Proceedings of Machine Learning Research 230:1–14, 2024 Conformal and Probabilistic Prediction with Applications

Conformal Prediction in Python with `crepes`

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

Editor: Simone Vantini, Matteo Fontana, Aldo Solari, Henrik Boström and Lars Carlsson

Abstract

`crepes` is a Python package for conformal prediction, which has been extended in several ways since its introduction. While the original version of the package focused on conformal regressors and predictive systems, the current version also includes conformal classifiers. New classes and methods for computing non-conformity scores and Mondrian categories have also been incorporated. Moreover, the package has been extended to allow for seamless embedding of classifiers and regressors in the conformal prediction framework; instead of generating conformal predictors that are separate from the learners, the latter can now be equipped with specific prediction methods that in addition to providing point predictions also can generate p-values, prediction sets and intervals, as well as conformal predictive distributions. Extensive documentation for the package has furthermore been developed. In this paper, these extensions are described, as implemented in `crepes`, version 0.7.0.

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

1. Introduction

The Python package `crepes` implements selected techniques from the field of conformal prediction (Vovk et al., 2005). The focus of the first version of the package, presented in (Boström, 2022), was on conformal regressors (Papadopoulos et al., 2002) and predictive systems (Vovk et al., 2020), through which point predictions are turned into prediction intervals by the former and into cumulative distribution functions (conformal predictive distributions) by the latter, in both cases employing the inductive approach, i.e., using a calibration set for which residuals between actual and predicted values are computed. Both standard and normalized conformal regressors and predictive systems were implemented, where the latter employ object-specific difficulty (quality) estimates, and the former uses the residuals only. In addition, the package also implemented Mondrian conformal regressors (Boström and Johansson, 2020) and Mondrian predictive systems (Boström et al., 2021), which both rely on partitioning the calibration and test objects into so-called Mondrian categories. Moreover, already in the first version of the package, there was an option for using out-of-bag predictions for calibration (Johansson et al., 2014).

Since its inception, `crepes` has been extended in several ways, most notably by the inclusion of conformal classifiers (Vovk et al., 2005). Another important addition concerns the coupling of the underlying model (which is used for making point predictions) and the conformal predictor generated on top of the former. In the original package, the underlying model was separated from the conformal predictor; objects belonging to the classes `ConformalRegressor` and `ConformalPredictiveSystem` were fitted using the output of the

© 2024 H. Boström.

model, and in order to make predictions for a test set, point predictions for this had to be provided. Although this gives quite some flexibility, e.g., the underlying model can have any format and does not have to be located at the same place as the conformal predictor, it is not always ideal from a usability perspective, as the user will need to make the necessary arrangements to keep the conformal predictor aligned with the underlying model. While this option is still available, now also for the new class `ConformalClassifier`, the classes `WrapRegressor` and `WrapClassifier` have been added, which allow for seamlessly extending the object containing the underlying model with a `calibrate` method for fitting the conformal predictor, and with the methods `predict_p` and `predict_set` to output p-values and prediction sets, respectively, `predict_int` to output prediction intervals, and `predict_cps` to output conformal predictive distributions.

In this paper, we give an overview of the above extensions. Examples will be given from using the package in conjunction with a separate module, called `crepes.extras`, which implements several options for defining difficulty estimates and Mondrian categories. In the next section, we describe how to install the package; in addition to installing from PyPI, which was the only option for the first version, the package can now also be installed from Anaconda. In Section 3, we show how to embed classifiers to obtain conformal classifiers, and in Section 4, we show how to embed regressors to obtain conformal regressors and predictive systems, while also illustrating new functionality for computing non-conformity scores and Mondrian categories. In Section 5, we give some examples of the extensive documentation of the package, which was non-existent for the first version, and finally, in section 6, we summarize the current implementation and outline some directions for further developments.

