# Orange - Conformal Prediction Documentation

*Release 1.0*

**Biolab**

Jul 19, 2016

Contents

CHAPTER 1

# Tutorial

## 1.1 Introduction

The Conformal Predictions add-on expands the Orange library with implementations of algorithms from the theoretical framework of conformal predictions (CP) to obtain error calibration under classification and regression settings.

In contrast with standard supervised machine learning, which for a given new data instance typically produces $\hat{y}$, called a *point prediction*, here we are interested in making a *region prediction*. For example, with conformal prediction we could produce a 95% prediction region — a set $\Gamma^{0.05}$ that contains the true label $y$ with probability at least 95%. In the case of regression, where $y$ is a number, $\Gamma^{0.05}$ is typically an interval around $\hat{y}$. In the case of classification, where $y$ has a limited number of possible values, $\Gamma^{0.05}$ may consist of a few of these values or, in the ideal case, just one. For a more detailed explanation of the conformal predictions theory refer to the paper *[Vovk08]* or the book *[Shafer05]*.

In this library the final method for conformal predictions is obtained by selecting a combination of pre-prepared components. Starting with the learning method (either classification or regression) used to fit predictive models, we need to link it with a suitable nonconformity measure and use them together in a selected conformal predictions procedure: transductive, inductive or cross. These CP procedures differ in the way data is split and used for training the predictive model and calibration, which computes the distribution of nonconformity scores used to evaluate possible new predicitions. Inductive CP requires two disjoint data sets to be provided - one for training, the other for calibration. Cross CP uses a single training data set and automatically prepares k different splits into training and calibration sets in the same manner as k-fold crossvalidation. Transductive CP on the other hand does not need a separate calibration set at all, but retrains the model with a new test instance included for each of its possible labels and compares the nonconformity to those of the labelled instances. This allows it to use the complete training set, but makes it computationally more expensive.

Sections below will explain how to use the implemented methods from this library through practical examples and use-cases. For a detailed documentation of implemented methods and classes along with their parameters consult the *Library reference*. For more code examples, take a look at the tests module.

### 1.1.1 References

## 1.2 Classification

All 3 types of conformal prediction are implemented for classification (transductive, inductive and cross), with several different nonconformity measures to choose from.

We will show how to train and use a conformal predictive model in the following simple, but fully functional example.

Let's load the iris data set and try to make a prediction for the last instance using the rest for learning.

```
>>> import Orange
>>> iris = Orange.data.Table('iris')
>>> train = iris[:-1]
>>> test_instance = iris[-1]
```

We will use a LogisticRegressionLearner from Orange and the inverse probability nonconformity score in a 5-fold cross conformal prediction classifier.

```
>>> lr = Orange.classification.LogisticRegressionLearner()
>>> ip = cp.nonconformity.InverseProbability(lr)
>>> ccp = cp.classification.CrossClassifier(ip, 5, train)
```

Predicting the 90% and 99% prediction regions gives the following results.

```
>>> print('Actual class:', test_instance.get_class())
Actual class: Iris-virginica
>>> print(ccp(test_instance.x, 0.1))
['Iris-virginica']
>>> print(ccp(test_instance.x, 0.01))
['Iris-versicolor', 'Iris-virginica']
```

We can see that in the first case only the correct class of 'Iris-virginica' was predicted. In the second case, with a much lower tolerance for errors, the model claims only that the instance belongs to one of two possible classes 'Iris-versicolor' or 'Iris-virginica', but not the third 'Iris-setosa'.

## 1.3 Regression

For regression inductive and cross conformal prediction are implemented along with several nonconformity measures.

Similarly to the classification example, let's combine some standard components to show how to train and use a conformal prediction model for regression.

