JKernelMachines: A Simple Framework for Kernel Machines
David Picard, Nicolas Thome, Matthieu Cord

To cite this version:

HAL Id: hal-00832030
https://hal.science/hal-00832030
Submitted on 10 Jun 2013

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.
**Abstract**

*JKernelMachines* is a Java library for learning with kernels. It is primarily designed to deal with custom kernels that are not easily found in standard libraries, such as kernels on structured data. These types of kernels are often used in computer vision or bioinformatics applications. We provide several kernels leading to state of the art classification performances in computer vision, as well as various kernels on sets. The main focus of the library is to be easily extended with new kernels. Standard SVM optimization algorithms are available, but also more sophisticated learning-based kernel combination methods such as Multiple Kernel Learning (MKL), and a recently published algorithm to learn powered products of similarities (Product Kernel Learning).

**Keywords:** classification, support vector machines, kernel, computer vision

1. Introduction

Support Vector Machines (SVM) are widely recognized as state-of-the-art classifiers for supervised learning problems. When using SVM, the similarity measure has to be a *kernel* function. Efficient libraries already exist implementing SVM optimization algorithms (liblinear, libsvm, weka, SHOGUN, etc). However, these solutions are generally limited to vectorial data and simple kernels (linear or Gaussian for example). It is not straightforward to use them when dealing with structured data, *e.g.*, sets, graphs, or strings. Designing similarities for such inputs is essential in areas such as bioinformatic and computer vision.

For image classification or object detection tasks, state of the art methods rely on similarity functions based on sets of features (edge, color, texture, *etc*) extracted locally on different image regions. Then, two main strategies can be used to design a similarity between sets of local descriptors, referred as "Bag of Features" (BoF). First, one can rely on the "Bag of Words" (BoW) model, where the local descriptors are quantized using a dictionary, and the projected features are aggregated to produce a histogram of visual words. An alternative to the BoW model is to directly define a kernel function between sets of local descriptors, thus ignoring the quantization step. The feature extraction and projection can also be included in the kernel function (as a first explicit mapping), which then becomes a similarity measure of high complexity. Recently, many kernels on sets have
been proposed in the computer vision community, for example by Bo and Sminchisescu (2009) or by T. Tuytelaars and Darrell (2011).

To sum it up, designing well adapted kernel functions is attracting a lot of research in the community. \textit{JKernelMachines} is dedicated to facilitate the use of such exotic kernels. It is thus not designed as an end user program, but as a library to be used in computer vision (or bioinformatics, \etc) pipelines. The goals of the library are to provide:

- An efficient framework for kernels:
  - Easy and intuitive use of the library
  - Very easy and rapid development of new kernels
- An effective SVM machine learning toolbox:
  - Up to date learning algorithms (primal, dual) with kernel combinations (MKL)
  - Good computational performances on current computer vision benchmarks

To fulfill these goals, \textit{JKernelMachines} is designed as a pure Java implementation without any further dependency than a standard JRE. It is licensed under the terms of GPLv3.

2. Description of the Library

The backbone of the library is the definition of data types and kernels. In order to use any type of input space, the library makes heavy use of the Java Generics. Kernels are defined on generic input space, with specific implementation delegated to child classes. Classifiers that use a kernel function will end up with the right similarity thanks to polymorphism.

2.1 Kernels

Given a sample of generic type \( T \), the library provides a parent class for all kernel functions:

```java
public abstract class Kernel<T> implements Serializable {
    /** compute the kernel similarity between two elements of input space */
    public abstract double valueOf(T t1, T t2);
}
```

To create a new kernel, one basically has to extend this class and implement the \texttt{valueOf()} method. For example, the source code below corresponds to the following kernel on bags of vectors of double:

\[
K(\{x_i\}_{i}, \{x_j\}_{j}) = \sum_{x_i \in \{x_i\}, x_j \in \{x_j\}} e^{-\frac{||x_i-x_j||^2}{2}}.
\]

```java
public class DoubleBagGaussian extends Kernel<List<double[]>> {
    public double valueOf(List<double[]> l1, List<double[]> l2) {
        double sum = 0.0;
        for (double[] t1 : l1)
            for (double[] t2 : l2) {
                for (double d = 0.0; int i = 0; i < t1.length; i++)
                    d += (t2[i] - t1[i]) * (t2[i] - t1[i]);
                sum += Math.exp(-d / 2.0);
            }
        return sum;
    }
}
```
Currently, JKernelMachines provides the following kernels:

- Standard kernels on vector (Linear, Gaussian, etc) for various types (int, double, etc)
- A wide variety of Gaussian kernels ($\chi^2$ distance, subset of selected axes, etc)
- Combination of kernels (weighted sum and weighted product of generic minor kernels)
- Generic kernels on lists (ordered or unordered, weighted or not)
- Kernels defined by a custom Gram matrix and kernels with various caching strategies