2. Installing crepes

The source code of the `crepes` package, which is licensed under the permissive BSD-3-Clause license, together with examples, Jupyter notebooks and references, can be found at GitHub,[1] while the documentation of the package can be found at Read*the*Docs.[2]

To install the package from the Python Package Index (PyPI)[3], the following command should be provided at a command prompt, assuming that `pip`[4] has already been installed:

```
pip install crepes
```

The package may alternatively be installed from the `conda-forge` channel of Anaconda[5]. Assuming `conda` has been installed[6], `crepes` may be installed by:

```
conda install conda-forge::crepes
```

1. https://github.com/henrikbostrom/crepes
2. https://crepes.readthedocs.io
3. https://pypi.org/project/crepes/
4. https://pip.pypa.io
5. https://anaconda.org/conda-forge/crepes
6. https://www.anaconda.com/download

3. Wrapping Classifiers

Let us illustrate how we may use `crepes` to generate and apply conformal classifiers with a dataset from `openml`[7], namely `qsar-biodeg`, which contains 1055 chemicals represented by 41 features (molecular descriptors) and the task is to classify the chemicals as ready or not ready biodegradable. We first split the dataset into a training and a test set using `train_test_split` from `scikit-learn`,[8] and then further split the training set into a proper training and a calibration set:

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

dataset = fetch_openml(name="qsar-biodeg", parser="auto")

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

X_prop_train, X_cal, y_prop_train, y_cal = train_test_split(X_train, y_train,
                                                            test_size=0.25)
```

We will first embed, or "wrap" in the terminology of `crepes`, a random forest classifier from `scikit-learn`, fit it to the proper training set, and fit a standard conformal classifier through the `calibrate` method:

```
from crepes import WrapClassifier
from sklearn.ensemble import RandomForestClassifier

rf = WrapClassifier(RandomForestClassifier(n_jobs=-1))

rf.fit(X_prop_train, y_prop_train)

rf.calibrate(X_cal, y_cal)
```

The wrapped classifier will largely act as the non-wrapped classifier in that the `fit`, `predict` and `predict_proba` methods will behave in the same way. The learner, which in the above example can be accessed by `rf.learner`, may be fitted before as well as after being wrapped.

In addition to generating standard point predictions for the test set through the methods `predict` and `predict_proba`, we may also obtain p-values using `predict_p`; these are provided in the form of a `NumPy` array with as many rows as there are test objects and as many columns as there are classes (two in this case), where the latter are ordered according to `classes_` of the underlying learner:

7. https://www.openml.org
8. https://scikit-learn.org

```
rf.predict_p(X_test)
```

```
array([[0.00427104, 0.74842304],
       [0.07874355, 0.2950549 ],
       [0.50529983, 0.01557963],
       ...,
       [0.8413356 , 0.00201167],
       [0.84402215, 0.00654927],
       [0.29601955, 0.07766093]])
```

We can also get prediction sets, represented by binary vectors indicating presence (1) or absence (0) of the class labels that correspond to the columns, here at the 90% confidence level, using `predict_set`:[9]

```
rf.predict_set(X_test, confidence=0.9)
```

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

Since we have access to the true class labels for the test set, we can evaluate the conformal classifier using the `evaluate` method. Unless we specify a list of metrics to use, through the argument `metrics`, all of them will be used by default, where `error` is the fraction of prediction sets not containing the true class label, `avg_c` is the average number of predicted class labels, `one_c` is the fraction of singleton prediction sets, `empty` is the fraction of empty prediction sets, `time_fit` is the time taken to fit the conformal classifier (not including the time to fit the underlying learner), and `time_evaluate` is the time taken for the evaluation.

```
rf.evaluate(X_test, y_test, confidence=0.99)
```

```
{'error': 0.005681818181818232,
 'avg_c': 1.691287878787879,
 'one_c': 0.3087121212121212,
 'empty': 0.0,
 'time_fit': 2.3365020751953125e-05,
 'time_evaluate': 0.017678260803222656}
```

To control the error level across different groups of objects of interest, we may use so-called Mondrian conformal classifiers. A Mondrian conformal classifier if formed by providing a function or a `MondrianCategorizer` (defined in `crepes.extras`) as an additional argument, named `mc`, for the calibrate method.

9. The columns are ordered according to the instance variable `classes_` of the underlying learner, which in this case is `rf.learner.classes_`.

For illustration, we will use the predicted labels of the underlying model to form the categories. Note that the prediction sets for the test objects are obtained using the same categorization as for the calibration objects.

```
rf_mond = WrapClassifier(rf.learner)

rf_mond.calibrate(X_cal, y_cal, mc=rf_mond.predict)

rf_mond.predict_set(X_test)
```

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

We may also form the categories using a `MondrianCategorizer`, which may be fitted in several different ways. Below we show how to form categories by equal-sized binning of the first feature value, using five bins (instead of the default which is 10); note that we need objects to get the threshold values for the categories (bins).

