crepes: a Python Package for Generating Conformal Regressors and Predictive Systems

Henrik Boström BOSTROMH@KTH.SE
School of Electrical Engineering and Computer Science,
KTH Royal Institute of Technology, Sweden

Editor: Ulf Johansson, Henrik Boström, Khuong An Nguyen, Zhiyuan Luo and Lars Carlsson

Abstract

The recently released Python package `crepes` can be used to generate both conformal regressors, which transform point predictions into prediction intervals for specified levels of confidence, and conformal predictive systems, which transform the point predictions into cumulative distribution functions (conformal predictive distributions). The `crepes` package implements standard, normalized and Mondrian conformal regressors and predictive systems, and is completely model-agnostic, using only the residuals for the calibration instances, possibly together with difficulty estimates and Mondrian categories as input, when forming the conformal regressors and predictive systems. This allows the user to easily incorporate and evaluate novel difficulty estimates and ways of forming Mondrian categories, as well as combinations thereof. Examples from using the package are given, illustrating how to incorporate some standard options for difficulty estimation, forming Mondrian categories and the use of out-of-bag predictions for calibration, through helper functions defined in a separate module, called `crepes.fillings`. The relation to other software packages for conformal regression is also pointed out.

Keywords: Conformal regressors, Conformal predictive systems, Mondrian conformal regressors, Mondrian conformal predictive systems, Python

1. Introduction

Conformal prediction is a framework for turning point predictions into set predictions, providing finite-sample validity guarantees (Vovk et al., 2005). Conformal regressors extend standard regressors, by forming prediction regions, such that the true regression values fall within the regions with a user-specified probability (the confidence level, c). The inductive (or split) approach to forming conformal regressors (Papadopoulos et al., 2002), uses one set of instances (the proper training instances) to train an underlying regression model and another (the calibration set) to calculate absolute residuals (differences of actual and predicted values) of the underlying model. From these scores, the cth percentile is used to form prediction intervals for the test instances; the lower and upper bounds of the intervals are obtained from subtracting and adding, respectively, this score from the point prediction of the underlying model. The intervals produced by such a (standard) conformal regressor will all have the same size and hence do not provide instance-specific uncertainty quantification. The procedure can however be extended to produce so-called normalized conformal regressors, by dividing the scores with difficulty (or quality) estimates and using

such also for the test instances; the lower and upper bounds are obtained by multiplying the score obtained from the calibration set with the difficulty estimate. This modified procedure now clearly provides instance-specific interval sizes.

As observed in (Boström and Johansson, 2020), the normalized conformal regressors suffer from two potential problems; i) the size of the produced intervals can be many times larger (or smaller) than anything previously observed, and ii) non-informative difficulty estimators result in larger variance of the interval sizes compared to more informative estimators, while the opposite is desired; if the estimator provides no information on the expected error, this should be reflected by intervals of uniform size. In (Boström and Johansson, 2020), the Mondrian conformal regressor was proposed as a remedy, by which the calibration instances are partitioned and a (standard) conformal regressor is formed from each category. By using binning of the difficulty estimates to form the categories, it was shown that the two problems can be handled without sacrificing efficiency (interval size).

Conformal predictive systems extend conformal regressors, by producing cumulative probability distributions (conformal predictive distributions) over the possible target values (Vovk et al., 2020). One may from such a distribution obtain the probability that the true target is below (or above) a certain threshold. Conversely, one may obtain the threshold value which is associated with a certain cumulative probability. The inductive (split) procedure for forming conformal predictive systems is very similar to that of conformal regressors; the main difference is that all residuals from the calibration set are used when making predictions, rather than the absolute value of a single residual; the residuals are added to the point prediction of a test instance, and the resulting values are used to estimate the cumulative probabilities. Similarly to conformal regressors, one may distinguish between *standard* and *normalized* conformal predictive systems, where the latter employ instance-specific difficulty (quality) estimates, while the former just uses the residuals without modification. Distributions output by a standard conformal predictive system may hence differ only in their location and not in their shape, while distributions of a normalized conformal predictive system may differ also in shape; the latter can be stretched out along the value dimension based on the difficulty.

Conformal predictive systems generalize both standard and conformal regressors, as they can still produce point predictions and prediction intervals, respectively. The latter can be obtained from a conformal predictive distribution through the threshold values that correspond to the lower and upper percentiles of interest, while a point prediction can be obtained, e.g., from the median or the mean of the distribution, instead of using the prediction of the underlying model. However, as observed in (Boström et al., 2021), for heteroscedastic residuals, neither standard nor normalized predictive distributions are very effective for calibrating the predictions, since they can be adjusted in one direction only. To overcome this problem, the idea of forming multiple conformal predictive systems through partitioning the instances was proposed and implemented through so-called Mondrian conformal predictive systems (Boström et al., 2021).

