PUNCC: a Python Library for Predictive Uncertainty Calibration and Conformalization

Mouhcine Mendil, Luca Mossina, David Vigouroux

PUNCC: a Python Library for Predictive Uncertainty Calibration and Conformalization

Mouhcine Mendil MOUHCINE.MENDIL@IRT-SAINTEXUPERY.COM
IRT Saint Exupéry, Toulouse, France
Luca Mossina LUCA.MOSSINA@IRT-SAINTEXUPERY.COM
IRT Saint Exupéry, Toulouse, France
David Vigouroux DAVID.VIGOUROUX@IRT-SAINTEXUPERY.COM
IRT Saint Exupéry, Toulouse, France

Editor: Harris Papadopoulos, Khuong An Nguyen, Henrik Boström and Lars Carlsson

Abstract

Predictive UNcertainty Calibration and Conformalization (PUNCC) is an open-source Python library integrating a collection of state-of-the-art *Conformal Prediction* (CP) algorithms and related techniques for regression and classification problems. This package aims to make conformal procedures accessible to non-experts using a simple and intuitive implementation. It is compatible with scikit-learn, PyTorch and TensorFlow and easily extensible to other prediction toolkits. PUNCC also comes with a low-level API that provides a unified workflow in a pythonic environment to build, combine and run inductive CP algorithms. It offers generic structures and consistent interfaces to design customized nonconformity scores, data partition schemes, and methods for constructing prediction sets. In this paper, we present the design of our library and demonstrate its use with various CP procedures, *Machine Learning* (ML) problems and models from different ML libraries. Source code, documentation and demos are available at https://github.com/deel-ai/puncc.

Keywords: Uncertainty Quantification, Conformal Prediction, Python, Library, Regression, Classification, Deep Learning.

1. Introduction

Conformal Prediction (CP) is a *distribution-free*, *model-agnostic* and *non-asymptotic* framework to estimate the predictive uncertainty of ML models by constructing confidence sets that include the true output with high probability. Such sets are theoretically valid, *i.e.* with guaranteed probability of marginal coverage, and desirably efficient, *i.e.* as small as possible (the set of all values of the output space is guaranteed to cover but is not informative). This can be a valuable tool particularly when deploying ML models in critical systems and moving towards their certification (Mamalet et al., 2021; Luo et al., 2023). For example, in a self-driving car, CP could be used to identify situations where the model is uncertain about the road conditions, allowing the car to slow down or take other precautionary measures to avoid accidents.

More formally, for a sequence[1] $\{(X_i, Y_i)\}_{i=1}^{n} \cup \{(X_{new}, Y_{new})\} \in (\mathcal{X} \times \mathcal{Y})^{n+1}$ of exchangeable (or more simply i.i.d) data and an error rate (or significance level) $\alpha \in (0, 1)$ set by the user, a CP procedure builds the prediction set $C_\alpha(X_{new})$ such that:

1. $D_{\text{cal}} = \{(X_i, Y_i)\}_{i=1}^{n}$ are the calibration data and (X_{new}, Y_{new}) is a test point.

$$\mathbb{P}\Big\{Y_{new} \in C_\alpha\left(X_{new}\right)\Big\} \geq 1-\alpha. \tag{1}$$

Over several data sequences, $C_\alpha(X_{new})$ will contain the true output Y_{new} with frequency *at least* $(1-\alpha)$. For example, the prediction intervals $C(X)$ obtained in Figure 1 for a regression task (detailed in Section 3) are centered on the point predictions $\widehat{f}(X)$ and have a constant width $2 \cdot \delta_\alpha$: $C(X) = [\widehat{f}(X) - \delta_\alpha, \widehat{f}(X) + \delta_\alpha]$ (details below for how to choose an appropriate δ_α). Within the CP framework, Inequality 1 holds for any model, any data distribution $\mathbb{P}_{XY}$ and any finite sample size n. As the coverage probability is marginalized over the input space $\mathcal{X}$, it is possible that undercoverage occurs in some regions of $\mathcal{X}$ (*e.g.* in Figure 1, the coverage is locally smaller than $1-\alpha$ for the clusters of red points within some time windows).

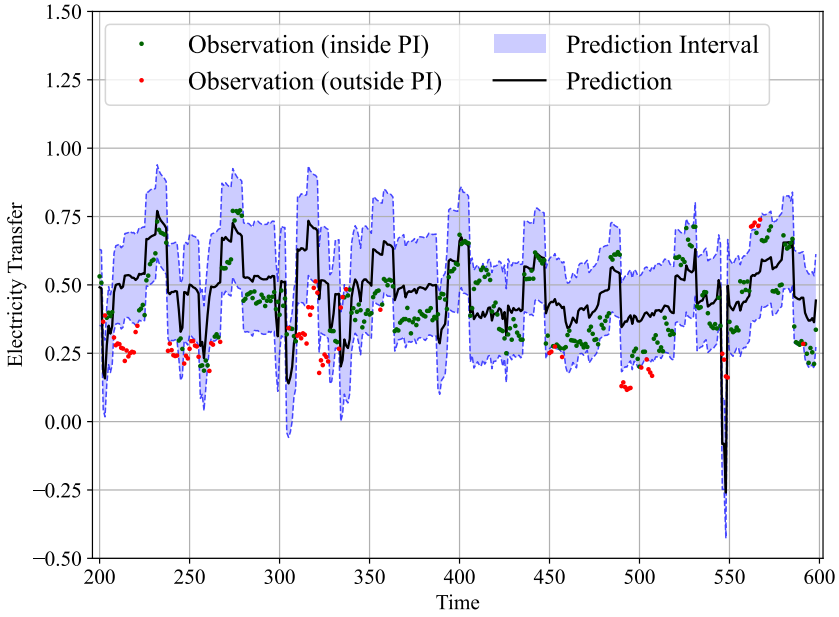

Figure 1: Example of conformal regression applied on *elec2*, a dataset of electricity market in Australia (more details in Section 3). The prediction intervals aim to cover the true outputs with an error rate at most 10%.

The canonical entry point to CP is the monograph by Vovk et al. (2005), collecting and expanding the early contributions to the field, notably Gammerman et al. (1998) and Papadopoulos et al. (2002). Since then, multiple studies have brought new ideas. A first achievement is the relaxation of full conformal, which originally requires the prediction algorithm to be invoked as many times as the cardinality of the output space $\mathcal{Y}$ (which is infinite for regression problems). Alternatively, split (or inductive) CP approach relies on held-out calibration data (Papadopoulos et al., 2002; Lei et al., 2018) to reduce the computational complexity. Another method, the jackknife+ by Barber et al. (2019), is preferred when data is scarce and it is not affordable to dedicate some of them for calibration only. Jackknife+ relies on *leave-one-out* or K-fold data schemes for better statistical efficiency and comes with the cost of training several models.