```
from crepes.extras import MondrianCategorizer

def get_values(X):
    return X[:,0]

mc = MondrianCategorizer()
mc.fit(X_cal, f=get_values, no_bins=5)

rf_mond = WrapClassifier(rf.learner)
rf_mond.calibrate(X_cal, y_cal, mc=mc)

rf_mond.predict_set(X_test)
```

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

For learners that use bagging, like random forests, we may consider an alternative strategy to dividing the original training set into a proper training and calibration set; we may use the out-of-bag (OOB) predictions, which allow us to use the full training set for both model building and calibration. It should be noted that this strategy does not come with

the theoretical validity guarantee of the above (inductive) conformal classifiers, due to that calibration and test instances are not handled in exactly the same way. In practice, however, conformal classifiers based on out-of-bag predictions rarely fail to meet the coverage requirements.

Below we show how to enable this in conjunction with a specific type of Mondrian conformal classifier, a so-called class-conditional conformal classifier, which uses the class labels as Mondrian categories:

```
rf = WrapClassifier(RandomForestClassifier(n_jobs=-1, n_estimators=500, oob_score=True))

rf.fit(X_train, y_train)

rf.calibrate(X_train, y_train, class_cond=True, oob=True)

rf.evaluate(X_test, y_test, confidence=0.9)
```

```
{'error': 0.10795454545454541,
 'avg_c': 1.0984848484848484,
 'one_c': 0.9015151515151515,
 'empty': 0.0,
 'time_fit': 0.0001518726348876953,
 'time_evaluate': 0.06513118743896484}
```

4. Wrapping Regressors

Let us also see how `crepes` can be used to generate conformal regressors and predictive systems. Again, we import a dataset from `openml`, which we split into a training and a test set and then further split the training set into a proper training set and a calibration set:

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

dataset = fetch_openml(name="house_sales", version=3, parser="auto")

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

X_prop_train, X_cal, y_prop_train, y_cal = train_test_split(X_train, y_train,
                                                            test_size=0.25)
```

Let us now embed ("wrap") a `RandomForestRegressor` from `scikit-learn` using the class `WrapRegressor` from `crepes` and fit it (in the usual way) to the proper training set:

```
from sklearn.ensemble import RandomForestRegressor
from crepes import WrapRegressor

rf = WrapRegressor(RandomForestRegressor())
rf.fit(X_prop_train, y_prop_train)
```

We may now fit a conformal regressor using the calibration set through the `calibrate` method:

```
rf.calibrate(X_cal, y_cal)
```

The conformal regressor can now produce prediction intervals for the test set using the `predict_int` method, here using a confidence level of 99%:

```
rf.predict_int(X_test, confidence=0.99)
```

```
array([[   8260.53, 1065083.53],
       [ -54858.5 , 1001964.5 ],
       [  -7779.25, 1049043.75],
       ...,
       [ 297229.8 , 1354052.8 ],
       [-270260.  ,  786563.  ],
       [-185146.94,  871676.06]])
```

The output is a `NumPy` array with a row for each test instance, and where the two columns specify the lower and upper bound of each prediction interval.

We may request that the intervals are cut to exclude impossible values, in this case below 0, and if we also rely on the default confidence level (0.95), the output intervals will be a bit tighter:

```
rf.predict_int(X_test, y_min=0)
```

```
array([[ 288602.83,  784741.23],
       [ 225483.8 ,  721622.2 ],
       [ 272563.05,  768701.45],
       ...,
       [ 577572.1 , 1073710.5 ],
       [  10082.3 ,  506220.7 ],
       [  95195.36,  591333.76]])
```

The above intervals are not normalized, i.e., they are all of the same size (at least before they are cut). We could make them more informative through normalization using difficulty estimates; objects considered more difficult will be assigned wider intervals.

We will use a `DifficultyEstimator` from the `crepes.extras` module for this purpose.[10] Here we estimate the difficulty by the standard deviation of the target of the `k` (default `k=25`) nearest neighbors in the proper training set to each object in the calibration set. A small

10. For a description of the available options, as well as their implementations, the reader is referred to the documentation; https://crepes.readthedocs.io/en/latest/crepes.extras.html

value (beta) is added to the estimates, which may be given through an argument to the function; below we just use the default, i.e., `beta=0.01`.

