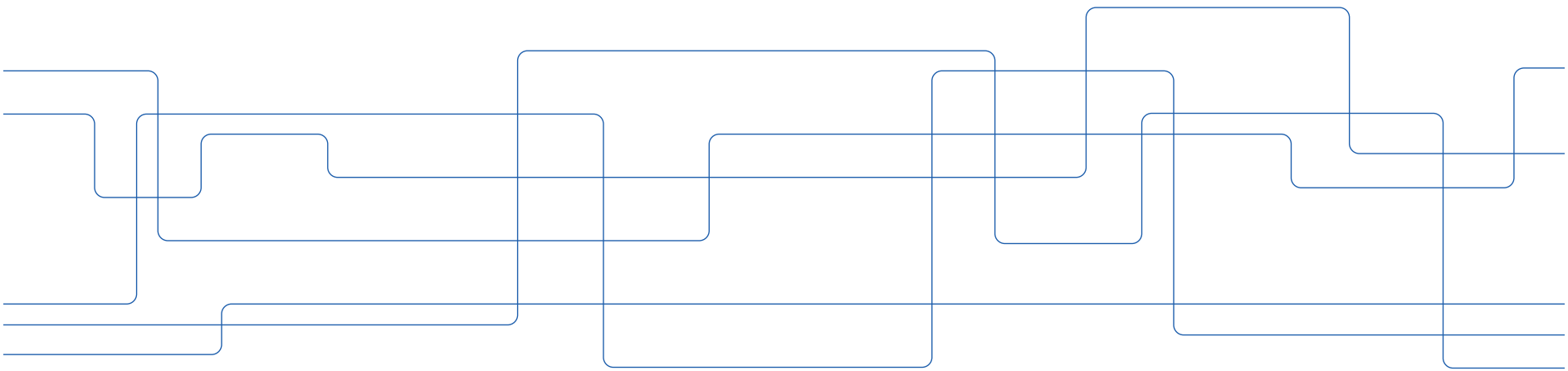
Conformal Prediction in Python with `crepes`

Henrik Boström

bostromh@kth.se

Division of Software and Computer Systems
Department of Computer Science
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Sep. 11, 2024



Outline

Conformal prediction

Conformal prediction turns point predictions into prediction sets

- a *conformal classifier* outputs sets of class labels

  instead of e.g. $\hat{y} = \text{edible}$, a prediction may be $\hat{Y} = \{\text{edible}\}$, $\hat{Y} = \{\text{edible}, \text{poisonous}\}$ or even $\hat{Y} = \emptyset$

- a *conformal regressor* outputs prediction intervals

  instead of e.g. $\hat{y} = 23.5$, a prediction may be $\hat{Y} = [21.0, 25.0]$

Given a confidence level $1 - \epsilon$, the framework guarantees* that the probability of including the correct target is at least $1 - \epsilon$.

*assuming *exchangeability*, a slightly weaker assumption than the IID

Conformal classifiers

An inductive/split conformal classifier is constructed as follows:

1. Randomly divide the training data into two disjoint subsets; the proper training set Z^t and the calibration set $Z^c = \{(\boldsymbol{x}_1, y_1)), \ldots, (\boldsymbol{x}_q, y_q)\}$
2. Calculate *nonconformity scores* $\alpha_1, \ldots, \alpha_q$, where $\alpha_i = A(Z^t, \boldsymbol{x}_i, y_i)$
3. Let $\alpha_{(1)}, \ldots, \alpha_{(q)}$ be the scores sorted in descending order
4. For a test object $\boldsymbol{x}$, output the set of labels:

$$\hat{Y}^\epsilon = \{y \in Y : A(Z^t, \boldsymbol{x}, y) \leq \alpha_{(p)}\}$$

where $p = \lfloor \epsilon(q+1) \rfloor$ and $1 - \epsilon$ is the confidence level

Nonconformity measures

The nonconformity measure is often defined using a model (h) obtained from the proper training set (Z^t):

$$A(Z^t, \boldsymbol{x}, y) = A_h(\boldsymbol{x}, y)$$

Two common nonconformity measures for conformal classifiers are:

$$A_h(\boldsymbol{x}, y) = 1 - h(\boldsymbol{x}, y) \qquad (Hinge)$$

$$A_h(\boldsymbol{x}, y) = max\{h(\boldsymbol{x}, y') : y' \in Y \setminus \{y\}\} - h(\boldsymbol{x}, y) \qquad (Margin)$$

The Python package *crepes*

https://github.com/henrikbostrom/crepes

https://crepes.readthedocs.io

Conformal classifiers

class **crepes.ConformalClassifier** [source]

A conformal classifier transforms non-conformity scores into p-values or prediction sets for a certain confidence level.

Methods

`evaluate`(alphas, classes, y[, bins, ...])	Evaluate conformal classifier.
`fit`(alphas[, bins])	Fit conformal classifier.
`predict_p`(alphas[, bins, confidence])	Obtain (smoothed) p-values from conformal classifier.
`predict_set`(alphas[, bins, confidence, ...])	Obtain prediction sets using conformal classifier.

https://crepes.readthedocs.io/en/latest/crepes.html#crepes.ConformalClassifier

Example

J Chem Inf Model. 2013 Apr 22;53(4):867-78. doi: 10.1021/ci4000213. Epub 2013 Mar 27.

Quantitative structure–activity relationship models for ready biodegradability of chemicals

Kamel Mansouri [1], Tine Ringsted, Davide Ballabio, Roberto Todeschini, Viviana Consonni

Affiliations – collapse

Affiliation

1 Milano Chemometrics and QSAR Research Group, Department of Earth and Environmental Sciences, University of Milano Bicocca, Milano, Italy.

PMID: 23469921 DOI: 10.1021/ci4000213

- 1055 chemical compounds classified into *ready biodegradable* (356 compounds) and *not ready biodegradable* (699 compounds)
- 41 features, such as number of heavy atoms, oxygens, and nitrogens, percentage of carbon atoms, etc.

Importing and splitting data

```
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 = np.array(["NRB" if v == "1" else "RB"
              for v in dataset.target])

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=1/4)

X_prop_train, X_cal, y_prop_train, y_cal = \
    train_test_split(X_train, y_train, test_size=1/3)
```

Fitting a model

```
from sklearn.ensemble import RandomForestClassifier


rf = RandomForestClassifier(n_jobs=-1, n_estimators=500)
rf.fit(X_prop_train, y_prop_train)
```

```
rf.classes_
```

```
array(['NRB', 'RB'], dtype='<U3')
```

```
rf.predict_proba(X_cal)
```

```
array([[0.986, 0.014],
       [0.998, 0.002],
       ...,
       [0.908, 0.092],
       [0.246, 0.754]])
```

Computing nonconformity scores

```
from crepes.extras import hinge

alphas_cal = hinge(rf.predict_proba(X_cal), rf.classes_, y_cal)
```

```
array([0.014, 0.008, 0.11 , 0.054, 0.02 , 0.28 , 0.37 , 0.334, 0.31 ,
       0.598, 0.05 , 0.306, 0.074, 0.024, 0.012, 0.532, 0.102, 0.038,
       ...,
       0.08 , 0.204, 0.008, 0.518, 0.152, 0.04 , 0.   , 0.016, 0.112,
       0.042, 0.08 , 0.282])
```

Fitting the conformal classifier

```
from crepes import ConformalClassifier

cc_std = ConformalClassifier()
cc_std.fit(alphas_cal)
```

```
ConformalClassifier(fitted=True, mondrian=False)
```

Making predictions

```
alphas_test = hinge(rf.predict_proba(X_test))
```

```
rray([[0.96 , 0.04 ],
       [0.638, 0.362],
       ...,
       [0.046, 0.954],
       [0.428, 0.572]])
```

```
cc_std.predict_set(alphas_test, confidence=0.9)
```

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

Computing p-values

To guarantee that the probability of including the correct target is exactly $1 - \epsilon$, we can use smoothed p-values:

$$p_{q+1} = \frac{|i : \alpha_i > \alpha_{q+1}| + \theta|i : \alpha_i = \alpha_{q+1}|}{q+1}$$

For a test object, we reject labels for which $p_{q+1} < \epsilon$.