The general inductive CP procedure can be summarized as follows:

1. Define a nonconformity score function $(x, y) \to s(x, y)$ on $\mathcal{X} \times \mathcal{Y} \to \mathbb{R}$, where $\mathcal{X}$ and $\mathcal{Y}$ are the input and the output spaces, respectively.
2. Compute δ_α, the $(1-\alpha)(1+\frac{1}{n})$-th empirical quantile of the scores $\{s(X_1, Y_2), \dots s(X_n, Y_n)\}$.
3. Build the prediction sets for new examples based on δ_α:

$$\widehat{C}_\alpha(X_{new}) = \big\{y : s(X_{new}, y) \leq \delta_\alpha\big\}. \tag{2}$$

While all conformal predictors are valid (assuming data exchangeability), the choice of a nonconformity score function is decisive to generate efficient prediction sets (the smallest possible). A substantial number of contributions focused on tailoring nonconformity scores for practical situations with more or less conditions on the underlying models (Papadopoulos et al., 2002; Lei et al., 2018; Romano et al., 2019, 2020; Angelopoulos et al., 2020).

2. Design of PUNCC

Conformal procedures are simple, flexible, usable for any model and deployable during training or afterwards (post-processing). The diversity of CP methods stems from the choice of the nonconformity scores and calibration data, all enabling rigorous *Uncertainty Quantification* (UQ) throughout valid and efficient prediction sets. The design of PUNCC aspires to leverage these properties and is guided by the following principles:

- Simplicity of use and clear documentation.
- Interoperability with diverse ML libraries.
- Flexibility to build and compare new CP procedures.

PUNCC comes with a collection of plug-&-play CP and related procedures from the literature, implemented as their authors intended in a unified environment that facilitates learning and benchmarks. Table 1 lists the methods currently implemented and their sources.

Conformal Method	Source	Task
Split Conformal Prediction (SCP)	Papadopoulos et al. (2002)	Regression
Locally Adaptive CP (LACP)	Lei et al. (2018)	Regression
Conformalized Quantile Regression (CQR)	Romano et al. (2019)	Regression
Cross-Validation+ (CV+)	Barber et al. (2019)	Regression
Adaptive Prediction Sets (APS)	Romano et al. (2020)	Classification†
Regularized Adaptive Prediction Sets (RAPS)	Angelopoulos et al. (2020)	Classification†
Ensemble batch Prediction Interval (EnbPI)*	Xu and Xie (2021)	Regression‡
Weighted Split Conformal Prediction (WSCP)*	Barber et al. (2022)	Regression

Table 1: CP methods currently implemented in PUNCC. *: these methods rely on additional hypotheses. †: binary or multiclass. ‡: developed for time series data.

Using the aforementioned methods in an UQ workflow can be done in a few lines of code and, at most, requires tuning only a couple of high-level hyperparameters (whose initialization is intuitive, documented and clearly explained in the tutorials). Such procedures are organized accordingly to their respective ML tasks in the library modules `puncc.regression` and `puncc.classification`.

Throughout the paper, the code contains variables named `X_fit`, `y_fit`, `X_calib` and `y_calib`, which are not commonly found in the existing ML terminology. These are common in CP, whenever one uses a method like SCP: the training data is split (disjointly) into $D_{\text{train}} = D_{\text{fit}} \cup D_{\text{cal}}$; D_{fit} (also known as "proper training set") is used to fit the underlying model and D_{cal} is used to compute the nonconformity scores. With CP and PUNCC, one is able to use pretrained models, which is convenient with Deep Learning architectures. In this case, only `X_calib` and `y_calib` will be used in the code, as access to training data is not necessary. The following code snippet executes the split conformal regression (Papadopoulos et al., 2002); examples of various use-cases are presented in Section 3.

```
from deel.puncc.regression import SplitCP
from deel.puncc.api.prediction import BasePredictor

# Load data
# X_train, y_train = ...
# X_test = ...

# Create a regression model
# my_model = ...

# Wrap model in a predictor (more details on this later)
my_predictor = BasePredictor(my_model, is_trained=False)

# CP method initialization
split_cp = SplitCP(my_predictor)

# The call to `fit` trains the model on the fitting set and
# computes the nonconformity scores on the calibration set.
# In this example, 80\% of the train samples are randomly
# assigned to the fitting set and 20\% to the calibration set
split_cp.fit(X_train=X_train, y_train=y_train, fit_ratio=0.8)

# The `predict` method returns the point estimates and
# prediction intervals w.r.t the significance level alpha = 10%
y_pred, y_pred_lower, y_pred_upper = split_cp.predict(X_test, alpha=0.1)
```

For research and experimentation purposes, users may want more flexibility into defining the CP approaches (*e.g.* new nonconformity scores or use of calibration data). In this case, PUNCC provides an **API** to enable more in-depth design of CP algorithms. Figure 2 presents an overview of the library's architecture.

2.1. The API

The purpose of PUNCC's **API** is to provide the flexibility to conceive CP methods in an organized and consistent framework. Its set of modules jointly define and build CP wrappers over ML models from most ML libraries. Let's say we want to fit/calibrate a neural network quantile estimator through cross-validation with a nonconformity score tailored to our application. Inductive CP is applicable here and needs to incorporate a set of building blocks. In the **API**, we propose to do so in a seamless way through our predictor-calibrator-splitter paradigm. The latter relies on three entities (assembled by an orchestrator):

- *Predictor*: wrapper for ML models to ensure interoperability with various ML libraries (such as TensorFlow, PyTorch and scikit-learn).
- *Calibrator*: entity composed of an estimator of nonconformity scores and a method to construct prediction sets.
- *Splitter*: implements a strategy to select the calibration data, such as K-fold cross-validation or some domain-dependent data partition.