2.2 Classifiers

Classifiers are also defined on generic data types, and use an instance of a specific Kernel. In order to be able to work on any type of input space, the data type of the samples are encapsulated in a generic class called TrainingSample. This class contains the sample of generic type T and the associated label (if the sample is labeled):

```java
public class TrainingSample<T> implements Serializable {
    T sample;
    int label;
}
```

Training of an SVM classifier consists of calling the method `train` with a list of `TrainingSample` as parameters. For example, an SVM with GaussianKernel on vectors of double would be done as follows:

```java
List<TrainingSample<double[]>> train = new ArrayList<TrainingSample<double[]>>();
// add code to feed the list with labeled samples
DoubleGaussL2 k = new DoubleGaussL2();
LaSVM<double[]> lasvm = new LaSVM<double[]>(k);
lasvm.train(train);
```

In this example, the evaluation of a new sample is as simple as:

```java
double[] sample; // add code to fill the sample
double value = lasvm.valueOf(sample);
```

Currently, JKernelMachines implements the following algorithms:

- SVM: LaSVM from Bordes et al. (2005), LaSVM-I from Ertekin et al. (2011), SMO from Platt (1999), Transductive SVM using S3VMLight from Joachims (1999)
- Density estimator: One-class SVM using SMO, Parzen window estimator
- Multiple kernel: SimpleMKL from Rakotomamonjy et al. (2008), TS-MKL from Kumar et al. (2012), Gradient-descent based $\ell_p$-Norm MKL from Kloft et al. (2011), Product combinations of Gaussian kernels from Picard et al. (2012)
- Fast primal linear SVM on double vectors: Pegasos from Singer and Srebro (2007), SGD from Bottou (2010), SGDQN from Bordes et al. (2009)

Since most of these algorithms are implemented using the Kernel interface, they can be used with any positive semi-definite implementation of the Kernel class. Examples are provided with the library (see the example package).
Table 1: Mean accuracies for Weka and JKernelMachines on UCI data sets.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Weka SMO</th>
<th>LaSVM</th>
<th>JKernelMachines SMO</th>
<th>LaSVM-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>ionosphere</td>
<td>86.1 ± 4.1</td>
<td>92.8 ± 3.6</td>
<td>91.5 ± 2.5</td>
<td>90.3 ± 3.2</td>
</tr>
<tr>
<td>heart</td>
<td>84.0 ± 3.8</td>
<td>81.9 ± 3.8</td>
<td>83.1 ± 4.3</td>
<td>84.4 ± 5.0</td>
</tr>
<tr>
<td>breast-cancer</td>
<td>97.3 ± 1.3</td>
<td>97.1 ± 1.0</td>
<td>94.2 ± 1.8</td>
<td>97.4 ± 1.5</td>
</tr>
<tr>
<td>german numbers</td>
<td>75.4 ± 2.3</td>
<td>75.0 ± 3.2</td>
<td>68.9 ± 2.9</td>
<td>75.4 ± 2.1</td>
</tr>
</tbody>
</table>

3. Validation

We first present results comparing JKernelMachines to the well known Weka library on common data sets from the UCI repository. We used the example class CrossValidationExample of package fr.lip6.jkernelmachines.example to produce the results. Following the setup of Rakotomamonjy et al. (2008), 20 random splits were drawn (80% for train, 20% for test).

Table 3 summaries the results. On the ionosphere data set, JKernelMachines offers significantly better results than Weka. On heart, breast cancer and german numbers both libraries gave about the same results. Regarding time, we found JKernelMachines implementation of LaSVM to be significantly faster than Weka (both libraries were fast enough).

Compared to other libraries more centered on kernels, like SHOGUN or OpenKernel, JKernelMachines clearly aims at adding new kernels easily and reliably by allowing the user to set any code inside the kernel computation. For instance, SHOGUN allows to use kernels defined by a custom Gram matrix, which has to be computed before the optimization. The custom matrix method may not be easy to handle when the kernel is relying on a parametric feature extraction process, where the parameters are to be varied (e.g., filter banks in signal processing applications). Moreover, in case of online evaluation, a new custom matrix has to be computed for each incoming test sample and manually set as the kernel used by the classifier, whereas a JKernelMachines classifier would work without any modification. OpenKernel allows for more complex kernels using base kernels (including n-gram kernels) and combination, but again the data have to be in a specific file format generated in advance which leads to the same difficulties.

JKernelMachines was used for PASCAL VOC challenges both in 2009 and in 2010\(^1\), and gave comparative results to other teams using the same features. The achievement in these challenges is two-folds: first, it validates the implementation of the learning algorithms. Second, it shows the library is usable on real world challenges regarding computing resources. Some of these experiments were published in Picard et al. (2010).

References


\(^1\) Results can be found at http://pascallin.ecs.soton.ac.uk/challenges/VOC/voc2009/.