Using smoothed p-values

```
cc_std.predict_set(alphas_test, smoothing=True)
```

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

```
cc_std.predict_p(alphas_test)
```

```
array([[8.48346601e-03, 7.99304781e-01],
       [9.79196236e-02, 2.43604665e-01],
       ...,
       [7.90745137e-01, 1.02768979e-02],
       [1.92050215e-01, 1.25448486e-01]])
```

Evaluating conformal classifiers

```
cc_std.evaluate(alphas_test, rf.classes_, y_test,
                smoothing=True, confidence=0.9)
```

```
{'error': 0.090909090909094,
 'avg_c': 1.1666666666666667,
 'one_c': 0.8333333333333334,
 'empty': 0.0,
 'time_fit': 0.00011634826660156250,
 'time_evaluate': 0.013993501663208008}
```

```
cc_std.evaluate(alphas_test, rf.classes_, y_test,
                smoothing=True, confidence=0.99)
```

```
{'error': 0.0,
 'avg_c': 1.7348484848484849,
 'one_c': 0.2651515151515151 4,
 'empty': 0.0,
 'time_fit': 0.00011634826660156250,
 'time_evaluate': 0.013120174407958984}
```

Mondrian conformal predictors

A *Mondrian conformal predictor* modifies a standard conformal predictor by dividing the calibration set into disjoint subsets according to a *Mondrian* taxonomy with k categories, and where a standard conformal predictor is produced for each subset.

A prediction for a test instance is obtained by assigning it to one of the k categories and using the standard conformal predictor of that category.

The coverage guarantee now holds for each category.

Mondrian conformal classifiers

```
bins_cal = X_cal[:,2] > 0 # Number of heavy atoms

cc_mond = ConformalClassifier()
cc_mond.fit(alphas_cal, bins_cal)
```

```
ConformalClassifier(fitted=True, mondrian=True)
```

```
bins_test = X_test[:,2] > 0

cc_mond.predict_p(alphas_test, bins_test)
```

```
array([[1.34118105e-03, 8.90896038e-01],
       [1.02425407e-01, 2.84082358e-01],
       ...,
       [8.84040331e-01, 3.91015108e-03],
       [2.10234092e-01, 1.36453236e-01]])
```

Using a MondrianCategorizer

```
from crepes.extras import MondrianCategorizer

def get_values(X):
    return X[:,0] # Leading eigenvalue from Laplace matrix

mc_scoring = MondrianCategorizer()
mc_scoring.fit(X_train, f=get_values, no_bins=5)
```

```
MondrianCategorizer(fitted=True, f=get_values, no_bins=5)
```

```
mc_scoring.apply(X_cal)
```

```
array([3, 4, 0, 4, 4, 3, 1, 0, 3, 0, 4, 4, 1, 4, 3, 2, 1, 1, 2, 3, 1, 0,
       ...,
       0, 1, 0, 0, 1, 1, 0, 2, 4, 1, 0, 0, 3, 3, 4, 4, 3, 3, 1, 4, 4, 0])
```

Wrapping classifiers

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.

https://crepes.readthedocs.io/en/latest/crepes.html#crepes.WrapClassifier

Generating conformal classifiers by wrapping

```
from crepes import WrapClassifier
from crepes.extras import margin

rf_mond = WrapClassifier(rf)
rf_mond.calibrate(X_cal, y_cal, mc=mc_scoring, nc=margin)
rf_mond.predict_p(X_test)
```

```
array([[2.08019749e-02, 9.48464726e-01],
       [1.57925006e-01, 3.08073551e-01],
       ...,
       [5.45490380e-01, 1.29174634e-02],
       [1.43440435e-01, 1.00707237e-01]])
```

Distribution of errors over classes

```
rf_std = WrapClassifier(rf)
rf_std.calibrate(X_cal, y_cal)
rf_std.evaluate(X_test[y_test=="RB"], y_test[y_test=="RB"])
```

```
{'error': 0.1111111111111116,
 'avg_c': 1.3580246913580247,
 'one_c': 0.6419753086419753,
 'empty': 0.0,
 'time_fit': 6.008148193359375e-05,
 'time_evaluate': 0.09965085983276367}
```

```
rf_std.evaluate(X_test[y_test=="NRB"], y_test[y_test=="NRB"])
```

```
{'error': 0.038251366120218624,
 'avg_c': 1.2950819672131149,
 'one_c': 0.7049180327868853,
 'empty': 0.0,
 'time_fit': 6.008148193359375e-05,
 'time_evaluate': 0.11482214927673334}
```

Class-conditional conformal classifiers

```
rf_class_cond = WrapClassifier(rf)
rf_class_cond.calibrate(X_cal, y_cal, class_cond=True)
rf_class_cond.evaluate(X_test[y_test=="RB"], y_test[y_test=="RB"])
```

```
{'error': 0.037037037037037,
 'avg_c': 1.308641975308642,
 'one_c': 0.691358024691358,
 'empty': 0.0,
 'time_fit': 0.0001995563507080078,
 'time_evaluate': 0.08428621292114258}
```

```
rf_class_cond.evaluate(X_test[y_test=="NRB"], y_test[y_test=="NRB"])
```

```
{'error': 0.06557377049180324,
 'avg_c': 1.3770491803278688,
 'one_c': 0.6229508196721312,
 'empty': 0.0,
 'time_fit': 0.0001652240753173828,
 'time_evaluate': 0.1238107681274414}
```

Standard conformal regressors

A *standard (inductive/split) conformal regressor* is constructed as follows:

1. randomly divide the training data into two disjoint subsets; the proper training set and the calibration set
2. train the underlying model h using the proper training set
3. calculate scores $\alpha_1, \ldots, \alpha_q$ for the calibration set, where

$$\alpha_i = |y_i - h(\boldsymbol{x}_i)|$$

4. let $\alpha_{(1)}, \ldots, \alpha_{(q)}$ be the scores sorted in descending order
5. for a test object $\boldsymbol{x}$, output the prediction interval:

$$\hat{Y}^{\epsilon} = h(\boldsymbol{x}) \pm \alpha_{(p)}$$

where $p = \lfloor \epsilon(q+1) \rfloor$ and $1 - \epsilon$ is the confidence level