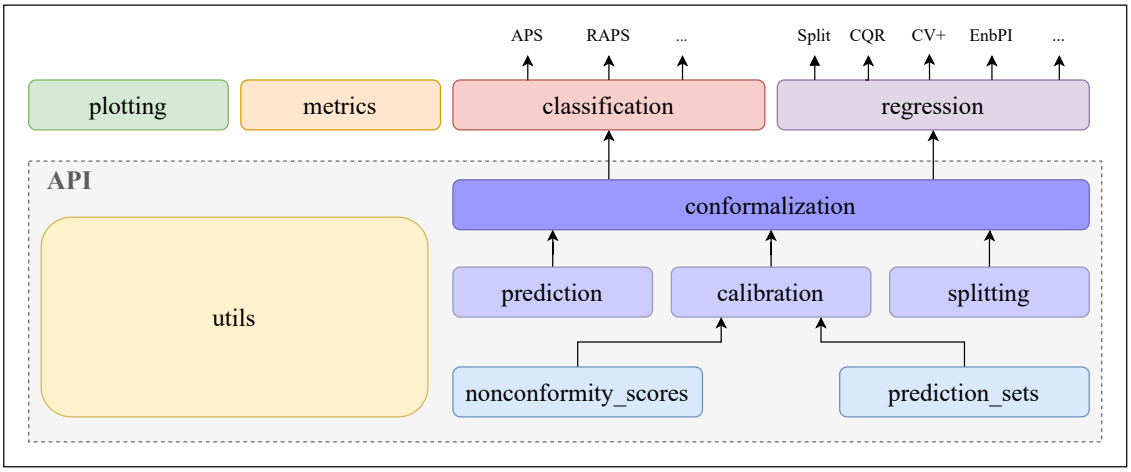

Figure 2: Overview of PUNCC's architecture.

`ConformalPredictor` class from `puncc.api.conformalization` module is the backbone of PUNCC's API, where the three entities are assembled to build a CP procedure. This canvas combines a *predictor*, a *calibrator* and a *splitter* (more details will follow):

```
from deel.puncc.api.conformalization import ConformalPredictor

# Define my_predictor, my_calibrator and my_splitter
# ...

# Conformal prediction canvas assembling our combination of predictor,
# calibrator and splitter
conformal_predictor = ConformalPredictor(predictor=my_predictor,
                                         calibrator=my_calibrator,
                                         splitter=my_splitter)
```

The `ConformalPredictor` entity is an orchestrator that characterizes the two main stages of any conformal procedure:

- `fit`: fits the *predictor* if needed (*e.g.*, in cross-validation) and computes nonconformity scores according to the calibrator and the data split strategy provided by the splitter.

```
# X_train and y_train are the training data samples.
# Internally, the splitter assigns data to the fit and calibration
# sets; then the calibrator computes nonconformity scores
conformal_predictor.fit(X_train, y_train)
```

- `predict`: for new samples, it estimates the point predictions and prediction sets, with respect to a chosen error (significance) level α.

```
# Point predictions are yielded by the predictor.
# Prediction sets are calculated from the calibrator
# for a marginal coverage target at least 1-alpha
y_pred, set_pred = conformal_predictor.predict(X_new, alpha=0.1)
```

One single call to `fit` on the `ConformalPredictor` is enough to compute the nonconformity scores on the calibration data. Subsequently, conformal predictions given different values of the target error rate α are performed with no extra computational overhead.

2.1.1. Predictor

Any model can be wrapped by a CP procedure to evaluate its predictive uncertainty, with no specific assumption about the underlying properties[2]. The only limitation is the ability to support certain model interfaces and data structures that come with ML libraries. Therefore, our **API** relies on *predictors*: wrappers that standardize the models interface and guarantee their compliance with the CP framework of PUNCC. The `BasePredictor` and `DualPredictor` classes in `puncc.api.prediction` cover two required behaviors via:

- `fit`: action used to train the model. It takes as arguments two sequences of sample (X_{train}, Y_{train}) (such as numpy arrays and tensors) and any additional configuration of the underlying model (such as random seed and number of epochs).
- `predict`: action used to predict outputs based on X. It takes as arguments an iterable X and any additional configuration of the underlying model (*e.g.* batch size).

The `BasePredictor` is built from the model to be wrapped, a flag to inform if the model is already trained (in which case the training can be skipped during the CP procedure) and the compilation configuration if the underlying model needs to be compiled (such as in TensorFlow and PyTorch). The constructor of `DualPredictor` is similar but takes as arguments a list of two models, a list of two trained flags and a list of two compilation configurations. Such *predictor* is useful when the calibration relies on several models (such as upper and lower quantiles in CQR or conditional mean and dispersion in LACP). Examples are provided in Section 3.

2. The only strong requirement is data exchangeability (or simply i.i.d).

In most situations (including with TensorFlow and PyTorch models, see Appendix A), creating a *predictor* is as simple as calling its constructor on the underlying models that natively implement `fit` and `predict`:

```
from deel.puncc.api.prediction import BasePredictor

# Definition of a predictor
my_predictor = BasePredictor(my_model)
```

Sometimes, the user has to subclass `BasePredictor` or `DualPredictor` to comply with the fit-predict interface. For instance: conformal classification built over a random forest classifier from scikit-learn. To calibrate such classifier, the APS procedure (for example) requires the model to output the estimated probability of each class, but `RandomForestClassifier .predict(.)` returns only the most likely class. We need to create a *predictor* where we redefine the `predict` call:

```
from sklearn.ensemble import RandomForestClassifier
from deel.puncc.api.prediction import BasePredictor

# Create a random forest classifier
rf_model = (n_estimators=100, random_state=0)

# Create a wrapper of the random forest model to redefine its predict method
# into logits predictions. Make sure to subclass BasePredictor.
# Note that we needed to build a new wrapper (over BasePredictor) only
# because the predict(.) method of RandomForestClassifier does not predict
# logits. Otherwise, it is enough to use BasePredictor (e.g., neural network
# with softmax).
class RFPredictor(BasePredictor):
    def predict(self, X, **kwargs):
        return self.model.predict_proba(X, **kwargs)

# Wrap model in the newly created RFPredictor
rf_predictor = RFPredictor(rf_model)
```

2.1.2. Calibrator

The *calibrator* provides a structure to compute the nonconformity scores on the calibration set and to compute the prediction sets for new samples. When building a `BaseCalibrator` from `deel.puncc.api.calibration` module, one decides which nonconformity score and prediction set functions to use. The *calibrator* instance computes the scores (*e.g.* mean absolute deviation) via the `fit` method on the calibration dataset. Based on the estimated quantiles of the nonconformity scores, the *calibrator* constructs and/or calibrates the prediction sets on a new data point via `calibrate`. For example, the `BaseCalibrator` in the split conformal regression (proposed by Papadopoulos et al.) uses the mean absolute deviation as nonconformity score and prediction sets are built as constant intervals. These and other functions are provided in the modules `puncc.api.nonconformity_scores` and `puncc.api.prediction_sets`.

```
from deel.puncc.api.calibration import BaseCalibrator
from deel.puncc.api.nonconformity_scores import mad
from deel.puncc.api.prediction_sets import constant_interval

# Calibrator construction
my_calibrator = BaseCalibrator(nonconf_score_func=mad,
                               pred_set_func=constant_interval)
```