Table 1: Software packages for conformal regressors and predictive systems

Software package	Lang	Ind	Trans	Agg	OOB	Mond	CPS
nonconformist[1]	Python	✓	✗	✓	✓	✗	✗
Orange3-Conformal[2]	Python	✓	✗	✓	✗	✗	✗
MAPIE[3]	Python	✗	✗	✓	✓	✗	✗
conformalInference[4]	R	✓	✓	✗	✗	✗	✗
crepes[5]	Python	✓	✗	✗	✓	✓	✓

There is a hence a range of possibilities for generating (split) conformal regressors or predictive systems, concerning the choice of:

- learning algorithm
- calibration instances, including whether or not to use out-of-bag predictions
- difficulty (quality) estimator
- Mondrian categories

To allow for efficiently exploring the space of possibilities, software packages are needed that allow for experimenting with different algorithms for generating the underlying models, etc. In Table 1, we provide pointers to some software packages for conformal regressors and predictive systems that are publicly available, together with some of their distinguishing features, namely: programming language (Lang), whether inductive/split (Ind) and transductive (Trans) conformal regressors or predictive systems may be generated, whether aggregated/cross-conformal predictors (Agg) may be generated, including the use of approaches such as jackknife+ (Barber et al., 2021), whether out-of-bag (OOB) predictions may be used for calibration, whether Mondrian conformal regressors or predictive systems can be generated, and finally whether conformal predictive systems (CPS) can be generated. As seen from the table, `crepes` is the only package from the selection that allows for generating conformal predictive systems as well as Mondrian conformal regressors/predictive systems, hence filling an important niche.

In this paper, we provide an overview of the `crepes` package, and illustrate how it may be used for generating conformal regressors and predictive systems. Examples are provided from using the package in conjunction with a separate module, called `crepes.fillings`, which implements some standard options for defining difficulty estimates and Mondrian categories.

In the next section, we describe how to install and set everything up for working with `crepes`. Then we show how to generate conformal regressors in section 3 and conformal

1. https://github.com/donlnz/nonconformist
2. https://github.com/biolab/orange3-conformal
3. https://github.com/scikit-learn-contrib/MAPIE
4. https://github.com/ryantibs/conformal
5. https://github.com/henrikbostrom/crepes

predictive systems in section 4. Finally, in section 5, we summarize the main conclusions from the current implementation and outline some directions for further developments.

2. Installing and importing crepes

The source code of the Python package `crepes`, which is licensed under the permissive BSD-3-Clause license, together with extensive documentation, example Jupyter notebooks and references, can be found at: https://github.com/henrikbostrom/crepes

To install the package from the Python Package Index (PyPI)[6], you may type the following at a command prompt for your operating system, assuming that you already have installed `pip`[7]:

```
pip install crepes
```

This will install the most recent version of `crepes`[8] and allow you to import the package to your Python programs, Jupyter notebooks, etc. Below, we show how this may be done, directly after importing the standard package NumPy[9] and classes from scikit-learn[10] that we will use in the examples. From the main package `crepes`, we below import the two classes `ConformalRegressor` and `ConformalPredictiveSystem` that will be explained below, together with some useful functions from the separate package `crepes.fillings`, that also will be explained below.

```
import numpy as np

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml

from crepes import ConformalRegressor, ConformalPredictiveSystem

from crepes.fillings import (sigma_variance,
                             sigma_variance_oob,
                             sigma_knn,
                             binning)
```

3. Generating Conformal Regressors

3.1. Splitting data and training an underlying model

Let us first import a dataset, e.g., from www.openml.org, and normalize the targets; the latter is not really necessary, but useful, e.g., when evaluating the efficiency.

6. https://pypi.org/
7. https://pip.pypa.io/en/stable/
8. https://pypi.org/project/crepes/
9. https://numpy.org/
10. https://scikit-learn.org

```
dataset = fetch_openml(name="house_sales",version=3)

X = dataset.data.values.astype(float)
y = dataset.target.values.astype(float)

y = np.array([(y[i]-y.min())/(y.max()-y.min()) for i in range(len(y))])
```

Now let us split the dataset into a training and a test set, the former into a proper training and a calibration set, and fit a random forest with 500 trees on the proper training set.

```
X_training, X_test, y_training, y_test = \
    train_test_split(X, y, test_size=0.5)

X_proper_training, X_calibration, y_proper_training, y_calibration = \
    train_test_split(X_training, y_training, test_size=0.25)

learner_proper = RandomForestRegressor(n_jobs=-1, n_estimators=500)

learner_proper.fit(X_proper_training, y_proper_training)
```

3.2. Standard conformal regressors

In this section, we will see how to create, fit and make predictions with a standard (inductive/split) conformal regressor. We start out by creating a `ConformalRegressor` object.

```
cr_standard = ConformalRegressor()
```

We may display the object, e.g., to see whether it has been fitted or not; this particular information is stored in `cr_standard.fitted`.

```
display(cr_standard)
```

```
ConformalRegressor(fitted=False)
```

To fit the conformal regressor, we need the residuals from the calibration set.

```
residuals_calibration = y_calibration - learner_proper.predict(
    X_calibration)
```

