

```
import scipy.stats
import statsmodels.api as sm
#import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
from statsmodels.stats.stattools import jarque_bera
```

```
%matplotlib inline
```

Datasets

Salary

```
try:
    salary = pd.read_csv("../datasets/salary_table.csv")
except:
    url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_
↪table.csv'
    salary = pd.read_csv(url)
```

Iris

```
# Load iris datset
iris = sm.datasets.get_rdataset("iris").data
iris.columns = [s.replace('.', '') for s in iris.columns]
```

4.1.2 Estimators of the main statistical measures

Mean

Properties of the expected value operator $\mathrm{E}(\cdot)$ of a random variable X

$$E(X + c) = E(X) + c \tag{4.1}$$
$$E(X + Y) = E(X) + E(Y) \tag{4.2}$$
$$E(aX) = aE(X) \tag{4.3}$$

The estimator $\bar{x}$ on a sample of size n: $x = x_1, ..., x_n$ is given by

$$\bar{x} = \frac{1}{n} \sum_i x_i$$

$\bar{x}$ is itself a random variable with properties:

- $E(\bar{x}) = \bar{x}$,
- $\mathrm{Var}(\bar{x}) = \frac{\mathrm{Var}(X)}{n}$.

Variance

$$\mathrm{Var}(X) = E((X - E(X))^2) = E(X^2) - (E(X))^2$$

The estimator is

$$\sigma_x^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2$$

Note here the subtracted 1 degree of freedom (df) in the divisor. In standard statistical practice, $df = 1$ provides an unbiased estimator of the variance of a hypothetical infinite population. With $df = 0$ it instead provides a maximum likelihood estimate of the variance for normally distributed variables.

Standard deviation

$$\mathrm{Std}(X) = \sqrt{\mathrm{Var}(X)}$$

The estimator is simply $\sigma_x = \sqrt{\sigma_x^2}$.

Covariance

$$\mathrm{Cov}(X, Y) = E((X - E(X))(Y - E(Y))) = E(XY) - E(X)E(Y).$$

Properties:

$$\mathrm{Cov}(X, X) = \mathrm{Var}(X) \tag{4.4}$$
$$\mathrm{Cov}(X, Y) = \mathrm{Cov}(Y, X) \tag{4.5}$$
$$\mathrm{Cov}(cX, Y) = c\,\mathrm{Cov}(X, Y) \tag{4.6}$$
$$\mathrm{Cov}(X + c, Y) = \mathrm{Cov}(X, Y) \tag{4.7}$$
$$\tag{4.8}$$

The estimator with $df = 1$ is

$$\sigma_{xy} = \frac{1}{n-1} \sum_i (x_i - \bar{x})(y_i - \bar{y}).$$

Correlation

$$\mathrm{Cor}(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Std}(X)\,\mathrm{Std}(Y)}$$

The estimator is

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}.$$

Standard Error (SE)

The standard error (SE) is the standard deviation (of the sampling distribution) of a statistic:

$$\mathrm{SE}(X) = \frac{\mathrm{Std}(X)}{\sqrt{n}}.$$

It is most commonly considered for the mean with the estimator

$$\mathrm{SE}(X) = \mathrm{Std}(X) = \sigma_{\bar{x}} \tag{4.9}$$

$$= \frac{\sigma_x}{\sqrt{n}}. \tag{4.10}$$

Descriptives statistics with numpy

- Generate 2 random samples: $x \sim N(1.78, 0.1)$ and $y \sim N(1.66, 0.1)$, both of size 10.
- Compute $\bar{x}, \sigma_x, \sigma_{xy}$ (`xbar`, `xvar`, `xycov`) using only the `np.sum()` operation. Explore the `np.` module to find out which numpy functions performs the same computations and compare them (using `assert`) with your previous results.

Caution! By default `np.var()` used the biased estimator (with ddof=0). Set ddof=1 to use unbiased estimator.

```
n = 10
x = np.random.normal(loc=1.78, scale=.1, size=n)
y = np.random.normal(loc=1.66, scale=.1, size=n)

xbar = np.mean(x)
assert xbar == np.sum(x) / x.shape[0]

xvar = np.var(x, ddof=1)
assert xvar == np.sum((x - xbar) ** 2) / (n - 1)

xycov = np.cov(x, y)
print(xycov)

ybar = np.sum(y) / n
assert np.allclose(xycov[0, 1], np.sum((x - xbar) * (y - ybar)) / (n - 1))
assert np.allclose(xycov[0, 0], xvar)
assert np.allclose(xycov[1, 1], np.var(y, ddof=1))
```

```
[[ 0.01025944 -0.00661557]
 [-0.00661557  0.0167    ]]
```

Descriptives statistics on Iris dataset

With Pandas

Columns' means

```
iris.mean()
```

```
SepalLength    5.843333
SepalWidth     3.057333
PetalLength    3.758000
PetalWidth     1.199333
dtype: float64
```

Columns' std-dev. Pandas normalizes by N-1 by default.

```
iris.std()
```

```
SepalLength    0.828066
SepalWidth     0.435866
PetalLength    1.765298
PetalWidth     0.762238
dtype: float64
```

With Numpy

```
X = iris[['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']].values
iris.columns
X.mean(axis=0)
```

```
array([5.84333333, 3.05733333, 3.758     , 1.19933333])
```

Columns' std-dev. Numpy normalizes by N by default. Set ddof=1 to normalize by N-1 to get the unbiased estimator.

```
X.std(axis=0, ddof=1)
```

```
array([0.82806613, 0.43586628, 1.76529823, 0.76223767])
```

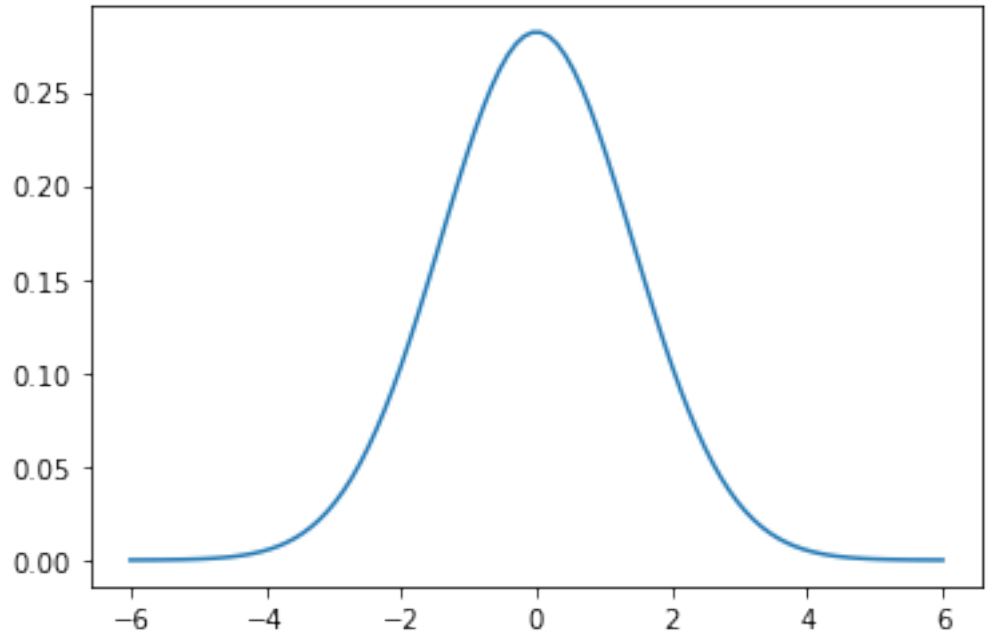
4.1.3 Main distributions

Normal distribution

The normal distribution, noted $\mathcal{N}(\mu, \sigma)$ with parameters: μ mean (location) and $\sigma > 0$ std-dev. Estimators: $\bar{x}$ and σ_x.

The normal distribution, noted $\mathcal{N}$, is useful because of the central limit theorem (CLT) which states that: given certain conditions, the arithmetic mean of a sufficiently large number of iterates of independent random variables, each with a well-defined expected value and well-defined variance, will be approximately normally distributed, regardless of the underlying distribution.

```
mu = 0 # mean
variance = 2 #variance
sigma = np.sqrt(variance) #standard deviation",
x = np.linspace(mu - 3 * variance, mu + 3 * variance, 100)
_ = plt.plot(x, scipy.stats.norm.pdf(x, mu, sigma))
```



The Chi-Square distribution

The chi-square or χ^2_n distribution with n degrees of freedom (df) is the distribution of a sum of the squares of n independent standard normal random variables $\mathcal{N}(0,1)$. Let $X \sim \mathcal{N}(\mu, \sigma^2)$, then, $Z = (X - \mu)/\sigma \sim \mathcal{N}(0,1)$, then:

- The squared standard $Z^2 \sim \chi^2_1$ (one df).
- **The distribution of sum of squares** of n normal random variables: $\sum_i^n Z_i^2 \sim \chi^2_n$

The sum of two χ^2 RV with p and q df is a χ^2 RV with $p + q$ df. This is useful when summing/subtracting sum of squares.

The χ^2-distribution is used to model **errors** measured as **sum of squares** or the distribution of the sample **variance**.

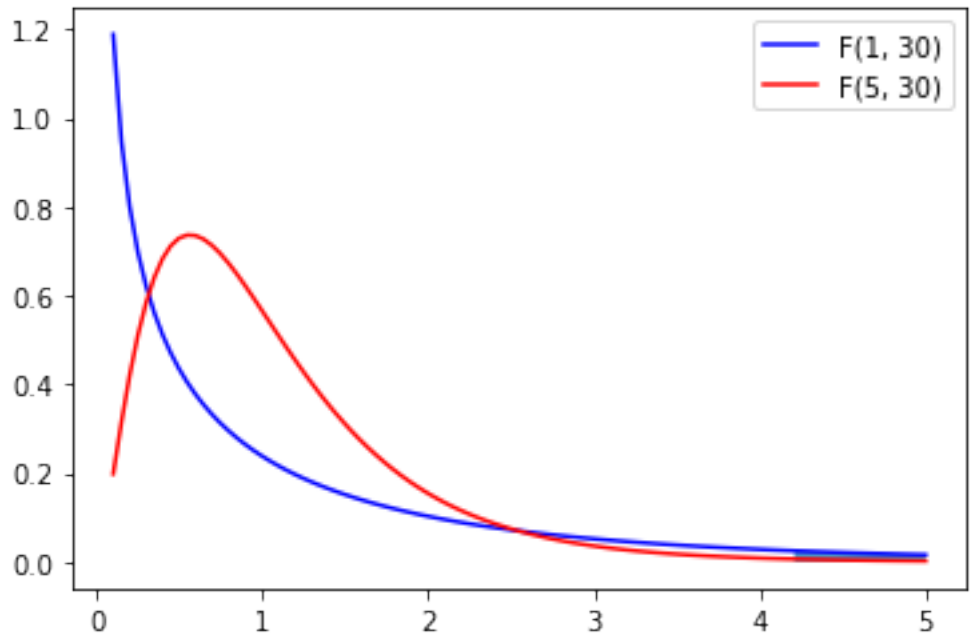
The Fisher's F-distribution

The F-distribution, $F_{n,p}$, with n and p degrees of freedom is the ratio of two independent χ^2 variables. Let $X \sim \chi^2_n$ and $Y \sim \chi^2_p$ then:

$$F_{n,p} = \frac{X/n}{Y/p}$$

The F-distribution plays a central role in hypothesis testing answering the question: **Are two variances equals?, is the ratio or two errors significantly large ?**.

```
fvalues = np.linspace(.1, 5, 100)

# pdf(x, df1, df2): Probability density function at x of F.
plt.plot(fvalues, scipy.stats.f.pdf(fvalues, 1, 30), 'b-', label="F(1, 30)")
plt.plot(fvalues, scipy.stats.f.pdf(fvalues, 5, 30), 'r-', label="F(5, 30)")
plt.legend()

# cdf(x, df1, df2): Cumulative distribution function of F.
# ie.
proba_at_f_inf_3 = scipy.stats.f.cdf(3, 1, 30) # P(F(1,30) < 3)

# ppf(q, df1, df2): Percent point function (inverse of cdf) at q of F.
f_at_proba_inf_95 = scipy.stats.f.ppf(.95, 1, 30) # q such P(F(1,30) < .95)
assert scipy.stats.f.cdf(f_at_proba_inf_95, 1, 30) == .95

# sf(x, df1, df2): Survival function (1 - cdf) at x of F.
proba_at_f_sup_3 = scipy.stats.f.sf(3, 1, 30) # P(F(1,30) > 3)
assert  proba_at_f_inf_3 + proba_at_f_sup_3 == 1

# p-value: P(F(1, 30)) < 0.05
low_proba_fvalues = fvalues[fvalues > f_at_proba_inf_95]
plt.fill_between(low_proba_fvalues, 0, scipy.stats.f.pdf(low_proba_fvalues, 1,␣
↪30),
                 alpha=.8, label="P < 0.05")
plt.show()
```



The Student's t-distribution

Let $M \sim \mathcal{N}(0,1)$ and $V \sim \chi^2_n$. The t-distribution, T_n, with n degrees of freedom is the ratio:

$$T_n = \frac{M}{\sqrt{V/n}}$$

The distribution of the difference between an estimated parameter and its true (or assumed) value divided by the standard deviation of the estimated parameter (standard error) follow a t-distribution. **Is this parameters different from a given value?**

4.1.4 Hypothesis Testing

Examples

- Test a proportion: Biased coin ? 200 heads have been found over 300 flips, is it coins biased ?
- Test the association between two variables.
  - Exemple height and sex: In a sample of 25 individuals (15 females, 10 males), is female height is different from male height ?
  - Exemple age and arterial hypertension: In a sample of 25 individuals is age height correlated with arterial hypertension ?

Steps

1. Model the data
2. Fit: estimate the model parameters (frequency, mean, correlation, regression coeficient)
3. Compute a test statistic from model the parameters.
4. Formulate the null hypothesis: What would be the (distribution of the) test statistic if the observations are the result of pure chance.
5. Compute the probability (p-value) to obtain a larger value for the test statistic by chance (under the null hypothesis).

Flip coin: Simplified example

Biased coin ? 2 heads have been found over 3 flips, is it coins biased ?

1. Model the data: number of heads follow a Binomial disctribution.
2. Compute model parameters: N=3, P = the frequency of number of heads over the number of flip: 2/3.
3. Compute a test statistic, same as frequency.
4. Under the null hypothesis the distribution of the number of tail is:

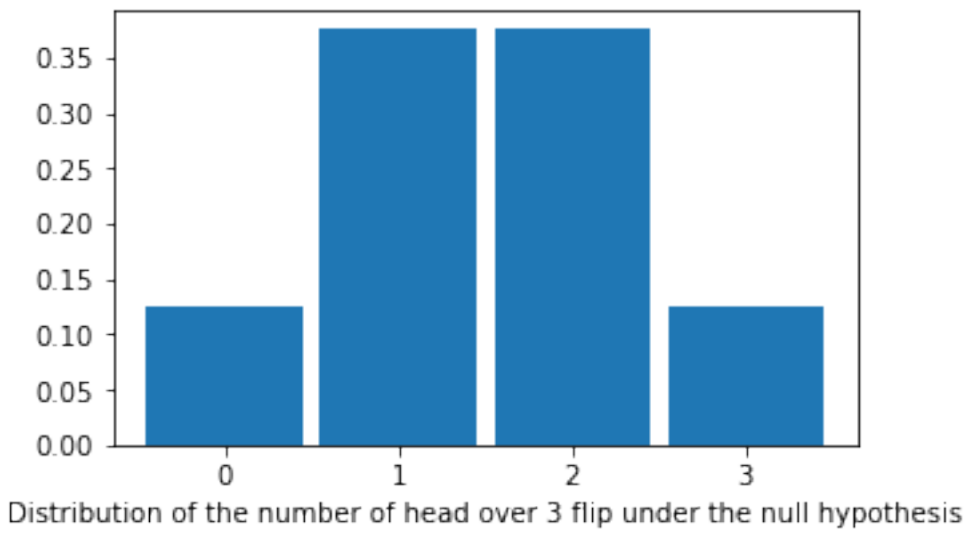
1	2	3	count #heads
			0
H			1
	H		1
		H	1
H	H		2
H		H	2
	H	H	2
H	H	H	3

8 possibles configurations, probabilities of differents values for p are: x measure the number of success.

- $P(x = 0) = 1/8$
- $P(x = 1) = 3/8$
- $P(x = 2) = 3/8$
- $P(x = 3) = 1/8$

```
plt.figure(figsize=(5, 3))
plt.bar([0, 1, 2, 3], [1/8, 3/8, 3/8, 1/8], width=0.9)
_ = plt.xticks([0, 1, 2, 3], [0, 1, 2, 3])
plt.xlabel("Distribution of the number of head over 3 flip under the null␣
↪hypothesis")
```

```
Text(0.5, 0, 'Distribution of the number of head over 3 flip under the null␣
↪hypothesis')
```



3. Compute the probability (p-value) to observe a value larger or equal that 2 under the null

hypothesis ? This probability is the p-value:

$$P(x \geq 2|H_0) = P(x = 2) + P(x = 3) = 3/8 + 1/8 = 4/8 = 1/2$$

Flip coin: Real Example

Biased coin ? 60 heads have been found over 100 flips, is it coins biased ?

1. Model the data: number of heads follow a Binomial disctribution.
2. Compute model parameters: N=100, P=60/100.
3. Compute a test statistic, same as frequency.
4. Compute a test statistic: 60/100.
5. Under the null hypothesis the distribution of the number of tail (k) follow the **binomial distribution** of parameters N=100, **P=0.5**:

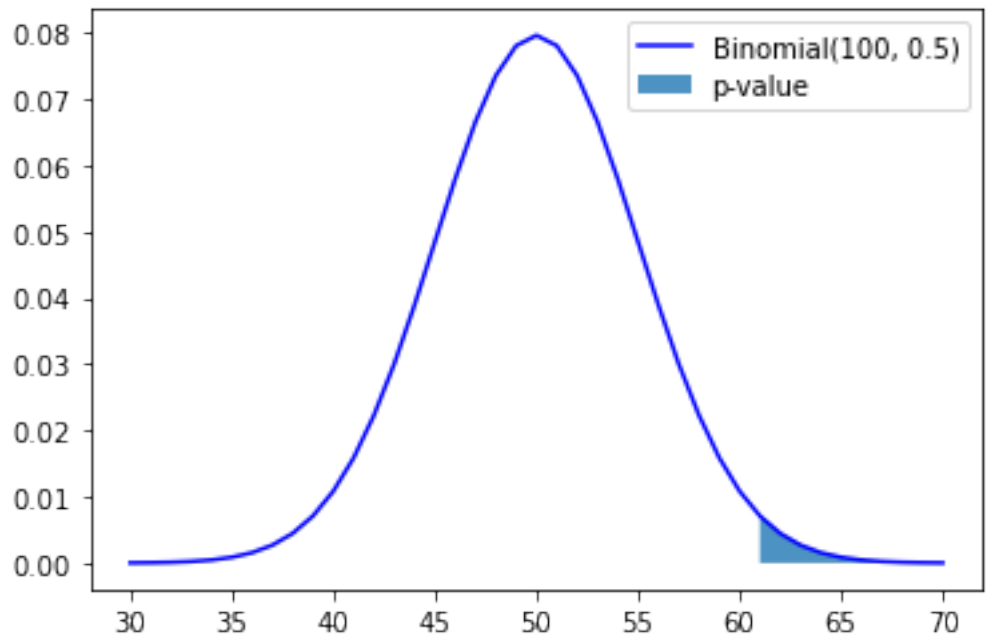
$$Pr(X = k|H_0) = Pr(X = k|n = 100, p = 0.5) = \binom{100}{k} 0.5^k (1 - 0.5)^{(100-k)}.$$

$$\begin{aligned} P(X = k \geq 60|H_0) &= \sum_{k=60}^{100} \binom{100}{k} 0.5^k (1 - 0.5)^{(100-k)} \\ &= 1 - \sum_{k=1}^{60} \binom{100}{k} 0.5^k (1 - 0.5)^{(100-k)}, \text{the cumulative distribution function.} \end{aligned}$$

Use tabulated binomial distribution

```
succes = np.linspace(30, 70, 41)
plt.plot(succes, scipy.stats.binom.pmf(succes, 100, 0.5),
         'b-', label="Binomial(100, 0.5)")
upper_succes_tvalues = succes[succes > 60]
plt.fill_between(upper_succes_tvalues, 0,
                 scipy.stats.binom.pmf(upper_succes_tvalues, 100, 0.5),
                 alpha=.8, label="p-value")
_ = plt.legend()


pval = 1 - scipy.stats.binom.cdf(60, 100, 0.5)
print(pval)
```

```
0.01760010010885238
```



Random sampling of the Binomial distribution under the null hypothesis

```
sccess_h0 = scipy.stats.binom.rvs(100, 0.5, size=10000, random_state=4)
print(sccess_h0)

pval_rnd = np.sum(sccess_h0 >= 60) / (len(sccess_h0) + 1)
print("P-value using monte-carlo sampling of the Binomial distribution under H0=",
      pval_rnd)
```

```
[60 52 51 ... 45 51 44]
P-value using monte-carlo sampling of the Binomial distribution under H0= 0.
↪025897410258974102
```

One sample t-test

The one-sample t-test is used to determine whether a sample comes from a population with a specific mean. For example you want to test if the average height of a population is $1.75\ m$.

Assumptions

1. Independence of **residuals** (ε_i). This assumptions **must** be satisfied.
2. Normality of residuals. Approximately normally distributed can be accepted.

Remarks: Although the parent population does not need to be normally distributed, the distribution of the population of sample means, $\overline{x}$, is assumed to be normal. By the central limit theorem, if the sampling of the parent population is independent then the sample means will be approximately normal.

1 Model the data

Assume that height is normally distributed: $X \sim \mathcal{N}(\mu, \sigma)$, ie:

$$\text{height}_i = \text{average height over the population} + \text{error}_i \tag{4.11}$$
$$x_i = \bar{x} + \varepsilon_i \tag{4.12}$$

The ε_i are called the residuals

2 Fit: estimate the model parameters

$\bar{x}, s_x$ are the estimators of μ, σ.

3 Compute a test statistic

In testing the null hypothesis that the population mean is equal to a specified value $\mu_0 = 1.75$, one uses the statistic:

$$t = \frac{\text{difference of means}}{\text{std-dev of noise}}\sqrt{n} \tag{4.13}$$
$$t = \text{effect size}\sqrt{n} \tag{4.14}$$
$$t = \frac{\bar{x} - \mu_0}{s_x}\sqrt{n} \tag{4.15}$$

4 Compute the probability of the test statistic under the null hypotheis. This require to have the distribution of the t statistic under H_0.

Example

Given the following samples, we will test whether its true mean is 1.75.

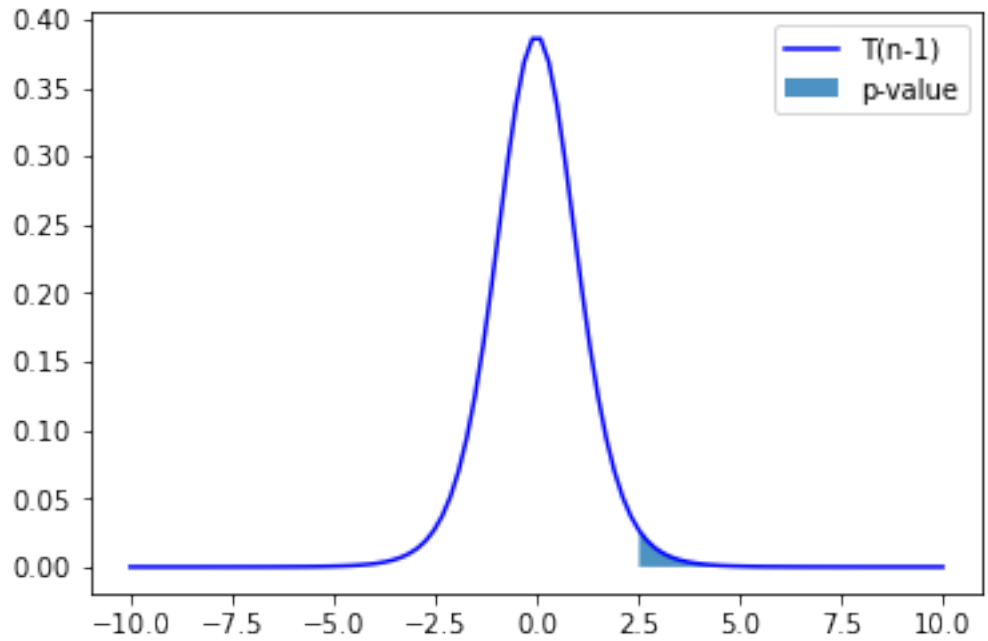
Warning, when computing the std or the variance, set `ddof=1`. The default value, `ddof=0`, leads to the biased estimator of the variance.

```
x = [1.83,  1.83,  1.73,  1.82,  1.83,  1.73,  1.99,  1.85,  1.68,  1.87]

xbar = np.mean(x)  # sample mean
mu0 = 1.75  # hypothesized value
s = np.std(x, ddof=1)  # sample standard deviation
n = len(x)  # sample size

print(xbar)

tobs = (xbar - mu0) / (s / np.sqrt(n))
print(tobs)
```

```
1.816
2.3968766311585883
```

The **:math:`p`-value** is the probability to observe a value t more extreme than the observed one t_{obs} under the null hypothesis H_0: $P(t > t_{obs}|H_0)$

```
tvalues = np.linspace(-10, 10, 100)
plt.plot(tvalues, scipy.stats.t.pdf(tvalues, n-1), 'b-', label="T(n-1)")
upper_tval_tvalues = tvalues[tvalues > tobs]
plt.fill_between(upper_tval_tvalues, 0, scipy.stats.t.pdf(upper_tval_tvalues, n-
↪1),
                 alpha=.8, label="p-value")
_ = plt.legend()
```



4.1.5 Testing pairwise associations

Univariate statistical analysis: explore association betweens pairs of variables.

- In statistics, a **categorical variable** or **factor** is a variable that can take on one of a limited, and usually fixed, number of possible values, thus assigning each individual to a particular group or "category". The levels are the possibles values of the variable. Number of levels = 2: binomial; Number of levels > 2: multinomial. There is no intrinsic ordering to the categories. For example, gender is a categorical variable having two categories (male and female) and there is no intrinsic ordering to the categories. For example, Sex (Female, Male), Hair color (blonde, brown, etc.).
- An **ordinal variable** is a categorical variable with a clear ordering of the levels. For example: drinks per day (none, small, medium and high).
- A **continuous** or **quantitative variable** $x \in \mathbb{R}$ is one that can take any value in a range of possible values, possibly infinite. E.g.: salary, experience in years, weight.

What statistical test should I use?

See: http://www.ats.ucla.edu/stat/mult_pkg/whatstat/

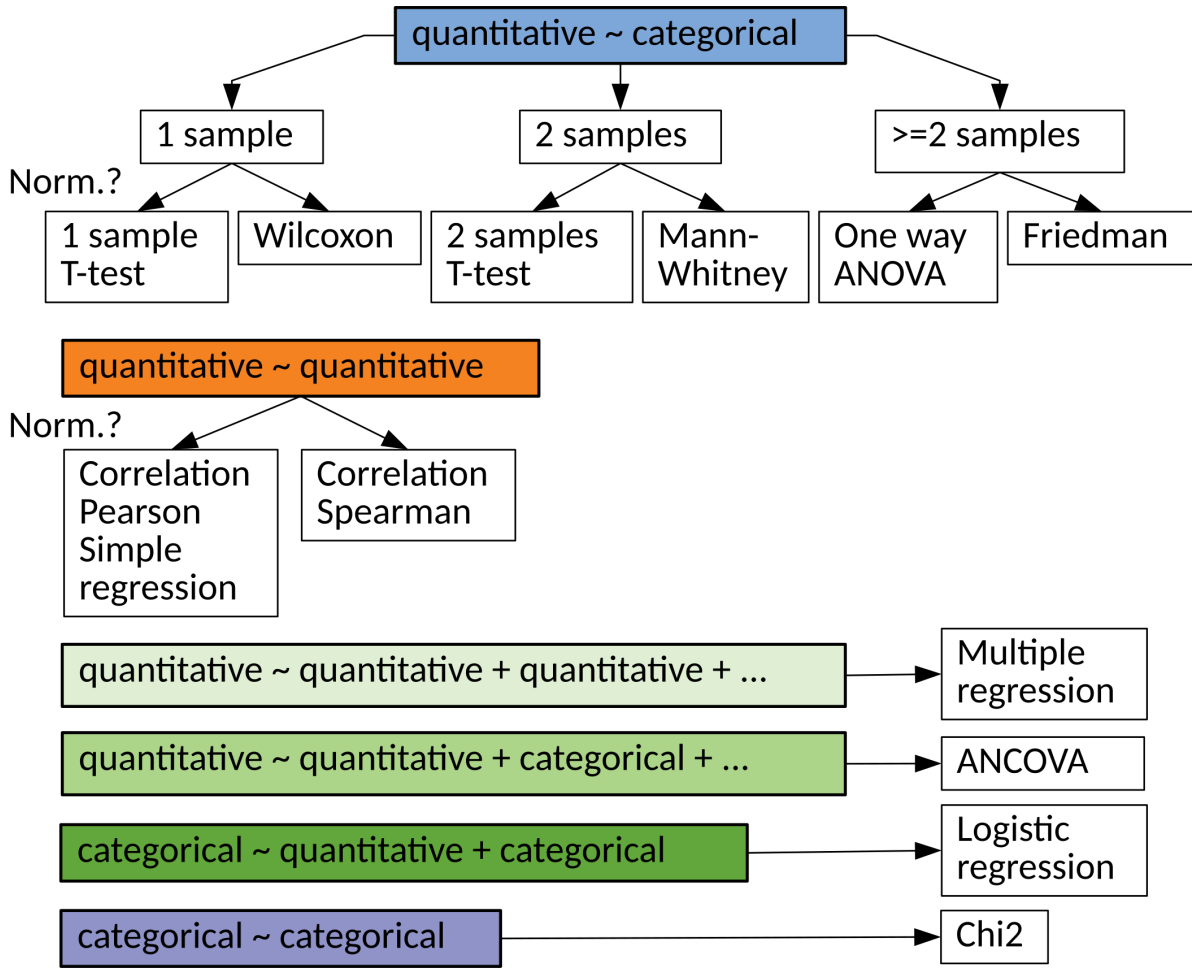


Fig. 1: Statistical tests

4.1.6 Pearson correlation test: test association between two quantitative variables

Test the correlation coefficient of two quantitative variables. The test calculates a Pearson correlation coefficient and the p-value for testing non-correlation.

Let x and y two quantitative variables, where n samples were obeserved. The linear correlation coeficient is defined as :

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}.$$

Under H_0, the test statistic $t = \sqrt{n-2}\frac{r}{\sqrt{1-r^2}}$ follow Student distribution with $n-2$ degrees of freedom.

```
n = 50
x = np.random.normal(size=n)
y = 2 * x + np.random.normal(size=n)

# Compute with scipy
```

(continues on next page)

(continued from previous page)

```
cor, pval = scipy.stats.pearsonr(x, y)
print(cor, pval)
```

```
0.8297883544365898 9.497428029783463e-14
```

4.1.7 Two sample (Student) t-test: compare two means

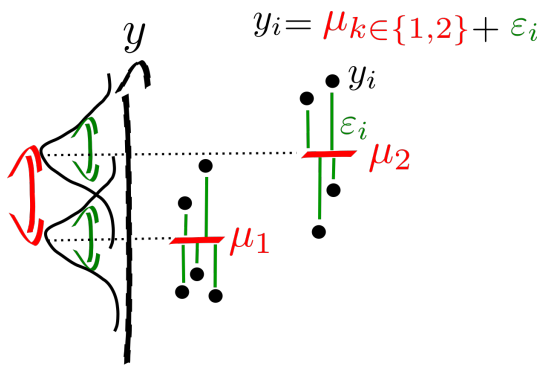


Fig. 2: Two-sample model

The two-sample t-test (Snedecor and Cochran, 1989) is used to determine if two population means are equal. There are several variations on this test. If data are paired (e.g. 2 measures, before and after treatment for each individual) use the one-sample t-test of the difference. The variances of the two samples may be assumed to be equal (a.k.a. homoscedasticity) or unequal (a.k.a. heteroscedasticity).

Assumptions

1. Independence of **residuals** (ε_i). This assumptions **must** be satisfied.
2. Normality of residuals. Approximately normally distributed can be accepted.
3. Homosedasticity use T-test, Heterosedasticity use Welch t-test.

1. Model the data

Assume that the two random variables are normally distributed: $y_1 \sim \mathcal{N}(\mu_1, \sigma_1), y_2 \sim \mathcal{N}(\mu_2, \sigma_2)$.

2. Fit: estimate the model parameters

Estimate means and variances: $\bar{y_1}, s^2_{y_1}, \bar{y_2}, s^2_{y_2}$.

3. t-test

The general principle is

$$
\begin{aligned}
t &= \frac{\text{difference of means}}{\text{standard dev of error}} && (4.16)\\
&= \frac{\text{difference of means}}{\text{its standard error}} && (4.17)\\
&= \frac{\bar{y_1} - \bar{y_2}}{\sqrt{\sum \varepsilon^2}}\sqrt{n-2} && (4.18)\\
&= \frac{\bar{y_1} - \bar{y_2}}{s_{\bar{y_1}-\bar{y_2}}} && (4.19)
\end{aligned}
$$

Since y_1 and y_2 are independant:

$$
\begin{aligned}
s^2_{\bar{y_1}-\bar{y_2}} &= s^2_{\bar{y_1}} + s^2_{\bar{y_2}} = \frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2} && (4.20)\\
&\text{thus} && (4.21)\\
s_{\bar{y_1}-\bar{y_2}} &= \sqrt{\frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2}} && (4.22)
\end{aligned}
$$

Equal or unequal sample sizes, unequal variances (Welch's t-test)

Welch's t-test defines the t statistic as

$$t = \frac{\bar{y_1} - \bar{y_2}}{\sqrt{\frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2}}}.$$

To compute the p-value one needs the degrees of freedom associated with this variance estimate. It is approximated using the Welch–Satterthwaite equation:

$$\nu \approx \frac{\left(\frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2}\right)^2}{\frac{s^4_{y_1}}{n_1^2(n_1-1)} + \frac{s^4_{y_2}}{n_2^2(n_2-1)}}.$$

Equal or unequal sample sizes, equal variances

If we assume equal variance (ie, $s^2_{y_1} = s^2_{y_1} = s^2$), where s^2 is an estimator of the common variance of the two samples:

$$s^2 = \frac{s^2_{y_1}(n_1 - 1) + s^2_{y_2}(n_2 - 1)}{n_1 + n_2 - 2} \tag{4.23}$$

$$= \frac{\sum_i^{n_1}(y_{1i} - \bar{y_1})^2 + \sum_j^{n_2}(y_{2j} - \bar{y_2})^2}{(n_1 - 1) + (n_2 - 1)} \tag{4.24}$$

then

$$s_{\bar{y_1} - \bar{y_2}} = \sqrt{\frac{s^2}{n_1} + \frac{s^2}{n_2}} = s\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

Therefore, the t statistic, that is used to test whether the means are different is:

$$t = \frac{\bar{y_1} - \bar{y_2}}{s \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}},$$

Equal sample sizes, equal variances

If we simplify the problem assuming equal samples of size $n_1 = n_2 = n$ we get

$$t = \frac{\bar{y_1} - \bar{y_2}}{s\sqrt{2}} \cdot \sqrt{n} \tag{4.25}$$

$$\approx \text{effect size} \cdot \sqrt{n} \tag{4.26}$$

$$\approx \frac{\text{difference of means}}{\text{standard deviation of the noise}} \cdot \sqrt{n} \tag{4.27}$$

Example

Given the following two samples, test whether their means are equal using the **standard t-test, assuming equal variance**.

```
height = np.array([ 1.83,  1.83,  1.73,  1.82,  1.83,  1.73,  1.99,  1.85,  1.68,␣
↪ 1.87,
                    1.66,  1.71,  1.73,  1.64,  1.70,  1.60,  1.79,  1.73,  1.62,␣
↪ 1.77])

grp = np.array(["M"] * 10 + ["F"] * 10)

# Compute with scipy
scipy.stats.ttest_ind(height[grp == "M"], height[grp == "F"], equal_var=True)
```

```
Ttest_indResult(statistic=3.5511519888466885, pvalue=0.00228208937112721)
```

4.1.8 ANOVA F-test (quantitative ~ categorial (>=2 levels))

Analysis of variance (ANOVA) provides a statistical test of whether or not the means of several (k) groups are equal, and therefore generalizes the t-test to more than two groups. ANOVAs are useful for comparing (testing) three or more means (groups or variables) for statistical significance. It is conceptually similar to multiple two-sample t-tests, but is less conservative.

Here we will consider one-way ANOVA with one independent variable, ie one-way anova.

Wikipedia:

- Test if any group is on average superior, or inferior, to the others versus the null hypothesis that all four strategies yield the same mean response
- Detect any of several possible differences.
- The advantage of the ANOVA F-test is that we do not need to pre-specify which strategies are to be compared, and we do not need to adjust for making multiple comparisons.
- The disadvantage of the ANOVA F-test is that if we reject the null hypothesis, we do not know which strategies can be said to be significantly different from the others.

Assumptions

1. The samples are randomly selected in an independent manner from the k populations.
2. All k populations have distributions that are approximately normal. Check by plotting groups distribution.
3. The k population variances are equal. Check by plotting groups distribution.

1. Model the data

Is there a difference in Petal Width in species from iris dataset. Let y_1, y_2 and y_3 be Petal Width in three species.

Here we assume (see assumptions) that the three populations were sampled from three random variables that are normally distributed. I.e., $Y_1 \sim N(\mu_1, \sigma_1), Y_2 \sim N(\mu_2, \sigma_2)$ and $Y_3 \sim N(\mu_3, \sigma_3)$.

2. Fit: estimate the model parameters

Estimate means and variances: $\bar{y}_i, \sigma_i, \quad \forall i \in \{1, 2, 3\}$.

3. F-test

The formula for the one-way ANOVA F-test statistic is

$$F = \frac{\text{Explained variance}}{\text{Unexplained variance}} \tag{4.28}$$

$$= \frac{\text{Between-group variability}}{\text{Within-group variability}} = \frac{s_B^2}{s_W^2}. \tag{4.29}$$

The "explained variance", or "between-group variability" is

$$s_B^2 = \sum_i n_i(\bar{y}_{i\cdot} - \bar{y})^2/(K-1),$$

where $\bar{y}_{i\cdot}$ denotes the sample mean in the ith group, n_i is the number of observations in the ith group, $\bar{y}$ denotes the overall mean of the data, and K denotes the number of groups.

The "unexplained variance", or "within-group variability" is

$$s_W^2 = \sum_{ij}(y_{ij} - \bar{y}_{i\cdot})^2/(N-K),$$

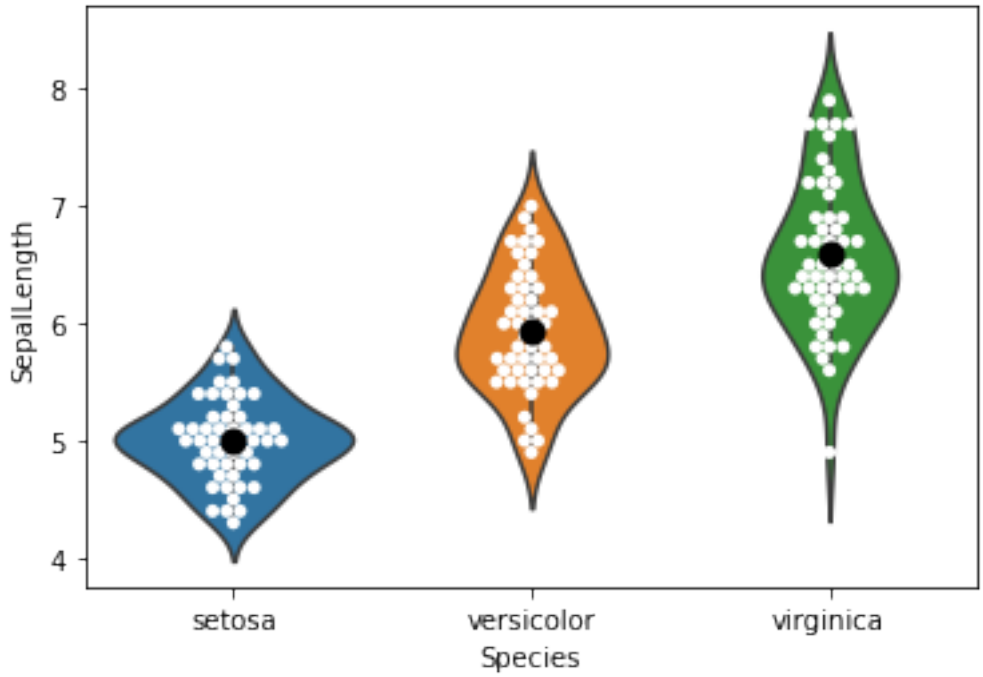
where y_{ij} is the jth observation in the ith out of K groups and N is the overall sample size. This F-statistic follows the F-distribution with $K-1$ and $N-K$ degrees of freedom under the null hypothesis. The statistic will be large if the between-group variability is large relative to the within-group variability, which is unlikely to happen if the population means of the groups all have the same value.

Note that when there are only two groups for the one-way ANOVA F-test, $F = t^2$ where t is the Student's t statistic.

Iris dataset:

```
# Group means
means = iris.groupby("Species").mean().reset_index()
print(means)

# Group Stds (equal variances ?)
stds = iris.groupby("Species").std(ddof=1).reset_index()
print(stds)

# Plot groups
ax = sns.violinplot(x="Species", y="SepalLength", data=iris)
ax = sns.swarmplot(x="Species", y="SepalLength", data=iris,
                   color="white")
ax = sns.swarmplot(x="Species", y="SepalLength",  color="black", data=means,␣
↪size=10)

# ANOVA
lm = smf.ols('SepalLength ~ Species', data=iris).fit()
sm.stats.anova_lm(lm, typ=2) # Type 2 ANOVA DataFrame
```

```
      Species  SepalLength  SepalWidth  PetalLength  PetalWidth
0      setosa        5.006       3.428        1.462       0.246
1  versicolor        5.936       2.770        4.260       1.326
2   virginica        6.588       2.974        5.552       2.026
      Species  SepalLength  SepalWidth  PetalLength  PetalWidth
0      setosa     0.352490    0.379064     0.173664    0.105386
1  versicolor     0.516171    0.313798     0.469911    0.197753
2   virginica     0.635880    0.322497     0.551895    0.274650
```



4.1.9 Chi-square, χ^2 (categorial ~ categorial)

Computes the chi-square, χ^2, statistic and p-value for the hypothesis test of independence of frequencies in the observed contingency table (cross-table). The observed frequencies are tested against an expected contingency table obtained by computing expected frequencies based on the marginal sums under the assumption of independence.

Example: 20 participants: 10 exposed to some chemical product and 10 non exposed (exposed = 1 or 0). Among the 20 participants 10 had cancer 10 not (cancer = 1 or 0). χ^2 tests the association between those two variables.

```
# Dataset:
# 15 samples:
# 10 first exposed
exposed = np.array([1] * 10 + [0] * 10)
# 8 first with cancer, 10 without, the last two with.
cancer = np.array([1] * 8 + [0] * 10 + [1] * 2)

crosstab = pd.crosstab(exposed, cancer, rownames=['exposed'],
                       colnames=['cancer'])
print("Observed table:")
```

(continues on next page)

(continued from previous page)

```
print("---------------")
print(crosstab)

chi2, pval, dof, expected = scipy.stats.chi2_contingency(crosstab)
print("Statistics:")
print("-----------")
print("Chi2 = %f, pval = %f" % (chi2, pval))
print("Expected table:")
print("---------------")
print(expected)
```

```
Observed table:
---------------
cancer   0  1
exposed
0        8  2
1        2  8
Statistics:
-----------
Chi2 = 5.000000, pval = 0.025347
Expected table:
---------------
[[5. 5.]
 [5. 5.]]
```

Computing expected cross-table

```
# Compute expected cross-table based on proportion
exposed_marg = crosstab.sum(axis=0)
exposed_freq = exposed_marg / exposed_marg.sum()

cancer_marg = crosstab.sum(axis=1)
cancer_freq = cancer_marg / cancer_marg.sum()

print('Exposed frequency? Yes: %.2f' % exposed_freq[0],
      'No: %.2f' % exposed_freq[1])
print('Cancer frequency? Yes: %.2f' % cancer_freq[0],
      'No: %.2f' % cancer_freq[1])

print('Expected frequencies:')
print(np.outer(exposed_freq, cancer_freq))

print('Expected cross-table (frequencies * N): ')
print(np.outer(exposed_freq, cancer_freq) * len(exposed))
```

```
Exposed frequency? Yes: 0.50 No: 0.50
Cancer frequency? Yes: 0.50 No: 0.50
Expected frequencies:
[[0.25 0.25]
```

(continues on next page)

(continued from previous page)

```
 [0.25 0.25]]
Expected cross-table (frequencies * N):
[[5. 5.]
 [5. 5.]]
```

4.1.10 Non-parametric test of pairwise associations

Spearman rank-order correlation (quantitative ~ quantitative)

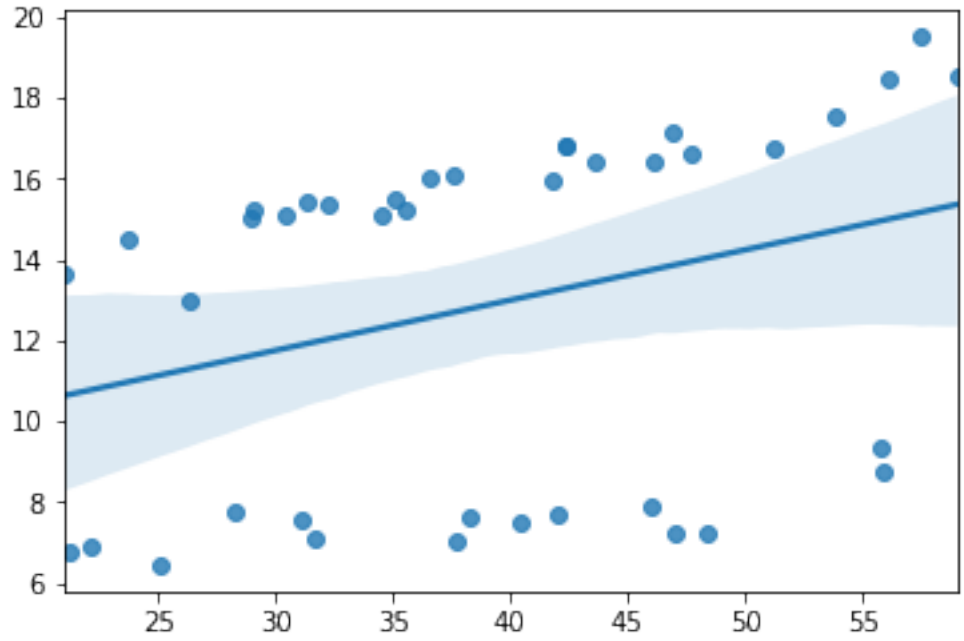
The Spearman correlation is a non-parametric measure of the monotonicity of the relationship between two datasets.

When to use it? Observe the data distribution: - presence of **outliers** - the distribution of the residuals is not Gaussian.

Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

```
np.random.seed(3)

# Age uniform distribution between 20 and 40
age = np.random.uniform(20, 60, 40)

# Systolic blood presure, 2 groups:
# - 15 subjects at 0.05 * age + 6
# - 25 subjects at 0.15 * age + 10
sbp = np.concatenate((0.05 * age[:15] + 6, 0.15 * age[15:] + 10)) + \
    .5 * np.random.normal(size=40)

sns.regplot(x=age, y=sbp)

# Non-Parametric Spearman
cor, pval = scipy.stats.spearmanr(age, sbp)
print("Non-Parametric Spearman cor test, cor: %.4f, pval: %.4f" % (cor, pval))

# "Parametric Pearson cor test
cor, pval = scipy.stats.pearsonr(age, sbp)
print("Parametric Pearson cor test: cor: %.4f, pval: %.4f" % (cor, pval))
```

```
Non-Parametric Spearman cor test, cor: 0.5122, pval: 0.0007
Parametric Pearson cor test: cor: 0.3085, pval: 0.0528
```



Wilcoxon signed-rank test (quantitative ~ cte)

Source: https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test). It is equivalent to one-sample test of the difference of paired samples.

It can be used as an alternative to the paired Student's t-test, t-test for matched pairs, or the t-test for dependent samples when the population cannot be assumed to be normally distributed.

When to use it? Observe the data distribution: - presence of outliers - the distribution of the residuals is not Gaussian

It has a lower sensitivity compared to t-test. May be problematic to use when the sample size is small.

Null hypothesis H_0: difference between the pairs follows a symmetric distribution around zero.

```
n = 20
# Buisness Volume time 0
bv0 = np.random.normal(loc=3, scale=.1, size=n)
# Buisness Volume time 1
bv1 = bv0 + 0.1 + np.random.normal(loc=0, scale=.1, size=n)

# create an outlier
bv1[0] -= 10

# Paired t-test
print(scipy.stats.ttest_rel(bv0, bv1))
```

(continues on next page)

(continued from previous page)

```
# Wilcoxon
print(scipy.stats.wilcoxon(bv0, bv1))
```

```
Ttest_relResult(statistic=0.7766377807752968, pvalue=0.44693401731548044)
WilcoxonResult(statistic=23.0, pvalue=0.001209259033203125)
```

Mann–Whitney U test (quantitative ~ categorial (2 levels))

In statistics, the Mann–Whitney U test (also called the Mann–Whitney–Wilcoxon, Wilcoxon rank-sum test or Wilcoxon–Mann–Whitney test) is a nonparametric test of the null hypothesis that two samples come from the same population against an alternative hypothesis, especially that a particular population tends to have larger values than the other.

It can be applied on unknown distributions contrary to e.g. a t-test that has to be applied only on normal distributions, and it is nearly as efficient as the t-test on normal distributions.

```
n = 20
# Buismess Volume group 0
bv0 = np.random.normal(loc=1, scale=.1, size=n)

# Buismess Volume group 1
bv1 = np.random.normal(loc=1.2, scale=.1, size=n)

# create an outlier
bv1[0] -= 10

# Two-samples t-test
print(scipy.stats.ttest_ind(bv0, bv1))

# Wilcoxon
print(scipy.stats.mannwhitneyu(bv0, bv1))
```

```
Ttest_indResult(statistic=0.6104564820307219, pvalue=0.5451934484051324)
MannwhitneyuResult(statistic=41.0, pvalue=9.037238869417781e-06)
```

4.1.11 Linear model

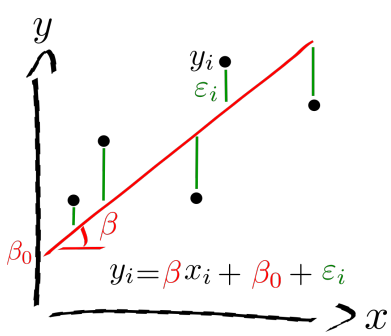


Fig. 3: Linear model

Given n random samples $(y_i, x_{1i}, \ldots, x_{pi})$, $i = 1, \ldots, n$, the linear regression models the relation between the observations y_i and the independent variables x_i^p is formulated as

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_p x_{pi} + \varepsilon_i \qquad i = 1, \ldots, n$$

- The β's are the model parameters, ie, the regression coeficients.
- β_0 is the intercept or the bias.
- ε_i are the **residuals**.
- **An independent variable (IV)**. It is a variable that stands alone and isn't changed by the other variables you are trying to measure. For example, someone's age might be an independent variable. Other factors (such as what they eat, how much they go to school, how much television they watch) aren't going to change a person's age. In fact, when you are looking for some kind of relationship between variables you are trying to see if the independent variable causes some kind of change in the other variables, or dependent variables. In Machine Learning, these variables are also called the **predictors**.
- A **dependent variable**. It is something that depends on other factors. For example, a test score could be a dependent variable because it could change depending on several factors such as how much you studied, how much sleep you got the night before you took the test, or even how hungry you were when you took it. Usually when you are looking for a relationship between two things you are trying to find out what makes the dependent variable change the way it does. In Machine Learning this variable is called a **target variable**.

Assumptions

1. Independence of residuals (ε_i). This assumptions **must** be satisfied
2. Normality of residuals (ε_i). Approximately normally distributed can be accepted.

Regression diagnostics: testing the assumptions of linear regression

Simple regression: test association between two quantitative variables

Using the dataset "salary", explore the association between the dependant variable (e.g. Salary) and the independent variable (e.g.: Experience is quantitative), considering only non-managers.

```
df = salary[salary.management == 'N']
```

1. Model the data

Model the data on some **hypothesis** e.g.: salary is a linear function of the experience.

$$\text{salary}_i = \beta_0 + \beta \text{ experience}_i + \epsilon_i,$$

more generally

$$y_i = \beta_0 + \beta\, x_i + \epsilon_i$$

This can be rewritten in the matrix form using the design matrix made of values of independant variable and the intercept:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ 1 & x_4 \\ 1 & x_5 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{bmatrix}$$

- β: the slope or coefficient or parameter of the model,
- β_0: the **intercept** or **bias** is the second parameter of the model,
- ϵ_i: is the ith error, or residual with $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The simple regression is equivalent to the Pearson correlation.

2. Fit: estimate the model parameters

The goal it so estimate β, β_0 and σ^2.

Minimizes the **mean squared error (MSE)** or the **Sum squared error (SSE)**. The so-called **Ordinary Least Squares (OLS)** finds β, β_0 that minimizes the $SSE = \sum_i \epsilon_i^2$

$$SSE = \sum_i (y_i - \beta\ x_i - \beta_0)^2$$

Recall from calculus that an extreme point can be found by computing where the derivative is zero, i.e. to find the intercept, we perform the steps:

$$\begin{aligned} \frac{\partial SSE}{\partial \beta_0} = \sum_i (y_i - \beta\ x_i - \beta_0) &= 0 \\ \sum_i y_i &= \beta\ \sum_i x_i + n\ \beta_0 \\ n\ \bar{y} &= n\ \beta\ \bar{x} + n\ \beta_0 \\ \beta_0 &= \bar{y} - \beta\ \bar{x} \end{aligned}$$

To find the regression coefficient, we perform the steps:

$$\frac{\partial SSE}{\partial \beta} = \sum_i x_i (y_i - \beta\ x_i - \beta_0) = 0$$

Plug in β_0:

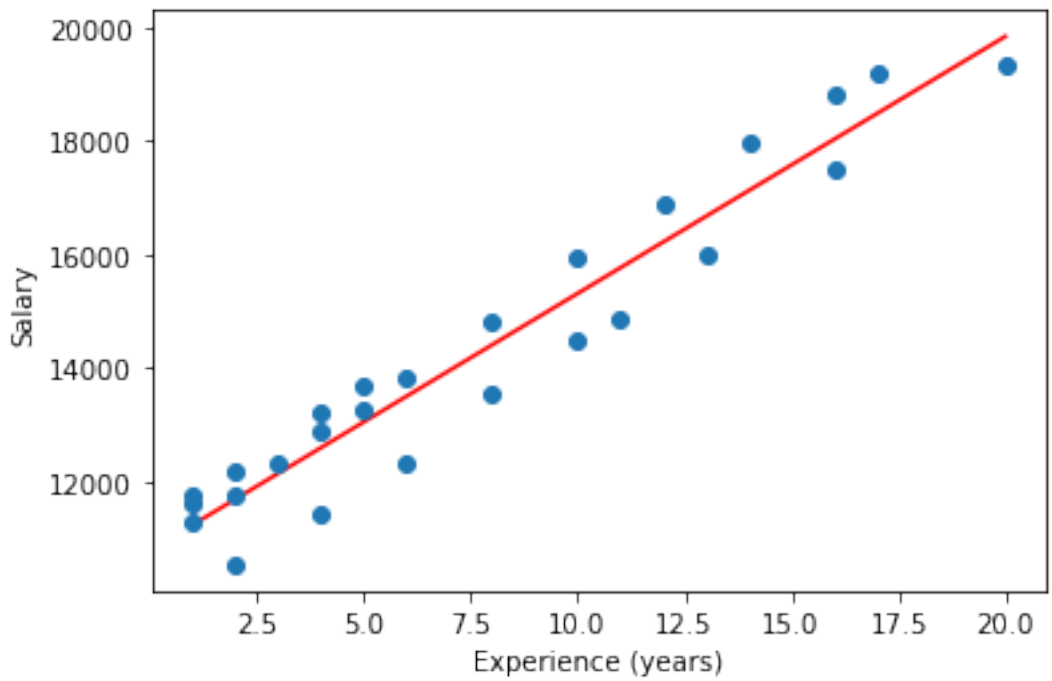
$$\begin{aligned} \sum_i x_i (y_i - \beta\ x_i - \bar{y} + \beta \bar{x}) &= 0 \\ \sum_i x_i y_i - \bar{y} \sum_i x_i &= \beta \sum_i (x_i - \bar{x}) \end{aligned}$$

Divide both sides by n:

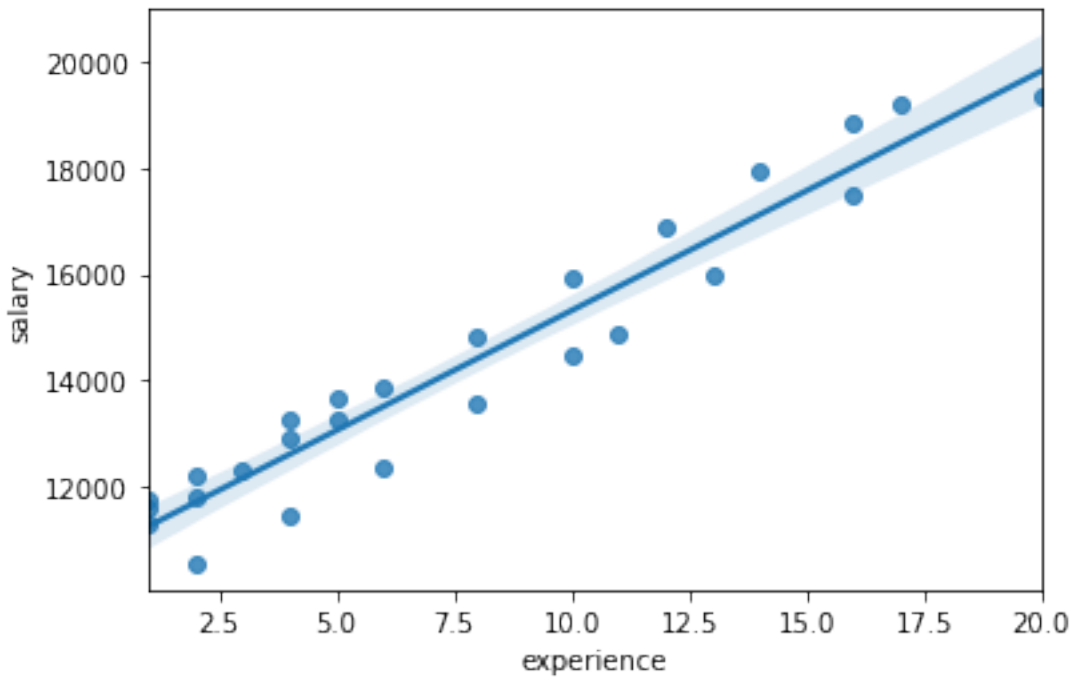
$$\begin{aligned} \frac{1}{n} \sum_i x_i y_i - \bar{y}\bar{x} &= \frac{1}{n} \beta \sum_i (x_i - \bar{x}) \\ \beta = \frac{\frac{1}{n} \sum_i x_i y_i - \bar{y}\bar{x}}{\frac{1}{n} \sum_i (x_i - \bar{x})} &= \frac{Cov(x, y)}{Var(x)}. \end{aligned}$$

```
y, x = df.salary, df.experience
beta, beta0, r_value, p_value, std_err = scipy.stats.linregress(x,y)
print("y = %f x + %f,  r: %f, r-squared: %f,\np-value: %f, std_err: %f"
      % (beta, beta0, r_value, r_value**2, p_value, std_err))

print("Regression line with the scatterplot")
yhat = beta * x  +  beta0 # regression line
plt.plot(x, yhat, 'r-', x, y,'o')
plt.xlabel('Experience (years)')
plt.ylabel('Salary')
plt.show()

print("Using seaborn")
ax = sns.regplot(x="experience", y="salary", data=df)
```

```
y = 452.658228 x + 10785.911392,  r: 0.965370, r-squared: 0.931939,
p-value: 0.000000, std_err: 24.970021
Regression line with the scatterplot
```




```
Using seaborn
```



Multiple regression

Theory

Muliple Linear Regression is the most basic supervised learning algorithm.

Given: a set of training data $\{x_1, ..., x_N\}$ with corresponding targets $\{y_1, ..., y_N\}$.

In linear regression, we assume that the model that generates the data involves only a linear combination of the input variables, i.e.

$$y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_P x_{iP} + \varepsilon_i,$$

or, simplified

$$y_i = \beta_0 + \sum_{j=1}^{P-1} \beta_j x_i^j + \varepsilon_i.$$

Extending each sample with an intercept, $x_i := [1, x_i] \in R^{P+1}$ allows us to use a more general notation based on linear algebra and write it as a simple dot product:

$$y_i = \mathbf{x}_i^T \beta + \varepsilon_i,$$

where $\beta \in R^{P+1}$ is a vector of weights that define the $P + 1$ parameters of the model. From now we have P regressors + the intercept.

Using the matrix notation:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1P} \\ 1 & x_{21} & \dots & x_{2P} \\ 1 & x_{31} & \dots & x_{3P} \\ 1 & x_{41} & \dots & x_{4P} \\ 1 & x_5 & \dots & x_5 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_P \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{bmatrix}$$

Let $X = [x_0^T, ..., x_N^T]$ be the $(N \times P + 1)$ **design matrix** of N samples of P input features with one column of one and let be $y = [y_1, ..., y_N]$ be a vector of the N targets.

$$y = X\beta + \varepsilon$$

Minimize the Mean Squared Error MSE loss:

$$MSE(\beta) == \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{x}_i^T\beta)^2$$

Using the matrix notation, the **mean squared error (MSE) loss can be rewritten**:

$$MSE(\beta) = \frac{1}{N}||y - X\beta||_2^2.$$

The β that minimises the MSE can be found by:

$$\nabla_\beta\left(\frac{1}{N}||y - X\beta||_2^2\right) = 0 \tag{4.30}$$

$$\frac{1}{N}\nabla_\beta(y - X\beta)^T(y - X\beta) = 0 \tag{4.31}$$

$$\frac{1}{N}\nabla_\beta(y^Ty - 2\beta^TX^Ty + \beta^TX^TX\beta) = 0 \tag{4.32}$$

$$-2X^Ty + 2X^TX\beta = 0 \tag{4.33}$$

$$X^TX\beta = X^Ty \tag{4.34}$$

$$\beta = (X^TX)^{-1}X^Ty, \tag{4.35}$$

where $(X^TX)^{-1}X^T$ is a pseudo inverse of X.

Simulated dataset where:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_{50} \end{bmatrix} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & x_{1,3} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{50,1} & x_{50,2} & x_{50,3} \end{bmatrix} \begin{bmatrix} 10 \\ 1 \\ 0.5 \\ 0.1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_{50} \end{bmatrix}$$

```
from scipy import linalg
np.random.seed(seed=42)  # make the example reproducible

# Dataset
N, P = 50, 4
X = np.random.normal(size= N * P).reshape((N, P))
## Our model needs an intercept so we add a column of 1s:
X[:, 0] = 1
print(X[:5, :])

betastar = np.array([10, 1., .5, 0.1])
e = np.random.normal(size=N)
y = np.dot(X, betastar) + e
```

```
[[ 1.         -0.1382643   0.64768854  1.52302986]
 [ 1.         -0.23413696  1.57921282  0.76743473]
 [ 1.          0.54256004 -0.46341769 -0.46572975]
 [ 1.         -1.91328024 -1.72491783 -0.56228753]
 [ 1.          0.31424733 -0.90802408 -1.4123037 ]]
```

Fit with numpy

Estimate the parameters

```
Xpinv = linalg.pinv2(X)
betahat = np.dot(Xpinv, y)
print("Estimated beta:\n", betahat)
```

```
Estimated beta:
 [10.14742501  0.57938106  0.51654653  0.17862194]
```

4.1.12 Linear model with statsmodels

Sources: http://statsmodels.sourceforge.net/devel/examples/

Multiple regression

Interface with statsmodels without formulae (sm)

```
## Fit and summary:
model = sm.OLS(y, X).fit()
print(model.summary())

# prediction of new values
ypred = model.predict(X)

# residuals + prediction == true values
assert np.all(ypred + model.resid == y)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.363
Model:                            OLS   Adj. R-squared:                  0.322
Method:                 Least Squares   F-statistic:                     8.748
Date:                Fri, 08 Jan 2021   Prob (F-statistic):           0.000106
Time:                        15:05:47   Log-Likelihood:                -71.271
No. Observations:                  50   AIC:                             150.5
Df Residuals:                      46   BIC:                             158.2
Df Model:                           3
Covariance Type:            nonrobust
```

(continues on next page)

(continued from previous page)

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         10.1474      0.150     67.520      0.000       9.845      10.450
x1             0.5794      0.160      3.623      0.001       0.258       0.901
x2             0.5165      0.151      3.425      0.001       0.213       0.820
x3             0.1786      0.144      1.240      0.221      -0.111       0.469
==============================================================================
Omnibus:                        2.493   Durbin-Watson:                   2.369
Prob(Omnibus):                  0.288   Jarque-Bera (JB):                1.544
Skew:                           0.330   Prob(JB):                        0.462
Kurtosis:                       3.554   Cond. No.                         1.27
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Statsmodels with Pandas using formulae (`smf`)

Use R language syntax for data.frame. For an additive model: $y_i = \beta^0 + x_i^1\beta^1 + x_i^2\beta^2 + \epsilon_i \equiv$ `y ~ x1 + x2`.

```
df = pd.DataFrame(np.column_stack([X, y]), columns=['inter', 'x1','x2', 'x3', 'y
↪'])
print(df.columns, df.shape)
# Build a model excluding the intercept, it is implicit
model = smf.ols("y~x1 + x2 + x3", df).fit()
print(model.summary())
```

```
Index(['inter', 'x1', 'x2', 'x3', 'y'], dtype='object') (50, 5)
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.363
Model:                            OLS   Adj. R-squared:                  0.322
Method:                 Least Squares   F-statistic:                     8.748
Date:                Fri, 08 Jan 2021   Prob (F-statistic):           0.000106
Time:                        15:05:47   Log-Likelihood:                -71.271
No. Observations:                  50   AIC:                             150.5
Df Residuals:                      46   BIC:                             158.2
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     10.1474      0.150     67.520      0.000       9.845      10.450
x1             0.5794      0.160      3.623      0.001       0.258       0.901
x2             0.5165      0.151      3.425      0.001       0.213       0.820
```

(continues on next page)

(continued from previous page)

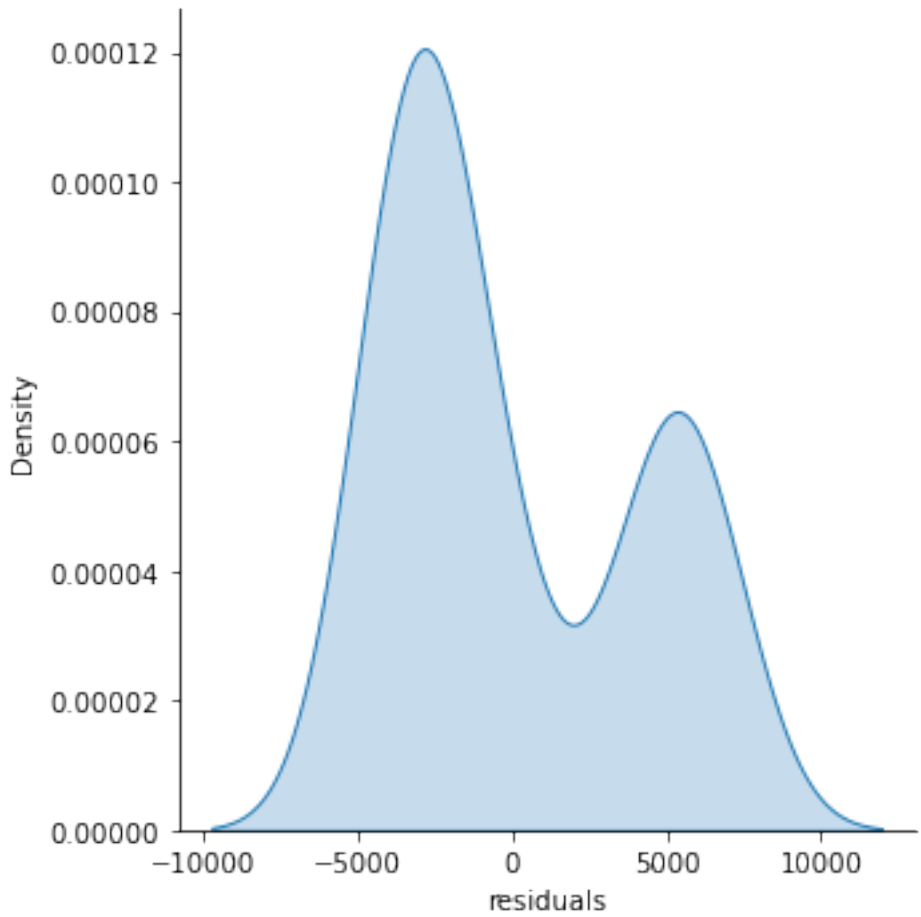
```
x3             0.1786      0.144      1.240      0.221     -0.111      0.469
==============================================================================
Omnibus:                        2.493   Durbin-Watson:                   2.369
Prob(Omnibus):                  0.288   Jarque-Bera (JB):                1.544
Skew:                           0.330   Prob(JB):                        0.462
Kurtosis:                       3.554   Cond. No.                         1.27
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

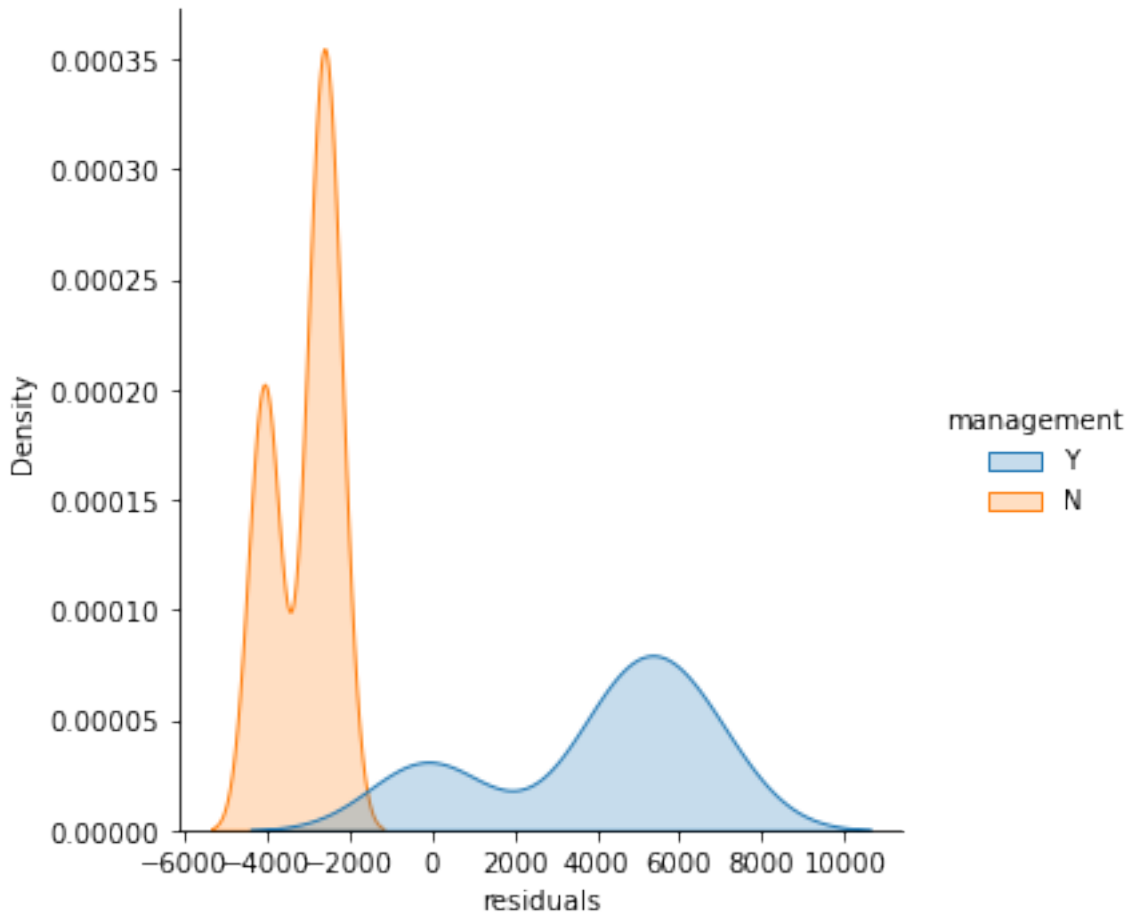
Multiple regression with categorical independent variables or factors: Analysis of covariance (ANCOVA)

Analysis of covariance (ANCOVA) is a linear model that blends ANOVA and linear regression. ANCOVA evaluates whether population means of a dependent variable (DV) are equal across levels of a categorical independent variable (IV) often called a treatment, while statistically controlling for the effects of other quantitative or continuous variables that are not of primary interest, known as covariates (CV).

```
df = salary.copy()

lm = smf.ols('salary ~ experience', df).fit()
df["residuals"] = lm.resid

print("Jarque-Bera normality test p-value %.5f" % \
      sm.stats.jarque_bera(lm.resid)[1])

ax = sns.displot(df, x='residuals', kind="kde", fill=True)
ax = sns.displot(df, x='residuals', kind="kde", hue='management', fill=True)
```

```
Jarque-Bera normality test p-value 0.04374
```


0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Density
−10000
−5000
0
5000
10000
residuals



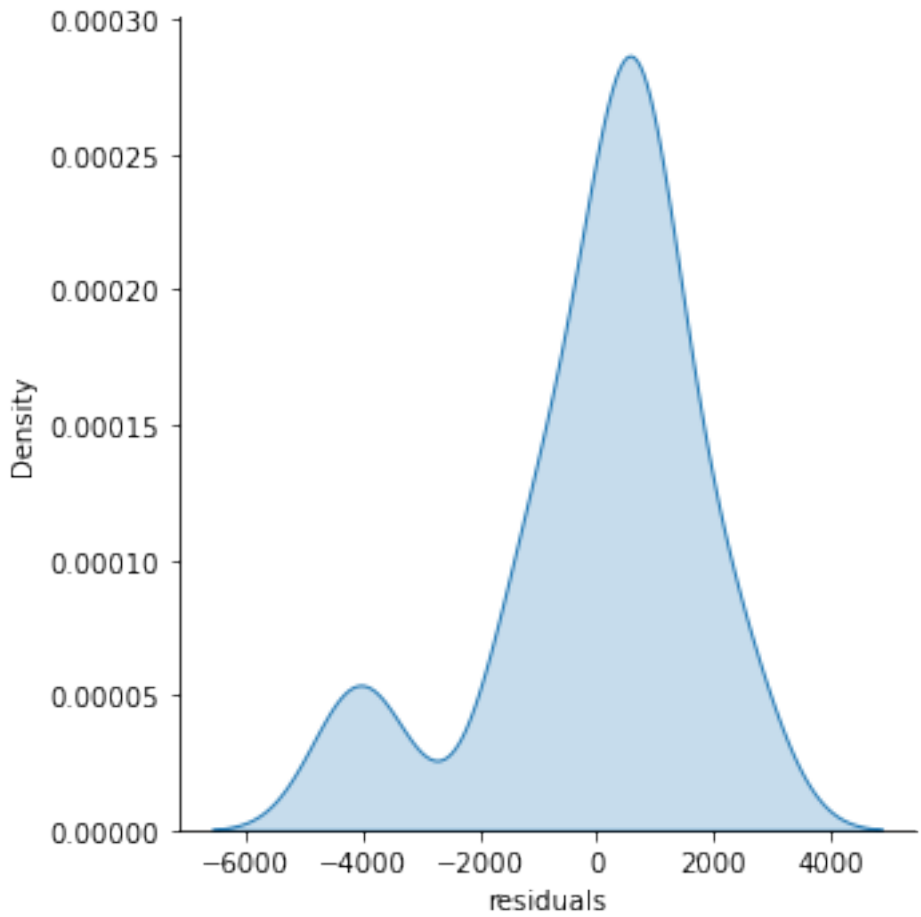
Normality assumption of the residuals can be rejected (p-value < 0.05). There is an efect of the "management" factor, take it into account.

One-way AN(C)OVA

- ANOVA: one categorical independent variable, i.e. one factor.
- ANCOVA: ANOVA with some covariates.

```
oneway = smf.ols('salary ~ management + experience', df).fit()
df["residuals"] = oneway.resid
sns.displot(df, x='residuals', kind="kde", fill=True)
print(sm.stats.anova_lm(oneway, typ=2))
print("Jarque-Bera normality test p-value %.3f" %  \
      sm.stats.jarque_bera(oneway.resid)[1])
```

```
                  sum_sq    df           F        PR(>F)
management  5.755739e+08   1.0  183.593466  4.054116e-17
experience  3.334992e+08   1.0  106.377768  3.349662e-13
Residual    1.348070e+08  43.0         NaN           NaN
Jarque-Bera normality test p-value 0.004
```



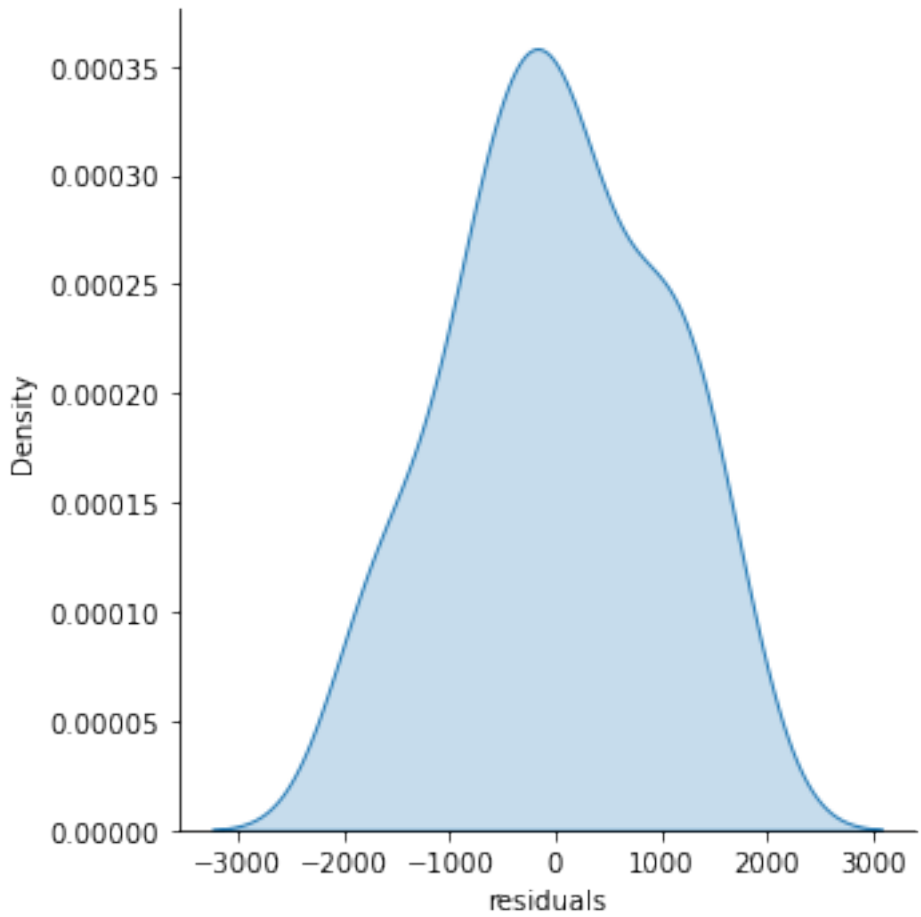
Distribution of residuals is still not normal but closer to normality. Both management and experience are significantly associated with salary.

Two-way AN(C)OVA

Ancova with two categorical independent variables, i.e. two factors.

```
twoway = smf.ols('salary ~ education + management + experience', df).fit()

df["residuals"] = twoway.resid
sns.displot(df, x='residuals', kind="kde", fill=True)
print(sm.stats.anova_lm(twoway, typ=2))

print("Jarque-Bera normality test p-value %.3f" %  \
      sm.stats.jarque_bera(twoway.resid)[1])
```

```
                  sum_sq    df           F        PR(>F)
education   9.152624e+07   2.0   43.351589  7.672450e-11
management  5.075724e+08   1.0  480.825394  2.901444e-24
experience  3.380979e+08   1.0  320.281524  5.546313e-21
Residual    4.328072e+07  41.0         NaN           NaN
Jarque-Bera normality test p-value 0.506
```



Normality assumtion cannot be rejected. Assume it. Education, management and experience are significantly associated with salary.

Comparing two nested models

`oneway` is nested within `twoway`. Comparing two nested models tells us if the additional predictors (i.e. `education`) of the full model significantly decrease the residuals. Such comparison can be done using an F-test on residuals:

```
print(twoway.compare_f_test(oneway))  # return F, pval, df
```

```
(43.35158945918107, 7.672449570495418e-11, 2.0)
```

twoway is significantly better than one way

Factor coding

See http://statsmodels.sourceforge.net/devel/contrasts.html

By default Pandas use "dummy coding". Explore:

```
print(twoway.model.data.param_names)
print(twoway.model.data.exog[:10, :])
```

```
['Intercept', 'education[T.Master]', 'education[T.Ph.D]', 'management[T.Y]',
↪'experience']
[[1. 0. 0. 1. 1.]
 [1. 0. 1. 0. 1.]
 [1. 0. 1. 1. 1.]
 [1. 1. 0. 0. 1.]
 [1. 0. 1. 0. 1.]
 [1. 1. 0. 1. 2.]
 [1. 1. 0. 0. 2.]
 [1. 0. 0. 0. 2.]
 [1. 0. 1. 0. 2.]
 [1. 1. 0. 0. 3.]]
```

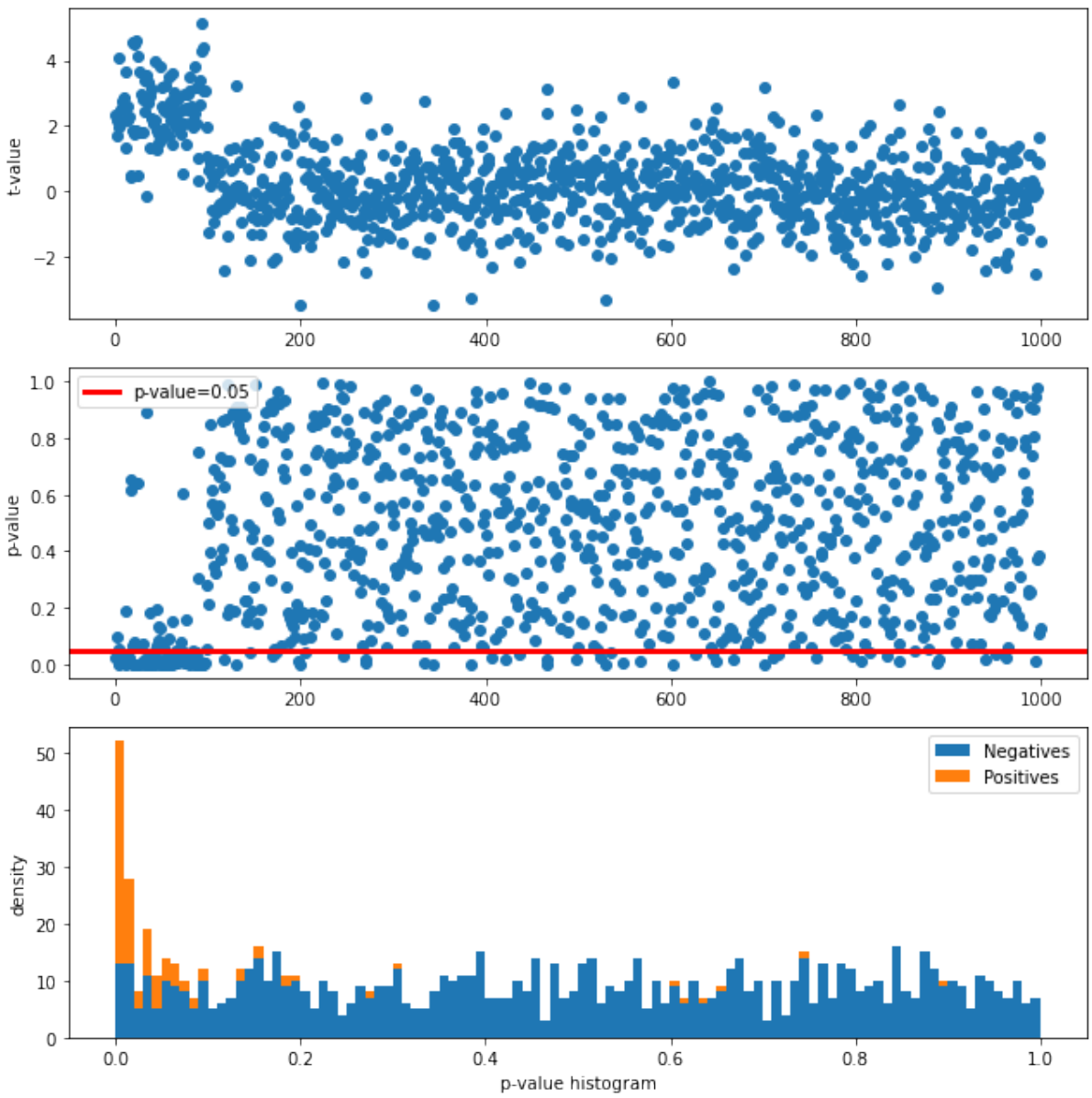
Contrasts and post-hoc tests

```
# t-test of the specific contribution of experience:
ttest_exp = twoway.t_test([0, 0, 0, 0, 1])
ttest_exp.pvalue, ttest_exp.tvalue
print(ttest_exp)

# Alternatively, you can specify the hypothesis tests using a string
twoway.t_test('experience')

# Post-hoc is salary of Master different salary of Ph.D?
# ie. t-test salary of Master = salary of Ph.D.
print(twoway.t_test('education[T.Master] = education[T.Ph.D]'))
```

```
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0           546.1840     30.519     17.896      0.000     484.549     607.819
==============================================================================
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0           147.8249    387.659      0.381      0.705    -635.069     930.719
==============================================================================
```

4.1.13 Multiple comparisons

```
np.random.seed(seed=42)  # make example reproducible

# Dataset
n_samples, n_features = 100, 1000
n_info = int(n_features/10)  # number of features with information
n1, n2 = int(n_samples/2), n_samples - int(n_samples/2)
snr = .5
Y = np.random.randn(n_samples, n_features)
grp = np.array(["g1"] * n1 + ["g2"] * n2)

# Add some group effect for Pinfo features
Y[grp=="g1", :n_info] += snr

#
import scipy.stats as stats
import matplotlib.pyplot as plt
tvals, pvals = np.full(n_features, np.NAN), np.full(n_features, np.NAN)
for j in range(n_features):
    tvals[j], pvals[j] = stats.ttest_ind(Y[grp=="g1", j], Y[grp=="g2", j],
                                         equal_var=True)

fig, axis = plt.subplots(3, 1, figsize=(9, 9))#, sharex='col')

axis[0].plot(range(n_features), tvals, 'o')
axis[0].set_ylabel("t-value")

axis[1].plot(range(n_features), pvals, 'o')
axis[1].axhline(y=0.05, color='red', linewidth=3, label="p-value=0.05")
#axis[1].axhline(y=0.05, label="toto", color='red')
axis[1].set_ylabel("p-value")
axis[1].legend()

axis[2].hist([pvals[n_info:], pvals[:n_info]],
    stacked=True, bins=100, label=["Negatives", "Positives"])
axis[2].set_xlabel("p-value histogram")
axis[2].set_ylabel("density")
axis[2].legend()

plt.tight_layout()
```



Note that under the null hypothesis the distribution of the p-values is uniform.

Statistical measures:

- **True Positive (TP)** equivalent to a hit. The test correctly concludes the presence of an effect.
- True Negative (TN). The test correctly concludes the absence of an effect.
- **False Positive (FP)** equivalent to a false alarm, **Type I error**. The test improperly concludes the presence of an effect. Thresholding at p-value < 0.05 leads to 47 FP.
- False Negative (FN) equivalent to a miss, Type II error. The test improperly concludes the absence of an effect.