One can also define custom functions and pass them as arguments to the *calibrator*:

```
from deel.puncc.api.calibration import BaseCalibrator

# Definition of a custom nonconformity scores function.
# Alternatively, several ready-to-use nonconf scores are provided in
# the module deel.puncc.nonconformity_scores
def my_ncf(y_pred, y_true):
    return abs(y_pred-y_true)

# Definition of a custom function to build prediction sets.
# Alternatively, several ready-to-use procedure are provided in
# the module deel.puncc.prediction_sets
def my_psf(y_pred, nonconf_scores_quantile):
    y_lower = y_pred - nonconf_scores_quantile
    y_upper = y_pred + nonconf_scores_quantile
    return (y_lower, y_upper)

# Calibrator construction
my_calibrator = BaseCalibrator(nonconf_score_func=my_ncf,
                               pred_set_func=my_psf)
```

2.1.3. Splitter

In inductive CP, assigning data samples in the calibration set is motivated by two criteria: data availability and computational resources. If data is abundant, we can split the training samples into disjoint subsets D_{fit} and D_{cal}. When data is scarce, a cross-validation strategy is more suited. These and other strategies are selectable in the `puncc.api.splitting` module. Their implementation is compliant with several data structure such as numpy arrays, PyTorch and TensorFlow tensors and pandas dataframes. Some examples are random K-fold and deterministic user-defined *splitters*:

```
from deel.puncc.api.splitting import KFoldSplitter

# Definition of a K-fold splitter that produces
# 20 folds of fit/calibration
kfold_splitter = KFoldSplitter(K=20)
```

```
from deel.puncc.api.splitting import IdSplitter

# Definition of an identity splitter that wraps
# a predetermined partition of fit  and calibration data
id_splitter = IdSplitter(X_fit, y_fit, X_calib, y_calib)
```

3. Examples

In this section, we provide several examples where PUNCC is used to quantify the uncertainty of prediction models. For each example, we present the experimental settings (dataset, learning task, etc.), discuss some key aspects about the implementation and obtained results. The full source code is available online[3] and additional code snippets are highlighted in the next sections and Appendix A. Note that an earlier version of PUNCC was used to benchmark several CP procedure on times series, specifically on an industrial dataset by Air Liquide for gas demand forecasting (Mendil et al., 2022).

3.1. Tabular Regression on California Housing Prices

In this experiment, we use PUNCC to run several conformal regressions on the California housing price dataset and compute their performance (validity and efficiency) for different error targets α. The underlying task consists of predicting median house prices within a given area based on structural features such as the number of rooms and household incomes (Pace and Barry, 1997). We compare different models from multiple ML libraries:

- **Neural networks** from the libraries PyTorch and TensorFlow to estimate the conditional mean median house prices.
- ***eXtreme Gradient Boosting*** **(XGBoost)** (resp. **nearest neighbors**) from the library xgboost (resp. scikit-learn) to estimate the conditional mean (resp. conditional mean absolute deviation of) median house prices.
- **Gradient boosting** from scikit-learn to estimate conditional upper and lower quantiles of the median house prices.

During the learning phase, the parameters of each model are obtained by minimization of the loss function on D_{fit} according to the fit-calibration scheme. The mean squared error is considered for the neural networks and XGBoost models; while the pinball loss is used for gradient boosting quantile regression. The following conformal procedures are performed to build and calibrate the interval predictions:

- **SCP** based on each of the neural network models. 80% (resp. 20%) of the train samples are randomly assigned to fitting (resp. calibration).
- **LACP** based on XGBoost and nearest neighbors models. 80% (resp. 20%) of the train samples are randomly assigned to fitting (resp. calibration).
- A combination of **CQR** and **CV+** built from the gradient boosting quantile regression models. 10 000 samples are divided based on 10-folds cross-validation.

3.1.1. Implementation Highlights

The experimental setup involves using different models from various ML libraries. One key aspect of the implementation is wrapping the defined models in `predictor` instances. As previously mentioned, the goal is to expose a consistent interface (`fit` and `predict`) to be called later in the CP procedures. Depending on the ML libraries, this wrapping can either

3. https://github.com/M-Mouhcine/PunccExperiments

be straightforward or may need some adaptations. The instances of the chosen models from TensorFlow, xgboost and scikit-learn all implement the `fit` and `predict` methods; therefore, they can be directly encapsulated. However, neural networks in PyTorch do not natively implement these calls and additional steps are to be taken to conform to PUNCC's interface. Examples of code snippets are provided in Appendix A, but the user is free to adopt any implementation that meets the requirements. SCP and LACP are instantiated from the module `puncc.regression` and the calibration in CQR with a cross-validation scheme uses PUNCC's **API**.

3.1.2. Results

We run the experiments with 10 000 train and test samples with 20-fold cross-validation and $\alpha \in \{0.1, 0.2, \ldots, 0.9\}$. The average validity and efficiency of the experiments, along with their standard deviations, are reported in Table 2. Validity is assessed by calculating the gap between the empirical coverage and the target $1 - \alpha$; efficiency is measured by the width of the *Prediction Interval*s (PIs). Both metrics are averaged over all 20 validation folds. The empirical results show that CP behaves as expected with negligible coverage gaps. The average width of the PIs, on the other hand, is linked to the chosen models and the conformalization procedure. The intervals predicted by LACP and CQR are tighter compared to SCP. This stems from their adaptiveness to heteroscedastic data, leading to tight PIs of varying sizes (conditionally to X, in contrast to the fixed-size intervals with SCP), which experimentally validates the claims of the authors (Lei et al., 2018; Romano et al., 2019).

Table 2: Coverage gap (between empirical and target) and width of the predicted intervals averaged across 20 folds.

Method	**Coverage Gap ($\pm$ std)**	**Width ($\pm$ std)**
CQR CV+	0.02 ± 0.012	0.80 ± 0.028
LACP	0.02 ± 0.010	0.62 ± 0.018
SCP (`PyTorch`)	0.02 ± 0.011	1.22 ± 0.025
SCP (`Tensorflow`)	0.02 ± 0.012	1.14 ± 0.093

We also evaluate the performance of the CP methods for target error rates α spread between 10% and 90%. Figure 3 displays the average empirical average for each α, with the perfect calibration line indicating the case where target coverage equals empirical coverage (ideal scenario in CP, i.e. validity without excessive conservativenes). The results demonstrate that the methods are well calibrated. Figure 4 shows an increasing PI width for smaller values of α. As expected, LACP and CQR exhibit adaptiveness, particularly for large target coverage $1 - \alpha$ where the PIs are twice as narrow as those produced by SCP.