Now let us actually fit the conformal regressor.

```
cr_standard.fit(residuals=residuals_calibration)
```

```
ConformalRegressor(fitted=True, normalized=False, mondrian=False)
```

We may now obtain prediction intervals, from the point predictions for the test set; here using a confidence level of 99%. The output is a NumPy array, where the two columns correspond to the lower and upper bounds of the intervals.

```
y_hat_proper = learner_proper.predict(X_test)

intervals = cr_standard.predict(y_hat=y_hat_proper, confidence=0.99)

display(intervals)
```

```
array([[-0.03033002,  0.11264116],
       [-0.04366944,  0.09930174],
       ...,
       [-0.04747455,  0.09549663],
       [-0.03782214,  0.10514904]])
```

We may request that the intervals are cut to exclude impossible values, in this case below 0 and above 1; below we also use the default confidence level (95%), which further tightens the intervals.

```
intervals_standard = cr_standard.predict(y_hat=y_hat_proper,
                                         y_min=0, y_max=1)

display(intervals_standard)
```

```
array([[0.01107024, 0.07124089],
       [0.        , 0.05790147],
       ...,
       [0.        , 0.05409637],
       [0.00357813, 0.06374878]])
```

3.3. Normalized conformal regressors

The above intervals are not normalized, i.e., they are all of the same size (at least before they are cut). We could make the intervals more informative through normalization using difficulty estimates; more difficult instances will be assigned wider intervals. K-nearest neighbors have been frequently used for this purpose, see e.g., (Papadopoulos et al., 2011; Johansson et al., 2014). `crepes.fillings` implements one such procedure through the helper function `sigma_knn`, which estimates the difficulty by the mean absolute errors of the k-nearest neighbors to each instance in the calibration set[11]. A small value (beta) is added to the estimates, which may be given through a (named) parameter to the function; in the call to the function below, we just use the default, i.e., `beta=0.01`, together with the default for k, i.e., `k=5`.

```
sigmas_calibration_knn = sigma_knn(X=X_calibration,
                                   residuals=residuals_calibration)
```

After having obtained the difficulty estimates, we create and fit the normalized (inductive/split) conformal regressor in just one step.

11. The scikit-learn implementation of the nearest neighbor algorithm is employed, after first having transformed the features using min-max normalization.

```
cr_normalized_knn = ConformalRegressor().fit(
    residuals=residuals_calibration, sigmas=sigmas_calibration_knn)
```

To generate prediction intervals for the test set, we need difficulty estimates for the latter too, which we get using the calibration objects and residuals.

```
sigmas_test_knn = sigma_knn(
    X=X_calibration, residuals=residuals_calibration, X_test=X_test)

intervals_normalized_knn = cr_normalized_knn.predict(
    y_hat=y_hat_proper, sigmas=sigmas_test_knn, y_min=0, y_max=1)
```

In case we have trained an ensemble model, like a `RandomForestRegressor`, we may alternatively use the helper function `sigma_variance`, which estimates the difficulty by the variance of the predictions of the constituent models, as investigated in (Boström et al., 2017). The function requires the trained model `learner` to be provided as input, assuming that `learner.estimators_` is a collection of base models, each implementing the `predict` method; this holds e.g., for `RandomForestRegressor`. Note that in contrast to `sigma_knn`, the residuals are not used here for difficulty estimation.

```
sigmas_calibration_var = sigma_variance(X=X_calibration,
                                        learner=learner_proper)

cr_normalized_var = ConformalRegressor().fit(
    residuals=residuals_calibration, sigmas=sigmas_calibration_var)
```

The difficulty estimates for the test set are generated in the same way; they are again needed to generate the normalized prediction intervals.

```
sigmas_test_var = sigma_variance(X=X_test, learner=learner_proper)

intervals_normalized_var = cr_normalized_var.predict(
    y_hat=y_hat_proper, sigmas=sigmas_test_var, y_min=0, y_max=1)
```

3.4. Mondrian Conformal Regressors

An alternative way of generating prediction intervals of varying size is to divide the object space into non-overlapping so-called Mondrian categories. A Mondrian conformal regressor (Boström and Johansson, 2020) is formed by providing the names of the categories as an additional parameter, named `bins`, for the `fit` method.

Here we employ the helper function `binning`, imported from `crepes.fillings`, which given a list/array of values assigns the values into named bins. If the optional parameter `bins` is an integer, the function will divide the values into equal-sized bins and return both the assigned bins and the bin boundaries. If `bins` instead is a set of bin boundaries, the function will just return the assigned bins.