```
P, N = n_info, n_features - n_info  # Positives, Negatives
TP = np.sum(pvals[:n_info ] < 0.05)  # True Positives
FP = np.sum(pvals[n_info: ] < 0.05)  # False Positives
print("No correction, FP: %i (expected: %.2f), TP: %i" % (FP, N * 0.05, TP))
```

```
No correction, FP: 47 (expected: 45.00), TP: 71
```

Bonferroni correction for multiple comparisons

The Bonferroni correction is based on the idea that if an experimenter is testing P hypotheses, then one way of maintaining the familywise error rate (FWER) is to test each individual hypothesis at a statistical significance level of $1/P$ times the desired maximum overall level.

So, if the desired significance level for the whole family of tests is α (usually 0.05), then the Bonferroni correction would test each individual hypothesis at a significance level of α/P. For example, if a trial is testing $P = 8$ hypotheses with a desired $\alpha = 0.05$, then the Bonferroni correction would test each individual hypothesis at $\alpha = 0.05/8 = 0.00625$.

```
import statsmodels.sandbox.stats.multicomp as multicomp

_, pvals_fwer, _, _  = multicomp.multipletests(pvals, alpha=0.05,
                                               method='bonferroni')
TP = np.sum(pvals_fwer[:n_info ] < 0.05)  # True Positives
FP = np.sum(pvals_fwer[n_info: ] < 0.05)  # False Positives
print("FWER correction, FP: %i, TP: %i" % (FP, TP))
```

```
FWER correction, FP: 0, TP: 6
```

The False discovery rate (FDR) correction for multiple comparisons

FDR-controlling procedures are designed to control the expected proportion of rejected null hypotheses that were incorrect rejections ("false discoveries"). FDR-controlling procedures provide less stringent control of Type I errors compared to the familywise error rate (FWER) controlling procedures (such as the Bonferroni correction), which control the probability of at least one Type I error. Thus, FDR-controlling procedures have greater power, at the cost of increased rates of Type I errors.

```
_, pvals_fdr, _, _  = multicomp.multipletests(pvals, alpha=0.05,
                                              method='fdr_bh')
TP = np.sum(pvals_fdr[:n_info ] < 0.05)  # True Positives
FP = np.sum(pvals_fdr[n_info: ] < 0.05)  # False Positives

print("FDR correction, FP: %i, TP: %i" % (FP, TP))
```

```
FDR correction, FP: 3, TP: 20
```

4.2 Lab: Brain volumes study

The study provides the brain volumes of grey matter (gm), white matter (wm) and cerebrospinal fluid) (csf) of 808 anatomical MRI scans.

4.2.1 Manipulate data

Set the working directory within a directory called "brainvol"

Create 2 subdirectories: *data* that will contain downloaded data and *reports* for results of the analysis.

```
import os
import os.path
import pandas as pd
import tempfile
import urllib.request

WD = os.path.join(tempfile.gettempdir(), "brainvol")
os.makedirs(WD, exist_ok=True)
#os.chdir(WD)

# use cookiecutter file organization
# https://drivendata.github.io/cookiecutter-data-science/
os.makedirs(os.path.join(WD, "data"), exist_ok=True)
#os.makedirs("reports", exist_ok=True)
```

Fetch data

- Demographic data *demo.csv* (columns: *participant_id*, *site*, *group*, *age*, *sex*) and tissue volume data: *group* is Control or Patient. *site* is the recruiting site.
- Gray matter volume *gm.csv* (columns: *participant_id*, *session*, *gm_vol*)
- White matter volume *wm.csv* (columns: *participant_id*, *session*, *wm_vol*)
- Cerebrospinal Fluid *csf.csv* (columns: *participant_id*, *session*, *csf_vol*)

```
base_url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/brain_
↪volumes/%s'
data = dict()
for file in ["demo.csv", "gm.csv", "wm.csv", "csf.csv"]:
    urllib.request.urlretrieve(base_url % file, os.path.join(WD, "data", file))

# Read all CSV in one line
# dicts = {k: pd.read_csv(os.path.join(WD, "data", "%s.csv" % k))
#          for k in ["demo", "gm", "wm", "csf"]}

demo = pd.read_csv(os.path.join(WD, "data", "demo.csv"))
gm = pd.read_csv(os.path.join(WD, "data", "gm.csv"))
wm = pd.read_csv(os.path.join(WD, "data", "wm.csv"))
csf = pd.read_csv(os.path.join(WD, "data", "csf.csv"))
```

(continues on next page)

(continued from previous page)

```
print("tables can be merge using shared columns")
print(gm.head())
```

Out:

```
tables can be merge using shared columns
  participant_id session  gm_vol
0    sub-S1-0002  ses-01    0.67
1    sub-S1-0002  ses-02    0.68
2    sub-S1-0002  ses-03    0.67
3    sub-S1-0004  ses-01    0.89
4    sub-S1-0004  ses-02    0.88
```

Merge tables according to *participant_id*

```
brain_vol = pd.merge(pd.merge(pd.merge(demo, gm), wm), csf)
assert brain_vol.shape == (808, 9)
```

Drop rows with missing values

```
brain_vol = brain_vol.dropna()
assert brain_vol.shape == (766, 9)
```

Compute Total Intra-cranial volume $tiv_vol = gm_vol + csf_vol + wm_vol$.

```
brain_vol["tiv_vol"] = brain_vol["gm_vol"] + brain_vol["wm_vol"] + brain_vol["csf_
↪vol"]
```

Compute tissue fractions $gm_f = gm_vol / tiv_vol$, $wm_f = wm_vol / tiv_vol$.

```
brain_vol["gm_f"] = brain_vol["gm_vol"] / brain_vol["tiv_vol"]
brain_vol["wm_f"] = brain_vol["wm_vol"] / brain_vol["tiv_vol"]
```

Save in a excel file *brain_vol.xlsx*

```
brain_vol.to_excel(os.path.join(WD, "data", "brain_vol.xlsx"),
                   sheet_name='data', index=False)
```

4.2.2 Descriptive Statistics

Load excel file *brain_vol.xlsx*

```
import os
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smfrmla
import statsmodels.api as sm

brain_vol = pd.read_excel(os.path.join(WD, "data", "brain_vol.xlsx"),
```

(continues on next page)

(continued from previous page)

```
                            sheet_name='data')
# Round float at 2 decimals when printing
pd.options.display.float_format = '{:,.2f}'.format
```

Descriptive statistics Most of participants have several MRI sessions (column *session*) Select on rows from session one "ses-01"

```
brain_vol1 = brain_vol[brain_vol.session == "ses-01"]
# Check that there are no duplicates
assert len(brain_vol1.participant_id.unique()) == len(brain_vol1.participant_id)
```

Global descriptives statistics of numerical variables

```
desc_glob_num = brain_vol1.describe()
print(desc_glob_num)
```

Out:

```
         age  gm_vol  wm_vol  csf_vol  tiv_vol   gm_f   wm_f
count 244.00  244.00  244.00   244.00   244.00 244.00 244.00
mean   34.54    0.71    0.44     0.31     1.46   0.49   0.30
std    12.09    0.08    0.07     0.08     0.17   0.04   0.03
min    18.00    0.48    0.05     0.12     0.83   0.37   0.06
25%    25.00    0.66    0.40     0.25     1.34   0.46   0.28
50%    31.00    0.70    0.43     0.30     1.45   0.49   0.30
75%    44.00    0.77    0.48     0.37     1.57   0.52   0.31
max    61.00    1.03    0.62     0.63     2.06   0.60   0.36
```

Global Descriptive statistics of categorical variable

```
desc_glob_cat = brain_vol1[["site", "group", "sex"]].describe(include='all')
print(desc_glob_cat)

print("Get count by level")
desc_glob_cat = pd.DataFrame({col: brain_vol1[col].value_counts().to_dict()
                              for col in ["site", "group", "sex"]})
print(desc_glob_cat)
```

Out:

```
       site    group  sex
count   244      244  244
unique    7        2    2
top      S7  Patient    M
freq     65      157  155
Get count by level
         site  group    sex
S7      65.00    nan    nan
S5      62.00    nan    nan
S8      59.00    nan    nan
```

(continues on next page)

(continued from previous page)

```
S3       29.00    nan    nan
S4       15.00    nan    nan
S1       13.00    nan    nan
S6        1.00    nan    nan
Patient    nan 157.00    nan
Control    nan  87.00    nan
M          nan    nan 155.00
F          nan    nan  89.00
```

Remove the single participant from site 6

```
brain_vol = brain_vol[brain_vol.site != "S6"]
brain_vol1 = brain_vol[brain_vol.session == "ses-01"]
desc_glob_cat = pd.DataFrame({col: brain_vol1[col].value_counts().to_dict()
                              for col in ["site", "group", "sex"]})
print(desc_glob_cat)
```

Out:

```
          site  group    sex
S7       65.00    nan    nan
S5       62.00    nan    nan
S8       59.00    nan    nan
S3       29.00    nan    nan
S4       15.00    nan    nan
S1       13.00    nan    nan
Patient    nan 157.00    nan
Control    nan  86.00    nan
M          nan    nan 155.00
F          nan    nan  88.00
```

Descriptives statistics of numerical variables per clinical status

```
desc_group_num = brain_vol1[["group", 'gm_vol']].groupby("group").describe()
print(desc_group_num)
```

Out:

```
        gm_vol
         count mean  std  min  25%  50%  75%  max
group
Control  86.00 0.72 0.09 0.48 0.66 0.71 0.78 1.03
Patient 157.00 0.70 0.08 0.53 0.65 0.70 0.76 0.90
```

4.2.3 Statistics

Objectives:

1. Site effect of gray matter atrophy
2. Test the association between the age and gray matter atrophy in the control and patient population independently.
3. Test for differences of atrophy between the patients and the controls
4. Test for interaction between age and clinical status, ie: is the brain atrophy process in patient population faster than in the control population.
5. The effect of the medication in the patient population.

```
import statsmodels.api as sm
import statsmodels.formula.api as smfrmla
import scipy.stats
import seaborn as sns
```

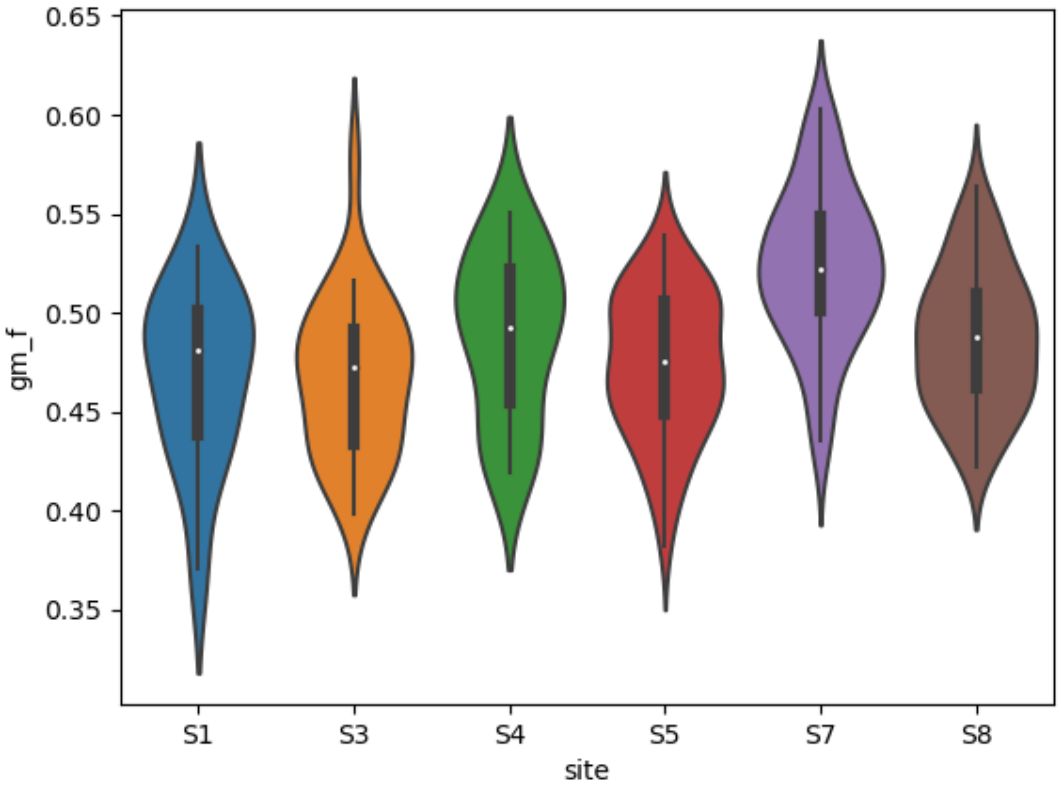
1 Site effect on Grey Matter atrophy

The model is Oneway Anova gm_f ~ site The ANOVA test has important assumptions that must be satisfied in order for the associated p-value to be valid.

- The samples are independent.
- Each sample is from a normally distributed population.
- The population standard deviations of the groups are all equal. This property is known as homoscedasticity.

Plot

```
sns.violinplot(x="site", y="gm_f", data=brain_vol1)
# sns.violinplot(x="site", y="wm_f", data=brain_vol1)
```



Out:

```
<AxesSubplot:xlabel='site', ylabel='gm_f'>
```

Stats with scipy

```
fstat, pval = scipy.stats.f_oneway(*[brain_vol1.gm_f[brain_vol1.site == s]
                                     for s in brain_vol1.site.unique()])
print("Oneway Anova gm_f ~ site F=%.2f, p-value=%E" % (fstat, pval))
```

Out:

```
Oneway Anova gm_f ~ site F=14.82, p-value=1.188136E-12
```

Stats with statsmodels

```
anova = smfrmla.ols("gm_f ~ site", data=brain_vol1).fit()
# print(anova.summary())
print("Site explains %.2f%% of the grey matter fraction variance" %
      (anova.rsquared * 100))

print(sm.stats.anova_lm(anova, typ=2))
```

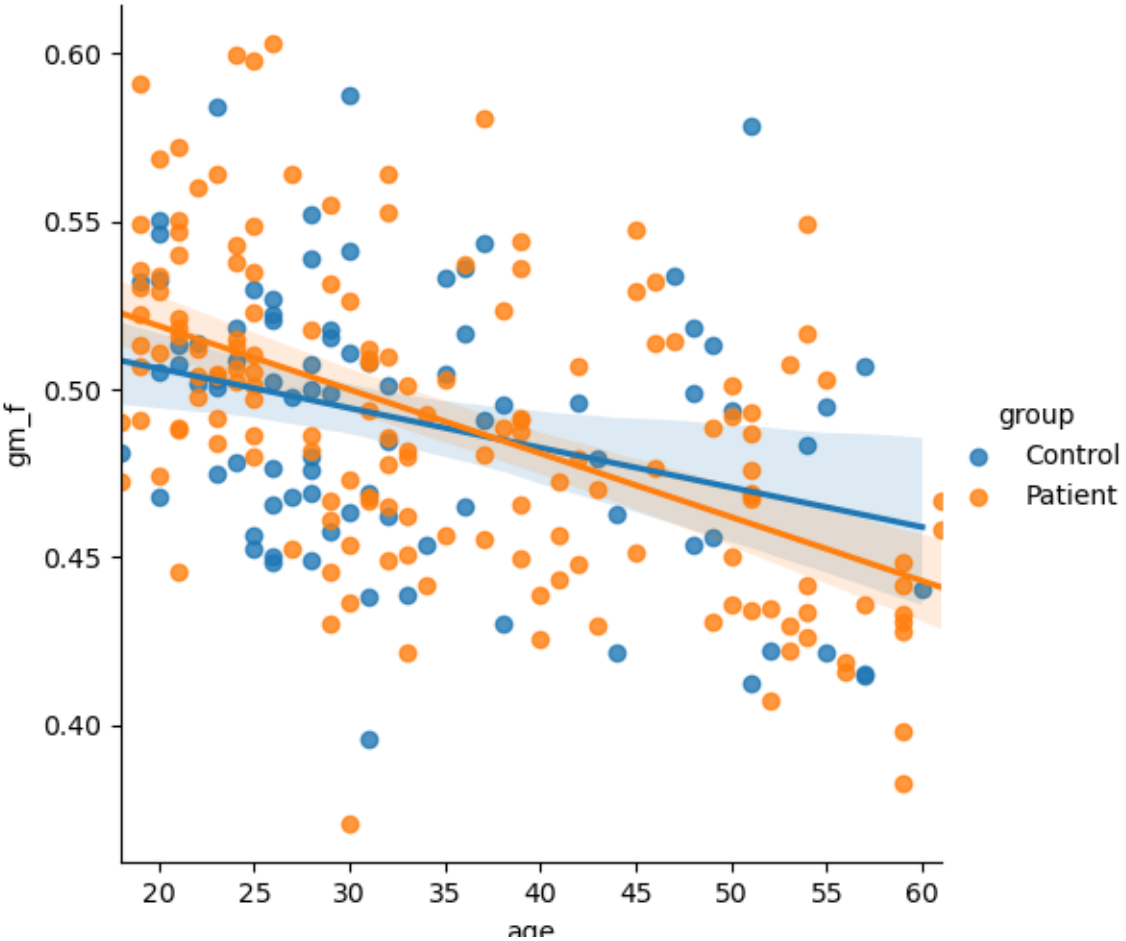
Out:

```
Site explains 23.82% of the grey matter fraction variance
          sum_sq     df     F  PR(>F)
site        0.11   5.00 14.82    0.00
Residual    0.35 237.00   nan     nan
```

2. Test the association between the age and gray matter atrophy in the control and patient population independently.

Plot

```
sns.lmplot(x="age", y="gm_f", hue="group", data=brain_vol1)

brain_vol1_ctl = brain_vol1[brain_vol1.group == "Control"]
brain_vol1_pat = brain_vol1[brain_vol1.group == "Patient"]
```



Stats with scipy

```
print("--- In control population ---")
beta, beta0, r_value, p_value, std_err = \
    scipy.stats.linregress(x=brain_vol1_ctl.age, y=brain_vol1_ctl.gm_f)

print("gm_f = %f * age + %f" % (beta, beta0))
print("Corr: %f, r-squared: %f, p-value: %f, std_err: %f"\
      % (r_value, r_value**2, p_value, std_err))
```

(continues on next page)

(continued from previous page)

```
print("--- In patient population ---")
beta, beta0, r_value, p_value, std_err = \
    scipy.stats.linregress(x=brain_vol1_pat.age, y=brain_vol1_pat.gm_f)

print("gm_f = %f * age + %f" % (beta, beta0))
print("Corr: %f, r-squared: %f, p-value: %f, std_err: %f"\
      % (r_value, r_value**2, p_value, std_err))

print("Decrease seems faster in patient than in control population")
```

Out:

```
--- In control population ---
gm_f = -0.001181 * age + 0.529829
Corr: -0.325122, r-squared: 0.105704, p-value: 0.002255, std_err: 0.000375
--- In patient population ---
gm_f = -0.001899 * age + 0.556886
Corr: -0.528765, r-squared: 0.279592, p-value: 0.000000, std_err: 0.000245
Decrease seems faster in patient than in control population
```

Stats with statsmodels

```
print("--- In control population ---")
lr = smfrmla.ols("gm_f ~ age", data=brain_vol1_ctl).fit()
print(lr.summary())
print("Age explains %.2f%% of the grey matter fraction variance" %
      (lr.rsquared * 100))

print("--- In patient population ---")
lr = smfrmla.ols("gm_f ~ age", data=brain_vol1_pat).fit()
print(lr.summary())
print("Age explains %.2f%% of the grey matter fraction variance" %
      (lr.rsquared * 100))
```

Out:

```
--- In control population ---
                            OLS Regression Results
==============================================================================
Dep. Variable:                   gm_f   R-squared:                       0.106
Model:                            OLS   Adj. R-squared:                  0.095
Method:                 Least Squares   F-statistic:                     9.929
Date:                ven., 08 janv. 2021   Prob (F-statistic):            0.00226
Time:                        16:12:34   Log-Likelihood:                 159.34
No. Observations:                  86   AIC:                            -314.7
Df Residuals:                      84   BIC:                            -309.8
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
```

(continues on next page)

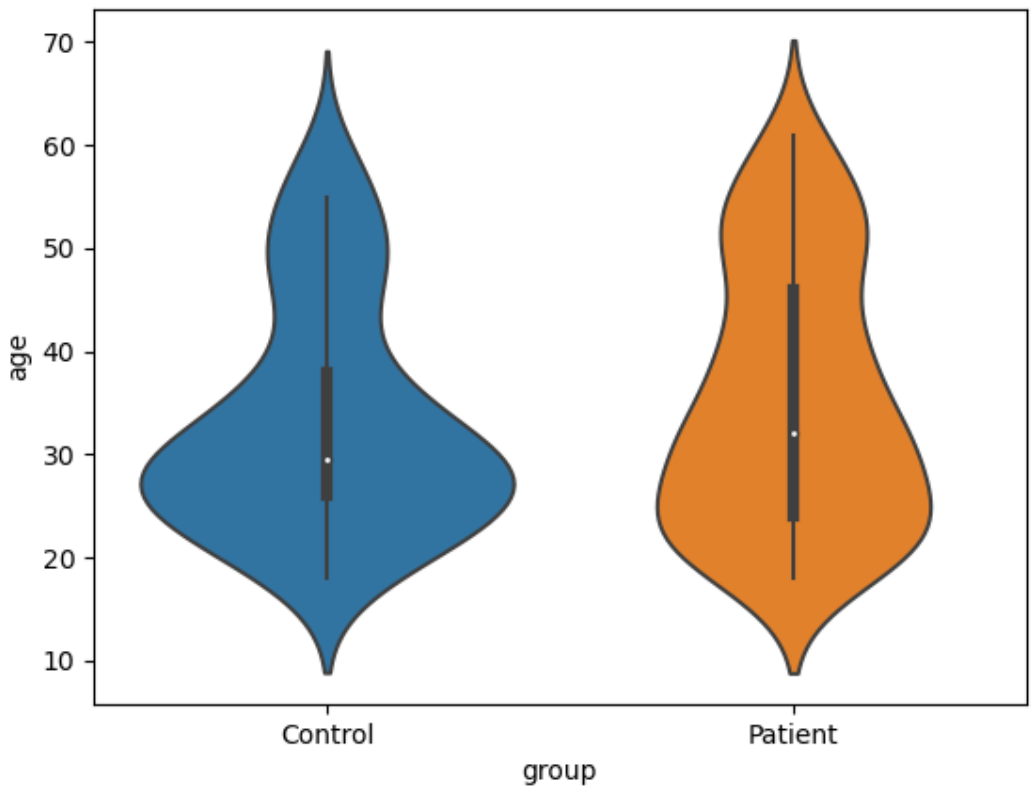
(continued from previous page)

```
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.5298      0.013     40.350      0.000       0.504       0.556
age           -0.0012      0.000     -3.151      0.002      -0.002      -0.000
==============================================================================
Omnibus:                        0.946   Durbin-Watson:                   1.628
Prob(Omnibus):                  0.623   Jarque-Bera (JB):                0.782
Skew:                           0.233   Prob(JB):                        0.676
Kurtosis:                       2.962   Cond. No.                         111.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
Age explains 10.57% of the grey matter fraction variance
--- In patient population ---
                            OLS Regression Results
==============================================================================
Dep. Variable:                   gm_f   R-squared:                       0.280
Model:                            OLS   Adj. R-squared:                  0.275
Method:                 Least Squares   F-statistic:                     60.16
Date:              ven., 08 janv. 2021   Prob (F-statistic):           1.09e-12
Time:                        16:12:34   Log-Likelihood:                 289.38
No. Observations:                 157   AIC:                            -574.8
Df Residuals:                     155   BIC:                            -568.7
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.5569      0.009     60.817      0.000       0.539       0.575
age           -0.0019      0.000     -7.756      0.000      -0.002      -0.001
==============================================================================
Omnibus:                        2.310   Durbin-Watson:                   1.325
Prob(Omnibus):                  0.315   Jarque-Bera (JB):                1.854
Skew:                           0.230   Prob(JB):                        0.396
Kurtosis:                       3.268   Cond. No.                         111.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
Age explains 27.96% of the grey matter fraction variance
```

Before testing for differences of atrophy between the patients ans the controls **Preliminary tests for age x group effect** (patients would be older or younger than Controls)

Plot

```
sns.violinplot(x="group", y="age", data=brain_vol1)
```



Out:

```
<AxesSubplot:xlabel='group', ylabel='age'>
```

Stats with scipy

```
print(scipy.stats.ttest_ind(brain_vol1_ctl.age, brain_vol1_pat.age))
```

Out:

```
Ttest_indResult(statistic=-1.2155557697674162, pvalue=0.225343592508479)
```

Stats with statsmodels

```
print(smfrmla.ols("age ~ group", data=brain_vol1).fit().summary())
print("No significant difference in age between patients and controls")
```

Out:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    age   R-squared:                       0.006
Model:                            OLS   Adj. R-squared:                  0.002
Method:                 Least Squares   F-statistic:                     1.478
Date:                ven., 08 janv. 2021   Prob (F-statistic):              0.225
Time:                        16:12:34   Log-Likelihood:                -949.69
```

(continues on next page)

(continued from previous page)

```
No. Observations:                 243   AIC:                             1903.
Df Residuals:                     241   BIC:                             1910.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                       coef    std err          t      P>|t|      [0.025      0.
↪975]
------------------------------------------------------------------------------
↪--
Intercept           33.2558      1.305     25.484      0.000      30.685      35.
↪826
group[T.Patient]     1.9735      1.624      1.216      0.225      -1.225       5.
↪172
==============================================================================
Omnibus:                       35.711   Durbin-Watson:                   2.096
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               20.726
Skew:                           0.569   Prob(JB):                     3.16e-05
Kurtosis:                       2.133   Cond. No.                         3.12
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
No significant difference in age between patients and controls
```

Preliminary tests for sex x group (more/less males in patients than in Controls)

```
crosstab = pd.crosstab(brain_vol1.sex, brain_vol1.group)
print("Obeserved contingency table")
print(crosstab)

chi2, pval, dof, expected = scipy.stats.chi2_contingency(crosstab)

print("Chi2 = %f, pval = %f" % (chi2, pval))
print("No significant difference in sex between patients and controls")
```

Out:

```
Obeserved contingency table
group  Control  Patient
sex
F           33       55
M           53      102
Chi2 = 0.143253, pval = 0.705068
No significant difference in sex between patients and controls
```

3. Test for differences of atrophy between the patients and the controls

```
print(sm.stats.anova_lm(smfrmla.ols("gm_f ~ group", data=brain_vol1).fit(),
                        typ=2))
```

(continues on next page)

(continued from previous page)

```
print("No significant difference in atrophy between patients and controls")
```

Out:

```
          sum_sq     df    F  PR(>F)
group       0.00   1.00 0.01    0.92
Residual    0.46 241.00  nan     nan
No significant difference in atrophy between patients and controls
```

This model is simplistic we should adjust for age and site

```
print(sm.stats.anova_lm(smfrmla.ols(
        "gm_f ~ group + age + site", data=brain_vol1).fit(), typ=2))
print("No significant difference in GM between patients and controls")
```

Out:

```
          sum_sq     df     F  PR(>F)
group       0.00   1.00  1.82    0.18
site        0.11   5.00 19.79    0.00
age         0.09   1.00 86.86    0.00
Residual    0.25 235.00   nan     nan
No significant difference in GM between patients and controls
```

Observe age effect

4. Test for interaction between age and clinical status, ie: is the brain atrophy process in patient population faster than in the control population.

```
ancova = smfrmla.ols("gm_f ~ group:age + age + site", data=brain_vol1).fit()
print(sm.stats.anova_lm(ancova, typ=2))

print("= Parameters =")
print(ancova.params)

print("%.3f%% of grey matter loss per year (almost %.1f%% per decade)" %
      (ancova.params.age * 100, ancova.params.age * 100 * 10))

print("grey matter loss in patients is accelerated by %.3f%% per decade" %
      (ancova.params['group[T.Patient]:age'] * 100 * 10))
```

Out:

```
           sum_sq     df     F  PR(>F)
site         0.11   5.00 20.28    0.00
age          0.10   1.00 89.37    0.00
group:age    0.00   1.00  3.28    0.07
Residual     0.25 235.00   nan     nan
= Parameters =
Intercept                0.52
site[T.S3]               0.01
```

(continues on next page)

(continued from previous page)

```
site[T.S4]              0.03
site[T.S5]              0.01
site[T.S7]              0.06
site[T.S8]              0.02
age                    -0.00
group[T.Patient]:age   -0.00
dtype: float64
-0.148% of grey matter loss per year (almost -1.5% per decade)
grey matter loss in patients is accelerated by -0.232% per decade
```

Total running time of the script: (0 minutes 3.007 seconds)

4.3 Linear Mixed Models

Acknowledgements: Firstly, it's right to pay thanks to the blogs and sources I have used in writing this tutorial. Many parts of the text are quoted from the brillant book from Brady T. West, Kathleen B. Welch and Andrzej T. Galecki, see [Brady et al. 2014] in the references section below.

4.3.1 Introduction

Quoted from [Brady et al. 2014]:A linear mixed model (LMM) is a parametric linear model for **clustered, longitudinal, or repeated-measures** data that quantifies the relationships between a continuous dependent variable and various predictor variables. An LMM may include both **fixed-effect** parameters associated with one or more continuous or categorical covariates and **random effects** associated with one or more random factors. The mix of fixed and random effects gives the linear mixed model its name. Whereas fixed-effect parameters describe the relationships of the covariates to the dependent variable for an entire population, random effects are specific to clusters or subjects within a population. LMM is closely related with hierarchical linear model (HLM).

Clustered/structured datasets

Quoted from [Bruin 2006]: Random effects, are used when there is non independence in the data, such as arises from a hierarchical structure with clustered data. For example, students could be sampled from within classrooms, or patients from within doctors. When there are multiple levels, such as patients seen by the same doctor, the variability in the outcome can be thought of as being either within group or between group. Patient level observations are not independent, as within a given doctor patients are more similar. Units sampled at the highest level (in our example, doctors) are independent.

The continuous outcome variables is **structured or clustered** into **units** within **observations are not independents**. Types of clustered data:

1. studies with clustered data, such as students in classrooms, or experimental designs with random blocks, such as batches of raw material for an industrial process
2. **longitudinal or repeated-measures** studies, in which subjects are measured repeatedly over time or under different conditions.

Mixed effects = fixed + random effects

Fixed effects may be associated with continuous covariates, such as weight, baseline test score, or socioeconomic status, which take on values from a continuous (or sometimes a multivalued ordinal) range, or with factors, such as gender or treatment group, which are categorical. Fixed effects are unknown constant parameters associated with either continuous covariates or the levels of categorical factors in an LMM. Estimation of these parameters in LMMs is generally of intrinsic interest, because they indicate the relationships of the covariates with the continuous outcome variable.

Random effect When the levels of a factor can be thought of as having been sampled from a sample space, such that each particular level is not of intrinsic interest (e.g., classrooms or clinics that are randomly sampled from a larger population of classrooms or clinics), the effects associated with the levels of those factors can be modeled as random effects in an LMM. In contrast to fixed effects, which are represented by constant parameters in an LMM, random effects are represented by (unobserved) random variables, which are usually assumed to follow a normal distribution.

4.3.2 Random intercept

The `score_parentedu_byclass` dataset measure a `score` obtained by 60 students, indexed by i, within 3 `classroom` (with different teacher), indexed by j, given the education level `edu` of their parents. We want to study the link between `score` and `edu`. Observations, `score` are strutured by the sampling of classroom, see Fig below. `score` from the same classroom are are not indendant from each other: they shifted upward or backward thanks to a classroom or teacher effect. There is an **intercept** for each classroom. But this effect is not known given a student (unlike the age or the sex), it is a consequence of a random sampling of the classrooms. It is called a **random intercept**.

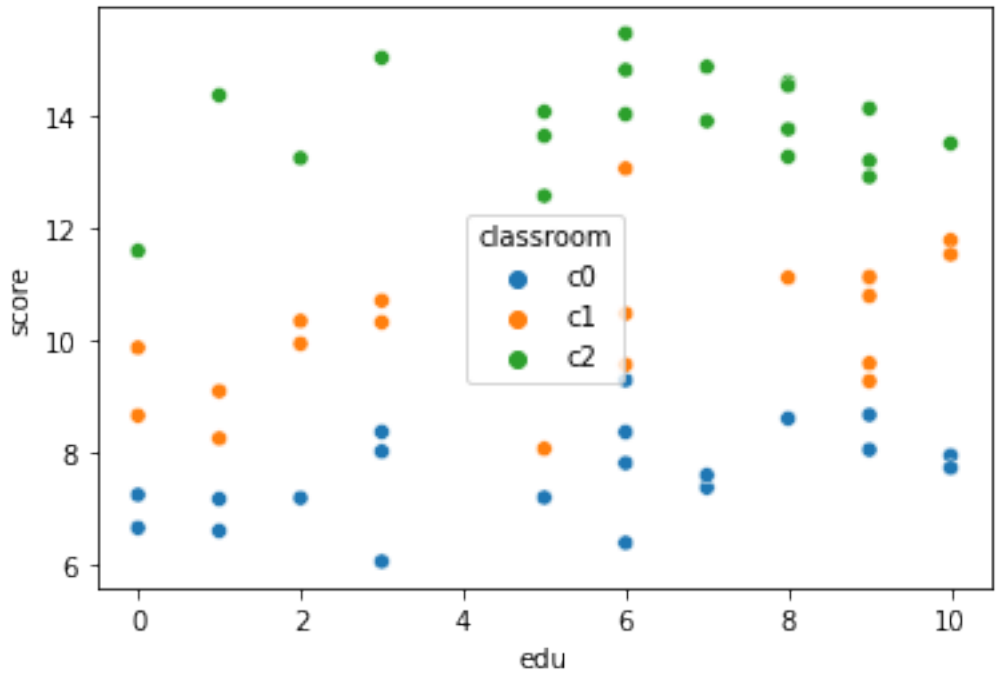
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

from stat_lmm_utils import rmse_coef_tstat_pval
from stat_lmm_utils import plot_lm_diagnosis
from stat_lmm_utils import plot_ancova_oneslope_grpintercept
from stat_lmm_utils import plot_lmm_oneslope_randintercept
from stat_lmm_utils import plot_ancova_fullmodel

results = pd.DataFrame(columns=["Model", "RMSE", "Coef", "Stat", "Pval"])

df = pd.read_csv('datasets/score_parentedu_byclass.csv')
print(df.head())
_ = sns.scatterplot(x="edu", y="score", hue="classroom", data=df)
```

```
  classroom  edu     score
0        c0    2  7.204352
```

(continues on next page)

(continued from previous page)

```
1         c0   10  7.963083
2         c0    3  8.383137
3         c0    5  7.213047
4         c0    6  8.379630
```



Global fixed effect

Global effect regresses the the independant variable $y =$ score on the dependant variable $x =$ edu without considering the any classroom effect. For each individual i the model is:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + \varepsilon_{ij},$$

where, β_0 is the global intercept, β_1 is the slope associated with edu and ε_{ij} is the random error at the individual level. Note that the classeroom, j index is not taken into account by the model.

The general R formula is: `y ~ x` which in this case is `score ~ edu`. This model is:

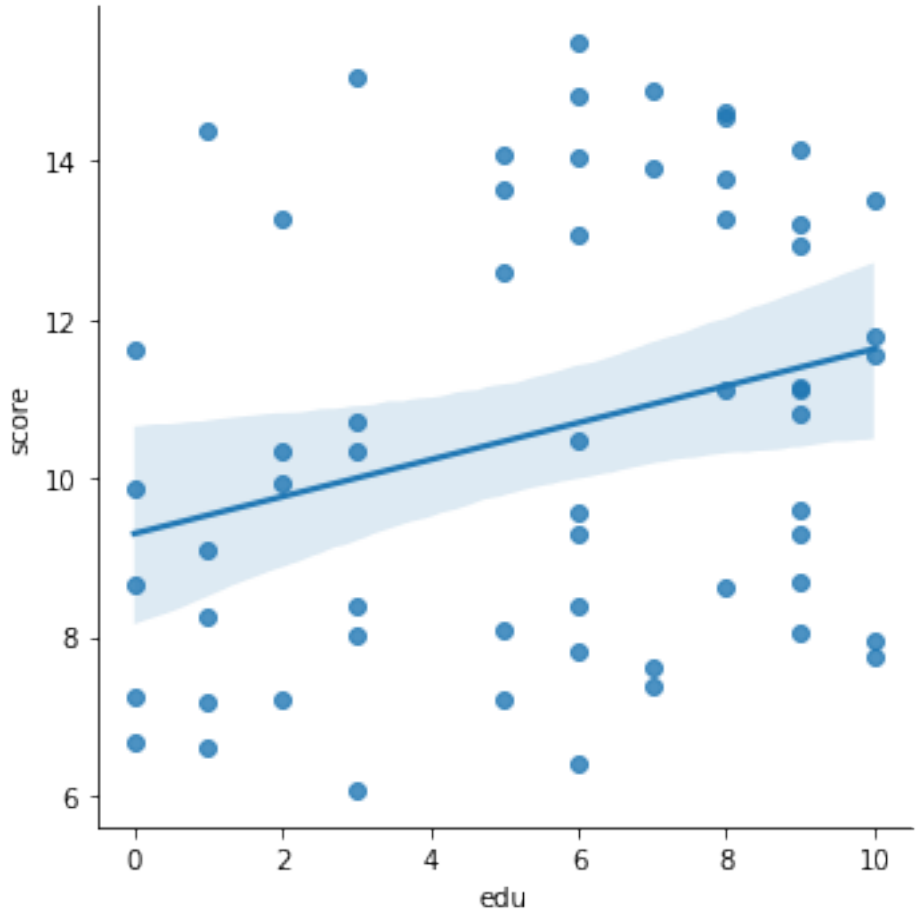
- **Not sensitive** since it does not model the classroom effect (high standard error).
- **Wrong** because, residuals are not normals, and it considers samples from the same classroom to be indenpendant.

```
lm_glob = smf.ols('score ~ edu', df).fit()

#print(lm_glob.summary())
print(lm_glob.t_test('edu'))
print("MSE=%.3f" % lm_glob.mse_resid)
results.loc[len(results)] = ["LM-Global (biased)"] +\
    list(rmse_coef_tstat_pval(mod=lm_glob, var='edu'))
```

```
                            Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0             0.2328      0.109      2.139      0.037       0.015       0.451
==============================================================================
MSE=7.262
```

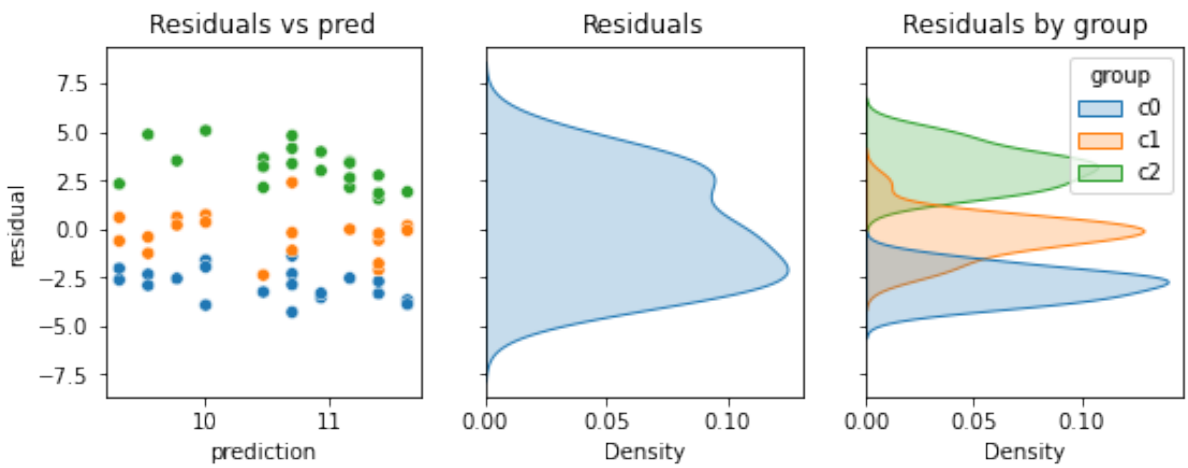
Plot

```
_ = sns.lmplot(x="edu", y="score", data=df)
```



Model diagnosis: plot the normality of the residuals and residuals vs prediction.

```
plot_lm_diagnosis(residual=lm_glob.resid,
                  prediction=lm_glob.predict(df), group=df.classroom)
```



Model a classroom intercept as a fixed effect: ANCOVA

Remember ANCOVA = ANOVA with covariates. Model the classroom $z =$ `classroom` (as a fixed effect), ie a vertical shift for each classroom. The slope is the same for all classrooms. For each individual i and each classroom j the model is:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j z_{ij} + \varepsilon_{ij},$$

where, u_j is the coefficient (an intercept, or a shift) associated with classroom j and $z_{ij} = 1$ if subject i belongs to classroom j else $z_{ij} = 0$.

The general R formula is: `y ~ x + z` which in this case is `score ~ edu + classroom`.

This model is:

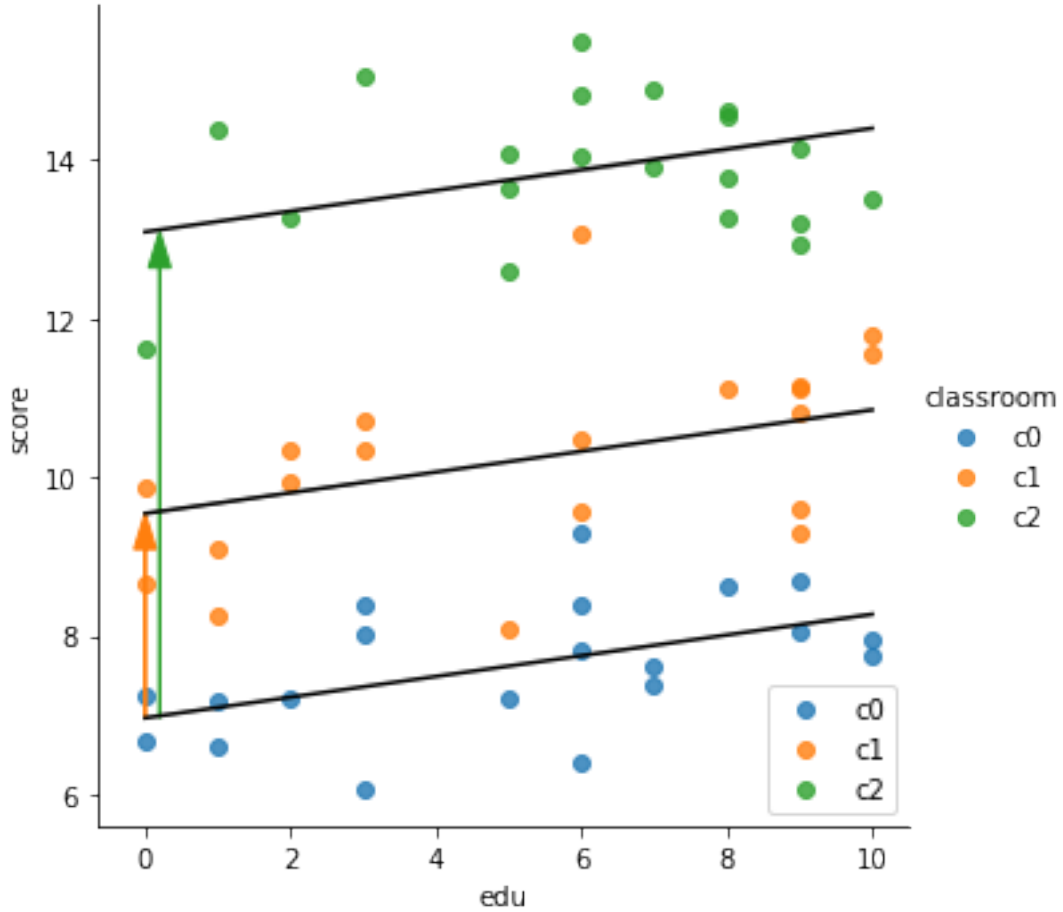
- **Sensitive** since it does not model the classroom effect (lower standard error). But,
- **questionable** because it considers the classroom to have a fixed constant effect without any uncertainty. However, those classrooms have been sampled from a larger samples of classrooms within the country.

```
ancova_inter = smf.ols('score ~ edu + classroom', df).fit()
# print(sm.stats.anova_lm(ancova_inter, typ=3))
# print(ancova_inter.summary())
print(ancova_inter.t_test('edu'))

print("MSE=%.3f" % ancova_inter.mse_resid)
results.loc[len(results)] = ["ANCOVA-Inter (biased)"] +\
    list(rmse_coef_tstat_pval(mod=ancova_inter, var='edu'))
```

```
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0             0.1307      0.038      3.441      0.001       0.055       0.207
==============================================================================
MSE=0.869
```

Plot

```
plot_ancova_oneslope_grpintercept(x="edu", y="score",
                                  group="classroom", model=ancova_inter, df=df)
```



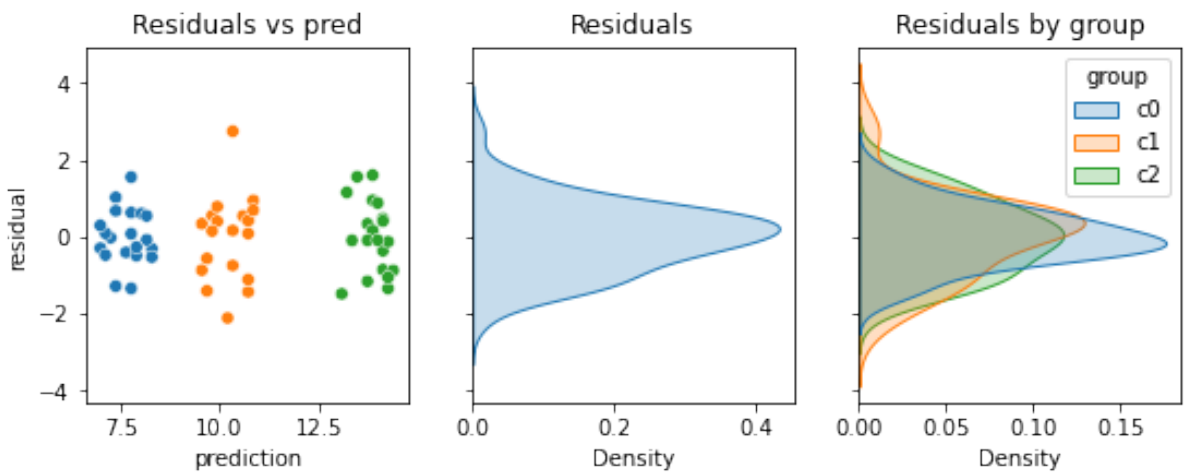
Explore the model

```
mod = ancova_inter

print("## Design matrix (independant variables):")
print(mod.model.exog_names)
print(mod.model.exog[:10])

print("## Outcome (dependant variable):")
print(mod.model.endog_names)
print(mod.model.endog[:10])

print("## Fitted model:")
print(mod.params)
sse_ = np.sum(mod.resid ** 2)
df_ = mod.df_resid
mod.df_model
print("MSE %f" % (sse_ / df_), "or", mod.mse_resid)

print("## Statistics:")
print(mod.tvalues, mod.pvalues)
```

```
## Design matrix (independant variables):
['Intercept', 'classroom[T.c1]', 'classroom[T.c2]', 'edu']
[[ 1.  0.  0.  2.]
 [ 1.  0.  0. 10.]
 [ 1.  0.  0.  3.]
 [ 1.  0.  0.  5.]
 [ 1.  0.  0.  6.]
 [ 1.  0.  0.  6.]
 [ 1.  0.  0.  3.]
 [ 1.  0.  0.  0.]
 [ 1.  0.  0.  6.]
 [ 1.  0.  0.  9.]]
## Outcome (dependant variable):
score
[7.20435162 7.96308267 8.38313712 7.21304665 8.37963003 6.40552793
 8.03417677 6.67164168 7.8268605  8.06401823]
## Fitted model:
Intercept          6.965429
classroom[T.c1]    2.577854
classroom[T.c2]    6.129755
edu                0.130717
dtype: float64
MSE 0.869278 or 0.869277616553041
## Statistics:
Intercept          24.474487
classroom[T.c1]     8.736851
classroom[T.c2]    20.620005
edu                 3.441072
dtype: float64 Intercept          1.377577e-31
classroom[T.c1]    4.815552e-12
classroom[T.c2]    7.876446e-28
edu                1.102091e-03
dtype: float64
```

Normality of the residuals

```
plot_lm_diagnosis(residual=ancova_inter.resid,
                  prediction=ancova_inter.predict(df), group=df.classroom)
```



Fixed effect is the coeficient or parameter (β_1 in the model) that is associated with a continuous covariates (age, education level, etc.) or (categorical) factor (sex, etc.) that is known without uncertainty once a subject is sampled.

Random effect, in contrast, is the coeficient or parameter (u_j in the model below) that is associated with a continuous covariates or factor (classroom, individual, etc.) that is not known without uncertainty once a subject is sampled. It generally conrespond to some random sampling. Here the classroom effect depends on the teacher which has been sampled from a larger samples of classrooms within the country. Measures are structured by units or a clustering structure that is possibly hierarchical. Measures within units are not independant. Measures between top level units are independant.

There are multiple ways to deal with structured data with random effect. One simple approach is to aggregate.

Aggregation of data into independent units

Aggregation of measure at classroom level: average all values within classrooms to perform statistical analysis between classroom. 1. **Level 1 (within unit)**: Average by classrom:

$$x_j = \text{mean}_i(x_{ij}), y_j = \text{mean}_i(y_{ij}), \text{for } j \in \{1, 2, 3\}.$$

2. **Level 2 (between independant units)** Regress averaged `score` on a averaged `edu`:

$$y_j = \beta_0 + \beta_1 x_j + \varepsilon_j$$

. The general R formula is: `y ~ x` which in this case is `score ~ edu`.

This model is:

- **Correct** because the aggregated data are independent.
- **Not sensitive** since all the within classroom association between edu and is lost. Moreover, at the aggregate level, there would only be three data points.

```
agregate = df.groupby('classroom').mean()
lm_agregate = smf.ols('score ~ edu', agregate).fit()
#print(lm_agregate.summary())
print(lm_agregate.t_test('edu'))
```

(continues on next page)

(continued from previous page)

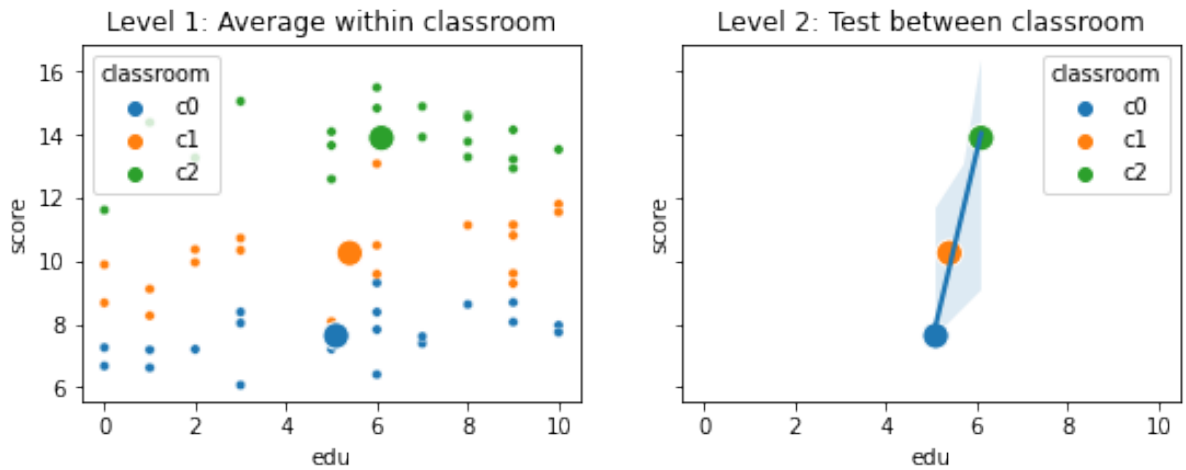
```
print("MSE=%.3f" % lm_agregate.mse_resid)
results.loc[len(results)] = ["Aggregation"] +\
    list(rmse_coef_tstat_pval(mod=lm_agregate, var='edu'))
```

```
                            Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0             6.0734      0.810      7.498      0.084      -4.219      16.366
==============================================================================
MSE=0.346
```

Plot

```
agregate = agregate.reset_index()
fig, axes = plt.subplots(1, 2, figsize=(9, 3), sharex=True, sharey=True)
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=df, ax=axes[0], s=20, legend=False)
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=agregate, ax=axes[0], s=150)
axes[0].set_title("Level 1: Average within classroom")

sns.regplot(x="edu", y="score", data=agregate, ax=axes[1])
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=agregate, ax=axes[1], s=150)
axes[1].set_title("Level 2: Test between classroom")
```

```
Text(0.5, 1.0, 'Level 2: Test between classroom')
```



Hierarchical/multilevel modeling

Another approach to hierarchical data is analyzing data from one unit at a time. Thus, we run three separate linear regressions - one for each classroom in the sample leading to three estimated parameters of the `score` vs `edu` association. Then the paramteres are tested across the classrooms:

1. Run three separate linear regressions - one for each classroom

$$y_{ij} = \beta_{0j} + \beta_{1j}x_{ij} + \varepsilon_{ij}, \text{for } j \in \{1, 2, 3\}$$

   The general R formula is: `y ~ x` which in this case is `score ~ edu` within classrooms.

2. Test across the classrooms if is the $\text{mean}_j(\beta_{1j}) = \beta_0 \neq 0$:

$$\beta_{1j} = \beta_0 + \varepsilon_j$$

   The general R formula is: `y ~ 1` which in this case is `beta_edu ~ 1`.

This model is:

- **Correct** because the invididual estimated parameters are independent.
- **sensitive** since it allows to model differents slope for each classroom (see fixed interaction or random slope below). But it is but **not optimally designed** since there are many models, and each one does not take advantage of the information in data from other classroom. This can also make the results "noisy" in that the estimates from each model are not based on very much data

```
# Level 1 model within classes
x, y, group = 'edu', 'score', 'classroom'

lv1 = [[group_lab, smf.ols('%s ~ %s' % (y, x), group_df).fit().params[x]]
       for group_lab, group_df in df.groupby(group)]

lv1 = pd.DataFrame(lv1, columns=[group, 'beta'])
print(lv1)

# Level 2 model test beta_edu != 0
lm_hm = smf.ols('beta ~ 1', lv1).fit()
print(lm_hm.t_test('Intercept'))
print("MSE=%.3f" % lm_hm.mse_resid)

results.loc[len(results)] = ["Hierarchical"] + \
    list(rmse_coef_tstat_pval(mod=lm_hm, var='Intercept'))
```

```
  classroom      beta
0        c0  0.129084
1        c1  0.177567
2        c2  0.055772
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
```

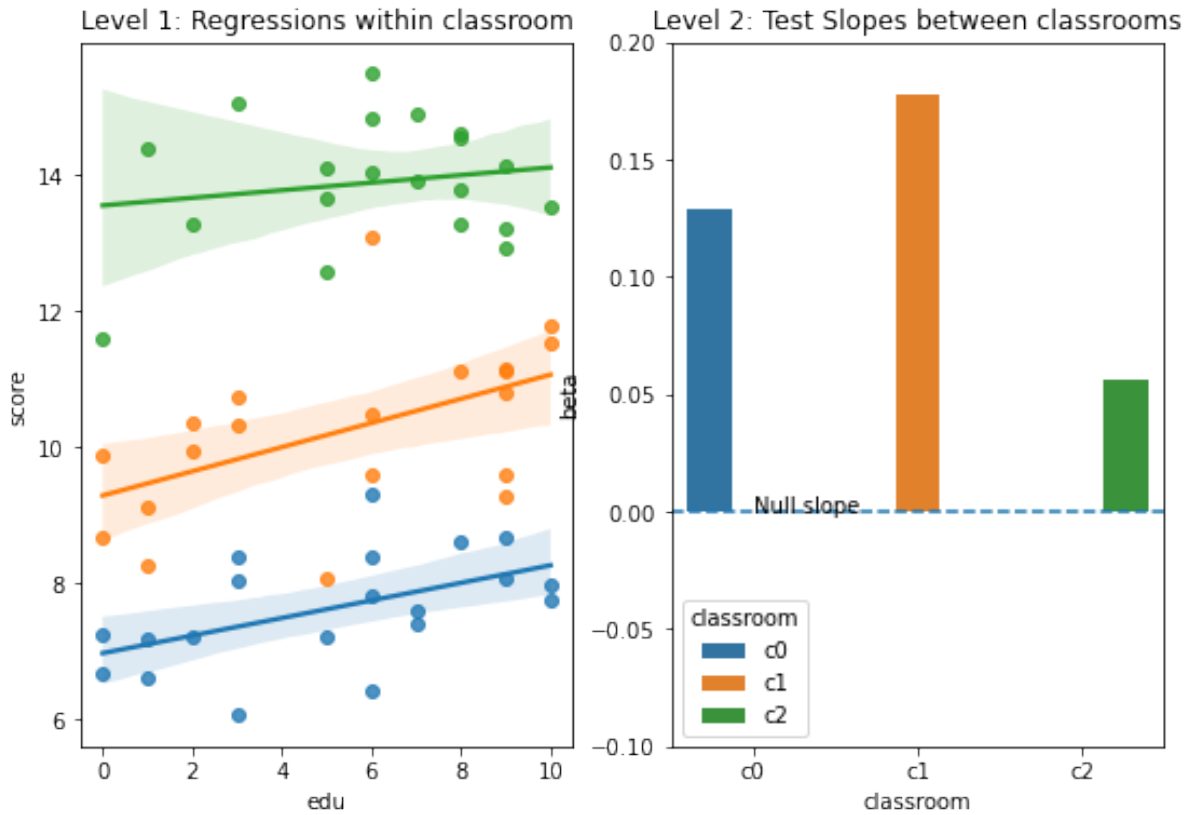
(continues on next page)

(continued from previous page)

```
c0             0.1208      0.035      3.412      0.076     -0.032      0.273
==============================================================================
MSE=0.004
```

Plot

```
fig, axes = plt.subplots(1, 2, figsize=(9, 6))
for group_lab, group_df in df.groupby(group):
    sns.regplot(x=x, y=y, data=group_df, ax=axes[0])

axes[0].set_title("Level 1: Regressions within %s" % group)

_ = sns.barplot(x=group, y="beta", hue=group, data=lv1, ax=axes[1])
axes[1].axhline(0, ls='--')
axes[1].text(0, 0, "Null slope")
axes[1].set_ylim(-.1, 0.2)
_ = axes[1].set_title("Level 2: Test Slopes between classrooms")
```



Model the classroom random intercept: linear mixed model

Linear mixed models (also called multilevel models) can be thought of as a trade off between these two alternatives. The individual regressions has many estimates and lots of data, but is noisy. The aggregate is less noisy, but may lose important differences by averaging all samples within each classroom. LMMs are somewhere in between.

Model the classroom z = `classroom` (as a random effect). For each individual i and each classroom j the model is:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j z_{ij} + \varepsilon_{ij},$$

where, u_j is a **random intercept** following a normal distribution associated with classroom j.

The general R formula is: `y ~ x + (1|z)` which in this case it is `score ~ edu + (1|classroom)`. For python statmodels, the grouping factor `|classroom` is omited an provided as `groups` parameter.

```
lmm_inter = smf.mixedlm("score ~ edu", df, groups=df["classroom"],
                        re_formula="~1").fit()
# But since the default use a random intercept for each group, the following
# formula would have provide the same result:
# lmm_inter = smf.mixedlm("score ~ edu", df, groups=df["classroom"]).fit()
print(lmm_inter.summary())

results.loc[len(results)] = ["LMM-Inter"] + \
    list(rmse_coef_tstat_pval(mod=lmm_inter, var='edu'))
```

```
        Mixed Linear Model Regression Results
=======================================================
Model:            MixedLM Dependent Variable: score
No. Observations: 60      Method:             REML
No. Groups:       3       Scale:              0.8693
Min. group size:  20      Log-Likelihood:     -88.8676
Max. group size:  20      Converged:          Yes
Mean group size:  20.0
-------------------------------------------------------
              Coef. Std.Err.   z   P>|z| [0.025 0.975]
-------------------------------------------------------
Intercept     9.865    1.789 5.514 0.000  6.359 13.372
edu           0.131    0.038 3.453 0.001  0.057  0.206
Group Var     9.427   10.337
=======================================================
```

Explore model

```
print("Fixed effect:")
print(lmm_inter.params)

print("Random effect:")
print(lmm_inter.random_effects)
```

(continues on next page)

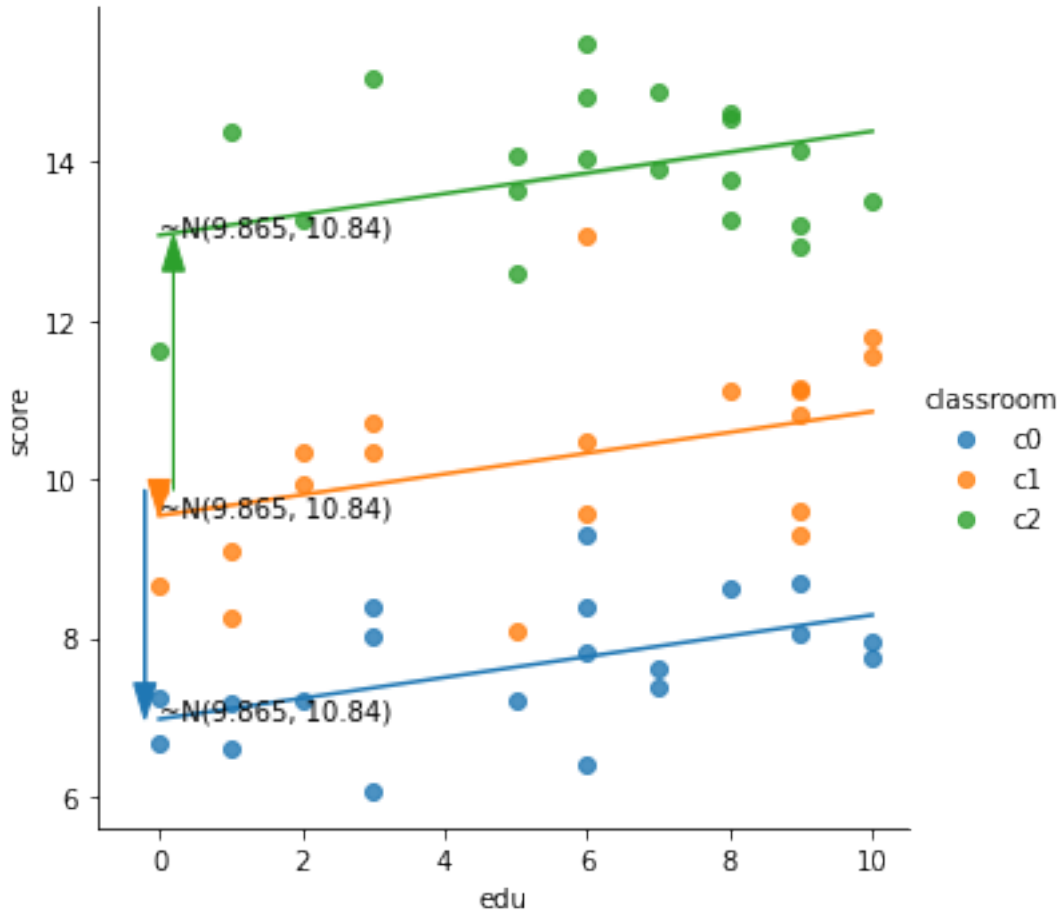
(continued from previous page)

```
intercept = lmm_inter.params['Intercept']
var = lmm_inter.params["Group Var"]
```

```
Fixed effect:
Intercept     9.865327
edu           0.131193
Group Var    10.844222
dtype: float64
Random effect:
{'c0': Group   -2.889009
dtype: float64, 'c1': Group   -0.323129
dtype: float64, 'c2': Group    3.212138
dtype: float64}
```

Plot

```
plot_lmm_oneslope_randintercept(x='edu', y='score',
                                group='classroom', df=df, model=lmm_inter)
```



4.3.3 Random slope

Now suppose that the classroom random effect is not just a vertical shift (random intercept) but that some teachers "compensate" or "amplify" educational disparity. The slope of the linear relation between score and edu for teachers that amplify will be larger. In the contrary, it will be smaller for teachers that compensate.

Model the classroom intercept and slode as a fixed effect: ANCOVA with interactions

1. Model the global association between edu and score: $y_{ij} = \beta_0 + \beta_1 x_{ij}$, in R: `score ~ edu`.
2. Model the classroom $z_j =$ `classroom` (as a fixed effect) as a vertical shift (intercept, u_j^1) for each classroom j indicated by z_{ij}: $y_{ij} = u_j^1 z_{ij}$, in R: `score ~ classroom`.
3. Model the classroom (as a fixed effect) specitic slope (u_j^α): $y_i = u_j^\alpha x_i z_j$ `score ~ edu:classroom`. The $x_i z_j$ forms 3 new columns with values of x_i for each edu level, ie.: for z_j `classroom` 1, 2 and 3.
4. Put everything together:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j^1 z_{ij} + u_j^\alpha z_{ij} x_{ij} + \varepsilon_{ij},$$

   in R: `score ~ edu + classroom edu:classroom` or mor simply `score ~ edu * classroom` that denotes the full model with the additive contribution of each regressor and all their interactions.

```
ancova_full = smf.ols('score ~ edu + classroom + edu:classroom', df).fit()
# Full model (including interaction) can use this notation:
# ancova_full = smf.ols('score ~ edu * classroom', df).fit()

# print(sm.stats.anova_lm(lm_fx, typ=3))
# print(lm_fx.summary())
print(ancova_full.t_test('edu'))
print("MSE=%.3f" % ancova_full.mse_resid)
results.loc[len(results)] = ["ANCOVA-Full (biased)"] + \
    list(rmse_coef_tstat_pval(mod=ancova_full, var='edu'))
```

```
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0             0.1291      0.065      1.979      0.053      -0.002       0.260
==============================================================================
MSE=0.876
```

The graphical representation of the model would be the same than the one provided for "Model a classroom intercept as a fixed effect: ANCOVA". The same slope (associated to edu) with different interpcept, depicted as dashed black lines. Moreover we added, as solid lines, the model's prediction that account different slopes.

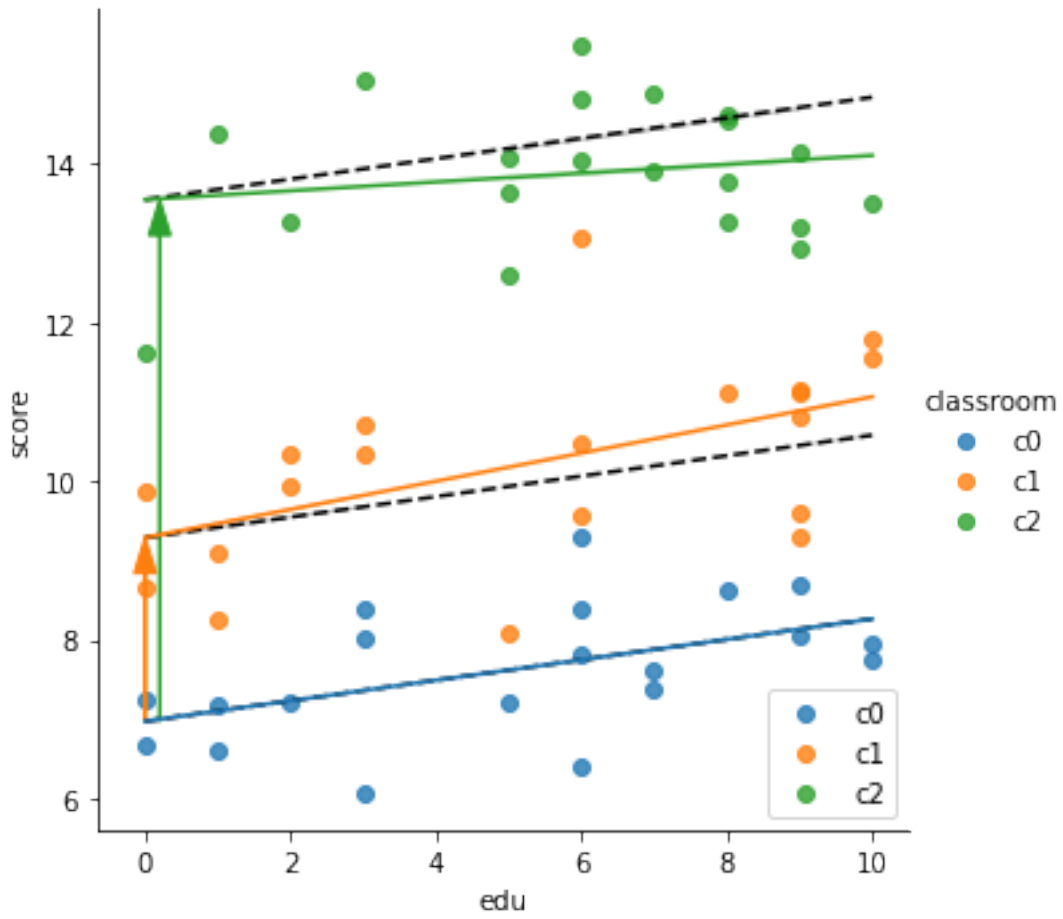
```
print("Model parameters:")
print(ancova_full.params)
```

(continues on next page)

(continued from previous page)

```
plot_ancova_fullmodel(x='edu', y='score',
                      group='classroom', df=df, model=ancova_full)
```

```
Model parameters:
Intercept              6.973753
classroom[T.c1]        2.316540
classroom[T.c2]        6.578594
edu                    0.129084
edu:classroom[T.c1]    0.048482
edu:classroom[T.c2]   -0.073313
dtype: float64
```



Model the classroom random intercept and slope with LMM

The model looks similar to the ANCOVA with interactions:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j^1 z_{ij} + u_j^\alpha z_{ij} x_{ij} + \varepsilon_{ij},$$

but:

- u_j^1 is a **random intercept** associated with classroom j following the same normal distribution for all classroom, $u_j^1 \sim \mathcal{N}(\mathbf{0}, \sigma^{\mathbf{1}})$.
- u_j^α is a **random slope** associated with classroom j following the same normal distribution for all classroom, $u_j^\alpha \sim \mathcal{N}(\mathbf{0}, \sigma^\alpha)$.

Note the difference with linear model: the variances parameters $(\sigma^1, \sigma^\alpha)$ should be estimated together with fixed effect $(\beta_0 + \beta_1)$ and random effect $(u^1, u_j^\alpha$, one pair of random intercept/slope per classroom). The R notation is: `score ~ edu + (edu | classroom)`. or `score ~ 1 + edu + (1 + edu | classroom)`, remember that intercepts are implicit. In statmodels, the notation is `~1+edu` or `~edu` since the groups is provided by the `groups` argument.

```
lmm_full = smf.mixedlm("score ~ edu", df, groups=df["classroom"],
                       re_formula="~1+edu").fit()
print(lmm_full.summary())
results.loc[len(results)] = ["LMM-Full (biased)"] + \
    list(rmse_coef_tstat_pval(mod=lmm_full, var='edu'))
```

```
          Mixed Linear Model Regression Results
==========================================================
Model:             MixedLM  Dependent Variable:  score
No. Observations:  60       Method:              REML
No. Groups:        3        Scale:               0.8609
Min. group size:   20       Log-Likelihood:      -88.5987
Max. group size:   20       Converged:           Yes
Mean group size:   20.0
----------------------------------------------------------
                Coef.  Std.Err.   z   P>|z| [0.025 0.975]
----------------------------------------------------------
Intercept        9.900    1.912 5.177 0.000  6.152 13.647
edu              0.127    0.046 2.757 0.006  0.037  0.218
Group Var       10.760   12.279
Group x edu Cov -0.121    0.318
edu Var          0.001    0.012
==========================================================
```

```
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/base/model.
↪py:566: ConvergenceWarning: Maximum Likelihood optimization failed to converge.␣
↪Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/regression/mixed_
↪linear_model.py:2200: ConvergenceWarning: Retrying MixedLM optimization with␣
↪lbfgs
  warnings.warn(
```

(continues on next page)

(continued from previous page)

```
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/regression/mixed_
↪linear_model.py:1634: UserWarning: Random effects covariance is singular
  warnings.warn(msg)
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/regression/mixed_
↪linear_model.py:2237: ConvergenceWarning: The MLE may be on the boundary of the␣
↪parameter space.
  warnings.warn(msg, ConvergenceWarning)
```

The warning results in a singular fit (correlation estimated at 1) caused by too little variance among the random slopes. It indicates that we should considere to remove random slopes.

4.3.4 Conclusion on modeling random effects

```
print(results)
```

```
                   Model      RMSE      Coef      Stat      Pval
0     LM-Global (biased)  2.694785  0.232842  2.139165  0.036643
1  ANCOVA-Inter (biased)  0.932351  0.130717  3.441072  0.001102
2            Aggregation  0.587859  6.073401  7.497672  0.084411
3           Hierarchical  0.061318  0.120808  3.412469  0.076190
4              LMM-Inter  0.916211  0.131193  3.453472  0.000553
5   ANCOVA-Full (biased)  0.935869  0.129084  1.978708  0.052959
6      LMM-Full (biased)  0.911742  0.127269  2.756917  0.005835
```

Random intercepts

1. LM-Global is wrong (consider residuals to be independent) and has a large error (RMSE, Root Mean Square Error) since it does not adjust for classroom effect.
2. ANCOVA-Inter is "wrong" (consider residuals to be independent) but it has a small error since it adjusts for classroom effect.
3. Aggregation is ok (units average are independent) but it has a very large error.
4. Hierarchical model is ok (unit average are independent) and it has a reasonable error (look at the statistic, not the RMSE).
5. LMM-Inter (with random intercept) is ok (it models residuals non-independence) and it has a small error.
6. ANCOVA-Inter, Hierarchical model and LMM provide similar coefficients for the fixed effect. So if statistical significance is not the key issue, the "biased" ANCOVA is a reasonable choice.
7. Hierarchical and LMM with random intercept are the best options (unbiased and sensitive), with an advantage to LMM.

Random slopes

Modeling individual slopes in both ANCOVA-Full and LMM-Full decreased the statistics, suggesting that the supplementary regressors (one per classroom) do not significantly improve the fit of the model (see errors).

4.3.5 Theory of Linear Mixed Models

If we consider only 6 samples ($i \in \{1, 6\}$, two sample for each classroom $j \in$ {c0, c1, c2}) and the random intercept model. Stacking the 6 observations, the equation $y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j z_j + \varepsilon_{ij}$ gives :

$$\begin{bmatrix} \text{score} \\ 7.2 \\ 7.9 \\ 9.1 \\ 11.1 \\ 14.6 \\ 14.0 \end{bmatrix} = \begin{bmatrix} \text{Inter} & \text{Edu} \\ 1 & 2 \\ 1 & 10 \\ 1 & 1 \\ 1 & 9 \\ 1 & 8 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} \text{Fix} \\ \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \text{c1} & \text{c2} & \text{c3} \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{Rand} \\ u_{11} \\ u_{12} \\ u_{13} \end{bmatrix} + \begin{bmatrix} \text{Err} \\ \epsilon_1 \\ \epsilon_2 \end{bmatrix}$$

where $\mathbf{u_1} = u_{11}, u_{12}, u_{12}$ are the 3 parameters associated with the 3 level of the single random factor `classroom`.

This can be re-written in a more general form as:

$$\mathbf{y} = \mathbf{X}^T \beta + \mathbf{Z}^T \mathbf{u} + \varepsilon,$$

where: - $\mathbf{y}$ is the $N \times 1$ vector of the N observations. - $\mathbf{X}$ is the $N \times P$ design matrix, which represents the known values of the P covariates for the N observations. - β is a $P \times 1$ vector unknown regression coefficients (or fixed-effect parameters) associated with the P covariates. - ε is a $N \times 1$ vector of residuals $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$, where $\mathbf{R}$ is a $N \times N$ matrix. - $\mathbf{Z}$ is a $N \times Q$ design matrix of random factors and covariates. In an LMM in which only the intercepts are assumed to vary randomly from Q units, the $\mathbf{Z}$ matrix would simply be Q columns of indicators 1 (if subject belong to unit q) or 0 otherwise. - $\mathbf{u}$ is a $Q \times 1$ vector of Q random effects associated with the Q covariates in the $\mathbf{Z}$ matrix. Note that one random factor of 3 levels will be coded by 3 coefficients in $\mathbf{u}$ and 3 columns $\mathbf{Z}$. $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{D})$ where $\mathbf{D}$ is plays a central role of the covariance structures associated with the mixed effect.

Covariance structures of the residuals covariance matrix: :math:`mathbf{R}`

Many different covariance structures are possible for the $\mathbf{R}$ matrix. The simplest covariance matrix for $\mathbf{R}$ is the diagonal structure, in which the residuals associated with observations on the same subject are assumed to be uncorrelated and to have equal variance: $\mathbf{R} = \sigma \mathbf{I}_N$. Note that in this case, the correlation between observation within unit stem from mixed effects, and will be encoded in the $\mathbf{D}$ below. However, other model exists: popular models are the compound symmetry and first-order autoregressive structure, denoted by AR(1).

Covariance structures associated with the mixed effect

Many different covariance structures are possible for the $\mathbf{D}$ matrix. The usual prartice associate a single variance parameter (a scalar, σ_k) to each random-effects factor k (eg. `classroom`). Hence $\mathbf{D}$ is simply parametrized by a set of scalars $\sigma_k, k \in \{1, K\}$ for the K random factors such the sum of levels of the K factors equals Q. In our case $K = 1$ with 3 levels ($Q = 3$), thus $\mathbf{D} = \sigma_k \mathbf{I}_Q$. Factors k define k **variance components** whose parameters σ_k should be estimated addition to the variance of the model errors σ. The σ_k and σ will define the overall covariance structure: $\mathbf{V}$, as define below.

In this model, the effect of a particular level (eg. classroom 0 `c0`) of a random factor is supposed to be sampled from a normal distritution of variance σ_k. This is a crucial aspect of LMM which is related to ℓ_2-regularization or Bayes Baussian prior. Indeed, the estimator of associated to each level u_i of a random effect is shrinked toward 0 since $u_i \sim \mathcal{N}(0, \sigma_k)$. Thus it tends to be

smaller than the estimated effects would be if they were computed by treating a random factor as if it were fixed.

Overall covariance structure as variance components :math:`mathbf{V}`

The overall covariance structure can be obtained by:

$$\mathbf{V} = \sum_k \sigma_k \mathbf{Z}\mathbf{Z}' + \mathbf{R}.$$

The $\sum_k \sigma_k \mathbf{Z}\mathbf{Z}'$ define the $N \times N$ variance structure, using k variance components, modeling the non-independance between the observations. In our case with only one component we get:

$$\begin{aligned}
\mathbf{V} &= \begin{bmatrix} \sigma_k & \sigma_k & 0 & 0 & 0 & 0 \\ \sigma_k & \sigma_k & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_k & \sigma_k & 0 & 0 \\ 0 & 0 & \sigma_k & \sigma_k & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_k & \sigma_k \\ 0 & 0 & 0 & 0 & \sigma_k & \sigma_k \end{bmatrix} + \begin{bmatrix} \sigma & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma \end{bmatrix} \\
&= \begin{bmatrix} \sigma_k + \sigma & \sigma_k & 0 & 0 & 0 & 0 \\ \sigma_k & \sigma_k + \sigma & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_k + \sigma & \sigma_k & 0 & 0 \\ 0 & 0 & \sigma_k & \sigma_k + \sigma & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_k + \sigma & \sigma_k \\ 0 & 0 & 0 & 0 & \sigma_k & \sigma_k + \sigma \end{bmatrix}
\end{aligned}$$

The model to be minimized

Here σ_k and σ are called variance components of the model. Solving the problem constist in the estimation the fixed effect β and the parameters σ, σ_k of the variance-covariance structure. This is obtained by minizing the The likelihood of the sample:

$$l(\mathbf{y}, \beta, \sigma, \sigma_k) = \frac{1}{2\pi^{n/2} \det(\mathbf{V})^{1/2}} \exp -\frac{1}{2}(\mathbf{y} - \mathbf{X}\beta)\mathbf{V}^{-1}(\mathbf{y} - \mathbf{X}\beta)$$

LMM introduces the variance-covariance matrix $\mathbf{V}$ to reweigtht the residuals according to the non-independance between observations. If $\mathbf{V}$ is known, of. The optimal value of be can be obtained analytically using generalized least squares (GLS, minimisation of mean squared error associated with Mahalanobis metric):

$$\hat{\beta} = \mathbf{X'}\hat{\mathbf{V}}^{-1}\mathbf{X}^{-1}\mathbf{X'}\hat{\mathbf{V}}^{-1}\mathbf{y}$$

In the general case, $\mathbf{V}$ is unknown, therefore iterative solvers should be use to estimate the fixed effect β and the parameters $(\sigma, \sigma_k, \ldots)$ of variance-covariance matrix $\mathbf{V}$. The ML Maximum Likelihood estimates provide biased solution for $\mathbf{V}$ because they do not take into account the loss of degrees of freedom that results from estimating the fixed-effect parameters in β. For this reason, REML (restricted (or residual, or reduced) maximum likelihood) is often preferred to ML estimation.

4.3.6 Checking model assumptions (Diagnostics)

Residuals plotted against predicted values represents a random pattern or not.

These residual vs. fitted plots are used to verify model assumptions and to detect outliers and potentially influential observations.

4.3.7 References

- Brady et al. 2014: Brady T. West, Kathleen B. Welch, Andrzej T. Galecki, Linear Mixed Models: A Practical Guide Using Statistical Software (2nd Edition), 2014
- Bruin 2006: Introduction to Linear Mixed Models, UCLA, Statistical Consulting Group.
- Statsmodel: Linear Mixed Effects Models
- Comparing R lmer to statsmodels MixedLM
- Statsmoels: Variance Component Analysis with nested groups

4.4 Multivariate statistics

Multivariate statistics includes all statistical techniques for analyzing samples made of two or more variables. The data set (a $N \times P$ matrix $\mathbf{X}$) is a collection of N independent samples column **vectors** $[\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_N]$ of length P

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1^T- \\ \vdots \\ -\mathbf{x}_i^T- \\ \vdots \\ -\mathbf{x}_P^T- \end{bmatrix} = \begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1P} \\ \vdots & & \vdots & & \vdots \\ x_{i1} & \cdots & x_{ij} & \cdots & x_{iP} \\ \vdots & & \vdots & & \vdots \\ x_{N1} & \cdots & x_{Nj} & \cdots & x_{NP} \end{bmatrix} = \begin{bmatrix} x_{11} & \ldots & x_{1P} \\ \vdots & & \vdots \\ & \mathbf{X} & \\ \vdots & & \vdots \\ x_{N1} & \ldots & x_{NP} \end{bmatrix}_{N \times P}.$$

4.4.1 Linear Algebra

Euclidean norm and distance

The Euclidean norm of a vector $\mathbf{a} \in \mathbb{R}^P$ is denoted

$$\|\mathbf{a}\|_2 = \sqrt{\sum_i^P {a_i}^2}$$

The Euclidean distance between two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^P$ is

$$\|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_i^P (a_i - b_i)^2}$$

Dot product and projection

Source: Wikipedia

Algebraic definition

The dot product, denoted "·" of two P-dimensional vectors $\mathbf{a} = [a_1, a_2, ..., a_P]$ and $\mathbf{a} = [b_1, b_2, ..., b_P]$ is defined as

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b} = \sum_i a_i b_i = \begin{bmatrix} a_1 & \dots & \mathbf{a}^T & \dots & a_P \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ \mathbf{b} \\ \vdots \\ b_P \end{bmatrix}.$$

The Euclidean norm of a vector can be computed using the dot product, as

$$\|\mathbf{a}\|_2 = \sqrt{\mathbf{a} \cdot \mathbf{a}}.$$

Geometric definition: projection

In Euclidean space, a Euclidean vector is a geometrical object that possesses both a magnitude and a direction. A vector can be pictured as an arrow. Its magnitude is its length, and its direction is the direction that the arrow points. The magnitude of a vector $\mathbf{a}$ is denoted by $\|\mathbf{a}\|_2$. The dot product of two Euclidean vectors $\mathbf{a}$ and $\mathbf{b}$ is defined by

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \, \|\mathbf{b}\|_2 \cos\theta,$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

In particular, if $\mathbf{a}$ and $\mathbf{b}$ are orthogonal, then the angle between them is 90° and

$$\mathbf{a} \cdot \mathbf{b} = 0.$$

At the other extreme, if they are codirectional, then the angle between them is 0° and

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \, \|\mathbf{b}\|_2$$

This implies that the dot product of a vector $\mathbf{a}$ by itself is

$$\mathbf{a} \cdot \mathbf{a} = \|\mathbf{a}\|_2^2.$$

The scalar projection (or scalar component) of a Euclidean vector $\mathbf{a}$ in the direction of a Euclidean vector $\mathbf{b}$ is given by

$$a_b = \|\mathbf{a}\|_2 \cos\theta,$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

In terms of the geometric definition of the dot product, this can be rewritten

$$a_b = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|_2},$$

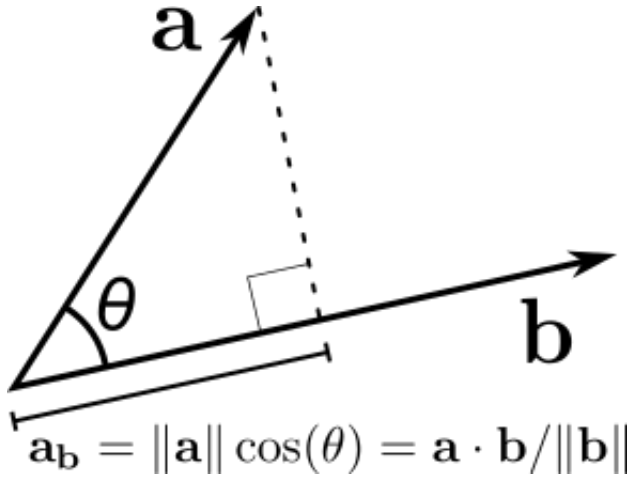


Fig. 4: Projection.