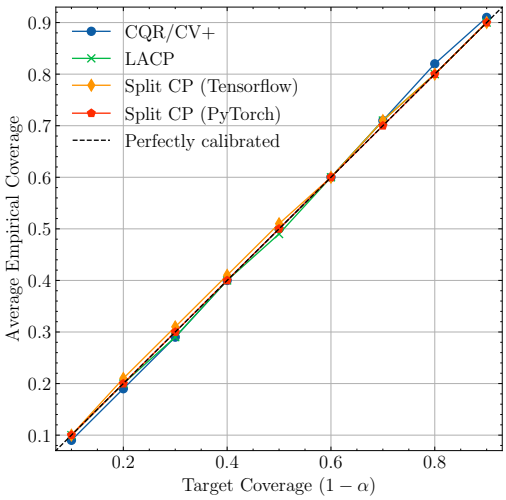

Figure 3: Average coverage gap (across 20 validation folds).

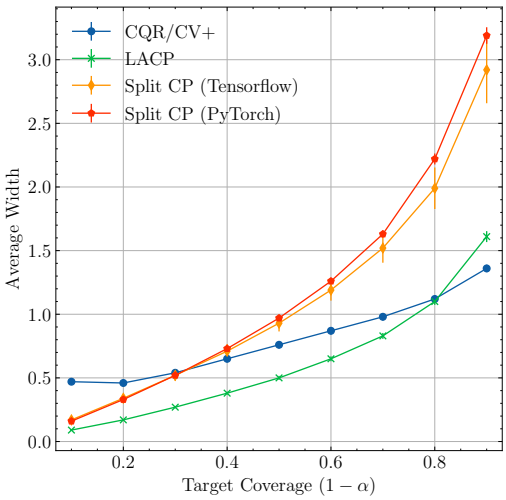

Figure 4: Average empirical size of PIs (across 20 validation folds).

3.2. Classification on Imagenette and Imagewoof

The purpose of this experiment is to estimate the predictive uncertainty of a ResNet-50, a classifier introduced by He et al. (2016) and pretrained on *Imagenet*. We consider two image datasets of increasing levels of difficulty provided by Howard et al. (2020). The first dataset is *Imagenette*, which comprises ten dissimilar classes from *Imagenet*. The second dataset, *Imagewoof*, includes ten classes of dog breeds from *Imagenet* that are inherently more difficult to classify due to their resemblance. The following conformal classifiers are called from PUNCC to build and calibrate the set predictions for different values of α:

- **APS** based on the pretrained ResNet-50. 2048 samples are used for calibration and 2048 for testing on each dataset.
- **RAPS** based on the pretrained ResNet-50. The regularization hyper-parameters chosen for this study are $\lambda = 0.01$ and $k_{\text{reg}} = 6$, representing the weight of the regularization term and the number of classes at which the regularization begins to take effect, as detailed by Angelopoulos et al. (2020). 2048 samples are employed for calibration, and a further 2048 are utilized for testing on each dataset.

3.2.1. Implementation Highlights

PUNCC enables the integration of pretrained blackbox models into the CP process. This involves wrapping the model in the predictor and specifying whether it has already been trained by setting the corresponding flag. The code snippet below shows how a pretrained ResNet-50 model from Keras is encapsulated:

```
import tensorflow as tf

# Load ResNet50 as a pretrained Keras model
model = tf.keras.applications.ResNet50(weights="imagenet")

# The classifier is wrapped in a predictor
# with the flag is_trained set to True
predictor = BasePredictor(model, is_trained=True)
```

Thereafter, the conformal procedure should be initialized with the `train` flag set to false. In this case, the call to `fit` on the conformal predictor will skip the model training and only trigger the computation of nonconformity scores. APS and RAPS are directly available in the high level module `puncc.classification`:

```
from deel.puncc.classification import APS, RAPS

# Initialize the APS and RAPS with restnet50 predictor
# and the flag train set to False to skip model fitting
aps = APS(predictor, train=False)
raps = RAPS(predictor, train=False, lambd=0.01, k_reg=6)
```

Besides, an interesting feature of PUNCC is caching the nonconformity scores computed on the calibration set. This way, generating prediction sets for different values of α will have no additional overhead and will utilize the appropriate quantile on the cached nonconformity scores to construct the prediction sets.

3.2.2. Results

Table 3 reports the average coverage gap and prediction set size obtained with APS and RAPS on the *Imagenette* and *Imagewoof* datasets. The initial accuracy of RestNet-50 is limited: 55% on *Imagenette* and 26% on *Imagewoof*, indicating the particular difficulty of classifying images from the second dataset. The difficulty of image classification has a significant impact on model uncertainty in the context of CP. Although both CP methods are almost valid, the average size of the prediction sets is nearly five times larger on *Imagewoof* compared to *Imagenette*. Additionally, the regularization in RAPS reduces the cardinality of the prediction sets by cutting out classes in the far-end tail of the distribution, resulting in exchange in a small increase in the coverage gap (at the third decimal place). Figure 5 and Figure 6 confirm the overall trend on specific coverage targets $1 - \alpha$. The complexity of *Imagewoof* leads to more conservative (larger) prediction sets.

Table 3: Average coverage gap (between empirical and target) and average size of the prediction sets.

	Accuracy		**Coverage Gap**		**Size**	
Dataset	Top-1	Top-5	APS	RAPS	APS	RAPS
Imagenette	0.55	0.77	0.01	0.01	7.45	4.78
Imagewoof	0.26	0.45	0.03	0.03	40.63	21.21

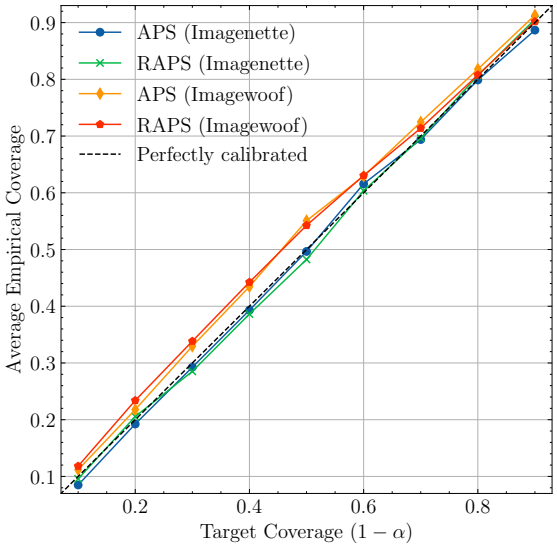

Figure 5: Average coverage gap.