Normalized conformal regressors

A *normalized (inductive/split) conformal regressor* modifies the standard conformal regressor by calculating calibration scores through:

$$\alpha_i = \frac{|y_i - h(\boldsymbol{x}_i)|}{\sigma_i}$$

where σ_i is a difficulty (quality) estimate of $\boldsymbol{x}_i$

The prediction interval at the confidence level $1 - \epsilon$ for a test object $\boldsymbol{x}$ with difficulty σ then becomes:

$$\hat{Y}^{\epsilon} = h(\boldsymbol{x}) \pm \alpha_{(p)}\sigma$$

Notes on conformal regressors

- As the probability of error is guaranteed by construction (similar to conformal classifiers), conformal regressors are often evaluated w.r.t. *efficiency*, i.e., the size of the prediction intervals
- The efficiency is affected by the performance of the underlying model; normalized conformal regressors are also affected by how well the difficulty estimate correlates with the actual error
- Some approaches to estimating the difficulty rely on training a separate model to predict the size of the error, e.g., using kNN or ANN; others exploit properties of the underlying model, e.g., using disagreement (variance) of the trees in a random forest

Generating conformal regressors by wrapping

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

A learner wrapped with a `ConformalRegressor` or `ConformalPredictiveSystem`.

Methods

`calibrate`(X, y[, de, mc, oob, cps])	Fit a `ConformalRegressor` or `ConformalPredictiveSystem` using the wrapped learner.
`evaluate`(X, y[, confidence, y_min, y_max, ...])	Evaluate `ConformalRegressor` or `ConformalPredictiveSystem`.
`fit`(X, y, **kwargs)	Fit learner.
`predict`(X)	Predict with learner.
`predict_cps`(X[, y, lower_percentiles, ...])	Predict using `ConformalPredictiveSystem`.
`predict_int`(X[, confidence, y_min, y_max])	Predict interval using fitted `ConformalRegressor` or `ConformalPredictiveSystem`.

https://crepes.readthedocs.io/en/latest/crepes.html#crepes.WrapRegressor

Example

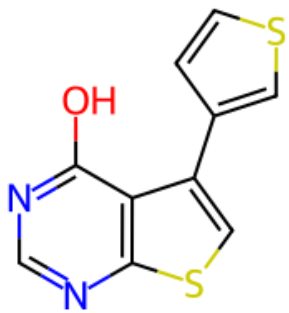
MoleculeNet

https://moleculenet.org/

Lipophilicity

- Experimental results of octanol/water distribution coefficient
- 4200 chemical compounds, represented by the simplified molecular-input line-entry system (SMILES)

`Oc1ncnc2scc(c3ccsc3)c12`



Importing and converting data using `rdkit`

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

url = ("https://deepchemdata.s3-us-west-1.amazonaws.com/datasets/"
       "Lipophilicity.csv")
df = pd.read_csv(url)

y = df["exp"].values

molecules = [rdkit.Chem.MolFromSmiles(s) for s in df["smiles"]]

fpgen = rdkit.Chem.AllChem.GetMorganGenerator(radius=2, fpSize=1024)

X = np.array([fpgen.GetFingerprint(m) for m in molecules])
```

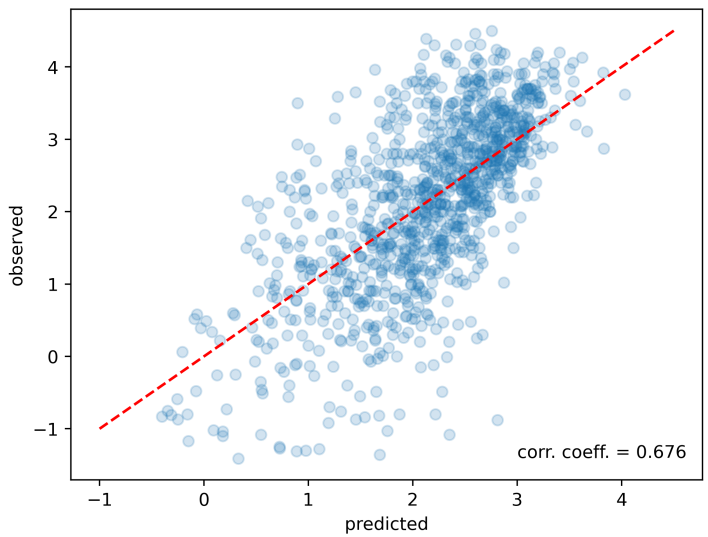
Splitting data

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=1/4)

X_prop_train, X_cal, y_prop_train, y_cal = \
    train_test_split(X_train, y_train, test_size=1/3)
```

Fitting a model

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

rf = WrapRegressor(RandomForestRegressor(n_jobs=-1, n_estimators=500,
                                         oob_score=True))
rf.fit(X_prop_train, y_prop_train)

rf.predict(X_test)
```

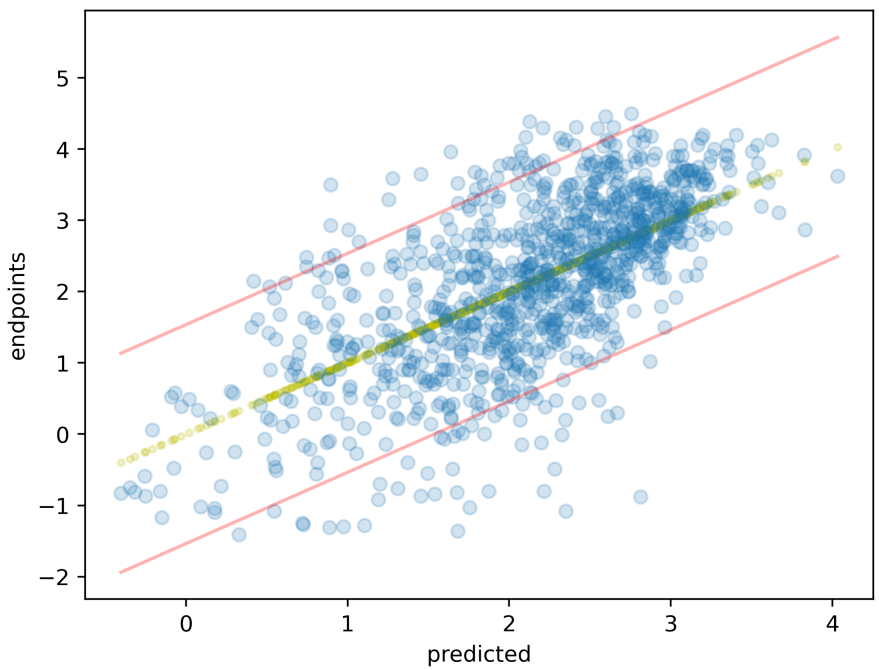
```
array([2.53669333, 2.5703, 1.88712667, ..., 2.84176, 2.271326, 2.23035667])
```



Standard conformal regressors