```
import numpy as np
np.random.seed(42)

a = np.random.randn(10)
b = np.random.randn(10)

np.dot(a, b)
```

```
-4.085788532659924
```

4.4.2 Mean vector

The mean $(P \times 1)$ column-vector μ whose estimator is

$$\bar{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x_i} = \frac{1}{N}\sum_{i=1}^{N} \begin{bmatrix} x_{i1} \\ \vdots \\ x_{ij} \\ \vdots \\ x_{iP} \end{bmatrix} = \begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_j \\ \vdots \\ \bar{x}_P \end{bmatrix}.$$

4.4.3 Covariance matrix

- The covariance matrix $\mathbf{\Sigma_{XX}}$ is a **symmetric** positive semi-definite matrix whose element in the j, k position is the covariance between the jth and k^{th} elements of a random vector i.e. the jth and k^{th} columns of $\mathbf{X}$.
- The covariance matrix generalizes the notion of covariance to multiple dimensions.
- The covariance matrix describe the shape of the sample distribution around the mean assuming an elliptical distribution:

$$\mathbf{\Sigma_{XX}} = E(\mathbf{X} - E(\mathbf{X}))^T E(\mathbf{X} - E(\mathbf{X})),$$

whose estimator $\mathbf{S_{XX}}$ is a $P \times P$ matrix given by

$$\mathbf{S_{XX}} = \frac{1}{N-1}(\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T)^T(\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T).$$

If we assume that $\mathbf{X}$ is centered, i.e. $\mathbf{X}$ is replaced by $\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T$ then the estimator is

$$\mathbf{S_{XX}} = \frac{1}{N-1}\mathbf{X}^T\mathbf{X} = \frac{1}{N-1}\begin{bmatrix} x_{11} & \cdots & x_{N1} \\ x_{1j} & \cdots & x_{Nj} \\ \vdots & & \vdots \\ x_{1P} & \cdots & x_{NP} \end{bmatrix}\begin{bmatrix} x_{11} & \cdots & x_{1k} & x_{1P} \\ \vdots & & \vdots & \vdots \\ x_{N1} & \cdots & x_{Nk} & x_{NP} \end{bmatrix} = \begin{bmatrix} s_1 & \cdots & s_{1k} & s_{1P} \\ & \ddots & s_{jk} & \vdots \\ & & s_k & s_{kP} \\ & & & s_P \end{bmatrix},$$

where

$$s_{jk} = s_{kj} = \frac{1}{N-1}\mathbf{x_j}^T\mathbf{x_k} = \frac{1}{N-1}\sum_{i=1}^{N} x_{ij}x_{ik}$$

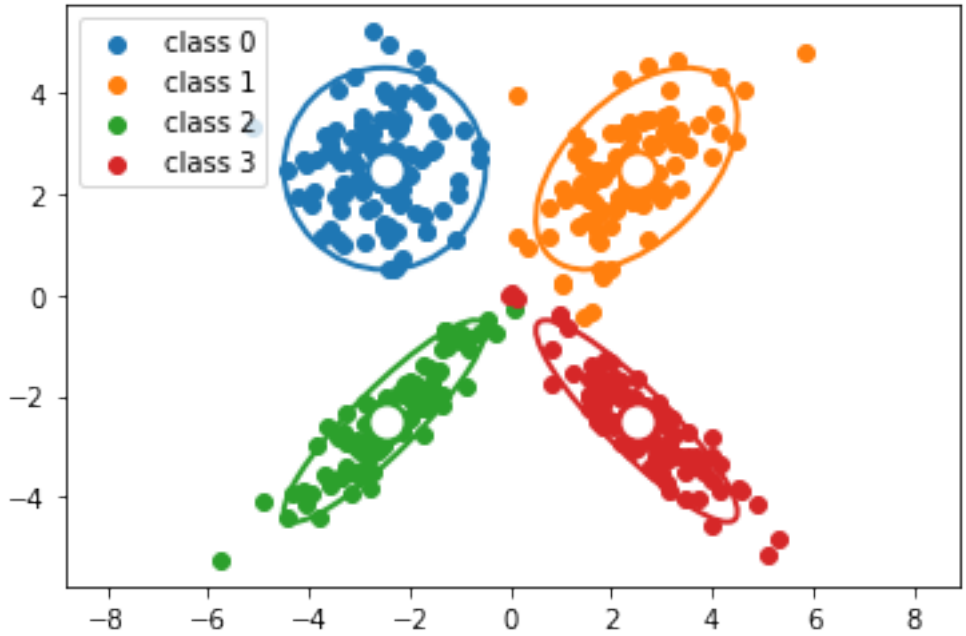
is an estimator of the covariance between the jth and k^{th} variables.

```
## Avoid warnings and force inline plot
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
##
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
import seaborn as sns  # nice color

np.random.seed(42)
colors = sns.color_palette()

n_samples, n_features = 100, 2

mean, Cov, X = [None] * 4, [None] * 4, [None] * 4
mean[0] = np.array([-2.5, 2.5])
Cov[0] = np.array([[1, 0],
                   [0, 1]])

mean[1] = np.array([2.5, 2.5])
Cov[1] = np.array([[1, .5],
                   [.5, 1]])

mean[2] = np.array([-2.5, -2.5])
Cov[2] = np.array([[1, .9],
                   [.9, 1]])

mean[3] = np.array([2.5, -2.5])
Cov[3] = np.array([[1, -.9],
                   [-.9, 1]])
```

(continues on next page)

(continued from previous page)

```
# Generate dataset
for i in range(len(mean)):
    X[i] = np.random.multivariate_normal(mean[i], Cov[i], n_samples)

# Plot
for i in range(len(mean)):
    # Points
    plt.scatter(X[i][:, 0], X[i][:, 1], color=colors[i], label="class %i" % i)
    # Means
    plt.scatter(mean[i][0], mean[i][1], marker="o", s=200, facecolors='w',
                edgecolors=colors[i], linewidth=2)
    # Ellipses representing the covariance matrices
    pystatsml.plot_utils.plot_cov_ellipse(Cov[i], pos=mean[i], facecolor='none',
                                          linewidth=2, edgecolor=colors[i])

plt.axis('equal')
_ = plt.legend(loc='upper left')
```



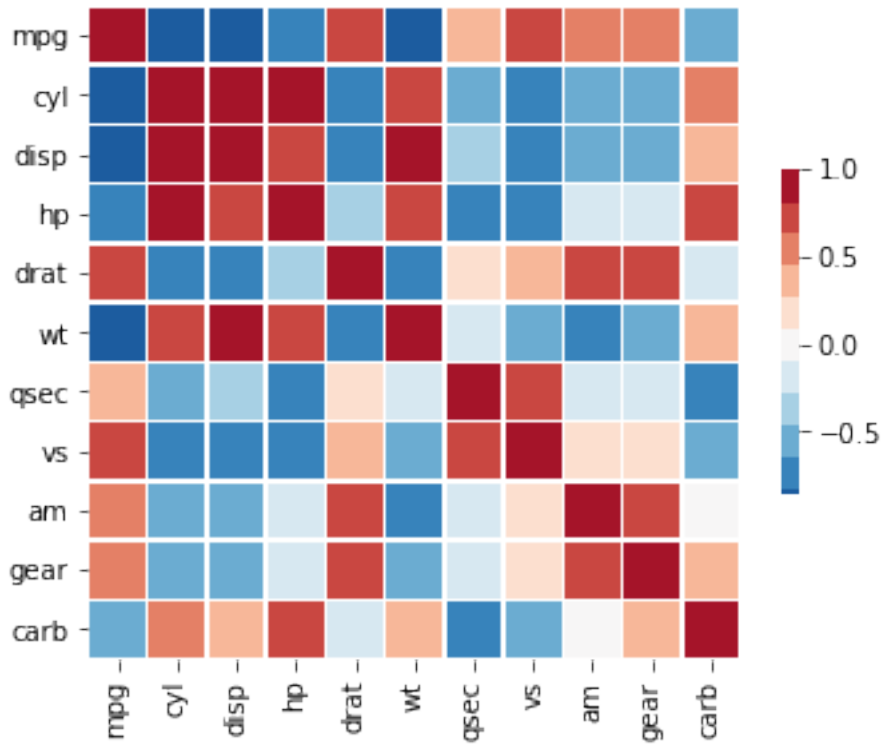
4.4.4 Correlation matrix

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

url = 'https://python-graph-gallery.com/wp-content/uploads/mtcars.csv'
df = pd.read_csv(url)
```

(continues on next page)

(continued from previous page)

```
# Compute the correlation matrix
corr = df.corr()

# Generate a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

f, ax = plt.subplots(figsize=(5.5, 4.5))
cmap = sns.color_palette("RdBu_r", 11)
# Draw the heatmap with the mask and correct aspect ratio
_ = sns.heatmap(corr, mask=None, cmap=cmap, vmax=1, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```



Re-order correlation matrix using AgglomerativeClustering

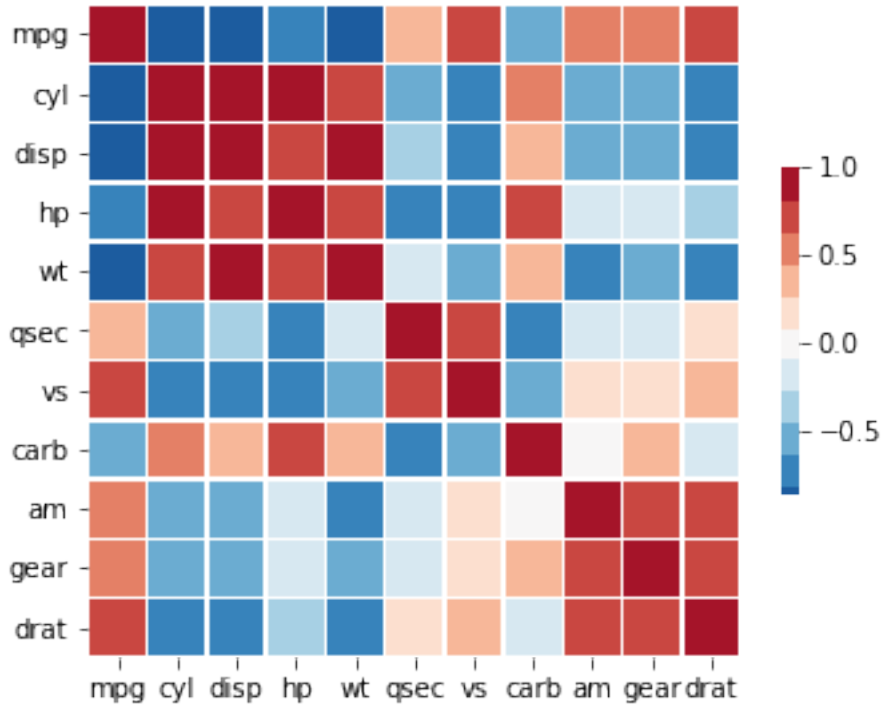
```
# convert correlation to distances
d = 2 * (1 - np.abs(corr))

from sklearn.cluster import AgglomerativeClustering
clustering = AgglomerativeClustering(n_clusters=3, linkage='single', affinity=
↪"precomputed").fit(d)
lab=0

clusters = [list(corr.columns[clustering.labels_==lab]) for lab in set(clustering.
↪labels_)]
print(clusters)

reordered = np.concatenate(clusters)
```

(continues on next page)

(continued from previous page)

```
R = corr.loc[reordered, reordered]

f, ax = plt.subplots(figsize=(5.5, 4.5))
# Draw the heatmap with the mask and correct aspect ratio
_ = sns.heatmap(R, mask=None, cmap=cmap, vmax=1, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

```
[['mpg', 'cyl', 'disp', 'hp', 'wt', 'qsec', 'vs', 'carb'], ['am', 'gear'], ['drat
↪']]
```



4.4.5 Precision matrix

In statistics, precision is the reciprocal of the variance, and the precision matrix is the matrix inverse of the covariance matrix.

It is related to **partial correlations** that measures the degree of association between two variables, while controlling the effect of other variables.

```
import numpy as np

Cov = np.array([[1.0, 0.9, 0.9, 0.0, 0.0, 0.0],
                [0.9, 1.0, 0.9, 0.0, 0.0, 0.0],
                [0.9, 0.9, 1.0, 0.0, 0.0, 0.0],
                [0.0, 0.0, 0.0, 1.0, 0.9, 0.0],
                [0.0, 0.0, 0.0, 0.9, 1.0, 0.0],
                [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]])
```

(continues on next page)

(continued from previous page)

```
print("# Precision matrix:")
Prec = np.linalg.inv(Cov)
print(Prec.round(2))

print("# Partial correlations:")
Pcor = np.zeros(Prec.shape)
Pcor[::] = np.NaN

for i, j in zip(*np.triu_indices_from(Prec, 1)):
    Pcor[i, j] = - Prec[i, j] / np.sqrt(Prec[i, i] * Prec[j, j])

print(Pcor.round(2))
```

```
# Precision matrix:
[[ 6.79 -3.21 -3.21  0.    0.    0.  ]
 [-3.21  6.79 -3.21  0.    0.    0.  ]
 [-3.21 -3.21  6.79  0.    0.    0.  ]
 [ 0.   -0.   -0.    5.26 -4.74 -0.  ]
 [ 0.    0.    0.   -4.74  5.26  0.  ]
 [ 0.    0.    0.    0.    0.    1.  ]]
# Partial correlations:
[[  nan  0.47  0.47 -0.   -0.   -0.  ]
 [  nan   nan  0.47 -0.   -0.   -0.  ]
 [  nan   nan   nan -0.   -0.   -0.  ]
 [  nan   nan   nan   nan  0.9   0.  ]
 [  nan   nan   nan   nan   nan -0.  ]
 [  nan   nan   nan   nan   nan   nan]]
```

4.4.6 Mahalanobis distance

- The Mahalanobis distance is a measure of the distance between two points $\mathbf{x}$ and μ where the dispersion (i.e. the covariance structure) of the samples is taken into account.
- The dispersion is considered through covariance matrix.

This is formally expressed as

$$D_M(\mathbf{x}, \mu) = \sqrt{(\mathbf{x} - \mu)^T \mathbf{\Sigma}^{-1} (\mathbf{x} - \mu)}.$$

Intuitions

- Distances along the principal directions of dispersion are contracted since they correspond to likely dispersion of points.
- Distances othogonal to the principal directions of dispersion are dilated since they correspond to unlikely dispersion of points.

For example

$$D_M(\mathbf{1}) = \sqrt{\mathbf{1}^T \mathbf{\Sigma}^{-1} \mathbf{1}}.$$

```
ones  = np.ones(Cov.shape[0])
d_euc = np.sqrt(np.dot(ones, ones))
d_mah = np.sqrt(np.dot(np.dot(ones, Prec), ones))

print("Euclidean norm of ones=%.2f. Mahalanobis norm of ones=%.2f" % (d_euc, d_
↪mah))
```

```
Euclidean norm of ones=2.45. Mahalanobis norm of ones=1.77
```

The first dot product that distances along the principal directions of dispersion are contracted:

```
print(np.dot(ones, Prec))
```

```
[0.35714286 0.35714286 0.35714286 0.52631579 0.52631579 1.        ]
```

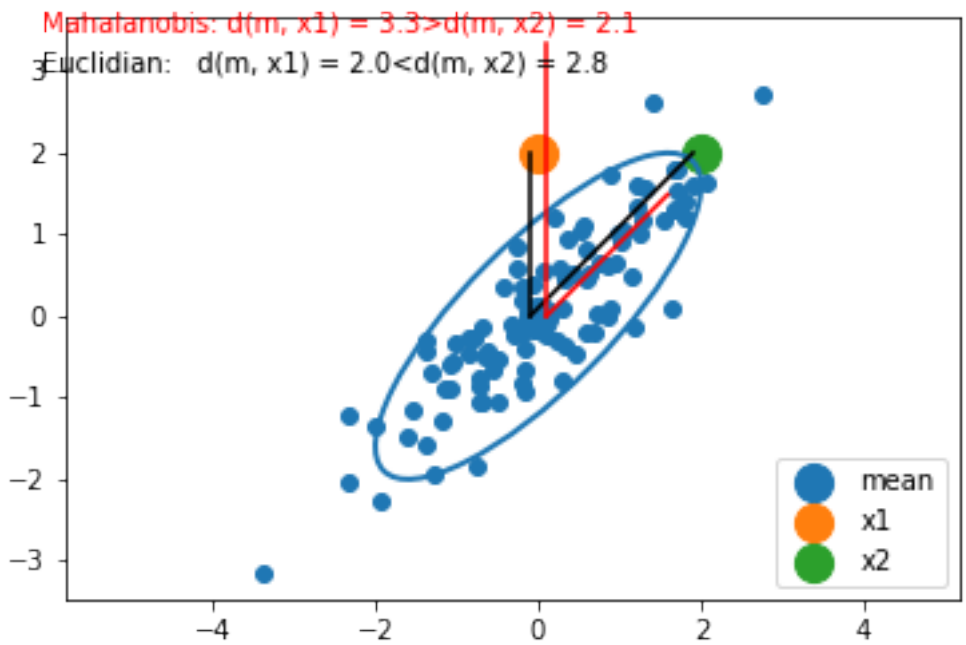
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
%matplotlib inline
np.random.seed(40)
colors = sns.color_palette()

mean = np.array([0, 0])
Cov = np.array([[1, .8],
                [.8, 1]])
samples = np.random.multivariate_normal(mean, Cov, 100)
x1 = np.array([0, 2])
x2 = np.array([2, 2])

plt.scatter(samples[:, 0], samples[:, 1], color=colors[0])
plt.scatter(mean[0], mean[1], color=colors[0], s=200, label="mean")
plt.scatter(x1[0], x1[1], color=colors[1], s=200, label="x1")
plt.scatter(x2[0], x2[1], color=colors[2], s=200, label="x2")

# plot covariance ellipsis
pystatsml.plot_utils.plot_cov_ellipse(Cov, pos=mean, facecolor='none',
                                      linewidth=2, edgecolor=colors[0])
# Compute distances
d2_m_x1 = scipy.spatial.distance.euclidean(mean, x1)
d2_m_x2 = scipy.spatial.distance.euclidean(mean, x2)

Covi = scipy.linalg.inv(Cov)
dm_m_x1 = scipy.spatial.distance.mahalanobis(mean, x1, Covi)
dm_m_x2 = scipy.spatial.distance.mahalanobis(mean, x2, Covi)

# Plot distances
vm_x1 = (x1 - mean) / d2_m_x1
```

(continues on next page)

(continued from previous page)

```
vm_x2 = (x2 - mean) / d2_m_x2
jitter = .1
plt.plot([mean[0] - jitter, d2_m_x1 * vm_x1[0] - jitter],
         [mean[1], d2_m_x1 * vm_x1[1]], color='k')
plt.plot([mean[0] - jitter, d2_m_x2 * vm_x2[0] - jitter],
         [mean[1], d2_m_x2 * vm_x2[1]], color='k')

plt.plot([mean[0] + jitter, dm_m_x1 * vm_x1[0] + jitter],
         [mean[1], dm_m_x1 * vm_x1[1]], color='r')
plt.plot([mean[0] + jitter, dm_m_x2 * vm_x2[0] + jitter],
         [mean[1], dm_m_x2 * vm_x2[1]], color='r')

plt.legend(loc='lower right')
plt.text(-6.1, 3,
         'Euclidian:   d(m, x1) = %.1f<d(m, x2) = %.1f' % (d2_m_x1, d2_m_x2),␣
↪color='k')
plt.text(-6.1, 3.5,
         'Mahalanobis: d(m, x1) = %.1f>d(m, x2) = %.1f' % (dm_m_x1, dm_m_x2),␣
↪color='r')

plt.axis('equal')
print('Euclidian   d(m, x1) = %.2f < d(m, x2) = %.2f' % (d2_m_x1, d2_m_x2))
print('Mahalanobis d(m, x1) = %.2f > d(m, x2) = %.2f' % (dm_m_x1, dm_m_x2))
```

```
Euclidian   d(m, x1) = 2.00 < d(m, x2) = 2.83
Mahalanobis d(m, x1) = 3.33 > d(m, x2) = 2.11
```



If the covariance matrix is the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. If the covariance matrix is diagonal, then the resulting distance measure is called a normalized Euclidean distance.

More generally, the Mahalanobis distance is a measure of the distance between a point $\mathbf{x}$ and a distribution $\mathcal{N}(\mathbf{x}|\mu, \mathbf{\Sigma})$. It is a multi-dimensional generalization of the idea of measuring how many standard deviations away $\mathbf{x}$ is from the mean. This distance is zero if $\mathbf{x}$ is at the mean, and grows as $\mathbf{x}$ moves away from the mean: along each principal component axis, it measures the number of standard deviations from $\mathbf{x}$ to the mean of the distribution.

4.4.7 Multivariate normal distribution

The distribution, or probability density function (PDF) (sometimes just density), of a continuous random variable is a function that describes the relative likelihood for this random variable to take on a given value.

The multivariate normal distribution, or multivariate Gaussian distribution, of a P-dimensional random vector $\mathbf{x} = [x_1, x_2, \ldots, x_P]^T$ is

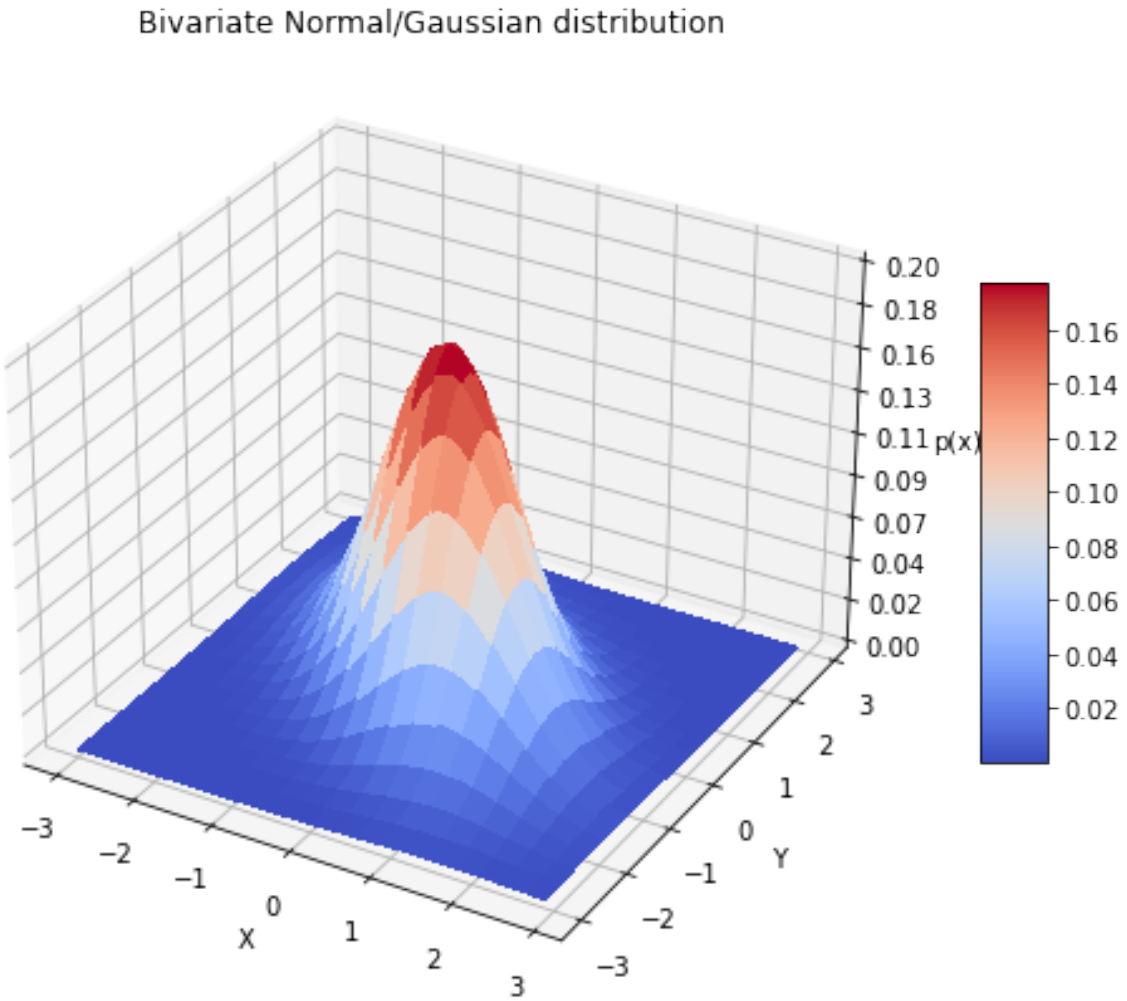
$$\mathcal{N}(\mathbf{x}|\mu, \mathbf{\Sigma}) = \frac{1}{(2\pi)^{P/2}|\mathbf{\Sigma}|^{1/2}} \exp\{-\frac{1}{2}(\mathbf{x}-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\}.$$

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats
from scipy.stats import multivariate_normal
from mpl_toolkits.mplot3d import Axes3D


def multivariate_normal_pdf(X, mean, sigma):
    """Multivariate normal probability density function over X (n_samples x n_
↪features)"""
    P = X.shape[1]
    det = np.linalg.det(sigma)
    norm_const = 1.0 / (((2*np.pi) ** (P/2)) * np.sqrt(det))
    X_mu = X - mu
    inv = np.linalg.inv(sigma)
    d2 = np.sum(np.dot(X_mu, inv) * X_mu, axis=1)
    return norm_const * np.exp(-0.5 * d2)

# mean and covariance
mu = np.array([0, 0])
sigma = np.array([[1, -.5],
                  [-.5, 1]])

# x, y grid
x, y = np.mgrid[-3:3:.1, -3:3:.1]
X = np.stack((x.ravel(), y.ravel())).T
norm = multivariate_normal_pdf(X, mean, sigma).reshape(x.shape)

# Do it with scipy
norm_scpy = multivariate_normal(mu, sigma).pdf(np.stack((x, y), axis=2))
assert np.allclose(norm, norm_scpy)

# Plot
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(10, 7))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, norm, rstride=3,
        cstride=3, cmap=plt.cm.coolwarm,
        linewidth=1, antialiased=False
    )

ax.set_zlim(0, 0.2)
ax.zaxis.set_major_locator(plt.LinearLocator(10))
ax.zaxis.set_major_formatter(plt.FormatStrFormatter('%.02f'))

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('p(x)')

plt.title('Bivariate Normal/Gaussian distribution')
fig.colorbar(surf, shrink=0.5, aspect=7, cmap=plt.cm.coolwarm)
plt.show()
```



4.4.8 Exercises

Dot product and Euclidean norm

Given $\mathbf{a} = [2, 1]^T$ and $\mathbf{b} = [1, 1]^T$

1. Write a function `euclidean(x)` that computes the Euclidean norm of vector, $\mathbf{x}$.
2. Compute the Euclidean norm of $\mathbf{a}$.
3. Compute the Euclidean distance of $\|\mathbf{a} - \mathbf{b}\|_2$.
4. Compute the projection of $\mathbf{b}$ in the direction of vector $\mathbf{a}$: b_a.
5. Simulate a dataset $\mathbf{X}$ of $N = 100$ samples of 2-dimensional vectors.
6. Project all samples in the direction of the vector $\mathbf{a}$.

Covariance matrix and Mahalanobis norm

1. Sample a dataset $\mathbf{X}$ of $N = 100$ samples of 2-dimensional vectors from the bivariate normal distribution $\mathcal{N}(\mu, \mathbf{\Sigma})$ where $\mu = [1, 1]^T$ and $\mathbf{\Sigma} = \begin{bmatrix} 1 & 0.8 \\ 0.8, 1 \end{bmatrix}$.
2. Compute the mean vector $\bar{\mathbf{x}}$ and center $\mathbf{X}$. Compare the estimated mean $\bar{\mathbf{x}}$ to the true mean, μ.
3. Compute the empirical covariance matrix $\mathbf{S}$. Compare the estimated covariance matrix $\mathbf{S}$ to the true covariance matrix, $\mathbf{\Sigma}$.
4. Compute $\mathbf{S}^{-1}$ (`Sinv`) the inverse of the covariance matrix by using `scipy.linalg.inv(S)`.
5. Write a function `mahalanobis(x, xbar, Sinv)` that computes the Mahalanobis distance of a vector $\mathbf{x}$ to the mean, $\bar{\mathbf{x}}$.
6. Compute the Mahalanobis and Euclidean distances of each sample $\mathbf{x}_i$ to the mean $\bar{\mathbf{x}}$. Store the results in a 100×2 dataframe.

4.5 Time series in python

Two libraries:

- Pandas: https://pandas.pydata.org/pandas-docs/stable/timeseries.html
- scipy http://www.statsmodels.org/devel/tsa.html

4.5.1 Stationarity

A TS is said to be stationary if its statistical properties such as mean, variance remain constant over time.

- constant mean
- constant variance
- an autocovariance that does not depend on time.

what is making a TS non-stationary. There are 2 major reasons behind non-stationaruty of a TS:

1. Trend – varying mean over time. For eg, in this case we saw that on average, the number of passengers was growing over time.
2. Seasonality – variations at specific time-frames. eg people might have a tendency to buy cars in a particular month because of pay increment or festivals.

4.5.2 Pandas time series data structure

A Series is similar to a list or an array in Python. It represents a series of values (numeric or otherwise) such as a column of data. It provides additional functionality, methods, and operators, which make it a more powerful version of a list.

```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Create a Series from a list
ser = pd.Series([1, 3])
print(ser)

# String as index
prices = {'apple': 4.99,
          'banana': 1.99,
          'orange': 3.99}
ser = pd.Series(prices)
print(ser)

x = pd.Series(np.arange(1,3), index=[x for x in 'ab'])
print(x)
print(x['b'])
```

```
0    1
1    3
dtype: int64
apple     4.99
```

(continues on next page)

(continued from previous page)

```
banana    1.99
orange    3.99
dtype: float64
a    1
b    2
dtype: int64
2
```

4.5.3 Time series analysis of Google trends

source: https://www.datacamp.com/community/tutorials/time-series-analysis-tutorial

Get Google Trends data of keywords such as 'diet' and 'gym' and see how they vary over time while learning about trends and seasonality in time series data.

In the Facebook Live code along session on the 4th of January, we checked out Google trends data of keywords 'diet', 'gym' and 'finance' to see how they vary over time. We asked ourselves if there could be more searches for these terms in January when we're all trying to turn over a new leaf?

In this tutorial, you'll go through the code that we put together during the session step by step. You're not going to do much mathematics but you are going to do the following:

- Read data
- Recode data
- Exploratory Data Analysis

4.5.4 Read data

```
try:
    url = "https://raw.githubusercontent.com/datacamp/datacamp_facebook_live_ny_
↪resolution/master/datasets/multiTimeline.csv"
    df = pd.read_csv(url, skiprows=2)
except:
    df = pd.read_csv("../datasets/multiTimeline.csv", skiprows=2)

print(df.head())

# Rename columns
df.columns = ['month', 'diet', 'gym', 'finance']

# Describe
print(df.describe())
```

```
     Month  diet: (Worldwide)  gym: (Worldwide)  finance: (Worldwide)
0  2004-01                100                31                    48
1  2004-02                 75                26                    49
2  2004-03                 67                24                    47
```

(continues on next page)

(continued from previous page)

```
3  2004-04                70                22                    48
4  2004-05                72                22                    43
             diet         gym     finance
count  168.000000  168.000000  168.000000
mean    49.642857   34.690476   47.148810
std      8.033080    8.134316    4.972547
min     34.000000   22.000000   38.000000
25%     44.000000   28.000000   44.000000
50%     48.500000   32.500000   46.000000
75%     53.000000   41.000000   50.000000
max    100.000000   58.000000   73.000000
```

4.5.5 Recode data

Next, you'll turn the 'month' column into a DateTime data type and make it the index of the DataFrame.

Note that you do this because you saw in the result of the .info() method that the 'Month' column was actually an of data type object. Now, that generic data type encapsulates everything from strings to integers, etc. That's not exactly what you want when you want to be looking at time series data. That's why you'll use .to_datetime() to convert the 'month' column in your DataFrame to a DateTime.

Be careful! Make sure to include the inplace argument when you're setting the index of the DataFrame df so that you actually alter the original index and set it to the 'month' column.

```
df.month = pd.to_datetime(df.month)
df.set_index('month', inplace=True)

print(df.head())
```

```
            diet  gym  finance
month
2004-01-01   100   31       48
2004-02-01    75   26       49
2004-03-01    67   24       47
2004-04-01    70   22       48
2004-05-01    72   22       43
```

4.5.6 Exploratory data analysis

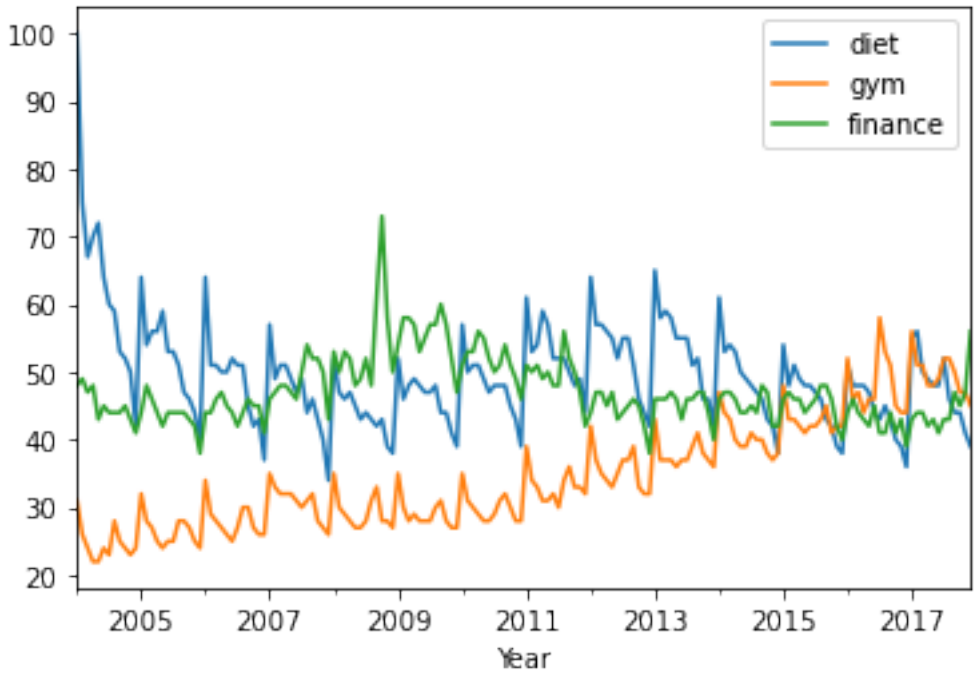
You can use a built-in pandas visualization method .plot() to plot your data as 3 line plots on a single figure (one for each column, namely, 'diet', 'gym', and 'finance').

```
df.plot()
plt.xlabel('Year');

# change figure parameters
```

(continues on next page)

(continued from previous page)

```
# df.plot(figsize=(20,10), linewidth=5, fontsize=20)

# Plot single column
# df[['diet']].plot(figsize=(20,10), linewidth=5, fontsize=20)
# plt.xlabel('Year', fontsize=20);
```



Note that this data is relative. As you can read on Google trends:

Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. Likewise a score of 0 means the term was less than 1% as popular as the peak.

4.5.7 Resampling, smoothing, windowing, rolling average: trends

Rolling average, for each time point, take the average of the points on either side of it. Note that the number of points is specified by a window size.

Remove Seasonality with pandas Series.

See: http://pandas.pydata.org/pandas-docs/stable/timeseries.html A: 'year end frequency' year frequency

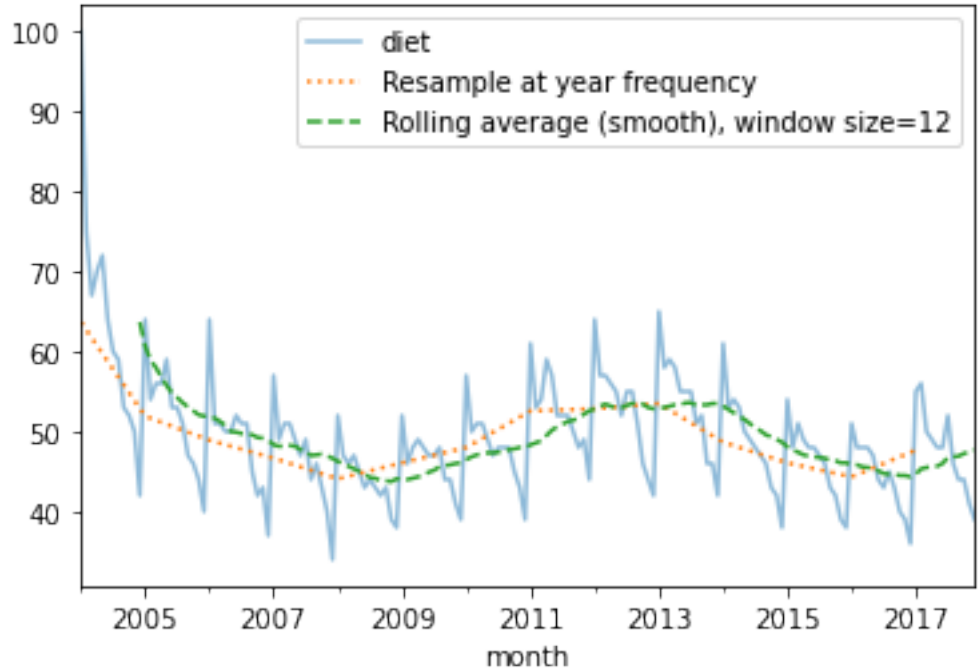
```
diet = df['diet']

diet_resamp_yr = diet.resample('A').mean()
diet_roll_yr = diet.rolling(12).mean()

ax = diet.plot(alpha=0.5, style='-') # store axis (ax) for latter plots
diet_resamp_yr.plot(style=':', label='Resample at year frequency', ax=ax)
```

(continues on next page)

(continued from previous page)

```
diet_roll_yr.plot(style='--', label='Rolling average (smooth), window size=12',␣
↪ax=ax)
ax.legend()
```

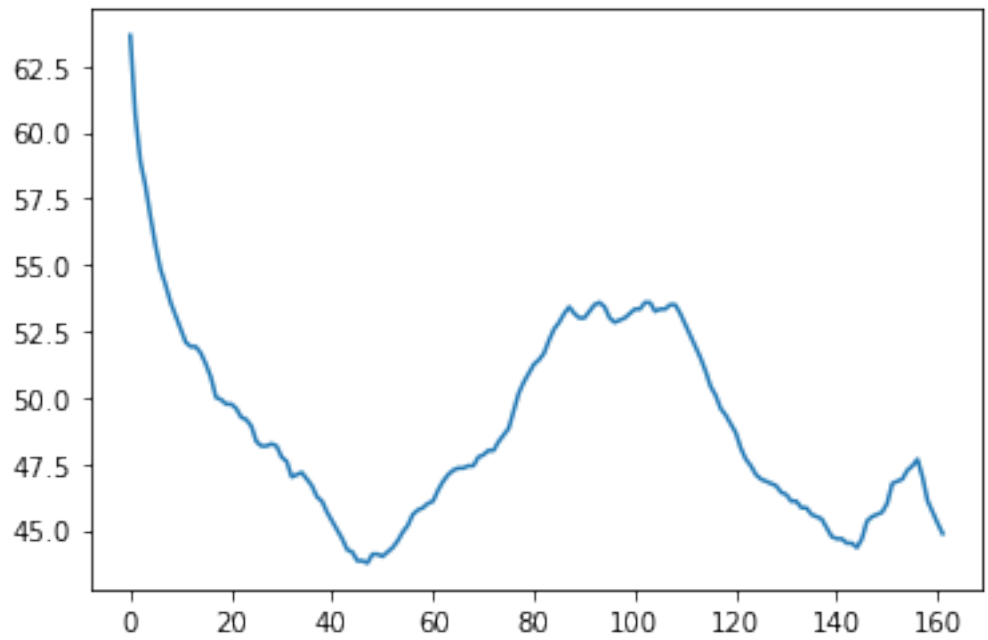
```
<matplotlib.legend.Legend at 0x7f670db34a10>
```



Rolling average (smoothing) with Numpy

```
x = np.asarray(df[['diet']])
win = 12
win_half = int(win / 2)
# print([((idx-win_half), (idx+win_half)) for idx in np.arange(win_half, len(x))])

diet_smooth = np.array([x[(idx-win_half):(idx+win_half)].mean() for idx in np.
↪arange(win_half, len(x))])
plt.plot(diet_smooth)
```

```
[<matplotlib.lines.Line2D at 0x7f670cbbb1d0>]
```

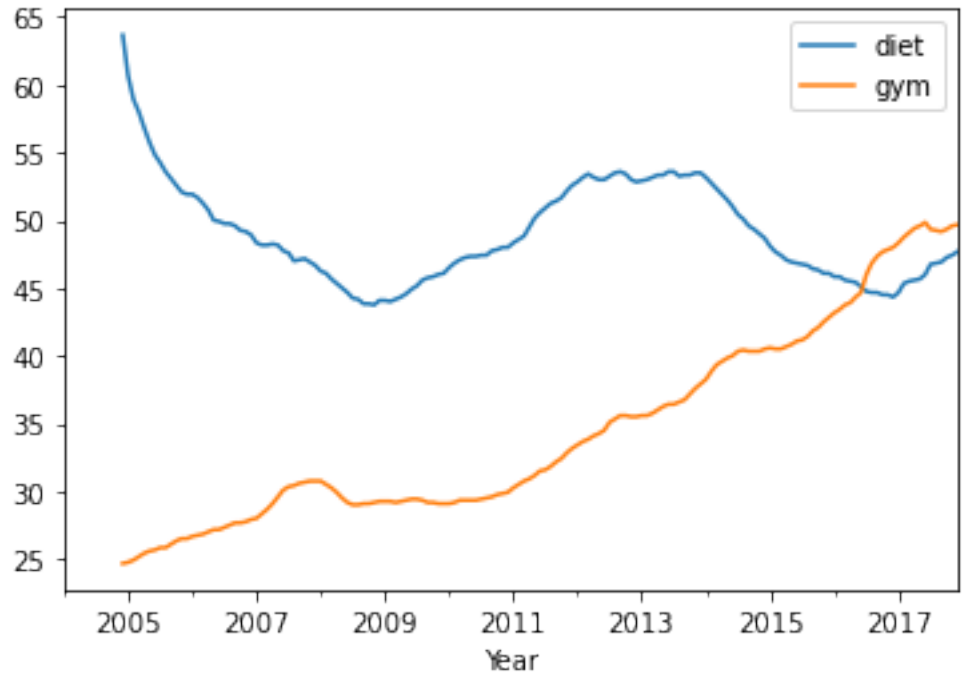


Trends Plot Diet and Gym

Build a new DataFrame which is the concatenation diet and gym smoothed data

```
gym = df['gym']

df_avg = pd.concat([diet.rolling(12).mean(), gym.rolling(12).mean()], axis=1)
df_avg.plot()
plt.xlabel('Year')
```

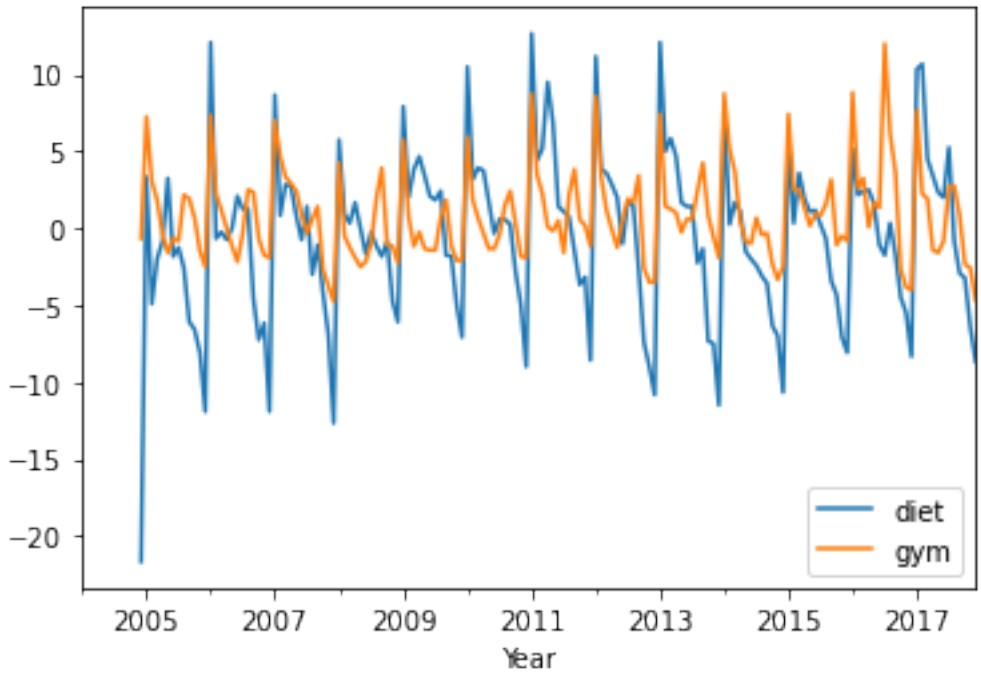
```
Text(0.5, 0, 'Year')
```



Detrending

```
df_dtrend = df[["diet", "gym"]] - df_avg
df_dtrend.plot()
plt.xlabel('Year')
```

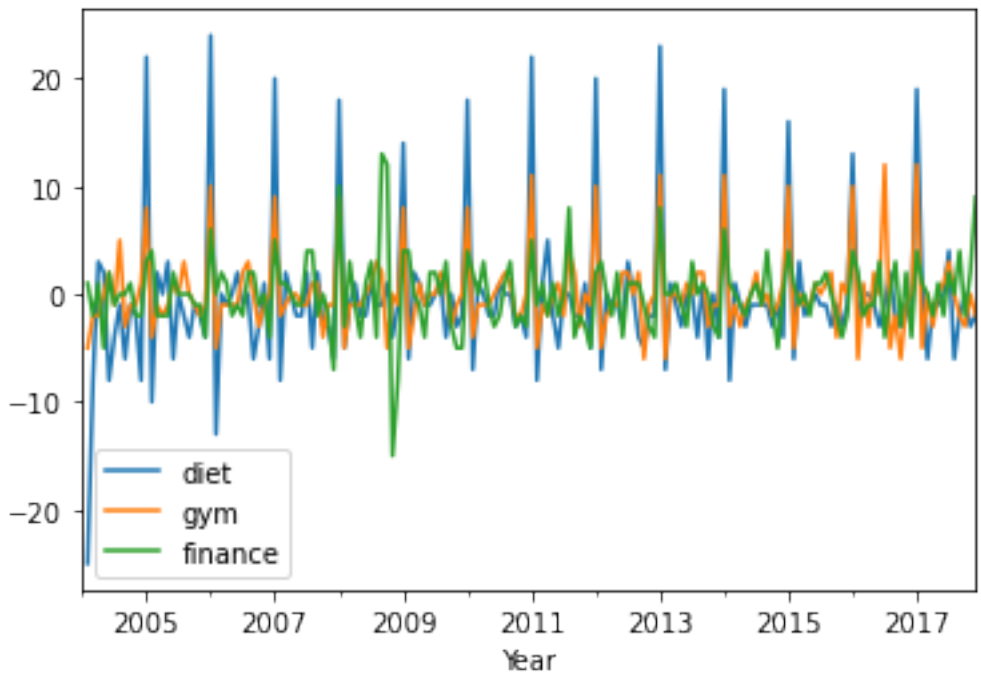
```
Text(0.5, 0, 'Year')
```



4.5.8 First-order differencing: seasonal patterns

```
# diff = original - shiftted data
# (exclude first term for some implementation details)
assert np.all((diet.diff() == diet - diet.shift())[1:])

df.diff().plot()
plt.xlabel('Year')
```

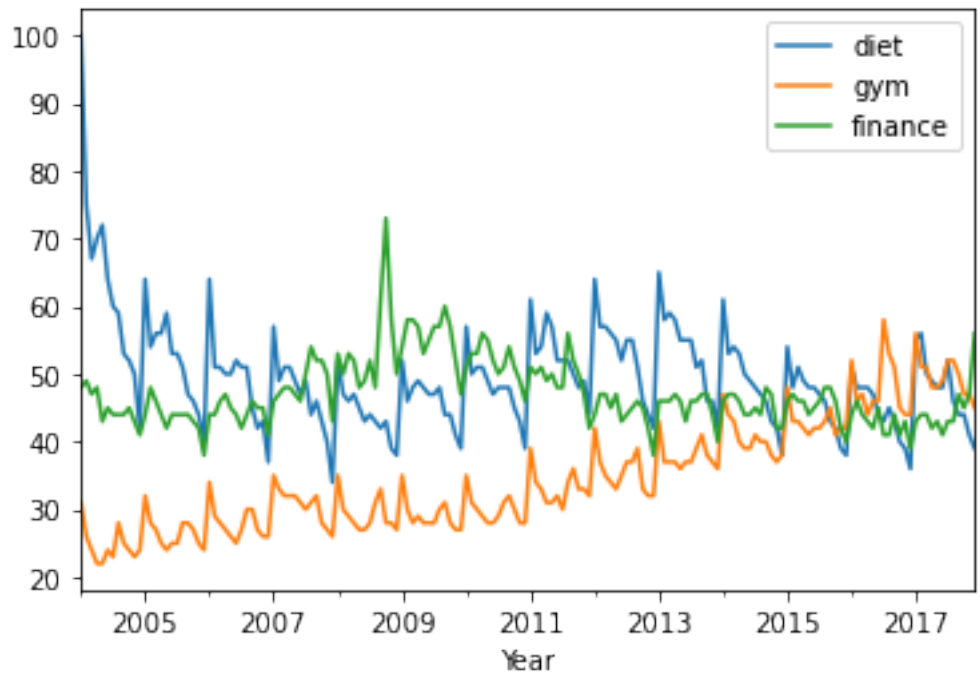
```
Text(0.5, 0, 'Year')
```



4.5.9 Periodicity and correlation

```
df.plot()
plt.xlabel('Year');
print(df.corr())
```

```
             diet       gym   finance
diet     1.000000 -0.100764 -0.034639
gym     -0.100764  1.000000 -0.284279
finance -0.034639 -0.284279  1.000000
```



Plot correlation matrix

```
print(df.corr())
```

```
             diet       gym   finance
diet     1.000000 -0.100764 -0.034639
gym     -0.100764  1.000000 -0.284279
finance -0.034639 -0.284279  1.000000
```

'diet' and 'gym' are negatively correlated! Remember that you have a seasonal and a trend component. From the correlation coefficient, 'diet' and 'gym' are negatively correlated:

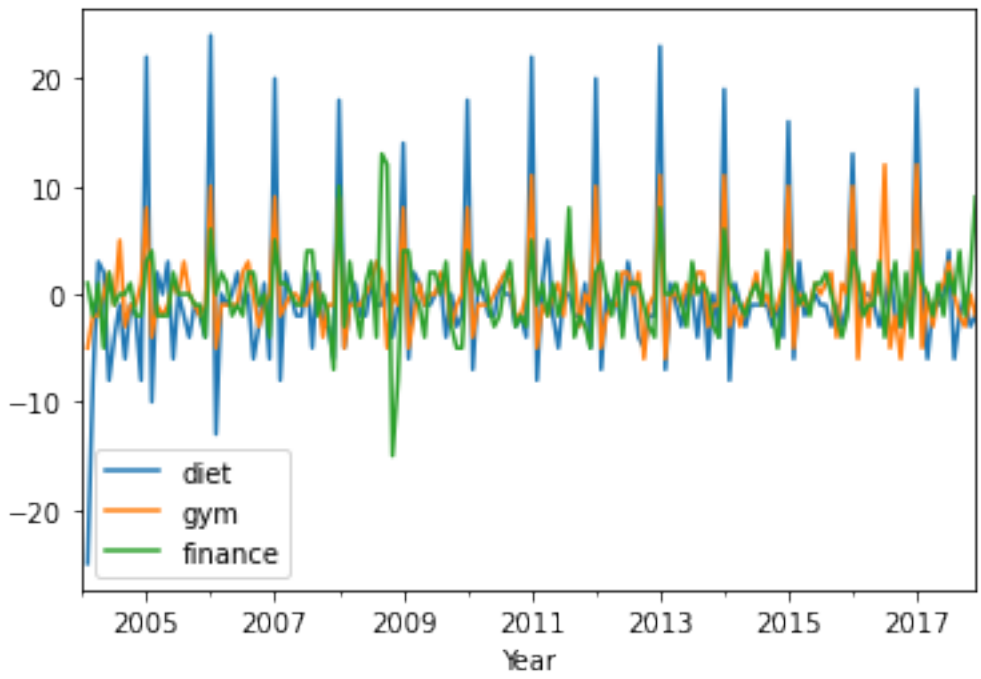
- trends components are negatively correlated.
- seasonal components would positively correlated and their

The actual correlation coefficient is actually capturing both of those.

Seasonal correlation: correlation of the first-order differences of these time series

```
df.diff().plot()
plt.xlabel('Year');

print(df.diff().corr())
```

```
             diet       gym   finance
diet     1.000000  0.758707  0.373828
gym      0.758707  1.000000  0.301111
finance  0.373828  0.301111  1.000000
```



Plot correlation matrix

```
print(df.diff().corr())
```

```
             diet       gym   finance
diet     1.000000  0.758707  0.373828
gym      0.758707  1.000000  0.301111
finance  0.373828  0.301111  1.000000
```

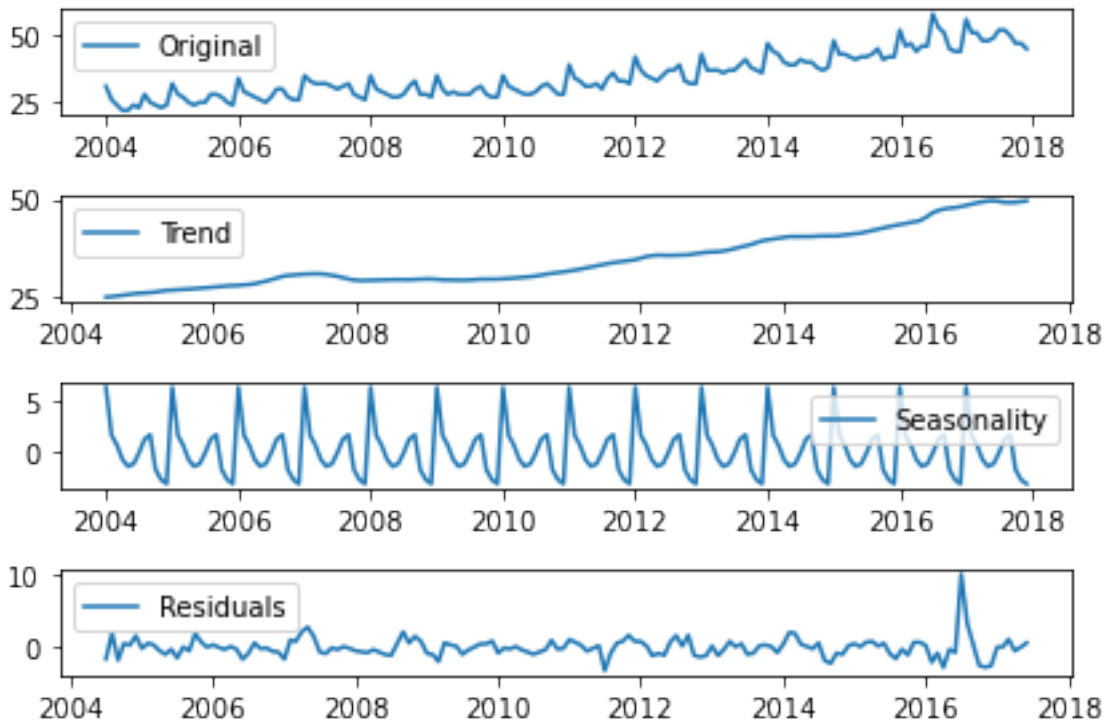
Decomposing time serie in trend, seasonality and residuals

```
from statsmodels.tsa.seasonal import seasonal_decompose

x = gym

x = x.astype(float) # force float
decomposition = seasonal_decompose(x)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(x, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
```

(continues on next page)

(continued from previous page)

```
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



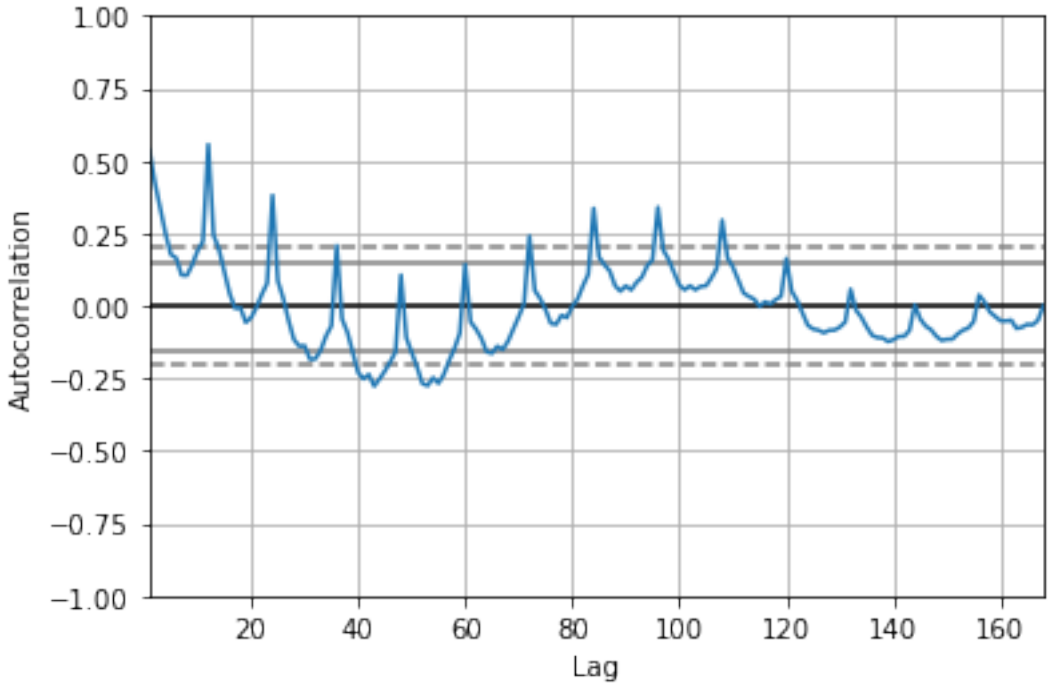
4.5.10 Autocorrelation

A time series is periodic if it repeats itself at equally spaced intervals, say, every 12 months. Autocorrelation Function (ACF): It is a measure of the correlation between the TS with a lagged version of itself. For instance at lag 5, ACF would compare series at time instant t1…t2 with series at instant t1-5…t2-5 (t1-5 and t2 being end points).

Plot

```
# from pandas.plotting import autocorrelation_plot
from pandas.plotting import autocorrelation_plot

x = df["diet"].astype(float)
autocorrelation_plot(x)
```

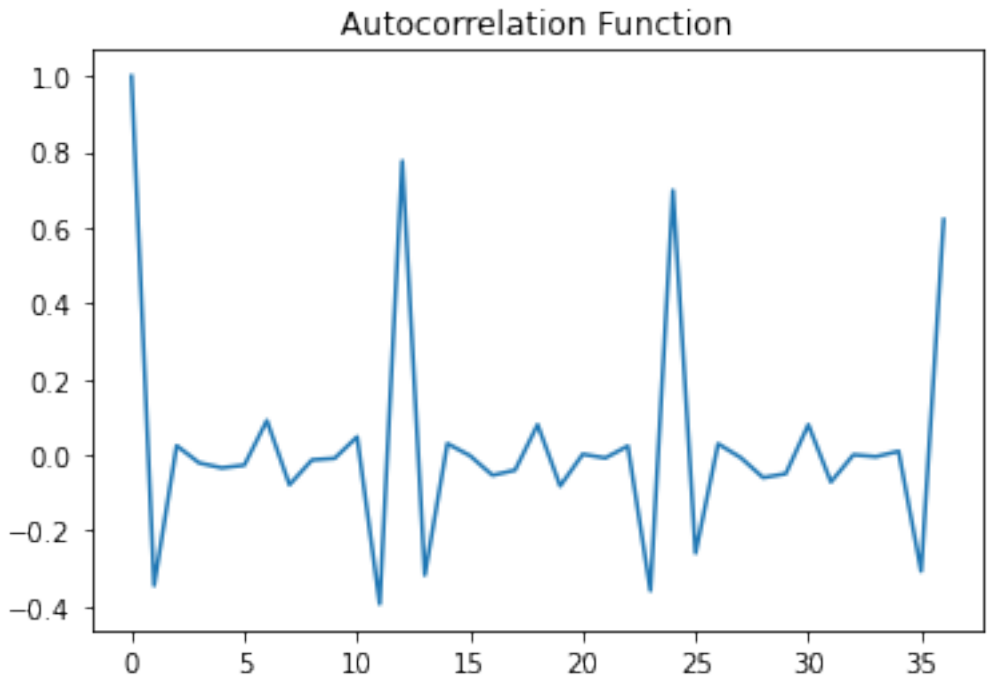
```
<AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



Compute Autocorrelation Function (ACF)

```
from statsmodels.tsa.stattools import acf

x_diff = x.diff().dropna() # first item is NA
lag_acf = acf(x_diff, nlags=36, fft=True)
plt.plot(lag_acf)
plt.title('Autocorrelation Function')
```

```
Text(0.5, 1.0, 'Autocorrelation Function')
```



ACF peaks every 12 months: Time series is correlated with itself shifted by 12 months.

4.5.11 Time series forecasting with Python using Autoregressive Moving Average (ARMA) models

Source:

- https://www.packtpub.com/mapt/book/big_data_and_business_intelligence/9781783553358/7/ch07lvl1sec77/arma-models
- http://en.wikipedia.org/wiki/Autoregressive%E2%80%93moving-average_model
- ARIMA: https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/

ARMA models are often used to forecast a time series. These models combine autoregressive and moving average models. In moving average models, we assume that a variable is the sum of the mean of the time series and a linear combination of noise components.

The autoregressive and moving average models can have different orders. In general, we can define an ARMA model with p autoregressive terms and q moving average terms as follows:

$$x_t = \sum_i^p a_i x_{t-i} + \sum_i^q b_i \varepsilon_{t-i} + \varepsilon_t$$

Choosing p and q

Plot the partial autocorrelation functions for an estimate of p, and likewise using the autocorrelation functions for an estimate of q.

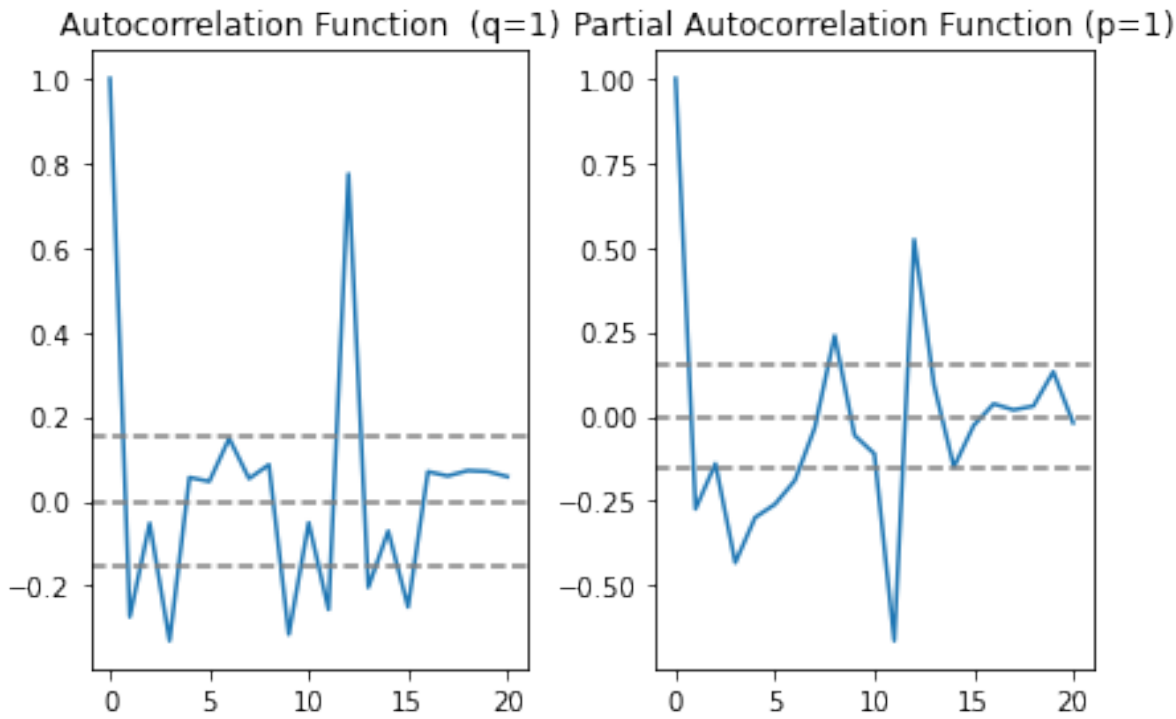
Partial Autocorrelation Function (PACF): This measures the correlation between the TS with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons. Eg at lag 5, it will check the correlation but remove the effects already explained by lags 1 to 4.

```
from statsmodels.tsa.stattools import acf, pacf

x = df["gym"].astype(float)

x_diff = x.diff().dropna() # first item is NA
# ACF and PACF plots:

lag_acf = acf(x_diff, nlags=20, fft=True)
lag_pacf = pacf(x_diff, nlags=20, method='ols')

#Plot ACF:
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.title('Autocorrelation Function  (q=1)')
```

(continues on next page)

(continued from previous page)

```
#Plot PACF:
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.title('Partial Autocorrelation Function (p=1)')
plt.tight_layout()
```



In this plot, the two dotted lines on either sides of 0 are the confidence interevals. These can be used to determine the p and q values as:

- p: The lag value where the PACF chart crosses the upper confidence interval for the first time, in this case p=1.
- q: The lag value where the ACF chart crosses the upper confidence interval for the first time, in this case q=1.

Fit ARMA model with statsmodels

1. Define the model by calling `ARMA()` and passing in the p and q parameters.
2. The model is prepared on the training data by calling the `fit()` function.
3. Predictions can be made by calling the `predict()` function and specifying the index of the time or times to be predicted.

```
from statsmodels.tsa.arima_model import ARMA
# from statsmodels.tsa.arima.model import ARIMA
```

(continues on next page)

(continued from previous page)

```
model = ARMA(x, order=(1, 1)).fit() # fit model

print(model.summary())
plt.plot(x)
plt.plot(model.predict(), color='red')
plt.title('RSS: %.4f'% sum((model.fittedvalues-x)**2))
```

```
/home/ed203246/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/arima_model.
↪py:472: FutureWarning:
statsmodels.tsa.arima_model.ARMA and statsmodels.tsa.arima_model.ARIMA have
been deprecated in favor of statsmodels.tsa.arima.model.ARIMA (note the .
between arima and model) and
statsmodels.tsa.SARIMAX. These will be removed after the 0.12 release.

statsmodels.tsa.arima.model.ARIMA makes use of the statespace framework and
is both well tested and maintained.

To silence this warning and continue using ARMA and ARIMA until they are
removed, use:

import warnings
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARMA',
                        FutureWarning)
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARIMA',
                        FutureWarning)

  warnings.warn(ARIMA_DEPRECATION_WARN, FutureWarning)
/home/ed203246/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_
↪model.py:527: ValueWarning: No frequency information was provided, so inferred␣
↪frequency MS will be used.
  % freq, ValueWarning)
```

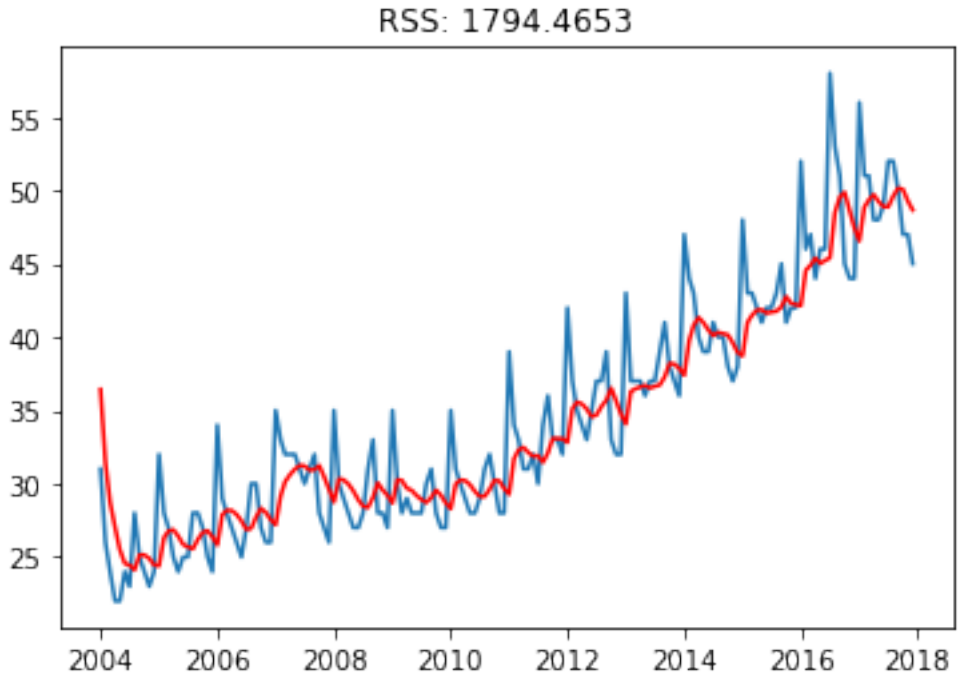
```
                              ARMA Model Results
==============================================================================
Dep. Variable:                    gym   No. Observations:                  168
Model:                     ARMA(1, 1)   Log Likelihood                -436.852
Method:                       css-mle   S.D. of innovations              3.229
Date:                Fri, 04 Dec 2020   AIC                            881.704
Time:                        13:05:20   BIC                            894.200
Sample:                    01-01-2004   HQIC                           886.776
                         - 12-01-2017
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         36.4315      8.827      4.127      0.000      19.131      53.732
ar.L1.gym      0.9967      0.005    220.566      0.000       0.988       1.006
ma.L1.gym     -0.7494      0.054    -13.931      0.000      -0.855      -0.644
                                    Roots
```

(continues on next page)

(continued from previous page)

```
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            1.0033           +0.0000j            1.0033            0.0000
MA.1            1.3344           +0.0000j            1.3344            0.0000
-----------------------------------------------------------------------------
```

```
Text(0.5, 1.0, 'RSS: 1794.4653')
```



CHAPTER

FIVE

MACHINE LEARNING

5.1 Linear dimension reduction and feature extraction

5.1.1 Introduction

In machine learning and statistics, dimensionality reduction or dimension reduction is the process of reducing the number of features under consideration, and can be divided into feature selection (not addressed here) and feature extraction.

Feature extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. Feature extraction is related to dimensionality reduction.

The input matrix $\mathbf{X}$, of dimension $N \times P$, is

$$\begin{bmatrix} x_{11} & \dots & x_{1P} \\ \vdots & \mathbf{X} & \vdots \\ x_{N1} & \dots & x_{NP} \end{bmatrix}$$

where the rows represent the samples and columns represent the variables. The goal is to learn a transformation that extracts a few relevant features.

Models:

1. Linear matrix decomposition/factorisation SVD/PCA. Those models exploit the covariance $\mathbf{\Sigma_{XX}}$ between the input features.
2. Non-linear models based on manifold learning: Isomap, t-SNE. Those models

5.1.2 Singular value decomposition and matrix factorization

Matrix factorization principles

Decompose the data matrix $\mathbf{X}_{N\times P}$ into a product of a mixing matrix $\mathbf{U}_{N\times K}$ and a dictionary matrix $\mathbf{V}_{P\times K}$.

$$\mathbf{X} = \mathbf{U}\mathbf{V}^T,$$

If we consider only a subset of components $K < rank(\mathbf{X}) < \min(P, N-1)$, $\mathbf{X}$ is approximated by a matrix $\hat{\mathbf{X}}$:

$$\mathbf{X} \approx \hat{\mathbf{X}} = \mathbf{U}\mathbf{V}^T,$$

Each line of $\mathbf{x_i}$ is a linear combination (mixing $\mathbf{u_i}$) of dictionary items $\mathbf{V}$.

N P-dimensional data points lie in a space whose dimension is less than $N-1$ (2 dots lie on a line, 3 on a plane, etc.).

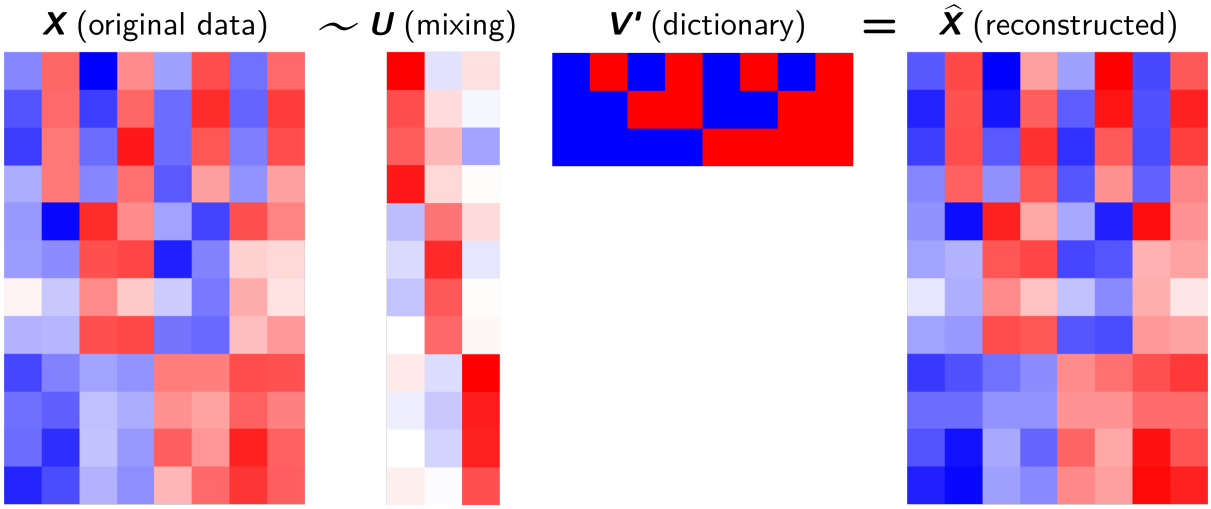


Fig. 1: Matrix factorization

Singular value decomposition (SVD) principles

Singular-value decomposition (SVD) factorises the data matrix $\mathbf{X}_{N\times P}$ into a product:

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T,$$

where

$$\begin{bmatrix} x_{11} & & x_{1P} \\ & \mathbf{X} & \\ x_{N1} & & x_{NP} \end{bmatrix} = \begin{bmatrix} u_{11} & & u_{1K} \\ & \mathbf{U} & \\ u_{N1} & & u_{NK} \end{bmatrix} \begin{bmatrix} d_1 & & 0 \\ & \mathbf{D} & \\ 0 & & d_K \end{bmatrix} \begin{bmatrix} v_{11} & & v_{1P} \\ & \mathbf{V}^T & \\ v_{K1} & & v_{KP} \end{bmatrix}.$$

$\mathbf{U}$: **right-singular**

- $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_K]$ is a $P \times K$ orthogonal matrix.
- It is a **dictionary** of patterns to be combined (according to the mixing coefficients) to reconstruct the original samples.
- $\mathbf{V}$ perfoms the initial **rotations** (**projection**) along the $K = \min(N, P)$ **principal component directions**, also called **loadings**.
- Each $\mathbf{v}_j$ performs the linear combination of the variables that has maximum sample variance, subject to being uncorrelated with the previous $\mathbf{v}_{j-1}$.

$\mathbf{D}$: **singular values**

- $\mathbf{D}$ is a $K \times K$ diagonal matrix made of the singular values of $\mathbf{X}$ with $d_1 \geq d_2 \geq \cdots \geq d_K \geq 0$.
- $\mathbf{D}$ scale the projection along the coordinate axes by $d_1, d_2, \cdots, d_K$.
- Singular values are the square roots of the eigenvalues of $\mathbf{X}^T\mathbf{X}$.

$\mathbf{V}$: **left-singular vectors**

- $\mathbf{U} = [\mathbf{u}_1, \cdots, \mathbf{u}_K]$ is an $N \times K$ orthogonal matrix.
- Each row $\mathbf{v_i}$ provides the **mixing coefficients** of dictionary items to reconstruct the sample $\mathbf{x_i}$
- It may be understood as the coordinates on the new orthogonal basis (obtained after the initial rotation) called **principal components** in the PCA.

SVD for variables transformation

$\mathbf{V}$ transforms correlated variables ($\mathbf{X}$) into a set of uncorrelated ones ($\mathbf{UD}$) that better expose the various relationships among the original data items.

$$\mathbf{X} = \mathbf{UDV}^T, \tag{5.1}$$
$$\mathbf{XV} = \mathbf{UDV}^T\mathbf{V}, \tag{5.2}$$
$$\mathbf{XV} = \mathbf{UDI}, \tag{5.3}$$
$$\mathbf{XV} = \mathbf{UD} \tag{5.4}$$

At the same time, SVD is a method for identifying and ordering the dimensions along which data points exhibit the most variation.

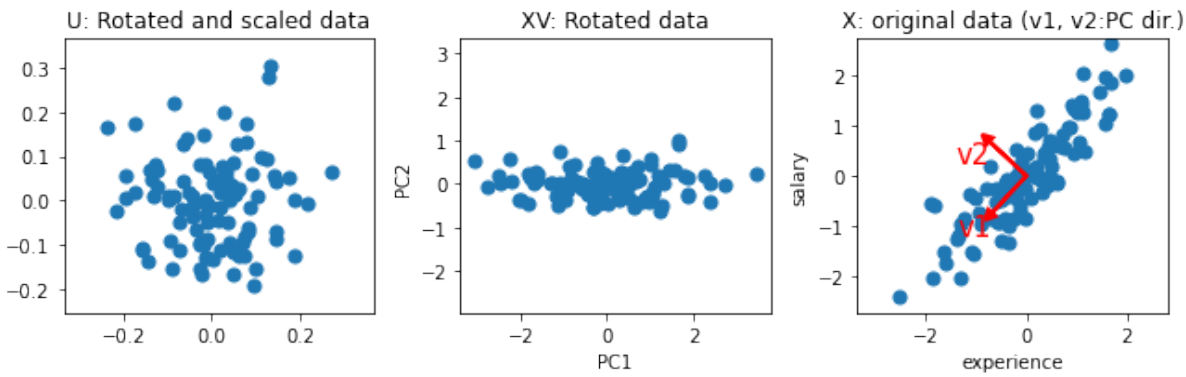
```
import numpy as np
import scipy
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

np.random.seed(42)

# dataset
n_samples = 100
experience = np.random.normal(size=n_samples)
salary = 1500 + experience + np.random.normal(size=n_samples, scale=.5)
X = np.column_stack([experience, salary])
print(X.shape)

# PCA using SVD
X -= X.mean(axis=0)  # Centering is required
U, s, Vh = scipy.linalg.svd(X, full_matrices=False)
# U : Unitary matrix having left singular vectors as columns.
#     Of shape (n_samples,n_samples) or (n_samples,n_comps), depending on
```

(continues on next page)

(continued from previous page)

```
#     full_matrices.
#
# s : The singular values, sorted in non-increasing order. Of shape (n_comps,),
#     with n_comps = min(n_samples, n_features).
#
# Vh: Unitary matrix having right singular vectors as rows.
#     Of shape (n_features, n_features) or (n_comps, n_features) depending
# on full_matrices.

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.scatter(U[:, 0], U[:, 1], s=50)
plt.axis('equal')
plt.title("U: Rotated and scaled data")

plt.subplot(132)

# Project data
PC = np.dot(X, Vh.T)
plt.scatter(PC[:, 0], PC[:, 1], s=50)
plt.axis('equal')
plt.title("XV: Rotated data")
plt.xlabel("PC1")
plt.ylabel("PC2")

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], s=50)
for i in range(Vh.shape[0]):
    plt.arrow(x=0, y=0, dx=Vh[i, 0], dy=Vh[i, 1], head_width=0.2,
              head_length=0.2, linewidth=2, fc='r', ec='r')
    plt.text(Vh[i, 0], Vh[i, 1],'v%i' % (i+1), color="r", fontsize=15,
             horizontalalignment='right', verticalalignment='top')
plt.axis('equal')
plt.ylim(-4, 4)

plt.title("X: original data (v1, v2:PC dir.)")
plt.xlabel("experience")
plt.ylabel("salary")

plt.tight_layout()
```

```
(100, 2)
```



5.1.3 Principal components analysis (PCA)

Sources:

- C. M. Bishop *Pattern Recognition and Machine Learning*, Springer, 2006
- Everything you did and didn't know about PCA
- Principal Component Analysis in 3 Simple Steps

Principles

- Principal components analysis is the main method used for linear dimension reduction.
- The idea of principal component analysis is to find the K **principal components directions** (called the **loadings**) $\mathbf{V}_{K\times P}$ that capture the variation in the data as much as possible.
- It converts a set of N P-dimensional observations $\mathbf{N}_{N\times P}$ of possibly correlated variables into a set of N K-dimensional samples $\mathbf{C}_{N\times K}$, where the $K < P$. The new variables are linearly uncorrelated. The columns of $\mathbf{C}_{N\times K}$ are called the **principal components**.
- The dimension reduction is obtained by using only $K < P$ components that exploit correlation (covariance) among the original variables.
- PCA is mathematically defined as an orthogonal linear transformation $\mathbf{V}_{K\times P}$ that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

$$\mathbf{C}_{N\times K} = \mathbf{X}_{N\times P}\mathbf{V}_{P\times K}$$

- PCA can be thought of as fitting a P-dimensional ellipsoid to the data, where each axis of the ellipsoid represents a principal component. If some axis of the ellipse is small, then the variance along that axis is also small, and by omitting that axis and its corresponding principal component from our representation of the dataset, we lose only a commensurately small amount of information.
- Finding the K largest axes of the ellipse will permit to project the data onto a space having dimensionality $K < P$ while maximizing the variance of the projected data.

Dataset preprocessing

Centering

Consider a data matrix, $\mathbf{X}$, with column-wise zero empirical mean (the sample mean of each column has been shifted to zero), ie. $\mathbf{X}$ is replaced by $\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T$.

Standardizing

Optionally, standardize the columns, i.e., scale them by their standard-deviation. Without standardization, a variable with a high variance will capture most of the effect of the PCA. The principal direction will be aligned with this variable. Standardization will, however, raise noise variables to the save level as informative variables.

The covariance matrix of centered standardized data is the correlation matrix.

Eigendecomposition of the data covariance matrix

To begin with, consider the projection onto a one-dimensional space ($K = 1$). We can define the direction of this space using a P-dimensional vector $\mathbf{v}$, which for convenience (and without loss of generality) we shall choose to be a unit vector so that $\|\mathbf{v}\|_2 = 1$ (note that we are only interested in the direction defined by $\mathbf{v}$, not in the magnitude of $\mathbf{v}$ itself). PCA consists of two mains steps:

Projection in the directions that capture the greatest variance

Each P-dimensional data point $\mathbf{x}_i$ is then projected onto $\mathbf{v}$, where the coordinate (in the coordinate system of $\mathbf{v}$) is a scalar value, namely $\mathbf{x}_i^T\mathbf{v}$. I.e., we want to find the vector $\mathbf{v}$ that maximizes these coordinates along $\mathbf{v}$, which we will see corresponds to maximizing the variance of the projected data. This is equivalently expressed as

$$\mathbf{v} = \arg\max_{\|\mathbf{v}\|=1} \frac{1}{N} \sum_i \left(\mathbf{x}_i^T\mathbf{v}\right)^2.$$

We can write this in matrix form as

$$\mathbf{v} = \arg\max_{\|\mathbf{v}\|=1} \frac{1}{N}\|\mathbf{X}\mathbf{v}\|^2 = \frac{1}{N}\mathbf{v}^T\mathbf{X}^T\mathbf{X}\mathbf{v} = \mathbf{v}^T\mathbf{S}_{\mathbf{XX}}\mathbf{v},$$

where $\mathbf{S}_{\mathbf{XX}}$ is a biased estiamte of the covariance matrix of the data, i.e.

$$\mathbf{S}_{\mathbf{XX}} = \frac{1}{N}\mathbf{X}^T\mathbf{X}.$$

We now maximize the projected variance $\mathbf{v}^T\mathbf{S}_{\mathbf{XX}}\mathbf{v}$ with respect to $\mathbf{v}$. Clearly, this has to be a constrained maximization to prevent $\|\mathbf{v}_2\| \to \infty$. The appropriate constraint comes from the normalization condition $\|\mathbf{v}\|_2 \equiv \|\mathbf{v}\|_2^2 = \mathbf{v}^T\mathbf{v} = 1$. To enforce this constraint, we introduce a Lagrange multiplier that we shall denote by λ, and then make an unconstrained maximization of

$$\mathbf{v}^T\mathbf{S}_{\mathbf{XX}}\mathbf{v} - \lambda(\mathbf{v}^T\mathbf{v} - 1).$$

By setting the gradient with respect to $\mathbf{v}$ equal to zero, we see that this quantity has a stationary point when

$$\mathbf{S_{XX}v} = \lambda\mathbf{v}.$$

We note that $\mathbf{v}$ is an eigenvector of $\mathbf{S_{XX}}$.

If we left-multiply the above equation by $\mathbf{v}^T$ and make use of $\mathbf{v}^T\mathbf{v} = 1$, we see that the variance is given by

$$\mathbf{v}^T\mathbf{S_{XX}v} = \lambda,$$

and so the variance will be at a maximum when $\mathbf{v}$ is equal to the eigenvector corresponding to the largest eigenvalue, λ. This eigenvector is known as the first principal component.

We can define additional principal components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance amongst all possible directions that are orthogonal to those already considered. If we consider the general case of a K-dimensional projection space, the optimal linear projection for which the variance of the projected data is maximized is now defined by the K eigenvectors, $\mathbf{v_1}, \ldots, \mathbf{v_K}$, of the data covariance matrix $\mathbf{S_{XX}}$ that corresponds to the K largest eigenvalues, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_K$.