We first fit the difficulty estimator and then calibrate the conformal regressor, using the calibration objects and labels together the difficulty estimator:

```
from crepes.extras import DifficultyEstimator

de = DifficultyEstimator()
de.fit(X_prop_train, y=y_prop_train)

rf.calibrate(X_cal, y_cal, de=de)
```

To obtain prediction intervals, we just have to provide test objects to the `predict_int` method, as the difficulty estimates will be computed by the incorporated difficulty estimator:

```
rf.predict_int(X_test, y_min=0)
```

```
array([[ 222036.82862012,  851307.23137988],
       [ 316413.83821721,  630692.16178279],
       [ 384784.44135415,  656480.05864585],
       ...,
       [ 110527.74801848, 1540754.85198152],
       [ 174799.94131735,  341503.05868265],
       [ 274305.55734858,  412223.56265142]])
```

Depending on the employed difficulty estimator, the normalized intervals may sometimes be unreasonably large, in the sense that they may be several times larger than any previously observed error. Moreover, if the difficulty estimator is uninformative, e.g., completely random, the varying interval sizes may give a false impression of that we can expect lower prediction errors for instances with tighter intervals. Ideally, a difficulty estimator providing little or no information on the expected error should instead lead to more uniformly distributed interval sizes.

A Mondrian conformal regressor can be used to address these problems, by dividing the object space into non-overlapping so-called Mondrian categories, and forming a (standard) conformal regressor for each category.

Similar to how a Mondrian conformal classifier is created, we may form a Mondrian conformal regressor by providing a function or a `MondrianCategorizer` through the argument `mc` for the calibrate method.

Here we employ a `MondrianCategorizer` and show how to form categories by binning of the difficulty estimates into 20 bins, using the difficulty estimator fitted above:

```
from crepes.extras import MondrianCategorizer

mc_diff = MondrianCategorizer()
mc_diff.fit(X_cal, de=de, no_bins=20)

rf.calibrate(X_cal, y_cal, mc=mc_diff)
```

When making predictions, the test objects will be assigned to Mondrian categories according to the incorporated `MondrianCategorizer` (or labeling function):

```
rf.predict_int(X_test, y_min=0)
```

```
array([[ 242624.89,  830719.17],
       [ 329358.5 ,  617747.5 ],
       [ 371028.  ,  670236.5 ],
       ...,
       [      0.  , 1730501.3 ],
       [ 157022.53,  359280.47],
       [ 266456.61,  420072.51]])
```

We can very easily switch from conformal regressors to conformal predictive systems. The latter produce cumulative distribution functions (conformal predictive distributions). From these we can generate prediction intervals, but we can also obtain percentiles, calibrated point predictions, as well as p-values for given target values. In order to consider conformal predictive systems (cps) instead of conformal regressors, we just have to provide `cps=True` to the `calibrate` method.

We can, for example, form normalized Mondrian conformal predictive systems, by providing both a Mondrian categorizer and difficulty estimator to the calibrate method. Here we will consider Mondrian categories formed from binning the point predictions:

```
mc_pred = MondrianCategorizer()
mc_pred.fit(X_cal, f=rf.predict, no_bins=5)

rf.calibrate(X_cal, y_cal, de=de, mc=mc_pred, cps=True)
```

We can now make predictions with the conformal predictive system, through the method `predict_cps`. The output of this method can be controlled quite flexibly; here we request prediction intervals with 95% confidence to be output:

```
rf.predict_cps(X_test, lower_percentiles=2.5, higher_percentiles=97.5, y_min=0)
```

```
array([[ 240114.65604157,  869014.03528742],
       [ 339706.24924814,  609239.58260891],
       [ 404920.87940518,  637934.16698199],
       ...,
       [      0.        , 1947549.10314688],
       [ 173038.55234664,  335836.19025193],
       [ 280187.36965593,  399290.04471503]])
```

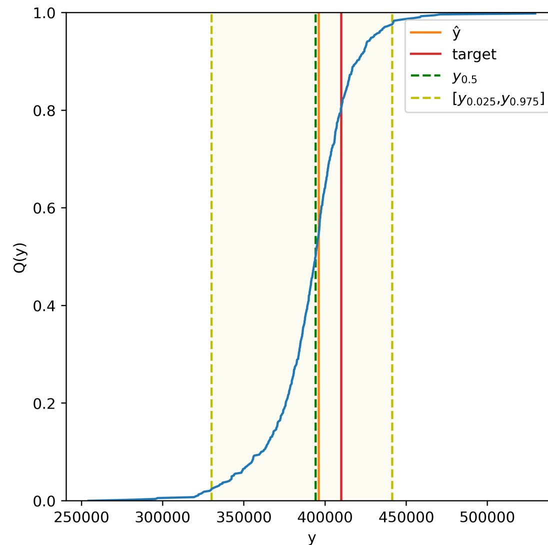

Figure 1: Conformal predictive distribution for a randomly selected test instance