```
rf.calibrate(X_cal, y_cal)
rf.predict_int(X_test, confidence=0.9)
```

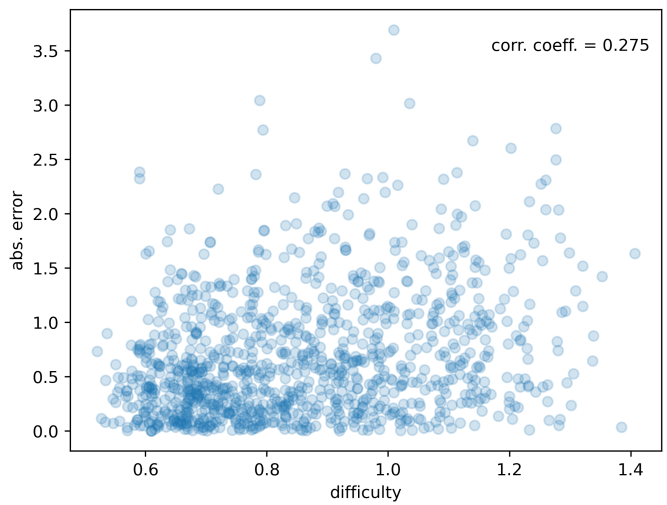
```
array([[0.99983333, 4.07355333],
       [1.03344   , 4.10716   ],
       ...,
       [0.734466  , 3.808186  ],
       [0.69349667, 3.76721667]])
```



Estimating difficulty using kNN

```
from crepes.extras import DifficultyEstimator

de_knn = DifficultyEstimator()
de_knn.fit(X=X_prop_train,
           k=5,
           scaler=True,
           y=y_prop_train,
           beta=0.5)
de_knn.apply(X_test)
```

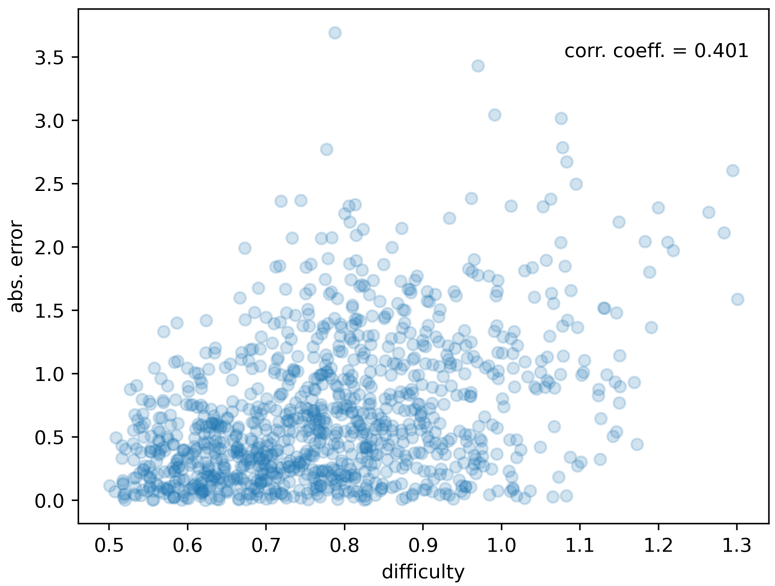
```
array([0.77553191, 0.6509697, 1.16927448, ..., 0.76630351, 0.80739162, 0.72732131])
```



Estimating difficulty using variance

```
de_var = DifficultyEstimator()
de_var.fit(X=X_prop_train,
           learner=rf.learner,
           scaler=True,
           beta=0.5)
de_var.apply(X_test)
```

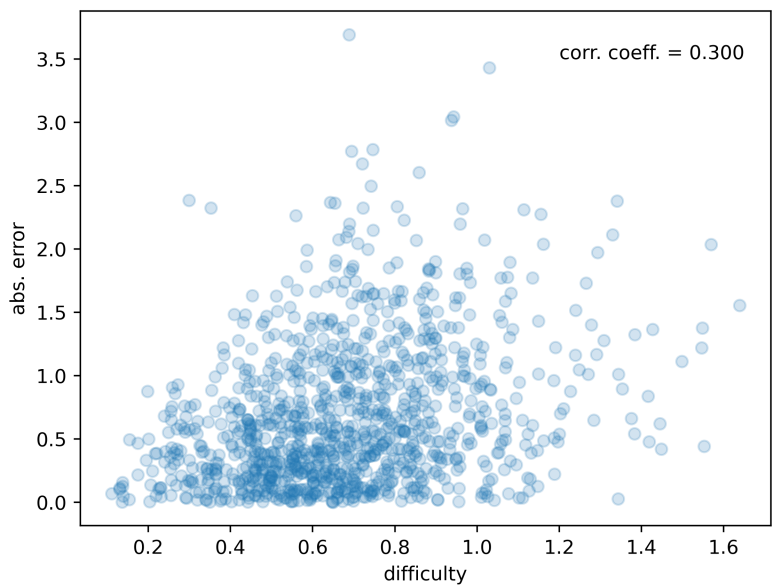
```
array([0.71421376, 0.64187222, 1.0189414 , ..., 0.63728258, 0.85153137, 0.69209821])
```



Estimating difficulty using a model

```
abs_error = np.abs(y_prop_train-rf.learner.oob_prediction_)
error_model = RandomForestRegressor(n_jobs=-1, n_estimators=500)
error_model.fit(X_prop_train, abs_error)
error_model.predict(X_test)
```

```
array([0.56079577, 0.57208813, 0.94819737, ..., 0.54071602, 0.61127624, 0.57837301])
```



Tailor-made difficulty estimator

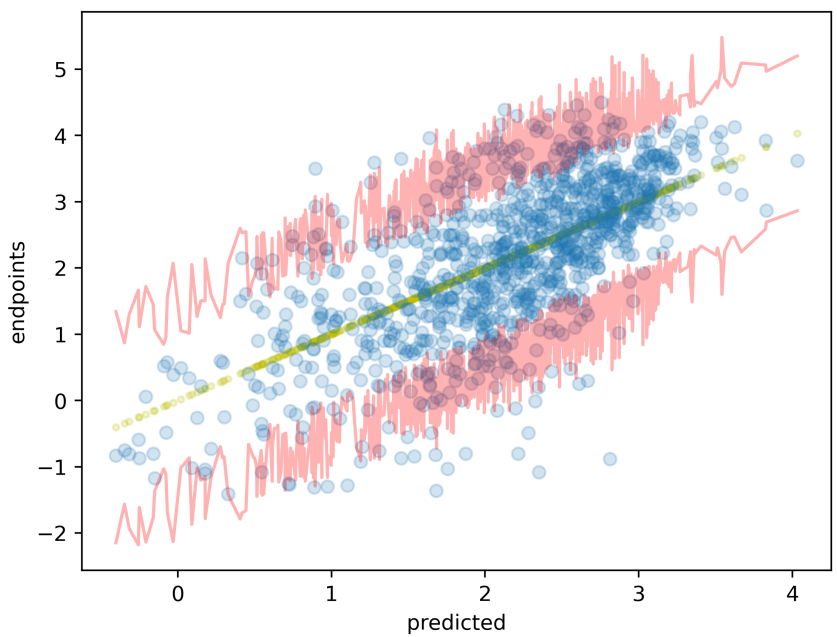
```
class ModelDifficultyEstimator:
    def __init__(self, model, beta=0.01):
        self.model = model
        self.beta = beta
    def apply(self, X):
        return self.model.predict(X) + self.beta