Back to SVD

The sample covariance matrix of **centered data** $\mathbf{X}$ is given by

$$\mathbf{S_{XX}} = \frac{1}{N-1}\mathbf{X}^T\mathbf{X}.$$

We rewrite $\mathbf{X}^T\mathbf{X}$ using the SVD decomposition of $\mathbf{X}$ as

$$\begin{aligned}
\mathbf{X}^T\mathbf{X} &= (\mathbf{UDV}^T)^T(\mathbf{UDV}^T) \\
&= \mathbf{VD}^T\mathbf{U}^T\mathbf{UDV}^T \\
&= \mathbf{VD}^2\mathbf{V}^T \\
\mathbf{V}^T\mathbf{X}^T\mathbf{XV} &= \mathbf{D}^2 \\
\frac{1}{N-1}\mathbf{V}^T\mathbf{X}^T\mathbf{XV} &= \frac{1}{N-1}\mathbf{D}^2 \\
\mathbf{V}^T\mathbf{S_{XX}V} &= \frac{1}{N-1}\mathbf{D}^2
\end{aligned}$$

.

Considering only the kth right-singular vectors $\mathbf{v}_k$ associated to the singular value d_k

$$\mathbf{v_k}^T\mathbf{S_{XX}v_k} = \frac{1}{N-1}d_k^2,$$

It turns out that if you have done the singular value decomposition then you already have the Eigenvalue decomposition for $\mathbf{X}^T\mathbf{X}$. Where - The eigenvectors of $\mathbf{S_{XX}}$ are equivalent to the right singular vectors, $\mathbf{V}$, of $\mathbf{X}$. - The eigenvalues, λ_k, of $\mathbf{S_{XX}}$, i.e. the variances of the components, are equal to $\frac{1}{N-1}$ times the squared singular values, d_k.

Moreover computing PCA with SVD do not require to form the matrix $\mathbf{X}^T\mathbf{X}$, so computing the SVD is now the standard way to calculate a principal components analysis from a data matrix, unless only a handful of components are required.

PCA outputs

The SVD or the eigendecomposition of the data covariance matrix provides three main quantities:

1. **Principal component directions** or **loadings** are the **eigenvectors** of $\mathbf{X}^T\mathbf{X}$. The $\mathbf{V}_{K\times P}$ or the **right-singular vectors** of an SVD of $\mathbf{X}$ are called principal component directions of $\mathbf{X}$. They are generally computed using the SVD of $\mathbf{X}$.

2. **Principal components** is the $N \times K$ matrix $\mathbf{C}$ which is obtained by projecting $\mathbf{X}$ onto the principal components directions, i.e.

$$\mathbf{C}_{N\times K} = \mathbf{X}_{N\times P}\mathbf{V}_{P\times K}.$$

Since $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$ and $\mathbf{V}$ is orthogonal ($\mathbf{V}^T\mathbf{V} = \mathbf{I}$):

$$\begin{aligned}
\mathbf{C}_{N\times K} &= \mathbf{U}\mathbf{D}\mathbf{V}^T_{N\times P}\mathbf{V}_{P\times K} && (5.5)\\
\mathbf{C}_{N\times K} &= \mathbf{U}\mathbf{D}^T_{N\times K}\mathbf{I}_{K\times K} && (5.6)\\
\mathbf{C}_{N\times K} &= \mathbf{U}\mathbf{D}^T_{N\times K} && (5.7)\\
& && (5.8)
\end{aligned}$$

Thus $\mathbf{c}_j = \mathbf{X}\mathbf{v}_j = \mathbf{u}_j d_j$, for $j = 1, \ldots K$. Hence $\mathbf{u}_j$ is simply the projection of the row vectors of $\mathbf{X}$, i.e., the input predictor vectors, on the direction $\mathbf{v}_j$, scaled by d_j.

$$\mathbf{c}_1 = \begin{bmatrix} x_{1,1}v_{1,1} + \ldots + x_{1,P}v_{1,P} \\ x_{2,1}v_{1,1} + \ldots + x_{2,P}v_{1,P} \\ \vdots \\ x_{N,1}v_{1,1} + \ldots + x_{N,P}v_{1,P} \end{bmatrix}$$

3. The **variance** of each component is given by the eigen values $\lambda_k, k = 1, \ldots K$. It can be obtained from the singular values:

$$\begin{aligned}
var(\mathbf{c}_k) &= \frac{1}{N-1}(\mathbf{X}\mathbf{v}_k)^2 && (5.9)\\
&= \frac{1}{N-1}(\mathbf{u}_k d_k)^2 && (5.10)\\
&= \frac{1}{N-1}d_k^2 && (5.11)
\end{aligned}$$

Determining the number of PCs

We must choose $K^* \in [1, \ldots, K]$, the number of required components. This can be done by calculating the explained variance ratio of the K^* first components and by choosing K^* such that the **cumulative explained variance** ratio is greater than some given threshold (e.g., $\approx 90\%$). This is expressed as

$$\text{cumulative explained variance}(\mathbf{c}_k) = \frac{\sum_j^{K^*} var(\mathbf{c}_k)}{\sum_j^{K} var(\mathbf{c}_k)}.$$

Interpretation and visualization

PCs

Plot the samples projeted on first the principal components as e.g. PC1 against PC2.

PC directions

Exploring the loadings associated with a component provides the contribution of each original variable in the component.

Remark: The loadings (PC directions) are the coefficients of multiple regression of PC on original variables:

$$\mathbf{c} = \mathbf{X}\mathbf{v} \tag{5.12}$$

$$\mathbf{X}^T\mathbf{c} = \mathbf{X}^T\mathbf{X}\mathbf{v} \tag{5.13}$$

$$(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{c} = \mathbf{v} \tag{5.14}$$

Another way to evaluate the contribution of the original variables in each PC can be obtained by computing the correlation between the PCs and the original variables, i.e. columns of $\mathbf{X}$, denoted $\mathbf{x}_j$, for $j = 1, \ldots, P$. For the kth PC, compute and plot the correlations with all original variables

$$cor(\mathbf{c}_k, \mathbf{x}_j), j = 1 \ldots K, j = 1 \ldots K.$$

These quantities are sometimes called the *correlation loadings*.

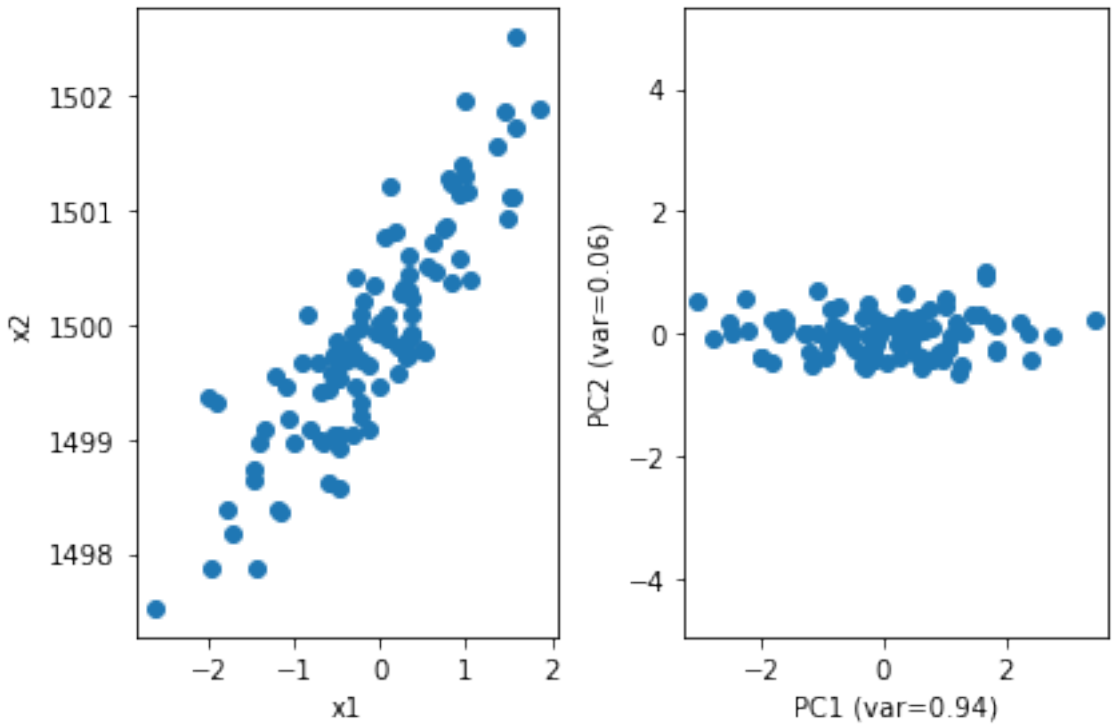
```
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

np.random.seed(42)

# dataset
n_samples = 100
experience = np.random.normal(size=n_samples)
salary = 1500 + experience + np.random.normal(size=n_samples, scale=.5)
X = np.column_stack([experience, salary])

# PCA with scikit-learn
pca = PCA(n_components=2)
pca.fit(X)
print(pca.explained_variance_ratio_)

PC = pca.transform(X)

plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1])
plt.xlabel("x1"); plt.ylabel("x2")

plt.subplot(122)
```

(continues on next page)

(continued from previous page)

```
plt.scatter(PC[:, 0], PC[:, 1])
plt.xlabel("PC1 (var=%.2f)" % pca.explained_variance_ratio_[0])
plt.ylabel("PC2 (var=%.2f)" % pca.explained_variance_ratio_[1])
plt.axis('equal')
plt.tight_layout()
```

```
[0.93646607 0.06353393]
```




```
from time import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import (manifold, datasets, decomposition, ensemble,
                     discriminant_analysis, random_projection, neighbors)
print(__doc__)

digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_neighbors = 30
```

```
Automatically created module for IPython interactive environment
```

5.1.4 Eigen faces

Sources: Scikit learn Faces decompositions

Load data

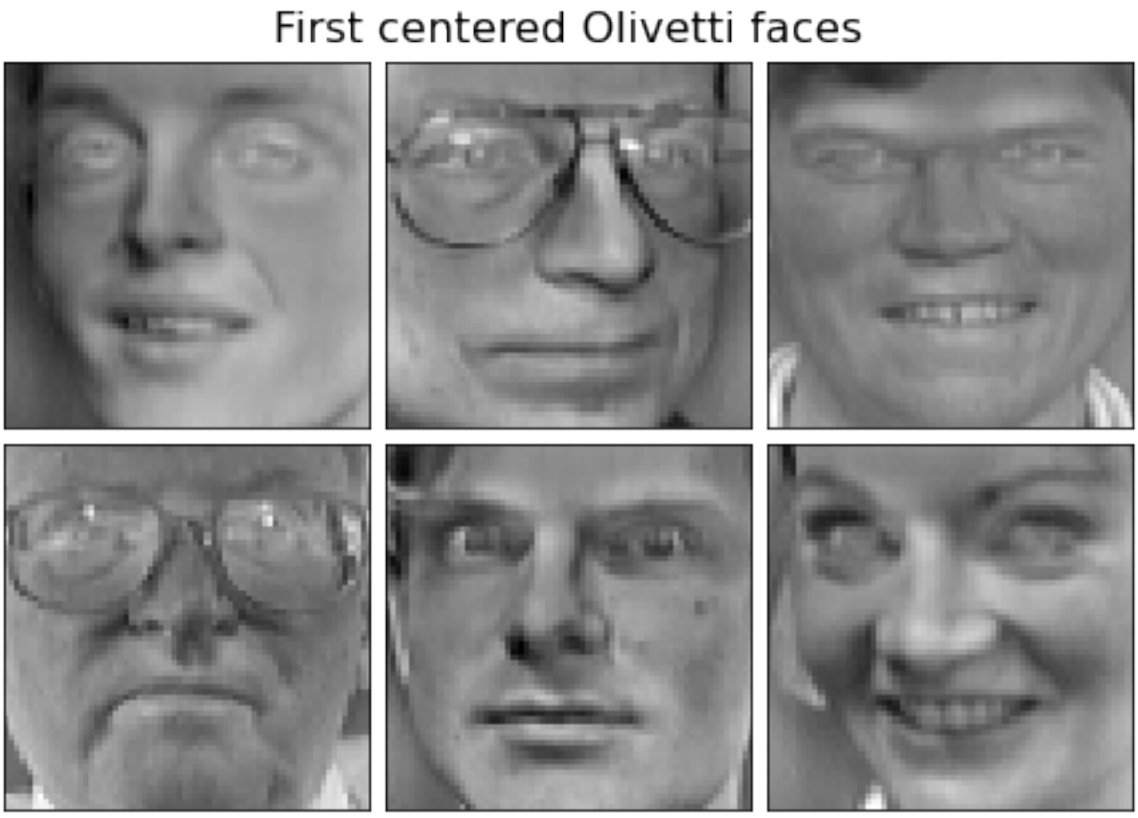
```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces
from sklearn import decomposition

n_row, n_col = 2, 3
n_components = n_row * n_col
image_shape = (64, 64)

faces, _ = fetch_olivetti_faces(return_X_y=True, shuffle=True,
                                random_state=1)
n_samples, n_features = faces.shape


# Utils function
def plot_gallery(title, images, n_col=n_col, n_row=n_row, cmap=plt.cm.gray):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=cmap,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)
```

Preprocessing

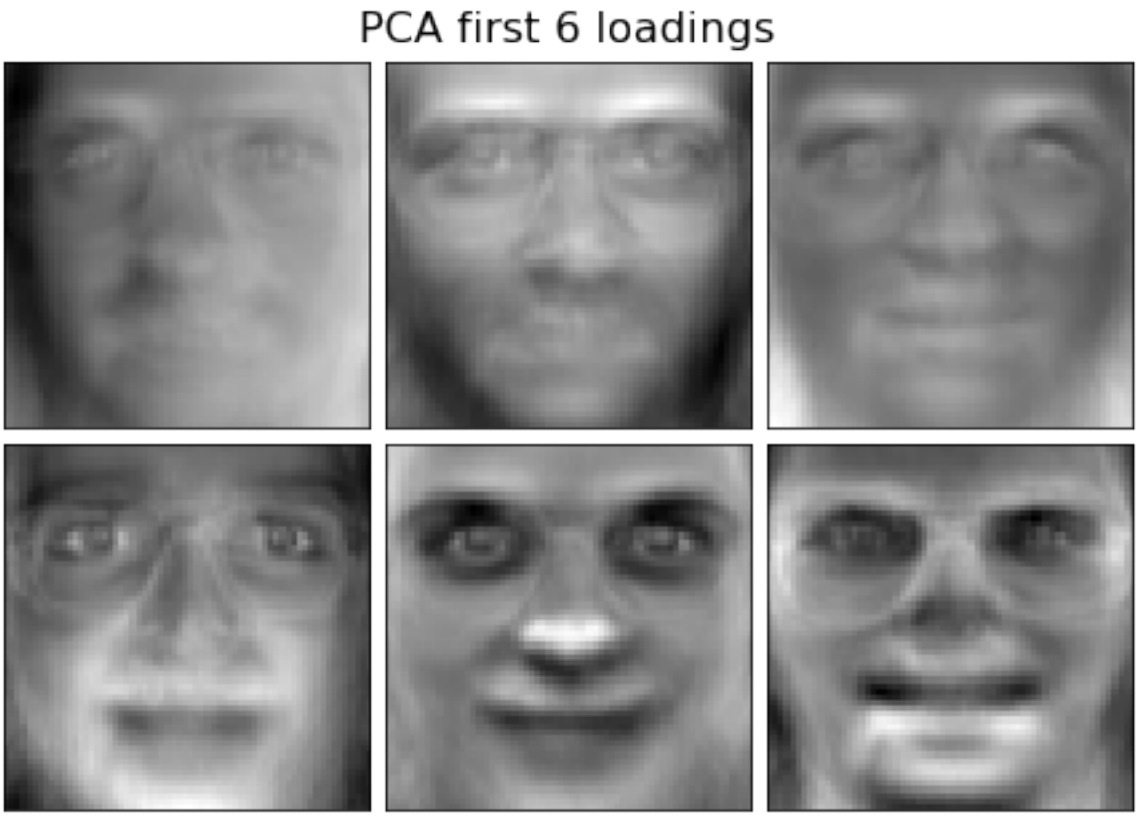
```
# global centering
faces_centered = faces - faces.mean(axis=0)

# local centering
faces_centered -= faces_centered.mean(axis=1).reshape(n_samples, -1)
```

First centered Olivetti faces

```
plot_gallery("First centered Olivetti faces", faces_centered[:n_components])
```




```
pca = decomposition.PCA(n_components=n_components)
pca.fit(faces_centered)
plot_gallery("PCA first %i loadings" % n_components, pca.components_[:n_
↪components])
```



5.1.5 Exercises

Write a basic PCA class

Write a class `BasicPCA` with two methods:

- `fit(X)` that estimates the data mean, principal components directions $\mathbf{V}$ and the explained variance of each component.
- `transform(X)` that projects the data onto the principal components.

Check that your `BasicPCA` gave similar results, compared to the results from `sklearn`.

Apply your Basic PCA on the `iris` dataset

The data set is available at: https://github.com/duchesnay/pystatsml/raw/master/datasets/iris.csv

- Describe the data set. Should the dataset been standardized?
- Describe the structure of correlations among variables.
- Compute a PCA with the maximum number of components.
- Compute the cumulative explained variance ratio. Determine the number of components K by your computed values.

- Print the K principal components directions and correlations of the K principal components with the original variables. Interpret the contribution of the original variables into the PC.
- Plot the samples projected into the K first PCs.
- Color samples by their species.

Run scikit-learn examples

Load the notebook or python file at the end of each examples

- Faces dataset decompositions
- Faces recognition example using eigenfaces and SVMs

5.2 Manifold learning: non-linear dimension reduction

Sources:

- Scikit-learn documentation
- Wikipedia

Nonlinear dimensionality reduction or **manifold learning** cover unsupervised methods that attempt to identify low-dimensional manifolds within the original P-dimensional space that represent high data density. Then those methods provide a mapping from the high-dimensional space to the low-dimensional embedding.

5.2.1 Multi-dimensional Scaling (MDS)

Resources:

- http://www.stat.pitt.edu/sungkyu/course/2221Fall13/lec8_mds_combined.pdf
- https://en.wikipedia.org/wiki/Multidimensional_scaling
- Hastie, Tibshirani and Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* New York: Springer, Second Edition.

The purpose of MDS is to find a low-dimensional projection of the data in which the pairwise distances between data points is preserved, as closely as possible (in a least-squares sense).

- Let $\mathbf{D}$ be the $(N \times N)$ pairwise distance matrix where d_{ij} is *a distance* between points i and j.
- The MDS concept can be extended to a wide variety of data types specified in terms of a similarity matrix.

Given the dissimilarity (distance) matrix $\mathbf{D}_{N\times N} = [d_{ij}]$, MDS attempts to find K-dimensional projections of the N points $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^K$, concatenated in an $\mathbf{X}_{N\times K}$ matrix, so that $d_{ij} \approx \|\mathbf{x}_i - \mathbf{x}_j\|$ are as close as possible. This can be obtained by the minimization of a loss function called the **stress function**

$$\text{stress}(\mathbf{X}) = \sum_{i\neq j} (d_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2 .$$

This loss function is known as *least-squares* or *Kruskal-Shepard* scaling.

A modification of *least-squares* scaling is the *Sammon mapping*

$$\text{stress}_{\text{Sammon}}(\mathbf{X}) = \sum_{i \neq j} \frac{(d_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2}{d_{ij}}.$$

The Sammon mapping performs better at preserving small distances compared to the *least-squares* scaling.

Classical multidimensional scaling

Also known as *principal coordinates analysis*, PCoA.

- The distance matrix, $\mathbf{D}$, is transformed to a *similarity matrix*, $\mathbf{B}$, often using centered inner products.
- The loss function becomes
$$\text{stress}_{\text{classical}}(\mathbf{X}) = \sum_{i \neq j} \left(b_{ij} - \langle \mathbf{x}_i, \mathbf{x}_j \rangle\right)^2.$$
- The stress function in classical MDS is sometimes called *strain*.
- The solution for the classical MDS problems can be found from the eigenvectors of the similarity matrix.
- If the distances in $\mathbf{D}$ are Euclidean and double centered inner products are used, the results are equivalent to PCA.

Example

The `eurodist` datset provides the road distances (in kilometers) between 21 cities in Europe. Given this matrix of pairwise (non-Euclidean) distances $\mathbf{D} = [d_{ij}]$, MDS can be used to recover the coordinates of the cities in *some* Euclidean referential whose orientation is arbitrary.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Pairwise distance between European cities
try:
    url = '../datasets/eurodist.csv'
    df = pd.read_csv(url)
except:
    url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/eurodist.csv
↪'
    df = pd.read_csv(url)

print(df.iloc[:5, :5])

city = df["city"]
D = np.array(df.iloc[:, 1:])  # Distance matrix
```

(continues on next page)

(continued from previous page)

```
# Arbitrary choice of K=2 components
from sklearn.manifold import MDS
mds = MDS(dissimilarity='precomputed', n_components=2, random_state=40, max_
↪iter=3000, eps=1e-9)
X = mds.fit_transform(D)
```

```
        city  Athens  Barcelona  Brussels  Calais
0     Athens       0       3313      2963    3175
1  Barcelona    3313          0      1318    1326
2   Brussels    2963       1318         0     204
3     Calais    3175       1326       204       0
4  Cherbourg    3339       1294       583     460
```

Recover coordinates of the cities in Euclidean referential whose orientation is arbitrary:

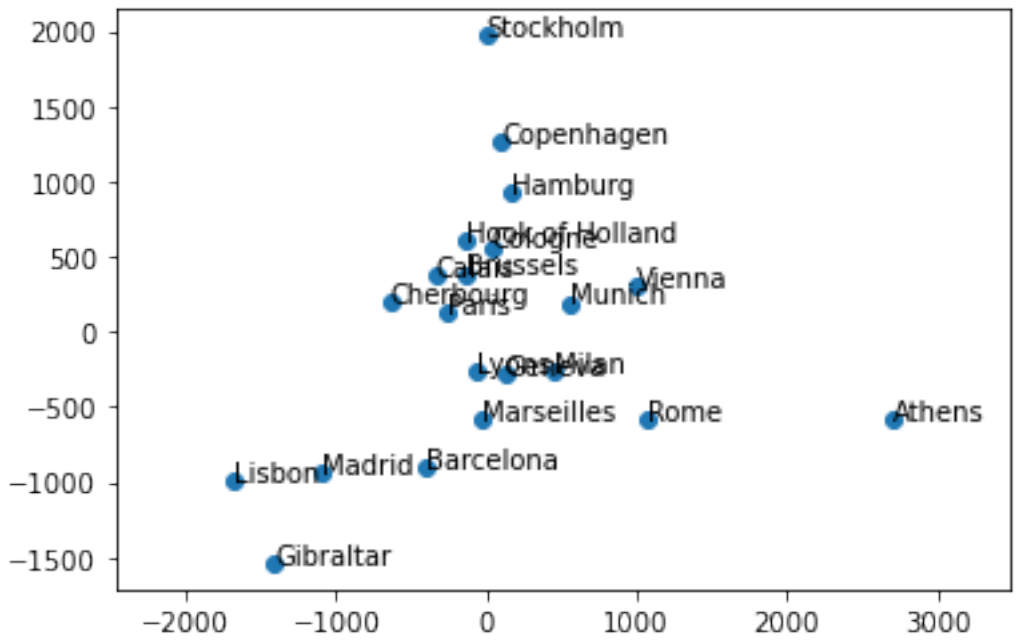
```
from sklearn import metrics
Deuclidean = metrics.pairwise.pairwise_distances(X, metric='euclidean')
print(np.round(Deuclidean[:5, :5]))
```

```
[[   0. 3116. 2994. 3181. 3428.]
 [3116.    0. 1317. 1289. 1128.]
 [2994. 1317.    0.  198.  538.]
 [3181. 1289.  198.    0.  358.]
 [3428. 1128.  538.  358.    0.]]
```

Plot the results:

```
# Plot: apply some rotation and flip
theta = 80 * np.pi / 180.
rot = np.array([[np.cos(theta), -np.sin(theta)],
                [np.sin(theta),  np.cos(theta)]])
Xr = np.dot(X, rot)
# flip x
Xr[:, 0] *= -1
plt.scatter(Xr[:, 0], Xr[:, 1])

for i in range(len(city)):
    plt.text(Xr[i, 0], Xr[i, 1], city[i])
plt.axis('equal')
```

```
(-1894.0919178069155,
 2914.3554370871234,
 -1712.9733697197494,
 2145.437068788015)
```



Determining the number of components

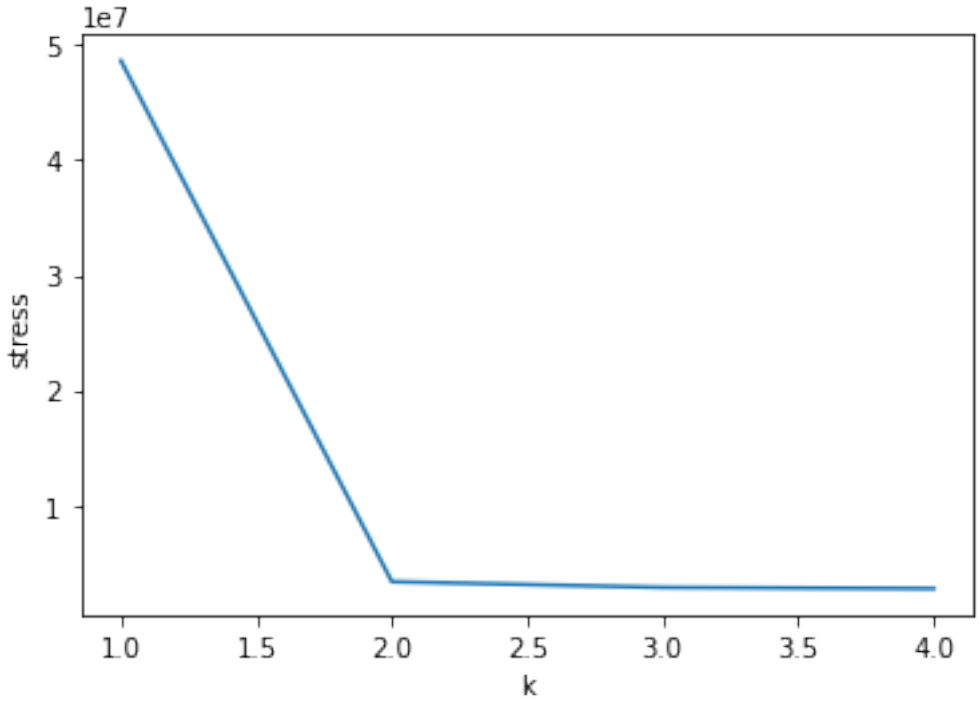
We must choose $K^* \in \{1, \ldots, K\}$ the number of required components. Plotting the values of the stress function, obtained using $k \leq N - 1$ components. In general, start with $1, \ldots K \leq 4$. Choose K^* where you can clearly distinguish an *elbow* in the stress curve.

Thus, in the plot below, we choose to retain information accounted for by the first *two* components, since this is where the *elbow* is in the stress curve.

```
k_range = range(1, min(5, D.shape[0]-1))
stress = [MDS(dissimilarity='precomputed', n_components=k,
           random_state=42, max_iter=300, eps=1e-9).fit(D).stress_ for k in k_
↪range]

print(stress)
plt.plot(k_range, stress)
plt.xlabel("k")
plt.ylabel("stress")
```

```
[48644495.28571428, 3356497.365752386, 2858455.495887962, 2756310.637628011]
```

```
Text(0, 0.5, 'stress')
```



Exercises

Apply MDS from `sklearn` on the `iris` dataset available at:

https://github.com/duchesnay/pystatsml/raw/master/datasets/iris.csv

- Center and scale the dataset.
- Compute Euclidean pairwise distances matrix.
- Select the number of components.
- Show that classical MDS on Euclidean pairwise distances matrix is equivalent to PCA.

Manifold learning

Dataset S curve:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import manifold, datasets

X, color = datasets.make_s_curve(1000, random_state=42)
```

5.2.2 Isomap

Isomap is a nonlinear dimensionality reduction method that combines a procedure to compute the distance matrix with MDS. The distances calculation is based on geodesic distances evaluated on neighborhood graph:

1. Determine the neighbors of each point. All points in some fixed radius or K nearest neighbors.
2. Construct a neighborhood graph. Each point is connected to other if it is a K nearest neighbor. Edge length equal to Euclidean distance.
3. Compute shortest path between pairwise of points d_{ij} to build the distance matrix $\mathbf{D}$.
4. Apply MDS on $\mathbf{D}$.

```
isomap = manifold.Isomap(n_neighbors=10, n_components=2)
X_isomap = isomap.fit_transform(X)
```

5.2.3 t-SNE

Sources:

- Wikipedia
- scikit-learn

Principles

1. Construct a (Gaussian) probability distribution between pairs of object in input (high-dimensional) space.
2. Construct a (student)) probability distribution between pairs of object in embeded (low-dimensional) space.
3. Minimize the Kullback–Leibler divergence (KL divergence) between the two distributions.

Features

- Isomap, LLE and variants are best suited to unfold a single continuous low dimensional manifold
- t-SNE will focus on the **local structure** of the data and will tend to extract clustered **local groups** of samples

```
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
X_tsne = tsne.fit_transform(X)
```

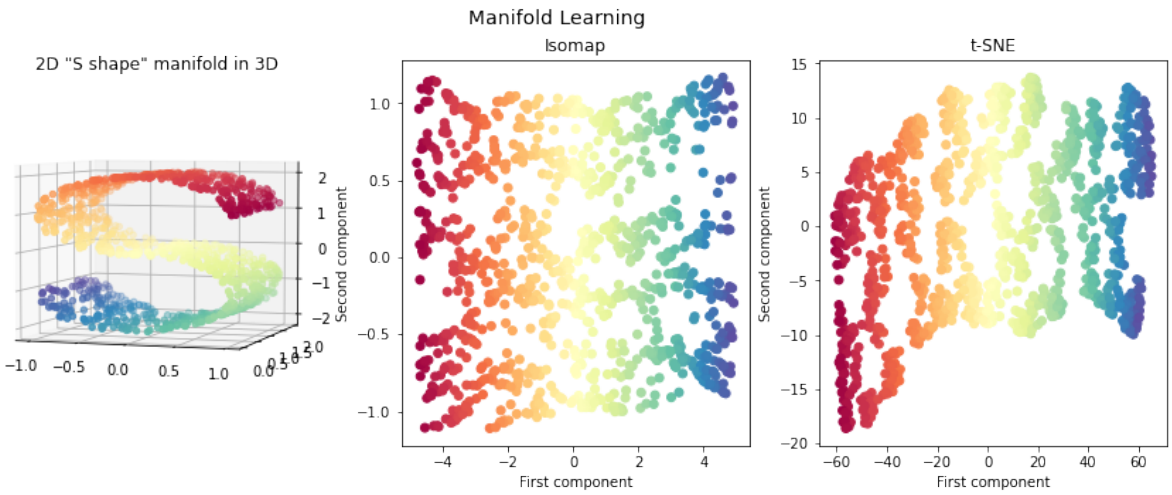
```
fig = plt.figure(figsize=(15, 5))
plt.suptitle("Manifold Learning", fontsize=14)

ax = fig.add_subplot(131, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)
plt.title('2D "S shape" manifold in 3D')
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(132)
plt.scatter(X_isomap[:, 0], X_isomap[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("Isomap")
plt.xlabel("First component")
plt.ylabel("Second component")

ax = fig.add_subplot(133)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("t-SNE")
plt.xlabel("First component")
plt.ylabel("Second component")
plt.axis('tight')
```

```
(-67.00072708129883, 71.46007766723633, -20.22462863922119, 15.324001502990722)
```



5.2.4 Exercises

Run Manifold learning on handwritten digits: Locally Linear Embedding, Isomap with scikit-learn

5.3 Clustering

Wikipedia: Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). Clustering is one of the main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, and bioinformatics.

Sources: http://scikit-learn.org/stable/modules/clustering.html

5.3.1 K-means clustering

Source: C. M. Bishop *Pattern Recognition and Machine Learning*, Springer, 2006

Suppose we have a data set $X = \{x_1, \cdots, x_N\}$ that consists of N observations of a random D-dimensional Euclidean variable x. Our goal is to partition the data set into some number, K, of clusters, where we shall suppose for the moment that the value of K is given. Intuitively, we might think of a cluster as comprising a group of data points whose inter-point distances are small compared to the distances to points outside of the cluster. We can formalize this notion by first introducing a set of D-dimensional vectors μ_k, where $k = 1, \ldots, K$, in which μ_k is a **prototype** associated with the kth cluster. As we shall see shortly, we can think of the μ_k as representing the centres of the clusters. Our goal is then to find an assignment of data points to clusters, as well as a set of vectors $\{\mu_k\}$, such that the sum of the squares of the distances of each data point to its closest prototype vector μ_k, is at a minimum.

It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point x_i , we introduce a corresponding set of binary indicator variables $r_{ik} \in \{0, 1\}$, where $k = 1, \ldots, K$, that describes which of the K clusters the data point x_i is assigned to, so that if data point x_i is assigned to cluster k then $r_{ik} = 1$, and $r_{ij} = 0$ for $j \neq k$. This is known as the 1-of-K coding scheme. We can then define an objective function, denoted **inertia**, as

$$J(r, \mu) = \sum_i^N \sum_k^K r_{ik} \|x_i - \mu_k\|_2^2$$

which represents the sum of the squares of the Euclidean distances of each data point to its assigned vector μ_k. Our goal is to find values for the $\{r_{ik}\}$ and the $\{\mu_k\}$ so as to minimize the function J. We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the r_{ik} and the μ_k . First we choose some initial values for the μ_k. Then in the first phase we minimize J with respect to the r_{ik}, keeping the μ_k fixed. In the second phase we minimize J with respect to the μ_k, keeping r_{ik} fixed. This two-stage optimization process is then repeated until convergence. We shall see that these two stages of updating r_{ik} and μ_k correspond respectively to the expectation (E) and maximization (M) steps of the expectation-maximisation (EM) algorithm, and to emphasize this we shall use the terms E step and M step in the context of the K-means algorithm.

Consider first the determination of the r_{ik} . Because J in is a linear function of r_{ik} , this optimization can be performed easily to give a closed form solution. The terms involving different i are independent and so we can optimize for each i separately by choosing r_{ik} to be 1 for whichever value of k gives the minimum value of $||x_i - \mu_k||^2$. In other words, we simply assign the ith data point to the closest cluster centre. More formally, this can be expressed as

$$r_{ik} = \begin{cases} 1, & \text{if } k = \arg\min_j ||x_i - \mu_j||^2. \\ 0, & \text{otherwise.} \end{cases} \tag{5.15}$$

Now consider the optimization of the μ_k with the r_{ik} held fixed. The objective function J is a quadratic function of μ_k, and it can be minimized by setting its derivative with respect to μ_k to zero giving

$$2 \sum_i r_{ik}(x_i - \mu_k) = 0$$

which we can easily solve for μ_k to give

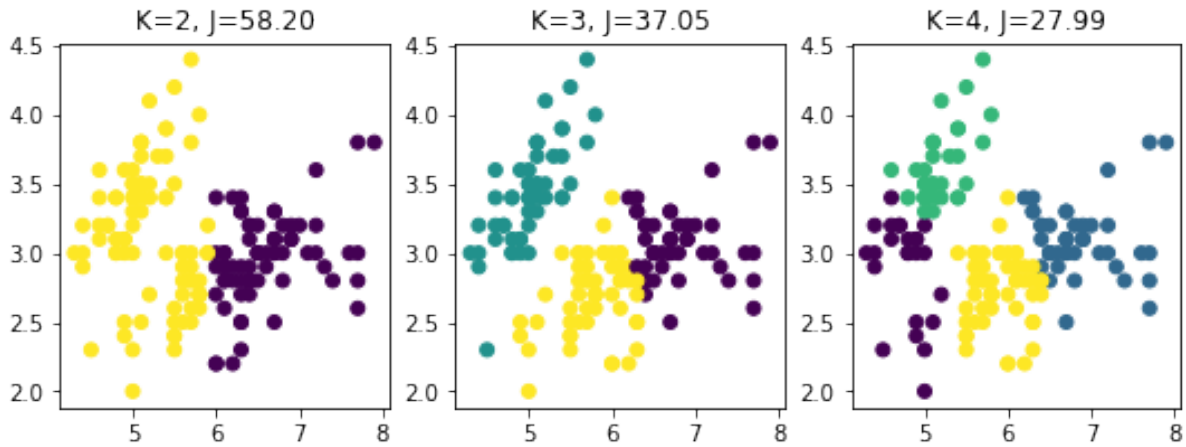
$$\mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}.$$

The denominator in this expression is equal to the number of points assigned to cluster k, and so this result has a simple interpretation, namely set μ_k equal to the mean of all of the data points x_i assigned to cluster k. For this reason, the procedure is known as the K-means algorithm.

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function J, convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J.

```
from sklearn import cluster, datasets
import matplotlib.pyplot as plt
import seaborn as sns  # nice color
%matplotlib inline

iris = datasets.load_iris()
X = iris.data[:, :2]  # use only 'sepal length and sepal width'
y_iris = iris.target

km2 = cluster.KMeans(n_clusters=2).fit(X)
km3 = cluster.KMeans(n_clusters=3).fit(X)
km4 = cluster.KMeans(n_clusters=4).fit(X)

plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=km2.labels_)
plt.title("K=2, J=%.2f" % km2.inertia_)

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=km3.labels_)
plt.title("K=3, J=%.2f" % km3.inertia_)

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=km4.labels_)#.astype(np.float))
plt.title("K=4, J=%.2f" % km4.inertia_)
```

```
Text(0.5, 1.0, 'K=4, J=27.99')
```



Exercises

1. Analyse clusters

- Analyse the plot above visually. What would a good value of K be?
- If you instead consider the inertia, the value of J, what would a good value of K be?
- Explain why there is such difference.
- For $K = 2$ why did K-means clustering not find the two "natural" clusters? See the assumptions of K-means: See sklearn doc.

2. Re-implement the K-means clustering algorithm (homework)

Write a function `kmeans(X, K)` that return an integer vector of the samples' labels.

5.3.2 Gaussian mixture models

The Gaussian mixture model (GMM) is a simple linear superposition of Gaussian components over the data, aimed at providing a rich class of density models. We turn to a formulation of Gaussian mixtures in terms of discrete latent variables: the K hidden classes to be discovered.

Differences compared to K-means:

- Whereas the K-means algorithm performs a hard assignment of data points to clusters, in which each data point is associated uniquely with one cluster, the GMM algorithm makes a soft assignment based on posterior probabilities.
- Whereas the classic K-means is only based on Euclidean distances, classic GMM use a Mahalanobis distances that can deal with non-spherical distributions. It should be noted that Mahalanobis could be plugged within an improved version of K-Means clustering. The Mahalanobis distance is unitless and scale-invariant, and takes into account the correlations of the data set.

The Gaussian mixture distribution can be written as a linear superposition of K Gaussians in the form:

$$p(x) = \sum_{k=1}^{K} \mathcal{N}(x \mid \mu_k, \Sigma_k) p(k),$$

where:

- The $p(k)$ are the mixing coefficients also know as the class probability of class k, and they sum to one: $\sum_{k=1}^{K} p(k) = 1$.
- $\mathcal{N}(x \mid \mu_k, \Sigma_k) = p(x \mid k)$ is the conditional distribution of x given a particular class k. It is the multivariate Gaussian distribution defined over a P-dimensional vector x of continuous variables.

The goal is to maximize the log-likelihood of the GMM:

$$\ln \prod_{i=1}^{N} p(x_i) = \ln \prod_{i=1}^{N} \left\{ \sum_{k=1}^{K} \mathcal{N}(x_i \mid \mu_k, \Sigma_k) p(k) \right\} = \sum_{i=1}^{N} \ln \left\{ \sum_{k=1}^{K} \mathcal{N}(x_i \mid \mu_k, \Sigma_k) p(k) \right\}.$$

To compute the classes parameters: $p(k), \mu_k, \Sigma_k$ we sum over all samples, by weighting each sample i by its responsibility or contribution to class k: $p(k \mid x_i)$ such that for each point its contribution to all classes sum to one $\sum_k p(k \mid x_i) = 1$. This contribution is the conditional probability of class k given x: $p(k \mid x)$ (sometimes called the posterior). It can be computed using Bayes' rule:

$$p(k \mid x) = \frac{p(x \mid k) p(k)}{p(x)} \tag{5.16}$$

$$= \frac{\mathcal{N}(x \mid \mu_k, \Sigma_k) p(k)}{\sum_{k=1}^{K} \mathcal{N}(x \mid \mu_k, \Sigma_k) p(k)} \tag{5.17}$$

Since the class parameters, $p(k)$, μ_k and Σ_k, depend on the responsibilities $p(k \mid x)$ and the responsibilities depend on class parameters, we need a two-step iterative algorithm: the expectation-maximization (EM) algorithm. We discuss this algorithm next.

The expectation-maximization (EM) algorithm for Gaussian mixtures

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters (comprised of the means and covariances of the components and the mixing coefficients).

Initialize the means μ_k, covariances Σ_k and mixing coefficients $p(k)$

1. **E step**. For each sample i, evaluate the responsibilities for each class k using the current parameter values

$$p(k \mid x_i) = \frac{\mathcal{N}(x_i \mid \mu_k, \Sigma_k) p(k)}{\sum_{k=1}^{K} \mathcal{N}(x_i \mid \mu_k, \Sigma_k) p(k)}$$

2. **M step**. For each class, re-estimate the parameters using the current responsibilities

$$\mu_k^{\text{new}} = \frac{1}{N_k} \sum_{i=1}^{N} p(k \mid x_i) x_i \tag{5.18}$$

$$\Sigma_k^{\text{new}} = \frac{1}{N_k} \sum_{i=1}^{N} p(k \mid x_i)(x_i - \mu_k^{\text{new}})(x_i - \mu_k^{\text{new}})^T \tag{5.19}$$

$$p^{\text{new}}(k) = \frac{N_k}{N} \tag{5.20}$$

3. Evaluate the log-likelihood

$$\sum_{i=1}^{N} \ln \left\{ \sum_{k=1}^{K} \mathcal{N}(x|\mu_k, \Sigma_k) p(k) \right\},$$

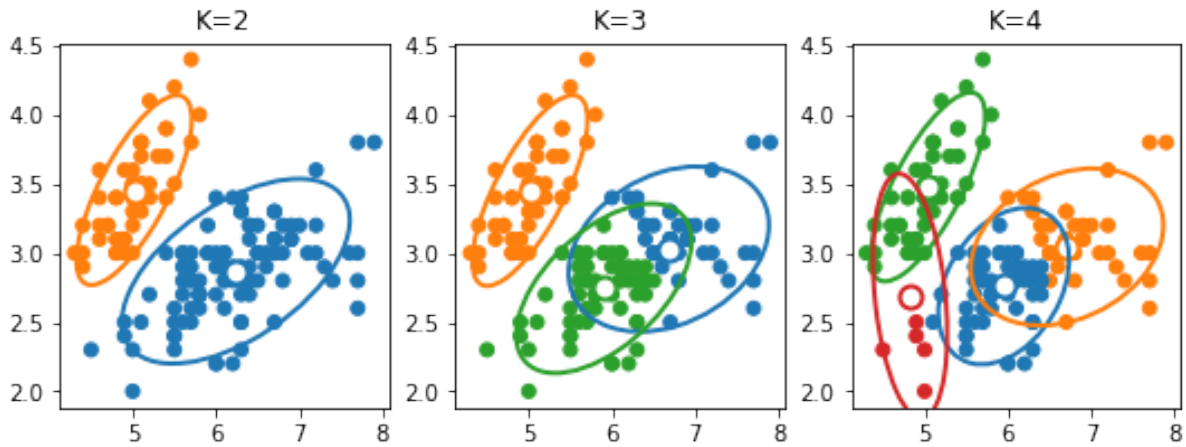
and check for convergence of either the parameters or the log-likelihood. If the convergence criterion is not satisfied return to step 1.

```
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
import seaborn as sns  # nice color
import sklearn
from sklearn.mixture import GaussianMixture

import pystatsml.plot_utils

colors = sns.color_palette()

iris = datasets.load_iris()
X = iris.data[:, :2]  # 'sepal length (cm)''sepal width (cm)'
y_iris = iris.target

gmm2 = GaussianMixture(n_components=2, covariance_type='full').fit(X)
gmm3 = GaussianMixture(n_components=3, covariance_type='full').fit(X)
gmm4 = GaussianMixture(n_components=4, covariance_type='full').fit(X)

plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab] for lab in gmm2.predict(X)])#,␣
↪color=colors)
for i in range(gmm2.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm2.covariances_[i, :], pos=gmm2.
↪means_[i, :],
                        facecolor='none', linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm2.means_[i, 0], gmm2.means_[i, 1], edgecolor=colors[i],
                marker="o", s=100, facecolor="w", linewidth=2)
plt.title("K=2")

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab] for lab in gmm3.predict(X)])
for i in range(gmm3.covariances_.shape[0]):
```

(continues on next page)

(continued from previous page)

```
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm3.covariances_[i, :], pos=gmm3.
↪means_[i, :],
                        facecolor='none', linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm3.means_[i, 0], gmm3.means_[i, 1], edgecolor=colors[i],
                marker="o", s=100, facecolor="w", linewidth=2)
plt.title("K=3")

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab] for lab in gmm4.predict(X)])  # .
↪astype(np.float))
for i in range(gmm4.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm4.covariances_[i, :], pos=gmm4.
↪means_[i, :],
                        facecolor='none', linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm4.means_[i, 0], gmm4.means_[i, 1], edgecolor=colors[i],
                marker="o", s=100, facecolor="w", linewidth=2)
_ = plt.title("K=4")
```



Models of covariances: parmeter `covariance_type` see Sklearn doc. K-means is almost a GMM with spherical covariance.

5.3.3 Model selection

Bayesian information criterion

In statistics, the Bayesian information criterion (BIC) is a criterion for model selection among a finite set of models; the model with the lowest BIC is preferred. It is based, in part, on the likelihood function and it is closely related to the Akaike information criterion (AIC).

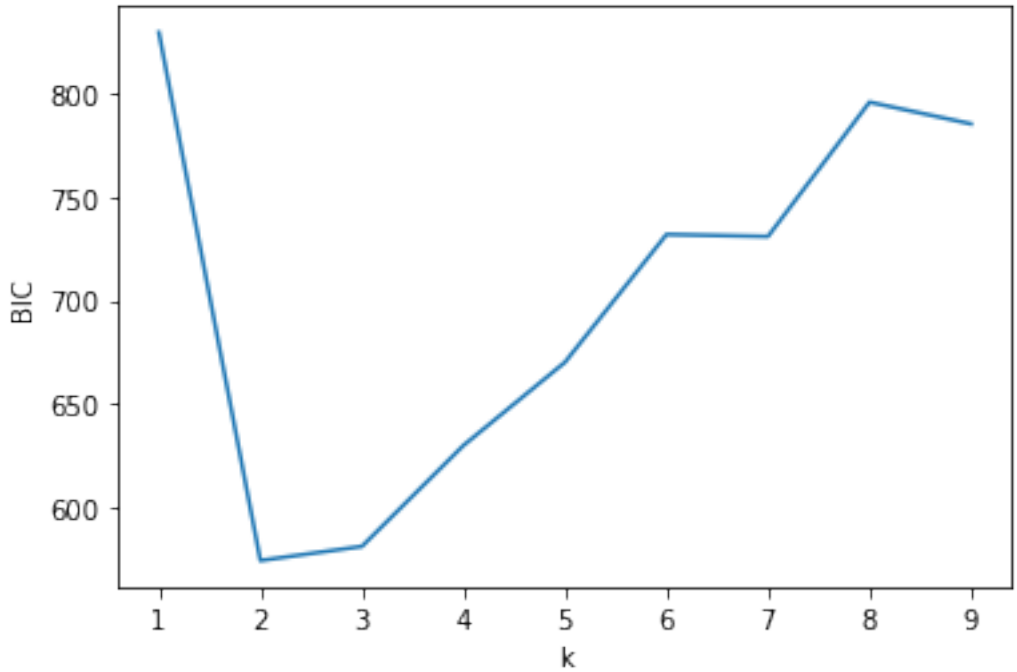
```
X = iris.data
y_iris = iris.target

bic = list()
#print(X)
```

(continues on next page)

(continued from previous page)

```
ks = np.arange(1, 10)

for k in ks:
    gmm = GaussianMixture(n_components=k, covariance_type='full')
    gmm.fit(X)
    bic.append(gmm.bic(X))

k_chosen = ks[np.argmin(bic)]

plt.plot(ks, bic)
plt.xlabel("k")
plt.ylabel("BIC")

print("Choose k=", k_chosen)
```

```
Choose k= 2
```



5.3.4 Hierarchical clustering

Hierarchical clustering is an approach to clustering that build hierarchies of clusters in two main approaches:

- **Agglomerative**: A *bottom-up* strategy, where each observation starts in their own cluster, and pairs of clusters are merged upwards in the hierarchy.
- **Divisive**: A *top-down* strategy, where all observations start out in the same cluster, and then the clusters are split recursively downwards in the hierarchy.

In order to decide which clusters to merge or to split, a measure of dissimilarity between clusters is introduced. More specific, this comprise a *distance* measure and a *linkage* criterion. The

distance measure is just what it sounds like, and the linkage criterion is essentially a function of the distances between points, for instance the minimum distance between points in two clusters, the maximum distance between points in two clusters, the average distance between points in two clusters, etc. One particular linkage criterion, the Ward criterion, will be discussed next.

Ward clustering

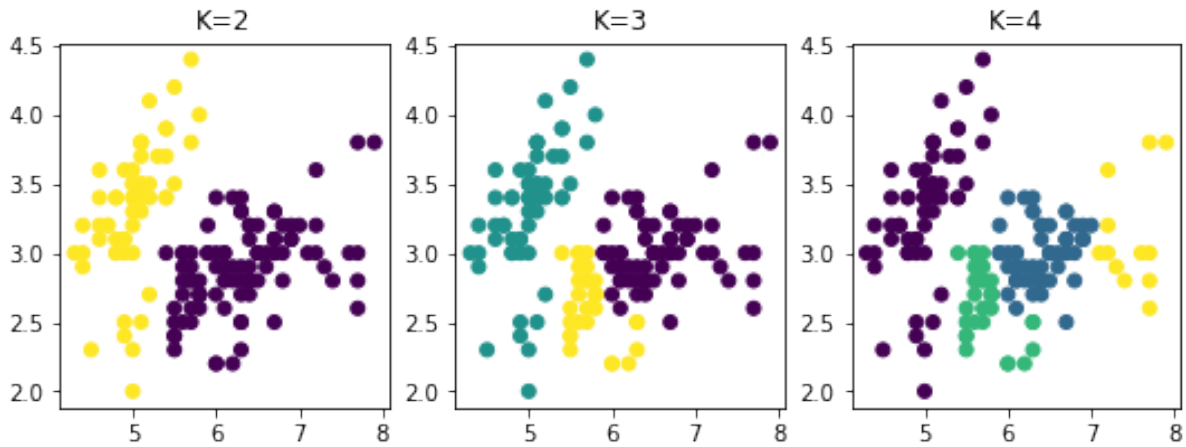
Ward clustering belongs to the family of agglomerative hierarchical clustering algorithms. This means that they are based on a "bottoms up" approach: each sample starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.

In Ward clustering, the criterion for choosing the pair of clusters to merge at each step is the minimum variance criterion. Ward's minimum variance criterion minimizes the total within-cluster variance by each merge. To implement this method, at each step: find the pair of clusters that leads to minimum increase in total within-cluster variance after merging. This increase is a weighted squared distance between cluster centers.

The main advantage of agglomerative hierarchical clustering over K-means clustering is that you can benefit from known neighborhood information, for example, neighboring pixels in an image.

```
from sklearn import cluster, datasets
import matplotlib.pyplot as plt
import seaborn as sns  # nice color

iris = datasets.load_iris()
X = iris.data[:, :2]  # 'sepal length (cm)''sepal width (cm)'
y_iris = iris.target

ward2 = cluster.AgglomerativeClustering(n_clusters=2, linkage='ward').fit(X)
ward3 = cluster.AgglomerativeClustering(n_clusters=3, linkage='ward').fit(X)
ward4 = cluster.AgglomerativeClustering(n_clusters=4, linkage='ward').fit(X)

plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=ward2.labels_)
plt.title("K=2")

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=ward3.labels_)
plt.title("K=3")

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=ward4.labels_)  # .astype(np.float))
plt.title("K=4")
```

```
Text(0.5, 1.0, 'K=4')
```



5.3.5 Exercises

Perform clustering of the iris dataset based on all variables using Gaussian mixture models. Use PCA to visualize clusters.

5.4 Linear models for regression problems

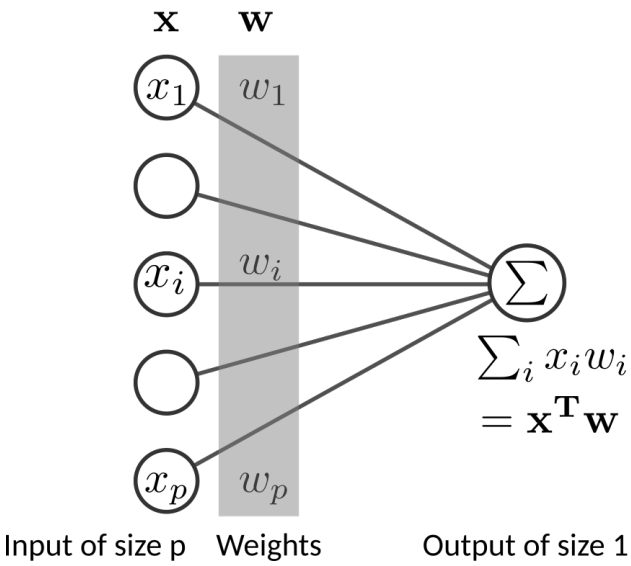


Fig. 2: Linear regression

5.4.1 Ordinary least squares

Linear regression models the **output**, or **target** variable $y \in \mathrm{R}$ as a linear combination of the P-dimensional input $\mathbf{x} \in \mathbb{R}^P$. Let $\mathbf{X}$ be the $N \times P$ matrix with each row an input vector (with a 1 in the first position), and similarly let $\mathbf{y}$ be the N-dimensional vector of outputs in the **training set**, the linear model will predict the $\mathbf{y}$ given $\mathbf{x}$ using the **parameter vector**, or **weight vector** $\mathbf{w} \in \mathbb{R}^P$ according to

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon,$$

where $\varepsilon \in \mathrm{R}^N$ are the **residuals**, or the errors of the prediction. The $\mathbf{w}$ is found by minimizing an **objective function**, which is the **loss function**, $L(\mathbf{w})$, i.e. the error measured on the data. This error is the **sum of squared errors (SSE) loss**.

$$\begin{aligned} L(\mathbf{w}) &= \mathrm{SSE}(\mathbf{w}) && (5.21)\\ &= \sum_i^N (y_i - \mathbf{x}_i^T\mathbf{w})^2 && (5.22)\\ &= (\mathbf{y} - \mathbf{X}^T\mathbf{w})^T(\mathbf{y} - \mathbf{X}^T\mathbf{w}) && (5.23)\\ &= \|\mathbf{y} - \mathbf{X}^T\mathbf{w}\|_2^2, && (5.24) \end{aligned}$$

Minimizing the SSE is the Ordinary Least Square **OLS** regression as objective function. which is a simple **ordinary least squares (OLS)** minimization whose analytic solution is:

$$\mathbf{w}_{\mathrm{OLS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

The gradient of the loss:

$$\partial\frac{L(\mathbf{w}, \mathbf{X}, \mathbf{y})}{\partial\mathbf{w}} = 2\sum_i \mathbf{x}_i(\mathbf{x}_i \cdot \mathbf{w} - y_i)$$

5.4.2 Linear regression with scikit-learn

Scikit learn offer many models for supervised learning, and they all follow the same application programming interface (API), namely:

```
model = Estimator()
model.fit(X, y)
predictions = model.predict(X)
```

```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn import datasets
import sklearn.linear_model as lm
```

(continues on next page)

(continued from previous page)

```
import sklearn.metrics as metrics

np.set_printoptions(precision=2)
pd.set_option('precision', 2)
```

Linear regression of `Advertising.csv` dataset with TV and Radio advertising as input features and Sales as target. The linear model that minimizes the MSE is a plan (2 input features) defined as: Sales = 0.05 TV + .19 Radio + 3:

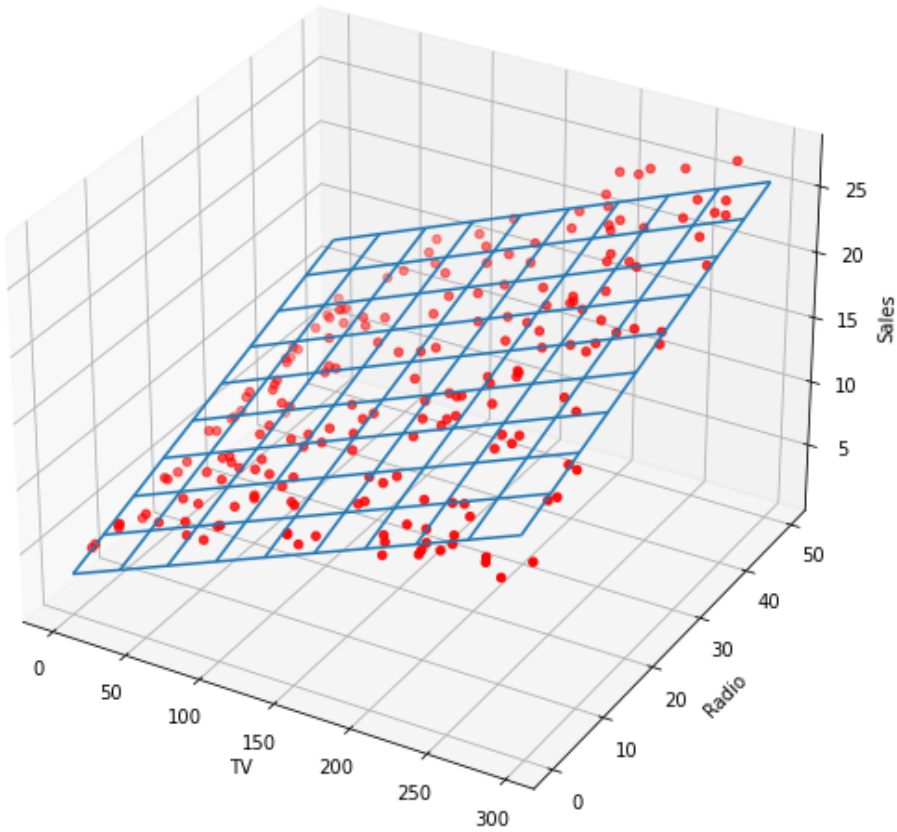


Fig. 3: Linear regression

5.4.3 Overfitting

In statistics and machine learning, overfitting occurs when a statistical model describes random errors or noise instead of the underlying relationships. Overfitting generally occurs when a model is **excessively complex**, such as having **too many parameters relative to the number of observations**. A model that has been overfit will generally have poor predictive performance, as it can exaggerate minor fluctuations in the data.

A learning algorithm is trained using some set of training samples. If the learning algorithm has the capacity to overfit the training samples the performance on the **training sample set** will improve while the performance on unseen **test sample set** will decline.

The overfitting phenomenon has three main explanations: - excessively complex models, - multicollinearity, and - high dimensionality.

Model complexity

Complex learners with too many parameters relative to the number of observations may overfit the training dataset.

Multicollinearity

Predictors are highly correlated, meaning that one can be linearly predicted from the others. In this situation the coefficient estimates of the multiple regression may change erratically in response to small changes in the model or the data. Multicollinearity does not reduce the predictive power or reliability of the model as a whole, at least not within the sample data set; it only affects computations regarding individual predictors. That is, a multiple regression model with correlated predictors can indicate how well the entire bundle of predictors predicts the outcome variable, but it may not give valid results about any individual predictor, or about which predictors are redundant with respect to others. In case of perfect multicollinearity the predictor matrix is singular and therefore cannot be inverted. Under these circumstances, for a general linear model $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon$, the ordinary least-squares estimator, $\mathbf{w}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$, does not exist.

An example where correlated predictor may produce an unstable model follows: We want to predict the business potential (pb) of some companies given their business volume (bv) and the taxes (tx) they are paying. Here pb $\sim$ 10% of bv. However, taxes = 20% of bv (tax and bv are highly collinear), therefore there is an infinite number of linear combinations of tax and bv that lead to the same prediction. Solutions with very large coefficients will produce excessively large predictions.

```
bv = np.array([10, 20, 30, 40, 50])             # business volume
tax  = .2 * bv                                  # Tax
bp = .1 * bv + np.array([-.1, .2, .1, -.2, .1]) # business potential

X = np.column_stack([bv, tax])
beta_star = np.array([.1, 0])  # true solution

'''
Since tax and bv are correlated, there is an infinite number of linear␣
↪combinations
```

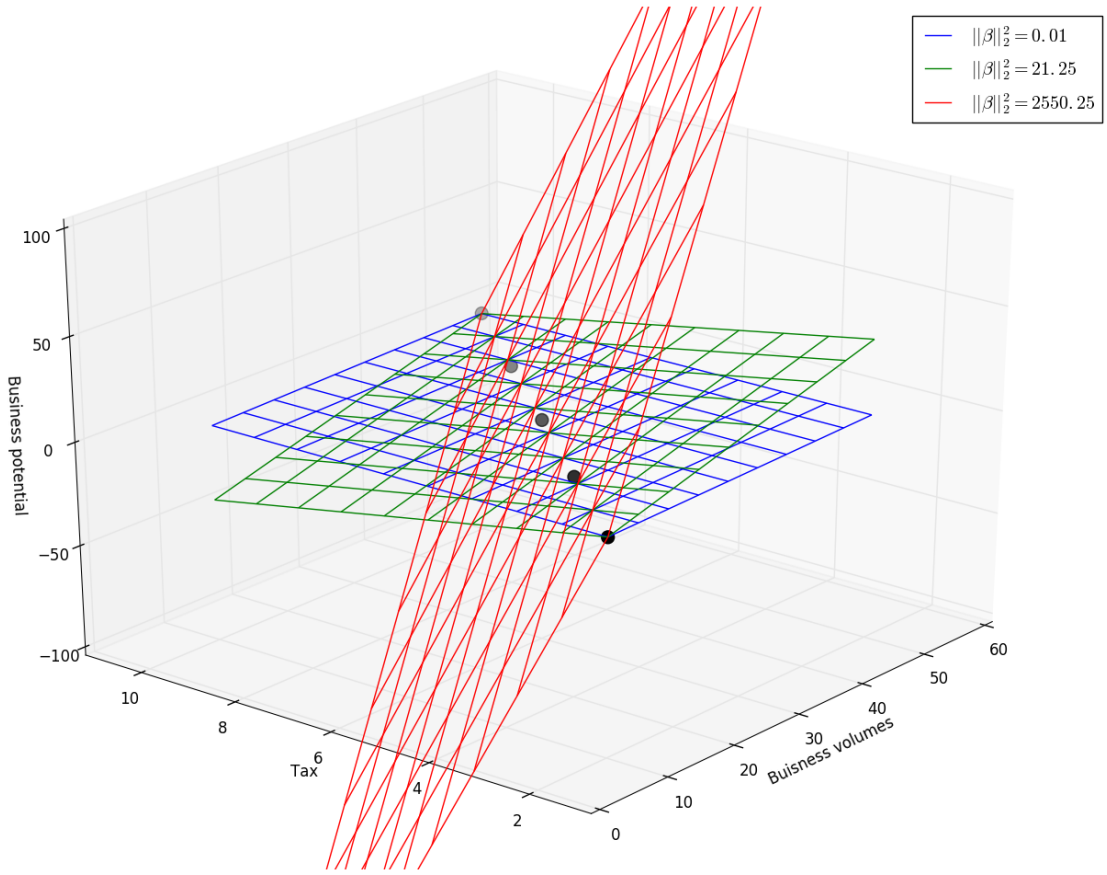
(continues on next page)

(continued from previous page)

```
leading to the same prediction.
'''

# 10 times the bv then subtract it 9 times using the tax variable:
beta_medium = np.array([.1 * 10, -.1 * 9 * (1/.2)])
# 100 times the bv then subtract it 99 times using the tax variable:
beta_large = np.array([.1 * 100, -.1 * 99 * (1/.2)])

print("L2 norm of coefficients: small:%.2f, medium:%.2f, large:%.2f." %
      (np.sum(beta_star ** 2), np.sum(beta_medium ** 2), np.sum(beta_large ** 2)))

print("However all models provide the exact same predictions.")
assert np.all(np.dot(X, beta_star) == np.dot(X, beta_medium))
assert np.all(np.dot(X, beta_star) == np.dot(X, beta_large))
```

```
L2 norm of coefficients: small:0.01, medium:21.25, large:2550.25.
However all models provide the exact same predictions.
```

Multicollinearity between the predictors: business volumes and tax produces unstable models with arbitrary large coefficients.



Dealing with multicollinearity:

- Regularisation by e.g. ℓ_2 shrinkage: Introduce a bias in the solution by making $(X^T X)^{-1}$ non-singular. See ℓ_2 shrinkage.

- Feature selection: select a small number of features. See: Isabelle Guyon and André Elisseeff *An introduction to variable and feature selection* The Journal of Machine Learning Research, 2003.
- Feature selection: select a small number of features using ℓ_1 shrinkage.
- Extract few independent (uncorrelated) features using e.g. principal components analysis (PCA), partial least squares regression (PLS-R) or regression methods that cut the number of predictors to a smaller set of uncorrelated components.

High dimensionality

High dimensions means a large number of input features. Linear predictor associate one parameter to each input feature, so a high-dimensional situation (P, number of features, is large) with a relatively small number of samples N (so-called large P small N situation) generally lead to an overfit of the training data. Thus it is generally a bad idea to add many input features into the learner. This phenomenon is called the **curse of dimensionality**.

One of the most important criteria to use when choosing a learning algorithm is based on the relative size of P and N.

- Remenber that the "covariance" matrix $\mathbf{X}^T\mathbf{X}$ used in the linear model is a $P \times P$ matrix of rank $\min(N, P)$. Thus if $P > N$ the equation system is overparameterized and admit an infinity of solutions that might be specific to the learning dataset. See also ill-conditioned or singular matrices.
- The sampling density of N samples in an P-dimensional space is proportional to $N^{1/P}$. Thus a high-dimensional space becomes very sparse, leading to poor estimations of samples densities. To preserve a constant density, an exponential growth in the number of observations is required. 50 points in 1D, would require 2 500 points in 2D and 125 000 in 3D!
- Another consequence of the sparse sampling in high dimensions is that all sample points are close to an edge of the sample. Consider N data points uniformly distributed in a P-dimensional unit ball centered at the origin. Suppose we consider a nearest-neighbor estimate at the origin. The median distance from the origin to the closest data point is given by the expression: $d(P, N) = \left(1 - \frac{1}{2}^{1/N}\right)^{1/P}$.

A more complicated expression exists for the mean distance to the closest point. For N = 500, P = 10 , $d(P, N) \approx 0.52$, more than halfway to the boundary. Hence most data points are closer to the boundary of the sample space than to any other data point. The reason that this presents a problem is that prediction is much more difficult near the edges of the training sample. One must extrapolate from neighboring sample points rather than interpolate between them. *(Source: T Hastie, R Tibshirani, J Friedman.*The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Second Edition, 2009.)*

- Structural risk minimization provides a theoretical background of this phenomenon. (See VC dimension.)
- See also bias–variance trade-off.

5.4.4 Regularization using penalization of coefficients

Regarding linear models, overfitting generally leads to excessively complex solutions (coefficient vectors), accounting for noise or spurious correlations within predictors. **Regularization** aims to alleviate this phenomenon by constraining (biasing or reducing) the capacity of the learning algorithm in order to promote simple solutions. Regularization penalizes "large" solutions forcing the coefficients to be small, i.e. to shrink them toward zeros.

The objective function $J(\mathbf{w})$ to minimize with respect to $\mathbf{w}$ is composed of a loss function $L(\mathbf{w})$ for goodness-of-fit and a penalty term $\Omega(\mathbf{w})$ (regularization to avoid overfitting). This is a trade-off where the respective contribution of the loss and the penalty terms is controlled by the regularization parameter λ.

Therefore the **loss function** $L(\mathbf{w})$ is combined with a **penalty function** $\Omega(\mathbf{w})$ leading to the general form:

$$J(\mathbf{w}) = L(\mathbf{w}) + \lambda\Omega(\mathbf{w}).$$

The respective contribution of the loss and the penalty is controlled by the **regularization parameter** λ.

For regression problems the loss is the SSE given by:

$$\begin{aligned} L(\mathbf{w}) = SSE(\mathbf{w}) &= \sum_i^N (y_i - \mathbf{x}_i^T\mathbf{w})^2 \\ &= \|\mathbf{y} - \mathbf{x}\mathbf{w}\|_2^2 \end{aligned}$$

Popular penalties are:

- Ridge (also called ℓ_2) penalty: $\|\mathbf{w}\|_2^2$. It shrinks coefficients toward 0.
- Lasso (also called ℓ_1) penalty: $\|\mathbf{w}\|_1$. It performs feature selection by setting some coefficients to 0.
- ElasticNet (also called $\ell_1\ell_2$) penalty: $\alpha\left(\rho\,\|\mathbf{w}\|_1 + (1-\rho)\,\|\mathbf{w}\|_2^2\right)$. It performs selection of group of correlated features by setting some coefficients to 0.

The next figure shows the predicted performance (r-squared) on train and test sets with an increasing number of input features. The number of predictive features is always 10% of the total number of input features. Therefore, the signal to noise ratio (SNR) increases by increasing the number of input features. The performances on the training set rapidly reach 100% (R2=1). However, the performance on the test set decreases with the increase of the input dimensionality. The difference between the train and test performances (blue shaded region) depicts the overfitting phenomena. Regularisation using penalties of the coefficient vector norm greatly limits the overfitting phenomena.

With scikit-learn:

```
# Dataset with some correlation
X, y, coef = datasets.make_regression(n_samples=100, n_features=10, n_
↪informative=5, random_state=0,
                                      effective_rank=3, coef=True)
```

(continues on next page)

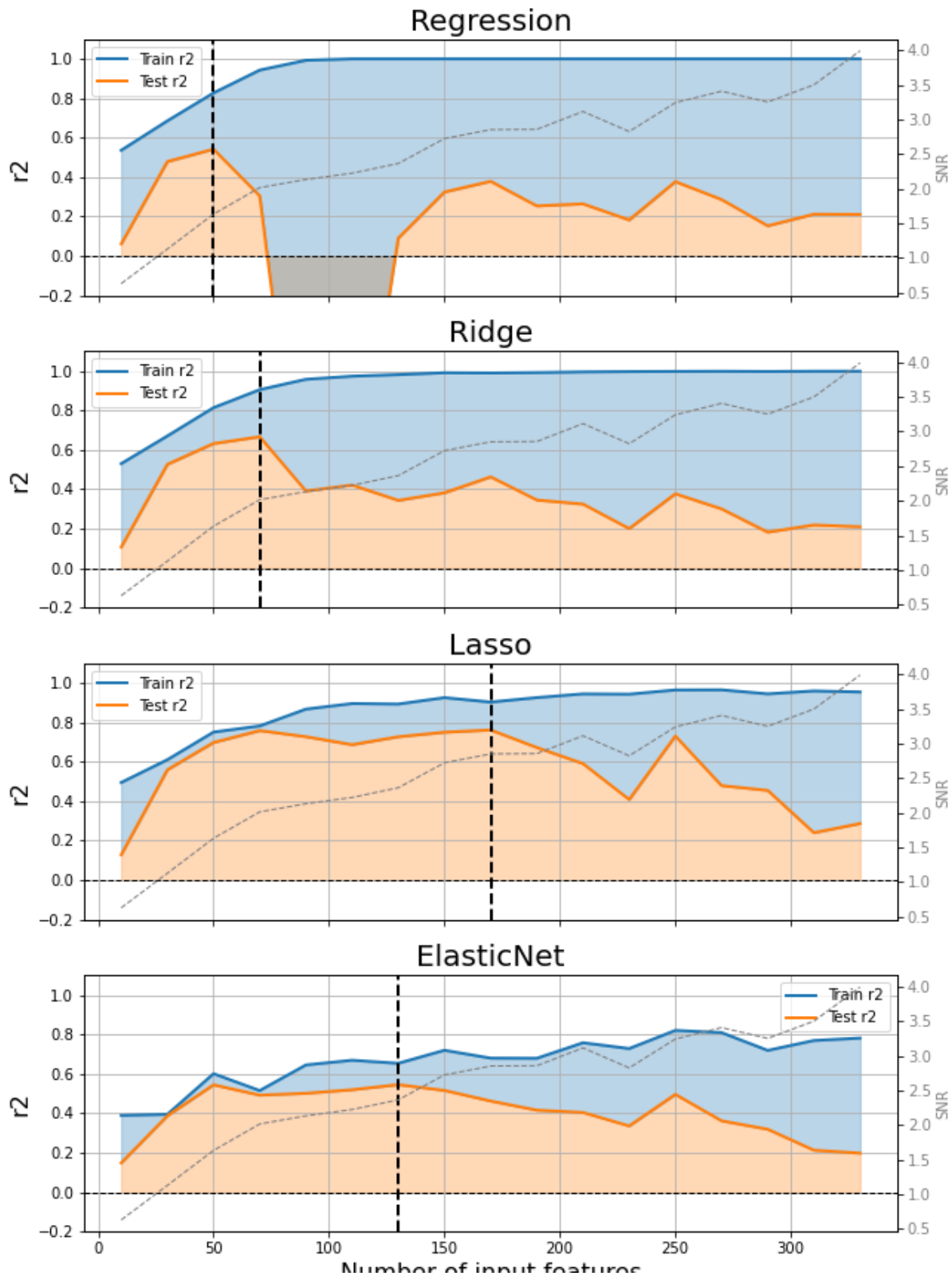


Fig. 4: Multicollinearity between the predictors

(continued from previous page)

```
lr = lm.LinearRegression().fit(X, y)

l2 = lm.Ridge(alpha=10).fit(X, y)  # lambda is alpha!

l1 = lm.Lasso(alpha=.1).fit(X, y)  # lambda is alpha !

l1l2 = lm.ElasticNet(alpha=.1, l1_ratio=.9).fit(X, y)

pd.DataFrame(np.vstack((coef, lr.coef_, l2.coef_, l1.coef_, l1l2.coef_)),
             index=['True', 'lr', 'l2', 'l1', 'l1l2'])
```

5.4.5 Ridge regression (ℓ_2-regularization)

Ridge regression impose a ℓ_2 penalty on the coefficients, i.e. it penalizes with the Euclidean norm of the coefficients while minimizing SSE. The objective function becomes:

$$\text{Ridge}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^T\mathbf{w})^2 + \lambda\|\mathbf{w}\|_2^2 \tag{5.25}$$

$$= \|\mathbf{y} - \mathbf{x}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_2^2. \tag{5.26}$$

The $\mathbf{w}$ that minimises $F_{Ridge}(\mathbf{w})$ can be found by the following derivation:

$$\nabla_{\mathbf{w}}\text{Ridge}(\mathbf{w}) = 0 \tag{5.27}$$

$$\nabla_{\mathbf{w}}\big((\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda\mathbf{w}^T\mathbf{w}\big) = 0 \tag{5.28}$$

$$\nabla_{\mathbf{w}}\big((\mathbf{y}^T\mathbf{y} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} + \lambda\mathbf{w}^T\mathbf{w})\big) = 0 \tag{5.29}$$

$$-2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\mathbf{w} + 2\lambda\mathbf{w} = 0 \tag{5.30}$$

$$-\mathbf{X}^T\mathbf{y} + (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\mathbf{w} = 0 \tag{5.31}$$

$$(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\mathbf{w} = \mathbf{x}^T\mathbf{y} \tag{5.32}$$

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{x}^T\mathbf{y} \tag{5.33}$$

- The solution adds a positive constant to the diagonal of $\mathbf{X}^T\mathbf{X}$ before inversion. This makes the problem nonsingular, even if $\mathbf{X}^T\mathbf{X}$ is not of full rank, and was the main motivation behind ridge regression.
- Increasing λ shrinks the $\mathbf{w}$ coefficients toward 0.
- This approach **penalizes** the objective function by the **Euclidian (:math:`ell_2`) norm** of the coefficients such that solutions with large coefficients become unattractive.

The gradient of the loss:

$$\partial\frac{L(\mathbf{w}, \mathbf{X}, \mathbf{y})}{\partial\mathbf{w}} = 2(\sum_i \mathbf{x}_i(\mathbf{x}_i \cdot \mathbf{w} - y_i) + \lambda\mathbf{w})$$

5.4.6 Lasso regression (ℓ_1-regularization)

Lasso regression penalizes the coefficients by the ℓ_1 norm. This constraint will reduce (bias) the capacity of the learning algorithm. To add such a penalty forces the coefficients to be small, i.e. it shrinks them toward zero. The objective function to minimize becomes:

$$\text{Lasso}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^T \mathbf{w})^2 + \lambda \|\mathbf{w}\|_1. \tag{5.34}$$

This penalty forces some coefficients to be exactly zero, providing a feature selection property.

Sparsity of the ℓ_1 norm

Occam's razor

Occam's razor (also written as Ockham's razor, and **lex parsimoniae** in Latin, which means law of parsimony) is a problem solving principle attributed to William of Ockham (1287-1347), who was an English Franciscan friar and scholastic philosopher and theologian. The principle can be interpreted as stating that **among competing hypotheses, the one with the fewest assumptions should be selected**.

Principle of parsimony

The simplest of two competing theories is to be preferred. Definition of parsimony: Economy of explanation in conformity with Occam's razor.

Among possible models with similar loss, choose the simplest one:

- Choose the model with the smallest coefficient vector, i.e. smallest ℓ_2 ($\|\mathbf{w}\|_2$) or ℓ_1 ($\|\mathbf{w}\|_1$) norm of $\mathbf{w}$, i.e. ℓ_2 or ℓ_1 penalty. See also bias-variance tradeoff.
- Choose the model that uses the smallest number of predictors. In other words, choose the model that has many predictors with zero weights. Two approaches are available to obtain this: (i) Perform a feature selection as a preprocessing prior to applying the learning algorithm, or (ii) embed the feature selection procedure within the learning process.

Sparsity-induced penalty or embedded feature selection with the ℓ_1 penalty

The penalty based on the ℓ_1 norm promotes **sparsity** (scattered, or not dense): it forces many coefficients to be exactly zero. This also makes the coefficient vector scattered.

The figure bellow illustrates the OLS loss under a constraint acting on the ℓ_1 norm of the coefficient vector. I.e., it illustrates the following optimization problem:

$$\begin{aligned} &\underset{\mathbf{w}}{\text{minimize}} \ \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 \\ &\text{subject to } \|\mathbf{w}\|_1 \leq 1. \end{aligned}$$

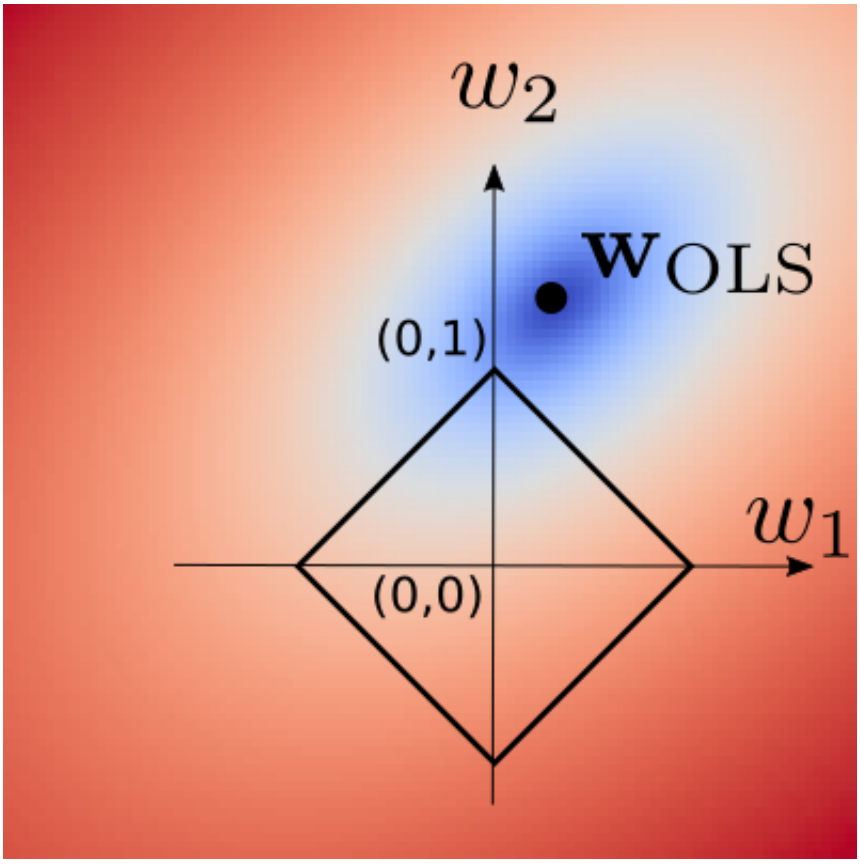


Fig. 5: Sparsity of L1 norm

Optimization issues

Section to be completed

- No more closed-form solution.
- Convex but not differentiable.
- Requires specific optimization algorithms, such as the fast iterative shrinkage-thresholding algorithm (FISTA): Amir Beck and Marc Teboulle, *A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems* SIAM J. Imaging Sci., 2009.

The ridge penalty shrinks the coefficients toward zero. The figure illustrates: the OLS solution on the left. The ℓ_1 and ℓ_2 penalties in the middle pane. The penalized OLS in the right pane. The right pane shows how the penalties shrink the coefficients toward zero. The black points are the minimum found in each case, and the white points represents the true solution used to generate the data.

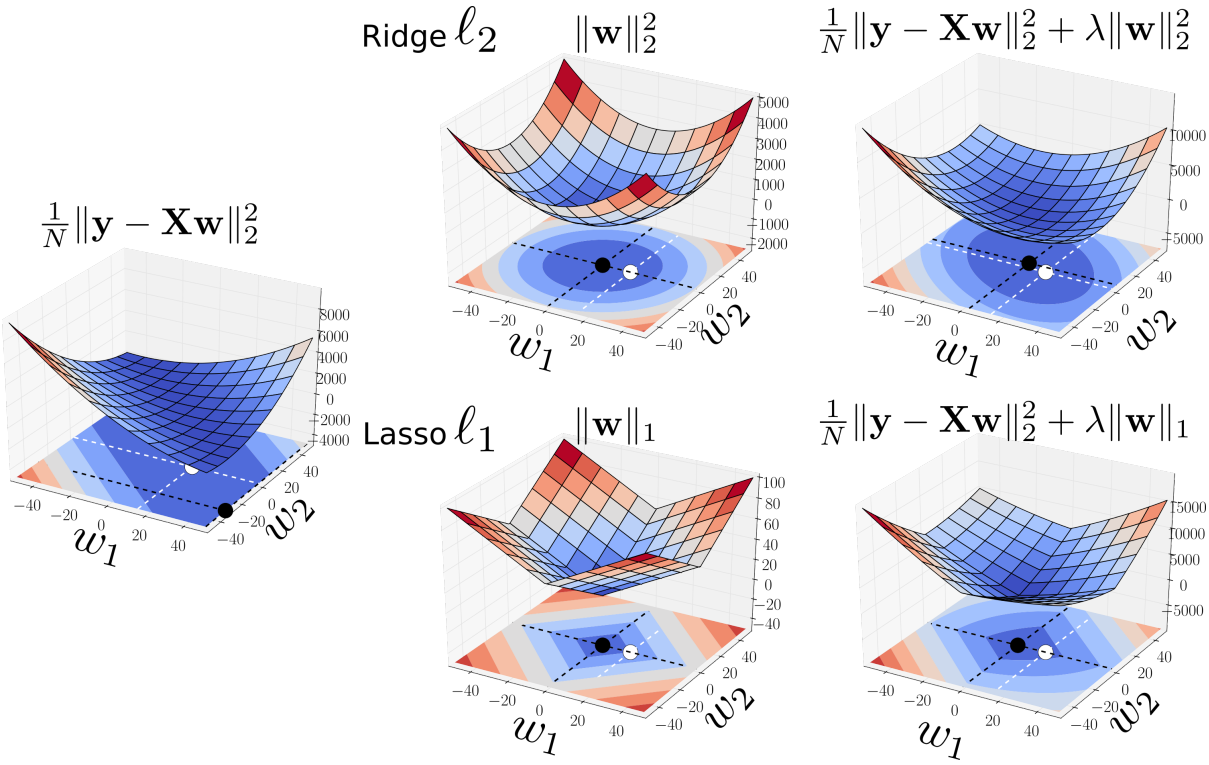


Fig. 6: ℓ_1 and ℓ_2 shrinkages

5.4.7 Elastic-net regression (ℓ_1-ℓ_2-regularization)

The Elastic-net estimator combines the ℓ_1 and ℓ_2 penalties, and results in the problem to

$$\text{Enet}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^T\mathbf{w})^2 + \alpha\left(\rho\,\|\mathbf{w}\|_1 + (1-\rho)\,\|\mathbf{w}\|_2^2\right), \tag{5.35}$$

where α acts as a global penalty and ρ as an ℓ_1/ℓ_2 ratio.

Rational

- If there are groups of highly correlated variables, Lasso tends to arbitrarily select only one from each group. These models are difficult to interpret because covariates that are strongly associated with the outcome are not included in the predictive model. Conversely, the elastic net encourages a grouping effect, where strongly correlated predictors tend to be in or out of the model together.
- Studies on real world data and simulation studies show that the elastic net often outperforms the lasso, while enjoying a similar sparsity of representation.

5.4.8 Regression performance evaluation metrics: R-squared, MSE and MAE

Common regression metrics are:

- R^2 : R-squared
- MSE: Mean Squared Error
- MAE: Mean Absolute Error

R-squared

The goodness of fit of a statistical model describes how well it fits a set of observations. Measures of goodness of fit typically summarize the discrepancy between observed values and the values expected under the model in question. We will consider the **explained variance** also known as the coefficient of determination, denoted R^2 pronounced **R-squared**.

The total sum of squares, SS_{tot} is the sum of the sum of squares explained by the regression, SS_{reg}, plus the sum of squares of residuals unexplained by the regression, SS_{res}, also called the SSE, i.e. such that

$$SS_{\text{tot}} = SS_{\text{reg}} + SS_{\text{res}}$$

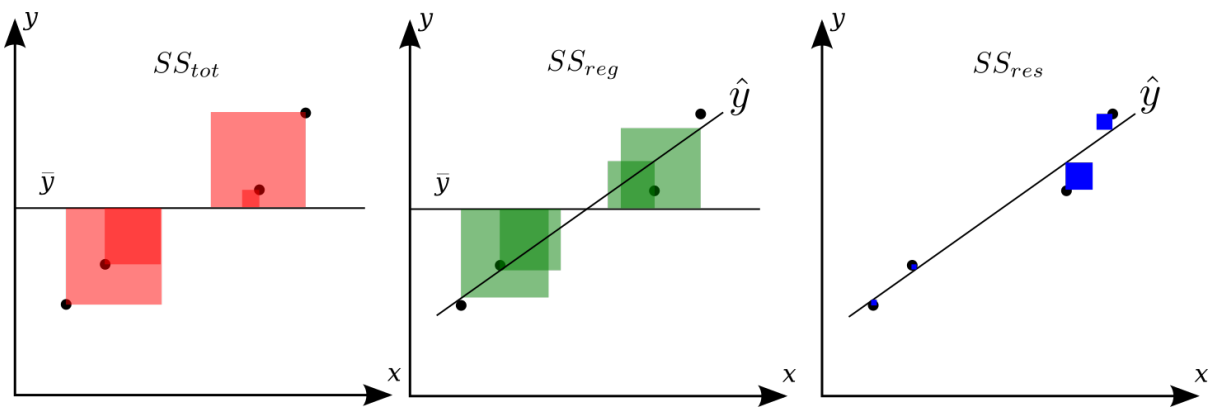


Fig. 7: title

The mean of y is

$$\bar{y} = \frac{1}{n}\sum_i y_i.$$

The total sum of squares is the total squared sum of deviations from the mean of y, i.e.