We can form the Mondrian categories in almost any way we like (as long as we do not use the labels), and here we will use binning of the difficulty estimates.

```
bins_calibration, bin_borders = binning(values=sigmas_calibration_knn,
                                        bins=20)

cr_mondrian = ConformalRegressor().fit(residuals=residuals_calibration,
                                       bins=bins_calibration)
```

Let us now obtain the categories for the test instances using the same Mondrian categorization, i.e., bin borders.

```
bins_test = binning(values=sigmas_test_knn, bins=bin_borders)
```

Now we can form prediction intervals for the test instances.

```
intervals_mondrian = cr_mondrian.predict(
    y_hat=y_hat_proper,bins=bins_test, y_min=0, y_max=1)
```

3.5. Standard conformal regressors with out-of-bag calibration

For learners that employ bagging, like random forests, we may, alternatively to dividing the original training set into a proper training and calibration set, use the out-of-bag (OOB) predictions, which allow us to use the full training set for both model building and calibration (Johansson et al., 2014). Since the full training set can be used, the underlying model tends to be stronger, compared to a model trained on the proper training set only. On the other hand, since approximately 63.2% of the ensemble members are left out for each OOB prediction, the observed absolute residuals tend to be larger than what would be expected from the full model on an independent test set, which hence leads to conservative prediction intervals.

Let us first generate a model from the full training set and then get the residuals using the OOB predictions, assuming that the learner has an attribute `oob_prediction_`, which e.g. is the case for a `RandomForestRegressor` if `oob_score` is set to `True` when created.

```
learner_full = RandomForestRegressor(n_jobs=-1, n_estimators=500,
                                     oob_score=True)

learner_full.fit(X_training, y_training)

residuals_oob = y_training - learner_full.oob_prediction_
```

We may now obtain a standard conformal regressor from these OOB residuals.

```
cr_standard_oob = ConformalRegressor().fit(residuals=residuals_oob)
```

The regressor now allows for applying it using the point predictions of the full model.

```
y_hat_full = learner_full.predict(X_test)

intervals_standard_oob = cr_standard_oob.predict(y_hat=y_hat_full,
                                                 y_min=0, y_max=1)
```

3.6. Normalized conformal regressors with out-of-bag calibration

We may also generate normalized conformal regressors from the OOB predictions. The helper function `sigma_knn` can just as well be used together with the OOB residuals.

```
sigmas_oob_knn = sigma_knn(X=X_training, residuals=residuals_oob)

cr_normalized_knn_oob = ConformalRegressor().fit(
    residuals=residuals_oob, sigmas=sigmas_oob_knn)
```

If we want to use variance as a difficulty estimate, we may use the helper function `sigma_variance_oob`, which as before requires that `learner.oob_prediction_` and `learner.estimators_` have been defined. The default value for `beta` (`= 0.01`) is used again.

```
sigmas_oob_var = sigma_variance_oob(X=X_training, learner=learner_full)

cr_normalized_var_oob = ConformalRegressor().fit(
    residuals=residuals_oob, sigmas=sigmas_oob_var)
```

In order to apply the normalized OOB regressors to the test set, we need to generate difficulty estimates for the latter in the same way.

```
sigmas_test_knn_oob = sigma_knn(X=X_training, residuals=residuals_oob,
                                X_test=X_test)

intervals_normalized_knn_oob = cr_normalized_knn_oob.predict(
    y_hat=y_hat_full, sigmas=sigmas_test_knn_oob, y_min=0, y_max=1)

sigmas_test_var_oob = sigma_variance_oob(X=X_test,
                                         learner=learner_full)

intervals_normalized_var_oob = cr_normalized_var_oob.predict(
    y_hat=y_hat_full, sigmas=sigmas_test_var_oob, y_min=0, y_max=1)
```

3.7. Mondrian conformal regressors with out-of-bag calibration

We may form the categories using the difficulty estimates obtained from the OOB predictions, find the categories for the test instances and finally, generate the prediction intervals:

Table 2: Coverage and size of prediction intervals

	Coverage	**Mean size**	**Median size**
Standard CR	0.9483	0.0589	0.0602
Standard OOB CR	0.9471	0.0584	0.0597
Norm. CR knn	0.9487	0.0522	0.0467
Norm. OOB CR knn	0.9486	0.0514	0.0454
Norm. CR var	0.9475	0.0555	0.0546
Norm. OOB CR var	0.9469	0.0551	0.0542
Mondrian CR	0.9529	0.0586	0.0436
Mondrian OOB CR	0.9521	0.0566	0.0451
Mean	0.9490	0.0558	0.0512

```
bins_oob, bin_borders_oob = binning(values=sigmas_oob_knn, bins=20)

cr_mondrian_oob = ConformalRegressor().fit(residuals_oob,
                                           bins=bins_oob)

bins_test_oob = binning(values=sigmas_test_knn_oob,
                        bins=bin_borders_oob)

intervals_mondrian_oob = cr_mondrian_oob.predict(
    y_hat_full, bins=bins_test_oob, y_min=0, y_max=1)
```

3.8. Investigating the prediction intervals

We may now take a look at the coverage of the prediction intervals for the test instances, i.e., the fraction of the intervals containing the true target, as output by the eight conformal regressors. In Table 2, the coverage as well as the mean and median interval sizes are shown.

For a more detailed view of the distribution of the interval sizes, Fig. 1 plots the cumulative probability for various interval sizes for the eight different regressors.[12]

Fig. 2 shows how the prediction intervals vary with the value predicted by the underlying model, for the four OOB variants, where the red lines indicate the lower and upper bounds of the prediction intervals, the blue dots indicate the true targets and the yellow dots indicate the predicted values, the latter in all cases located at the center of the prediction intervals.