de_mod = ModelDifficultyEstimator(error_model, 0.5)
de_mod.apply(X_test)
```

```
array([1.06079577, 1.07208813, 1.44819737, ..., 1.04071602, 1.11127624, 1.07837301])
```

Normalized conformal regressors

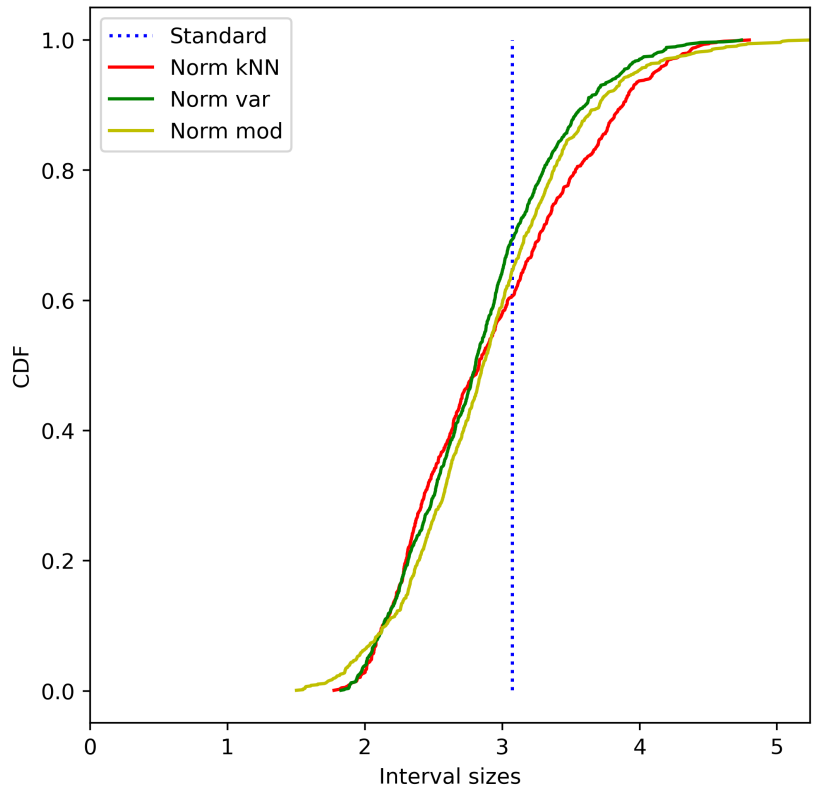
```
rf_norm_knn = WrapRegressor(rf.learner)
rf_norm_knn.calibrate(X_cal, y_cal, de=de_knn) # alt. de_var or de_mod
rf_norm_knn.predict_int(X_test, confidence=0.9)
```

```
array([[ 1.21371781,  3.85966886],
       [ 1.45981446,  3.68078554],
       ...,
       [ 0.89400118,  3.64865082],
       [ 0.98962334,  3.47108999]])
```



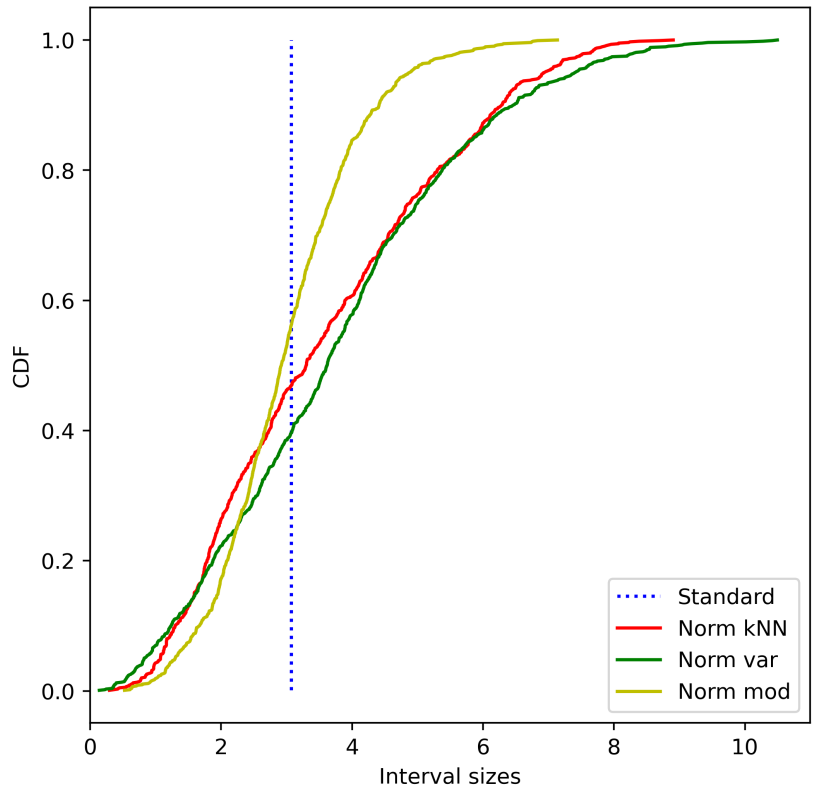
Comparing the conformal regressors ($\beta = 0.5$)

	Coverage	Mean size	Median size
Standard	0.9114	3.0737	3.0737
Norm kNN	0.9019	2.9218	2.8309
Norm var	0.9133	2.8404	2.8053
Norm mod	0.9086	2.9034	2.8609
Mean	0.9088	2.9348	2.8927



Comparing the conformal regressors ($\beta = 0.01$)

	Coverage	Mean size	Median size
Standard	0.9114	3.0737	3.0737
Norm kNN	0.8895	3.5601	3.3013
Norm var	0.9210	3.7449	3.6201
Norm mod	0.9019	2.9957	2.9207
Mean	0.9060	3.3436	3.2289



Mondrian conformal regressors

A *Mondrian conformal regressor* modifies a standard conformal regressor by dividing the calibration set into disjoint subsets according to a *Mondrian* taxonomy with k categories, and where a standard conformal regressor is produced for each subset.

A prediction interval for a test instance is obtained by assigning it to one of the k categories and using the standard conformal regressor of that category.

One option is to form the categories by binning of the difficulty estimates.

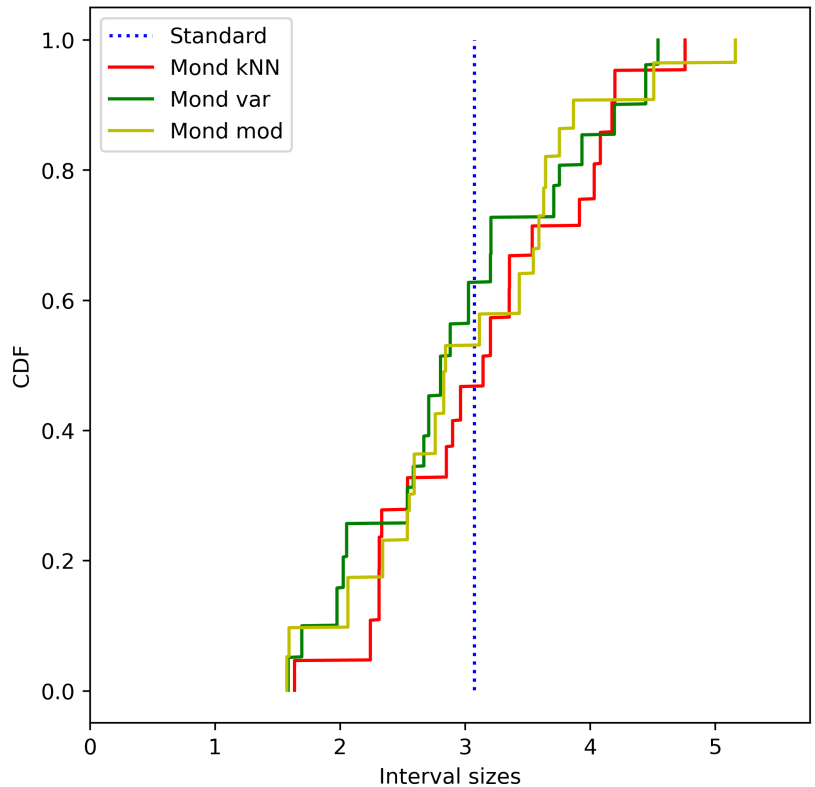
Mondrian conformal regressors