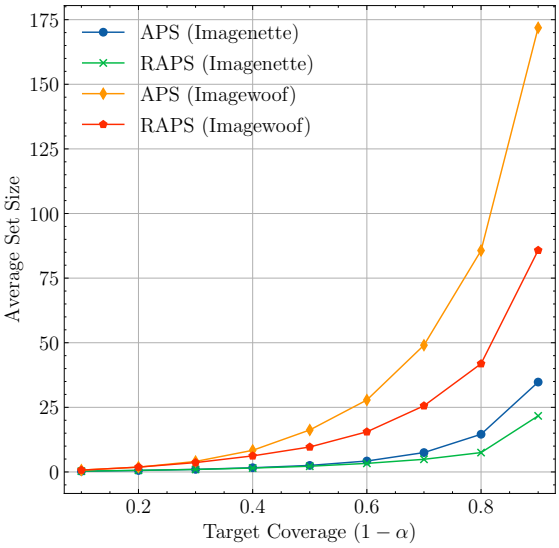

Figure 6: Average size of prediction sets.

3.3. Time Series Regression on Australian Electricity Market

This experiment aims to estimate uncertainty for time series regression using PUNCC. We consider the *Elec2* dataset introduced by Harries et al. (1999), which consists of records of electricity transfers between the Australian states of New South Wales and Victoria. The transfers occur every 30 minutes and are affected by the varying electricity demand and price in each state. Vovk et al. (2021) reported the presence of distribution drifts in *Elec2*, which makes the data non-exchangeable and can invalidate the guarantee of CP marginal coverage.

To capture the uncertainty, we work with two state-of-the-art methods, WSCP by Barber et al. (2022) and EnbPI by Xu and Xie (2021) for $\alpha = 0.1$. Despite the data not being exchangeable, we use *Online Sequential Split Conformal Prediction* (OSSCP) as a baseline like Zaffran et al. (2022). We use a linear regressor from scikit-learn as the underlying model, for which the fitting, calibration and training sets have a constant size and are updated each time a new test point is available (similarly to an online setting with incoming data flow) as shown in Figure 7. The split scheme used with WSCP and OSSCP produces two disjoint sequential subsets, while EnbPI uses the whole dataset at time t with a bootstrap procedure.

The WSCP is a modified version of SCP that uses weighted nonconformity scores. A higher weight reflects the belief that the calibration point comes from (nearly) the same distribution as the test point. We use weights that decay exponentially over time so that most recent samples have larger weights:

$$w_i = 0.99^{k+1-i},$$

where k is the size of the calibration set and i is the time index.

EnbPI combines B predictors to compute the nonconformity scores with a technique inspired by Jackknife+-after-Bootstrap (Kim et al., 2020); we set $B = 20$.

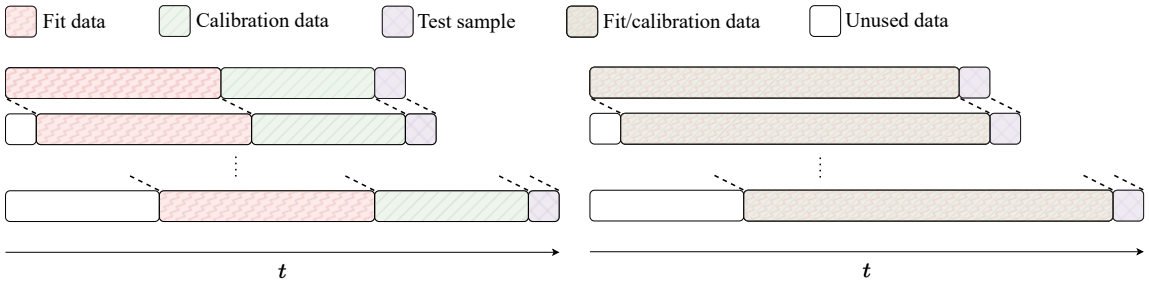

Figure 7: Data partition used for WSCP and OSSCP (left) and EnbPI (right) (adapted from Zaffran et al. (2022)).

3.3.1. Implementation Highlights

PUNCC can integrate any user-defined weight function in several state-of-the-art CP methods implemented in `puncc.regression` (or more generally during `calibrators` definition). The code snippet below illustrates the construction of a WSCP with exponentially decaying weights:

```
from deel.puncc.regression import SplitCP

# Definition of a predictor wrapping the underlying model
predictor = ...

# Definition of exponentially decaying weights function
def exp_decay(nc_scores):
    k = len(nc_scores)
    return [.99 ** (k + 1 - i) for i in range(k)]

# Definition of split conformal predictor whose nonconformity scores
# are weighted according to the function defined above
wscp = SplitCP(predictor, weight_func=exp_decay)
```

EnbPI interval predictor is also directly available in `puncc.regression`:

```
from deel.puncc.regression import EnbPI

# Definition of a predictor wrapping the underling model
predictor = ...

# Definition of EnbPI interval predictor with 20 bootstrap models
enbpi = EnbPI(predictor, B=20)
```

The sequential data partition scheme (see Figure 7) is deterministic and can be accomplished by explicitly providing updated fit and calibration sets to the interval predictors:

```
for X, y in data_sequence:
    # Decompose data into fit and calibration subsets
    # with a defined user_split function
    X_fit, y_fit, X_calib, y_calib = user_split(X, y)

    # The call to fit for WSCP trains the model on the fit set
    # and computes the nonconformity scores on the calibration set
    wscp.fit(X_fit=X_fit, y_fit=y_fit, X_calib=X_calib, y_calib=y_calib)

    # The call to fit for EnbPI trains the model and computes
    # the nonconformity scores in a Jackknife+-after-bootstrap fashion
    enbpi.fit(X, y)
```

Finally, the PIs are generated by running the `predict` method on the interval predictors `wscp` and `enbpi` for an chosen α. Note that in case of WSCP, the nonconformity scores are factored by the user-defined weights and the quantile is computed consistently with Tibshirani et al. (2019) quantile lemma.

3.3.2. Results

Figure 8 reports the rolling average coverage and PI width obtained with three uncertainty estimation methods. The coverage and width are averaged over a sliding window of 100 points. Results indicate that WSCP and EnbPI exhibit higher resilience to the heavy drop in coverage compared to the PIs constructed via conformal prediction (OSSCP), which lose coverage and drop to around 60% at a given time. The decrease in coverage is less noticeable for WSCP and EnbPI. WSCP adaptively inflates the PIs to recover coverage while EnbPI produces PIs of steady size but relies on a more stable underlying model (less overfitting thanks to bootstrapping), leading to a better approximation of conditional coverage. Figure 9 displays the PIs obtained on 600 test points via the plotting module of PUNCC.

In this experiment, the coverages of WSCP (89%) and EnbPI (90%) are valid while OSSCP (84%) is approximately valid. In general, the performance of CP (or CP-related) algorithms on times series depends on the problem and the data (Mendil et al., 2022).