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

The regression sum of squares, also called the explained sum of squares:

$$SS_{\text{reg}} = \sum_i (\hat{y}_i - \bar{y})^2,$$

where $\hat{y}_i = \beta x_i + \beta_0$ is the estimated value of salary $\hat{y}_i$ given a value of experience x_i.

The sum of squares of the residuals (**SSE, Sum Squared Error**), also called the residual sum of squares (RSS) is:

$$SS_{\text{res}} = \sum_i (y_i - \hat{y}_i)^2.$$

R^2 is the explained sum of squares of errors. It is the variance explain by the regression divided by the total variance, i.e.

$$R^2 = \frac{\text{explained SS}}{\text{total SS}} = \frac{SS_{\text{reg}}}{SS_{tot}} = 1 - \frac{SS_{res}}{SS_{tot}}.$$

Test

Let $\hat{\sigma}^2 = SS_{\text{res}}/(n-2)$ be an estimator of the variance of ϵ. The 2 in the denominator stems from the 2 estimated parameters: intercept and coefficient.

- **Unexplained variance**: $\frac{SS_{\text{res}}}{\hat{\sigma}^2} \sim \chi^2_{n-2}$
- **Explained variance**: $\frac{SS_{\text{reg}}}{\hat{\sigma}^2} \sim \chi^2_1$. The single degree of freedom comes from the difference between $\frac{SS_{\text{tot}}}{\hat{\sigma}^2} (\sim \chi^2_{n-1})$ and $\frac{SS_{\text{res}}}{\hat{\sigma}^2} (\sim \chi^2_{n-2})$, i.e. $(n-1)-(n-2)$ degree of freedom.

The Fisher statistics of the ratio of two variances:

$$F = \frac{\text{Explained variance}}{\text{Unexplained variance}} = \frac{SS_{\text{reg}}/1}{SS_{\text{res}}/(n-2)} \sim F(1, n-2)$$

Using the F-distribution, compute the probability of observing a value greater than F under H_0, i.e.: $P(x > F|H_0)$, i.e. the survival function $(1 - \text{Cumulative Distribution Function})$ at x of the given F-distribution.

```
import sklearn.metrics as metrics
from sklearn.model_selection import train_test_split

X, y = datasets.make_regression(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=1)

lr = lm.LinearRegression()
lr.fit(X_train, y_train)
yhat = lr.predict(X_test)

r2 = metrics.r2_score(y_test, yhat)
mse = metrics.mean_squared_error(y_test, yhat)
mae = metrics.mean_absolute_error(y_test, yhat)

print("r2: %.3f, mae: %.3f, mse: %.3f" % (r2, mae, mse))
```

```
r2: 0.050, mae: 71.834, mse: 7891.217
```

In pure numpy:

```
res = y_test - lr.predict(X_test)

y_mu = np.mean(y_test)
```

(continues on next page)

(continued from previous page)

```
ss_tot = np.sum((y_test - y_mu) ** 2)
ss_res = np.sum(res ** 2)

r2 = (1 - ss_res / ss_tot)
mse = np.mean(res ** 2)
mae = np.mean(np.abs(res))

print("r2: %.3f, mae: %.3f, mse: %.3f" % (r2, mae, mse))
```

```
r2: 0.050, mae: 71.834, mse: 7891.217
```

5.5 Linear models for classification problems

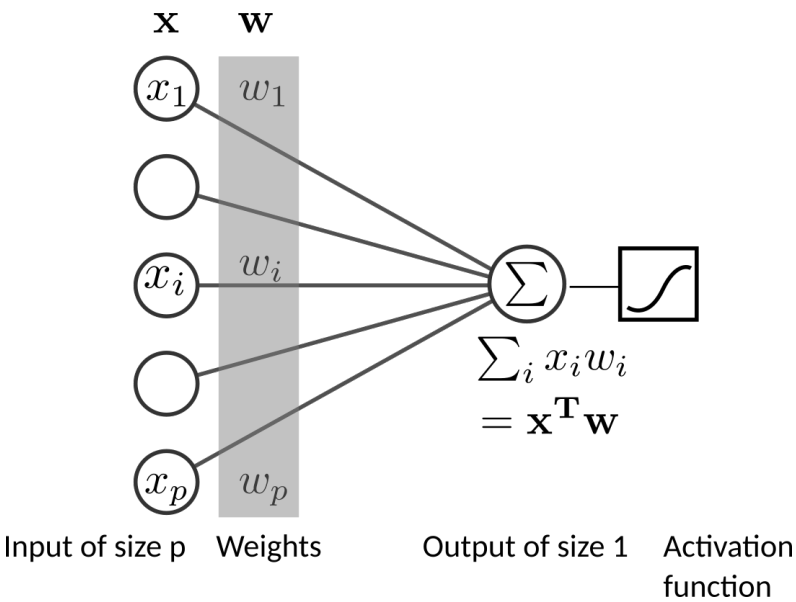


Fig. 8: Linear (logistic) classification

Given a training set of N samples, $D = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_N}, y_N)\}$, where $\boldsymbol{x_i}$ is a multidimensional input vector with dimension P and class label (target or response).

Multiclass Classification problems can be seen as several binary classification problems $y_i \in \{0, 1\}$ where the classifier aims to discriminate the sample of the current class (label 1) versus the samples of other classes (label 0).

Therfore, for each class the classifier seek for a vector of parameters $\boldsymbol{w}$ that performs a linear combination of the input variables, $\boldsymbol{x}^T\boldsymbol{w}$. This step performs a **projection** or a **rotation** of input sample into a good discriminative one-dimensional sub-space, that best discriminate sample of current class vs sample of other classes.

This score (a.k.a decision function) is tranformed, using the nonlinear activation funtion $f(.)$, to a "posterior probabilities" of class 1: $p(y = 1|\boldsymbol{x}) = f(\boldsymbol{x}^T\boldsymbol{w})$, where, $p(y = 1|\boldsymbol{x}) = 1 - p(y = 0|\boldsymbol{x})$.

The decision surfaces (orthogonal hyperplan to $\boldsymbol{w}$) correspond to $f(x) =$ constant, so that $\boldsymbol{x}^T\boldsymbol{w} =$ constant and hence the decision surfaces are linear functions of $\boldsymbol{x}$, even if the function $f(.)$ is nonlinear.

A thresholding of the activation (shifted by the bias or intercept) provides the predicted class label.

The vector of parameters, that defines the discriminative axis, minimizes an **objective function** $J(\boldsymbol{w})$ that is a sum of of **loss function** $L(\boldsymbol{w})$ and some penalties on the weights vector $\Omega(\boldsymbol{w})$.

$$\min_{\boldsymbol{w}} \; J = \sum_i L(y_i, f(\boldsymbol{x_i}^T \boldsymbol{w})) + \Omega(\boldsymbol{w}),$$

5.5.1 Fisher's linear discriminant with equal class covariance

This geometric method does not make any probabilistic assumptions, instead it relies on distances. It looks for the **linear projection** of the data points onto a vector, $\boldsymbol{w}$, that maximizes the between/within variance ratio, denoted $F(\boldsymbol{w})$. Under a few assumptions, it will provide the same results as linear discriminant analysis (LDA), explained below.

Suppose two classes of observations, C_0 and C_1, have means $\boldsymbol{\mu_0}$ and $\boldsymbol{\mu_1}$ and the same total within-class scatter ("covariance") matrix,

$$\begin{aligned}
\boldsymbol{S_W} &= \sum_{i \in C_0} (\boldsymbol{x_i} - \boldsymbol{\mu_0})(\boldsymbol{x_i} - \boldsymbol{\mu_0})^T + \sum_{j \in C_1} (\boldsymbol{x_j} - \boldsymbol{\mu_1})(\boldsymbol{x_j} - \boldsymbol{\mu_1})^T && (5.36)\\
&= \boldsymbol{X_c}^T \boldsymbol{X_c}, && (5.37)
\end{aligned}$$

where $\boldsymbol{X_c}$ is the $(N \times P)$ matrix of data centered on their respective means:

$$\boldsymbol{X_c} = \begin{bmatrix} \boldsymbol{X_0} - \boldsymbol{\mu_0} \\ \boldsymbol{X_1} - \boldsymbol{\mu_1} \end{bmatrix},$$

where $\boldsymbol{X_0}$ and $\boldsymbol{X_1}$ are the $(N_0 \times P)$ and $(N_1 \times P)$ matrices of samples of classes C_0 and C_1.

Let $\boldsymbol{S_B}$ being the scatter "between-class" matrix, given by

$$\boldsymbol{S_B} = (\boldsymbol{\mu_1} - \boldsymbol{\mu_0})(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})^T.$$

The linear combination of features $\boldsymbol{w}^T x$ have means $\boldsymbol{w}^T \mu_i$ for $i = 0, 1$, and variance $\boldsymbol{w}^T \boldsymbol{X_c^T} \boldsymbol{X_c} \boldsymbol{w}$. Fisher defined the separation between these two distributions to be the ratio of the variance between the classes to the variance within the classes:

$$\begin{aligned}
F_{\text{Fisher}}(\boldsymbol{w}) &= \frac{\sigma^2_{\text{between}}}{\sigma^2_{\text{within}}} && (5.38)\\
&= \frac{(\boldsymbol{w}^T \boldsymbol{\mu_1} - \boldsymbol{w}^T \boldsymbol{\mu_0})^2}{\boldsymbol{w}^T X_c^T \boldsymbol{X_c} \boldsymbol{w}} && (5.39)\\
&= \frac{(\boldsymbol{w}^T (\boldsymbol{\mu_1} - \boldsymbol{\mu_0}))^2}{\boldsymbol{w}^T X_c^T \boldsymbol{X_c} \boldsymbol{w}} && (5.40)\\
&= \frac{\boldsymbol{w}^T (\boldsymbol{\mu_1} - \boldsymbol{\mu_0})(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})^T w}{\boldsymbol{w}^T X_c^T \boldsymbol{X_c} \boldsymbol{w}} && (5.41)\\
&= \frac{\boldsymbol{w}^T \boldsymbol{S_B} w}{\boldsymbol{w}^T \boldsymbol{S_W} \boldsymbol{w}}. && (5.42)
\end{aligned}$$

The Fisher most discriminant projection

In the two-class case, the maximum separation occurs by a projection on the $(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})$ using the Mahalanobis metric $\boldsymbol{S_W}^{-1}$, so that

$$\boldsymbol{w} \propto \boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0}).$$

Demonstration

Differentiating $F_{\text{Fisher}}(w)$ with respect to w gives

$$\begin{aligned}
\nabla_{\boldsymbol{w}} F_{\text{Fisher}}(\boldsymbol{w}) &= 0 \\
\nabla_{\boldsymbol{w}} \left(\frac{\boldsymbol{w}^T \boldsymbol{S_B} w}{\boldsymbol{w}^T \boldsymbol{S_W} \boldsymbol{w}} \right) &= 0 \\
(\boldsymbol{w}^T \boldsymbol{S_W} \boldsymbol{w})(2\boldsymbol{S_B} \boldsymbol{w}) - (\boldsymbol{w}^T \boldsymbol{S_B} \boldsymbol{w})(2\boldsymbol{S_W} \boldsymbol{w}) &= 0 \\
(\boldsymbol{w}^T \boldsymbol{S_W} \boldsymbol{w})(\boldsymbol{S_B} \boldsymbol{w}) &= (\boldsymbol{w}^T \boldsymbol{S_B} \boldsymbol{w})(\boldsymbol{S_W} \boldsymbol{w}) \\
\boldsymbol{S_B} \boldsymbol{w} &= \frac{\boldsymbol{w}^T \boldsymbol{S_B} \boldsymbol{w}}{\boldsymbol{w}^T \boldsymbol{S_W} \boldsymbol{w}} (\boldsymbol{S_W} \boldsymbol{w}) \\
\boldsymbol{S_B} \boldsymbol{w} &= \lambda (\boldsymbol{S_W} \boldsymbol{w}) \\
\boldsymbol{S_W}^{-1} \boldsymbol{S_B} \boldsymbol{w} &= \lambda \boldsymbol{w}.
\end{aligned}$$

Since we do not care about the magnitude of $\boldsymbol{w}$, only its direction, we replaced the scalar factor $(\boldsymbol{w}^T \boldsymbol{S_B} \boldsymbol{w})/(\boldsymbol{w}^T \boldsymbol{S_W} \boldsymbol{w})$ by λ.

In the multiple-class case, the solutions w are determined by the eigenvectors of $\boldsymbol{S_W}^{-1} \boldsymbol{S_B}$ that correspond to the $K - 1$ largest eigenvalues.

However, in the two-class case (in which $\boldsymbol{S_B} = (\boldsymbol{\mu_1} - \boldsymbol{\mu_0})(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})^T$) it is easy to show that $\boldsymbol{w} = \boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})$ is the unique eigenvector of $\boldsymbol{S_W}^{-1} \boldsymbol{S_B}$:

$$\begin{aligned}
\boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})^T \boldsymbol{w} &= \lambda \boldsymbol{w} \\
\boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})^T \boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0}) &= \lambda \boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0}),
\end{aligned}$$

where here $\lambda = (\boldsymbol{\mu_1} - \boldsymbol{\mu_0})^T \boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})$. Which leads to the result

$$\boldsymbol{w} \propto \boldsymbol{S_W}^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0}).$$

The separating hyperplane

The separating hyperplane is a $P - 1$-dimensional hyper surface, orthogonal to the projection vector, w. There is no single best way to find the origin of the plane along w, or equivalently the classification threshold that determines whether a point should be classified as belonging to C_0 or to C_1. However, if the projected points have roughly the same distribution, then the threshold can be chosen as the hyperplane exactly between the projections of the two means, i.e. as

$$T = \boldsymbol{w} \cdot \frac{1}{2}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0}).$$

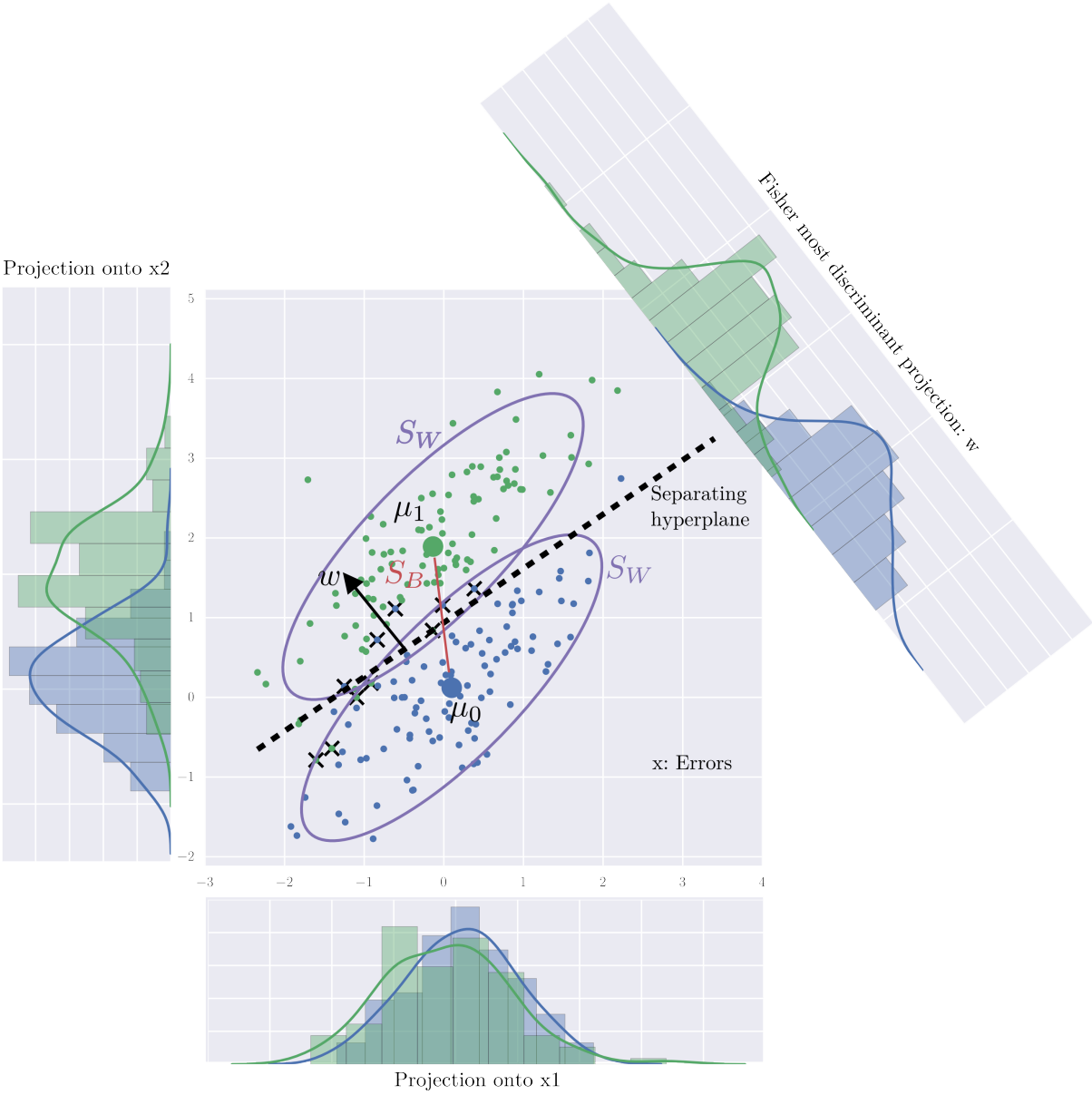


Fig. 9: The Fisher most discriminant projection

```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics

np.set_printoptions(precision=2)
pd.set_option('precision', 2)
```

5.5.2 Linear discriminant analysis (LDA)

Linear discriminant analysis (LDA) is a probabilistic generalization of Fisher's linear discriminant. It uses Bayes' rule to fix the threshold based on prior probabilities of classes.

1. First compute the class-**conditional distributions** of $\boldsymbol{x}$ given class C_k: $p(x|C_k) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_k}, \boldsymbol{S_W})$. Where $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_k}, \boldsymbol{S_W})$ is the multivariate Gaussian distribution defined over a P-dimensional vector x of continuous variables, which is given by

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu_k}, \boldsymbol{S_W}) = \frac{1}{(2\pi)^{P/2}|\boldsymbol{S_W}|^{1/2}} \exp\{-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu_k})^T \boldsymbol{S_W}^{-1}(x - \boldsymbol{\mu_k})\}$$

2. Estimate the **prior probabilities** of class k, $p(C_k) = N_k/N$.

3. Compute **posterior probabilities** (ie. the probability of a each class given a sample) combining conditional with priors using Bayes' rule:

$$p(C_k|\boldsymbol{x}) = \frac{p(C_k)p(\boldsymbol{x}|C_k)}{p(\boldsymbol{x})}$$

Where $p(x)$ is the marginal distribution obtained by suming of classes: As usual, the denominator in Bayes' theorem can be found in terms of the quantities appearing in the numerator, because

$$p(x) = \sum_k p(\boldsymbol{x}|C_k)p(C_k)$$

4. Classify $\boldsymbol{x}$ using the Maximum-a-Posteriori probability: $C_k = \arg\max_{C_k} p(C_k|\boldsymbol{x})$

LDA is a **generative model** since the class-conditional distributions cal be used to generate samples of each classes.

LDA is useful to deal with imbalanced group sizes (eg.: $N_1 \gg N_0$) since priors probabilities can be used to explicitly re-balance the classification by setting $p(C_0) = p(C_1) = 1/2$ or whatever seems relevant.

LDA can be generalised to the multiclass case with $K > 2$.

With $N_1 = N_0$, LDA lead to the same solution than Fisher's linear discriminant.

Exercise

How many parameters are required to estimate to perform a LDA ?

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

# Dataset 2 two multivariate normal
n_samples, n_features = 100, 2
mean0, mean1 = np.array([0, 0]), np.array([0, 2])
Cov = np.array([[1, .8],[.8, 1]])
np.random.seed(42)
X0 = np.random.multivariate_normal(mean0, Cov, n_samples)
X1 = np.random.multivariate_normal(mean1, Cov, n_samples)
X = np.vstack([X0, X1])
y = np.array([0] * X0.shape[0] + [1] * X1.shape[0])


# LDA with scikit-learn
lda = LDA()
proj = lda.fit(X, y).transform(X)
y_pred_lda = lda.predict(X)

errors =  y_pred_lda != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
↪lda)))
```

```
Nb errors=10, error rate=0.05
```

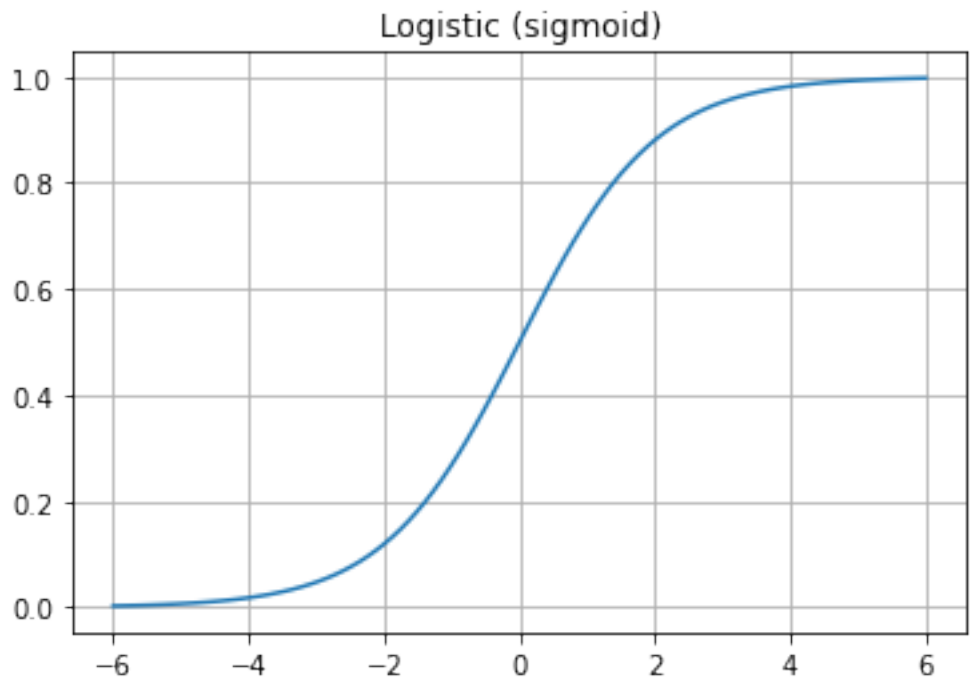
5.5.3 Logistic regression

Logistic regression is called a generalized linear models. ie.: it is a linear model with a link function that maps the output of linear multiple regression to the posterior probability of class 1 $p(1|x)$ using the logistic sigmoid function:

$$p(1|\boldsymbol{w}, \boldsymbol{x_i}) = \frac{1}{1 + \exp(-\boldsymbol{w} \cdot \boldsymbol{x_i})}$$

```
def logistic(x): return 1 / (1 + np.exp(-x))

x = np.linspace(-6, 6, 100)
plt.plot(x, logistic(x))
plt.grid(True)
plt.title('Logistic (sigmoid)')
```

```
Text(0.5, 1.0, 'Logistic (sigmoid)')
```



Logistic regression is a **discriminative model** since it focuses only on the posterior probability of each class $p(C_k|x)$. It only requires to estimate the P weights of the $\boldsymbol{w}$ vector. Thus it should be favoured over LDA with many input features. In small dimension and balanced situations it would provide similar predictions than LDA.

However imbalanced group sizes cannot be explicitly controlled. It can be managed using a reweighting of the input samples.

```
logreg = lm.LogisticRegression(penalty='none').fit(X, y)
# This class implements regularized logistic regression.
# C is the Inverse of regularization strength.
# Large value => no regularization.

logreg.fit(X, y)
y_pred_logreg = logreg.predict(X)

errors =  y_pred_logreg != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
↪logreg)))
print(logreg.coef_)
```

```
Nb errors=10, error rate=0.05
[[-5.15  5.57]]
```

Exercise

Explore the `Logistic Regression` parameters and proposes a solution in cases of highly imbalanced training dataset $N_1 \gg N_0$ when we know that in reality both classes have the same probability $p(C_1) = p(C_0)$.

5.5.4 Losses

Negative log likelihood or cross-entropy

The **Loss function** for sample i is the negative log of the probability:

$$L(\boldsymbol{w}, \boldsymbol{x_i}, y_i) = \begin{cases} -\log(p(1|w, \boldsymbol{x_i})) & \text{if } y_i = 1 \\ -\log(1 - p(1|w, \boldsymbol{x_i}) & \text{if } y_i = 0 \end{cases}$$

For the whole dataset $\boldsymbol{X}, \boldsymbol{y} = \{\boldsymbol{x_i}, y_i\}$ the loss function to minimize $L(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y})$ is the negative negative log likelihood (nll) that can be simplied using a 0/1 coding of the label in the case of binary classification:

$$\begin{aligned} L(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) &= -\log \mathcal{L}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) && (5.43) \\ &= -\log \Pi_i \{p(1|\boldsymbol{w}, \boldsymbol{x_i})^{y_i} (1 - p(1|\boldsymbol{w}, \boldsymbol{x_i})^{(1-y_i)}\} && (5.44) \\ &= \sum_i \{y_i \log p(1|\boldsymbol{w}, \boldsymbol{x_i}) + (1 - y_i) \log(1 - p(1|\boldsymbol{w}, \boldsymbol{x_i}))\}, && (5.45) \end{aligned}$$

This is known as the **cross-entropy** between the true label y and the predicted probability p.

For the logistic regression case, we have:

$$L(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) = \sum_i \{y_i \boldsymbol{w} \cdot \boldsymbol{x_i} - \log(1 + \exp(\boldsymbol{w} \cdot \boldsymbol{x_i}))\}$$

This is solved by numerical method using the gradient of the loss:

$$\partial \frac{L(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y})}{\partial \boldsymbol{w}} = \sum_i \boldsymbol{x_i}(y_i - p(1|\boldsymbol{w}, \boldsymbol{x_i}))$$

See also Scikit learn doc

Hinge loss or ℓ_1 loss

TODO

5.5.5 Overfitting

VC dimension (for Vapnik–Chervonenkis dimension) is a measure of the **capacity** (complexity, expressive power, richness, or flexibility) of a statistical classification algorithm, defined as the cardinality of the largest set of points that the algorithm can shatter.

Theorem: Linear classifier in R^P have VC dimension of $P + 1$. Hence in dimension two ($P = 2$) any random partition of 3 points can be learned.

Fig. 10: In 2D we can shatter any three non-collinear points

5.5.6 Regularization using penalization of coefficients

The penalties use in regression are also used in classification. The only difference is the loss function generally the negative log likelihood (cross-entropy) or the hinge loss. We will explore:

- Ridge (also called ℓ_2) penalty: $\|\mathbf{w}\|_2^2$. It shrinks coefficients toward 0.
- Lasso (also called ℓ_1) penalty: $\|\mathbf{w}\|_1$. It performs feature selection by setting some coefficients to 0.
- ElasticNet (also called $\ell_1\ell_2$) penalty: $\alpha\left(\rho\,\|\mathbf{w}\|_1 + (1-\rho)\,\|\mathbf{w}\|_2^2\right)$. It performs selection of group of correlated features by setting some coefficients to 0.

```
# Dataset with some correlation
X, y = datasets.make_classification(n_samples=100, n_features=10,
                                    n_informative=5, n_redundant=3,
                                    n_classes=2, random_state=3,␣
↪shuffle=False)

lr = lm.LogisticRegression(penalty='none').fit(X, y)

l2 = lm.LogisticRegression(penalty='l2', C=.1).fit(X, y)  # lambda = 1 / C!

# use solver 'saga' to handle L1 penalty
l1 = lm.LogisticRegression(penalty='l1', C=.1, solver='saga').fit(X, y)  # lambda␣
↪= 1 / C!

l1l2 = lm.LogisticRegression(penalty='elasticnet',  C=.1, l1_ratio=0.5, solver=
↪'saga').fit(X, y)  # lambda = 1 / C!


pd.DataFrame(np.vstack((lr.coef_, l2.coef_, l1.coef_, l1l2.coef_)),
             index=['lr', 'l2', 'l1', 'l1l2'])
```

5.5.7 Ridge Fisher's linear classification (ℓ_2-regularization)

When the matrix $\boldsymbol{S_W}$ is not full rank or $P \gg N$, the The Fisher most discriminant projection estimate of the is not unique. This can be solved using a biased version of $\boldsymbol{S_W}$:

$$\boldsymbol{S_W}^{Ridge} = \boldsymbol{S_W} + \lambda \boldsymbol{I}$$

where I is the $P \times P$ identity matrix. This leads to the regularized (ridge) estimator of the Fisher's linear discriminant analysis:

$$\boldsymbol{w}^{Ridge} \propto (\boldsymbol{S_W} + \lambda \boldsymbol{I})^{-1}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0})$$

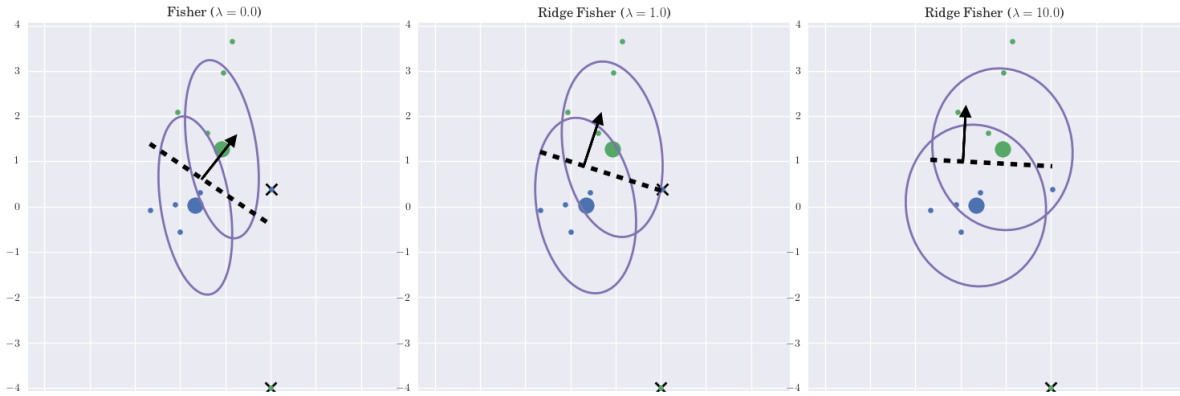


Fig. 11: The Ridge Fisher most discriminant projection

Increasing λ will:

- Shrinks the coefficients toward zero.
- The covariance will converge toward the diagonal matrix, reducing the contribution of the pairwise covariances.

5.5.8 Ridge logistic regression (ℓ_2-regularization)

The **objective function** to be minimized is now the combination of the logistic loss (negative log likelyhood) $-\log \mathcal{L}(\boldsymbol{w})$ with a penalty of the L2 norm of the weights vector. In the two-class case, using the 0/1 coding we obtain:

$$\min_{\boldsymbol{w}} \text{ Logistic ridge}(\boldsymbol{w}) = -\log \mathcal{L}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) + \lambda \, \|\boldsymbol{w}\|^2$$

```
from sklearn import linear_model
lrl2 = linear_model.LogisticRegression(penalty='l2', C=.1)
# This class implements regularized logistic regression. C is the Inverse of␣
↪regularization strength.
# Large value => no regularization.

lrl2.fit(X, y)
y_pred_l2 = lrl2.predict(X)
prob_pred_l2 = lrl2.predict_proba(X)

print("Probas of 5 first samples for class 0 and class 1:")
print(prob_pred_l2[:5, :])

print("Coef vector:")
print(lrl2.coef_)

# Retrieve proba from coef vector
probas = 1 / (1 + np.exp(- (np.dot(X, lrl2.coef_.T) + lrl2.intercept_))).ravel()
print("Diff", np.max(np.abs(prob_pred_l2[:, 1] - probas)))
```

(continues on next page)

(continued from previous page)

```
errors =  y_pred_l2 != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y)))
```

```
Probas of 5 first samples for class 0 and class 1:
[[0.89 0.11]
 [0.72 0.28]
 [0.73 0.27]
 [0.75 0.25]
 [0.48 0.52]]
Coef vector:
[[-0.05  0.52 -0.21  0.34  0.26 -0.05  0.14  0.27 -0.25  0.21]]
Diff 0.0
Nb errors=24, error rate=0.24
```

5.5.9 Lasso logistic regression (ℓ_1-regularization)

The **objective function** to be minimized is now the combination of the logistic loss $-\log \mathcal{L}(\boldsymbol{w})$ with a penalty of the L1 norm of the weights vector. In the two-class case, using the 0/1 coding we obtain:

$$\min_{\boldsymbol{w}} \text{Logistic Lasso}(w) = -\log \mathcal{L}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) + \lambda \|\boldsymbol{w}\|_1$$

```
from sklearn import linear_model
lrl1 = lm.LogisticRegression(penalty='l1', C=.1, solver='saga') # lambda = 1 / C!

# This class implements regularized logistic regression. C is the Inverse of␣
↪regularization strength.
# Large value => no regularization.

lrl1.fit(X, y)
y_pred_lrl1 = lrl1.predict(X)

errors =  y_pred_lrl1 != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
↪lrl1)))

print("Coef vector:")
print(lrl1.coef_)
```

```
Nb errors=27, error rate=0.27
Coef vector:
[[0.   0.31 0.   0.1  0.   0.   0.   0.26 0.   0.  ]]
```

5.5.10 Ridge linear Support Vector Machine (ℓ_2-regularization)

Support Vector Machine seek for separating hyperplane with maximum margin to enforce robustness against noise. Like logistic regression it is a **discriminative method** that only focuses of predictions.

Here we present the non separable case of Maximum Margin Classifiers with ± 1 coding (ie.: y_i $\{-1, +1\}$). In the next figure the legend aply to samples of "dot" class.

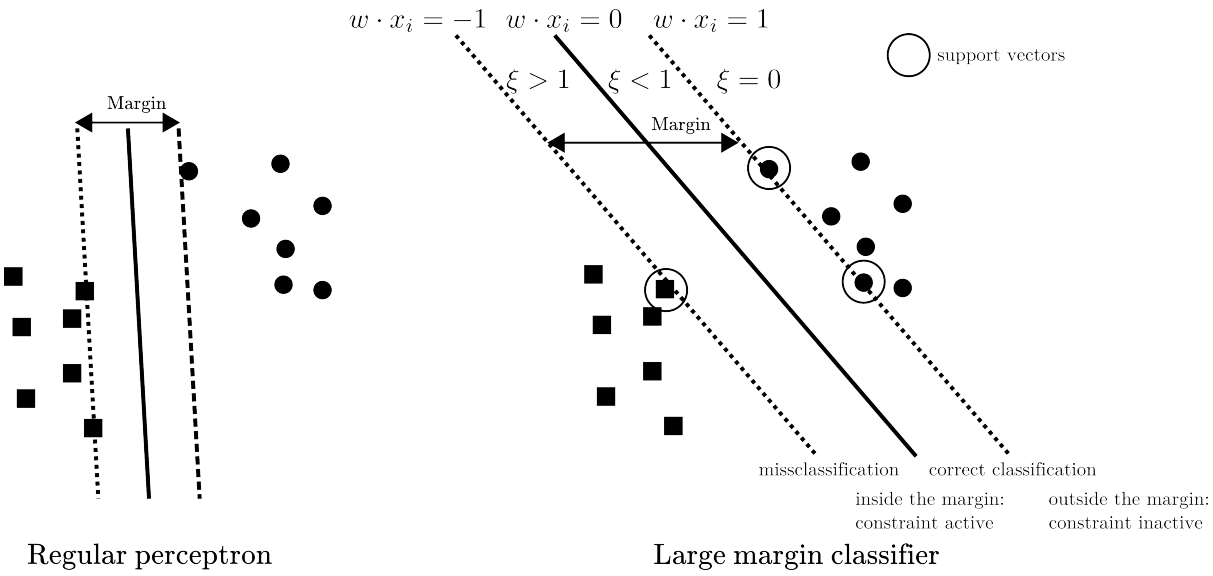


Fig. 12: Linear lar margin classifiers

Linear SVM for classification (also called SVM-C or SVC) minimizes:

$$\begin{array}{lll} \min & \text{Linear SVM}(\boldsymbol{w}) & = \text{penalty}(w) + C \text{ Hinge loss}(w) \\ & & = \|w\|_2^2 + C \sum_i^N \xi_i \\ \text{with} & \forall i & y_i(w \cdot \boldsymbol{x_i}) \geq 1 - \xi_i \end{array}$$

Here we introduced the slack variables: ξ_i, with $\xi_i = 0$ for points that are on or inside the correct margin boundary and $\xi_i = |y_i - (w\ cdot \cdot \boldsymbol{x_i})|$ for other points. Thus:

1. If $y_i(w \cdot \boldsymbol{x_i}) \geq 1$ then the point lies outside the margin but on the correct side of the decision boundary. In this case $\xi_i = 0$. The constraint is thus not active for this point. It does not contribute to the prediction.
2. If $1 > y_i(w \cdot \boldsymbol{x_i}) \geq 0$ then the point lies inside the margin and on the correct side of the decision boundary. In this case $0 < \xi_i \leq 1$. The constraint is active for this point. It does contribute to the prediction as a support vector.
3. If $0 < y_i(w \cdot \boldsymbol{x_i}))$ then the point is on the wrong side of the decision boundary (missclassification). In this case $0 < \xi_i > 1$. The constraint is active for this point. It does contribute to the prediction as a support vector.

This loss is called the hinge loss, defined as:

$$\max(0, 1 - y_i\ (w \cdot \boldsymbol{x_i}))$$

So linear SVM is closed to Ridge logistic regression, using the hinge loss instead of the logistic loss. Both will provide very similar predictions.

```
from sklearn import svm

svmlin = svm.LinearSVC(C=.1)
# Remark: by default LinearSVC uses squared_hinge as loss
svmlin.fit(X, y)
y_pred_svmlin = svmlin.predict(X)

errors =  y_pred_svmlin != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
↪svmlin)))
print("Coef vector:")
print(svmlin.coef_)
```

```
Nb errors=20, error rate=0.20
Coef vector:
[[-0.    0.32 -0.09  0.17  0.16 -0.01  0.06  0.13 -0.16  0.13]]
```

5.5.11 Lasso linear Support Vector Machine (ℓ_1-regularization)

Linear SVM for classification (also called SVM-C or SVC) with l1-regularization

$$\begin{array}{lll} \min & F_{\text{Lasso linear SVM}}(w) & = ||w||_1 + C\ \sum_i^N \xi_i \\ \text{with} & \forall i & y_i(w \cdot \boldsymbol{x_i}) \geq 1 - \xi_i \end{array}$$

```
from sklearn import svm

svmlinl1 = svm.LinearSVC(penalty='l1', dual=False)
# Remark: by default LinearSVC uses squared_hinge as loss

svmlinl1.fit(X, y)
y_pred_svmlinl1 = svmlinl1.predict(X)

errors =  y_pred_svmlinl1 != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
↪svmlinl1)))
print("Coef vector:")
print(svmlinl1.coef_)
```

```
Nb errors=20, error rate=0.20
Coef vector:
[[-0.01  0.37 -0.12  0.24  0.17  0.    0.    0.1  -0.16  0.13]]
```

Exercise

Compare predictions of Logistic regression (LR) and their SVM counterparts, ie.: L2 LR vs L2 SVM and L1 LR vs L1 SVM

- Compute the correlation between pairs of weights vectors.
- Compare the predictions of two classifiers using their decision function:

  - Give the equation of the decision function for a linear classifier, assuming that their is no intercept.
  - Compute the correlation decision function.
  - Plot the pairwise decision function of the classifiers.
- Conclude on the differences between Linear SVM and logistic regression.

5.5.12 Elastic-net classification ($\ell_1\ell_2$-regularization)

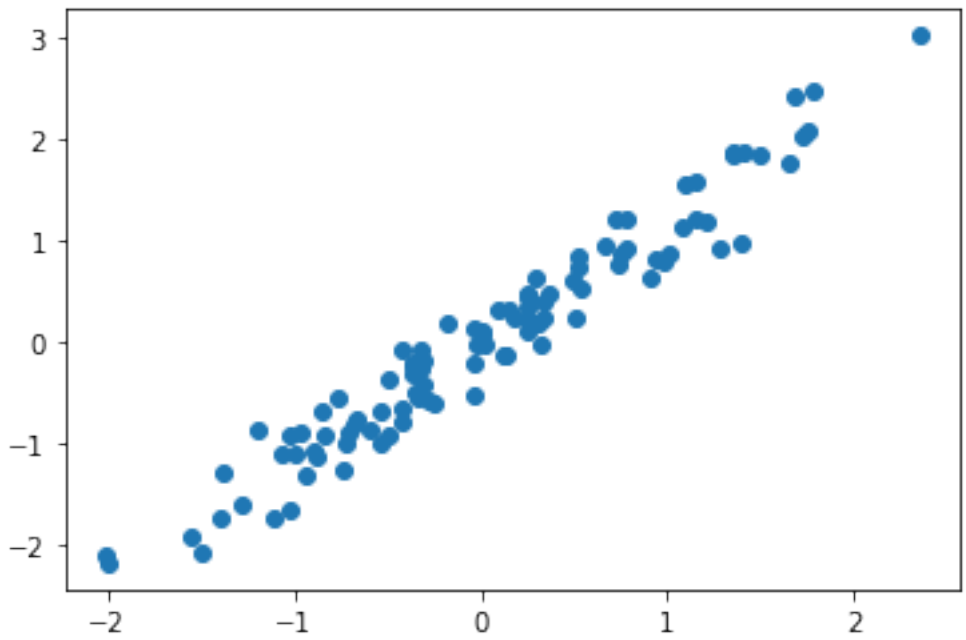
The **objective function** to be minimized is now the combination of the logistic loss $\log L(\boldsymbol{w})$ or the hinge loss with combination of L1 and L2 penalties. In the two-class case, using the 0/1 coding we obtain:

$$\min \text{ Logistic enet}(\boldsymbol{w}) = -\log \mathcal{L}(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) + \alpha \left(\rho \|\boldsymbol{w}\|_1 + (1-\rho) \|\boldsymbol{w}\|_2^2\right) \tag{5.46}$$

$$\min \text{ Hinge enet}(\boldsymbol{w}) = \text{Hinge loss}(\boldsymbol{w}) + \alpha \left(\rho \|\boldsymbol{w}\|_1 + (1-\rho) \|\boldsymbol{w}\|_2^2\right) \tag{5.47}$$

```
# Use SGD solver
enetlog = lm.SGDClassifier(loss="log", penalty="elasticnet",
                           alpha=0.1, l1_ratio=0.5, random_state=42)
enetlog.fit(X, y)

# Or saga solver:
# enetloglike = lm.LogisticRegression(penalty='elasticnet',
#                                     C=.1, l1_ratio=0.5, solver='saga')

enethinge = lm.SGDClassifier(loss="hinge", penalty="elasticnet",
                           alpha=0.1, l1_ratio=0.5, random_state=42)
enethinge.fit(X, y)

print("Hinge loss and logistic loss provide almost the same predictions.")
print("Confusion matrix")
metrics.confusion_matrix(enetlog.predict(X), enethinge.predict(X))

print("Decision_function log x hinge losses:")
_ = plt.plot(enetlog.decision_function(X),
             enethinge.decision_function(X), "o")
```

```
Hinge loss and logistic loss provide almost the same predictions.
Confusion matrix
Decision_function log x hinge losses:
```



5.5.13 Classification performance evaluation metrics

source: https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Imagine a study evaluating a new test that screens people for a disease. Each person taking the test either has or does not have the disease. The test outcome can be positive (classifying the person as having the disease) or negative (classifying the person as not having the disease). The test results for each subject may or may not match the subject's actual status. In that setting:

- True positive (TP): Sick people correctly identified as sick
- False positive (FP): Healthy people incorrectly identified as sick
- True negative (TN): Healthy people correctly identified as healthy
- False negative (FN): Sick people incorrectly identified as healthy
- **Accuracy** (ACC):

  ACC = (TP + TN) / (TP + FP + FN + TN)
- **Sensitivity** (SEN) or **recall** of the positive class or true positive rate (TPR) or hit rate:

  SEN = TP / P = TP / (TP+FN)
- **Specificity** (SPC) or **recall** of the negative class or true negative rate:

  SPC = TN / N = TN / (TN+FP)
- **Precision** or positive predictive value (PPV):

  PPV = TP / (TP + FP)
- **Balanced accuracy** (bACC):is a useful performance measure is the balanced accuracy which avoids inflated performance estimates on imbalanced datasets (Brodersen, et al. (2010). "The balanced accuracy and its posterior distribution"). It is defined as the arithmetic mean of sensitivity and specificity, or the average accuracy obtained on either class:

bACC = 1/2 * (SEN + SPC)

- F1 Score (or F-score) which is a weighted average of precision and recall are usefull to deal with imballaced datasets

The four outcomes can be formulated in a 2×2 contingency table or confusion matrix https://en.wikipedia.org/wiki/Sensitivity_and_specificity

For more precision see: http://scikit-learn.org/stable/modules/model_evaluation.html

```
from sklearn import metrics
y_pred = [0, 1, 0, 0]
y_true = [0, 1, 0, 1]

metrics.accuracy_score(y_true, y_pred)

# The overall precision an recall
metrics.precision_score(y_true, y_pred)
metrics.recall_score(y_true, y_pred)

# Recalls on individual classes: SEN & SPC
recalls = metrics.recall_score(y_true, y_pred, average=None)
recalls[0] # is the recall of class 0: specificity
recalls[1] # is the recall of class 1: sensitivity

# Balanced accuracy
b_acc = recalls.mean()

# The overall precision an recall on each individual class
p, r, f, s = metrics.precision_recall_fscore_support(y_true, y_pred)
```

Significance of classification rate

P-value associated to classification rate. Compared the number of correct classifications (=accuracy $\times N$) to the null hypothesis of Binomial distribution of parameters p (typically 50% of chance level) and N (Number of observations).

Is 65% of accuracy a significant prediction rate among 70 observations?

Since this is an exact, **two-sided** test of the null hypothesis, the p-value can be divided by 2 since we test that the accuracy is superior to the chance level.

```
import scipy.stats

acc, N = 0.65, 70
pval = scipy.stats.binom_test(x=int(acc * N), n=N, p=0.5) / 2
print(pval)
```

```
0.01123144774625465
```

Area Under Curve (AUC) of Receiver operating characteristic (ROC)

Some classifier may have found a good discriminative projection w. However if the threshold to decide the final predicted class is poorly adjusted, the performances will highlight an high specificity and a low sensitivity or the contrary.

In this case it is recommended to use the AUC of a ROC analysis which basically provide a measure of overlap of the two classes when points are projected on the discriminative axis. For more detail on ROC and AUC see:https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

```
score_pred = np.array([.1 ,.2, .3, .4, .5, .6, .7, .8])
y_true = np.array([0, 0, 0, 0, 1, 1, 1, 1])
thres = .9
y_pred = (score_pred > thres).astype(int)

print("With a threshold of %.2f, the rule always predict 0. Predictions:" % thres)
print(y_pred)
metrics.accuracy_score(y_true, y_pred)

# The overall precision an recall on each individual class
r = metrics.recall_score(y_true, y_pred, average=None)
print("Recalls on individual classes are:", r, "ie, 100% of specificity, 0% of␣
↪sensitivity")

# However AUC=1 indicating a perfect separation of the two classes
auc = metrics.roc_auc_score(y_true, score_pred)
print("But the AUC of %.2f demonstrate a good classes separation." % auc)
```

```
With a threshold of 0.90, the rule always predict 0. Predictions:
[0 0 0 0 0 0 0 0]
Recalls on individual classes are: [1. 0.] ie, 100% of specificity, 0% of␣
↪sensitivity
But the AUC of 1.00 demonstrate a good classes separation.
```

5.5.14 Imbalanced classes

Learning with discriminative (logistic regression, SVM) methods is generally based on minimizing the misclassification of training samples, which may be unsuitable for imbalanced datasets where the recognition might be biased in favor of the most numerous class. This problem can be addressed with a generative approach, which typically requires more parameters to be determined leading to reduced performances in high dimension.

Dealing with imbalanced class may be addressed by three main ways (see Japkowicz and Stephen (2002) for a review), resampling, reweighting and one class learning.

In **sampling strategies**, either the minority class is oversampled or majority class is undersampled or some combination of the two is deployed. Undersampling (Zhang and Mani, 2003) the majority class would lead to a poor usage of the left-out samples. Sometime one cannot afford such strategy since we are also facing a small sample size problem even for the majority class. Informed oversampling, which goes beyond a trivial duplication of minority class samples, requires the estimation of class conditional distributions in order to generate synthetic samples. Here generative models are required. An alternative, proposed in (Chawla et al., 2002) generate

samples along the line segments joining any/all of the k minority class nearest neighbors. Such procedure blindly generalizes the minority area without regard to the majority class, which may be particularly problematic with high-dimensional and potentially skewed class distribution.

Reweighting, also called cost-sensitive learning, works at an algorithmic level by adjusting the costs of the various classes to counter the class imbalance. Such reweighting can be implemented within SVM (Chang and Lin, 2001) or logistic regression (Friedman et al., 2010) classifiers. Most classifiers of Scikit learn offer such reweighting possibilities.

The `class_weight` parameter can be positioned into the `"balanced"` mode which uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as $N/(2N_k)$.

```
# dataset
X, y = datasets.make_classification(n_samples=500,
                            n_features=5,
                            n_informative=2,
                            n_redundant=0,
                            n_repeated=0,
                            n_classes=2,
                            random_state=1,
                            shuffle=False)

print(*["#samples of class %i = %i;" % (lev, np.sum(y == lev)) for lev in np.
↪unique(y)])

print('# No Reweighting balanced dataset')
lr_inter = linear_model.LogisticRegression(C=1)
lr_inter.fit(X, y)
p, r, f, s = metrics.precision_recall_fscore_support(y, lr_inter.predict(X))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => The predictions are balanced in sensitivity and specificity\n')

# Create imbalanced dataset, by subsampling sample of class 0: keep only 10% of
# class 0's samples and all class 1's samples.
n0 = int(np.rint(np.sum(y == 0) / 20))
subsample_idx = np.concatenate((np.where(y == 0)[0][:n0], np.where(y == 1)[0]))
Ximb = X[subsample_idx, :]
yimb = y[subsample_idx]
print(*["#samples of class %i = %i;" % (lev, np.sum(yimb == lev)) for lev in
        np.unique(yimb)])

print('# No Reweighting on imbalanced dataset')
lr_inter = linear_model.LogisticRegression(C=1)
lr_inter.fit(Ximb, yimb)
p, r, f, s = metrics.precision_recall_fscore_support(yimb, lr_inter.predict(Ximb))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => Sensitivity >> specificity\n')

print('# Reweighting on imbalanced dataset')
lr_inter_reweight = linear_model.LogisticRegression(C=1, class_weight="balanced")
lr_inter_reweight.fit(Ximb, yimb)
```

(continues on next page)

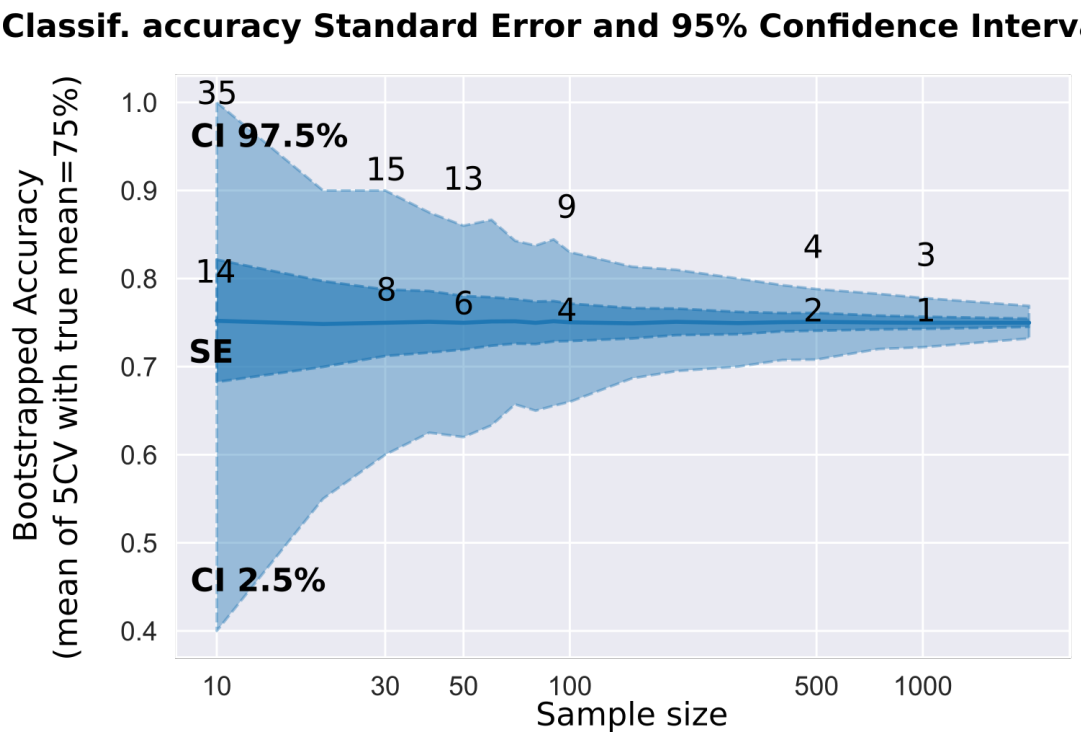
(continued from previous page)

```
p, r, f, s = metrics.precision_recall_fscore_support(yimb,
                                                     lr_inter_reweight.
↪predict(Ximb))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => The predictions are balanced in sensitivity and specificity\n')
```

```
#samples of class 0 = 250; #samples of class 1 = 250;
# No Reweighting balanced dataset
SPC: 0.940; SEN: 0.928
# => The predictions are balanced in sensitivity and specificity

#samples of class 0 = 12; #samples of class 1 = 250;
# No Reweighting on imbalanced dataset
SPC: 0.750; SEN: 0.996
# => Sensitivity >> specificity

# Reweighting on imbalanced dataset
SPC: 1.000; SEN: 0.980
# => The predictions are balanced in sensitivity and specificity
```

5.5.15 Confidence interval cross-validation

Confidence interval CI classification accuracy measured by cross-validation:



5.5.16 Exercise

Fisher linear discriminant rule

Write a class `FisherLinearDiscriminant` that implements the Fisher's linear discriminant analysis. This class must be compliant with the scikit-learn API by providing two methods: - `fit(X, y)` which fits the model and returns the object itself; - `predict(X)` which returns a vector of the predicted values. Apply the object on the dataset presented for the LDA.

5.6 Non-linear models

Here we focuse on non-linear models for classification. Nevertheless, each classification model has its regression counterpart.

```
# get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import train_test_split

np.set_printoptions(precision=2)
pd.set_option('precision', 2)
```

5.6.1 Support Vector Machines (SVM)

SVM are based kernel methods require only a user-specified kernel function $K(x_i, x_j)$, i.e., a **similarity function** over pairs of data points (x_i, x_j) into kernel (dual) space on which learning algorithms operate linearly, i.e. every operation on points is a linear combination of $K(x_i, x_j)$. Outline of the SVM algorithm:

1. Map points x into kernel space using a kernel function: $x \rightarrow K(x, .)$.
2. Learning algorithms operates linearly by dot product into high-kernel space $K(., x_i) \cdot K(., x_j)$.
    - Using the kernel trick (Mercer's Theorem) replaces dot product in high dimensional space by a simpler operation such that $K(., x_i) \cdot K(., x_j) = K(x_i, x_j)$. Thus we only need to compute a similarity measure for each pairs of point and store in a $N \times N$ Gram matrix.

- Finally, The learning process consist of estimating the $alpha_i$ of the decision function that maximises the hinge loss (of $f(x)$) plus some penalty when applied on all training points.

$$f(x) = \text{sign}\left(\sum_i^N \alpha_i\ y_i\ K(x_i, x)\right).$$

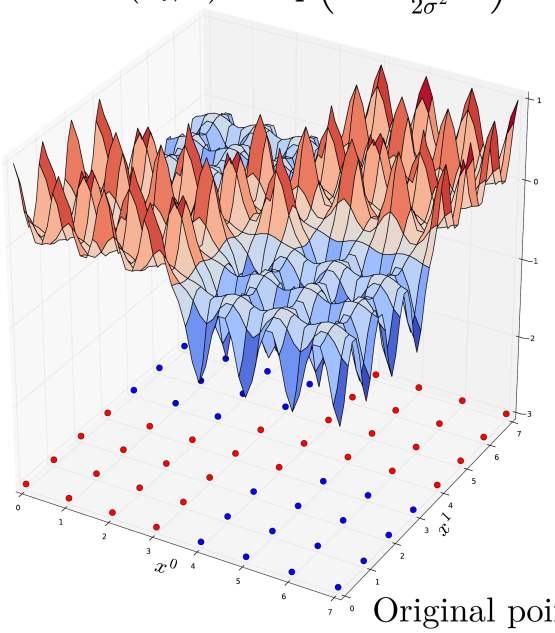
3. Predict a new point x using the decision function.

(1) Kernel mapping:

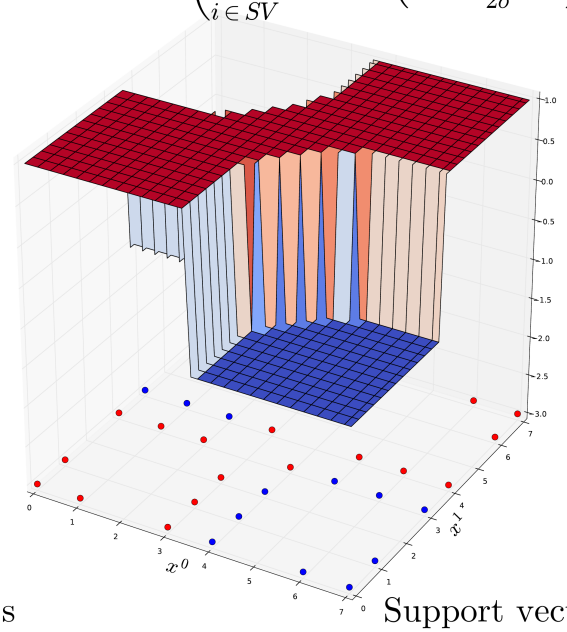
$$x \rightarrow K(x_i, x) = \exp\left(-\frac{||x_i - x_j||^2}{2\sigma^2}\right)$$

(2) Learn the decision function:

$$f(x) = sign\left(\sum_{i \in SV} \alpha_i y_i \exp\left(-\frac{||x_i - x_j||^2}{2\sigma^2}\right)\right)$$



Original points



Support vectors

Gaussian kernel (RBF, Radial Basis Function):

One of the most commonly used kernel is the Radial Basis Function (RBF) Kernel. For a pair of points x_i, x_j the RBF kernel is defined as:

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \tag{5.48}$$

$$= \exp\left(-\gamma\ \|x_i - x_j\|^2\right) \tag{5.49}$$

Where σ (or γ) defines the kernel width parameter. Basically, we consider a Gaussian function centered on each training sample x_i. it has a ready interpretation as a similarity measure as it decreases with squared Euclidean distance between the two feature vectors.

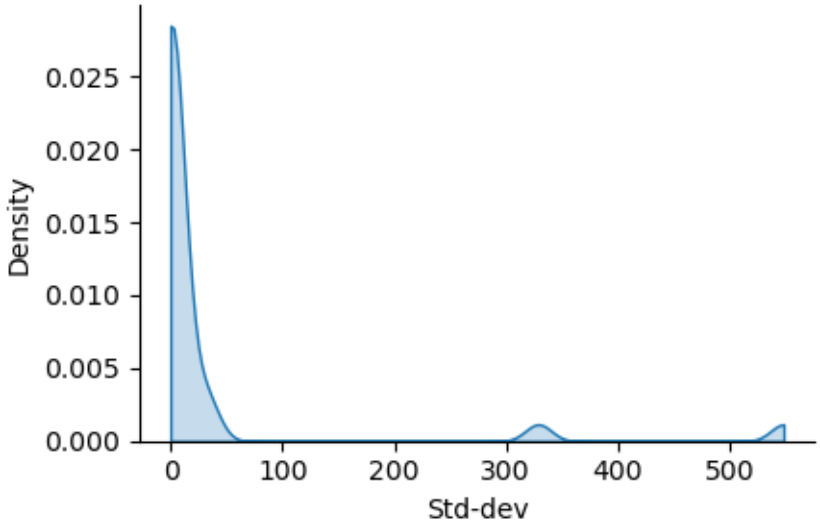
Non linear SVM also exists for regression problems.

dataset

```
X, y = datasets.load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5, stratify=y, random_state=42)
```

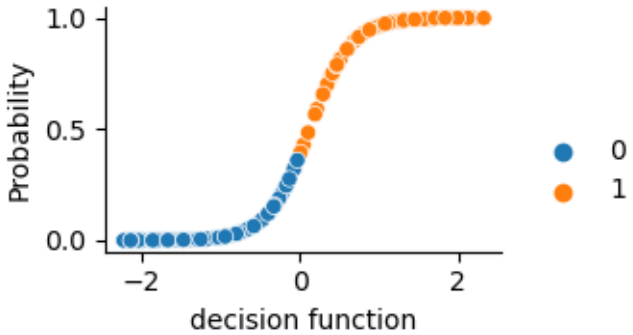
Preprocessing: unequal variance of input features, requires scaling for svm.

```
ax = sns.displot(x=X_train.std(axis=0), kind="kde", bw_adjust=.2, cut=0,
                 fill=True, height=3, aspect=1.5,)
_ = ax.set_xlabels("Std-dev").tight_layout()

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```



Fit-predict Probalility is a logistic of the decision_function

```
svm = SVC(kernel='rbf', probability=True).fit(X_train, y_train)
y_pred = svm.predict(X_test)
y_score = svm.decision_function(X_test)
y_prob = svm.predict_proba(X_test)[:, 1]

ax = sns.relplot(x=y_score, y=y_prob, hue=y_pred, height=2, aspect=1.5)
_ = ax.set_axis_labels("decision function", "Probability").tight_layout()
```




```
print("bAcc: %.2f, AUC: %.2f (AUC with proba: %.2f)" % (
      metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
      metrics.roc_auc_score(y_true=y_test, y_score=y_score),
      metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

(continues on next page)

(continued from previous page)

```
# Usefull internals: indices of support vectors within original X
np.all(X_train[svm.support_, :] == svm.support_vectors_)
```

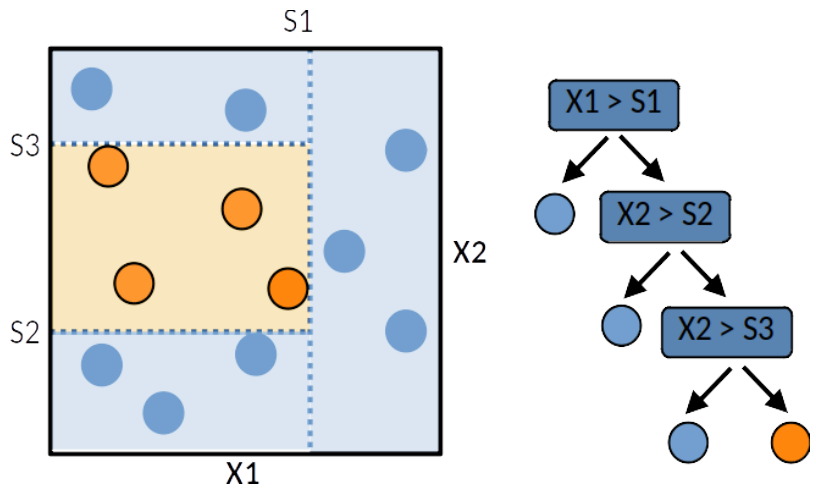
Out:

```
bAcc: 0.97, AUC: 0.99 (AUC with proba: 0.99)

True
```

5.6.2 Random forest

Decision tree

A tree can be "learned" by splitting the training dataset into subsets based on an features value test. Each internal node represents a "test" on an feature resulting on the split of the current sample. At each step the algorithm selects the feature and a cutoff value that maximises a given metric. Different metrics exist for regression tree (target is continuous) or classification tree (the target is qualitative). This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This general principle is implemented by many recursive partitioning tree algorithms.

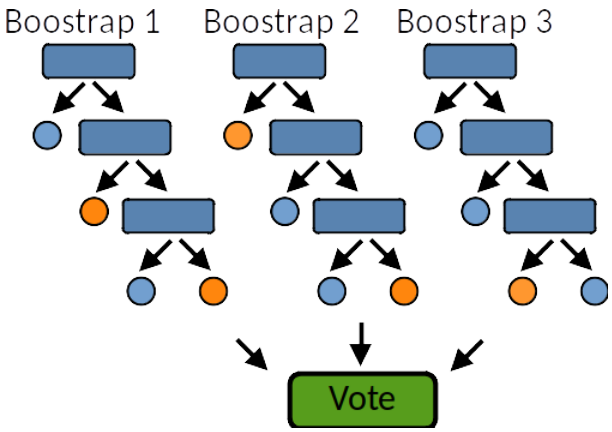


Decision trees are simple to understand and interpret however they tend to overfit the data. However decision trees tend to overfit the training set. Leo Breiman propose random forest to deal with this issue.

A single decision tree is usually overfits the data it is learning from because it learn from only one pathway of decisions. Predictions from a single decision tree usually don't make accurate predictions on new data.

Forest

A random forest is a meta estimator that fits a number of **decision tree learners** on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. Random forest models reduce the risk of overfitting by introducing randomness by:



- building multiple trees (n_estimators)
- drawing observations with replacement (i.e., a bootstrapped sample)
- splitting nodes on the best split among a random subset of the features selected at every node

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(n_estimators = 100)
forest.fit(X_train, y_train)

y_pred = forest.predict(X_test)
y_prob = forest.predict_proba(X_test)[:, 1]
```

```
print("bAcc: %.2f, AUC: %.2f " % (
      metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
      metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

Out:

```
bAcc: 0.93, AUC: 0.98
```

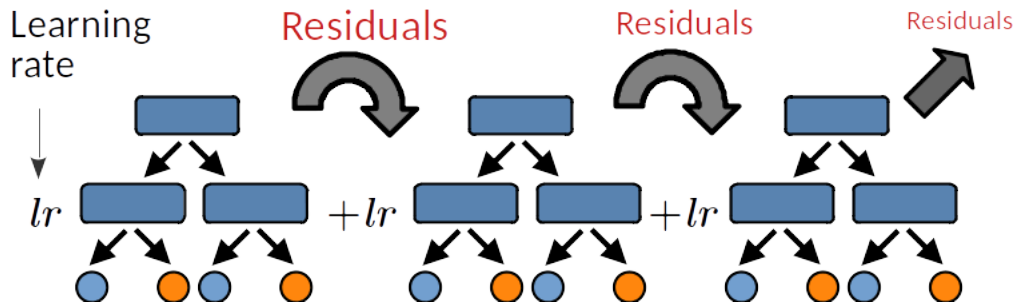
Extra Trees (Low Variance)

Extra Trees is like Random Forest, in that it builds multiple trees and splits nodes using random subsets of features, but with two key differences: it does not bootstrap observations (meaning it samples without replacement), and nodes are split on random splits, not best splits. So, in summary, ExtraTrees: builds multiple trees with bootstrap = False by default, which means it samples without replacement nodes are split based on random splits among a random subset of the features selected at every node In Extra Trees, randomness doesn't come from bootstrapping of data, but rather comes from the random splits of all observations. ExtraTrees is named for (Extremely Randomized Trees).

5.6.3 Gradient boosting

Gradient boosting is a meta estimator that fits a sequence of **weak learners**. Each learner aims to reduce the residuals (errors) produced by the previous learner. The two main hyper-parameters are:

- The **learning rate** (*lr*) controls over-fitting: decreasing the *lr* limits the capacity of a learner to overfit the residuals, ie, it slows down the learning speed and thus increases the **regularisation**.
- The **sub-sampling fraction** controls the fraction of samples to be used for fitting the learners. Values smaller than 1 leads to **Stochastic Gradient Boosting**. It thus controls for over-fitting reducing variance and incresing bias.




```
from sklearn.ensemble import GradientBoostingClassifier

gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,
                                subsample=0.5, random_state=0)
gb.fit(X_train, y_train)

y_pred = gb.predict(X_test)
y_prob = gb.predict_proba(X_test)[:, 1]

print("bAcc: %.2f, AUC: %.2f " % (
      metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
      metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

Out:

```
bAcc: 0.94, AUC: 0.98
```

Total running time of the script: (0 minutes 0.769 seconds)

5.7 Resampling methods

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn import datasets
```

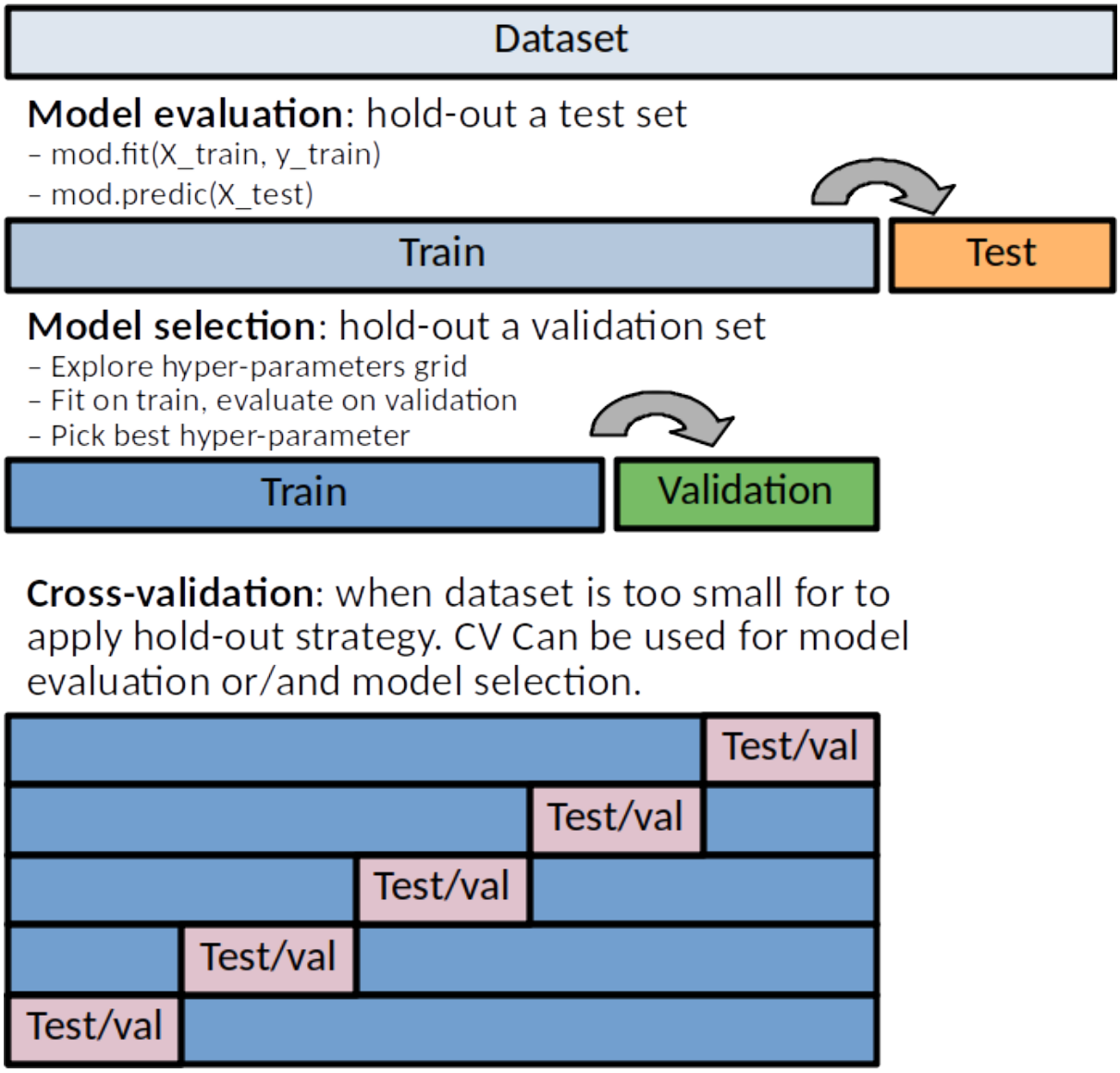
(continues on next page)

(continued from previous page)

```
import sklearn.linear_model as lm
from sklearn.model_selection import train_test_split, KFold, PredefinedSplit
from sklearn.model_selection import cross_val_score, GridSearchCV

import sklearn.metrics as metrics
X, y = datasets.make_regression(n_samples=100, n_features=100,
                                n_informative=10, random_state=42)
```

5.7.1 Train, validation and test sets

Machine learning algorithms overfit taining data. Predictive performances **MUST** be evaluated on independant hold-out dataset.



1. **Training dataset**: Dataset used to fit the model (set the model parameters like weights). The *training error* can be easily calculated by applying the statistical learning method to the observations used in its training. But because of overfitting, the **training error rate can dramatically underestimate the error** that would be obtained on new samples.

2. **Validation dataset**: Dataset used to provide an unbiased evaluation of a model fit on the training dataset while **tuning model hyperparameters**, ie. **model selection**. The validation error is the average error that results from a learning method to predict the response on a new (validation) samples that is, on samples that were not used in training the method.

3. **Test dataset**: Dataset used to provide an unbiased **evaluation of a final model** fitted on the training dataset. It is only used once a model is completely trained (using the train and validation sets).

What is the Difference Between Test and Validation Datasets? by [Jason Brownlee](https://machinelearningmastery.com/difference-test-validation-datasets/)

Thus the original dataset is generally split in a training, validation and a test data sets. Large training+validation set (80%) small test set (20%) might provide a poor estimation of the predictive performances (same argument stands for train vs validation samples). On the contrary, large test set and small training set might produce a poorly estimated learner. This is why, on situation where we cannot afford such split, cross-validation scheme can be use for model selection or/and for model evaluation.

If sample size is limited, train/validation/test split may not be possible. **Cross Validation (CV)** (see below) can be used to replace:

- Outer (train/test) split of model evaluation.
- Inner train/validation split of model selection (more frequent situation).
- Inner and outer splits, leading to two nested CV.

5.7.2 Split dataset in train/test sets for model evaluation

```
X_train, X_test, y_train, y_test =\
    train_test_split(X, y, test_size=0.25, shuffle=True, random_state=42)

mod = lm.Ridge(alpha=10)

mod.fit(X_train, y_train)

y_pred_test = mod.predict(X_test)
print("Test R2: %.2f" % metrics.r2_score(y_test, y_pred_test))
```

Out:

```
Test R2: 0.74
```

5.7.3 Train/validation/test splits: model selection and model evaluation

The **Grid search procedure** (*GridSearchCV*) performs a model selection of the best **hyper-parameters** α over a grid of possible values. Train set is "splitted (inner split) into train/validation sets.

Model selection with grid search procedure:

1. Fit the learner (ie. estimate **parameters** $\mathbf{\Omega}_k$) on training set: $\mathbf{X}_{train}, \mathbf{y}_{train} \rightarrow f_{\alpha_k, \mathbf{\Omega}_k}(.)$
2. Evaluate the model on the validation set and keep the hyper-parameter(s) that minimises the error measure $\alpha_* = \arg\min L(f_{\alpha_k, \mathbf{\Omega}_k}(\mathbf{X}_{val}), \mathbf{y}_{val})$
3. Refit the learner on all training + validation data, $\mathbf{X}_{train \cup val}, \mathbf{y}_{train \cup val}$, using the best hyper parameters (α_*): $\rightarrow f_{\alpha_*, \mathbf{\Omega}_*}(.)$

Model evaluation: on the test set: $L(f_{\alpha_*, \mathbf{\Omega}_*}(\mathbf{X}_{test}), \mathbf{y}_{test})$

```
train_idx, validation_idx = train_test_split(np.arange(X_train.shape[0]),
                                             test_size=0.25, shuffle=True,
                                             random_state=42)

split_inner = PredefinedSplit(test_fold=validation_idx)
print("Train set size: %i" % X_train[train_idx].shape[0])
print("Validation set size: %i" % X_train[validation_idx].shape[0])
print("Test set size: %i" % X_test.shape[0])

lm_cv = GridSearchCV(lm.Ridge(), {'alpha': 10. ** np.arange(-3, 3)},
                     cv=split_inner, n_jobs=5)

# Fit, indluding model selection with internal Train/validation split
lm_cv.fit(X_train, y_train)

# Predict
y_pred_test = lm_cv.predict(X_test)
print("Test R2: %.2f" % metrics.r2_score(y_test, y_pred_test))
```

Out:

```
Train set size: 56
Validation set size: 19
Test set size: 25
Test R2: 0.80
```

5.7.4 Cross-Validation (CV)

If sample size is limited, train/validation/test split may not be possible. **Cross Validation (CV)** can be used to replace train/validation split and/or train+validation / test split.

Cross-Validation scheme randomly divides the set of observations into K groups, or **folds**, of approximately equal size. The first fold is treated as a validation set, and the method $f()$ is fitted on the remaining union of *K - 1* folds: $(f(\boldsymbol{X}_{-K}, \boldsymbol{y}_{-K}))$. The measure of performance (the score function $\mathcal{S}$), either a error measure or an correct prediction measure is an average of a loss error or correct prediction measure, noted $\mathcal{L}$, between a true target value and the predicted target value. The score function is evaluated of the on the observations in the held-out fold. For each sample i we consider the model estimated $f(\boldsymbol{X}_{-k(i)}, \boldsymbol{y}_{-k(i)}$ on the data set without the group k that contains i noted *-k(i)*. This procedure is repeated K times; each time, a different group of observations is treated as a test set. Then we compare the predicted value $(f_{-k(i)}(\boldsymbol{x}_i) = \hat{y}_i)$ with true value y_i using a Error or Loss function $\mathcal{L}(y, \hat{y})$.

For 10-fold we can either average over 10 values (Macro measure) or concatenate the 10 experiments and compute the micro measures.

Two strategies [micro vs macro estimates](https://stats.stackexchange.com/questions/34611/meanscores-vs-scoreconcatenation-in-cross-validation):

- **Micro measure: average(individual scores)**: compute a score $\mathcal{S}$ for each sample and average over all samples. It is simillar to **average score(concatenation)**: an averaged score computed over all concatenated samples.
- **Macro measure mean(CV scores)** (the most commonly used method): compute a score $\mathcal{S}$ on each each fold k and average accross folds:

These two measures (an average of average vs. a global average) are generaly similar. They may differ slightly is folds are of different sizes. This validation scheme is known as the **K-Fold CV**. Typical choices of K are 5 or 10, [Kohavi 1995]. The extreme case where $K = N$ is known as **leave-one-out cross-validation, LOO-CV**.

CV for regression

Usually the error function $\mathcal{L}()$ is the r-squared score. However other function (MAE, MSE) can be used.

CV with explicit loop:

```
from sklearn.model_selection import KFold

estimator = lm.Ridge(alpha=10)

cv = KFold(n_splits=5, shuffle=True, random_state=42)
r2_train, r2_test = list(), list()

for train, test in cv.split(X):
    estimator.fit(X[train, :], y[train])
    r2_train.append(metrics.r2_score(y[train], estimator.predict(X[train, :])))
    r2_test.append(metrics.r2_score(y[test], estimator.predict(X[test, :])))
```

(continues on next page)

(continued from previous page)

```
print("Train r2:%.2f" % np.mean(r2_train))
print("Test  r2:%.2f" % np.mean(r2_test))
```

Out:

```
Train r2:0.99
Test  r2:0.67
```

Scikit-learn provides user-friendly function to perform CV:

cross_val_score(): single metric

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=estimator, X=X, y=y, cv=5)
print("Test  r2:%.2f" % scores.mean())

cv = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(estimator=estimator, X=X, y=y, cv=cv)
print("Test  r2:%.2f" % scores.mean())
```

Out:

```
Test  r2:0.73
Test  r2:0.67
```

cross_validate(): multi metric, + time, etc.

```
from sklearn.model_selection import cross_validate

scores = cross_validate(estimator=mod, X=X, y=y, cv=cv,
                        scoring=['r2', 'neg_mean_absolute_error'])

print("Test R2:%.2f; MAE:%.2f" % (scores['test_r2'].mean(),
                                  -scores['test_neg_mean_absolute_error'].mean()))
```

Out:

```
Test R2:0.67; MAE:55.27
```

CV for classification: stratifiy for the target label

With classification problems it is essential to sample folds where each set contains approximately the same percentage of samples of each target class as the complete set. This is called **stratification**. In this case, we will use `StratifiedKFold` with is a variation of k-fold which returns stratified folds. Usually the error function $L()$ are, at least, the sensitivity and the specificity. However other function could be used.

CV with explicit loop:

```
from sklearn.model_selection import StratifiedKFold

X, y = datasets.make_classification(n_samples=100, n_features=100, shuffle=True,
                                    n_informative=10, random_state=42)

mod = lm.LogisticRegression(C=1, solver='lbfgs')

cv = StratifiedKFold(n_splits=5)

# Lists to store scores by folds (for macro measure only)
bacc, auc = [], []

for train, test in cv.split(X, y):
    mod.fit(X[train, :], y[train])
    bacc.append(metrics.roc_auc_score(y[test], mod.decision_function(X[test, :])))
    auc.append(metrics.balanced_accuracy_score(y[test], mod.predict(X[test, :])))

print("Test AUC:%.2f; bACC:%.2f" % (np.mean(bacc), np.mean(auc)))
```

Out:

```
Test AUC:0.86; bACC:0.80
```

cross_val_score(): single metric

```
scores = cross_val_score(estimator=mod, X=X, y=y, cv=5)

print("Test  ACC:%.2f" % scores.mean())
```

Out:

```
Test  ACC:0.80
```

Provide your own CV and score

```
def balanced_acc(estimator, X, y, **kwargs):
    """Balanced acuracy scorer."""
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()

scores = cross_val_score(estimator=mod, X=X, y=y, cv=cv,
                         scoring=balanced_acc)
print("Test  bACC:%.2f" % scores.mean())
```

Out:

```
Test  bACC:0.80
```

cross_validate(): multi metric, + time, etc.

```
from sklearn.model_selection import cross_validate
```

(continues on next page)

(continued from previous page)

```
scores = cross_validate(estimator=mod, X=X, y=y, cv=cv,
                        scoring=['balanced_accuracy', 'roc_auc'])

print("Test AUC:%.2f; bACC:%.2f" % (scores['test_roc_auc'].mean(),
                                    scores['test_balanced_accuracy'].mean()))
```

Out:

```
Test AUC:0.86; bACC:0.80
```

5.7.5 Cross-validation for model selection

Combine CV and grid search: Re-split (inner split) train set into CV folds train/validation folds and build a *GridSearchCV* out of it:

```
# Outer split:
X_train, X_test, y_train, y_test =\
    train_test_split(X, y, test_size=0.25, shuffle=True, random_state=42)

cv_inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Cross-validation for model selection
lm_cv = GridSearchCV(lm.LogisticRegression(), {'C': 10. ** np.arange(-3, 3)},
                     cv=cv_inner, n_jobs=5)

# Fit, indluding model selection with internal CV
lm_cv.fit(X_train, y_train)

# Predict
y_pred_test = lm_cv.predict(X_test)
print("Test bACC: %.2f" % metrics.balanced_accuracy_score(y_test, y_pred_test))
```

Out:

```
Test bACC: 0.63
```

5.7.6 Cross-validation for both model (outer) evaluation and model (inner) selection

```
cv_outer = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Cross-validation for model (inner) selection
lm_cv = GridSearchCV(lm.Ridge(), {'alpha': 10. ** np.arange(-3, 3)},
                     cv=cv_inner, n_jobs=5)

# Cross-validation for model (outer) evaluation
scores = cross_validate(estimator=mod, X=X, y=y, cv=cv_outer,
```

(continues on next page)

(continued from previous page)

```
                        scoring=['balanced_accuracy', 'roc_auc'])

print("Test AUC:%.2f; bACC:%.2f, Time: %.2fs" % (scores['test_roc_auc'].mean(),
                                                 scores['test_balanced_accuracy'].mean(),
                                                 scores['fit_time'].sum()))
```

Out:

```
Test AUC:0.85; bACC:0.74, Time: 0.03s
```

5.7.7 Models with built-in cross-validation

Let sklearn select the best parameters over a default grid.

Classification

```
print("== Logistic Ridge (L2 penalty) ==")
mod_cv = lm.LogisticRegressionCV(class_weight='balanced', scoring='balanced_
↪accuracy',
                                 n_jobs=-1, cv=5)
scores = cross_val_score(estimator=mod_cv, X=X, y=y, cv=5)
print("Test  ACC:%.2f" % scores.mean())
```

Out:

```
== Logistic Ridge (L2 penalty) ==
Test  ACC:0.78
```

Regression

```
X, y, coef = datasets.make_regression(n_samples=50, n_features=100, noise=10,
                         n_informative=2, random_state=42, coef=True)

print("== Ridge (L2 penalty) ==")
model = lm.RidgeCV(cv=3)
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test  r2:%.2f" % scores.mean())

print("== Lasso (L1 penalty) ==")
model = lm.LassoCV(n_jobs=-1, cv=3)
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test  r2:%.2f" % scores.mean())

print("== ElasticNet (L1 penalty) ==")
model = lm.ElasticNetCV(l1_ratio=[.1, .5, .9], n_jobs=-1, cv=3)
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test  r2:%.2f" % scores.mean())
```

Out:

```
== Ridge (L2 penalty) ==
Test  r2:0.16
== Lasso (L1 penalty) ==
Test  r2:0.74
== ElasticNet (L1 penalty) ==
Test  r2:0.58
```

5.7.8 Random Permutations: sample the null distribution

A permutation test is a type of non-parametric randomization test in which the null distribution of a test statistic is estimated by randomly permuting the observations.

Permutation tests are highly attractive because they make no assumptions other than that the observations are independent and identically distributed under the null hypothesis.

1. Compute a observed statistic t_{obs} on the data.
2. Use randomization to compute the distribution of t under the null hypothesis: Perform N random permutation of the data. For each sample of permuted data, i the data compute the statistic t_i. This procedure provides the distribution of t under the null hypothesis H_0: $P(t|H_0)$
3. Compute the p-value = $P(t > t_{obs}|H_0)\,|\{t_i > t_{obs}\}|$, where $t_i' sinclude : math :$.