```
from crepes.extras import MondrianCategorizer

mc_knn = MondrianCategorizer()
mc_knn.fit(X_cal, de=de_knn, no_bins=20) # alt. de_var or de_mod

rf_mond_knn = WrapRegressor(rf.learner)
rf_mond_knn.calibrate(X_cal, y_cal, mc=mc_knn)
rf_mond_knn.predict_int(X_test, confidence=0.9)
```

```
array([[ 1.05551167,  4.017875  ],
       [ 0.99910143,  4.14149857],
       [-0.21139333,  3.98564667],
       ...,
       [ 1.36057833,  4.32294167],
       [ 0.846741  ,  3.695911  ],
       [ 1.07498333,  3.38573   ]])
```

Comparing the Mondrian conformal regressors

	Coverage	Mean size	Median size
Standard	0.9114	3.0737	3.0737
Mond kNN	0.9190	3.1550	3.1424
Mond var	0.9143	2.9444	2.8022
Mond mod	0.9162	3.0357	2.8427
Mean	0.9152	3.0522	2.9653



Conformal predictive systems

Conformal predictive systems is a generalization of conformal regressors by which the point predictions are transformed into cumulative distribution functions.

Conformal predictive systems come with a *validity* guarantee; the output of the conformal predictive distributions (the p-values) for the correct target values are distributed uniformly on $[0, 1]$.

This allows us to control the error level when making predictions on whether the target values are higher (or lower) than a certain threshold value.

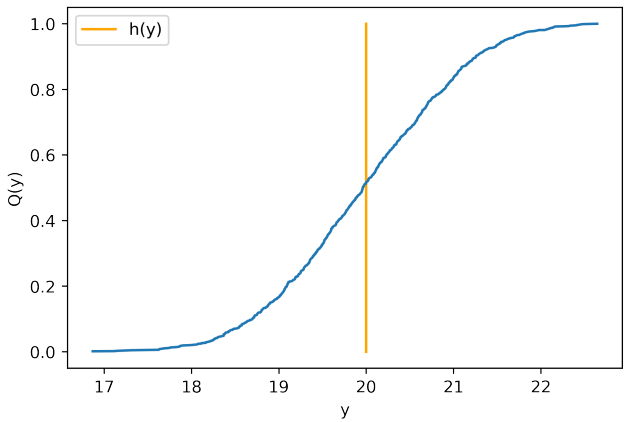
Conformal predictive systems

A *split conformal predictive system* can be constructed as follows:

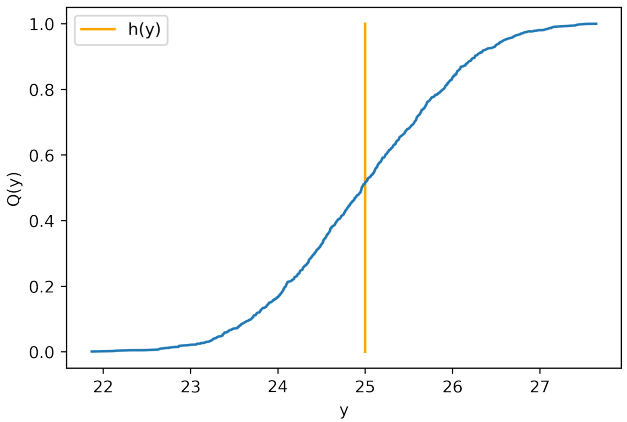
1. randomly divide the training data into two disjoint subsets; the proper training set and the calibration set
2. train the underlying model h using the proper training set
3. calculate scores $\alpha_1, \ldots, \alpha_q$ for the calibration set, where
$$\alpha_i = \frac{y_i - h(\boldsymbol{x}_i)}{\sigma_i}$$
4. let $\alpha_{(1)}, \ldots, \alpha_{(q)}$ be the scores sorted in ascending order
5. for a test object $\boldsymbol{x}$ with difficulty σ:
   let $C_{(i)} = h(\boldsymbol{x}) + \alpha_{(i)}\sigma$ for $i \in \{1, ..., q\}$
   let $C_{(0)} = -\infty$ and $C_{(q+1)} = \infty$
   output the conformal predictive distribution:
$$Q(y) = \begin{cases} \frac{n+\tau}{q+1} & \text{if } y \in \left(C_{(n)}, C_{(n+1)}\right) \text{ for } n \in \{0, ..., q\} \\ \frac{n'-1+(n''-n'+2)\tau}{q+1} & \text{if } y = C_{(n)} \text{ for } n \in \{1, ..., q\} \end{cases}$$

Conformal predictive distributions

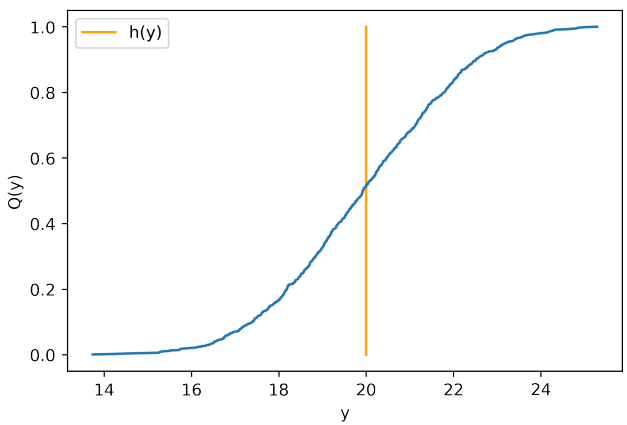
$h(x) = 20\ \sigma = 1$



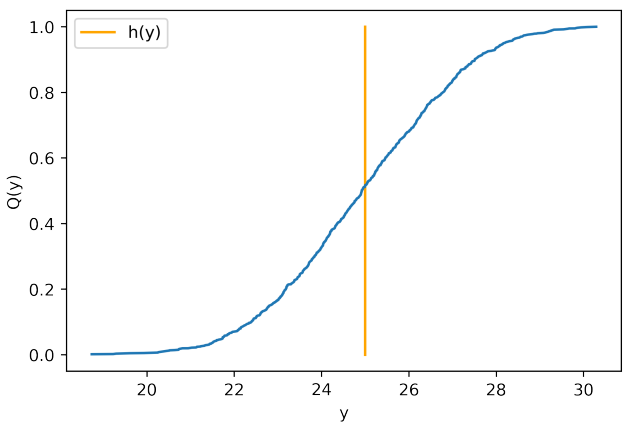
$h(x) = 25\ \sigma = 1$



$h(x) = 20\ \sigma = 2$



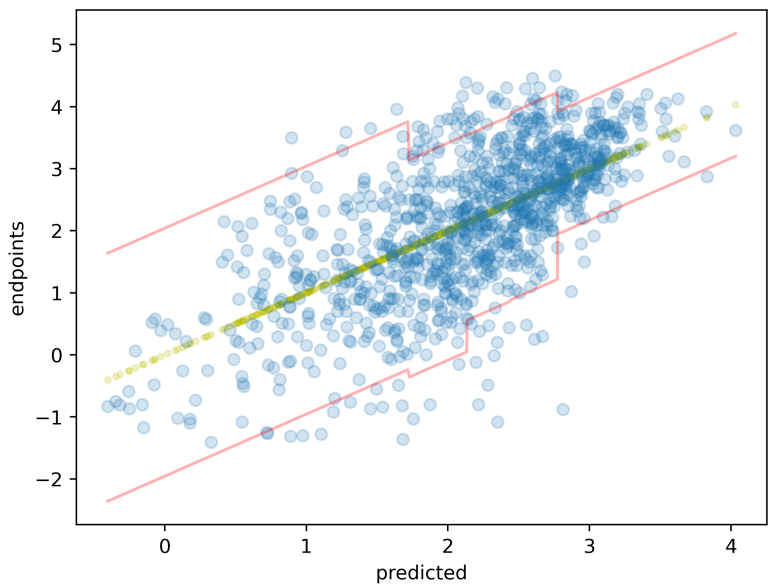
$h(x) = 25\ \sigma = 2$



Mondrian conformal predictive systems

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

cps_mond = WrapRegressor(rf.learner)
cps_mond.calibrate(X_cal, y_cal, mc=mc_pred, cps=True)
cps_mond.predict_int(X_test, confidence=0.9)
```