If we would like to take a look at the p-values for the true targets (these can be expected to be uniformly distributed), we just have to provide these through the argument `y`:

```
rf.predict_cps(X_test, y=y_test)
```

```
array([0.38424814, 0.54023864, 0.28727364, ..., 0.35291685, 0.6110545,
       0.60037036])
```

We may request that the `predict_cps` method returns the full conformal predictive distribution (CPD) for each test instance, as defined by the threshold values, by setting `return_cpds=True`. 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.

```
cpds = rf.predict_cps(X_test, return_cpds=True)
```

The resulting vector of arrays is not displayed here, but we instead provide a plot for the CPD of a random test instance in Fig. 1.

class crepes.WrapClassifier(*learner*) [source]

A learner wrapped with a ConformalClassifier.

Methods

calibrate(X, y[, oob, class_cond, nc, mc])	Fit a ConformalClassifier using learner.
evaluate(X, y[, confidence, smoothing, metrics])	Evaluate ConformalClassifier.
fit(X, y, **kwargs)	Fit learner.
predict(X)	Predict with learner.
predict_p(X)	Obtain (smoothed) p-values using conformal classifier.
predict_proba(X)	Predict with learner.
predict_set(X[, confidence, smoothing])	Obtain prediction sets using conformal classifier.

Figure 2: High-level documentation for the class `WrapClassifier`

5. Documentation

One of the most important things lacking in the early versions of `crepes` was a proper documentation; this has now been addressed and the documentation can be found at https://crepes.readthedocs.io. In addition to installation instructions and example Jupyter notebooks, the classes and functions in the main `crepes` package as well as in the accompanying module `crepes.extras` are described and exemplified in the documentation. As an example, the highest level of documentation for the class `WrapClassifer` is shown in Fig. 2. Additional documentation can be found for the underlined keywords, e.g., `calibrate`; the detailed documentation for this method is shown in Fig. 3.

6. Concluding remarks

We have presented recent extensions to the Python package `crepes`, which can be used to generate, apply and evaluate standard and Mondrian conformal classifiers as well as standard, normalized and Mondrian conformal regressors and predictive systems. In addition to the original approach of keeping the conformal predictors separate from the underlying (point) predictors, the package now also allows for a tight integration of the learners and conformal predictors through wrapper objects.

There are several directions in which the package can be further developed. One straightforward such extension is to include additional functions for computing non-conformity scores. Incorporation of aggregation techniques, as described in (Linusson et al., 2017), is also a possibility. Related approaches that could be considered in this context are also the jackknife+ (Barber et al., 2021) and jackknife+-after-bootstrap (Kim et al., 2020). Another possible extension concerns techniques for handling covariate shift, e.g., through weighted conformal prediction (Tibshirani et al., 2019), and censored data (Boström et al., 2019, 2023; Candès et al., 2023).

calibrate(*X, y, oob=False, class_cond=False, nc=<function hinge>, mc=None*) [source]

Fit a `ConformalClassifier` using learner.

Parameters:

- **X** (*array-like of shape (n_samples, n_features),*) – set of objects
- **y** (*array-like of shape (n_samples,),*) – target values
- **oob** (*bool, default=False*) – use out-of-bag estimation
- **class_cond** (*bool, default=False*) – if class_cond=True, the method fits a Mondrian `ConformalClassifier` using the class labels as categories
- **nc** (function, default = `crepes.extras.hinge()`) – function to compute non-conformity scores
- **mc** (function or `crepes.extras.MondrianCategorizer`, default=None) – function or `crepes.extras.MondrianCategorizer` for computing Mondrian categories

Returns:

self – Wrap object updated with a fitted `ConformalClassifier`

Return type:

object

Examples

Assuming `X_cal` and `y_cal` to be an array and vector, respectively, with objects and labels for the calibration set, and `w` is a `WrapClassifier` object for which the learner has been fitted, a standard conformal classifier can be formed by:

```
w.calibrate(X_cal, y_cal)
```