4. Conclusion

We presented PUNCC, a new open-source library in Python for uncertainty quantification based on CP. The library is model-agnostic and can be used with popular ML frameworks such as TensorFlow, PyTorch and scikit-learn. The high-level modules of the library provide a set of state-of-the-art CP and CP-related algorithms for regression and classification, each implemented as described by the authors within a unified environment to facilitate learning and benchmarks. The library's **API** enables users to design advanced procedures and customize every step of inductive CP algorithms. It relies on a paradigm that encompasses 1) a predictor: wrapper to standardize models interface and to guarantee their iteroperability with PUNCC, 2) a calibrator: structure to estimate nonconformity scores on the calibration set and to compute the prediction sets for new samples and 3) a splitter: entity to assign data to the fit and calibration sets and to enforce compliance with several data structures

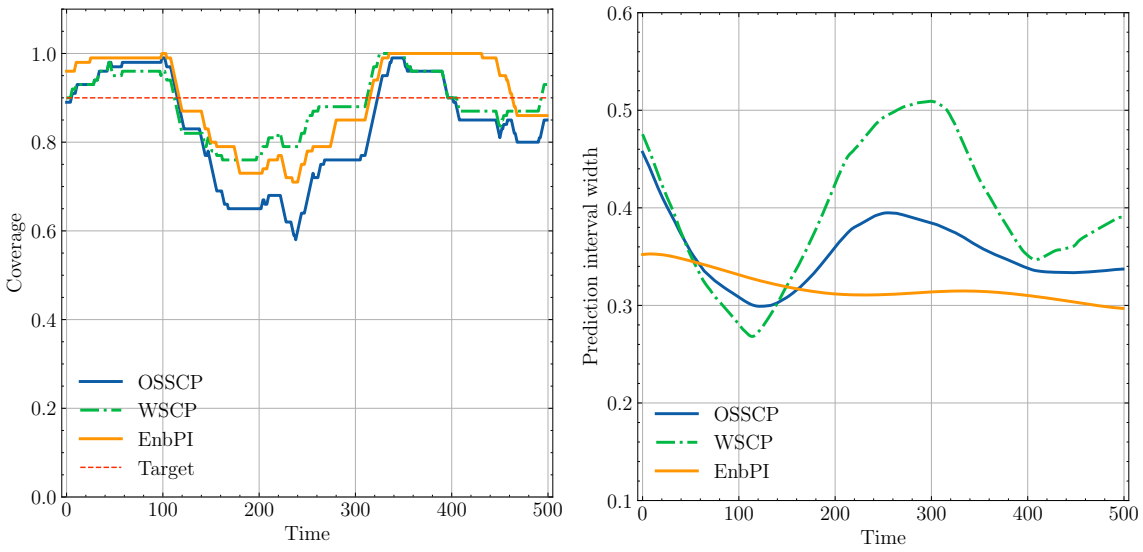

Figure 8: Rolling average (over a window of 100 time points) of coverage and width of PIs generated with WSCP, EnbPI and OSSCP on *Elec2* data.

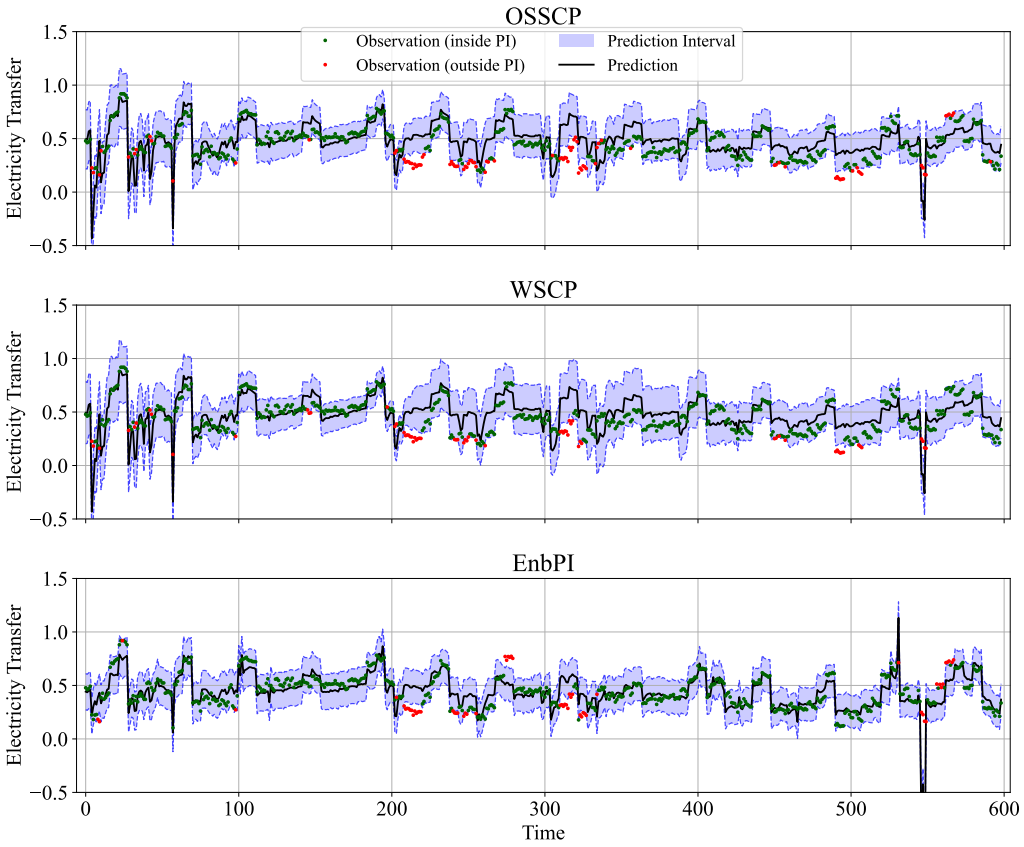

Figure 9: PIs on 600 test samples for three different algorithms (using `puncc.plotting`).

such as arrays and tensors. We demonstrated how PUNCC can generate prediction sets on various classification and regression problems. Its unified environment makes it easy to explore, sketch and compare different CP procedures. PUNCC also offers functionalities to analyze and display the results using the the metrics and plotting modules, respectively.

In the future, we plan to expand the state-of-the-art CP collection and update it regulary with newly released methods. We also aim to extend PUNCC to online CP and include other tasks such as anomaly detection and multivariate prediction.