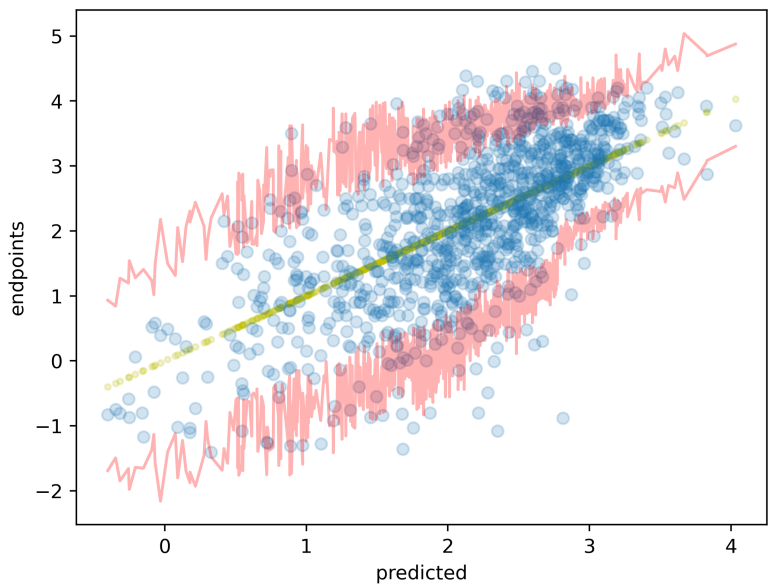
```
array([[ 0.97988933,  3.99312   ],
       [ 1.013496  ,  4.02672667],
       ...,
       [ 0.693298  ,  3.67807533],
       [ 0.65232867,  3.637106  ]])
```



Mondrian conformal predictive systems

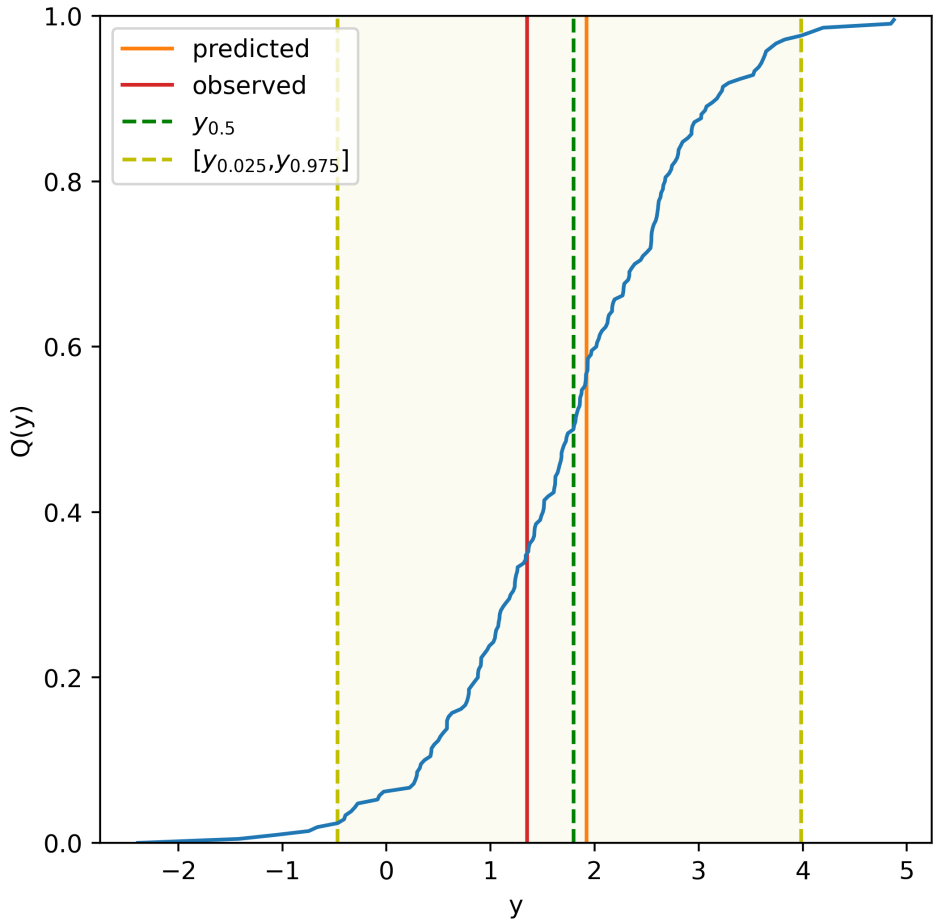
```
cps_mond_norm = WrapRegressor(rf.learner)
cps_mond_norm.calibrate(X_cal, y_cal, mc=mc_pred, de=de_var, cps=True)
cps_mond_norm.predict_int(X_test, confidence=0.9)
```

```
array([[ 1.03111744,  3.7729037 ],
       [ 1.21722141,  3.68129665],
       ...,
       [ 0.57637959,  3.73293946],
       [ 0.85275695,  3.41831062]])
```