Assuming that `get_categories` is a function that returns a vector of Mondrian categories (bin labels), a Mondrian conformal classifier can be generated by:

```
w.calibrate(X_cal, y_cal, mc=get_categories)
```

By providing the option `oob=True`, the conformal classifier will be calibrating using out-of-bag predictions, allowing the full set of training objects (`X_train`) and labels (`y_train`) to be used, e.g.,

```
w.calibrate(X_train, y_train, oob=True)
```

By providing the option `class_cond=True`, a Mondrian conformal classifier will be formed using the class labels as categories, e.g.,

```
w.calibrate(X_cal, y_cal, class_cond=True)
```

Note

Any Mondrian categorizer specified by the `mc` argument will be ignored by `calibrate()`, if `class_cond=True`, as the latter implies that Mondrian categories are formed using the labels in `y`.

Note

Enabling out-of-bag calibration, i.e., setting `oob=True`, requires that the wrapped learner has an attribute `oob_decision_function_`, which e.g., is the case for a `sklearn.ensemble.RandomForestClassifier`, if enabled when created, e.g., `RandomForestClassifier(oob_score=True)`

Note

The use of out-of-bag calibration, as enabled by `oob=True`, does not come with the theoretical validity guarantees of the regular (inductive) conformal classifiers, due to that calibration and test instances are not handled in exactly the same way.

Figure 3: Detailed documentation for the `calibrate` method of the class `WrapClassifier`

Acknowledgements

HB was partly funded by 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. crepes: a python package for generating conformal regressors and predictive systems. In Ulf Johansson, Henrik Boström, Khuong An Nguyen, Zhiyuan Luo, and Lars Carlsson, editors, *Proceedings of the Eleventh Symposium on Conformal and Probabilistic Prediction and Applications*, volume 179 of *Proceedings of Machine Learning Research*. PMLR, 2022.

Henrik Boström and Ulf Johansson. Mondrian conformal regressors. In Alexander Gammerman, Vladimir Vovk, Zhiyuan Luo, Evgueni N. Smirnov, Giovanni Cherubin, and Marco Christini, editors, *Conformal and Probabilistic Prediction and Applications, COPA 2020, 9-11 September 2020, Virtual Event*, volume 128 of *Proceedings of Machine Learning Research*, pages 114–133. PMLR, 2020.

Henrik Boström, Ulf Johansson, and Anders Vesterberg. Predicting with confidence from survival data. In Alex Gammerman, Vladimir Vovk, Zhiyuan Luo, and Evgueni N. Smirnov, editors, *Conformal and Probabilistic Prediction and Applications, COPA 2019, 9-11 September 2019, Golden Sands, Bulgaria*, volume 105 of *Proceedings of Machine Learning Research*, pages 123–141. PMLR, 2019.

Henrik Boström, Ulf Johansson, and Tuwe Löfström. Mondrian conformal predictive distributions. In Lars Carlsson, Zhiyuan Luo, Giovanni Cherubin, and Khuong An Nguyen, editors, *Conformal and Probabilistic Prediction and Applications, 8-10 September 2021, Virtual Event*, volume 152 of *Proceedings of Machine Learning Research*, pages 24–38. PMLR, 2021.

Henrik Boström, Henrik Linusson, and Anders Vesterberg. Mondrian predictive systems for censored data. In Harris Papadopoulos, Khuong An Nguyen, Henrik Boström, and Lars Carlsson, editors, *Conformal and Probabilistic Prediction with Applications, 13-15 September 2023, Limassol, Cyprus*, volume 204 of *Proceedings of Machine Learning Research*, pages 399–412. PMLR, 2023.

Emmanuel Candès, Lihua Lei, and Zhimei Ren. Conformalized survival analysis. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 85(1):24–45, 2023.

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 Alex Gammerman, Vladimir Vovk, Zhiyuan Luo, and Harris Papadopoulos, editors, *Conformal and Probabilistic Prediction and Applications, COPA 2017, 13-16 June 2017, Stockholm, Sweden*, volume 60 of *Proceedings of Machine Learning Research*, 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.

Ryan J Tibshirani, Rina Foygel Barber, Emmanuel Candes, and Aaditya Ramdas. Conformal prediction under covariate shift. *Advances in neural information processing systems*, 32, 2019.

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.