Example Ridge regression

Sample the distributions of r-squared and coefficients of ridge regression under the null hypothesis. Simulated dataset:

```
# Regression dataset where first 2 features are predictives
np.random.seed(0)
n_features = 5
n_features_info = 2
n_samples = 100
X = np.random.randn(100, 5)
beta = np.zeros(n_features)
beta[:n_features_info] = 1
Xbeta = np.dot(X, beta)
eps = np.random.randn(n_samples)
y = Xbeta + eps
```

5.7.9 Random permutations

```
# Fit model on all data (!! risk of overfit)
model = lm.RidgeCV()
model.fit(X, y)
print("Coefficients on all data:")
print(model.coef_)

# Random permutation loop
nperm = 1000  # !! Should be at least 1000 (to assess a p-value at 1%)
```

(continues on next page)

(continued from previous page)

```
scores_names = ["r2"]
scores_perm = np.zeros((nperm + 1, len(scores_names)))
coefs_perm = np.zeros((nperm + 1, X.shape[1]))

scores_perm[0, :] = metrics.r2_score(y, model.predict(X))
coefs_perm[0, :] = model.coef_

orig_all = np.arange(X.shape[0])
for perm_i in range(1, nperm + 1):
    model.fit(X, np.random.permutation(y))
    y_pred = model.predict(X).ravel()
    scores_perm[perm_i, :] = metrics.r2_score(y, y_pred)
    coefs_perm[perm_i, :] = model.coef_

# One-tailed empirical p-value
pval_pred_perm = np.sum(scores_perm >= scores_perm[0]) / scores_perm.shape[0]
pval_coef_perm = np.sum(coefs_perm >= coefs_perm[0, :], axis=0) / coefs_perm.
↪shape[0]

print("R2 p-value: %.3f" % pval_pred_perm)
print("Coeficients p-values:", np.round(pval_coef_perm, 3))
```

Out:

```
Coefficients on all data:
[ 1.02  1.06  0.21 -0.02 -0.05]
R2 p-value: 0.001
Coeficients p-values: [0.   0.   0.1  0.57 0.63]
```

Compute p-values corrected for multiple comparisons using FWER max-T (Westfall and Young, 1993) procedure.

```
pval_coef_perm_tmax = np.array([np.sum(coefs_perm.max(axis=1) >= coefs_perm[0, j])
                                for j in range(coefs_perm.shape[1])]) / coefs_
↪perm.shape[0]
print("P-values with FWER (Westfall and Young) correction")
print(pval_coef_perm_tmax)
```

Out:

```
P-values with FWER (Westfall and Young) correction
[0.   0.   0.41 0.98 0.99]
```

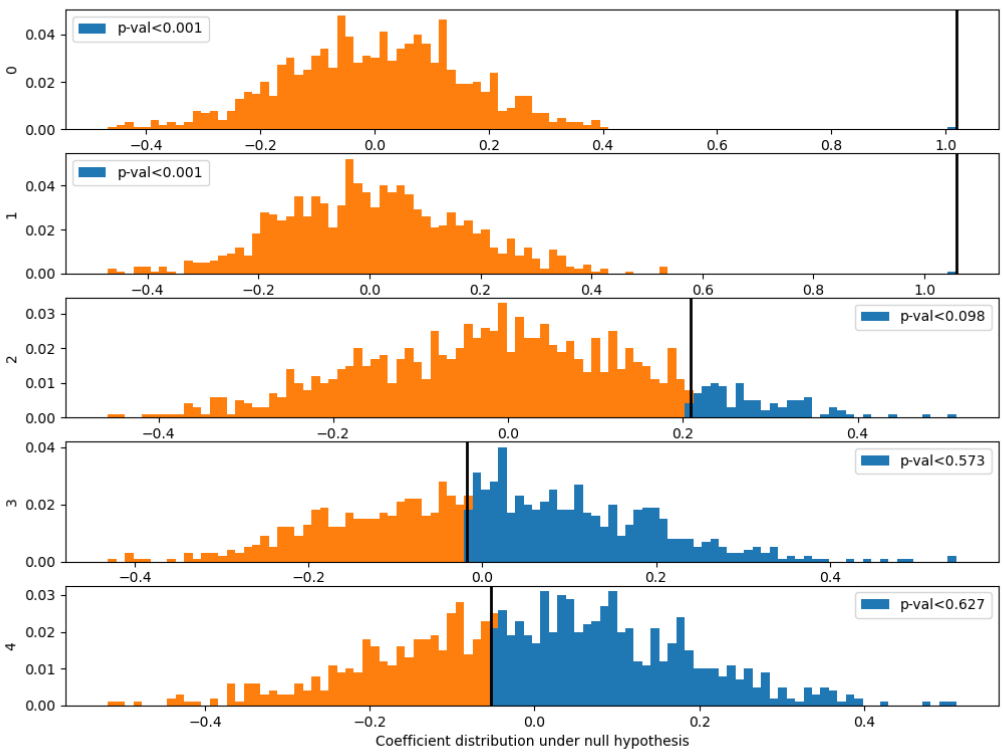
Plot distribution of third coefficient under null-hypothesis Coeffitients 0 and 1 are significantly different from 0.

```
def hist_pvalue(perms, ax, name):
    """Plot statistic distribution as histogram.

    Paramters
```

(continues on next page)

(continued from previous page)

```
    ---------
    perms: 1d array, statistics under the null hypothesis.
           perms[0] is the true statistic .
    """
    # Re-weight to obtain distribution
    pval = np.sum(perms >= perms[0]) / perms.shape[0]
    weights = np.ones(perms.shape[0]) / perms.shape[0]
    ax.hist([perms[perms >= perms[0]], perms], histtype='stepfilled',
             bins=100, label="p-val<%.3f" % pval,
             weights=[weights[perms >= perms[0]], weights])
    ax.axvline(x=perms[0], color="k", linewidth=2)#, label="observed statistic")
    ax.set_ylabel(name)
    ax.legend()
    return ax

n_coef = coefs_perm.shape[1]
fig, axes = plt.subplots(n_coef, 1, figsize=(12, 9))
for i in range(n_coef):
    hist_pvalue( coefs_perm[:, i], axes[i], str(i))

_ = axes[-1].set_xlabel("Coefficient distribution under null hypothesis")
```



Exercise

Given the logistic regression presented above and its validation given a 5 folds CV.

1. Compute the p-value associated with the prediction accuracy measured with 5CV using a permutation test.
2. Compute the p-value associated with the prediction accuracy using a parametric test.

5.7.10 Bootstrapping

Bootstrapping is a statistical technique which consists in generating sample (called bootstrap samples) from an initial dataset of size N by randomly drawing with replacement N observations. It provides sub-samples with the same distribution than the original dataset. It aims to:

1. Assess the variability (standard error, [confidence intervals.](https://sebastianraschka.com/blog/2016/model-evaluation-selection-part2.html#the-bootstrap-method-and-empirical-confidence-intervals)) of performances scores or estimated parameters (see Efron et al. 1986).
2. Regularize model by fitting several models on bootstrap samples and averaging their predictions (see Baging and random-forest).

A great advantage of bootstrap is its simplicity. It is a straightforward way to derive estimates of standard errors and confidence intervals for complex estimators of complex parameters of the distribution, such as percentile points, proportions, odds ratio, and correlation coefficients.

1. Perform B sampling, with replacement, of the dataset.
2. For each sample i fit the model and compute the scores.
3. Assess standard errors and confidence intervals of scores using the scores obtained on the B resampled dataset. Or, average models predictions.

References:

[Efron B, Tibshirani R. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. Stat Sci 1986;1:54–75](https://projecteuclid.org/download/pdf_1/euclid.ss/1177013815)

```
# Bootstrap loop
nboot = 100  # !! Should be at least 1000
scores_names = ["r2"]
scores_boot = np.zeros((nboot, len(scores_names)))
coefs_boot = np.zeros((nboot, X.shape[1]))

orig_all = np.arange(X.shape[0])
for boot_i in range(nboot):
    boot_tr = np.random.choice(orig_all, size=len(orig_all), replace=True)
    boot_te = np.setdiff1d(orig_all, boot_tr, assume_unique=False)
    Xtr, ytr = X[boot_tr, :], y[boot_tr]
    Xte, yte = X[boot_te, :], y[boot_te]
    model.fit(Xtr, ytr)
    y_pred = model.predict(Xte).ravel()
    scores_boot[boot_i, :] = metrics.r2_score(yte, y_pred)
    coefs_boot[boot_i, :] = model.coef_
```

Compute Mean, SE, CI Coeffitients 0 and 1 are significantly different from 0.

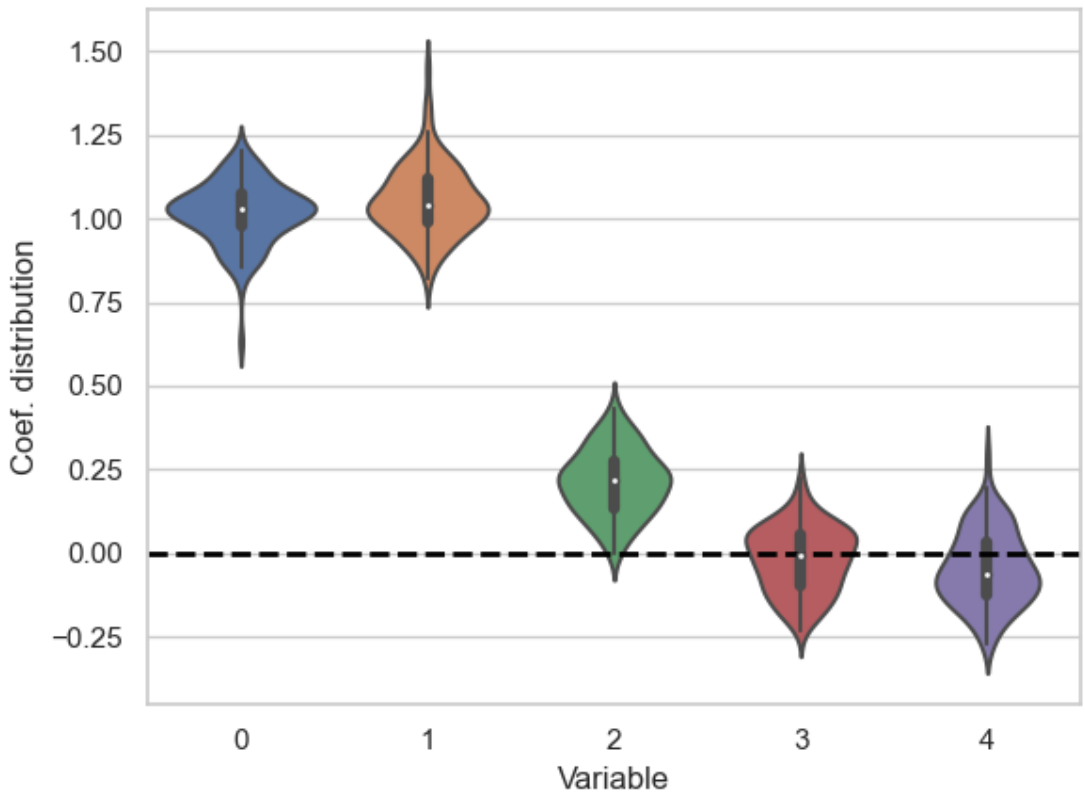
```
scores_boot = pd.DataFrame(scores_boot, columns=scores_names)
scores_stat = scores_boot.describe(percentiles=[.975, .5, .025])

print("r-squared: Mean=%.2f, SE=%.2f, CI=(%.2f %.2f)" %      tuple(scores_stat.
↪loc[["mean", "std", "2.5%", "97.5%"], "r2"]))

coefs_boot = pd.DataFrame(coefs_boot)
coefs_stat = coefs_boot.describe(percentiles=[.975, .5, .025])
print("Coefficients distribution")
print(coefs_stat)
```

Out:

```
r-squared: Mean=0.59, SE=0.09, CI=(0.40 0.73)
Coefficients distribution
            0       1         2       3       4
count  100.00  100.00  1.00e+02  100.00  100.00
mean     1.02    1.05  2.12e-01   -0.02   -0.05
std      0.09    0.11  9.75e-02    0.10    0.11
min      0.63    0.82 -2.69e-03   -0.23   -0.27
2.5%     0.86    0.88  3.27e-02   -0.20   -0.23
50%      1.03    1.04  2.17e-01   -0.01   -0.06
97.5%    1.17    1.29  3.93e-01    0.15    0.14
max      1.20    1.45  4.33e-01    0.22    0.29
```

Plot coefficient distribution

```
df = pd.DataFrame(coefs_boot)
staked = pd.melt(df, var_name="Variable", value_name="Coef. distribution")
sns.set_theme(style="whitegrid")
ax = sns.violinplot(x="Variable", y="Coef. distribution", data=staked)
_ = ax.axhline(0, ls='--', lw=2, color="black")
```



5.7.11 Parallel computation with joblib

Dataset

```
import numpy as np
from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.model_selection import StratifiedKFold
X, y = datasets.make_classification(n_samples=20, n_features=5, n_informative=2,␣
↪random_state=42)
cv = StratifiedKFold(n_splits=5)
```

Use *cross_validate* function

```
from sklearn.model_selection import cross_validate

estimator = lm.LogisticRegression(C=1, solver='lbfgs')
cv_results = cross_validate(estimator, X, y, cv=cv, n_jobs=5)
print(np.mean(cv_results['test_score']), cv_results['test_score'])
```

Out:

```
0.8 [0.5 0.5 1.  1.  1. ]
```

Sequential computation

If we want have full control of the operations performed within each fold (retrieve the models parameters, etc.). We would like to parallelize the folowing sequetial code:

```
# In[22]:


estimator = lm.LogisticRegression(C=1, solver='lbfgs')
y_test_pred_seq = np.zeros(len(y)) # Store predictions in the original order
coefs_seq = list()
for train, test in cv.split(X, y):
    X_train, X_test, y_train, y_test = X[train, :], X[test, :], y[train], y[test]
    estimator.fit(X_train, y_train)
    y_test_pred_seq[test] = estimator.predict(X_test)
    coefs_seq.append(estimator.coef_)

test_accs = [metrics.accuracy_score(y[test], y_test_pred_seq[test]) for train,␣
↪test in cv.split(X, y)]
print(np.mean(test_accs), test_accs)
coefs_cv = np.array(coefs_seq)
print(coefs_cv)

print(coefs_cv.mean(axis=0))
print("Std Err of the coef")
print(coefs_cv.std(axis=0) / np.sqrt(coefs_cv.shape[0]))
```

Out:

```
0.8 [0.5, 0.5, 1.0, 1.0, 1.0]
[[[-0.88  0.63  1.19 -0.31 -0.38]]

 [[-0.75  0.62  1.1   0.2  -0.4 ]]

 [[-0.96  0.51  1.12  0.08 -0.26]]

 [[-0.86  0.52  1.07 -0.11 -0.29]]

 [[-0.9   0.51  1.09 -0.25 -0.28]]]
[[-0.87  0.56  1.11 -0.08 -0.32]]
Std Err of the coef
[[0.03 0.02 0.02 0.09 0.03]]
```

5.7.12 Parallel computation with joblib

```
from joblib import Parallel, delayed
from sklearn.base import is_classifier, clone

def _split_fit_predict(estimator, X, y, train, test):
    X_train, X_test, y_train, y_test = X[train, :], X[test, :], y[train], y[test]
    estimator.fit(X_train, y_train)
    return [estimator.predict(X_test), estimator.coef_]

estimator = lm.LogisticRegression(C=1, solver='lbfgs')

parallel = Parallel(n_jobs=5)
cv_ret = parallel(
    delayed(_split_fit_predict)(
        clone(estimator), X, y, train, test)
    for train, test in cv.split(X, y))

y_test_pred_cv, coefs_cv = zip(*cv_ret)

# Retrieve predictions in the original order
y_test_pred = np.zeros(len(y))
for i, (train, test) in enumerate(cv.split(X, y)):
    y_test_pred[test] = y_test_pred_cv[i]

test_accs = [metrics.accuracy_score(y[test], y_test_pred[test]) for train, test␣
↪in cv.split(X, y)]
print(np.mean(test_accs), test_accs)
```

Out:

```
0.8 [0.5, 0.5, 1.0, 1.0, 1.0]
```

Test same predictions and same coeficients

```
assert np.all(y_test_pred == y_test_pred_seq)
assert np.allclose(np.array(coefs_cv).squeeze(), np.array(coefs_seq).squeeze())
```

Total running time of the script: (0 minutes 4.727 seconds)

5.8 Ensemble learning: bagging, boosting and stacking

These methods are **Ensemble learning** techniques. These models are machine learning paradigms where multiple models (often called "weak learners") are trained to **solve the same problem** and **combined** to get **better** results. The main hypothesis is that when **weak models** are **correctly combined** we can obtain **more accurate and/or robust models**.

5.8.1 Single weak learner

In machine learning, no matter if we are facing a classification or a regression problem, the choice of the model is extremely important to have any chance to obtain good results. This choice can depend on many variables of the problem: quantity of data, dimensionality of the space, distribution hypothesis...

A **low bias and a low variance**, although they most often vary in opposite directions, are the **two most fundamental features** expected for a model. Indeed, to be able to "solve" a problem, we want our model to have **enough degrees of freedom** to resolve the underlying complexity of the data we are working with, but we also want it to have **not too much degrees of freedom** to avoid **high variance** and be **more robust**. This is the well known **bias-variance tradeoff**.

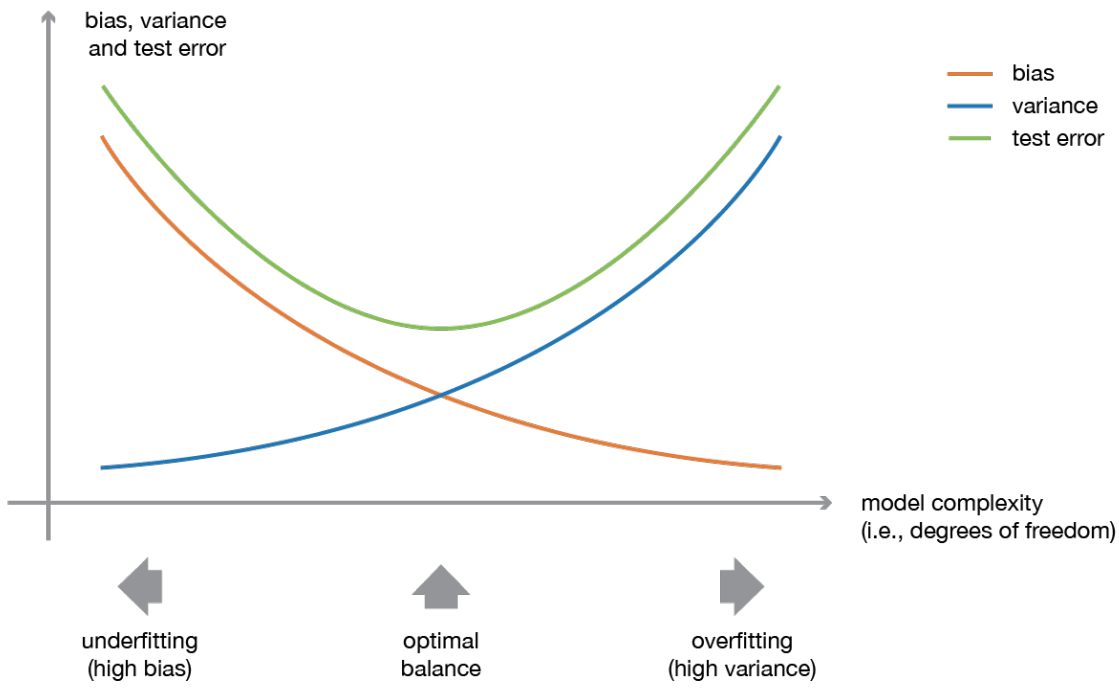


Fig. 13: towardsdatascience blog

Illustration of the bias-variance tradeoff.

In ensemble learning theory, we call **weak learners** (or **base models**) models that can be used as building blocks for designing more complex models by **combining several of them**. Most of the time, these basics models **perform not so well** by themselves either because they have a **high bias** (low degree of freedom models, for example) **or** because they have **too much variance** to be robust (high degree of freedom models, for example). Then, the idea of ensemble methods is to combining several of them together in order to create a **strong learner** (or **ensemble model**) that achieves better performances.

Usually, ensemble models are used in order to :

- **decrease the variance** for **bagging** (Bootstrap Aggregating) technique
- **reduce bias** for the boosting technique
- **improving the predictive force** for stacking technique.

To understand these techniques, first, we will explore what is boostrapping and its different **hypothesis**.

5.8.2 Bagging

In **parallel methods** we fit the different considered learners independently from each others and, so, it is possible to train them concurrently. The most famous such approach is "bagging" (standing for "**b**ootstrap **agg**regat**ing**") that aims at producing an ensemble model that is **more robust** than the individual models composing it.

When training a model, no matter if we are dealing with a classification or a regression problem, we obtain a function that takes an input, returns an output and that is defined with respect to the training dataset.

The idea of bagging is then simple: we want to fit several independent models and "average" their predictions in order to obtain a model with a lower variance. However, we can't, in practice, fit fully independent models because it would require too much data. So, we rely on the good "approximate properties" of bootstrap samples (representativity and independence) to fit models that are almost independent.

First, we create **multiple bootstrap samples** so that each new bootstrap sample will act as another (almost) independent dataset drawn from true distribution. Then, we can **fit a weak learner for each of these samples and finally aggregate them such that we kind of "average" their outputs** and, so, obtain an ensemble model with **less variance** that its components. Roughly speaking, as the bootstrap samples are approximatively **independent and identically distributed (i.i.d.)**, so are the learned base models. Then, **"averaging" weak learners outputs** do not change the expected answer but reduce its variance.

So, assuming that we have L bootstrap samples (approximations of L independent datasets) of size B denoted

$$\{z_1^1, z_2^1, ..., z_B^1\}, \{z_1^2, z_2^2, ..., z_B^2\}, ..., \{z_1^L, z_2^L, ..., z_B^L\} \qquad z_b^l \equiv b\text{-th observation of the } l\text{-th bootstrap sample}$$

Fig. 14: Medium Science Blog

Each $\{....\}$ is a bootstrap sample of B observation

we can fit L almost independent weak learners (one on each dataset)

$$w_1(.), w_2(.), ..., w_L(.)$$

Fig. 15: Medium Science Blog

and then aggregate them into some kind of averaging process in order to get an ensemble model with a lower variance. For example, we can define our strong model such that

$$s_L(.) = \frac{1}{L}\sum_{l=1}^{L} w_l(.) \qquad \text{(simple average, for regression problem)}$$

$$s_L(.) = \arg\max_k [card(l|w_l(.) = k)] \qquad \text{(simple majority vote, for classification problem)}$$

Fig. 16: Medium Science Blog

There are several possible ways to aggregate the multiple models fitted in parallel. - For a **regression problem**, the outputs of individual models can literally be **averaged** to obtain the output of the ensemble model. - For **classification problem** the class outputted by each model can be seen as a **vote** and the class that receives the **majority of the votes** is returned by the ensemble model (this is called **hard-voting**). Still for a classification problem, we can **also** consider the **probabilities of each classes** returned by all the models, **average these probabilities** and keep the class with the **highest average probability** (this is called **soft-voting**). –> Averages or votes can either be simple or weighted if any relevant weights can be used.

Finally, we can mention that one of the big advantages of bagging is that **it can be parallelised**. As the different models are fitted independently from each others, intensive parallelisation techniques can be used if required.

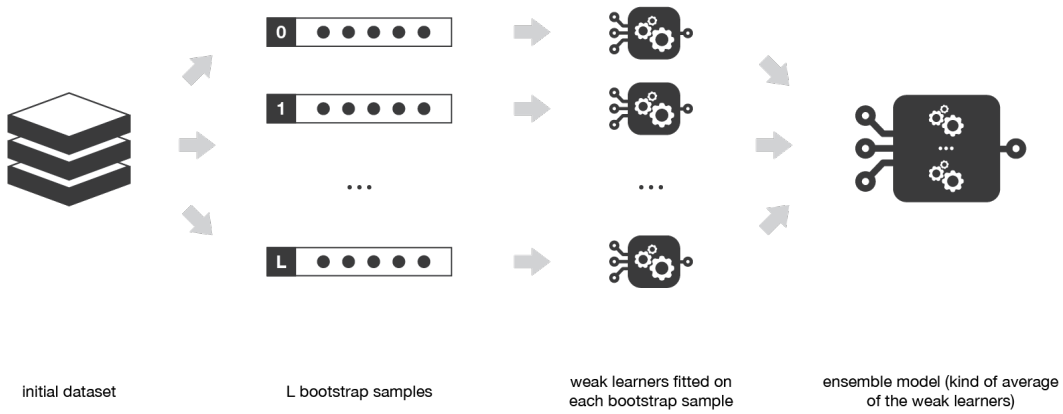


Fig. 17: Medium Science Blog

Bagging consists in fitting several base models on different bootstrap samples and build an ensemble model that "average" the results of these weak learners.

Question : - Can you name an algorithms based on Bagging technique , Hint : **leaf**

Examples

Here, we are trying some example of **stacking**

- Bagged Decision Trees for Classification

```
import pandas
from sklearn import model_selection
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier


names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv("https://raw.githubusercontent.com/jbrownlee/Datasets/
↪master/pima-indians-diabetes.data.csv",names=names)

array = dataframe.values
x = array[:,0:8]
```

(continues on next page)

(continued from previous page)

```
y = array[:,8]
max_features = 3

kfold = model_selection.KFold(n_splits=10, random_state=2020)
rf = DecisionTreeClassifier(max_features=max_features)
num_trees = 100

model = BaggingClassifier(base_estimator=rf, n_estimators=num_trees, random_
↪state=2020)
results = model_selection.cross_val_score(model, x, y, cv=kfold)
print("Accuracy: %0.2f (+/- %0.2f)" % (results.mean(), results.std()))
```

- Random Forest Classification

```
import pandas
from sklearn import model_selection
from sklearn.ensemble import RandomForestClassifier

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv("https://raw.githubusercontent.com/jbrownlee/Datasets/
↪master/pima-indians-diabetes.data.csv",names=names)

array = dataframe.values
x = array[:,0:8]
y = array[:,8]

kfold = model_selection.KFold(n_splits=10, random_state=2020)
rf = DecisionTreeClassifier()
num_trees = 100
max_features = 3

kfold = model_selection.KFold(n_splits=10, random_state=2020)
model = RandomForestClassifier(n_estimators=num_trees, max_features=max_features)
results = model_selection.cross_val_score(model, x, y, cv=kfold)
print("Accuracy: %0.2f (+/- %0.2f)" % (results.mean(), results.std()))
```

Both of these algorithms will print, **Accuracy: 0.77 (+/- 0.07)**. They are **equivalent**.

5.8.3 Boosting

In **sequential methods** the different combined weak models are **no longer** fitted **independently** from each others. The idea is to fit models **iteratively** such that the training of model at a given step depends on the models fitted at the previous steps. "Boosting" is the most famous of these approaches and it produces an ensemble model that is in general **less biased** than the weak learners that compose it.

Boosting methods work in the same **spirit** as **bagging** methods: we build a **family of models** that are **aggregated** to obtain a strong learner that performs better.

However, **unlike bagging that mainly aims at reducing variance, boosting is a technique that consists in fitting sequentially multiple weak learners in a very adaptative** way: **each**

model in the sequence is fitted giving more importance to observations in the dataset that were badly handled by the previous models in the sequence. Intuitively, each new model **focus its efforts on the most difficult observations** to fit up to now, so that we obtain, **at the end of the process**, a strong learner with **lower bias** (even if we can notice that boosting can also have the effect of reducing variance).

–> Boosting, like bagging, can be used for regression as well as for classification problems.

Being **mainly focused at reducing bias**, the **base models** that are often considered for boosting are* *models with low variance but high bias**. For example, if we want to use**trees**as our base models, we will choose**most of the time shallow decision trees with only a few depths.**

Another important reason that motivates the use of low variance but high bias models as weak learners for boosting is that these models are in general less computationally expensive to fit (few degrees of freedom when parametrised). Indeed, as computations to fit the different models **can't be done in parallel** (unlike bagging), it could become too expensive to fit sequentially several complex models.

Once the weak learners have been chosen, we still need to define **how** they will be sequentially **fitted** and **how** they will be **aggregated**. We will discuss these questions in the two following subsections, describing more especially two important boosting algorithms: **adaboost** and **gradient boosting**.

In a nutshell, these two meta-algorithms **differ** on how they **create and aggregate** the weak learners during the sequential process. **Ada**ptive **boost**ing **updates the weights attached to each of the training dataset observations** whereas **gradient boosting updates the value of these observations**. This main difference comes from the way both methods try to **solve the optimisation** problem of finding the best model that can be written as a weighted sum of weak learners.

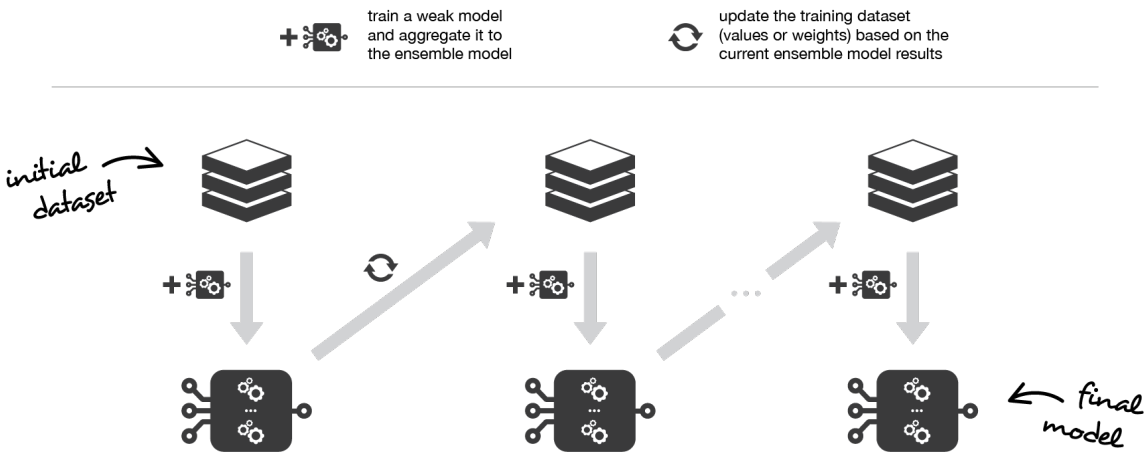


Fig. 18: Medium Science Blog

Boosting consists in, iteratively, fitting a weak learner, aggregate it to the ensemble model and "update" the training dataset to better take into account the strengths and weakness of the current ensemble model when fitting the next base model.

1/ Adaptative boosting

In adaptative boosting (often called "adaboost"), we try to define our ensemble model as a weighted sum of L weak learners

$$s_L(.) = \sum_{l=1}^{L} c_l \times w_l(.) \qquad \text{where } c_l\text{'s are coefficients and } w_l\text{'s are weak learners}$$

Fig. 19: Medium Science Blog

Finding the best ensemble model with this form is a difficult optimisation problem. Then, **instead** of trying to **solve it in one single shot** (finding all the coefficients and weak learners that give the best overall additive model), we make use of an **iterative optimisation process** that is much more tractable, even if it can lead to a sub-optimal solution. More especially, we **add** the **weak learners one by one**, looking at each iteration for the **best possible pair** (**coefficient, weak learner**) to add to the current ensemble model. In other words, we define recurrently the (s_l)'s such that

$$s_l(.) = s_{l-1}(.) + c_l \times w_l(.)$$

Fig. 20: towardsdatascience Blog

where c_l and w_l are chosen such that s_l is the model that fit the best the training data and, so, that is the **best possible improvement over s_(l-1)**. We can then denote

$$(c_l, w_l(.)) = \underset{c,w(.)}{\arg\min} E(s_{l-1}(.) + c \times w(.)) = \underset{c,w(.)}{\arg\min} \sum_{n=1}^{N} e(y_n, s_{l-1}(x_n) + c \times w(x_n))$$

Fig. 21: towardsdatascience Blog

where E(.) is the fitting error of the given model and e(.,.) is the loss/error function. Thus, instead of optimising "globally" over all the L models in the sum, we **approximate the optimum by optimising "locally"** building and adding the weak learners to the strong model one by one.

More especially, when considering a **binary classification**, we can show that the adaboost algorithm can be re-written into a process that proceeds as follow. First, it updates the **observations weights** in the dataset and train a new weak learner with a **special focus** given to the **observations misclassified** by the current ensemble model. Second, it adds the weak learner to the weighted sum according to an update coefficient that expresse the performances of this weak model: **the better a weak learner performs, the more it contributes to the strong learner**.

So, assume that we are facing a **binary classification** problem, with **N** observations in our dataset and we want to use **adaboost** algorithm with a given family of weak models. At the **very beginning** of the algorithm (first model of the sequence), **all the observations have the same weights 1/N**. Then, we **repeat L times (for the L learners in the sequence)** the following steps:

fit the best possible weak model with the current observations weights

compute the value of the update coefficient that is some kind of scalar evaluation metric of the weak learner that indicates how much this weak learner should be taken into account into the ensemble model

update the strong learner by adding the new weak learner multiplied by its update coefficient

compute new observations weights that expresse which observations we would like to focus on at the next iteration (weights of observations wrongly predicted by the aggregated model increase and weights of the correctly predicted observations decrease)

Repeating these steps, we have then build **sequentially** our **L models** and **aggregate** them into a **simple linear combination weighted by coefficients expressing the performance of each learner**.

Notice that there exists variants of the initial adaboost algorithm such that LogitBoost (classification) or L2Boost (regression) that mainly differ by their choice of loss function.

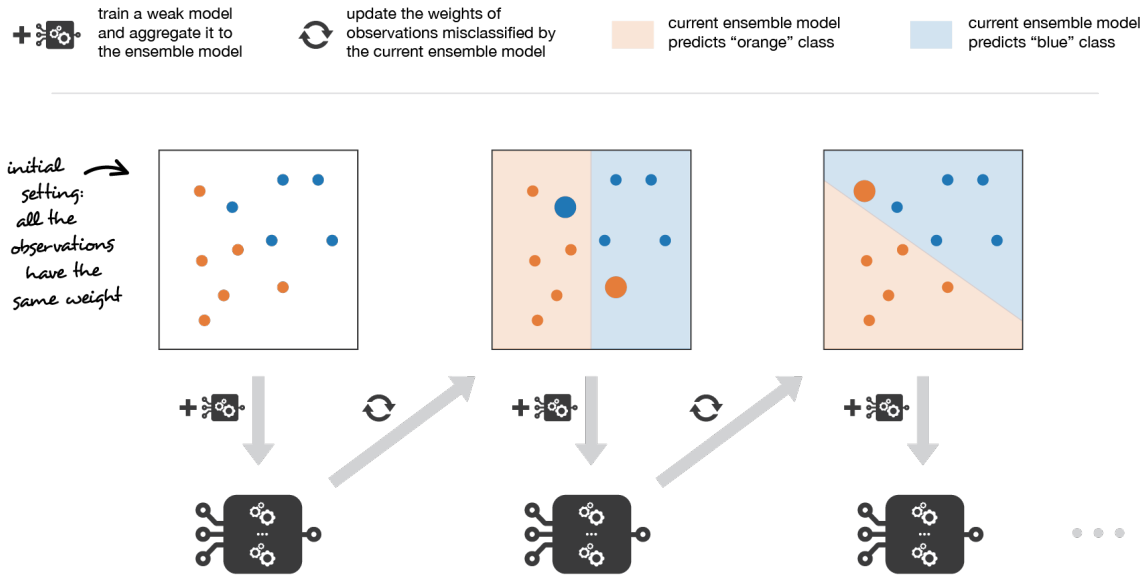


Fig. 22: Medium Science Blog

Adaboost updates weights of the observations at each iteration. Weights of well classified observations decrease relatively to weights of misclassified observations. Models that perform better have higher weights in the final ensemble model.

2/ Gradient boosting

In **gradient boosting**, the ensemble model we try to build is **also a weighted sum of weak learners**

$$s_L(.) = \sum_{l=1}^{L} c_l \times w_l(.) \qquad \text{where } c_l\text{'s are coefficients and } w_l\text{'s are weak learners}$$

Fig. 23: Medium Science Blog

Just **as** we mentioned for **adaboost**, finding the **optimal model under this form is too difficult and an iterative approach is required**. The **main difference** with **adaptative boosting** is in the **definition of the sequential optimisation process.** Indeed, **gradient boosting** casts the problem into a **gradient descent one**: at **each iteration** we fit a **weak learner** to the **opposite**

of the **gradient of the current fitting error with respect to the current ensemble model**. Let's try to clarify this last point. First, theoretical gradient descent process over the ensemble model can be written

$$s_l(.) = s_{l-1}(.) - c_l \times \nabla_{s_{l-1}} E(s_{l-1})(.)$$

Fig. 24: Medium Science Blog

where E(.) is the fitting error of the given model, c_l is a coefficient corresponding to the step size and

$$-\nabla_{s_{l-1}} E(s_{l-1})(.)$$

Fig. 25: Medium Science Blog

This entity is the **opposite of the gradient of the fitting error with respect to the ensemble model at step l-1**. This opposite of the gradient is a function that can, in practice, only be evaluated for observations in the training dataset (for which we know inputs and outputs): these evaluations are called pseudo-residuals attached to each observations. Moreover, even if we know for the observations the values of these pseudo-residuals, we don't want to add to our ensemble model any kind of function: we only want to add a new instance of weak model. So, the natural thing to do is to fit a weak learner to the pseudo-residuals computed for each observation. Finally, the coefficient c_l is computed following a one dimensional optimisation process (line-search to obtain the best step size c_l).

So, assume that we want to use gradient boosting technique with a given family of weak models. At the very beginning of the algorithm (first model of the sequence), the pseudo-residuals are set equal to the observation values. Then, we repeat L times (for the L models of the sequence) the following steps:

fit the best possible weak model to pseudo-residuals (approximate the opposite of the gradient with respect to the current strong learner)

compute the value of the optimal step size that defines by how much we update the ensemble model in the direction of the new weak learner

update the ensemble model by adding the new weak learner multiplied by the step size (make a step of gradient descent)

compute new pseudo-residuals that indicate, for each observation, in which direction we would like to update next the ensemble model predictions

Repeating these steps, we have then build sequentially our L models and aggregate them **following a gradient descent approach**. Notice that, **while adaptative boosting tries to solve at each iteration exactly the "local" optimisation problem (find the best weak learner and its coefficient to add to the strong model), gradient boosting uses instead a gradient descent approach and can more easily be adapted to large number of loss functions**. Thus, **gradient boosting** can be considered as a **generalization of adaboost to arbitrary differentiable loss functions**.

Note There is an algorithm which gained huge popularity after a **Kaggle's competitions**. It is **XGBoost (Extreme Gradient Boosting)**. This is a gradient boosting algorithm which has more

flexibility (varying number of terminal nodes and left weights) parameters to avoid sub-learners correlations. Having these important qualities, **XGBOOST** is one of the most used algorithm in data science. **LIGHTGBM is a recent implementation** of this algorithm. It was published by **Microsoft** and it gives us the same scores (if parameters are equivalents) but it runs **quicker** than a classic **XGBOOST**.

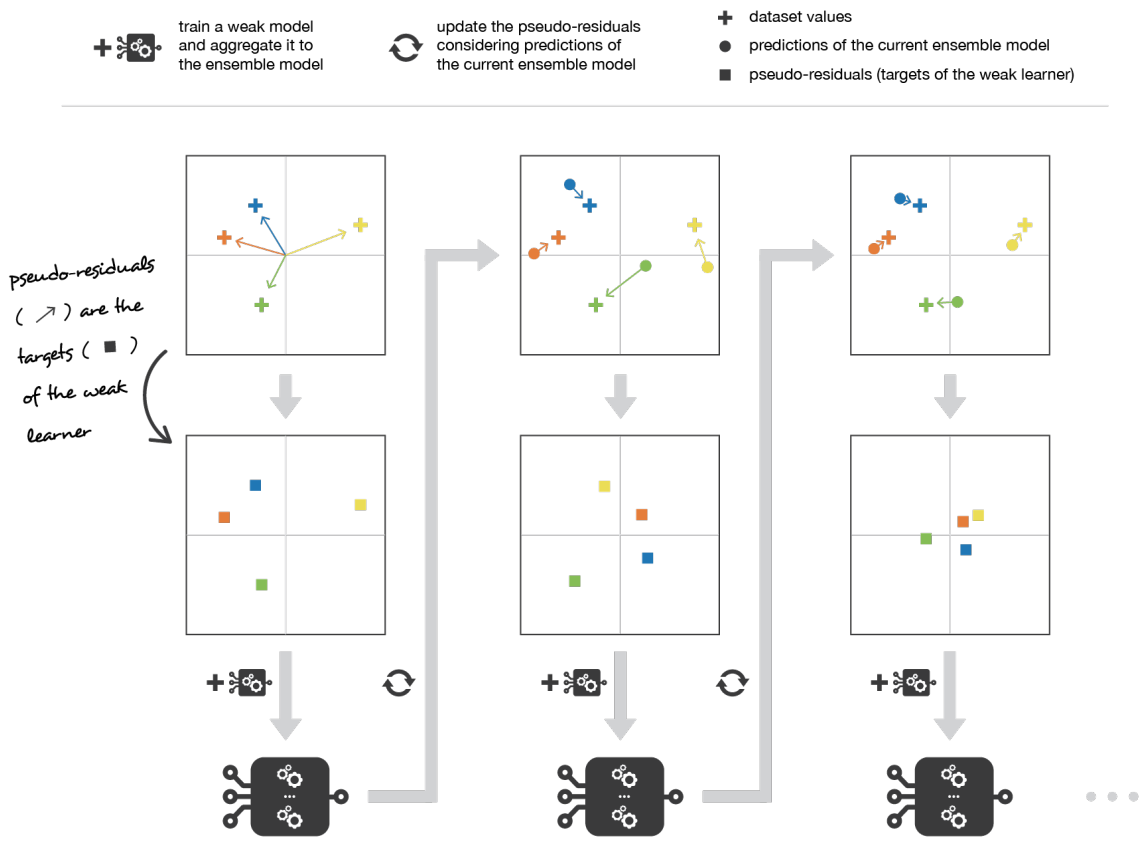


Fig. 26: Medium Science Blog

Gradient boosting updates values of the observations at each iteration. Weak learners are trained to fit the pseudo-residuals that indicate in which direction to correct the current ensemble model predictions to lower the error.

Examples

Here, we are trying an example of **Boosting** and compare it to a **Bagging**. Both of algorithms take the same weak learners to build the macro-model

- Adaboost Classifier

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import LabelEncoder
```

(continues on next page)

(continued from previous page)

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score

breast_cancer = load_breast_cancer()
x = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
y = pd.Categorical.from_codes(breast_cancer.target, breast_cancer.target_names)
# Transforming string Target to an int
encoder = LabelEncoder()
binary_encoded_y = pd.Series(encoder.fit_transform(y))

#Train Test Split
train_x, test_x, train_y, test_y = train_test_split(x, binary_encoded_y, random_
↪state=1)
clf_boosting = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1),
    n_estimators=200
)
clf_boosting.fit(train_x, train_y)
predictions = clf_boosting.predict(test_x)
print("For Boosting : F1 Score {}, Accuracy {}".format(round(f1_score(test_y,
↪predictions),2),round(accuracy_score(test_y,predictions),2)))
```

- Random Forest as a **bagging classifier**

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.ensemble import RandomForestClassifier

breast_cancer = load_breast_cancer()
x = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
y = pd.Categorical.from_codes(breast_cancer.target, breast_cancer.target_names)
# Transforming string Target to an int
encoder = LabelEncoder()
binary_encoded_y = pd.Series(encoder.fit_transform(y))

#Train Test Split
train_x, test_x, train_y, test_y = train_test_split(x, binary_encoded_y, random_
↪state=1)
clf_bagging = RandomForestClassifier(n_estimators=200, max_depth=1)
clf_bagging.fit(train_x, train_y)
predictions = clf_bagging.predict(test_x)
```

(continues on next page)

(continued from previous page)

```
print("For Bagging : F1 Score {}, Accuracy {}".format(round(f1_score(test_y,
↪predictions),2),round(accuracy_score(test_y,predictions),2)))
```

Comparaison

Metric	Bagging	Boosting
Accuracy	0.91	0.97
F1-Score	0.88	0.95

5.8.4 Overview of stacking

Stacking mainly **differ** from **bagging and boosting** on two points : - First stacking often considers **heterogeneous weak learners** (different learning algorithms are combined) whereas bagging and boosting consider mainly homogeneous weak learners. - Second, stacking learns to combine the base models using a **meta-model** whereas bagging and boosting combine weak learners following deterministic algorithms.

As we already mentioned, the idea of stacking is to learn several different weak learners and **combine them by training a meta-model** to output predictions based on the multiple predictions returned by these weak models. So, we need to define two things in order to build our stacking model: the L learners we want to fit and the meta-model that combines them.

For example, for a classification problem, we can choose as weak learners a KNN classifier, a logistic regression and a SVM, and decide to learn a neural network as meta-model. Then, the neural network will take as inputs the outputs of our three weak learners and will learn to return final predictions based on it.

So, assume that we want to fit a stacking ensemble composed of L weak learners. Then we have to follow the steps thereafter:

- split the **training data in two folds**
- choose **L weak learners** and **fit** them to data of the **first fold**
- for each of the L weak learners, **make predictions** for observations in the **second fold**
- fit the **meta-model** on the **second fold**, using **predictions made by the weak learners as inputs**

In the previous steps, we split the dataset in two folds because predictions on data that have been used for the training of the weak learners are **not relevant for the training of the meta-model**.

Stacking consists in training a meta-model to produce outputs based on the outputs returned by some lower layer weak learners.

A possible extension of stacking is multi-level stacking. It consists in doing **stacking with multiple layers**. As an example,

Multi-level stacking considers several layers of stacking: some meta-models are trained on outputs returned by lower layer meta-models and so on. Here we have represented a 3-layers stacking model.

Examples

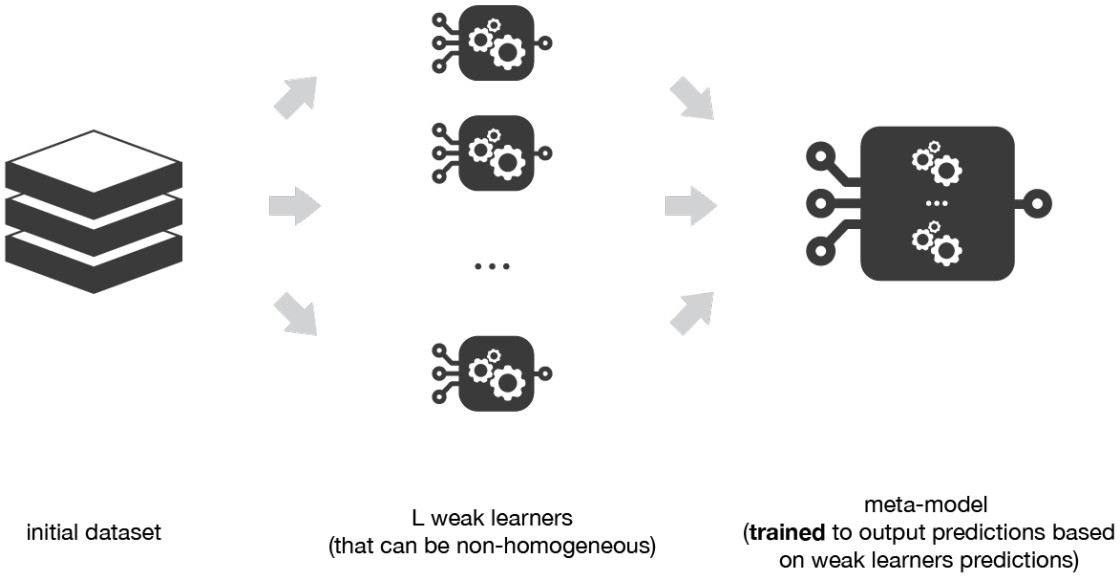


Fig. 27: Medium Science Blog

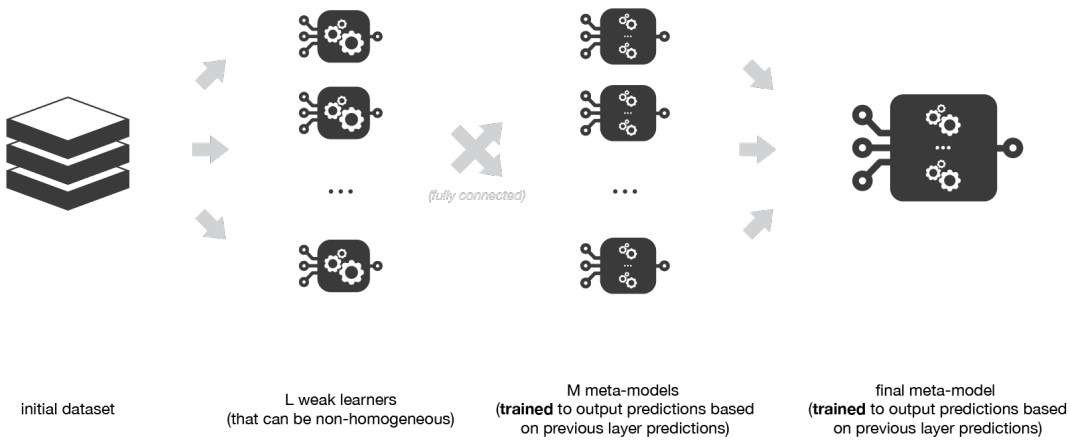


Fig. 28: Medium Science Blog

Here, we are trying an example of **Stacking** and compare it to a **Bagging** & a **Boosting**. We note that, many other applications (datasets) would show more difference between these techniques.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression

breast_cancer = load_breast_cancer()
x = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
y = pd.Categorical.from_codes(breast_cancer.target, breast_cancer.target_names)

# Transforming string Target to an int
encoder = LabelEncoder()
binary_encoded_y = pd.Series(encoder.fit_transform(y))

#Train Test Split
train_x, test_x, train_y, test_y = train_test_split(x, binary_encoded_y, random_
↪state=2020)

boosting_clf_ada_boost= AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1),
    n_estimators=3
)
bagging_clf_rf = RandomForestClassifier(n_estimators=200, max_depth=1,random_
↪state=2020)


clf_rf = RandomForestClassifier(n_estimators=200, max_depth=1,random_state=2020)
clf_ada_boost = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1,random_state=2020),
    n_estimators=3
)

clf_logistic_reg = LogisticRegression(solver='liblinear',random_state=2020)

#Customizing and Exception message
class NumberOfClassifierException(Exception):
    pass

#Creating a stacking class
class Stacking():
```

(continues on next page)

(continued from previous page)

```
    '''
        This is a test class for stacking !
        Please fill Free to change it to fit your needs
        We suppose that at least the First N-1 Classifiers have
        a predict_proba function.
    '''

    def __init__(self,classifiers):
        if(len(classifiers) < 2):
            raise numberOfClassifierException("You must fit your classifier with␣
↪2 classifiers at least");
        else:
            self._classifiers = classifiers


    def fit(self,data_x,data_y):

        stacked_data_x = data_x.copy()
        for classfier in self._classifiers[:-1]:
            classfier.fit(data_x,data_y)
            stacked_data_x = np.column_stack((stacked_data_x,classfier.predict_
↪proba(data_x)))


        last_classifier = self._classifiers[-1]
        last_classifier.fit(stacked_data_x,data_y)


    def predict(self,data_x):

        stacked_data_x = data_x.copy()
        for classfier in self._classifiers[:-1]:
            prob_predictions = classfier.predict_proba(data_x)
            stacked_data_x = np.column_stack((stacked_data_x,prob_predictions))

        last_classifier = self._classifiers[-1]
        return last_classifier.predict(stacked_data_x)



bagging_clf_rf.fit(train_x, train_y)
boosting_clf_ada_boost.fit(train_x, train_y)

classifers_list = [clf_rf,clf_ada_boost,clf_logistic_reg]
clf_stacking = Stacking(classifers_list)
clf_stacking.fit(train_x,train_y)

predictions_bagging = bagging_clf_rf.predict(test_x)
predictions_boosting = boosting_clf_ada_boost.predict(test_x)
```

(continues on next page)

(continued from previous page)

```
predictions_stacking = clf_stacking.predict(test_x)

print("For Bagging : F1 Score {}, Accuracy {}".format(round(f1_score(test_y,
↪predictions_bagging),2),round(accuracy_score(test_y,predictions_bagging),2)))
print("For Boosting : F1 Score {}, Accuracy {}".format(round(f1_score(test_y,
↪predictions_boosting),2),round(accuracy_score(test_y,predictions_boosting),2)))
print("For Stacking : F1 Score {}, Accuracy {}".format(round(f1_score(test_y,
↪predictions_stacking),2),round(accuracy_score(test_y,predictions_stacking),2)))
```

Comparaison

Metric	Bagging	Boosting	Stacking
Accuracy	0.90	0.94	0.98
F1-Score	0.88	0.93	0.98

5.9 Gradient descent

Gradient descent is an **optimization algorithm** used to **minimize some function** by **iteratively moving in the direction of steepest descent** as defined by the **negative of the gradient**. In machine learning, we use gradient descent to **update the parameters of our model**. **Parameters** refer to **coefficients** in **Linear Regression** and **weights** in **neural networks**.

This section aims to provide you an explanation of gradient descent and **intuitions** towards the **behaviour of different algorithms for optimizing it**. These explanations will help you put them to use.

We are first going to introduce the gradient descent, solve it for a regression problem and look at its different variants. Then, we will then briefly summarize challenges during training. Finally, we will introduce the **most common optimization algorithms** by showing their motivation to resolve these challenges and list some advices for facilitate the algorithm choice.

5.9.1 Introduction

Consider the 3-dimensional graph below in **the context of a cost function**. Our **goal** is to **move from the mountain in the top right corner (high cost) to the dark blue sea in the bottom left (low cost)**. The **arrows** represent the **direction** of steepest descent (negative gradient) from any given point–the direction that **decreases the cost function** as quickly as possible

Gradient descent intuition.

Starting at the top of the mountain, we take our first step **downhill** in the **direction specified by the negative gradient**. Next we **recalculate the negative gradient** (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this **process iteratively until** we get to the **bottom of our graph**, or to a **point where we can no longer move downhill–a local minimum**.

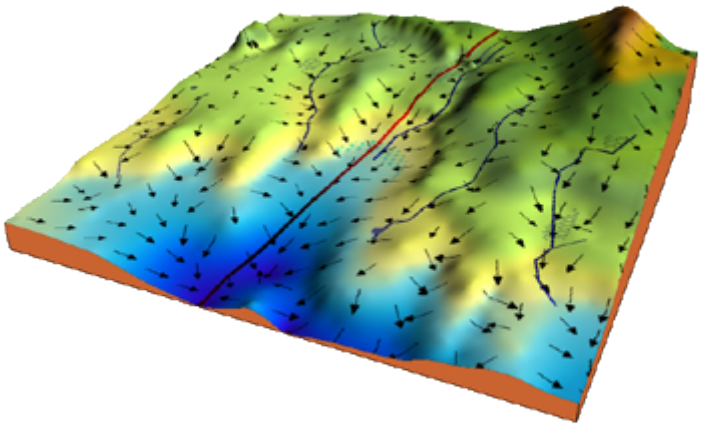


Fig. 29: adalta.it

Learning rate

The **size of these steps** is called the **learning rate**. With a **high learning rate** we can cover more ground each step, but we **risk overshooting the lowest point** since the slope of the hill is constantly changing. With a **very low learning rate**, we can **confidently move in the direction of the negative gradient** since we are **recalculating it so frequently**. A **low learning rate is more precise**, but calculating the gradient is **time-consuming**, so it will take us a very long time to get to the bottom.

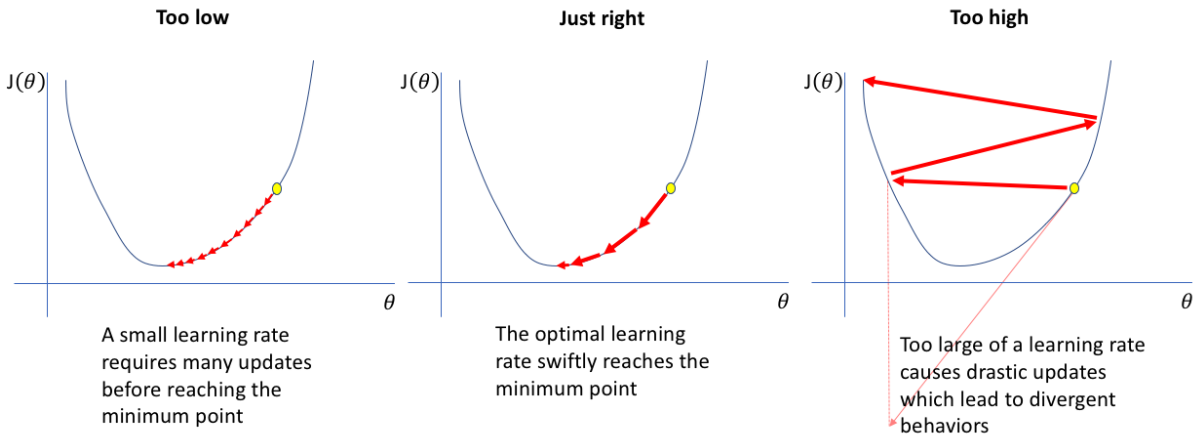


Fig. 30: jeremyjordan

impacts of learning rate choice.

Cost function

A **Loss Function (Error function)** tells us **"how good"** our **model** is at making predictions for a **given set of parameters**. The cost function has its own curve and its own gradients. The **slope of this curve** tells us how to **update our parameters** to make the model more **accurate**.

5.9.2 Numerical solution for gradient descent

Let's run gradient descent using a **linear regression cost function**.

There are **two parameters** in our cost function we can control: - $ .. raw:: latex

beta`_0$: **(the bias)** - $:raw-latex:*beta*_1 $: **(weight or coefficient)**

Since we need to consider the **impact each one** has on the final prediction, we need to use **partial derivatives**. We calculate the **partial derivatives of the cost function with respect to each parameter and store the results in a gradient**.

Given the cost function

$$f(\beta_0, \beta_1) = \frac{1}{2}\frac{\partial MSE}{\partial \beta} = \frac{1}{2N}\sum_{i=1}^{n}(y_i - (\beta_1 x_i + \beta_0))^2 = \frac{1}{2N}\sum_{i=1}^{n}((\beta_1 x_i + \beta_0) - y_i)^2$$

The gradient can be calculated as

$$f'(\beta_0, \beta_1) = \begin{bmatrix} \frac{\partial f}{\partial \beta_0} \\ \frac{\partial f}{\partial \beta_1} \end{bmatrix} = \begin{bmatrix} \frac{1}{2N}\sum -2((\beta_1 x_i + \beta_0) - y_i) \\ \frac{1}{2N}\sum -2x_i((\beta_1 x_i + \beta_0) - y_i) \end{bmatrix} = \begin{bmatrix} \frac{-1}{N}\sum((\beta_1 x_i + \beta_0) - y_i) \\ \frac{-1}{N}\sum x_i((\beta_1 x_i + \beta_0) - y_i) \end{bmatrix}$$

To solve for the gradient, we **iterate** through our **data points** using our **:math:beta_1 and :math:beta_0 values** and compute the

partial derivatives. This **new gradient** tells us the **slope of our cost function at our current position** (current parameter values) and the **direction we should move to update our parameters**. The **size of our update** is **controlled by the learning rate**.

Pseudocode of this algorithm

```
Function  gradient_descent(X, Y, learning_rate, number_iterations):

    m : 1
    b : 1
    m_deriv : 0
    b_deriv : 0
    data_length : length(X)
    loop i : 1  --> number_iterations:
        loop i : 1  ->  data_length  :
            m_deriv : m_deriv -X[i] * ((m*X[i] + b) - Y[i])
            b_deriv : b_deriv - ((m*X[i] + b) - Y[i])
        m : m - (m_deriv / data_length) * learning_rate
        b : b - (b_deriv / data_length) * learning_rate

    return m, b
```

5.9.3 Gradient descent variants

There are **three variants of gradient descent**, which **differ in how much data we use to compute the gradient** of the objective function. Depending on the **amount of data**, we make a **trade-off between the accuracy of the parameter update and the time it takes to perform an update**.

Batch gradient descent

Batch gradient descent, known also as Vanilla gradient descent, computes the gradient of the cost function with respect to the parameters θ for the **entire training dataset** :

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

As we need to calculate the gradients for the **whole dataset** to **perform just one update**, **batch** gradient descent can be **very slow** and **is intractable for datasets that don't fit in memory**. Batch gradient descent also **doesn't allow us to update our model online**.

Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$

- Choose an initial vector of parameters w and learning rate η.
- Repeat until an approximate minimum is obtained:
  - Randomly shuffle examples in the training set.
  - For $i \in 1, \ldots, n$
    - $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$

Batch gradient descent performs **redundant computations for large datasets**, as it **recomputes** gradients for **similar examples** before each **parameter update**. SGD does away with **this redundancy by performing one update at a time**. It is therefore usually **much faster and can also be used to learn online**. **SGD** performs **frequent updates with a high variance that cause the objective function to fluctuate heavily as in the image below**.

SGD fluctuation.

While batch gradient descent **converges to the minimum of the basin** the parameters are placed in, SGD's fluctuation, on the one hand, **enables it to jump to new and potentially better local minima**. On the other hand, **this ultimately complicates convergence to the exact minimum**, as **SGD will keep overshooting**. However, it **has been shown** that when we **slowly decrease the learning rate**, SGD shows the same **convergence behaviour as batch gradient descent**, almost certainly **converging to a local or the global minimum for non-convex and convex optimization respectively**.

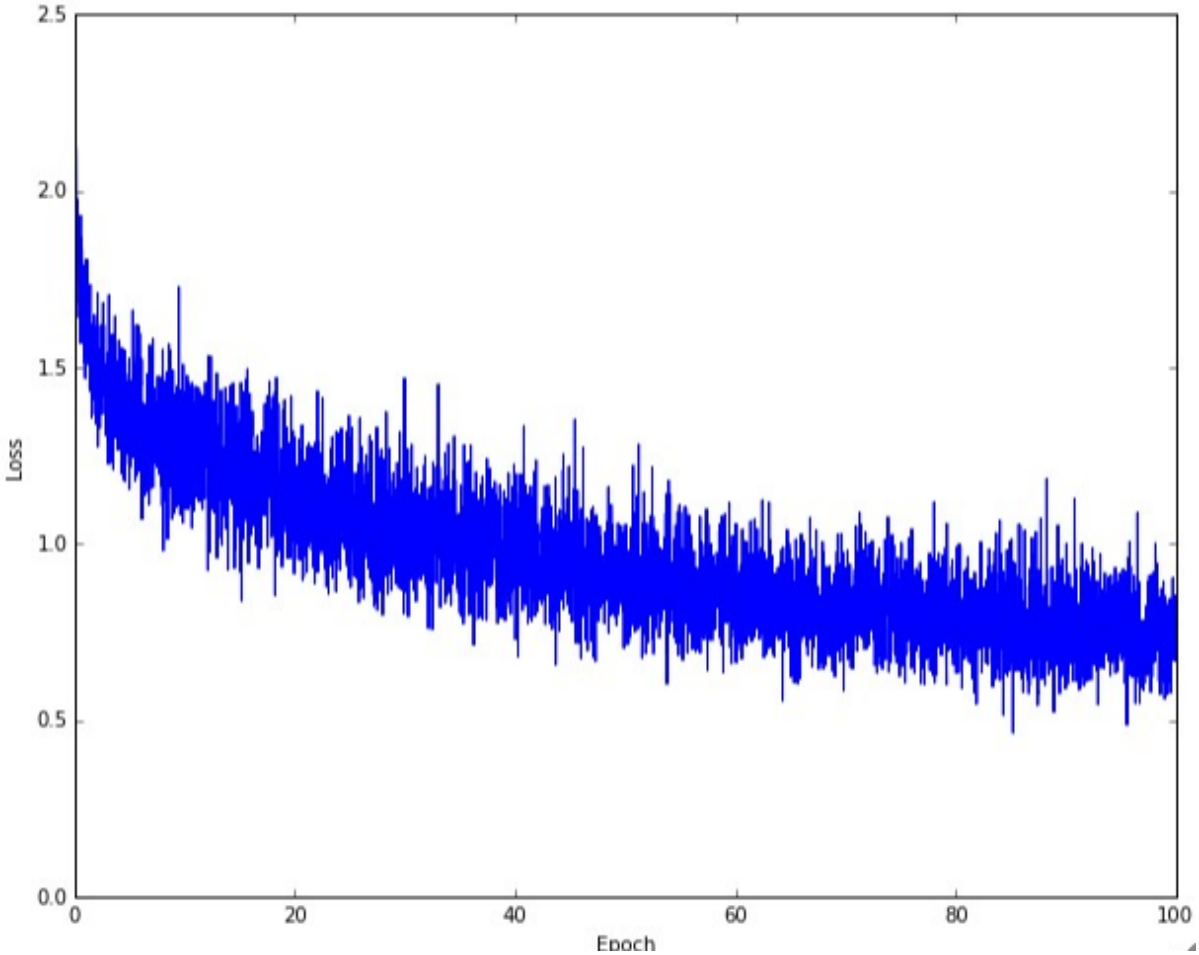


Fig. 31: Wikipedia

Mini-batch gradient descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

This way, it :

- **reduces the variance of the parameter updates**, which can lead to **more stable convergence**.
- can make use of **highly optimized matrix optimizations** common to state-of-the-art deep learning libraries that make computing the gradient very efficient. **Common mini-batch sizes range between 50 and 256**, but can vary for different applications.

Mini-batch gradient descent is **typically the algorithm of choice when training a neural network**.

5.9.4 Gradient Descent challenges

Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

- Choosing a proper **learning rate** can be difficult. A learning rate that is **too small** leads to **painfully slow convergence**, while a learning rate that is **too large** can hinder convergence and cause the loss function to **fluctuate** around the minimum or even to **diverge**.
- **Learning rate schedules** try to **adjust the learning rate during training** by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics.
- Additionally, the same learning rate applies to all parameter updates. **If our data is sparse and our features have very different frequencies**, we might not want to update all of them to the same extent, but perform **a larger update for rarely occurring features**.
- Another key challenge of **minimizing highly non-convex error** functions common for neural networks is **avoiding** getting **trapped in their numerous suboptimal local minima**. These **saddle points (local minimas)** are usually surrounded by a plateau of the same error, which makes it **notoriously hard for SGD to escape**, as the gradient is close to zero in all dimensions.

5.9.5 Gradient descent optimization algorithms

In the following, we will outline some **algorithms** that are **widely** used by the **deep learning community** to deal with the aforementioned **challenges**.

Momentum

SGD has trouble **navigating ravines (areas where the surface curves much more steeply in one dimension than in another)**, which are common **around local optima**. In these scenarios, **SGD oscillates across the slopes of the ravine while only making hesitant progress** along the bottom towards the local optimum as in the image below.

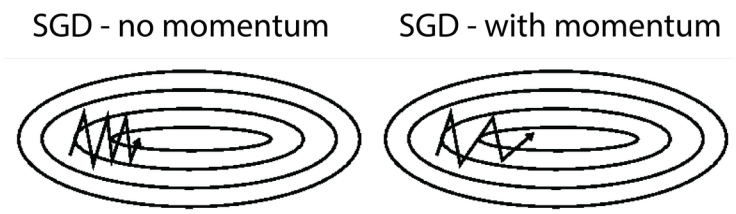


Fig. 32: Wikipedia

SGD and momentum.

Source

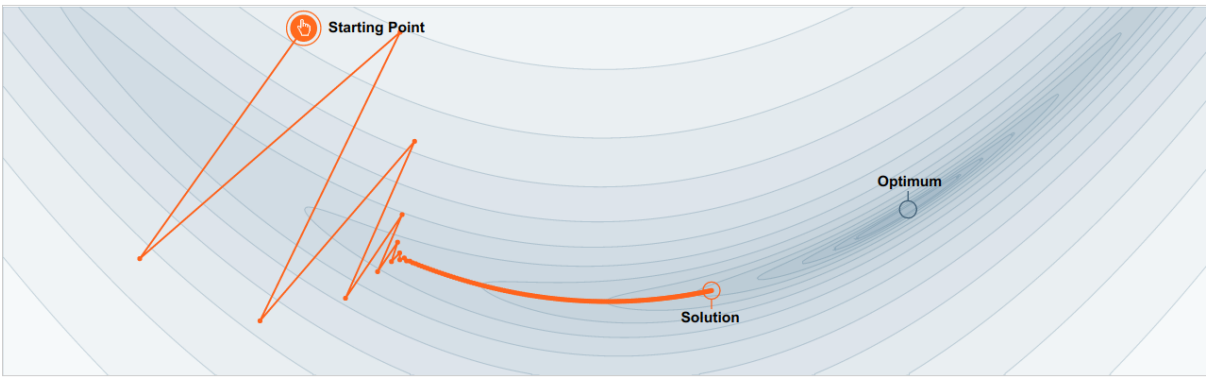


Fig. 33: No momentum: oscillations toward local largest gradient

No momentum: moving toward local largest gradient create oscillations.

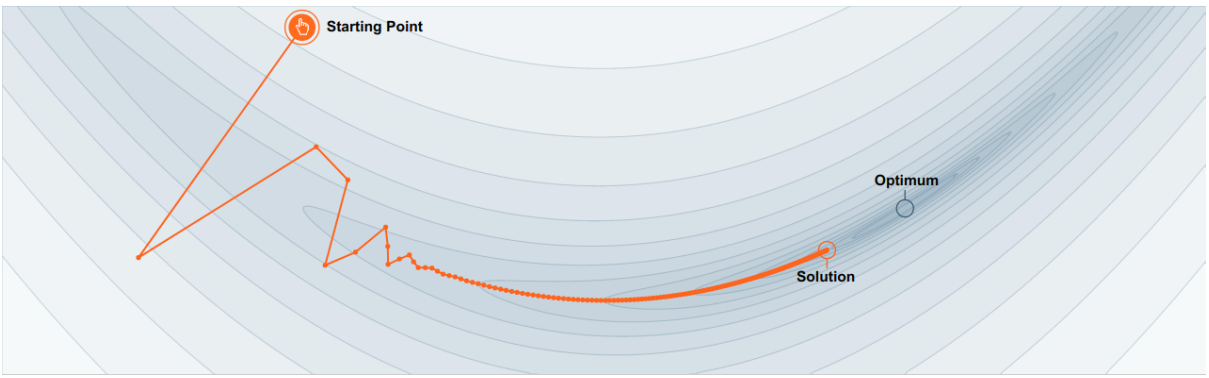


Fig. 34: With momentum: accumulate velocity to avoid oscillations

With momentum: accumulate velocity to avoid oscillations.

Momentum is a method that helps **accelerate SGD** in the **relevant direction and dampens oscillations** as can be seen in image above. It does this by **adding a fraction :math:`gamma`**

of the update vector of the **past time step to the current update vector**

$$\begin{aligned} v_t &= \rho v_{t-1} + \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{5.50}$$

```
vx = 0
while True:
    dx = gradient(J, x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

Note: The momentum term **:math:`rho`** is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push **a ball down a hill**. The **ball accumulates momentum as it rolls downhill**, becoming faster and faster on the way **(until it reaches its terminal velocity if there is air resistance, i.e. :math:`rho` <1)**.

The same thing happens to our parameter updates: **The momentum term increases** for **dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions**. As a result, we **gain faster convergence** and **reduced oscillation**.

AdaGrad: adaptive learning rates

- Added element-wise scaling of the gradient based on the historical sum of squares in each dimension.
- "Per-parameter learning rates" or "adaptive learning rates"

```
grad_squared = 0
while True:
    dx = gradient(J, x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

- Progress along "steep" directions is damped.
- Progress along "flat" directions is accelerated.
- Problem: step size over long time => Decays to zero.

RMSProp: "Leaky AdaGrad"

```
grad_squared = 0
while True:
    dx = gradient(J, x)
    grad_squared += decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

- `decay_rate = 1`: gradient descent
- `decay_rate = 0`: AdaGrad

Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly **unsatisfactory**. We'd like to have a smarter ball, a ball that has **a notion of where it is going** so that it **knows to slow down before the hill slopes up again**. Nesterov accelerated gradient (NAG) is a way to give **our momentum term this kind of prescience**. We know that we will use our momentum term γv_{t-1} to move the parameters θ.

Computing $\theta - \gamma v_{t-1}$ thus gives us **an approximation of the next position of the parameters** (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\ \theta &= \theta - v_t \end{aligned} \tag{5.51}$$

Again, we set the momentum term γ to a value of around 0.9. While **Momentum first computes the current gradient and then takes a big jump in the direction of the updated accumulated gradient** , **NAG** first **makes a big jump in the direction of the previous accumulated gradient, measures the gradient and then makes a correction, which results in the complete NAG update**. This anticipatory update **prevents** us from **going too fast** and results in **increased responsiveness**, which has significantly **increased the performance of RNNs** on a number of tasks

Adam

Adaptive Moment Estimation (Adam) is a method that computes **adaptive learning rates** for each parameter. In addition to storing an **exponentially decaying average of past squared gradients :math:`v_t`**, Adam also keeps an **exponentially decaying average of past gradients :math:`m_t`, similar to momentum**. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a **heavy ball with friction**, which thus prefers **flat minima in the error surface**. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta J(\theta) \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \nabla_\theta J(\theta)^2 \end{aligned} \tag{5.52}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = gradient(J, x)
    # Momentum:
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    # AdaGrad/RMSProp
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1). They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{5.53}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{5.54}$$

They then use these to update the parameters (Adam update rule):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- $\hat{m}_t$ Accumulate gradient: velocity.
- $\hat{v}_t$ Element-wise scaling of the gradient based on the historical sum of squares in each dimension.
- Choose Adam as default optimizer
- Default values of 0.9 for β_1, 0.999 for β_2, and 10^{-7} for ϵ.
- learning rate in a range between $1e-3$ and $5e-4$

5.10 Lab: Faces recognition using various learning models

This lab is inspired by a scikit-learn lab: Faces recognition example using eigenfaces and SVMs

It uses scikit-learan and pytorch models using skorch (slides).

- skorch provides scikit-learn compatible neural network library that wraps PyTorch.
- skorch abstracts away the training loop, making a lot of boilerplate code obsolete. A simple *net.fit(X, y)* is enough.

Note that more sofisticated models can be used, see for a overview.

Models:

- Eigenfaces unsupervized exploratory analysis.
- LogisticRegression with L2 regularization (includes model selection with 5CV`_
- SVM-RBF (includes model selection with 5CV.
- MLP using sklearn using sklearn (includes model selection with 5CV)
- MLP using skorch classifier
- Basic Convnet (ResNet18) using skorch.
- Pretrained ResNet18 using skorch.

Pipelines:

- Univariate feature filtering (Anova) with Logistic-L2
- PCA with LogisticRegression with L2 regularization

```
import numpy as np
from time import time
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

# Preprocesing
from sklearn import preprocessing
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import SelectKBest, f_classif

# Dataset
from sklearn.datasets import fetch_lfw_people

# Models
from sklearn.decomposition import PCA
import sklearn.manifold as manifold
import sklearn.linear_model as lm
import sklearn.svm as svm
from sklearn.neural_network import MLPClassifier
# from sklearn.ensemble import RandomForestClassifier
# from sklearn.ensemble import GradientBoostingClassifier

# Pytorch Models
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from skorch import NeuralNetClassifier
import skorch

# Use [skorch](https://github.com/skorch-dev/skorch). Install:
# `conda install -c conda-forge skorch`

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

5.10.1 Utils

```
def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    """Plot a gallery of portraits."""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(min(images.shape[0], n_row * n_col)):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())


def title(y_pred, y_test, target_names, i):
    """Plot the result of the prediction on a portion of the test set."""
    pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1]
    true_name = target_names[y_test[i]].rsplit(' ', 1)[-1]
    return 'predicted: %s\ntrue:      %s' % (pred_name, true_name)


def label_proportion(x, decimals=2):
    """Labels's proportions."""
    unique, counts = np.unique(x, return_counts=True)
    return dict(zip(unique, np.round(counts / len(x), decimals)))
```

5.10.2 Download the data

```
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# introspect the images arrays to find the shapes (for plotting)
n_samples, h, w = lfw_people.images.shape

# for machine learning we use the 2 data directly (as relative pixel
# positions info is ignored by this model)
X = lfw_people.data
n_features = X.shape[1]

# the label to predict is the id of the person
y = lfw_people.target
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

print("Total dataset size:")
print("n_samples: %d" % n_samples)
print("n_features: %d" % n_features)
print("n_classes: %d" % n_classes)
```

Out:

```
Total dataset size:
n_samples: 1288
n_features: 1850
n_classes: 7
```

5.10.3 Split into a training and testing set in stratified way

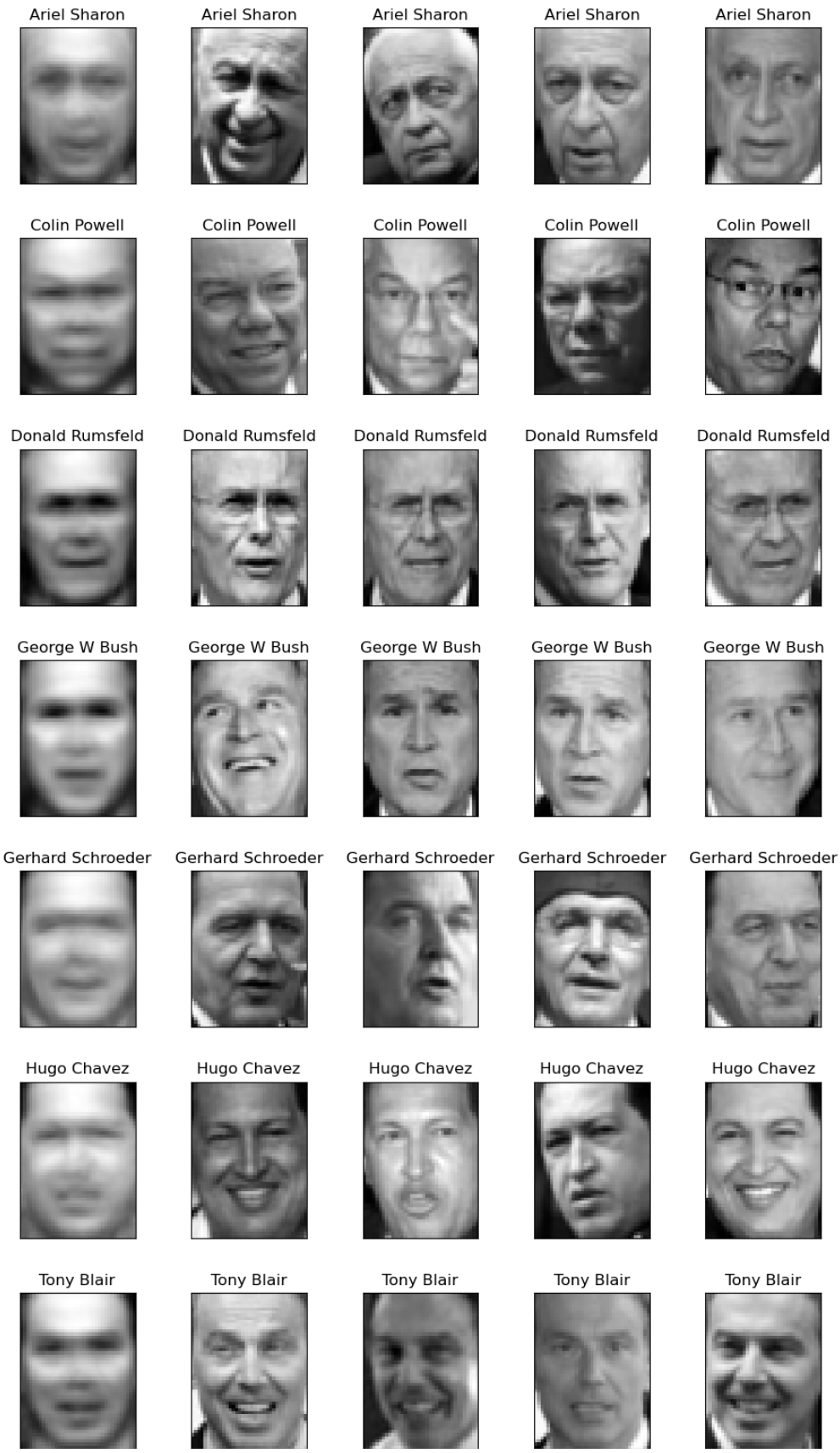
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=1, stratify=y)

print({target_names[lab]: prop for lab, prop in
       label_proportion(y_train).items()})
```

Out:

```
{'Ariel Sharon': 0.06, 'Colin Powell': 0.18, 'Donald Rumsfeld': 0.09, 'George W␣
↪Bush': 0.41, 'Gerhard Schroeder': 0.08, 'Hugo Chavez': 0.05, 'Tony Blair': 0.11}
```

Plot mean faces and 4 samples of each individual

```
single_faces = [X_train[y_train == lab][:5] for lab in np.unique(y_train)]
single_faces = np.vstack(single_faces).reshape((5 * n_classes, h, w))

mean_faces = [X_train[y_train == lab].mean(axis=0) for lab in
              np.unique(y_train)]
mean_faces = np.vstack(mean_faces).reshape((n_classes, h, w))

single_faces[::5, :, :] = mean_faces
titles = [n for name in target_names for n in [name] * 5]
plot_gallery(single_faces, titles, h, w, n_row=n_classes, n_col=5)
```


Ariel Sharon
Ariel Sharon
Ariel Sharon
Ariel Sharon
Ariel Sharon
Colin Powell
Colin Powell
Colin Powell
Colin Powell
Colin Powell
Donald Rumsfeld
Donald Rumsfeld
Donald Rumsfeld
Donald Rumsfeld
Donald Rumsfeld
George W Bush
George W Bush
George W Bush
George W Bush
George W Bush
Gerhard Schroeder
Gerhard Schroeder
Gerhard Schroeder
Gerhard Schroeder
Gerhard Schroeder
Hugo Chavez
Hugo Chavez
Hugo Chavez
Hugo Chavez
Hugo Chavez
Tony Blair
Tony Blair
Tony Blair
Tony Blair
Tony Blair

5.10.4 Eigenfaces

Compute a PCA (eigenfaces) on the face dataset (treated as unlabeled dataset): unsupervised feature extraction / dimensionality reduction

```
n_components = 150

print("Extracting the top %d eigenfaces from %d faces"
      % (n_components, X_train.shape[0]))
t0 = time()
pca = PCA(n_components=n_components, svd_solver='randomized',
          whiten=True).fit(X_train)
print("done in %0.3fs" % (time() - t0))

eigenfaces = pca.components_.reshape((n_components, h, w))

print("Explained variance", pca.explained_variance_ratio_[:2])
```

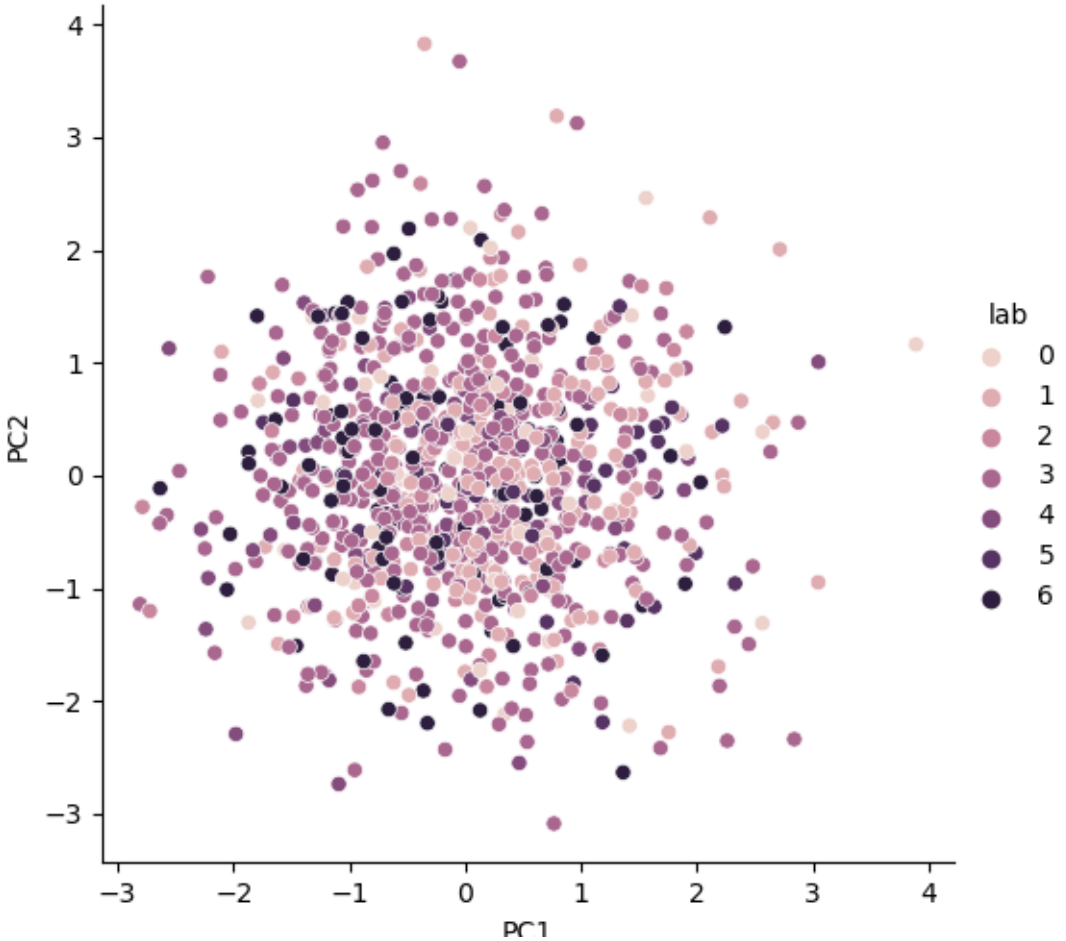
Out:

```
Extracting the top 150 eigenfaces from 966 faces
done in 0.114s
Explained variance [0.19 0.15]
```

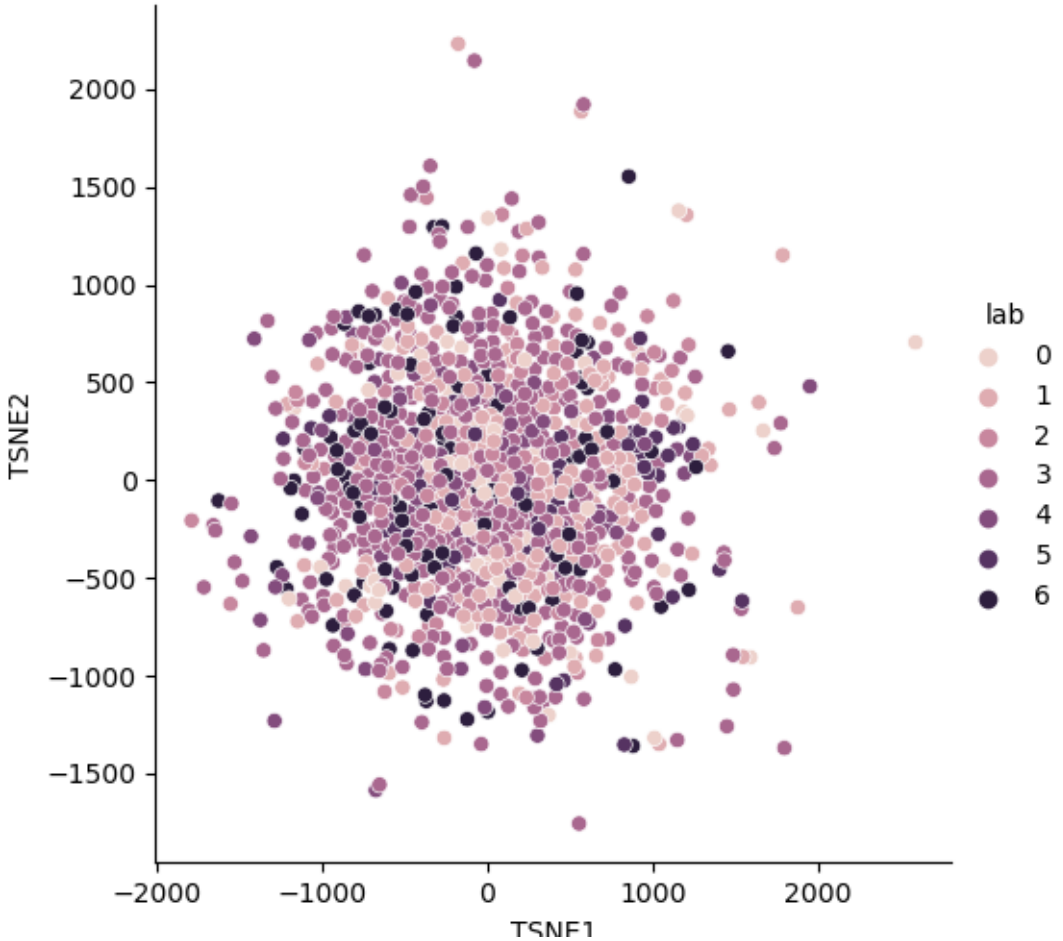
T-SNE

```
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
X_tsne = tsne.fit_transform(X_train)
```

```
print("Projecting the input data on the eigenfaces orthonormal basis")
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
df = pd.DataFrame(dict(lab=y_train,
                       PC1=X_train_pca[:, 0],
                       PC2=X_train_pca[:, 1],
                       TSNE1=X_tsne[:, 0],
                       TSNE2=X_tsne[:, 1]))

sns.relplot(x="PC1", y="PC2", hue="lab", data=df)

sns.relplot(x="TSNE1", y="TSNE2", hue="lab", data=df)
```


PC2
PC1
lab
0
1
2
3
4
5
6

•

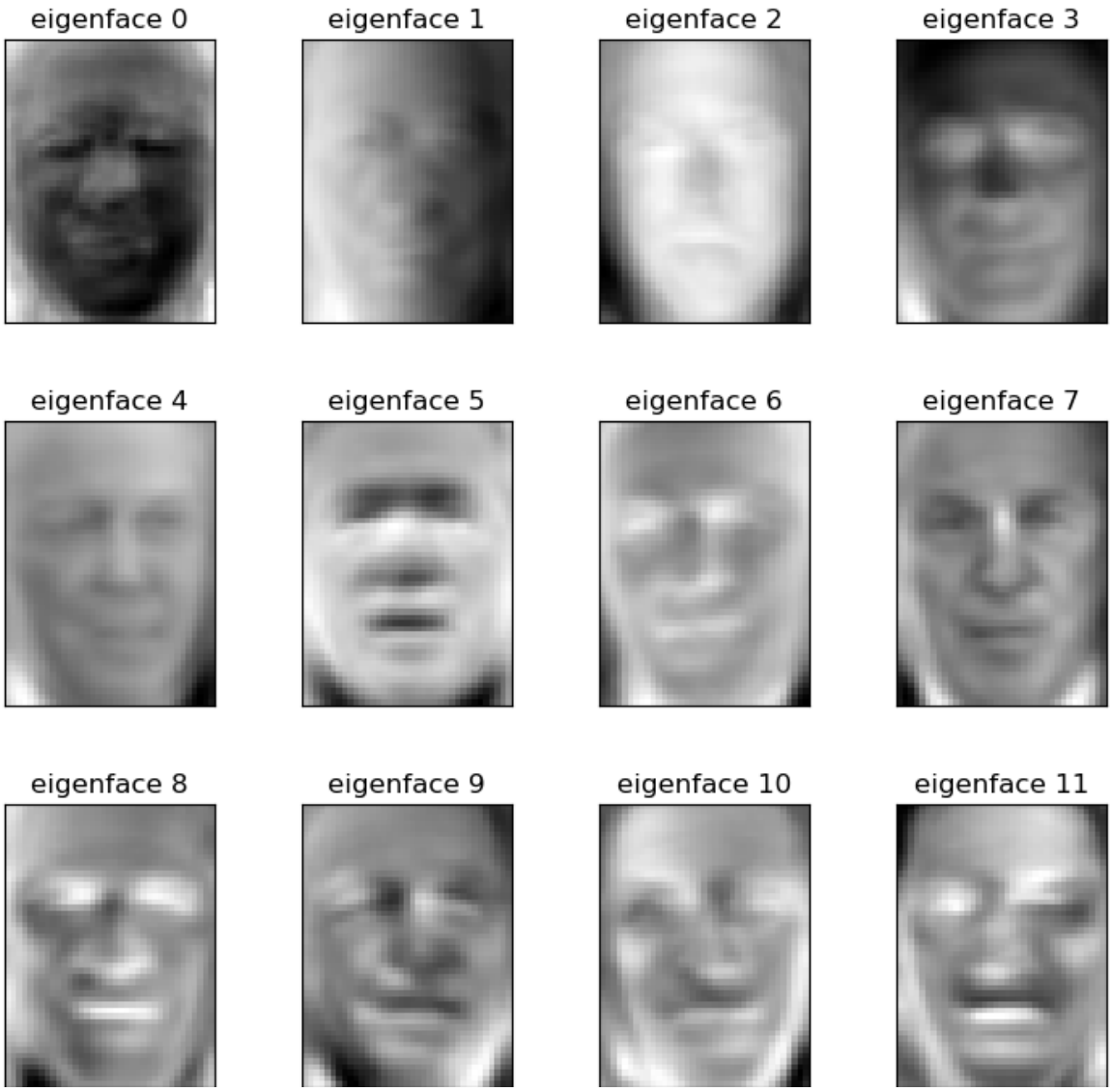


Out:

```
Projecting the input data on the eigenfaces orthonormal basis

<seaborn.axisgrid.FacetGrid object at 0x7fe42eba3250>
```

Plot eigenfaces:

```
eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)
```



5.10.5 LogisticRegression with L2 penalty (with CV-based model selection)

Our goal is to obtain a good balanced accuracy, ie, the macro average (*macro avg*) of classes' reccalls. In this perspective, the good practices are:

- Scale input features using either *StandardScaler()* or *MinMaxScaler()* "It doesn't harm".
- Re-balance classes' contributions *class_weight='balanced'*
- Do not include an intercept (*fit_intercept=False*) in the model. This should reduce the global accuracy *weighted avg*. But rememember that we decided to maximize the balanced accuracy.