12. Code for generating the tables and plots are provided in Jupyter notebooks at https://github.com/henrikbostrom/crepes.

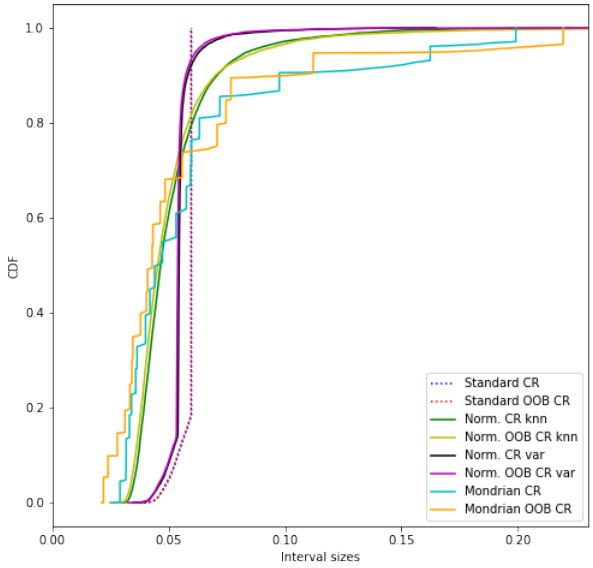

Figure 1: Interval sizes for the eight conformal regressors

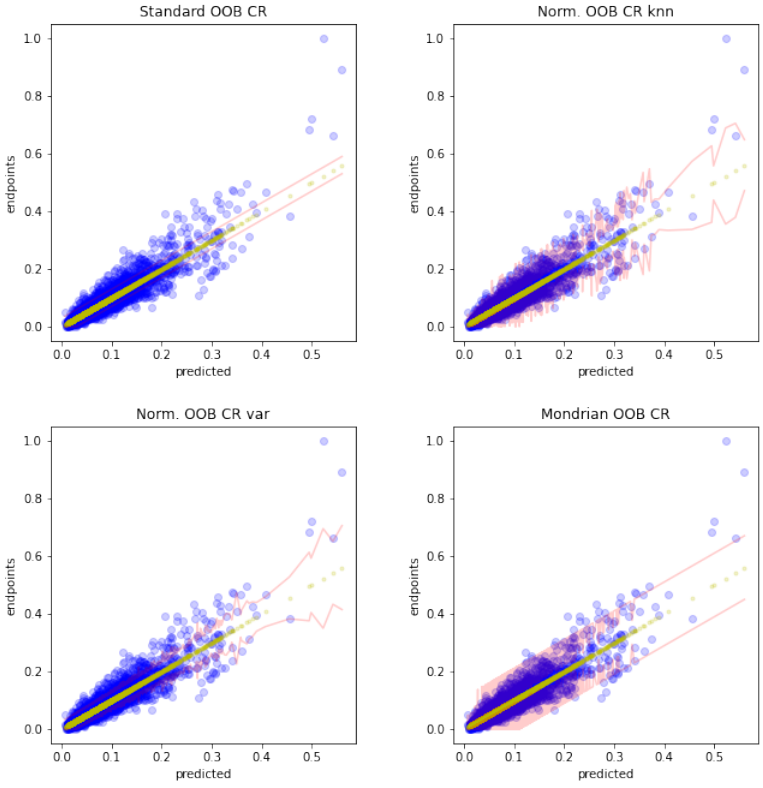

Figure 2: Predicted intervals for the OOB conformal regressors

Table 3: Result of evaluating the Mondrian OOB conformal regressor on all metrics and three confidence levels

Metric/Conf. level	**0.90**	**0.95**	**0.99**
`error`	0.0993	0.0479	0.0093
`efficiency`	0.0419	0.0566	0.0955
`time_fit`	0.0010	0.0010	0.0010
`time_evaluate`	0.0011	0.0007	0.0006

3.9. Evaluating the conformal regressors

The class `ConformalRegressor` includes a method `evaluate`, which has the following parameters:

```
evaluate(y_hat, y, sigmas, bins, confidence, y_min, y_max, metrics)
```

The first two parameters, which correspond to the predicted and actual targets, respectively, are mandatory for all conformal regressors, while `sigmas` and `bins` are required only for normalized and Mondrian conformal regressors, respectively. The confidence level to use is determined by the optional parameter `confidence`, which defaults to 0.95. The parameter `metrics` should include a subset of the values `["error","efficiency","time_fit","time_evaluate"]`, and defaults to the full set. The first two metrics correspond to the fraction of true targets that are not included in the prediction intervals, which should be close to the chosen confidence level, and the mean size of the prediction intervals. The last two correspond to the wall clock time in seconds[13] for fitting and making predictions with the conformal regressor. The output of the method is a dictionary, with one key for each value `metrics`, together with the corresponding measurement. Using this method on the previously generated Mondrian OOB conformal regressor, using three confidence levels, gives the output that are summarized in Table 4. It can be noted that the computation times are practically negligible even in this case when more than 10 000 instances are used for fitting and testing, respectively. This is mainly due to the use of the highly optimized NumPy library in the `crepes` package, together with the fact that all the training and application of the underlying model are done prior to calling the methods of the package.