Acknowledgments

This work has benefited from the AI Interdisciplinary Institute ANITI, which is funded by the French "Investing for the Future - PIA3" program under the Grant agreement ANR-19-P3IA-0004. The authors gratefully acknowledge the support of the DEEL project (https://www.deel.ai)

References

Anastasios Angelopoulos, Stephen Bates, Jitendra Malik, and Michael I Jordan. Uncertainty sets for image classifiers using conformal prediction. *arXiv:2009.14193*, 2020.

Rina Foygel Barber, Emmanuel J. Candes, Aaditya Ramdas, and Ryan J. Tibshirani. Predictive inference with the jackknife+, 2019. URL https://arxiv.org/abs/1905.02928.

Rina Foygel Barber, Emmanuel J. Candes, Aaditya Ramdas, and Ryan J. Tibshirani. Conformal prediction beyond exchangeability, 2022.

A Gammerman, V Vovk, and V Vapnik. Learning by transduction. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 148–155, 1998.

Michael Harries, New South Wales, et al. Splice-2 comparative evaluation: Electricity pricing. Technical report, University of New South Wales, 1999.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Jeremy Howard et al. Imagenette, 2020. URL https://github.com/fastai/imagenette.

Byol Kim, Chen Xu, and Rina Barber. Predictive inference is free with the jackknife+-after-bootstrap. *Advances in Neural Information Processing Systems*, 33:4138–4149, 2020.

Jing Lei, Max G'Sell, Alessandro Rinaldo, Ryan J. Tibshirani, and Larry Wasserman. Distribution-free predictive inference for regression. *Journal of the American Statistical Association*, 113(523):1094–1111, 2018.

Rachel Luo, Shengjia Zhao, et al. Sample-efficient safety assurances using conformal prediction. In *Algorithmic Foundations of Robotics XV*. Springer, 2023.

Franck Mamalet, Eric Jenn, et al. White Paper Machine Learning in Certified Systems. Research report, IRT Saint Exupéry; ANITI, 2021. URL hal.science/hal-03176080.

Mouhcine Mendil, Luca Mossina, Marc Nabhan, and Kevin Pasini. Robust gas demand forecasting with conformal prediction. In *Proceedings of COPA*. PMLR, 2022.

R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

Harris Papadopoulos, Kostas Proedrou, Volodya Vovk, and Alex Gammerman. Inductive confidence machines for regression. In *Machine Learning: ECML 2002*, 2002.

Yaniv Romano, Evan Patterson, and Emmanuel Candes. Conformalized quantile regression. In *Advances in Neural Information Processing Systems*, 2019.

Yaniv Romano, Matteo Sesia, and Emmanuel Candes. Classification with valid and adaptive coverage. *Advances in Neural Information Processing Systems*, 33:3581–3591, 2020.

Ryan J Tibshirani, Rina Foygel Barber, Emmanuel Candes, and Aaditya Ramdas. Conformal prediction under covariate shift. *NeurIPS*, 32, 2019.

Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*, volume 29. Springer, 2005.

Vladimir Vovk, Ivan Petej, and Alex Gammerman. Protected probabilistic classification. In *Conformal and Probabilistic Prediction and Applications*, pages 297–299. PMLR, 2021.

Chen Xu and Yao Xie. Conformal prediction interval for dynamic time-series. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

Margaux Zaffran, Olivier Féron, Yannig Goude, Julie Josse, and Aymeric Dieuleveut. Adaptive conformal predictions for time series. In *Proceedings of ICML*. PMLR, 2022.

Appendix A. Implementation Highlights for California Housing Prices

Without loss of generality[4], here is an example of model encapsulation from TensorFlow, xgboost and scikit-learn accordingly to the setup presented in section 3.1:

```
from tensorflow.keras import Sequential, layers
from deel.puncc.api.prediction import BasePredictor

# Model definition: one hidden layer composed of 20 neurons
nnet = Sequential([layers.Dense(20, activation='relu'), layers.Dense(1)])

# TensorFlow models need to be compiled, for example:
compile_kwargs={'optimizer':'sgd', 'loss':'mse'}

# Model wrapped in a predictor
predictor = BasePredictor(nnet, **compile_kwargs)
```

4. The chosen models are simple and their hyperparameters are not optimized; the main aim being to demonstrate PUNCC's features. The results are still relevant because CP is model-agnostic.

```
from xgboost import XGBRegressor
from sklearn import neighbors
from deel.puncc.api.prediction import BasePredictor

# Model definition
xgb = XGBRegressor(n_estimators=100)
knn = neighbors.KNeighborsRegressor(n_neighbors=5)

# Models wrapped in a predictor
predictor = DualPredictor([xgb, knn])
```

The wrapper for neural networks in PyTorch requires extra adaptation to conform to PUNCC's interface. The following code snippet is an example of solutions:

```
import torch
from deel.puncc.api.experimental import TorchPredictor

# Model definition: one hidden layer composed of 20 neurons
class LinearTorchModel(torch.nn.Module):
    def __init__(self, input_feat, output_feat):
        super(LinearTorchModel, self).__init__()
        self.linear1 = torch.nn.Linear(input_feat, 20)
        self.relu = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(20, output_feat)

    def forward(self, x):
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        return out

# For input_shape and output_shape already defined
nnet = LinearTorchModel(input_shape, output_shape)

# Torch models need to be compiled, for example:
compile_kwargs= {"lr": 1e-3}

# Model wrapped in a predictor
predictor = TorchPredictor(
                model = nnet,
                optimizer=torch.optim.Adam,
                criterion=torch.nn.MSELoss(reduction="sum"),
                **compile_kwargs)
```

The initialization of SCP and LACP instances is straighforward from the module `puncc.regression`. The calibration in CQR procedure with a cross validation scheme is achieved using the **API** as follows:

```
from deel.puncc.api.calibration import DualPredictor
from deel.puncc.api.nonconformity_scores import cqr_score
from deel.puncc.api.prediction_sets import cqr_interval
from deel.puncc.api.splitting import KFoldSplitter
from deel.puncc.api.conformalization import ConformalPredictor
```

```
# Definition of the upper quantile model
regressor_q_hi = ...

# Definition of the lower quantile model
regressor_q_low = ...

# Models wrapped in a predictor
predictor = DualPredictor(models=[regressor_q_low, regressor_q_hi])

# Custom calibrator using CQR nonconformity scores and prediction sets
calibrator = BaseCalibrator(nonconf_score_func = cqr_score,
                            pred_set_func = cqr_interval)

# Definition of a Kfold splitter
splitter = KFoldSplitter(K=10)

# Canvas assembling the predictor, calibrator and splitter
conformal_predictor = ConformalPredictor(
                            predictor = predictor,
                            calibrator = calibrator,
                            splitter = splitter)
```