```
lrl2_cv = make_pipeline(
    preprocessing.StandardScaler(),
    # preprocessing.MinMaxScaler(),  # Would have done the job either
    GridSearchCV(lm.LogisticRegression(max_iter=1000, class_weight='balanced',
```

(continues on next page)

(continued from previous page)

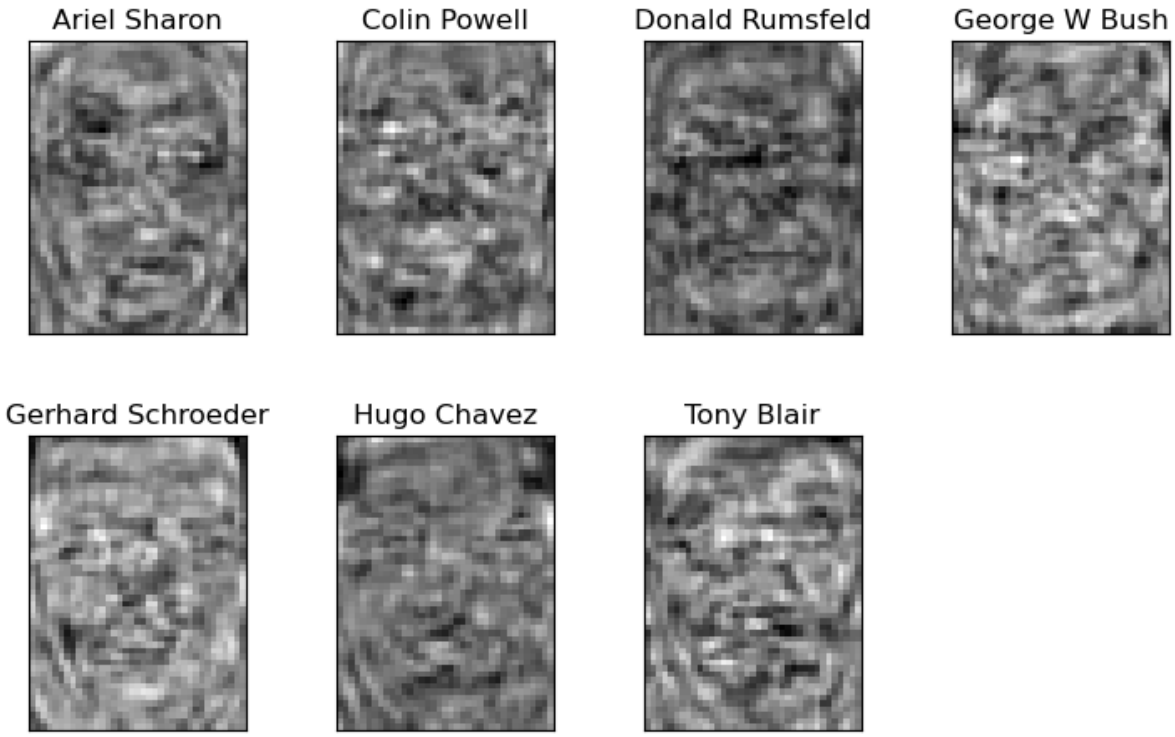
```
                                        fit_intercept=False),
                 {'C': 10. ** np.arange(-3, 3)},
                 cv=5, n_jobs=5))

t0 = time()
lrl2_cv.fit(X=X_train, y=y_train)
print("done in %0.3fs" % (time() - t0))
print("Best params found by grid search:")
print(lrl2_cv.steps[-1][1].best_params_)

y_pred = lrl2_cv.predict(X_test)
print(classification_report(y_test, y_pred, target_names=target_names))
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))
```

Out:

```
done in 5.383s
Best params found by grid search:
{'C': 1.0}
                    precision    recall  f1-score   support

     Ariel Sharon       0.59      0.89      0.71        19
     Colin Powell       0.91      0.83      0.87        59
  Donald Rumsfeld       0.71      0.80      0.75        30
    George W Bush       0.91      0.80      0.86       133
Gerhard Schroeder       0.70      0.78      0.74        27
      Hugo Chavez       0.62      0.56      0.59        18
       Tony Blair       0.71      0.83      0.77        36

         accuracy                           0.80       322
        macro avg       0.74      0.79      0.75       322
     weighted avg       0.82      0.80      0.80       322

[[ 17   0   1   0   0   1   0]
 [  2  49   3   3   0   0   2]
 [  3   0  24   1   0   1   1]
 [  7   3   4 107   5   3   4]
 [  0   0   1   0  21   1   4]
 [  0   2   0   3   2  10   1]
 [  0   0   1   3   2   0  30]]
```

Coeficients

```
coefs = lrl2_cv.steps[-1][1].best_estimator_.coef_
coefs = coefs.reshape(-1, h, w)
plot_gallery(coefs, target_names, h, w)
```



5.10.6 SVM (with CV-based model selection)

Remarks: - RBF generally requires "large" C (>1) - Poly generally requires "small" C (<1)

```
svm_cv = make_pipeline(
    # preprocessing.StandardScaler(),
    preprocessing.MinMaxScaler(),
    GridSearchCV(svm.SVC(class_weight='balanced'),
                 {'kernel': ['poly', 'rbf'], 'C': 10. ** np.arange(-2, 3)},
                 # {'kernel': ['rbf'], 'C': 10. ** np.arange(-1, 4)},
                 cv=5, n_jobs=5))

t0 = time()
svm_cv.fit(X_train, y_train)
print("done in %0.3fs" % (time() - t0))
print("Best params found by grid search:")
```

(continues on next page)

(continued from previous page)

```
print(svm_cv.steps[-1][1].best_params_)

y_pred = svm_cv.predict(X_test)
print(classification_report(y_test, y_pred, target_names=target_names))
```

Out:

```
done in 23.788s
Best params found by grid search:
{'C': 0.1, 'kernel': 'poly'}
                   precision    recall  f1-score   support

     Ariel Sharon       0.71      0.89      0.79        19
     Colin Powell       0.84      0.88      0.86        59
  Donald Rumsfeld       0.76      0.87      0.81        30
    George W Bush       0.90      0.89      0.90       133
Gerhard Schroeder       0.77      0.74      0.75        27
      Hugo Chavez       0.90      0.50      0.64        18
       Tony Blair       0.82      0.78      0.80        36

         accuracy                           0.84       322
        macro avg       0.82      0.79      0.79       322
     weighted avg       0.85      0.84      0.84       322
```

5.10.7 MLP with sklearn and CV-based model selection

Default parameters: - alpha, default=0.0001 L2 penalty (regularization term) parameter. - batch_size=min(200, n_samples) - learning_rate_init = 0.001 (the important one since we uses adam) - solver default='adam'

- sgd: momentum=0.9
- adam: beta_1, beta_2 default=0.9, 0.999 Exponential decay rates for the first and second moment.
- L2 penalty (regularization term) parameter, *alpha* default=0.0001
- tol, default=1e-4

```
mlp_param_grid = {"hidden_layer_sizes":
                  # Configurations with 1 hidden layer:
                  [(100, ), (50, ), (25, ), (10, ), (5, ),
                   # Configurations with 2 hidden layers:
                   (100, 50, ), (50, 25, ), (25, 10, ), (10, 5, ),
                   # Configurations with 3 hidden layers:
                   (100, 50, 25, ), (50, 25, 10, ), (25, 10, 5, )],
                  "activation": ["relu"], "solver": ["adam"], 'alpha': [0.0001]}

mlp_cv = make_pipeline(
    # preprocessing.StandardScaler(),
```

(continues on next page)

(continued from previous page)

```
    preprocessing.MinMaxScaler(),
    GridSearchCV(estimator=MLPClassifier(random_state=1, max_iter=400),
                 param_grid=mlp_param_grid,
                 cv=5, n_jobs=5))

t0 = time()
mlp_cv.fit(X_train, y_train)
print("done in %0.3fs" % (time() - t0))
print("Best params found by grid search:")
print(mlp_cv.steps[-1][1].best_params_)

y_pred = mlp_cv.predict(X_test)
print(classification_report(y_test, y_pred, target_names=target_names))
```

Out:

```
done in 128.638s
Best params found by grid search:
{'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100,), 'solver':
↪'adam'}
                   precision    recall  f1-score   support

     Ariel Sharon       0.76      0.84      0.80        19
     Colin Powell       0.88      0.85      0.86        59
  Donald Rumsfeld       0.68      0.77      0.72        30
    George W Bush       0.89      0.91      0.90       133
Gerhard Schroeder       0.77      0.74      0.75        27
      Hugo Chavez       1.00      0.39      0.56        18
       Tony Blair       0.73      0.83      0.78        36

         accuracy                           0.83       322
        macro avg       0.82      0.76      0.77       322
     weighted avg       0.84      0.83      0.83       322
```

5.10.8 MLP with pytorch and no model selection

```
class SimpleMLPClassifierPytorch(nn.Module):
    """Simple (one hidden layer) MLP Classifier with Pytorch."""

    def __init__(self):
        super(SimpleMLPClassifierPytorch, self).__init__()

        self.dense0 = nn.Linear(1850, 100)
        self.nonlin = nn.ReLU()
        self.output = nn.Linear(100, 7)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, X, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        X = self.nonlin(self.dense0(X))
        X = self.softmax(self.output(X))
        return X


mlp = NeuralNetClassifier(  # Match the parameters with sklearn
    SimpleMLPClassifierPytorch,
    criterion=torch.nn.NLLLoss,
    max_epochs=100,
    batch_size=200,
    optimizer=torch.optim.Adam,
    # optimizer=torch.optim.SGD,
    optimizer__lr=0.001,
    optimizer__betas=(0.9, 0.999),
    optimizer__eps=1e-4,
    optimizer__weight_decay=0.0001,  # L2 regularization
    # Shuffle training data on each epoch
    iterator_train__shuffle=True,
    device=device,
    verbose=0)

scaler = preprocessing.MinMaxScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

t0 = time()
mlp.fit(X_train_s, y_train)
print("done in %0.3fs" % (time() - t0))

y_pred = mlp.predict(X_test_s)
print(classification_report(y_test, y_pred, target_names=target_names))
```

Out:

```
done in 3.142s
                   precision    recall  f1-score   support

     Ariel Sharon       0.70      0.84      0.76        19
     Colin Powell       0.92      0.75      0.82        59
  Donald Rumsfeld       0.62      0.70      0.66        30
    George W Bush       0.81      0.92      0.86       133
Gerhard Schroeder       0.71      0.56      0.63        27
      Hugo Chavez       0.88      0.39      0.54        18
       Tony Blair       0.78      0.78      0.78        36

         accuracy                           0.79       322
        macro avg       0.77      0.70      0.72       322
     weighted avg       0.80      0.79      0.78       322
```

5.10.9 Univariate feature filtering (Anova) with Logistic-L2

```
anova_l2lr = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('anova', SelectKBest(f_classif)),
    ('l2lr', lm.LogisticRegression(max_iter=1000, class_weight='balanced',
                                   fit_intercept=False))
])

param_grid = {'anova__k': [50, 100, 500, 1000, 1500, X_train.shape[1]],
              'l2lr__C': 10. ** np.arange(-3, 3)}
anova_l2lr_cv = GridSearchCV(anova_l2lr, cv=5,  param_grid=param_grid,
                             n_jobs=5)

t0 = time()
anova_l2lr_cv.fit(X=X_train, y=y_train)
print("done in %0.3fs" % (time() - t0))

print("Best params found by grid search:")
print(anova_l2lr_cv.best_params_)

y_pred = anova_l2lr_cv.predict(X_test)
print(classification_report(y_test, y_pred, target_names=target_names))
```

Out:

```
done in 18.828s
Best params found by grid search:
{'anova__k': 1850, 'l2lr__C': 100.0}
                   precision    recall  f1-score   support

     Ariel Sharon       0.59      0.89      0.71        19
     Colin Powell       0.89      0.83      0.86        59
  Donald Rumsfeld       0.71      0.80      0.75        30
    George W Bush       0.91      0.80      0.85       133
Gerhard Schroeder       0.72      0.78      0.75        27
      Hugo Chavez       0.62      0.56      0.59        18
       Tony Blair       0.71      0.81      0.75        36

         accuracy                           0.80       322
        macro avg       0.74      0.78      0.75       322
     weighted avg       0.81      0.80      0.80       322
```

5.10.10 PCA with LogisticRegression with L2 regularization

```
pca_lrl2_cv = make_pipeline(
    PCA(n_components=150, svd_solver='randomized', whiten=True),
    GridSearchCV(lm.LogisticRegression(max_iter=1000, class_weight='balanced',
                                       fit_intercept=False),
                 {'C': 10. ** np.arange(-3, 3)},
                 cv=5, n_jobs=5))

t0 = time()
pca_lrl2_cv.fit(X=X_train, y=y_train)
print("done in %0.3fs" % (time() - t0))

print("Best params found by grid search:")
print(pca_lrl2_cv.steps[-1][1].best_params_)

y_pred = pca_lrl2_cv.predict(X_test)
print(classification_report(y_test, y_pred, target_names=target_names))
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))
```

Out:

```
done in 0.333s
Best params found by grid search:
{'C': 0.1}
                   precision    recall  f1-score   support

     Ariel Sharon       0.50      0.89      0.64        19
     Colin Powell       0.85      0.78      0.81        59
  Donald Rumsfeld       0.66      0.77      0.71        30
    George W Bush       0.94      0.71      0.81       133
Gerhard Schroeder       0.61      0.74      0.67        27
      Hugo Chavez       0.44      0.67      0.53        18
       Tony Blair       0.69      0.75      0.72        36

         accuracy                           0.74       322
        macro avg       0.67      0.76      0.70       322
     weighted avg       0.79      0.74      0.75       322

[[17  0  1  0  0  1  0]
 [ 4 46  2  3  1  0  3]
 [ 3  1 23  0  0  2  1]
 [ 8  5  8 94  4 12  2]
 [ 1  0  0  1 20  0  5]
 [ 0  1  0  1  3 12  1]
 [ 1  1  1  1  5  0 27]]
```

5.10.11 Basic ConvNet

Note that to simplify, do not use pipeline (scaler + CNN) here. But it would have been simple to do so, since pytorch is warpped in skorch object that is compatible with sklearn.

Sources:

- ConvNet on MNIST
- NeuralNetClassifier

```
class Cnn(nn.Module):
    """Basic ConvNet Conv(1, 32, 64) -> FC(100, 7) -> softmax."""

    def __init__(self, dropout=0.5, fc_size=4928, n_outputs=7, debug=False):
        super(Cnn, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.conv2_drop = nn.Dropout2d(p=dropout)
        self.fc1 = nn.Linear(fc_size, 100)
        self.fc2 = nn.Linear(100, n_outputs)
        self.fc1_drop = nn.Dropout(p=dropout)
        self.debug = debug

    def forward(self, x):
        x = torch.relu(F.max_pool2d(self.conv1(x), 2))
        x = torch.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))

        # flatten over channel, height and width = 1600
        x = x.view(-1, x.size(1) * x.size(2) * x.size(3))

        if self.debug:  # trick to get the size of the first FC
            print("### DEBUG: Shape of last convnet=", x.shape,
                  ". FC size=", np.prod(x.shape[1:]))

        x = torch.relu(self.fc1_drop(self.fc1(x)))
        x = torch.softmax(self.fc2(x), dim=-1)
        return x


torch.manual_seed(0)
cnn = NeuralNetClassifier(
        Cnn,
        max_epochs=100,
        lr=0.001,
        optimizer=torch.optim.Adam,
        device=device,
        train_split=skorch.dataset.CVSplit(cv=5, stratified=True),
        verbose=0)

scaler = preprocessing.MinMaxScaler()
X_train_s = scaler.fit_transform(X_train).reshape(-1, 1, h, w)
```

(continues on next page)

(continued from previous page)

```
X_test_s = scaler.transform(X_test).reshape(-1, 1, h, w)

t0 = time()
cnn.fit(X_train_s, y_train)
print("done in %0.3fs" % (time() - t0))

y_pred = cnn.predict(X_test_s)
print(classification_report(y_test, y_pred, target_names=target_names))
```

Out:

```
done in 39.086s
                   precision    recall  f1-score   support

     Ariel Sharon       0.79      0.79      0.79        19
     Colin Powell       0.95      0.90      0.92        59
  Donald Rumsfeld       0.79      0.77      0.78        30
    George W Bush       0.88      0.95      0.91       133
Gerhard Schroeder       0.71      0.89      0.79        27
      Hugo Chavez       0.78      0.39      0.52        18
       Tony Blair       0.90      0.78      0.84        36

         accuracy                           0.86       322
        macro avg       0.83      0.78      0.79       322
     weighted avg       0.86      0.86      0.85       322
```

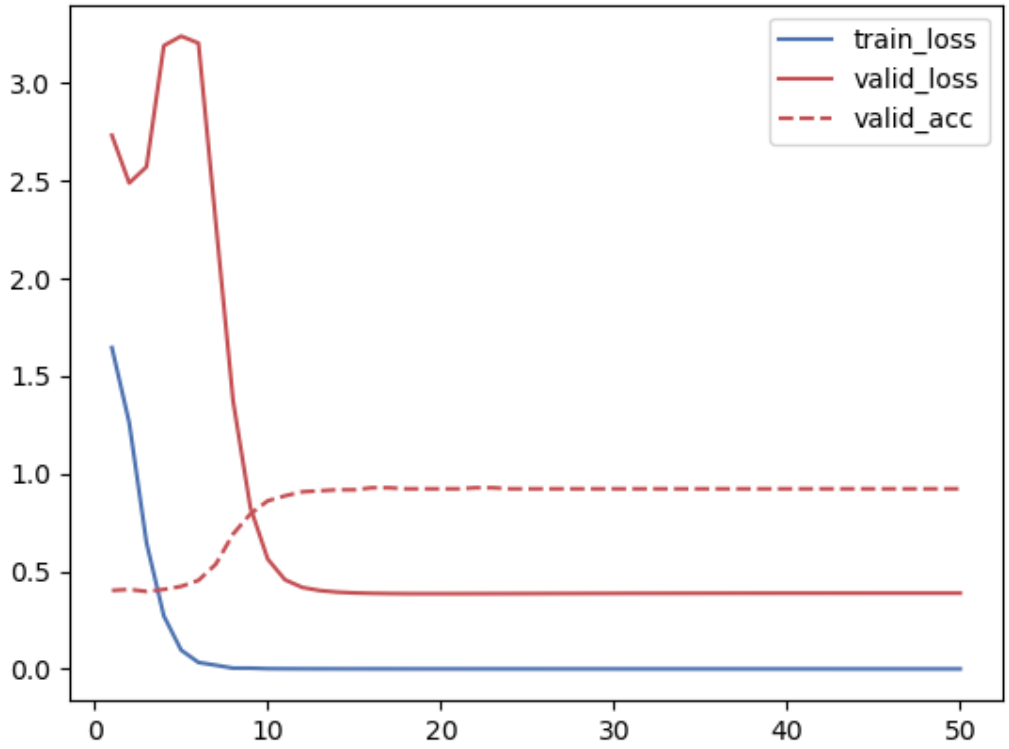
5.10.12 ConvNet with Resnet18

```
class Resnet18(nn.Module):
    """ResNet 18, pretrained, with one input chanel and 7 outputs."""

    def __init__(self, in_channels=1, n_outputs=7):
        super(Resnet18, self).__init__()

        # self.model = torchvision.models.resnet18()
        self.model = torchvision.models.resnet18(pretrained=True)

        # original definition of the first layer on the renset class
        # self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
        #                        bias=False)
        # one channel input (greyscale):
        self.model.conv1 = nn.Conv2d(in_channels, 64, kernel_size=7, stride=2,
                                     padding=3, bias=False)

        # Last layer
        num_ftrs = self.model.fc.in_features
        self.model.fc = nn.Linear(num_ftrs, n_outputs)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        return self.model(x)


torch.manual_seed(0)
resnet = NeuralNetClassifier(
    Resnet18,
    # `CrossEntropyLoss` combines `LogSoftmax and `NLLLoss`
    criterion=nn.CrossEntropyLoss,
    max_epochs=50,
    batch_size=128,  # default value
    optimizer=torch.optim.Adam,
    # optimizer=torch.optim.SGD,
    optimizer__lr=0.001,
    optimizer__betas=(0.9, 0.999),
    optimizer__eps=1e-4,
    optimizer__weight_decay=0.0001,  # L2 regularization
    # Shuffle training data on each epoch
    # iterator_train__shuffle=True,
    train_split=skorch.dataset.CVSplit(cv=5, stratified=True),
    device=device,
    verbose=0)

scaler = preprocessing.MinMaxScaler()
X_train_s = scaler.fit_transform(X_train).reshape(-1, 1, h, w)
X_test_s = scaler.transform(X_test).reshape(-1, 1, h, w)

t0 = time()
resnet.fit(X_train_s, y_train)
print("done in %0.3fs" % (time() - t0))

# Continue training a model (warm re-start):
# resnet.partial_fit(X_train_s, y_train)

y_pred = resnet.predict(X_test_s)
print(classification_report(y_test, y_pred, target_names=target_names))

epochs = np.arange(len(resnet.history[:, 'train_loss'])) + 1
plt.plot(epochs, resnet.history[:, 'train_loss'], '-b', label='train_loss')
plt.plot(epochs, resnet.history[:, 'valid_loss'], '-r', label='valid_loss')
plt.plot(epochs, resnet.history[:, 'valid_acc'], '--r', label='valid_acc')
plt.legend()
plt.show()
```



Out:

```
done in 116.626s
                   precision    recall  f1-score   support

     Ariel Sharon       0.93      0.68      0.79        19
     Colin Powell       0.93      0.93      0.93        59
  Donald Rumsfeld       0.90      0.87      0.88        30
    George W Bush       0.84      0.96      0.90       133
Gerhard Schroeder       0.80      0.74      0.77        27
      Hugo Chavez       0.91      0.56      0.69        18
       Tony Blair       0.94      0.81      0.87        36

         accuracy                           0.87       322
        macro avg       0.89      0.79      0.83       322
     weighted avg       0.88      0.87      0.87       322
```

Total running time of the script: (5 minutes 46.507 seconds)

CHAPTER

SIX

DEEP LEARNING

6.1 Backpropagation

6.1.1 Course outline:

1. Backpropagation and chaine rule
2. Lab: with numpy and pytorch

```
%matplotlib inline
```

6.1.2 Backpropagation and chaine rule

We will set up a two layer network source pytorch tuto :

$$\mathbf{Y} = \max(\mathbf{X}\mathbf{W}^{(1)}, 0)\mathbf{W}^{(2)}$$

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x using Euclidean error.

Chaine rule

Forward pass with **local** partial derivatives of ouput given inputs:

$$\begin{array}{ccccccc}
x \rightarrow & \boxed{z^{(1)} = x^T w^{(1)}} & \rightarrow & \boxed{h^{(1)} = \max(z^{(1)}, 0)} & \rightarrow & \boxed{z^{(2)} = h^{(1)T} w^{(2)}} & \rightarrow & \boxed{L(z^{(2)}, y) = (z^{(2)} - y)^2} \\
w^{(1)} \nearrow & & & w^{(2)} \nearrow & & & & \\
 & \dfrac{\partial z^{(1)}}{\partial w^{(1)}} = x & & \dfrac{\partial h^{(1)}}{\partial z^{(1)}} = \{^{1 \text{ if } z^{(1)}>0}_{\text{else } 0} & & \dfrac{\partial z^{(2)}}{\partial w^{(2)}} = h^{(1)} & & \dfrac{\partial L}{\partial z^{(2)}} = 2(z^{(2)} - y) \\
 & \dfrac{\partial z^{(1)}}{\partial x} = w^{(1)} & & & & \dfrac{\partial z^{(2)}}{\partial h^{(1)}} = w^{(2)} & &
\end{array}$$

Backward: compute gradient of the loss given each parameters vectors applying chaine rule from the loss downstream to the parameters:

For $w^{(2)}$:

$$\frac{\partial L}{\partial w^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(2)}} \tag{6.1}$$

$$= 2(z^{(2)} - y)h^{(1)} \tag{6.2}$$

For $w^{(1)}$:

$$\frac{\partial L}{\partial w^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial w^{(1)}} \tag{6.3}$$

$$= 2(z^{(2)} - y)w^{(2)} \{^{1 \text{ if } z^{(1)}>0}_{\text{else } 0} x \tag{6.4}$$

Recap: Vector derivatives

Given a function $z = x$ with z the output, x the input and w the coeficients.

- Scalar to Scalar: $x \in \mathbb{R}, z \in \mathbb{R}$, $w \in \mathbb{R}$

Regular derivative:

$$\frac{\partial z}{\partial w} = x \in \mathbb{R}$$

If w changes by a small amount, how much will z change?

- Vector to Scalar: $x \in \mathbb{R}^N, z \in \mathbb{R}$, $w \in \mathbb{R}^N$

Derivative is **Gradient** of partial derivative: $\frac{\partial z}{\partial w} \in \mathbb{R}^N$

$$\frac{\partial z}{\partial w} = \nabla_w z = \begin{bmatrix} \frac{\partial z}{\partial w_1} \\ \vdots \\ \frac{\partial z}{\partial w_i} \\ \vdots \\ \frac{\partial z}{\partial w_N} \end{bmatrix} \tag{6.5}$$

For each element w_i of w, if it changes by a small amount then how much will y change?

- Vector to Vector: $w \in \mathbb{R}^N, z \in \mathbb{R}^M$

Derivative is **Jacobian** of partial derivative:

TO COMPLETE

$\frac{\partial z}{\partial w} \in \mathbb{R}^{N \times M}$

Backpropagation summary

Backpropagation algorithm in a graph: 1. Forward pass, for each node compute local partial derivatives of ouput given inputs 2. Backward pass: apply chain rule from the end to each parameters - Update parameter with gradient descent using the current upstream gradient and the current local gradient - Compute upstream gradient for the backward nodes

Think locally and remember that at each node: - For the loss the gradient is the error - At each step, the upstream gradient is obtained by multiplying the upstream gradient (an error) with the current parameters (vector of matrix). - At each step, the current local gradient equal the input, therfore the current update is the current upstream gradient time the input.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn.model_selection
```

6.1.3 Lab: with numpy and pytorch

Load iris data set

Goal: Predict Y = [petal_length, petal_width] = f(X = [sepal_length, sepal_width])

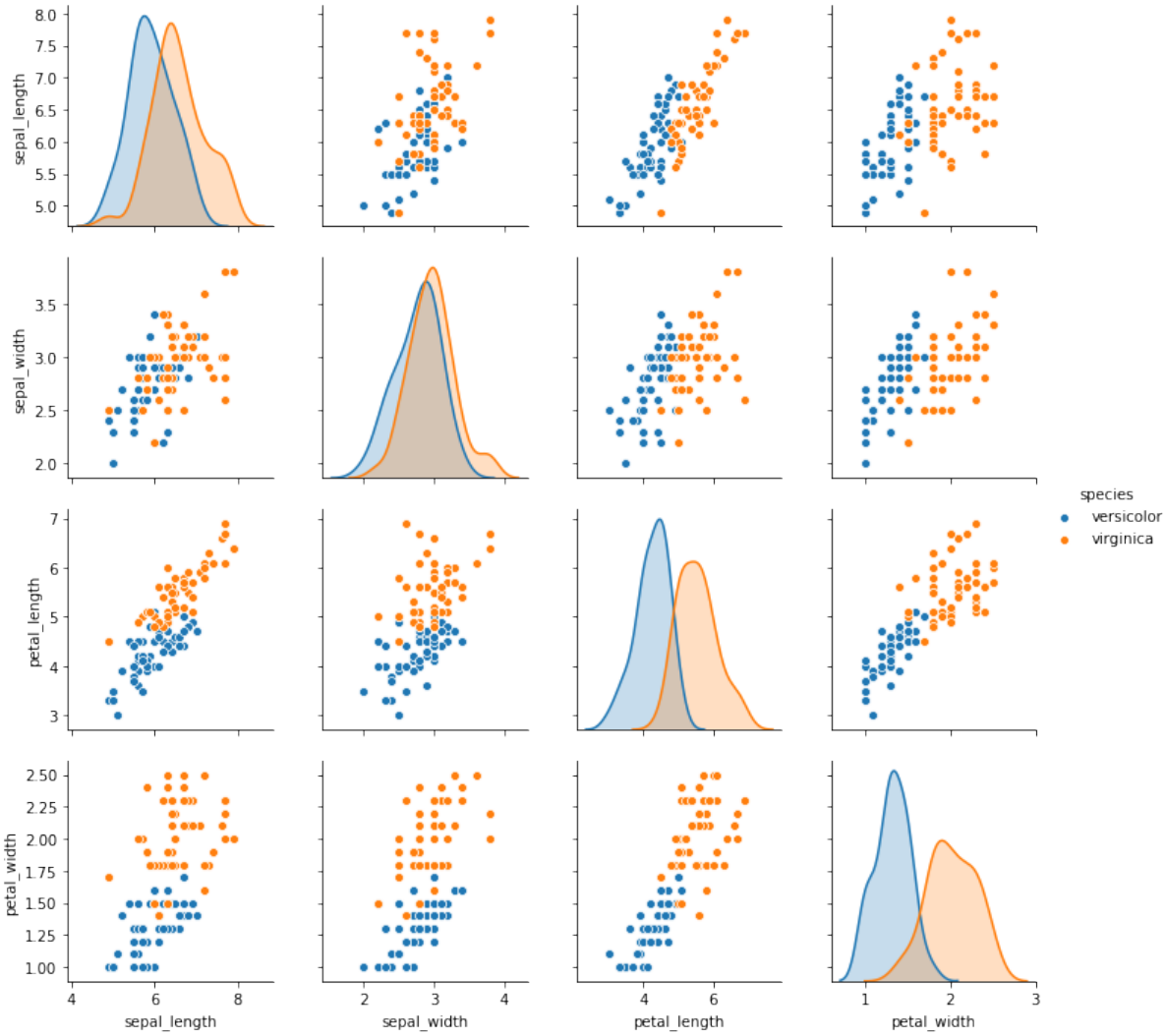
- Plot data with seaborn
- Remove setosa samples
- Recode 'versicolor':1, 'virginica':2
- Scale X and Y
- Split data in train/test 50%/50%

```
iris = sns.load_dataset("iris")
#g = sns.pairplot(iris, hue="species")
df = iris[iris.species != "setosa"]
g = sns.pairplot(df, hue="species")
df['species_n'] = iris.species.map({'versicolor':1, 'virginica':2})

# Y = 'petal_length', 'petal_width'; X = 'sepal_length', 'sepal_width')
X_iris = np.asarray(df.loc[:, ['sepal_length', 'sepal_width']], dtype=np.float32)
Y_iris = np.asarray(df.loc[:, ['petal_length', 'petal_width']], dtype=np.float32)
label_iris = np.asarray(df.species_n, dtype=int)

# Scale
from sklearn.preprocessing import StandardScaler
scalerx, scalery = StandardScaler(), StandardScaler()
X_iris = scalerx.fit_transform(X_iris)
Y_iris = StandardScaler().fit_transform(Y_iris)

# Split train test
X_iris_tr, X_iris_val, Y_iris_tr, Y_iris_val, label_iris_tr, label_iris_val = \
    sklearn.model_selection.train_test_split(X_iris, Y_iris, label_iris, train_
↪size=0.5, stratify=label_iris)
```

(continues on next page)

(continued from previous page)

```
/home/edouard/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:␣
↪SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/
↪indexing.html#indexing-view-versus-copy
  """
```



Backpropagation with numpy

This implementation uses numpy to manually compute the forward pass, loss, and backward pass.

```
# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val

def two_layer_regression_numpy_train(X, Y, X_val, Y_val, lr, nite):
    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    # N, D_in, H, D_out = 64, 1000, 100, 10
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    W1 = np.random.randn(D_in, H)
    W2 = np.random.randn(H, D_out)

    losses_tr, losses_val = list(), list()

    learning_rate = lr
    for t in range(nite):
        # Forward pass: compute predicted y
        z1 = X.dot(W1)
        h1 = np.maximum(z1, 0)
        Y_pred = h1.dot(W2)

        # Compute and print loss
        loss = np.square(Y_pred - Y).sum()

        # Backprop to compute gradients of w1 and w2 with respect to loss
        grad_y_pred = 2.0 * (Y_pred - Y)
        grad_w2 = h1.T.dot(grad_y_pred)
        grad_h1 = grad_y_pred.dot(W2.T)
        grad_z1 = grad_h1.copy()
        grad_z1[z1 < 0] = 0
        grad_w1 = X.T.dot(grad_z1)

        # Update weights
        W1 -= learning_rate * grad_w1
        W2 -= learning_rate * grad_w2

        # Forward pass for validation set: compute predicted y
        z1 = X_val.dot(W1)
        h1 = np.maximum(z1, 0)
        y_pred_val = h1.dot(W2)
        loss_val = np.square(y_pred_val - Y_val).sum()

        losses_tr.append(loss)
        losses_val.append(loss_val)

        if t % 10 == 0:
            print(t, loss, loss_val)
```

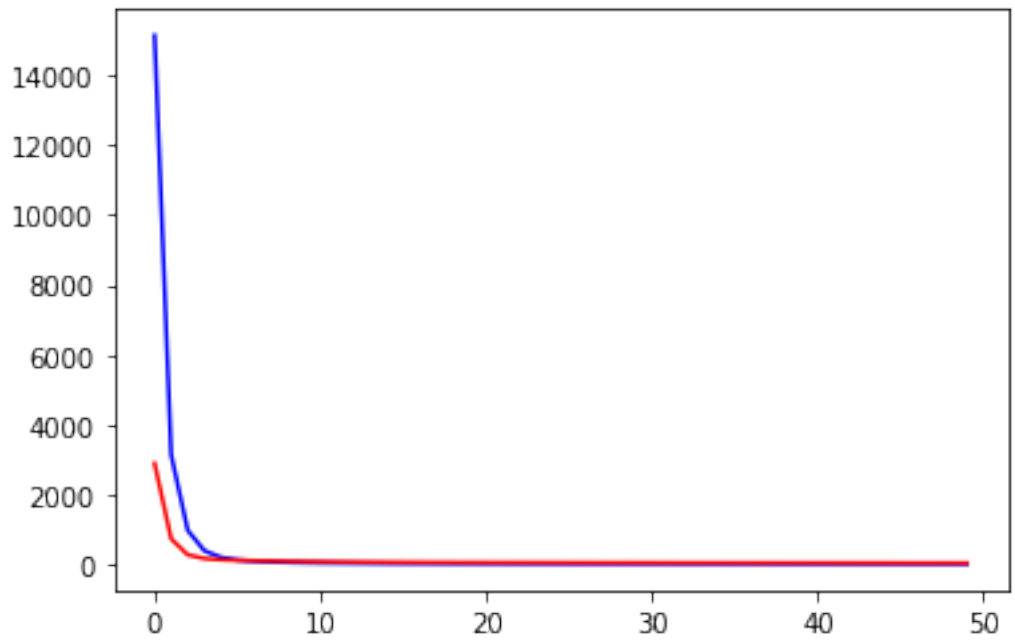
(continues on next page)

(continued from previous page)

```
    return W1, W2, losses_tr, losses_val

W1, W2, losses_tr, losses_val = two_layer_regression_numpy_train(X=X_iris_tr, Y=Y_
↪iris_tr, X_val=X_iris_val, Y_val=Y_iris_val,
                                                                 lr=1e-4, nite=50)
plt.plot(np.arange(len(losses_tr)), losses_tr, "-b", np.arange(len(losses_val)),␣
↪losses_val, "-r")
```

```
0 15126.224825529907 2910.260853330454
10 71.5381374591153 104.97056197642135
20 50.756938353833334 80.02800827986354
30 46.546510744624236 72.85211241738614
40 44.41413064447564 69.31127324764276
```

```
[<matplotlib.lines.Line2D at 0x7f960cf5e9b0>,
 <matplotlib.lines.Line2D at 0x7f960cf5eb00>]
```



Backpropagation with PyTorch Tensors

source

Numpy is a great framework, but it cannot utilize GPUs to accelerate its numerical computations. For modern deep neural networks, GPUs often provide speedups of 50x or greater, so unfortunately numpy won't be enough for modern deep learning. Here we introduce the most fundamental PyTorch concept: the Tensor. A PyTorch Tensor is conceptually identical to a numpy array: a Tensor is an n-dimensional array, and PyTorch provides many functions for operating on these Tensors. Behind the scenes, Tensors can keep track of a computational graph and gradients, but they're also useful as a generic tool for scientific computing. Also unlike numpy, PyTorch Tensors can utilize GPUs to accelerate their numeric computations. To run a

PyTorch Tensor on GPU, you simply need to cast it to a new datatype. Here we use PyTorch Tensors to fit a two-layer network to random data. Like the numpy example above we need to manually implement the forward and backward passes through the network:

```
import torch

# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val

def two_layer_regression_tensor_train(X, Y, X_val, Y_val, lr, nite):

    dtype = torch.float
    device = torch.device("cpu")
    # device = torch.device("cuda:0") # Uncomment this to run on GPU

    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    # Create random input and output data
    X = torch.from_numpy(X)
    Y = torch.from_numpy(Y)
    X_val = torch.from_numpy(X_val)
    Y_val = torch.from_numpy(Y_val)

    # Randomly initialize weights
    W1 = torch.randn(D_in, H, device=device, dtype=dtype)
    W2 = torch.randn(H, D_out, device=device, dtype=dtype)

    losses_tr, losses_val = list(), list()

    learning_rate = lr
    for t in range(nite):
        # Forward pass: compute predicted y
        z1 = X.mm(W1)
        h1 = z1.clamp(min=0)
        y_pred = h1.mm(W2)

        # Compute and print loss
        loss = (y_pred - Y).pow(2).sum().item()

        # Backprop to compute gradients of w1 and w2 with respect to loss
        grad_y_pred = 2.0 * (y_pred - Y)
        grad_w2 = h1.t().mm(grad_y_pred)
        grad_h1 = grad_y_pred.mm(W2.t())
        grad_z1 = grad_h1.clone()
        grad_z1[z1 < 0] = 0
        grad_w1 = X.t().mm(grad_z1)

        # Update weights using gradient descent
        W1 -= learning_rate * grad_w1
        W2 -= learning_rate * grad_w2
```

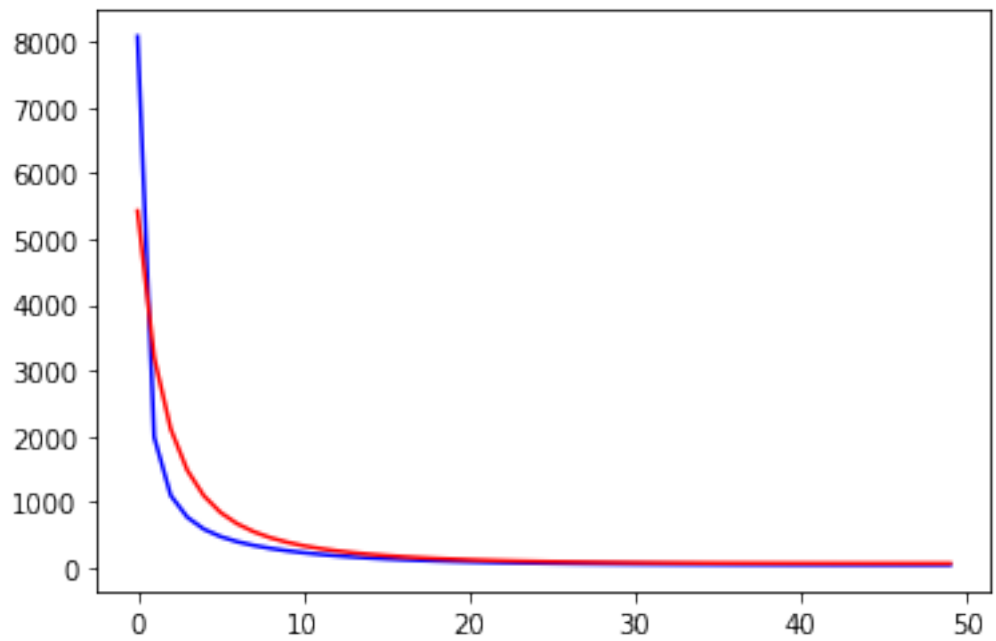
(continues on next page)

(continued from previous page)

```
        # Forward pass for validation set: compute predicted y
        z1 = X_val.mm(W1)
        h1 = z1.clamp(min=0)
        y_pred_val = h1.mm(W2)
        loss_val = (y_pred_val - Y_val).pow(2).sum().item()

        losses_tr.append(loss)
        losses_val.append(loss_val)

        if t % 10 == 0:
            print(t, loss, loss_val)

    return W1, W2, losses_tr, losses_val

W1, W2, losses_tr, losses_val = two_layer_regression_tensor_train(X=X_iris_tr,␣
↪Y=Y_iris_tr, X_val=X_iris_val, Y_val=Y_iris_val,
                                                                  lr=1e-4, nite=50)

plt.plot(np.arange(len(losses_tr)), losses_tr, "-b", np.arange(len(losses_val)),␣
↪losses_val, "-r")
```

```
0 8086.1591796875 5429.57275390625
10 225.77589416503906 331.83734130859375
20 86.46501159667969 117.72447204589844
30 52.375606536865234 73.84156036376953
40 43.16458511352539 64.0667495727539
```

```
[<matplotlib.lines.Line2D at 0x7f960033c470>,
 <matplotlib.lines.Line2D at 0x7f960033c5c0>]
```



Backpropagation with PyTorch: Tensors and autograd

source

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. A PyTorch Tensor represents a node in a computational graph. If `x` is a Tensor that has `x.requires_grad=True` then `x.grad` is another Tensor holding the gradient of `x` with respect to some scalar value.

```
import torch

# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val
# del X, Y, X_val, Y_val

def two_layer_regression_autograd_train(X, Y, X_val, Y_val, lr, nite):

    dtype = torch.float
    device = torch.device("cpu")
    # device = torch.device("cuda:0") # Uncomment this to run on GPU

    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    # Setting requires_grad=False indicates that we do not need to compute␣
↪gradients
    # with respect to these Tensors during the backward pass.
    X = torch.from_numpy(X)
    Y = torch.from_numpy(Y)
    X_val = torch.from_numpy(X_val)
    Y_val = torch.from_numpy(Y_val)

    # Create random Tensors for weights.
    # Setting requires_grad=True indicates that we want to compute gradients with
    # respect to these Tensors during the backward pass.
    W1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
    W2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

    losses_tr, losses_val = list(), list()

    learning_rate = lr
    for t in range(nite):
        # Forward pass: compute predicted y using operations on Tensors; these
        # are exactly the same operations we used to compute the forward pass␣
↪using
        # Tensors, but we do not need to keep references to intermediate values␣
↪since
        # we are not implementing the backward pass by hand.
        y_pred = X.mm(W1).clamp(min=0).mm(W2)
```

(continues on next page)

(continued from previous page)

```
        # Compute and print loss using operations on Tensors.
        # Now loss is a Tensor of shape (1,)
        # loss.item() gets the scalar value held in the loss.
        loss = (y_pred - Y).pow(2).sum()

        # Use autograd to compute the backward pass. This call will compute the
        # gradient of loss with respect to all Tensors with requires_grad=True.
        # After this call w1.grad and w2.grad will be Tensors holding the gradient
        # of the loss with respect to w1 and w2 respectively.
        loss.backward()

        # Manually update weights using gradient descent. Wrap in torch.no_grad()
        # because weights have requires_grad=True, but we don't need to track this
        # in autograd.
        # An alternative way is to operate on weight.data and weight.grad.data.
        # Recall that tensor.data gives a tensor that shares the storage with
        # tensor, but doesn't track history.
        # You can also use torch.optim.SGD to achieve this.
        with torch.no_grad():
            W1 -= learning_rate * W1.grad
            W2 -= learning_rate * W2.grad

            # Manually zero the gradients after updating weights
            W1.grad.zero_()
            W2.grad.zero_()

            y_pred = X_val.mm(W1).clamp(min=0).mm(W2)

            # Compute and print loss using operations on Tensors.
            # Now loss is a Tensor of shape (1,)
            # loss.item() gets the scalar value held in the loss.
            loss_val = (y_pred - Y).pow(2).sum()

        if t % 10 == 0:
            print(t, loss.item(), loss_val.item())

        losses_tr.append(loss.item())
        losses_val.append(loss_val.item())

    return W1, W2, losses_tr, losses_val

W1, W2, losses_tr, losses_val = two_layer_regression_autograd_train(X=X_iris_tr,␣
↪Y=Y_iris_tr, X_val=X_iris_val, Y_val=Y_iris_val,
                                                                    lr=1e-4, nite=50)
plt.plot(np.arange(len(losses_tr)), losses_tr, "-b", np.arange(len(losses_val)),␣
↪losses_val, "-r")
```

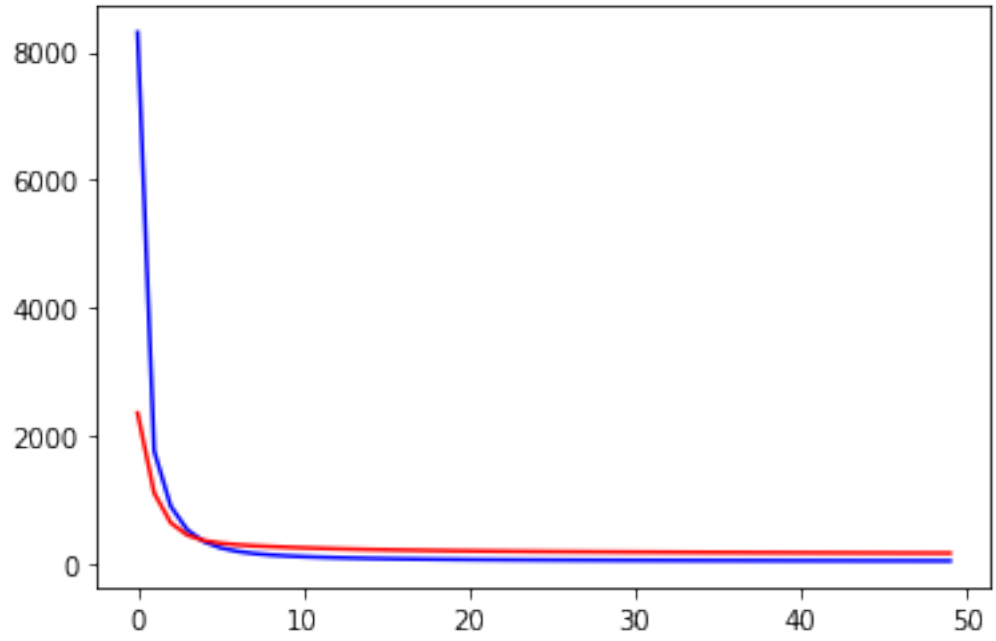
```
0 8307.1806640625 2357.994873046875
```

(continues on next page)

(continued from previous page)

```
10 111.97289276123047 250.04209899902344
20 65.83244323730469 201.63694763183594
30 53.70908737182617 183.17051696777344
40 48.719329833984375 173.3616943359375
```

```
[<matplotlib.lines.Line2D at 0x7f95ff2ad978>,
 <matplotlib.lines.Line2D at 0x7f95ff2adac8>]
```



Backpropagation with PyTorch: nn

source

This implementation uses the nn package from PyTorch to build the network. PyTorch autograd makes it easy to define computational graphs and take gradients, but raw autograd can be a bit too low-level for defining complex neural networks; this is where the nn package can help. The nn package defines a set of Modules, which you can think of as a neural network layer that has produces output from input and may have some trainable weights.

```
import torch

# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val
# del X, Y, X_val, Y_val

def two_layer_regression_nn_train(X, Y, X_val, Y_val, lr, nite):

    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    X = torch.from_numpy(X)
```

(continues on next page)

(continued from previous page)

```
    Y = torch.from_numpy(Y)
    X_val = torch.from_numpy(X_val)
    Y_val = torch.from_numpy(Y_val)

    # Use the nn package to define our model as a sequence of layers. nn.
↪Sequential
    # is a Module which contains other Modules, and applies them in sequence to
    # produce its output. Each Linear Module computes output from input using a
    # linear function, and holds internal Tensors for its weight and bias.
    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
    )

    # The nn package also contains definitions of popular loss functions; in this
    # case we will use Mean Squared Error (MSE) as our loss function.
    loss_fn = torch.nn.MSELoss(reduction='sum')

    losses_tr, losses_val = list(), list()

    learning_rate = lr
    for t in range(nite):
        # Forward pass: compute predicted y by passing x to the model. Module␣
↪objects
        # override the __call__ operator so you can call them like functions. When
        # doing so you pass a Tensor of input data to the Module and it produces
        # a Tensor of output data.
        y_pred = model(X)

        # Compute and print loss. We pass Tensors containing the predicted and␣
↪true
        # values of y, and the loss function returns a Tensor containing the
        # loss.
        loss = loss_fn(y_pred, Y)

        # Zero the gradients before running the backward pass.
        model.zero_grad()

        # Backward pass: compute gradient of the loss with respect to all the␣
↪learnable
        # parameters of the model. Internally, the parameters of each Module are␣
↪stored
        # in Tensors with requires_grad=True, so this call will compute gradients␣
↪for
        # all learnable parameters in the model.
        loss.backward()

        # Update the weights using gradient descent. Each parameter is a Tensor,␣
↪so
```

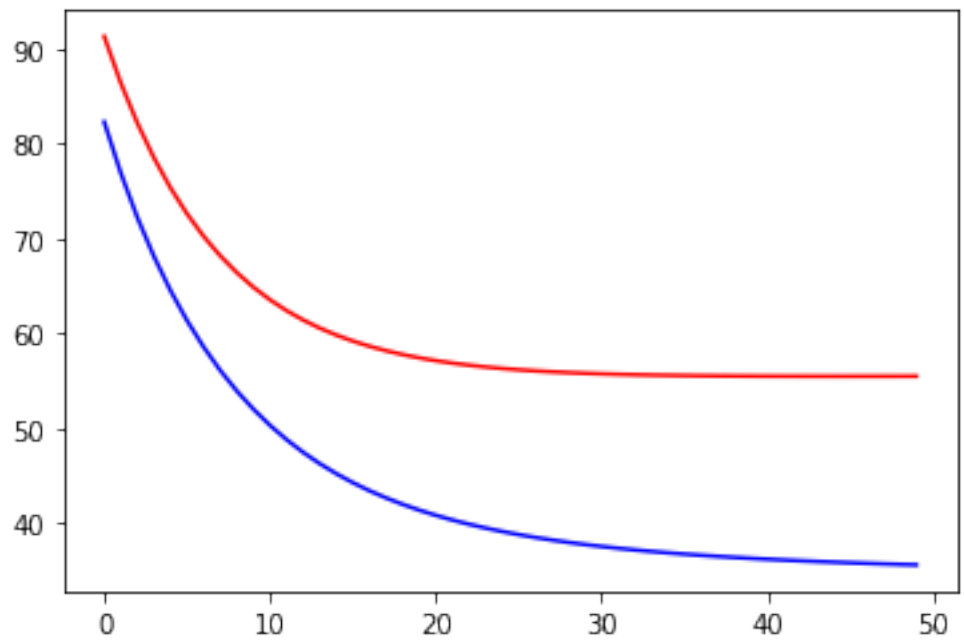
(continues on next page)

(continued from previous page)

```
        # we can access its gradients like we did before.
        with torch.no_grad():
            for param in model.parameters():
                param -= learning_rate * param.grad
            y_pred = model(X_val)
            loss_val = (y_pred - Y_val).pow(2).sum()

        if t % 10 == 0:
            print(t, loss.item(), loss_val.item())

        losses_tr.append(loss.item())
        losses_val.append(loss_val.item())

    return model, losses_tr, losses_val

model, losses_tr, losses_val = two_layer_regression_nn_train(X=X_iris_tr, Y=Y_
↪iris_tr, X_val=X_iris_val, Y_val=Y_iris_val,
                                                                lr=1e-4, nite=50)

plt.plot(np.arange(len(losses_tr)), losses_tr, "-b", np.arange(len(losses_val)),␣
↪losses_val, "-r")
```

```
0 82.32025146484375 91.3389892578125
10 50.322200775146484 63.563087463378906
20 40.825225830078125 57.13555145263672
30 37.53572082519531 55.74506378173828
40 36.191200256347656 55.499732971191406
```

```
[<matplotlib.lines.Line2D at 0x7f95ff296668>,
 <matplotlib.lines.Line2D at 0x7f95ff2967b8>]
```



Backpropagation with PyTorch optim

This implementation uses the nn package from PyTorch to build the network. Rather than manually updating the weights of the model as we have been doing, we use the optim package to define an Optimizer that will update the weights for us. The optim package defines many optimization algorithms that are commonly used for deep learning, including SGD+momentum, RMSProp, Adam, etc.

```
import torch

# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val

def two_layer_regression_nn_optim_train(X, Y, X_val, Y_val, lr, nite):

    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    X = torch.from_numpy(X)
    Y = torch.from_numpy(Y)
    X_val = torch.from_numpy(X_val)
    Y_val = torch.from_numpy(Y_val)

    # Use the nn package to define our model and loss function.
    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
    )
    loss_fn = torch.nn.MSELoss(reduction='sum')

    losses_tr, losses_val = list(), list()

    # Use the optim package to define an Optimizer that will update the weights of
    # the model for us. Here we will use Adam; the optim package contains many␣
↪other
    # optimization algoriths. The first argument to the Adam constructor tells the
    # optimizer which Tensors it should update.
    learning_rate = lr
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    for t in range(nite):
        # Forward pass: compute predicted y by passing x to the model.
        y_pred = model(X)

        # Compute and print loss.
        loss = loss_fn(y_pred, Y)

        # Before the backward pass, use the optimizer object to zero all of the
        # gradients for the variables it will update (which are the learnable
        # weights of the model). This is because by default, gradients are
        # accumulated in buffers( i.e, not overwritten) whenever .backward()
```

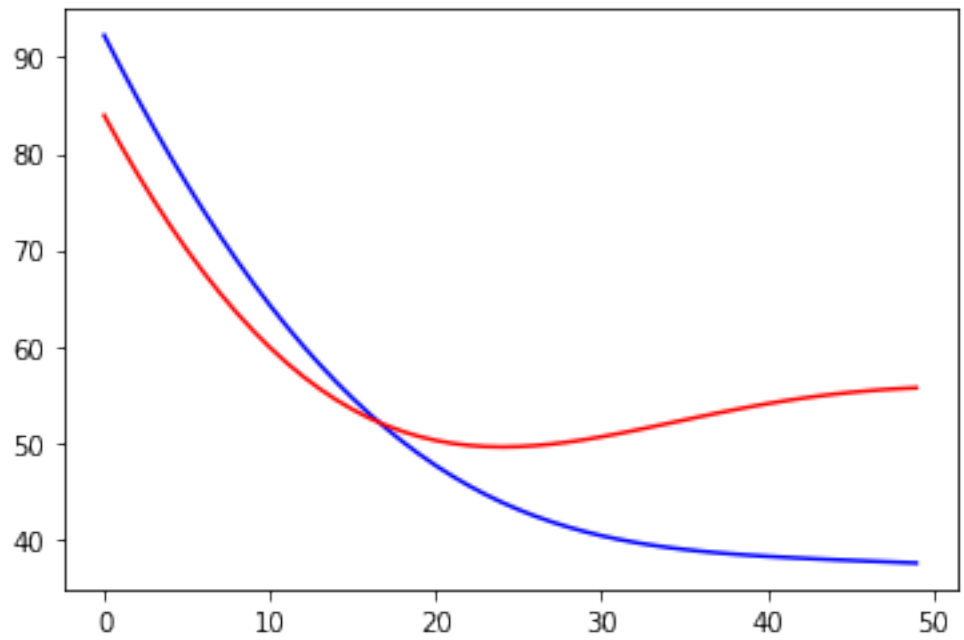
(continues on next page)

(continued from previous page)

```
        # is called. Checkout docs of torch.autograd.backward for more details.
        optimizer.zero_grad()

        # Backward pass: compute gradient of the loss with respect to model
        # parameters
        loss.backward()

        # Calling the step function on an Optimizer makes an update to its
        # parameters
        optimizer.step()

        with torch.no_grad():
            y_pred = model(X_val)
            loss_val = loss_fn(y_pred, Y_val)

        if t % 10 == 0:
            print(t, loss.item(), loss_val.item())

        losses_tr.append(loss.item())
        losses_val.append(loss_val.item())

    return model, losses_tr, losses_val

model, losses_tr, losses_val = two_layer_regression_nn_optim_train(X=X_iris_tr,␣
↪Y=Y_iris_tr, X_val=X_iris_val, Y_val=Y_iris_val,
                                                                   lr=1e-3, nite=50)
plt.plot(np.arange(len(losses_tr)), losses_tr, "-b", np.arange(len(losses_val)),␣
↪losses_val, "-r")
```

```
0 92.271240234375 83.96189880371094
10 64.25907135009766 59.872535705566406
20 47.6252555847168 50.228126525878906
30 40.33802032470703 50.60377502441406
40 38.19448471069336 54.03163528442383
```

```
[<matplotlib.lines.Line2D at 0x7f95ff200080>,
 <matplotlib.lines.Line2D at 0x7f95ff2001d0>]
```



6.2 Multilayer Perceptron (MLP)

6.2.1 Course outline:

1. Recall of linear classifier
2. MLP with scikit-learn
3. MLP with pytorch
4. Test several MLP architectures
5. Limits of MLP

Sources:

Deep learning

- cs231n.stanford.edu

Pytorch

- WWW tutorials
- github tutorials
- github examples

MNIST and pytorch:

- MNIST nextjournal.com/gkoehler/pytorch-mnist
- MNIST github/pytorch/examples
- MNIST kaggle

```
%matplotlib inline

import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
from torchvision import transforms
from torchvision import datasets
from torchvision import models
#
from pathlib import Path
import matplotlib.pyplot as plt

# Device configuration
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device = 'cpu' # Force CPU
print(device)
```

```
cpu
```

Hyperparameters

6.2.2 Dataset: MNIST Handwritten Digit Recognition

```
from pathlib import Path
WD = os.path.join(Path.home(), "data", "pystatml", "dl_mnist_pytorch")
os.makedirs(WD, exist_ok=True)
os.chdir(WD)
print("Working dir is:", os.getcwd())
os.makedirs("data", exist_ok=True)
os.makedirs("models", exist_ok=True)

def load_mnist(batch_size_train, batch_size_test):

    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,)) # Mean and␣
↪Std of the MNIST dataset
                       ])),
        batch_size=batch_size_train, shuffle=True)

    val_loader = torch.utils.data.DataLoader(
        datasets.MNIST('data', train=False, transform=transforms.Compose([
```

(continues on next page)

(continued from previous page)

```
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,)) # Mean and Std of the␣
↪MNIST dataset
        ])),
        batch_size=batch_size_test, shuffle=True)
    return train_loader, val_loader


train_loader, val_loader = load_mnist(64, 10000)

dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shapes:", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪ 'val']})
print("N input features:", D_in, "Output classes:", D_out)
```

```
Working dir is: /home/ed203246/data/pystatml/dl_mnist_pytorch
Datasets shapes: {'train': torch.Size([60000, 28, 28]), 'val': torch.Size([10000,␣
↪28, 28])}
N input features: 784 Output classes: 10
```

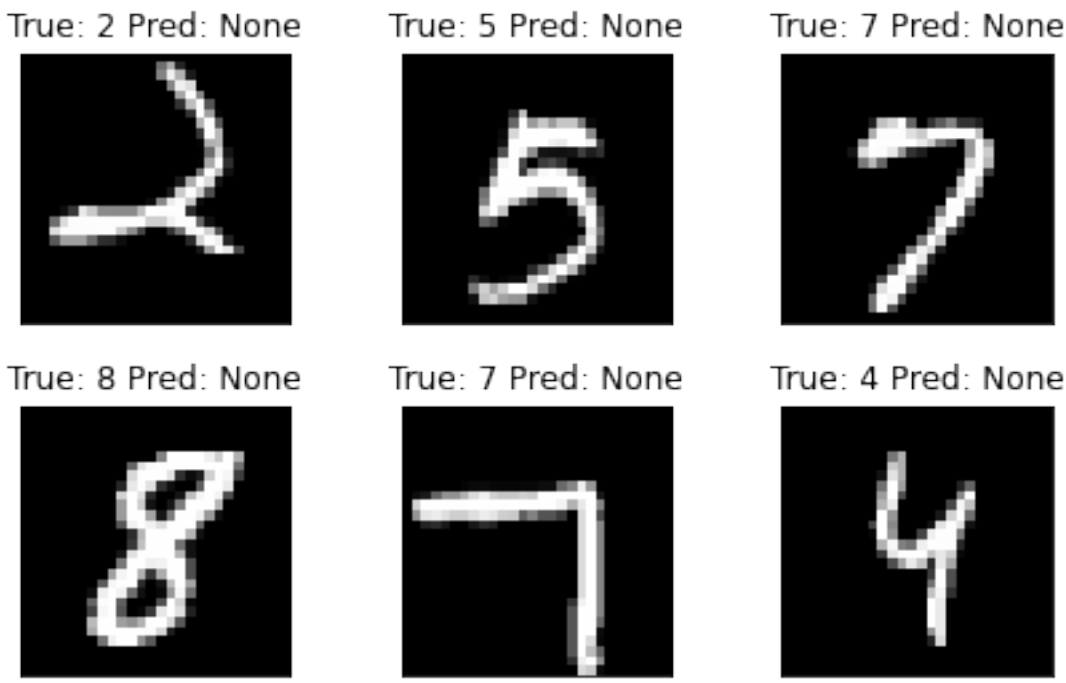
Now let's take a look at some mini-batches examples.

```
batch_idx, (example_data, example_targets) = next(enumerate(train_loader))
print("Train batch:", example_data.shape, example_targets.shape)
batch_idx, (example_data, example_targets) = next(enumerate(val_loader))
print("Val batch:", example_data.shape, example_targets.shape)
```

```
Train batch: torch.Size([64, 1, 28, 28]) torch.Size([64])
Val batch: torch.Size([10000, 1, 28, 28]) torch.Size([10000])
```

So one test data batch is a tensor of shape: . This means we have 1000 examples of 28x28 pixels in grayscale (i.e. no rgb channels, hence the one). We can plot some of them using matplotlib.

```
def show_data_label_prediction(data, y_true, y_pred=None, shape=(2, 3)):
    y_pred = [None] * len(y_true) if y_pred is None else y_pred
    fig = plt.figure()
    for i in range(np.prod(shape)):
        plt.subplot(*shape, i+1)
        plt.tight_layout()
        plt.imshow(data[i][0], cmap='gray', interpolation='none')
        plt.title("True: {} Pred: {}".format(y_true[i], y_pred[i]))
        plt.xticks([])
        plt.yticks([])

show_data_label_prediction(data=example_data, y_true=example_targets, y_pred=None,
↪ shape=(2, 3))
```



6.2.3 Recall of linear classifier

Binary logistic regression

1 neuron as output layer

$$f(x) = \sigma(x^T w)$$

Softmax Classifier (Multinomial Logistic Regression)

- Input x: a vector of dimension (0) (layer 0).
- Ouput $f(x)$ a vector of (1) (layer 1) possible labels

The model as (1) neurons as output layer

$$f(x) = \text{softmax}(x^T W + b)$$

Where W is a $(0) \times (1)$ of coefficients and b is a (1)-dimentional vector of bias.

MNIST classfification using multinomial logistic

source: Logistic regression MNIST

Here we fit a multinomial logistic regression with L2 penalty on a subset of the MNIST digits classification task.

source: scikit-learn.org

```
X_train = train_loader.dataset.data.numpy()
#print(X_train.shape)
X_train = X_train.reshape((X_train.shape[0], -1))
```

(continues on next page)

(continued from previous page)

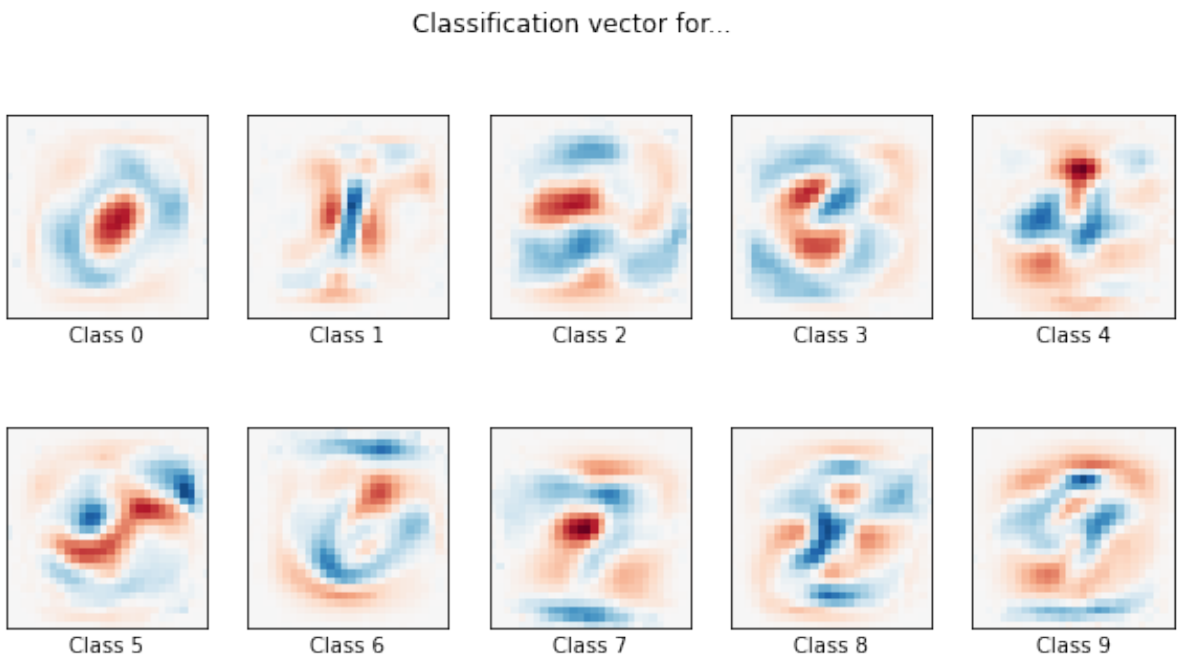
```
y_train = train_loader.dataset.targets.numpy()

X_test = val_loader.dataset.data.numpy()
X_test = X_test.reshape((X_test.shape[0], -1))
y_test = val_loader.dataset.targets.numpy()

print(X_train.shape, y_train.shape)
```

```
(60000, 784) (60000,)
```

```
import matplotlib.pyplot as plt
import numpy as np

#from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression
#from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Turn up tolerance for faster convergence
clf = LogisticRegression(C=50., multi_class='multinomial', solver='sag', tol=0.1)
clf.fit(X_train, y_train)
#sparsity = np.mean(clf.coef_ == 0) * 100
score = clf.score(X_test, y_test)

print("Test score with penalty: %.4f" % score)
```

```
Test score with penalty: 0.8997
```

```
coef = clf.coef_.copy()
plt.figure(figsize=(10, 5))
scale = np.abs(coef).max()
for i in range(10):
    l1_plot = plt.subplot(2, 5, i + 1)
    l1_plot.imshow(coef[i].reshape(28, 28), interpolation='nearest',
                   cmap=plt.cm.RdBu, vmin=-scale, vmax=scale)
    l1_plot.set_xticks(())
    l1_plot.set_yticks(())
    l1_plot.set_xlabel('Class %i' % i)
plt.suptitle('Classification vector for...')

plt.show()
```



6.2.4 Model: Two Layer MLP

MLP with Scikit-learn

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(100, ), max_iter=5, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1,
                    learning_rate_init=0.01, batch_size=64)

mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))

print("Coef shape=", len(mlp.coefs_))

fig, axes = plt.subplots(4, 4)
# use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()
```

```
Iteration 1, loss = 0.28828673
Iteration 2, loss = 0.13388073
```

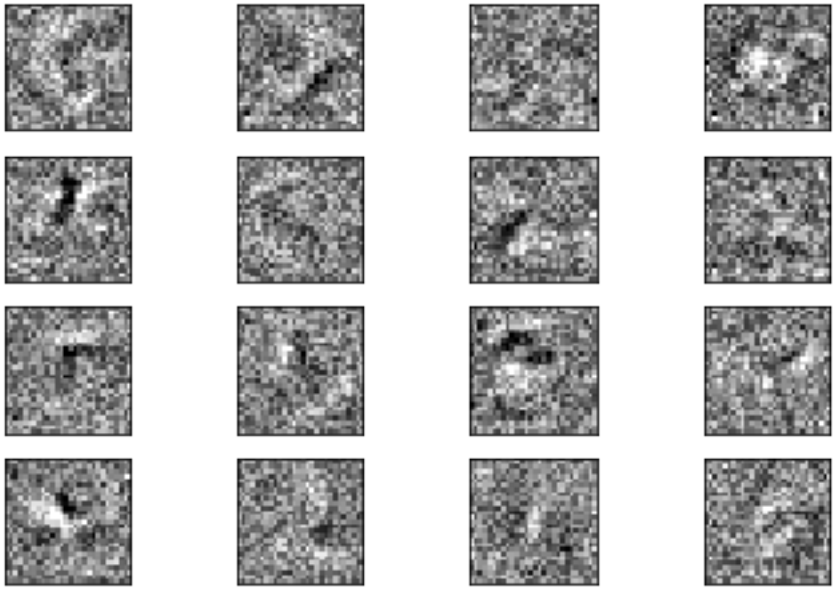
(continues on next page)

(continued from previous page)

```
Iteration 3, loss = 0.09366379
Iteration 4, loss = 0.07317648
Iteration 5, loss = 0.05340251
```

```
/home/ed203246/anaconda3/lib/python3.7/site-packages/sklearn/neural_network/_
↪multilayer_perceptron.py:585: ConvergenceWarning: Stochastic Optimizer: Maximum␣
↪iterations (5) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

```
Training set score: 0.989067
Test set score: 0.971900
Coef shape= 2
```



MLP with pytorch

```
class TwoLayerMLP(nn.Module):

    def __init__(self, d_in, d_hidden, d_out):
        super(TwoLayerMLP, self).__init__()
        self.d_in = d_in

        self.linear1 = nn.Linear(d_in, d_hidden)
        self.linear2 = nn.Linear(d_hidden, d_out)

    def forward(self, X):
        X = X.view(-1, self.d_in)
        X = self.linear1(X)
        return F.log_softmax(self.linear2(X), dim=1)
```

Train the Model

- First we want to make sure our network is in training mode.
- Iterate over epochs
- Alternate train and validation dataset
- Iterate over all training/val data once per epoch. Loading the individual batches is handled by the DataLoader.
- Set the gradients to zero using `optimizer.zero_grad()` since PyTorch by default accumulates gradients.
- Forward pass:
  - `model(inputs)`: Produce the output of our network.
  - `torch.max(outputs, 1)`: softmax predictions.
  - `criterion(outputs, labels)`: loss between the output and the ground truth label.
- In training mode, backward pass `backward()`: collect a new set of gradients which we propagate back into each of the network's parameters using `optimizer.step()`.
- We'll also keep track of the progress with some printouts. In order to create a nice training curve later on we also create two lists for saving training and testing losses. On the x-axis we want to display the number of training examples the network has seen during training.
- Save model state: Neural network modules as well as optimizers have the ability to save and load their internal state using `.state_dict()`. With this we can continue training from previously saved state dicts if needed - we'd just need to call `.load_state_dict(state_dict)`.

```
# %load train_val_model.py
```

```
# %load train_val_model.py
import numpy as np
import torch
import time
import copy


def train_val_model(model, criterion, optimizer, dataloaders, num_epochs=25,
        scheduler=None, log_interval=None):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    # Store losses and accuracies accross epochs
    losses, accuracies = dict(train=[], val=[]), dict(train=[], val=[])

    for epoch in range(num_epochs):
        if log_interval is not None and epoch % log_interval == 0:
            print('Epoch {}/{}'.format(epoch, num_epochs - 1))
```

(continues on next page)

(continued from previous page)

```
            print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            nsamples = 0
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                nsamples += inputs.shape[0]

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if scheduler is not None and phase == 'train':
                scheduler.step()

            #nsamples = dataloaders[phase].dataset.data.shape[0]
            epoch_loss = running_loss / nsamples
            epoch_acc = running_corrects.double() / nsamples

            losses[phase].append(epoch_loss)
            accuracies[phase].append(epoch_acc)
            if log_interval is not None and epoch % log_interval == 0:
                print('{} Loss: {:.4f} Acc: {:.2f}%'.format(
```

(continues on next page)

(continued from previous page)

```
                    phase, epoch_loss, 100 * epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
        if log_interval is not None and epoch % log_interval == 0:
            print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:.2f}%'.format(100 * best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)

    return model, losses, accuracies
```

Run one epoch and save the model

```
model = TwoLayerMLP(D_in, 50, D_out).to(device)
print(next(model.parameters()).is_cuda)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =", np.sum([np.prod(parameter.shape) for␣
↪parameter in model.parameters()]))

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=1, log_interval=1)

print(next(model.parameters()).is_cuda)
torch.save(model.state_dict(), 'models/mod-%s.pth' % model.__class__.__name__)
```

```
False
torch.Size([50, 784])
torch.Size([50])
torch.Size([10, 50])
torch.Size([10])
Total number of parameters = 39760
Epoch 0/0
----------
train Loss: 0.4431 Acc: 87.93%
```

(continues on next page)

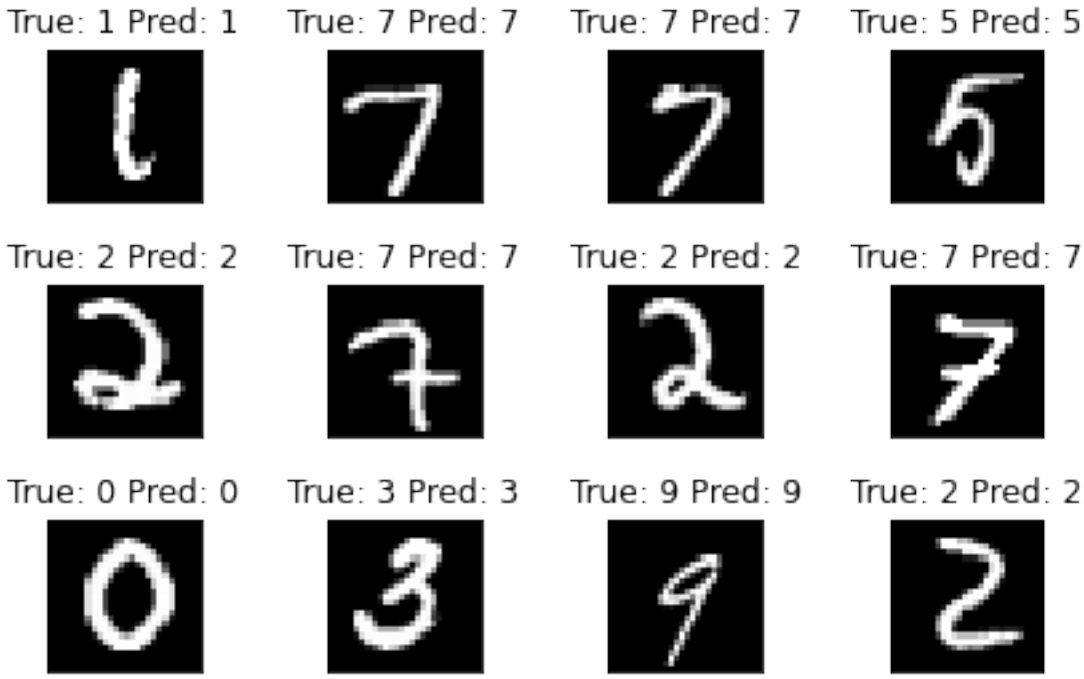
(continued from previous page)

```
val Loss: 0.3062 Acc: 91.21%

Training complete in 0m 7s
Best val Acc: 91.21%
False
```

Use the model to make new predictions. Consider the device, ie, load data on device `example_data.to(device)` from prediction, then move back to cpu `example_data.cpu()`.

```
batch_idx, (example_data, example_targets) = next(enumerate(val_loader))
example_data = example_data.to(device)

with torch.no_grad():
  output = model(example_data).cpu()

example_data = example_data.cpu()

# print(output.is_cuda)

# Softmax predictions
preds = output.argmax(dim=1)

print("Output shape=", output.shape, "label shape=", preds.shape)
print("Accuracy = {:.2f}%".format((example_targets == preds).sum().item() * 100. /
↪ len(example_targets)))

show_data_label_prediction(data=example_data, y_true=example_targets, y_
↪pred=preds, shape=(3, 4))
```

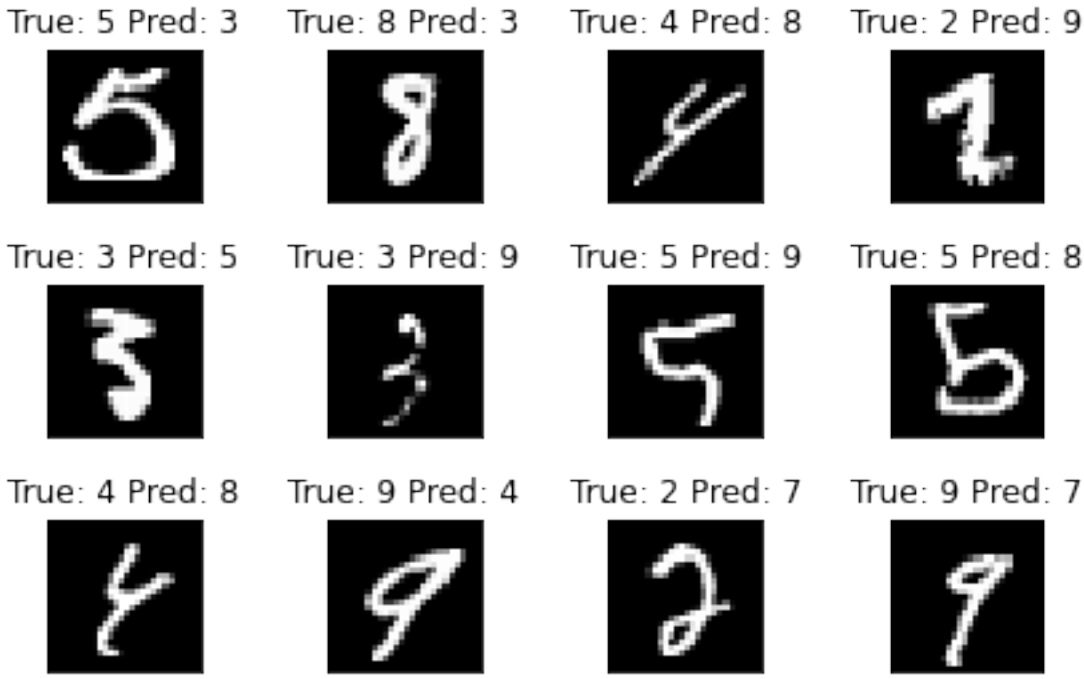
```
Output shape= torch.Size([10000, 10]) label shape= torch.Size([10000])
Accuracy = 91.21%
```



Plot missclassified samples

```
errors = example_targets != preds
#print(errors, np.where(errors))
print("Nb errors = {}, (Error rate = {:.2f}%)".format(errors.sum(), 100 * errors.
↪sum().item() / len(errors)))
err_idx = np.where(errors)[0]
show_data_label_prediction(data=example_data[err_idx], y_true=example_targets[err_
↪idx],
                           y_pred=preds[err_idx], shape=(3, 4))
```

```
Nb errors = 879, (Error rate = 8.79%)
```