4. Generating Conformal Predictive Systems

The methods of the class `ConformalPredictiveSystem` are named identically as for the class `ConformalRegressor`, and they mainly share also the same set of parameters, with a few notable differences for the `predict` and `evaluate` methods. The creation of conformal predictive systems is hence very similar to that of conformal regressors. Below, we first illustrate the generation of standard and normalized conformal predictive systems, using the same residuals and difficulty estimates that were introduced in the previous section.

13. as measured using the function `time()` in the standard library `time`; the measurements have been obtained using a machine equipped with a i9-11950H processor, 32 GB RAM and Ubuntu 20.04.

```
cps_standard = ConformalPredictiveSystem()
cps_standard.fit(residuals=residuals_calibration)

cps_normalized = ConformalPredictiveSystem()
cps_normalized.fit(residuals=residuals_calibration,
                   sigmas=sigmas_calibration_knn)

cps_standard_oob = ConformalPredictiveSystem()
cps_standard_oob.fit(residuals=residuals_oob)

cps_normalized_oob = ConformalPredictiveSystem()
cps_normalized_oob.fit(residuals=residuals_oob, sigmas=sigmas_oob_knn)
```

The generation of Mondrian conformal predictive systems is again very similar to that of Mondrian conformal regressors. Below, we generate Mondrian normalized conformal predictive systems, where the categories are formed by binning the predictions of the underlying model, as investigated in (Boström et al., 2021).

```
cps_bins_calibration, cps_bin_borders = binning(
    values=learner_proper.predict(X_calibration), bins=5)

cps_mondrian = ConformalPredictiveSystem().fit(
    residuals=residuals_calibration, sigmas=sigmas_calibration_knn,
    bins=cps_bins_calibration)

cps_bins_oob, cps_bin_borders_oob = binning(
    values=learner_full.oob_prediction_, bins=5)

cps_mondrian_oob = ConformalPredictiveSystem().fit(
    residuals=residuals_oob, sigmas=sigmas_oob_knn,
    bins=cps_bins_oob)
```

The `predict` method includes the following parameters:

```
predict(y_hat, sigmas, bins, y, lower_percentiles, higher_percentiles,
        y_min, y_max, return_cpds, cpds_by_bins)
```

The method can be used to obtain both cumulative probabilities for given values or percentiles from the conformal predictive distributions (or both at the same time). The parameter `y` specifies zero (the default), a single value or a list/array of values of the same length as `y_hat`, for which cumulative probabilities should be output. The parameters `lower_percentiles` and `higher_percentiles` specify what percentiles should be returned from the conformal predictive distributions. Each of the two may be an empty list (the default), which means that no percentile should be returned, a single value or a list/array of values. For the former, in case a percentile lies between two values, the lower value will be returned, similar to the option `interpolation="lower"` in `numpy.percentile`, while

the latter instead results in that the higher of the two values is returned, corresponding to the option `interpolation="higher"` in the same package.

The parameter setting `return_cpds=True`, will make the method output the actual conformal predictive distributions (the default is `return_cpds=False`). The format of the distributions vary with the type of conformal predictive system; for a standard and normalized CPS, the output is an array with a row for each test instance and a column for each calibration instance (residual), while for a Mondrian CPS, the default output is a vector containing one CPD per test instance (since the number of values may vary between categories). If the desired output instead is an array of distributions per category, where all distributions in a category have the same number of columns, which in turn depends on the number of calibration instances in the corresponding category, then `cpds_by_bins=True` may be specified (the default is `cpds_by_bins=False`). In case `return_cpds=True` is specified together with `y`, `lower_percentiles` or `higher_percentiles`, the output of `predict` will be a pair, with the first element holding the results of the above type and the second element will contain the CPDs.

The remaining parameters are exactly the same as for conformal regressors. For example, to obtain (conservative) prediction intervals from a conformal predictive system, here without having specified any bins and bounds of the intervals, the method may be called in the following way:

```
predict(y_hat, sigmas, lower_percentiles=2.5, higher_percentiles=97.5)
```

If we instead are interested in obtaining p-values for a set of target values, e.g., the true target values in a test set, we may call the method in this way:

```
predict(y_hat, sigmas, y=y_test)
```

We may also combine the above two usages in one call, as here illustrated for the normalized OOB conformal predictive system:

```
cps_normalized_oob.predict(y_hat=y_hat_full, sigmas=sigmas_test_knn_oob,
                           y=y_test,
                           lower_percentiles=2.5, higher_percentiles=97.5,
                           y_min=0, y_max=1)
```

```
array([[0.96631909, 0.02006038, 0.06364274],
       [0.75770344, 0.00562086, 0.05369188],
       [0.30226483, 0.03387309, 0.10665441],
       ...,
       [0.02202959, 0.07538331, 0.11967962],
       [0.18542882, 0.00578612, 0.04518795],
       [0.19365324, 0.01722526, 0.05433577]])
```

The output array has three columns in this case; the first contains the p-values, while the second and third contain the lower and higher percentiles, for each test instance. If we

are interested in plotting the conformal predictive distributions, we may obtain the full distributions from the `predict` method by specifying setting `return_cpds=True`.