Obtaining conformal predictive distributions

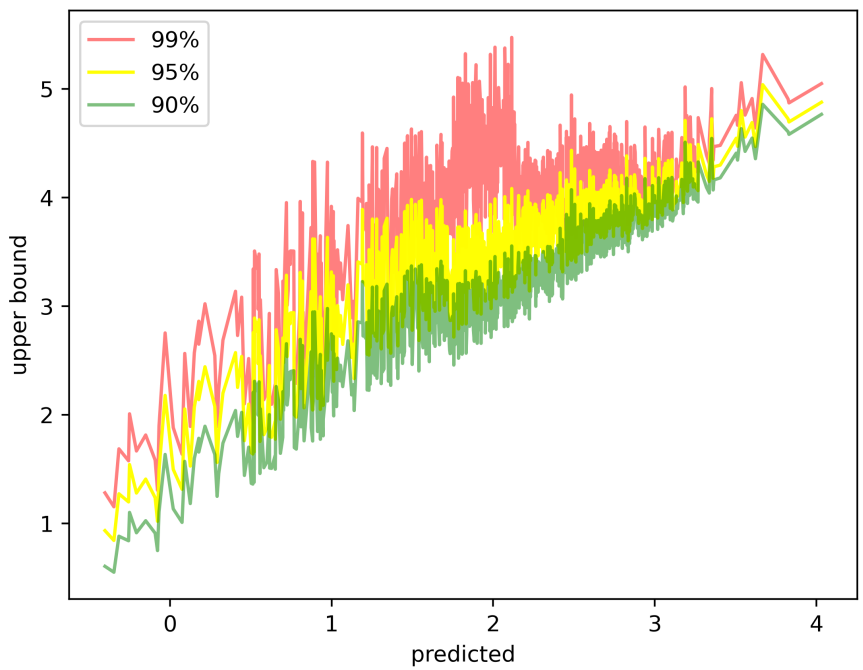
```
cps_mond_norm.predict_cps(X_test, return_cpds=True)
```



Computing upper bounds

```
cps_mond_norm.predict_cps(X_test, higher_percentiles=[90,95,99])
```

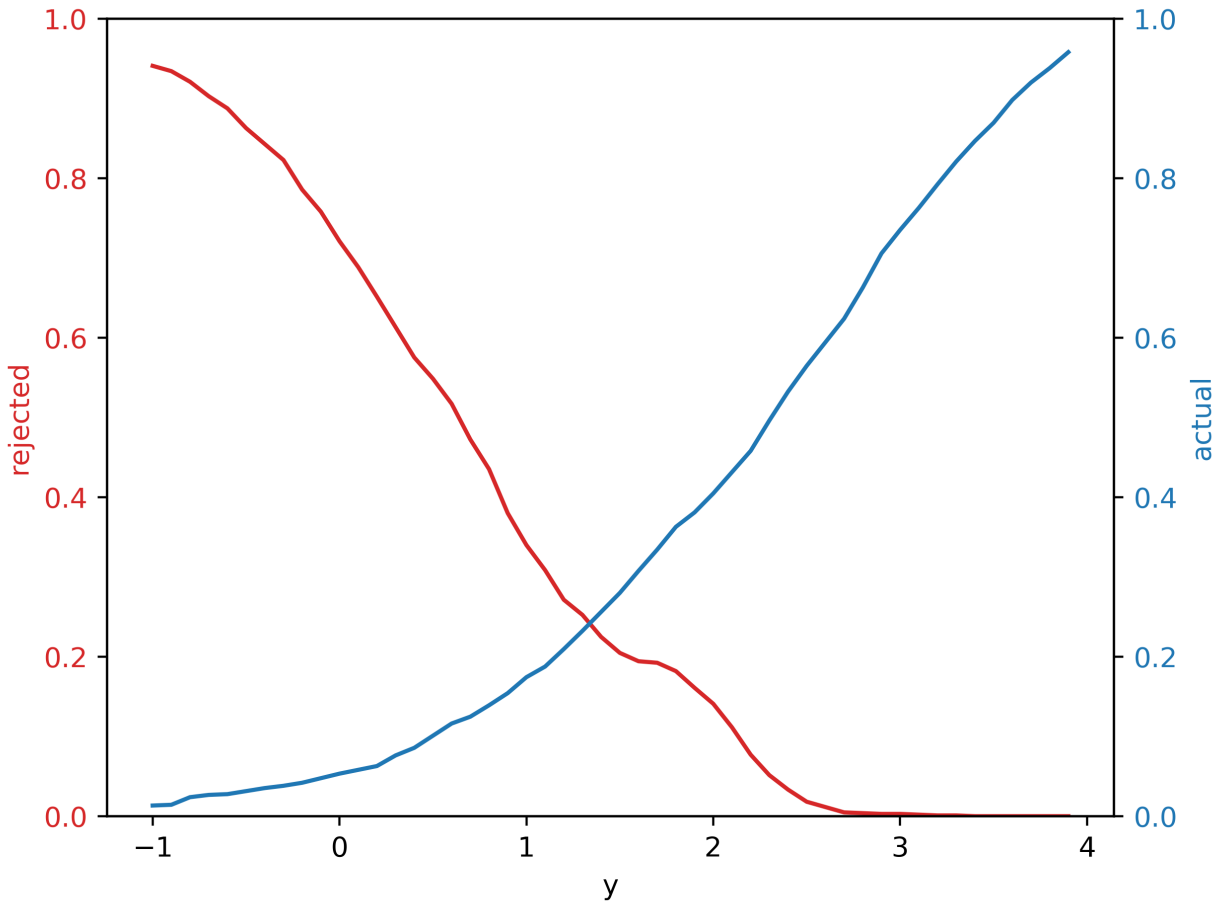
```
array([[3.54295102, 3.7729037 , 4.09602868],
       [3.4746355 , 3.68129665, 3.97169283],
       ...,
       [3.32060887, 3.73293946, 4.08507032],
       [3.08318114, 3.41831062, 3.70451163]])
```



Computing p-values

```
cps_mond_norm.predict_cps(X_test, y=0)
```

```
array([0.00876593, 0.00888225, 0.06650352, ..., 0.00130917, 0.03168884, 0.01191915])
```



Concluding remarks

- Conformal prediction allows for controlling the probability of error of any classifier or regressor by turning point predictions into prediction sets.
- Standard conformal regressors produce equisized intervals, while normalized conformal regressors may produce more informative and tighter intervals.
- Normalized conformal regressors may however produce intervals that appear informative without actually being so and the sizes may be unreasonably large; a remedy for this is provided by Mondrian conformal regressors.

Concluding remarks (cont.)

- Conformal predictive systems provide predictions in the form of conformal predictive distributions, which are more informative than prediction intervals; the latter can be derived from the former.
- Mondrian conformal predictive systems allow for affecting the shape of the output conformal predictive distributions, beyond changing the location and scale.
- The `crepes` package is under continuous development; suggestions and feedback are most welcome!