Continue training from checkpoints: reload the model and run 10 more epochs

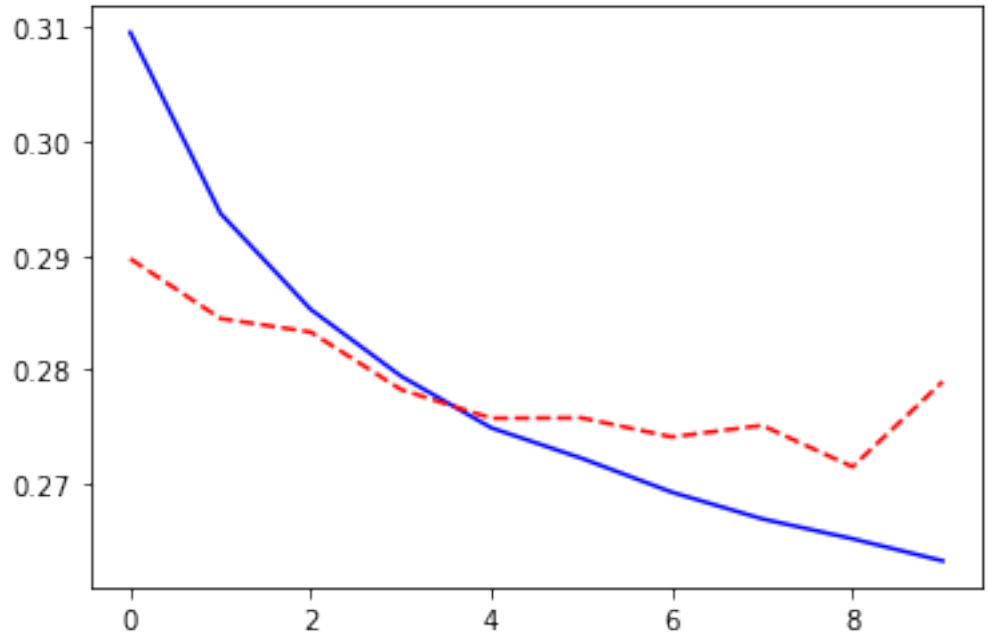
```
model = TwoLayerMLP(D_in, 50, D_out)
model.load_state_dict(torch.load('models/mod-%s.pth' % model.__class__.__name__))
model.to(device)

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=10, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/9
----------
train Loss: 0.3096 Acc: 91.11%
val Loss: 0.2897 Acc: 91.65%

Epoch 2/9
----------
train Loss: 0.2853 Acc: 92.03%
val Loss: 0.2833 Acc: 92.04%

Epoch 4/9
----------
train Loss: 0.2749 Acc: 92.36%
```

(continues on next page)

(continued from previous page)

```
val Loss: 0.2757 Acc: 92.01%

Epoch 6/9
----------
train Loss: 0.2692 Acc: 92.51%
val Loss: 0.2741 Acc: 92.29%

Epoch 8/9
----------
train Loss: 0.2651 Acc: 92.61%
val Loss: 0.2715 Acc: 92.32%

Training complete in 1m 14s
Best val Acc: 92.32%
```



6.2.5 Test several MLP architectures

- Define a `MultiLayerMLP([D_in, 512, 256, 128, 64, D_out])` class that take the size of the layers as parameters of the constructor.
- Add some non-linearity with relu acivation function

```
class MLP(nn.Module):

    def __init__(self, d_layer):
        super(MLP, self).__init__()
        self.d_layer = d_layer
        layer_list = [nn.Linear(d_layer[l], d_layer[l+1]) for l in range(len(d_
↪layer) - 1)]
        self.linears = nn.ModuleList(layer_list)
```

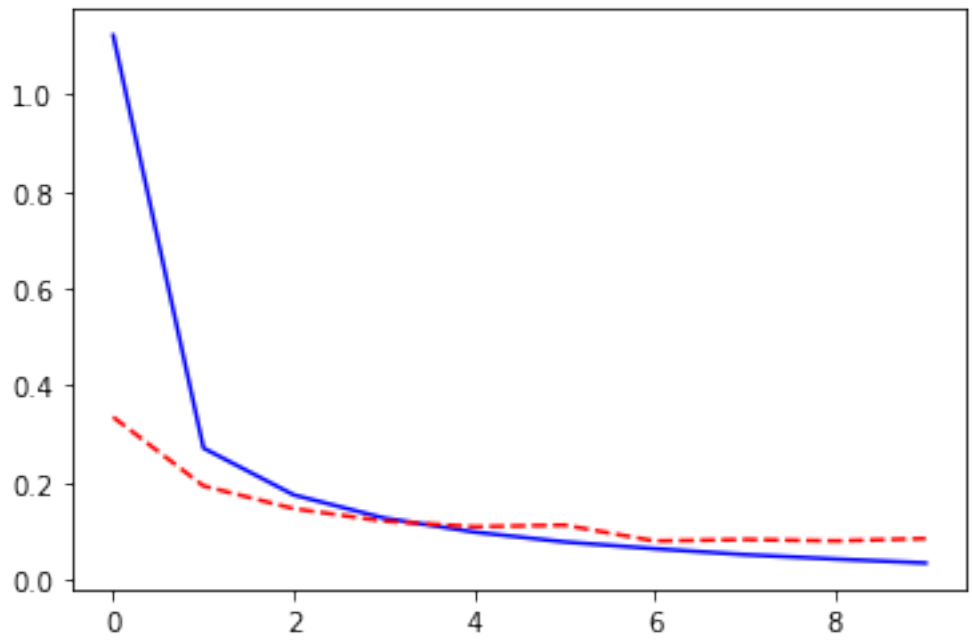
(continues on next page)

(continued from previous page)

```
    def forward(self, X):
        X = X.view(-1, self.d_layer[0])
        # relu(Wl x) for all hidden layer
        for layer in self.linears[:-1]:
            X = F.relu(layer(X))
        # softmax(Wl x) for output layer
        return F.log_softmax(self.linears[-1](X), dim=1)
```

```
model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=10, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/9
----------
train Loss: 1.1216 Acc: 66.19%
val Loss: 0.3347 Acc: 90.71%

Epoch 2/9
----------
train Loss: 0.1744 Acc: 94.94%
val Loss: 0.1461 Acc: 95.52%

Epoch 4/9
----------
train Loss: 0.0979 Acc: 97.14%
val Loss: 0.1089 Acc: 96.49%

Epoch 6/9
----------
train Loss: 0.0635 Acc: 98.16%
val Loss: 0.0795 Acc: 97.68%

Epoch 8/9
----------
train Loss: 0.0422 Acc: 98.77%
val Loss: 0.0796 Acc: 97.54%

Training complete in 1m 53s
Best val Acc: 97.68%
```



6.2.6 Reduce the size of training dataset

Reduce the size of the training dataset by considering only `10` minibatche for size`16`.

```
train_loader, val_loader = load_mnist(16, 1000)

train_size = 10 * 16

# Stratified sub-sampling
targets = train_loader.dataset.targets.numpy()
nclasses = len(set(targets))

indices = np.concatenate([np.random.choice(np.where(targets == lab)[0], int(train_
↪size / nclasses),replace=False)
    for lab in set(targets)])
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(train_loader.dataset, batch_size=16,
    sampler=torch.utils.data.SubsetRandomSampler(indices))

# Check train subsampling
train_labels = np.concatenate([labels.numpy() for inputs, labels in train_loader])
print("Train size=", len(train_labels), " Train label count=", {lab:np.sum(train_
↪labels == lab) for lab in set(train_labels)})
print("Batch sizes=", [inputs.size(0) for inputs, labels in train_loader])

# Put together train and val
dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
```

(continues on next page)

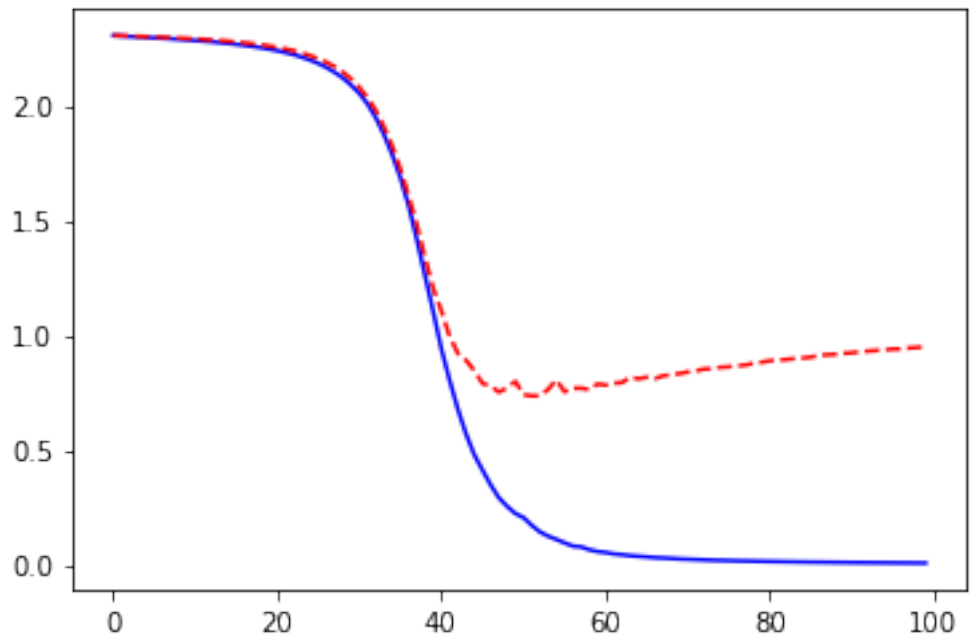
(continued from previous page)

```
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shape", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪'val']})
print("N input features", D_in, "N output", D_out)
```

```
Train size= 160  Train label count= {0: 16, 1: 16, 2: 16, 3: 16, 4: 16, 5: 16, 6:␣
↪16, 7: 16, 8: 16, 9: 16}
Batch sizes= [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
Datasets shape {'train': torch.Size([60000, 28, 28]), 'val': torch.Size([10000,␣
↪28, 28])}
N input features 784 N output 10
```

```
model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/99
----------
train Loss: 2.3050 Acc: 10.00%
val Loss: 2.3058 Acc: 8.92%

Epoch 20/99
----------
train Loss: 2.2389 Acc: 42.50%
val Loss: 2.2534 Acc: 29.90%

Epoch 40/99
----------
train Loss: 0.9381 Acc: 83.75%
val Loss: 1.1041 Acc: 68.36%

Epoch 60/99
----------
train Loss: 0.0533 Acc: 100.00%
val Loss: 0.7823 Acc: 76.69%

Epoch 80/99
----------
train Loss: 0.0138 Acc: 100.00%
val Loss: 0.8884 Acc: 76.88%

Training complete in 2m 17s
Best val Acc: 77.08%
```



Use an opimizer with an adaptative learning rate: Adam

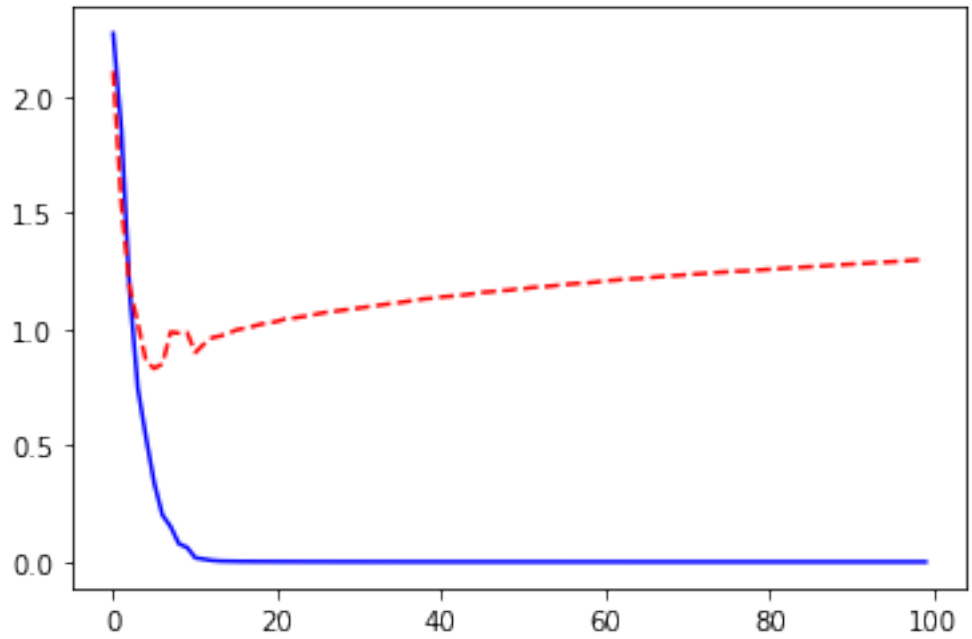
```
model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/99
----------
train Loss: 2.2706 Acc: 23.75%
val Loss: 2.1079 Acc: 44.98%

Epoch 20/99
----------
train Loss: 0.0012 Acc: 100.00%
val Loss: 1.0338 Acc: 78.23%

Epoch 40/99
----------
train Loss: 0.0003 Acc: 100.00%
val Loss: 1.1383 Acc: 78.24%

Epoch 60/99
----------
train Loss: 0.0002 Acc: 100.00%
val Loss: 1.2075 Acc: 78.17%
```

(continues on next page)

(continued from previous page)

```
Epoch 80/99
----------
train Loss: 0.0001 Acc: 100.00%
val Loss: 1.2571 Acc: 78.26%

Training complete in 2m 28s
Best val Acc: 78.35%
```



6.2.7 Run MLP on CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each: - airplane
- automobile
- bird
- cat
- deer
- dog
- frog
- horse

- ship
- truck

Load CIFAR-10 dataset

```
from pathlib import Path
WD = os.path.join(Path.home(), "data", "pystatml", "dl_cifar10_pytorch")
os.makedirs(WD, exist_ok=True)
os.chdir(WD)
print("Working dir is:", os.getcwd())
os.makedirs("data", exist_ok=True)
os.makedirs("models", exist_ok=True)

import numpy as np
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
num_epochs = 5
learning_rate = 0.001

# Image preprocessing modules
transform = transforms.Compose([
    transforms.Pad(4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()])

# CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='data/',
                                             train=True,
                                             transform=transform,
                                             download=True)

val_dataset = torchvision.datasets.CIFAR10(root='data/',
                                           train=False,
                                           transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(dataset=val_dataset,
```

(continues on next page)

(continued from previous page)

```
                                        batch_size=100,
                                        shuffle=False)

# Put together train and val
dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(set(dataloaders["train"].dataset.targets))
print("Datasets shape:", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪'val']})
print("N input features:", D_in, "N output:", D_out)
```

```
Working dir is: /home/ed203246/data/pystatml/dl_cifar10_pytorch
Files already downloaded and verified
Datasets shape: {'train': (50000, 32, 32, 3), 'val': (10000, 32, 32, 3)}
N input features: 3072 N output: 10
```

```
model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                      num_epochs=50, log_interval=10)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-36-13724f7cb709> in <module>
----> 1 model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
      2 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
      3 criterion = nn.NLLLoss()
      4
      5 model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,


~/anaconda3/lib/python3.7/site-packages/torch/nn/modules/module.py in to(self,␣
↪*args, **kwargs)
    424             return t.to(device, dtype if t.is_floating_point() else None,␣
↪non_blocking)
    425
--> 426         return self._apply(convert)
    427
```

(continues on next page)

(continued from previous page)

```
    428     def register_backward_hook(self, hook):


~/anaconda3/lib/python3.7/site-packages/torch/nn/modules/module.py in _apply(self,
↪ fn)
    200     def _apply(self, fn):
    201         for module in self.children():
--> 202             module._apply(fn)
    203
    204         def compute_should_use_set_data(tensor, tensor_applied):


~/anaconda3/lib/python3.7/site-packages/torch/nn/modules/module.py in _apply(self,
↪ fn)
    200     def _apply(self, fn):
    201         for module in self.children():
--> 202             module._apply(fn)
    203
    204         def compute_should_use_set_data(tensor, tensor_applied):


~/anaconda3/lib/python3.7/site-packages/torch/nn/modules/module.py in _apply(self,
↪ fn)
    222                 # `with torch.no_grad():`
    223                 with torch.no_grad():
--> 224                     param_applied = fn(param)
    225                 should_use_set_data = compute_should_use_set_data(param,␣
↪param_applied)
    226                 if should_use_set_data:


~/anaconda3/lib/python3.7/site-packages/torch/nn/modules/module.py in convert(t)
    422
    423         def convert(t):
--> 424             return t.to(device, dtype if t.is_floating_point() else None,␣
↪non_blocking)
    425
    426         return self._apply(convert)


RuntimeError: CUDA error: all CUDA-capable devices are busy or unavailable
```

6.3 Convolutional neural network

6.3.1 Outline

2. Architecures
3. Train and test functions
4. CNN models
5. MNIST
6. CIFAR-10

Sources:

Deep learning - cs231n.stanford.edu

CNN - Stanford cs231n

Pytorch - WWW tutorials - github tutorials - github examples

MNIST and pytorch: - MNIST nextjournal.com/gkoehler/pytorch-mnist - MNIST github/pytorch/examples - MNIST kaggle

6.3.2 Architectures

Sources:

- cv-tricks.com
- [zhenye-na.github.io(]https://zhenye-na.github.io/2018/12/01/cnn-deep-leearning-ai-week2.html)

LeNet

The first Convolutional Networks were developed by Yann LeCun in 1990's.

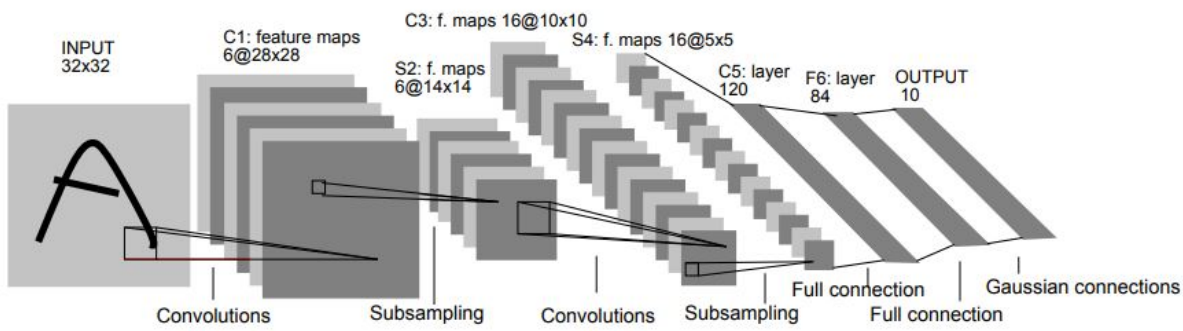


Fig. 1: LeNet

AlexNet

(2012, Alex Krizhevsky, Ilya Sutskever and Geoff Hinton)

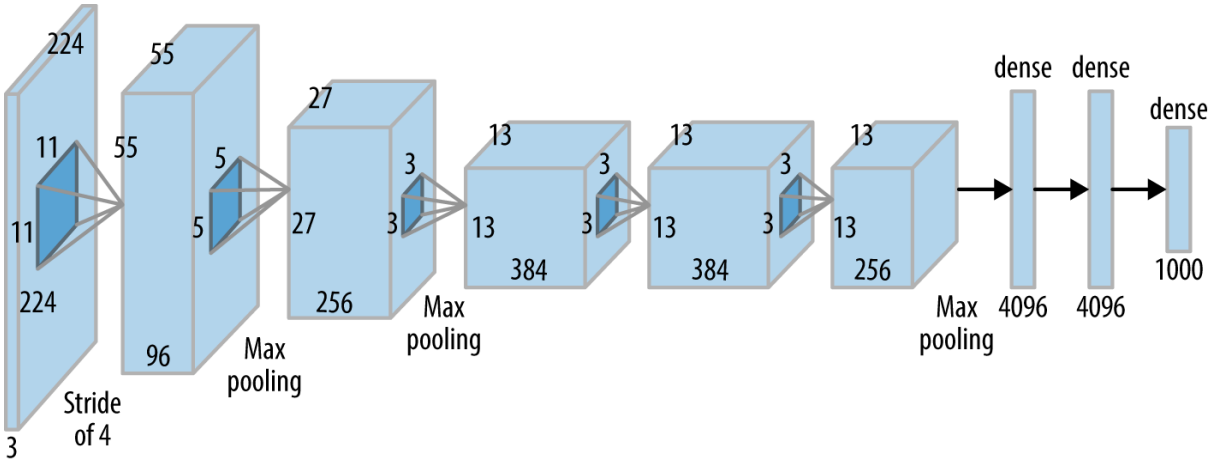


Fig. 2: AlexNet

AlexNet Network - Structural Details													
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
Total													**62,378,344**

Fig. 3: AlexNet architecture

- Deeper, bigger,
- Featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
- **ReLu(Rectified Linear Unit)** for the non-linear part, instead of a Tanh or Sigmoid.

The advantage of the ReLu over sigmoid is that it trains much faster than the latter because the derivative of sigmoid becomes very small in the saturating region and therefore the updates to the weights almost vanish. This is called **vanishing gradient problem**.

- **Dropout**: reduces the over-fitting by using a Dropout layer after every FC layer. Dropout layer has a probability,(p), associated with it and is applied at every neuron of the response map separately. It randomly switches off the activation with the probability p.

Why does DropOut work?

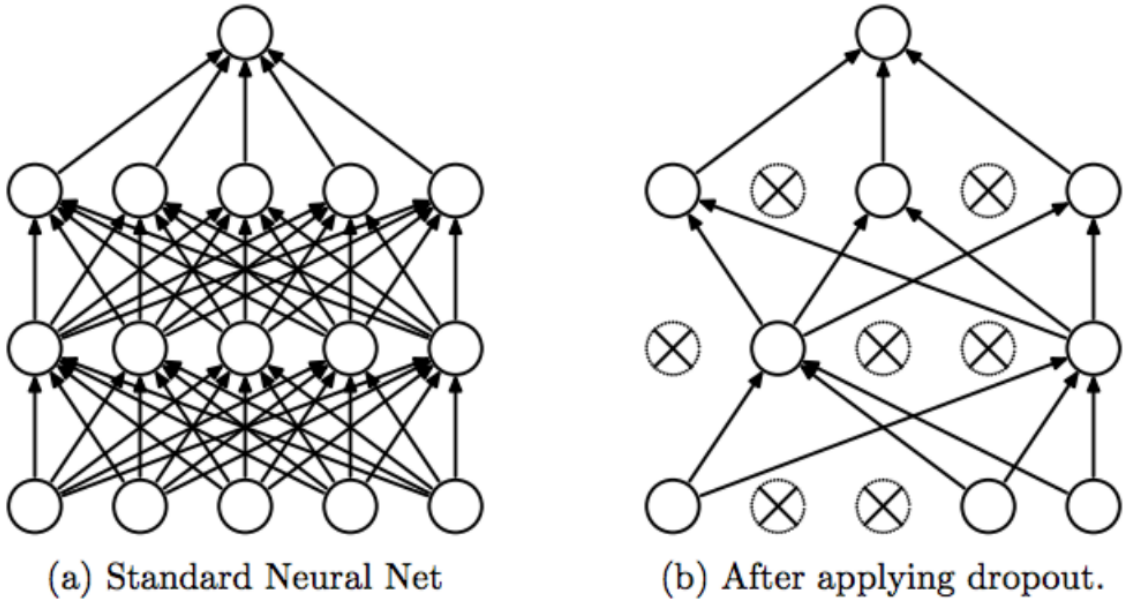


Fig. 4: Dropout

The idea behind the dropout is similar to the model ensembles. Due to the dropout layer, different sets of neurons which are switched off, represent a different architecture and all these different architectures are trained in parallel with weight given to each subset and the summation of weights being one. For n neurons attached to DropOut, the number of subset architectures formed is 2^n. So it amounts to prediction being averaged over these ensembles of models. This provides a structured model regularization which helps in avoiding the over-fitting. Another view of DropOut being helpful is that since neurons are randomly chosen, they tend to avoid developing co-adaptations among themselves thereby enabling them to develop meaningful features, independent of others.

- **Data augmentation** is carried out to reduce over-fitting. This Data augmentation includes mirroring and cropping the images to increase the variation in the training data-set.

GoogLeNet. (Szegedy et al. from Google 2014) was a Convolutional Network . Its main contribution was the development of an

- **Inception Module** that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M).
- There are also several followup versions to the GoogLeNet, most recently Inception-v4.

VGGNet. (Karen Simonyan and Andrew Zisserman 2014)

- 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture.
- Only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Replace large kernel-sized filters(11 and 5 in the first and second convolutional layer, respectively) with multiple 3X3 kernel-sized filters one after another.

With a given receptive field(the effective area size of input image on which output depends), multiple stacked smaller size kernel is better than the one with a larger size kernel because multiple non-linear layers increases the depth of the network which enables it to learn more complex features, and that too at a lower cost. For example, three 3X3 filters on top of each other with stride 1 ha a receptive size of 7, but the number of parameters involved is 3*(9^2) in comparison to 49^2 parameters of kernels with a size of 7.

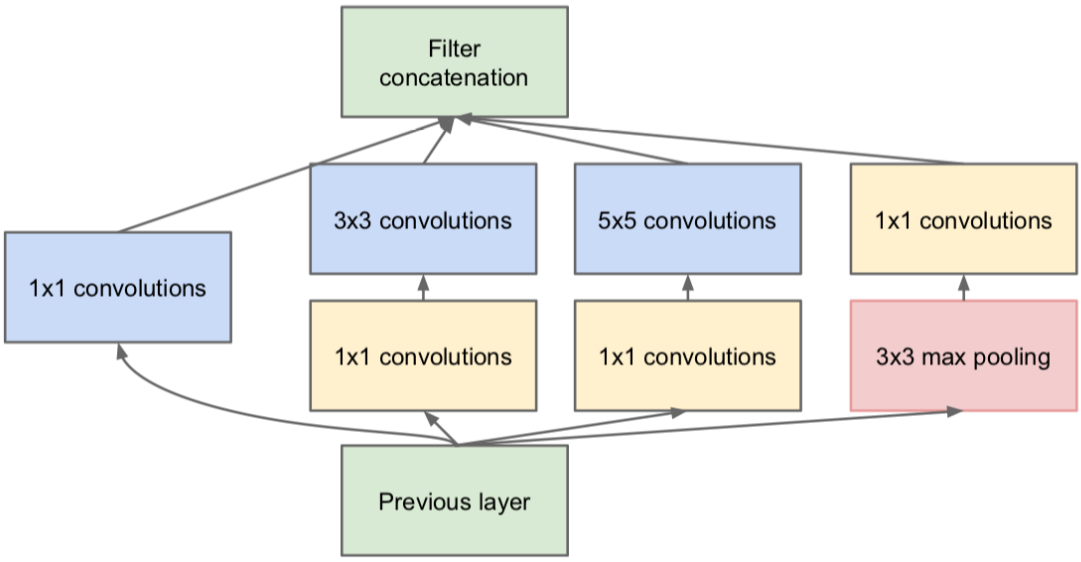


Fig. 5: Inception Module

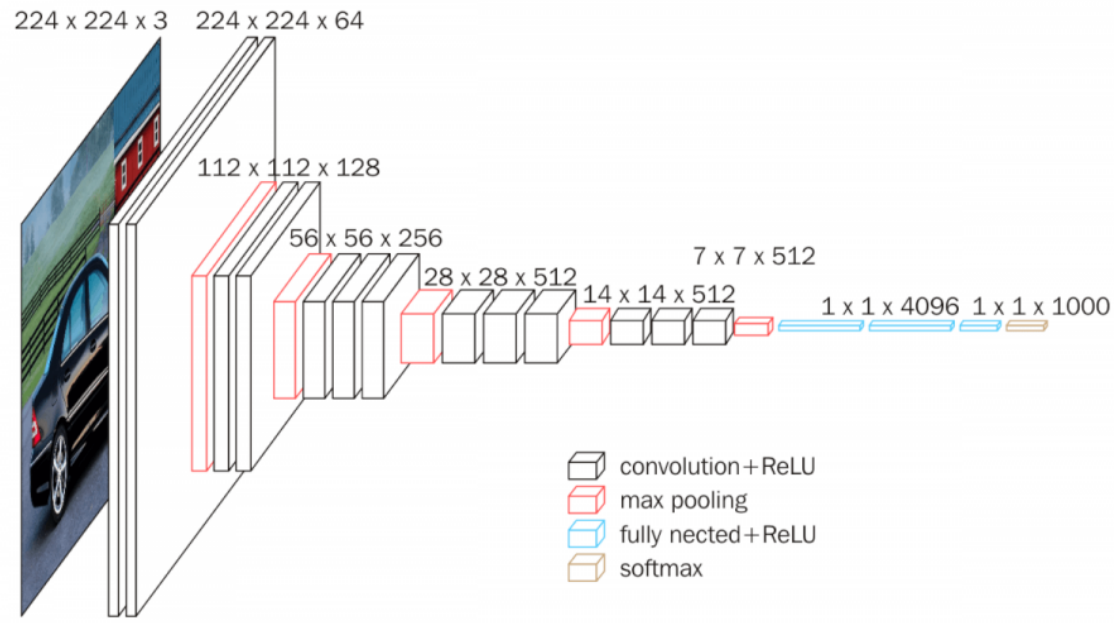


Fig. 6: VGGNet

VGG16 - Structural Details														
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													**138,423,208**

Fig. 7: VGGNet architecture

- Lot more memory and parameters (140M)

ResNet. (Kaiming He et al. 2015)

Resnet block variants (Source):

- Skip connections
- Batch normalization.
- State of the art CNN models and are the default choice (as of May 10, 2016). In particular, also see more
- Recent developments that tweak the original architecture from Kaiming He et al. Identity Mappings in Deep Residual Networks (published March 2016).

Models in pytorch

6.3.3 Architecures general guidelines

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures: stack 3x3, instead of 5x5
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like [(CONV-RELU) x N POOL?] x M (FC-RELU) x K, SOFTMAX where N is usually up to ~5, M is large, 0 <= K <= 2.
- but recent advances such as ResNet/GoogLeNet have challenged this paradigm

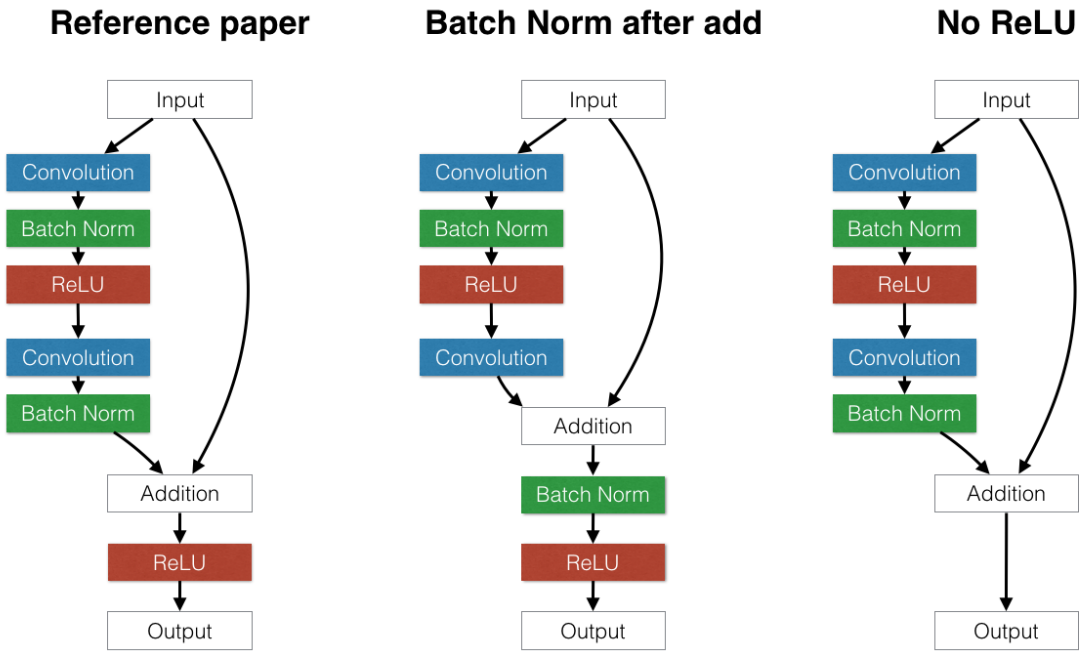


Fig. 8: ResNet block

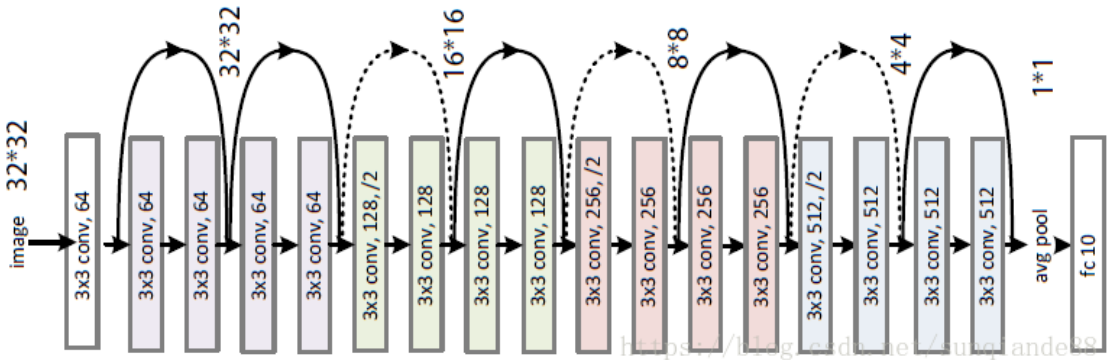


Fig. 9: ResNet 18

ResNet18 - Structural Details														
#	Input Image			output			Layer	Stride	Pad	Kernel		in	out	Param
1	227	227	3	112	112	64	conv1	2	1	7	7	3	64	9472
	112	112	64	56	56	64	maxpool	2	0.5	3	3	64	64	0
2	56	56	64	56	56	64	conv2-1	1	1	3	3	64	64	36928
3	56	56	64	56	56	64	conv2-2	1	1	3	3	64	64	36928
4	56	56	64	56	56	64	conv2-3	1	1	3	3	64	64	36928
5	56	56	64	56	56	64	conv2-4	1	1	3	3	64	64	36928
6	56	56	64	28	28	128	conv3-1	2	0.5	3	3	64	128	73856
7	28	28	128	28	28	128	conv3-2	1	1	3	3	128	128	147584
8	28	28	128	28	28	128	conv3-3	1	1	3	3	128	128	147584
9	28	28	128	28	28	128	conv3-4	1	1	3	3	128	128	147584
10	28	28	128	14	14	256	conv4-1	2	0.5	3	3	128	256	295168
11	14	14	256	14	14	256	conv4-2	1	1	3	3	256	256	590080
12	14	14	256	14	14	256	conv4-3	1	1	3	3	256	256	590080
13	14	14	256	14	14	256	conv4-4	1	1	3	3	256	256	590080
14	14	14	256	7	7	512	conv5-1	2	0.5	3	3	256	512	1180160
15	7	7	512	7	7	512	conv5-2	1	1	3	3	512	512	2359808
16	7	7	512	7	7	512	conv5-3	1	1	3	3	512	512	2359808
17	7	7	512	7	7	512	conv5-4	1	1	3	3	512	512	2359808
	7	7	512	1	1	512	avg pool	7	0	7	7	512	512	0
18	1	1	512	1	1	1000	fc					512	1000	513000
Total														11,511,784

Fig. 10: ResNet 18 architecture

6.3.4 Train function

```
%matplotlib inline

import os
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
import torchvision.transforms as transforms
from torchvision import models
#
from pathlib import Path
import matplotlib.pyplot as plt

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = 'cpu' # Force CPU
```

```
# %load train_val_model.py
import numpy as np
import torch
import time
import copy
```

(continues on next page)

(continued from previous page)

```
def train_val_model(model, criterion, optimizer, dataloaders, num_epochs=25,
        scheduler=None, log_interval=None):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    # Store losses and accuracies accross epochs
    losses, accuracies = dict(train=[], val=[]), dict(train=[], val=[])

    for epoch in range(num_epochs):
        if log_interval is not None and epoch % log_interval == 0:
            print('Epoch {}/{}'.format(epoch, num_epochs - 1))
            print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            nsamples = 0
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                nsamples += inputs.shape[0]

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
```

(continues on next page)

(continued from previous page)

```
                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if scheduler is not None and phase == 'train':
                scheduler.step()

            #nsamples = dataloaders[phase].dataset.data.shape[0]
            epoch_loss = running_loss / nsamples
            epoch_acc = running_corrects.double() / nsamples

            losses[phase].append(epoch_loss)
            accuracies[phase].append(epoch_acc)
            if log_interval is not None and epoch % log_interval == 0:
                print('{} Loss: {:.4f} Acc: {:.2f}%'.format(
                    phase, epoch_loss, 100 * epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
        if log_interval is not None and epoch % log_interval == 0:
            print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:.2f}%'.format(100 * best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)

    return model, losses, accuracies
```

6.3.5 CNN models

LeNet-5

Here we implement LeNet-5 with relu activation. Sources: (1), (2).

```
import torch.nn as nn
import torch.nn.functional as F

class LeNet5(nn.Module):
    """
    layers: (nb channels in input layer,
             nb channels in 1rst conv,
             nb channels in 2nd conv,
```

(continues on next page)

(continued from previous page)

```
            nb neurons for 1rst FC: TO BE TUNED,
            nb neurons for 2nd FC,
            nb neurons for 3rd FC,
            nb neurons output FC TO BE TUNED)
    """
    def __init__(self, layers = (1, 6, 16, 1024, 120, 84, 10), debug=False):
        super(LeNet5, self).__init__()
        self.layers = layers
        self.debug = debug
        self.conv1 = nn.Conv2d(layers[0], layers[1], 5, padding=2)
        self.conv2 = nn.Conv2d(layers[1], layers[2], 5)
        self.fc1   = nn.Linear(layers[3], layers[4])
        self.fc2   = nn.Linear(layers[4], layers[5])
        self.fc3   = nn.Linear(layers[5], layers[6])

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), 2) # same shape / 2
        x = F.max_pool2d(F.relu(self.conv2(x)), 2) # -4 / 2
        if self.debug:
            print("### DEBUG: Shape of last convnet=", x.shape[1:], ". FC size=",␣
↪np.prod(x.shape[1:]))
        x = x.view(-1, self.layers[3])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

VGGNet like: conv-relu blocks

```
# Defining the network (LeNet-5)
import torch.nn as nn
import torch.nn.functional as F

class MiniVGGNet(torch.nn.Module):

    def __init__(self, layers=(1, 16, 32, 1024, 120, 84, 10), debug=False):
        super(MiniVGGNet, self).__init__()
        self.layers = layers
        self.debug = debug

        # Conv block 1
        self.conv11 = nn.Conv2d(in_channels=layers[0], out_channels=layers[1],␣
↪kernel_size=3,
                                stride=1, padding=0, bias=True)
        self.conv12 = nn.Conv2d(in_channels=layers[1], out_channels=layers[1],␣
↪kernel_size=3,
                                stride=1, padding=0, bias=True)
```

(continues on next page)

(continued from previous page)

```
        # Conv block 2
        self.conv21 = nn.Conv2d(in_channels=layers[1], out_channels=layers[2],␣
↪kernel_size=3,
                                stride=1, padding=0, bias=True)
        self.conv22 = nn.Conv2d(in_channels=layers[2], out_channels=layers[2],␣
↪kernel_size=3,
                                stride=1, padding=1, bias=True)

        # Fully connected layer
        self.fc1   = nn.Linear(layers[3], layers[4])
        self.fc2   = nn.Linear(layers[4], layers[5])
        self.fc3   = nn.Linear(layers[5], layers[6])

    def forward(self, x):
        x = F.relu(self.conv11(x))
        x = F.relu(self.conv12(x))
        x = F.max_pool2d(x, 2)

        x = F.relu(self.conv21(x))
        x = F.relu(self.conv22(x))
        x = F.max_pool2d(x, 2)

        if self.debug:
            print("### DEBUG: Shape of last convnet=", x.shape[1:], ". FC size=",␣
↪np.prod(x.shape[1:]))
        x = x.view(-1, self.layers[3])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return F.log_softmax(x, dim=1)
```

ResNet-like Model:

Stack multiple resnet blocks

```
# ---------------------------------------------------------------------------- #
# An implementation of https://arxiv.org/pdf/1512.03385.pdf                    #
# See section 4.2 for the model architecture on CIFAR-10                       #
# Some part of the code was referenced from below                              #
# https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py   #
# ---------------------------------------------------------------------------- #
import torch.nn as nn

# 3x3 convolution
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)
```

(continues on next page)

(continued from previous page)

```
# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1], 2)
        self.layer3 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
```

(continues on next page)

(continued from previous page)

```
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return F.log_softmax(out, dim=1)
        #return out
```

ResNet9

- DAWNBench on cifar10
- ResNet9: train to 94% CIFAR10 accuracy in 100 seconds

6.3.6 MNIST digit classification

```
from pathlib import Path
from torchvision import datasets, transforms
import os

WD = os.path.join(Path.home(), "data", "pystatml", "dl_mnist_pytorch")
os.makedirs(WD, exist_ok=True)
os.chdir(WD)
print("Working dir is:", os.getcwd())
os.makedirs("data", exist_ok=True)
os.makedirs("models", exist_ok=True)


def load_mnist(batch_size_train, batch_size_test):

    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=batch_size_train, shuffle=True)

    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('data', train=False, transform=transforms.Compose([
            transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```
            transforms.Normalize((0.1307,), (0.3081,))
        ])),
        batch_size=batch_size_test, shuffle=True)
    return train_loader, test_loader

train_loader, val_loader = load_mnist(64, 1000)

dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
data_shape = dataloaders["train"].dataset.data.shape[1:]
D_in = np.prod(data_shape)
D_out = len(dataloaders["train"].dataset.targets)
print("Datasets shape", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪'val']})
print("N input features", D_in, "N output", D_out)
```

```
Working dir is: /home/ed203246/data/pystatml/dl_mnist_pytorch
Datasets shape {'train': torch.Size([60000, 28, 28]), 'val': torch.Size([10000,␣
↪28, 28])}
N input features 784 N output 60000
```

LeNet

Dry run in debug mode to get the shape of the last convnet layer.

```
model = LeNet5((1, 6, 16, 1, 120, 84, 10), debug=True)
batch_idx, (data_example, target_example) = next(enumerate(train_loader))
print(model)
_ = model(data_example)
```

```
LeNet5(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([16, 5, 5]) . FC size= 400
```

Set First FC layer to 400

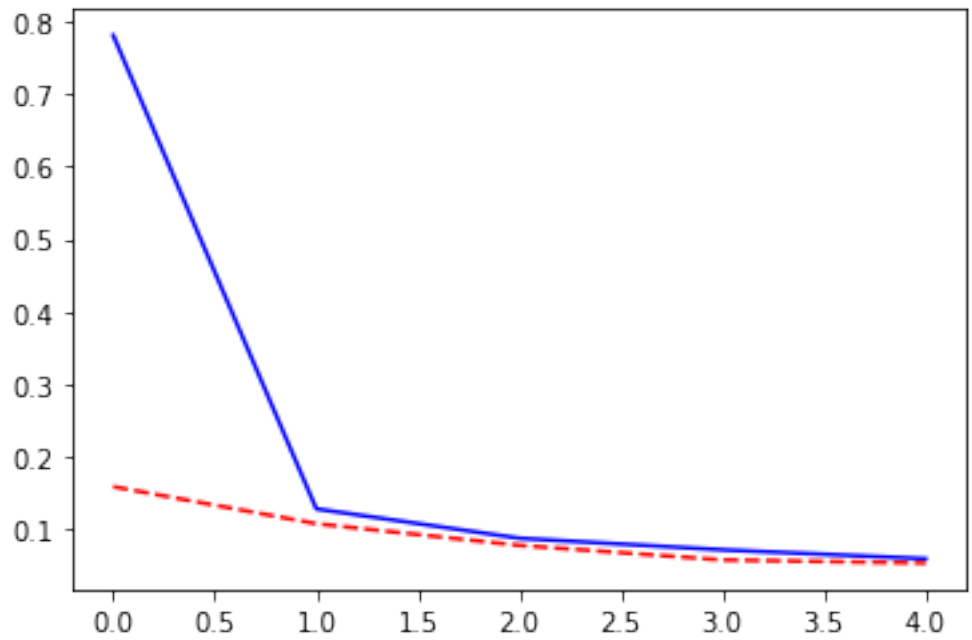
```
model = LeNet5((1, 6, 16, 400, 120, 84, 10)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
```

(continues on next page)

(continued from previous page)

```
    print(parameter.shape)

print("Total number of parameters =", np.sum([np.prod(parameter.shape) for␣
↪parameter in model.parameters()]))

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=5, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
torch.Size([6, 1, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 400])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters = 61706
Epoch 0/4
----------
train Loss: 0.7807 Acc: 75.65%
val Loss: 0.1586 Acc: 94.96%

Epoch 2/4
----------
train Loss: 0.0875 Acc: 97.33%
val Loss: 0.0776 Acc: 97.47%

Epoch 4/4
----------
train Loss: 0.0592 Acc: 98.16%
val Loss: 0.0533 Acc: 98.30%

Training complete in 1m 29s
Best val Acc: 98.30%
```



MiniVGGNet

```
model = MiniVGGNet(layers=(1, 16, 32, 1, 120, 84, 10), debug=True)

print(model)
_ = model(data_example)
```

```
MiniVGGNet(
  (conv11): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv21): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv22): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([32, 5, 5]) . FC size= 800
```

Set First FC layer to 800

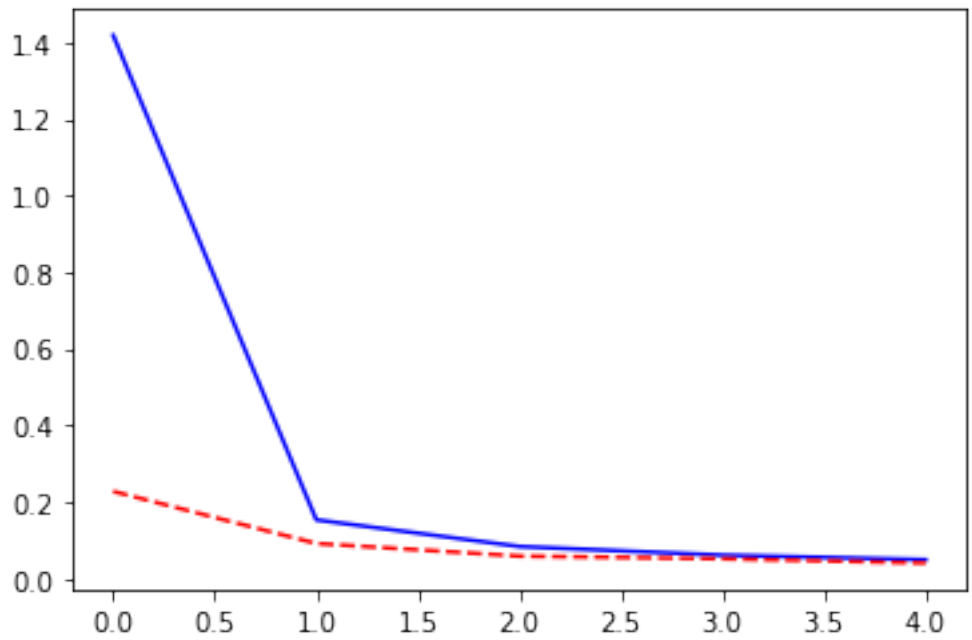
```
model = MiniVGGNet((1, 16, 32, 800, 120, 84, 10)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =", np.sum([np.prod(parameter.shape) for␣
↪parameter in model.parameters()]))
```

(continues on next page)

(continued from previous page)

```
model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=5, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
torch.Size([16, 1, 3, 3])
torch.Size([16])
torch.Size([16, 16, 3, 3])
torch.Size([16])
torch.Size([32, 16, 3, 3])
torch.Size([32])
torch.Size([32, 32, 3, 3])
torch.Size([32])
torch.Size([120, 800])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters = 123502
Epoch 0/4
----------
train Loss: 1.4180 Acc: 48.27%
val Loss: 0.2277 Acc: 92.68%

Epoch 2/4
----------
train Loss: 0.0838 Acc: 97.41%
val Loss: 0.0587 Acc: 98.14%

Epoch 4/4
----------
train Loss: 0.0495 Acc: 98.43%
val Loss: 0.0407 Acc: 98.63%

Training complete in 3m 10s
Best val Acc: 98.63%
```



Reduce the size of training dataset

Reduce the size of the training dataset by considering only `10` minibatche for size`16`.

```
train_loader, val_loader = load_mnist(16, 1000)

train_size = 10 * 16

# Stratified sub-sampling
targets = train_loader.dataset.targets.numpy()
nclasses = len(set(targets))

indices = np.concatenate([np.random.choice(np.where(targets == lab)[0], int(train_
↪size / nclasses),replace=False)
    for lab in set(targets)])
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(train_loader.dataset, batch_size=16,
    sampler=torch.utils.data.SubsetRandomSampler(indices))

# Check train subsampling
train_labels = np.concatenate([labels.numpy() for inputs, labels in train_loader])
print("Train size=", len(train_labels), " Train label count=", {lab:np.sum(train_
↪labels == lab) for lab in set(train_labels)})
print("Batch sizes=", [inputs.size(0) for inputs, labels in train_loader])

# Put together train and val
dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
data_shape = dataloaders["train"].dataset.data.shape[1:]
```

(continues on next page)

(continued from previous page)

```
D_in = np.prod(data_shape)
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shape", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪'val']})
print("N input features", D_in, "N output", D_out)
```

```
Train size= 160  Train label count= {0: 16, 1: 16, 2: 16, 3: 16, 4: 16, 5: 16, 6:␣
↪16, 7: 16, 8: 16, 9: 16}
Batch sizes= [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
Datasets shape {'train': torch.Size([60000, 28, 28]), 'val': torch.Size([10000,␣
↪28, 28])}
N input features 784 N output 10
```

LeNet5

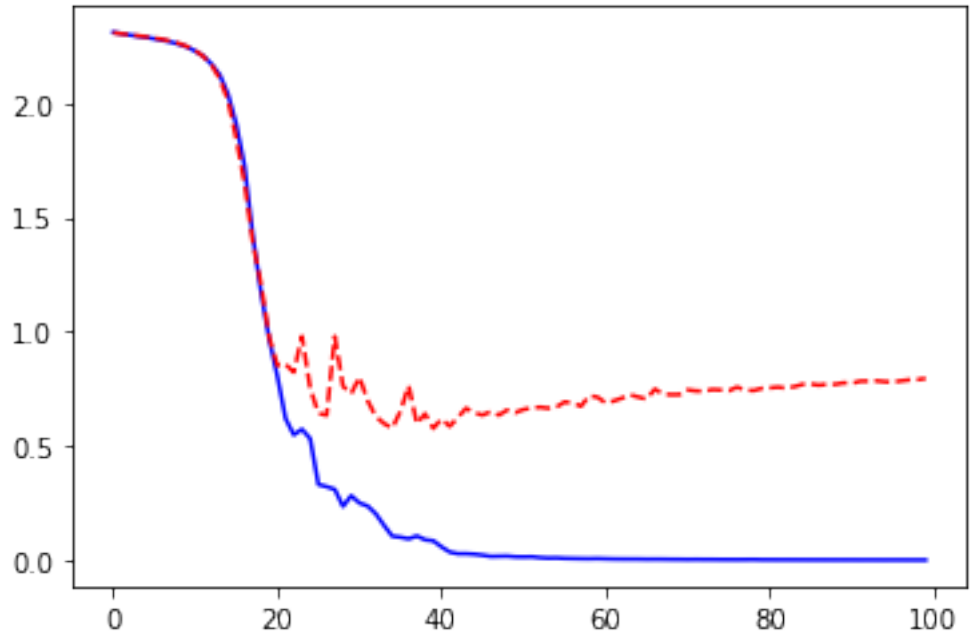
```
model = LeNet5((1, 6, 16, 400, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/99
----------
train Loss: 2.3086 Acc: 11.88%
val Loss: 2.3068 Acc: 14.12%

Epoch 20/99
----------
train Loss: 0.8060 Acc: 76.25%
val Loss: 0.8522 Acc: 72.84%

Epoch 40/99
----------
train Loss: 0.0596 Acc: 99.38%
val Loss: 0.6188 Acc: 82.67%

Epoch 60/99
----------
train Loss: 0.0072 Acc: 100.00%
val Loss: 0.6888 Acc: 83.08%

Epoch 80/99
----------
train Loss: 0.0033 Acc: 100.00%
val Loss: 0.7546 Acc: 82.96%
```

(continues on next page)

(continued from previous page)

```
Training complete in 3m 10s
Best val Acc: 83.46%
```



MiniVGGNet

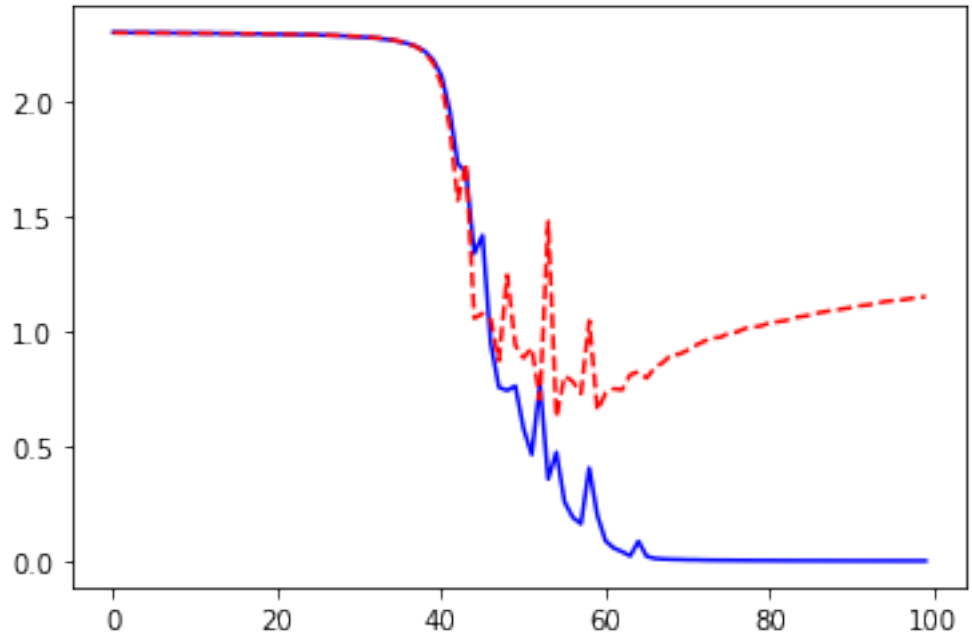
```
model = MiniVGGNet((1, 16, 32, 800, 120, 84, 10)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/99
----------
train Loss: 2.3040 Acc: 10.00%
val Loss: 2.3025 Acc: 10.32%

Epoch 20/99
----------
train Loss: 2.2963 Acc: 10.00%
val Loss: 2.2969 Acc: 10.35%

Epoch 40/99
----------
train Loss: 2.1158 Acc: 37.50%
val Loss: 2.0764 Acc: 38.06%
```

(continues on next page)

(continued from previous page)

```
Epoch 60/99
----------
train Loss: 0.0875 Acc: 97.50%
val Loss: 0.7315 Acc: 80.50%

Epoch 80/99
----------
train Loss: 0.0023 Acc: 100.00%
val Loss: 1.0397 Acc: 81.69%

Training complete in 5m 38s
Best val Acc: 82.02%
```



6.3.7 CIFAR-10 dataset

Source Yunjey Choi

```
from pathlib import Path
WD = os.path.join(Path.home(), "data", "pystatml", "dl_cifar10_pytorch")
os.makedirs(WD, exist_ok=True)
os.chdir(WD)
print("Working dir is:", os.getcwd())
os.makedirs("data", exist_ok=True)
os.makedirs("models", exist_ok=True)

import numpy as np
import torch
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```
import torchvision
import torchvision.transforms as transforms


# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
num_epochs = 5
learning_rate = 0.001

# Image preprocessing modules
transform = transforms.Compose([
    transforms.Pad(4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()])

# CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='data/',
                                             train=True,
                                             transform=transform,
                                             download=True)

val_dataset = torchvision.datasets.CIFAR10(root='data/',
                                           train=False,
                                           transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(dataset=val_dataset,
                                         batch_size=100,
                                         shuffle=False)

# Put together train and val
dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
data_shape = dataloaders["train"].dataset.data.shape[1:]
D_in = np.prod(data_shape)
D_out = len(set(dataloaders["train"].dataset.targets))
print("Datasets shape:", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪'val']})
print("N input features:", D_in, "N output:", D_out)
```

```
Working dir is: /home/ed203246/data/pystatml/dl_cifar10_pytorch
```

(continues on next page)

(continued from previous page)

```
Files already downloaded and verified
Datasets shape: {'train': (50000, 32, 32, 3), 'val': (10000, 32, 32, 3)}
N input features: 3072 N output: 10
```

LeNet

```
model = LeNet5((3, 6, 16, 1, 120, 84, D_out), debug=True)
batch_idx, (data_example, target_example) = next(enumerate(train_loader))
print(model)
_ = model(data_example)
```

```
LeNet5(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([16, 6, 6]) . FC size= 576
```

Set 576 neurons to the first FC layer

SGD with momentum `lr=0.001, momentum=0.5`

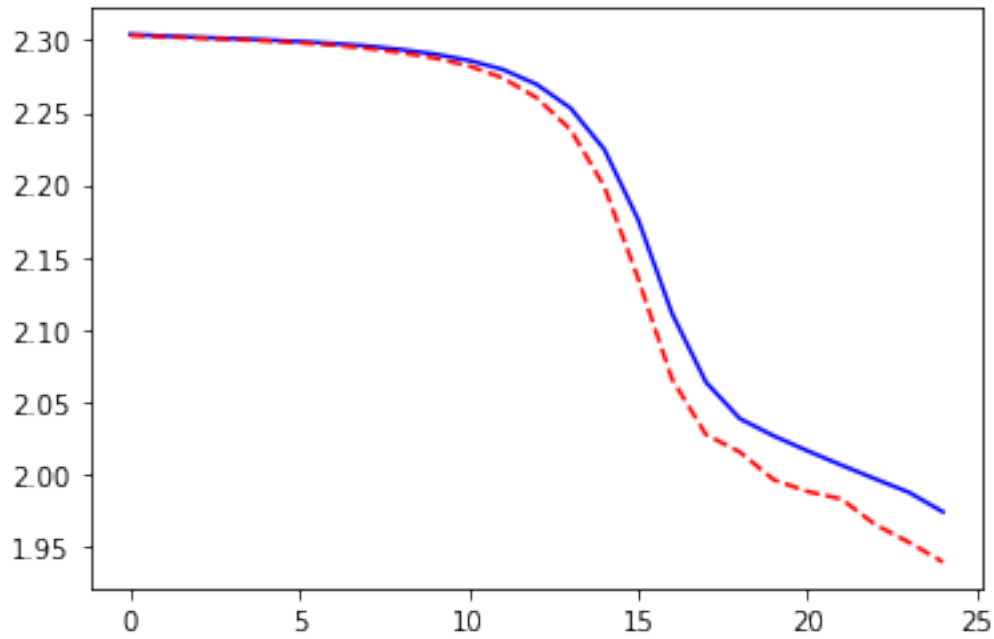
```
model = LeNet5((3, 6, 16, 576, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =", np.sum([np.prod(parameter.shape) for␣
↪parameter in model.parameters()]))

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
torch.Size([6, 3, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 576])
torch.Size([120])
torch.Size([84, 120])
```

(continues on next page)

(continued from previous page)

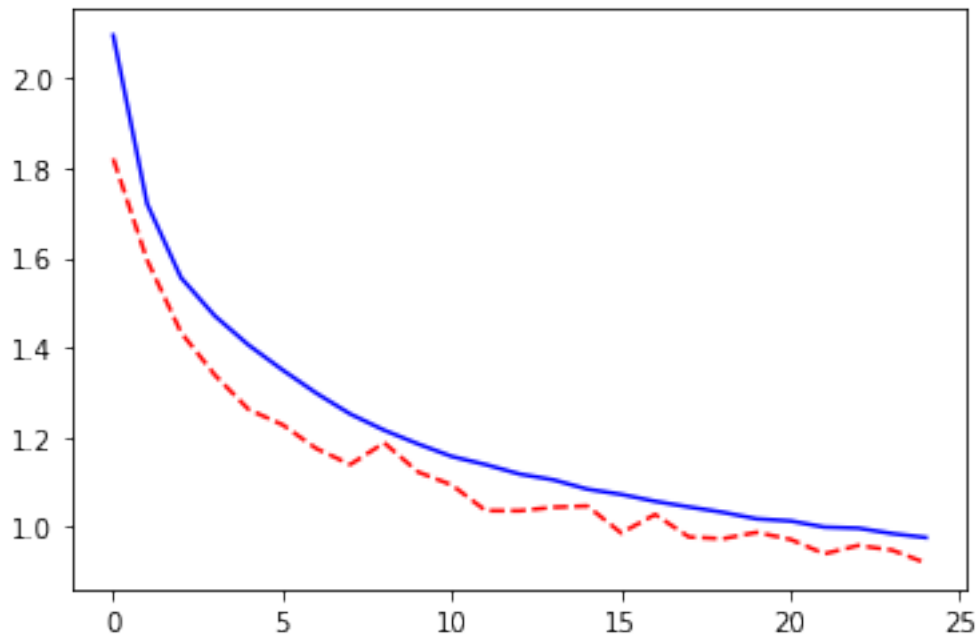
```
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters = 83126
Epoch 0/24
----------
train Loss: 2.3041 Acc: 10.00%
val Loss: 2.3033 Acc: 10.00%

Epoch 5/24
----------
train Loss: 2.2991 Acc: 11.18%
val Loss: 2.2983 Acc: 11.00%

Epoch 10/24
----------
train Loss: 2.2860 Acc: 10.36%
val Loss: 2.2823 Acc: 10.60%

Epoch 15/24
----------
train Loss: 2.1759 Acc: 18.83%
val Loss: 2.1351 Acc: 20.74%

Epoch 20/24
----------
train Loss: 2.0159 Acc: 25.35%
val Loss: 1.9878 Acc: 26.90%

Training complete in 7m 26s
Best val Acc: 28.98%
```



Increase learning rate and momentum `lr=0.01, momentum=0.9`

```
model = LeNet5((3, 6, 16, 576, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 2.0963 Acc: 21.65%
val Loss: 1.8211 Acc: 33.49%

Epoch 5/24
----------
train Loss: 1.3500 Acc: 51.34%
val Loss: 1.2278 Acc: 56.40%

Epoch 10/24
----------
train Loss: 1.1569 Acc: 58.79%
val Loss: 1.0933 Acc: 60.95%

Epoch 15/24
----------
train Loss: 1.0724 Acc: 62.12%
val Loss: 0.9863 Acc: 65.34%

Epoch 20/24
----------
train Loss: 1.0131 Acc: 64.41%
val Loss: 0.9720 Acc: 66.14%

Training complete in 7m 17s
Best val Acc: 67.87%
```



Adaptative learning rate: Adam

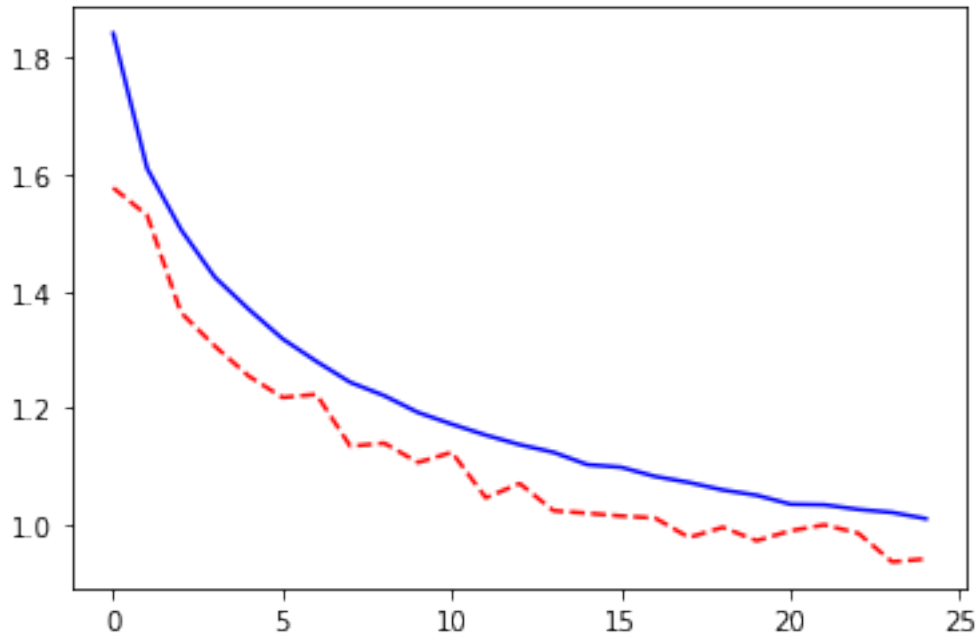
```
model = LeNet5((3, 6, 16, 576, 120, 84, D_out)).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 1.8411 Acc: 30.21%
val Loss: 1.5768 Acc: 41.22%

Epoch 5/24
----------
train Loss: 1.3185 Acc: 52.17%
val Loss: 1.2181 Acc: 55.71%

Epoch 10/24
----------
train Loss: 1.1724 Acc: 57.89%
val Loss: 1.1244 Acc: 59.17%

Epoch 15/24
----------
train Loss: 1.0987 Acc: 60.98%
val Loss: 1.0153 Acc: 63.82%
```

(continues on next page)

(continued from previous page)

```
Epoch 20/24
----------
train Loss: 1.0355 Acc: 63.01%
val Loss: 0.9901 Acc: 64.90%

Training complete in 7m 30s
Best val Acc: 66.88%
```



MiniVGGNet

```
model = MiniVGGNet(layers=(3, 16, 32, 1, 120, 84, D_out), debug=True)
print(model)
_ = model(data_example)
```

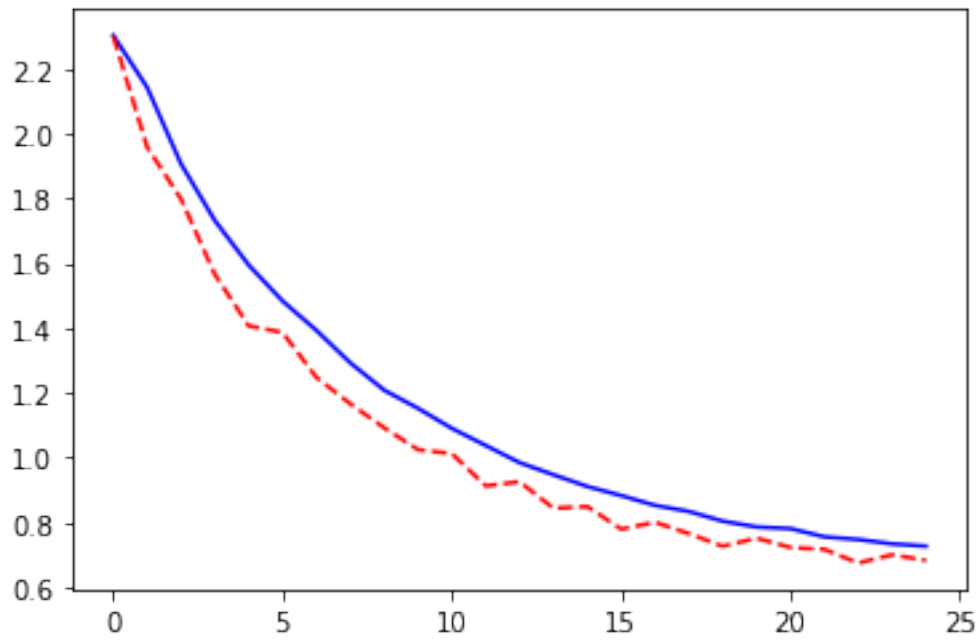
```
MiniVGGNet(
  (conv11): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv21): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv22): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([32, 6, 6]) . FC size= 1152
```

Set 1152 neurons to the first FC layer

SGD with large momentum and learning rate

```
model = MiniVGGNet((3, 16, 32, 1152, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 2.3027 Acc: 10.14%
val Loss: 2.3010 Acc: 10.00%

Epoch 5/24
----------
train Loss: 1.4829 Acc: 46.08%
val Loss: 1.3860 Acc: 50.39%

Epoch 10/24
----------
train Loss: 1.0899 Acc: 61.43%
val Loss: 1.0121 Acc: 64.59%

Epoch 15/24
----------
train Loss: 0.8825 Acc: 69.02%
val Loss: 0.7788 Acc: 72.73%

Epoch 20/24
----------
train Loss: 0.7805 Acc: 72.73%
val Loss: 0.7222 Acc: 74.72%

Training complete in 15m 19s
Best val Acc: 76.62%
```



Adam

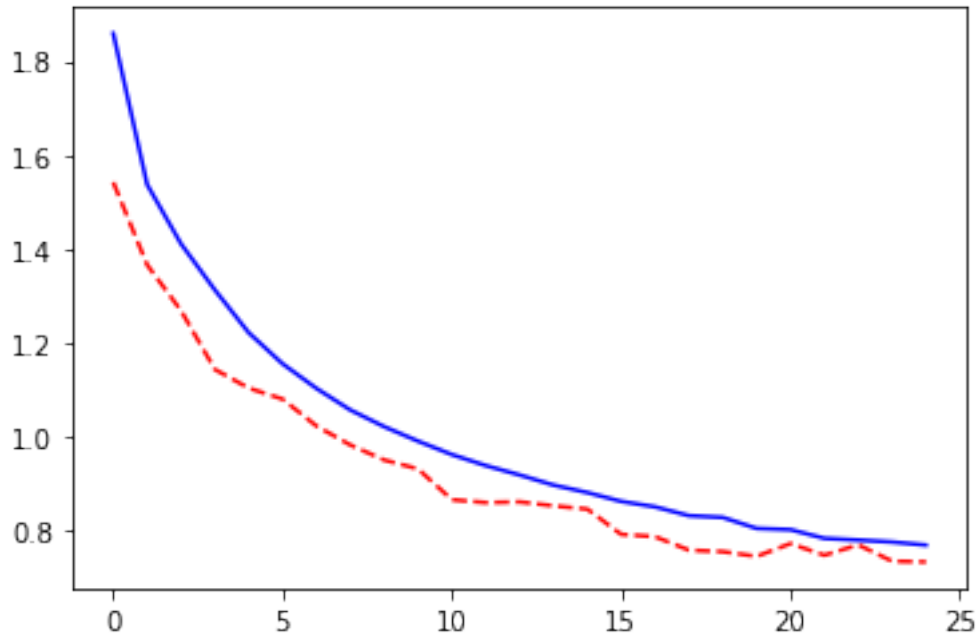
```
model = MiniVGGNet((3, 16, 32, 1152, 120, 84, D_out)).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 1.8591 Acc: 30.74%
val Loss: 1.5424 Acc: 43.46%

Epoch 5/24
----------
train Loss: 1.1562 Acc: 58.46%
val Loss: 1.0811 Acc: 61.87%

Epoch 10/24
----------
train Loss: 0.9630 Acc: 65.69%
val Loss: 0.8669 Acc: 68.94%

Epoch 15/24
----------
train Loss: 0.8634 Acc: 69.38%
val Loss: 0.7933 Acc: 72.33%
```

(continues on next page)

(continued from previous page)

```
Epoch 20/24
----------
train Loss: 0.8033 Acc: 71.75%
val Loss: 0.7737 Acc: 73.57%

Training complete in 15m 37s
Best val Acc: 74.86%
```



ResNet

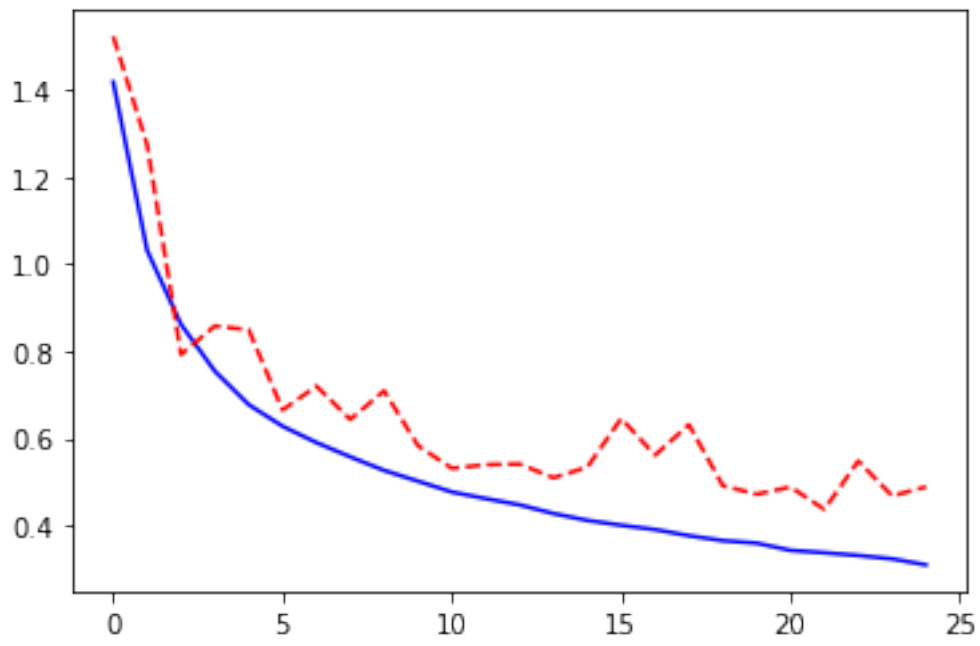
```
model = ResNet(ResidualBlock, [2, 2, 2], num_classes=D_out).to(device) # 195738␣
↪parameters
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,␣
↪dataloaders,
                        num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 1.4169 Acc: 48.11%
val Loss: 1.5213 Acc: 48.08%

Epoch 5/24
----------
```

(continues on next page)

(continued from previous page)

```
train Loss: 0.6279 Acc: 78.09%
val Loss: 0.6652 Acc: 77.49%

Epoch 10/24
----------
train Loss: 0.4772 Acc: 83.57%
val Loss: 0.5314 Acc: 82.09%

Epoch 15/24
----------
train Loss: 0.4010 Acc: 86.09%
val Loss: 0.6457 Acc: 79.03%

Epoch 20/24
----------
train Loss: 0.3435 Acc: 88.07%
val Loss: 0.4887 Acc: 84.34%

Training complete in 103m 30s
Best val Acc: 85.66%
```



6.4 Transfer Learning Tutorial

Sources:

- cs231n @ Stanford
- Sasank Chilamkurthy

Quote cs231n @ Stanford:

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

These two major transfer learning scenarios look as follows:

- **ConvNet as fixed feature extractor**:
  - Take a ConvNet pretrained on ImageNet,
  - Remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet)
  - Treat the rest of the ConvNet as a fixed feature extractor for the new dataset.

  In practice:
  - Freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.
- **Finetuning the convnet**:

fine-tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet

Instead of random initializaion, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.

```
%matplotlib inline

import os
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
import torchvision.transforms as transforms
from torchvision import models
#
from pathlib import Path
import matplotlib.pyplot as plt

# Device configuration
```

(continues on next page)

(continued from previous page)

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = 'cpu' # Force CPU
```

6.4.1 Training function

Combine train and test/validation into a single function.

Now, let's write a general function to train a model. Here, we will illustrate:

- Scheduling the learning rate
- Saving the best model

In the following, parameter `scheduler` is an LR scheduler object from `torch.optim.lr_scheduler`.

```
# %load train_val_model.py
import numpy as np
import torch
import time
import copy


def train_val_model(model, criterion, optimizer, dataloaders, num_epochs=25,
        scheduler=None, log_interval=None):

    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    # Store losses and accuracies accross epochs
    losses, accuracies = dict(train=[], val=[]), dict(train=[], val=[])

    for epoch in range(num_epochs):
        if log_interval is not None and epoch % log_interval == 0:
            print('Epoch {}/{}'.format(epoch, num_epochs - 1))
            print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
```

(continues on next page)

(continued from previous page)

```
            nsamples = 0
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                nsamples += inputs.shape[0]

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if scheduler is not None and phase == 'train':
                scheduler.step()

            #nsamples = dataloaders[phase].dataset.data.shape[0]
            epoch_loss = running_loss / nsamples
            epoch_acc = running_corrects.double() / nsamples

            losses[phase].append(epoch_loss)
            accuracies[phase].append(epoch_acc)
            if log_interval is not None and epoch % log_interval == 0:
                print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                    phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
        if log_interval is not None and epoch % log_interval == 0:
            print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))
```

(continues on next page)

(continued from previous page)

```
    # load best model weights
    model.load_state_dict(best_model_wts)

    return model, losses, accuracies
```

6.4.2 CIFAR-10 dataset

Source Yunjey Choi

```
WD = os.path.join(Path.home(), "data", "pystatml", "dl_cifar10_pytorch")
os.makedirs(WD, exist_ok=True)
os.chdir(WD)
print("Working dir is:", os.getcwd())
os.makedirs("data", exist_ok=True)
os.makedirs("models", exist_ok=True)

# Image preprocessing modules
transform = transforms.Compose([
    transforms.Pad(4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()])

# CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='data/',
                                             train=True,
                                             transform=transform,
                                             download=True)

test_dataset = torchvision.datasets.CIFAR10(root='data/',
                                            train=False,
                                            transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                         batch_size=100,
                                         shuffle=False)

# Put together train and val
dataloaders = dict(train=train_loader, val=val_loader)

# Info about the dataset
data_shape = dataloaders["train"].dataset.data.shape[1:]
```

(continues on next page)

(continued from previous page)

```
D_in = np.prod(data_shape)
D_out = len(set(dataloaders["train"].dataset.targets))
print("Datasets shape", {x: dataloaders[x].dataset.data.shape for x in ['train',
↪'val']})
print("N input features", D_in, "N output", D_out)
```

```
Working dir is: /home/ed203246/data/pystatml/dl_cifar10_pytorch
Files already downloaded and verified
Datasets shape {'train': (50000, 32, 32, 3), 'val': (10000, 32, 32, 3)}
N input features 3072 N output 10
```

Finetuning the convnet

- Load a pretrained model and reset final fully connected layer.
- SGD optimizer.

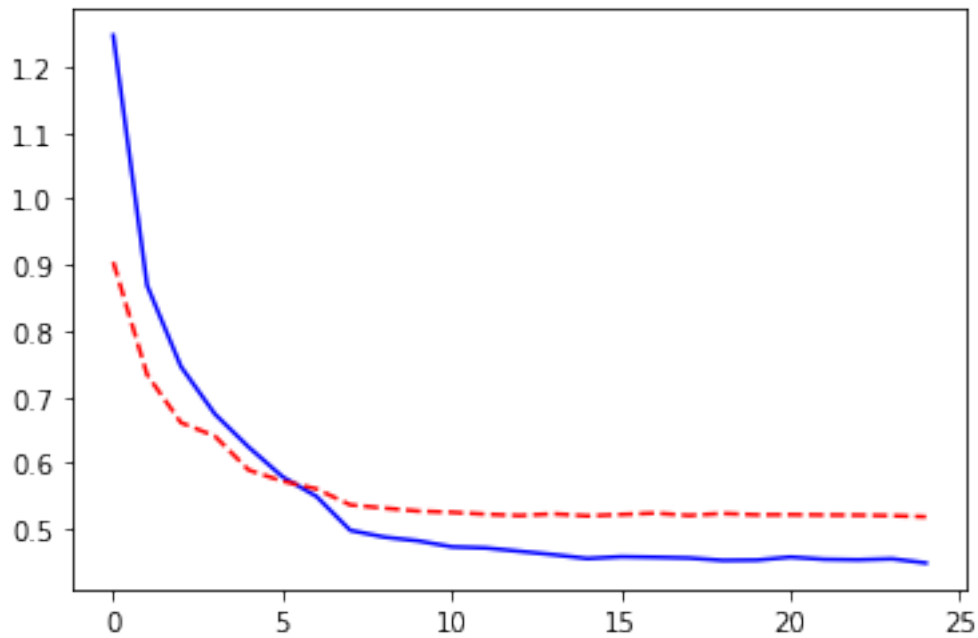
```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 10.
model_ft.fc = nn.Linear(num_ftrs, D_out)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

model, losses, accuracies = train_val_model(model_ft, criterion, optimizer_ft,
    dataloaders, scheduler=exp_lr_scheduler, num_epochs=25, log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 1.2476 Acc: 0.5593
val Loss: 0.9043 Acc: 0.6818

Epoch 5/24
----------
train Loss: 0.5791 Acc: 0.7978
val Loss: 0.5725 Acc: 0.8035
```

(continues on next page)

(continued from previous page)

```
Epoch 10/24
----------
train Loss: 0.4731 Acc: 0.8351
val Loss: 0.5254 Acc: 0.8217

Epoch 15/24
----------
train Loss: 0.4581 Acc: 0.8388
val Loss: 0.5220 Acc: 0.8226

Epoch 20/24
----------
train Loss: 0.4575 Acc: 0.8394
val Loss: 0.5218 Acc: 0.8236

Training complete in 138m 32s
Best val Acc: 0.825100
```



Adam optimizer

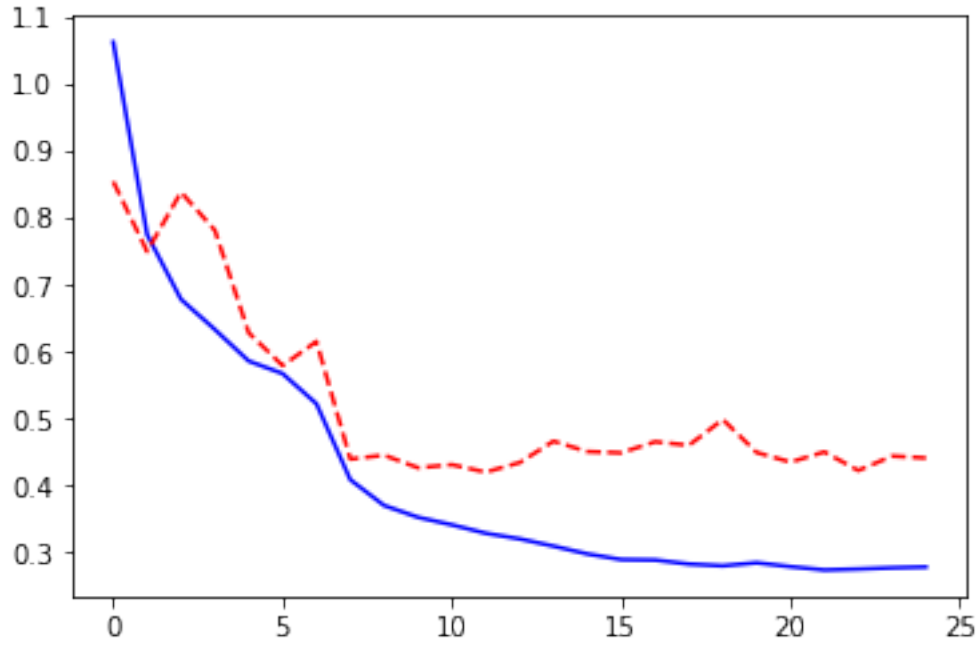
```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 10.
model_ft.fc = nn.Linear(num_ftrs, D_out)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = torch.optim.Adam(model_ft.parameters(), lr=0.001)
```

(continues on next page)

(continued from previous page)

```
# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

model, losses, accuracies = train_val_model(model_ft, criterion, optimizer_ft,
    dataloaders, scheduler=exp_lr_scheduler, num_epochs=25, log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['val'], '--r')
```

```
Epoch 0/24
----------
train Loss: 1.0622 Acc: 0.6341
val Loss: 0.8539 Acc: 0.7066

Epoch 5/24
----------
train Loss: 0.5674 Acc: 0.8073
val Loss: 0.5792 Acc: 0.8019

Epoch 10/24
----------
train Loss: 0.3416 Acc: 0.8803
val Loss: 0.4313 Acc: 0.8577

Epoch 15/24
----------
train Loss: 0.2898 Acc: 0.8980
val Loss: 0.4491 Acc: 0.8608

Epoch 20/24
----------
train Loss: 0.2792 Acc: 0.9014
val Loss: 0.4352 Acc: 0.8631

Training complete in 147m 23s
Best val Acc: 0.863800
```



ResNet as a feature extractor

Freeze all the network except the final layer: `requires_grad == False` to freeze the parameters so that the gradients are not computed in `backward()`.

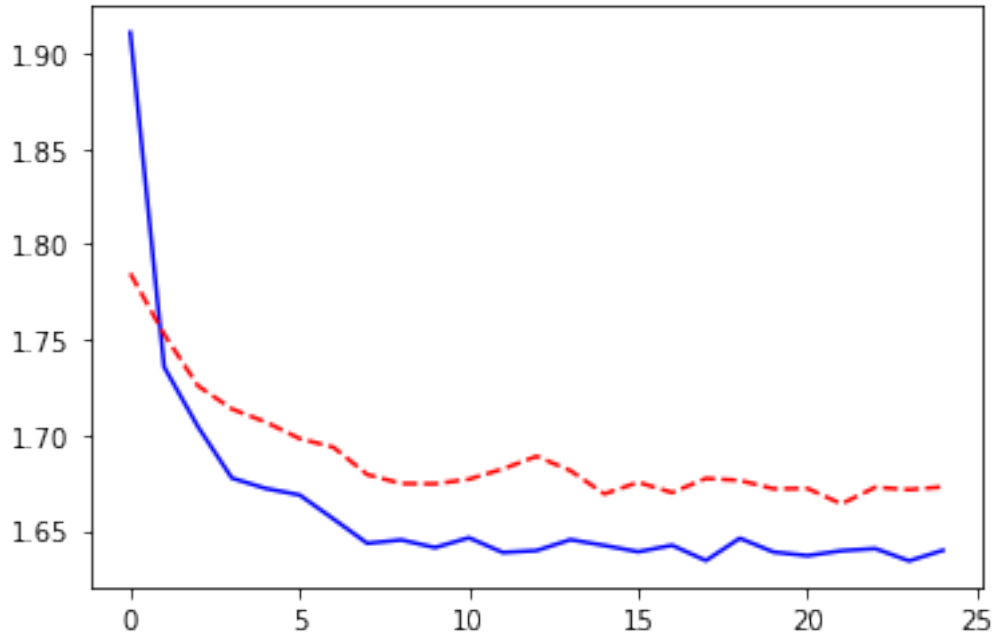
```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, D_out)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
model, losses, accuracies = train_val_model(model_conv, criterion, optimizer_conv,
    dataloaders, scheduler=exp_lr_scheduler, num_epochs=25, log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['val'], '--r')
```

```
Epoch 0/24
```

(continues on next page)

(continued from previous page)

```
----------
train Loss: 1.9108 Acc: 0.3277
val Loss: 1.7846 Acc: 0.3804

Epoch 5/24
----------
train Loss: 1.6686 Acc: 0.4170
val Loss: 1.6981 Acc: 0.4146

Epoch 10/24
----------
train Loss: 1.6462 Acc: 0.4267
val Loss: 1.6768 Acc: 0.4210

Epoch 15/24
----------
train Loss: 1.6388 Acc: 0.4296
val Loss: 1.6752 Acc: 0.4226

Epoch 20/24
----------
train Loss: 1.6368 Acc: 0.4325
val Loss: 1.6720 Acc: 0.4240

Training complete in 42m 23s
Best val Acc: 0.429600
```



Adam optimizer

```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
```

(continues on next page)

(continued from previous page)

```
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, D_out)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.Adam(model_conv.fc.parameters(), lr=0.001)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

model, losses, accuracies = train_val_model(model_conv, criterion, optimizer_conv,
    exp_lr_scheduler, dataloaders, num_epochs=25, log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['val'], '--r')
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-7-dde92868b554> in <module>
     19
     20 model, losses, accuracies = train_val_model(model_conv, criterion,␣
↪optimizer_conv,
---> 21     exp_lr_scheduler, dataloaders, num_epochs=25, log_interval=5)
     22
     23 epochs = np.arange(len(losses['train']))


TypeError: train_val_model() got multiple values for argument 'num_epochs'
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search