We may also plot the conformal predictive distribution for some test object. In case the calibration set is very large, one may consider plotting an approximation of the full distribution by using a grid of values for `lower_percentiles` or `higher_percentiles`, instead of setting `return_cpds=True`. For the Mondrian CPS, the size of the calibration set for each bin is reasonable in this case, so we may just plot the distributions directly. In Fig. 3, we show the result of plotting the full conformal predictive distribution for a randomly selected test instance, using the Mondrian normalized OOB conformal predictive system.

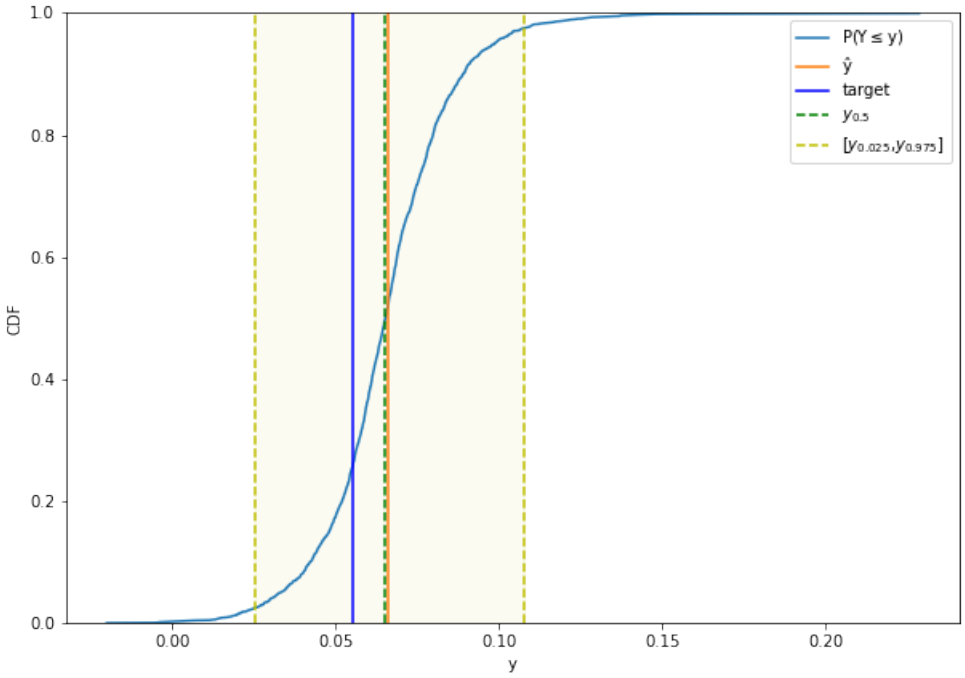

Figure 3: Conformal predictive distribution for a randomly selected test instance

Fig. 4 shows how the interval sizes are distributed for the above six conformal predictive systems, when applied to the test set with the lower and higher percentiles of 2.5 and 97.5, respectively. Since normalization is performed within each Mondrian category, the resulting curves are smoother than the ones we saw earlier for the Mondrian conformal regressors.

Table 4: Result of evaluating the Mondrian normalized OOB conformal predictive system on all metrics and three confidence levels

Metric/Conf. level	**0.90**	**0.95**	**0.99**
`error`	0.0944	0.0476	0.0094
`efficiency`	0.0417	0.0532	0.0895
`CRPS`	0.0068	0.0068	0.0068
`time_fit`	0.0009	0.0009	0.0009
`time_evaluate`	0.3553	0.3503	0.3460

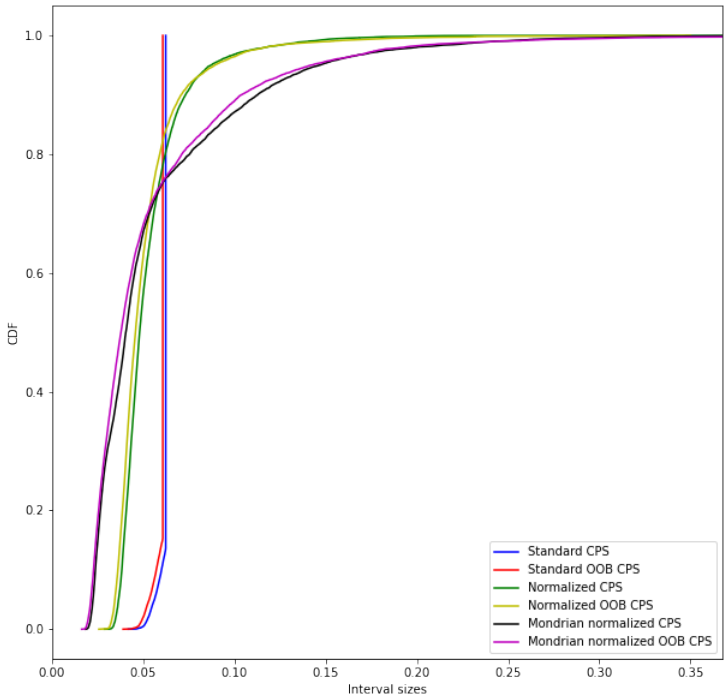

Figure 4: Interval sizes for six conformal predictive systems

The `evaluate` method for a `ConformalPredictiveSystem` behaves similarly to the corresponding method of conformal regressors, with the difference that the set of metrics now also includes `"CRPS"`, i.e., continuous-ranked probability score (Vovk et al., 2020).

Compared to when evaluating conformal regressors, the evaluation time is no longer negligible; the cost of calculating CRPS grows linearly with the size of the calibration set and can become quite costly, in particular for the OOB approaches which use the full training

set to obtain calibration scores. This is partly remedied by the Mondrian approaches, since they partition the calibration set according to the categories.

5. Concluding remarks

It has been demonstrated how the recently released Python package `crepes` can be used to generate, apply and evaluate standard, normalized and Mondrian conformal regressors and predictive systems. It has been shown how functions from the accompanying package `crepes.fillings` can be utilized for estimating difficulty and forming Mondrian categories. The main package is completely model-agnostic, while some functionality of the latter package is tailored for models that follow conventions of the scikit-learn package. Since the training and application of the underlying regression models are kept separate from generating the conformal regressors and predictive systems, the latter can be generated and applied at very low computational cost. The only major cost comes from evaluating conformal predictive systems using continuous-ranked probability score (CRPS), which still can be done efficiently, thanks to the extensive use of the highly optimized NumPy library, allowing 10 000 test instances to be evaluated with respect to CRPS using conformal predictive distributions formed from 10 000 calibration instances within a second on a modern laptop.

There are several directions for future developments of the package. One direction concerns targeting transductive conformal regressors and predictive systems, which would avoid the need for setting aside a separate calibration set, even for models for which forming out-of-bag predictions is not an option, at the cost of substantially increased computation time. Related approaches here are also the jackknife+ (Barber et al., 2021) and jackknife+-after-bootstrap (Kim et al., 2020), which have been implemented in the MAPIE package. These in turn also relate to other approaches to form conformal predictors through aggregation (Linusson et al., 2017), something which is currently not included in the `crepes` package. Another main direction for extending the package concerns including a wider set of helper functions with more options for difficulty estimation and forming Mondrian categories.

Acknowledgements

HB was partly funded by the Swedish Foundation for Strategic Research (CDA, grant no. BD15-0006) and Vinnova (RAPIDS, grant no. 2021-02522).

References

Rina Foygel Barber, Emmanuel J Candes, Aaditya Ramdas, and Ryan J Tibshirani. Predictive inference with the jackknife+. *The Annals of Statistics*, 49(1):486–507, 2021.

Henrik Boström and Ulf Johansson. Mondrian conformal regressors. In *Proceedings of the Ninth Symposium on Conformal and Probabilistic Prediction and Applications*, pages 114–133, 2020.

Henrik Boström, Henrik Linusson, Tuve Löfström, and Ulf Johansson. Accelerating difficulty estimation for conformal regression forests. *Ann. Math. Artif. Intell.*, 81(1-2): 125–144, 2017.

Henrik Boström, Ulf Johansson, and Tuwe Löfström. Mondrian conformal predictive distributions. In *Proceedings of the Tenth Symposium on Conformal and Probabilistic Prediction and Applications*, pages 24–38, 2021.

Ulf Johansson, Henrik Boström, Tuve Löfström, and Henrik Linusson. Regression conformal prediction with random forests. *Machine Learning*, 97(1-2):155–176, 2014. ISSN 0885-6125.

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.

Henrik Linusson, Ulf Norinder, Henrik Boström, Ulf Johansson, and Tuve Löfström. On the calibration of aggregated conformal predictors. In *Conformal and probabilistic prediction and applications*, pages 154–173. PMLR, 2017.

Harris Papadopoulos, Kostas Proedrou, Volodya Vovk, and Alexander Gammerman. Inductive confidence machines for regression. In *Proc. of the 13th European Conference on Machine Learning*, volume 2430 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2002.

Harris Papadopoulos, Vladimir Vovk, and Alexander Gammerman. Regression conformal prediction with nearest neighbours. *Journal of Artificial Intelligence Research*, pages 815–840, 2011.

Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*. Springer-Verlag New York, Inc., 2005.

Vladimir Vovk, Ivan Petej, Ilia Nouretdinov, Valery Manokhin, and Alexander Gammerman. Computationally efficient versions of conformal predictive distributions. *Neurocomputing*, 397:292–308, 2020.