Let's load the housing data set and try to make a prediction for the last instance using the rest for learning.

```
>>> import Orange
>>> housing = Orange.data.Table('housing')
>>> train = housing[:-1]
>>> test_instance = housing[-1]
```

We will use a LinearRegressionLearner from Orange and the absolute error nonconformity score in a 5-fold cross conformal regressor.

```
>>> lr = Orange.regression.LinearRegressionLearner()
>>> abs_err = cp.nonconformity.AbsError(lr)
>>> ccr = cp.regression.CrossRegressor(abs_err, 5, train)
```

Predicting the 90% and 99% prediction regions gives the following results.

```
>>> print('Actual target value:', test_instance.get_class())
Actual target value: 11.900
>>> print(ccr(test_instance.x, 0.1))
(13.708550425853684, 31.417230194137165)
>>> print(ccr(test_instance.x, 0.01))
(-0.98542733224618217, 46.111207952237031)
```

We can see that in the first case the predicted interval was smaller, but did not contain the correct value (this should not happend more than 10% of the time). In the second case, with a much lower tolerance for errors, the model predicted a larger interval, which did contain the correct value.

## 1.4 Evaluation

The evaluation module provides many useful classes and functions for evaluating the performance and validity of conformal predictions. The main two classes, which represent the results of a conformal classifier and regressor, are `cp.evaluation.ResultsClass` and `cp.evaluation.ResultsRegr`.

For ease of use, the evaluation results can be obtained using utility functions that evaluate the selected conformal predictor on data defined by the provided sampler (`cp.evaluation.run()`) or explicitly provided by the user (`cp.evaluation.run_train_test()`).

As an example, let's take a look at how to quickly evaluate a conformal classifier on a test data set and compute some of the performance metrics:

```
>>> import Orange
>>> import cp
>>> iris = Orange.data.Table('iris')
>>> train, test = iris[::2], iris[1::2]
>>> lr = Orange.classification.LogisticRegressionLearner()
>>> ip = cp.nonconformity.InverseProbability(lr)
>>> ccp = cp.classification.CrossClassifier(ip, 5)
>>> res = cp.evaluation.run_train_test(ccp, 0.1, train, test)
```

The results are an instance of `cp.evaluation.ResultsClass` mentioned above, and can be used to compute the accuracy of predictions (fraction of predictions including the actual class). For a *valid* predictor it needs to hold that the error (1 - accuracy) is lower or equal to the specified significance level. In addition to *validity*, we are often interested in the *efficiency* of a predictor. For classification, this is often measured with the fraction of cases with a single predicted class (`cp.evaluation.ResultsClass.singleton_criterion()`). For regression, one might measure the widths of predicted intervals and e.g. report the average value (`cp.evaluation.ResultsRegr.mean_range()`).

```
>>> print('Accuracy:', res.accuracy())
Accuracy: 0.946666666667
>>> print('Singletons:', res.singleton_criterion())
Singletons: 0.96
```

Another very useful visual validation approach is to plot the dependency of the actual measured error rate at different levels of the specified significance so the user can quickly see that the error is indeed controlled by the parameter. There is a function in the evaluation module that prepares a calibration plot for the specified predictor and data:

```
>>> cp.evaluation.calibration_plot(ccp, iris, fname='calibration.png')
```

CHAPTER 2

# Library reference

## 2.1 cp.base

**class** `cp.base.`**`ConformalPredictor`**

Bases: `object`

Base class for conformal predictors.

**`__call__`**(*example*, *eps*)

Extending classes should implement this method to return predicted values for a given example and significance level.

**`predict`**(*example*, *eps*)

Extending classes should implement this method to return a prediction object. for a given example and significance level.

## 2.2 cp.classification

Classification module contains methods for conformal classification.

Conformal classifiers predict a set of classes (not always a single class) under a given significance level (error rate). Every classifier works in combination with a nonconformity measure and on average predicts the correct class with the given error rate. Lower error rates result in smaller sets of predicted classes.

Structure:

- **ConformalClassifier**
  - Transductive (*`TransductiveClassifier`*)
  - Inductive (*`InductiveClassifier`*)
  - Cross (*`CrossClassifier`*)

**class** `cp.classification.`**`PredictionClass`**(*p*, *eps*)

Bases: `object`

Conformal classification prediction object, which is produced by the *`ConformalClassifier.predict()`* method.

**`p`**

*List* – List of pairs (p-value, class)

**eps**
*float* – Default significance level (error rate).

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> tcp = TransductiveClassifier(InverseProbability(NaiveBayesLearner()), train)
```

```
>>> prediction = tcp.predict(test[0].x, 0.1)
>>> print(prediction.confidence(), prediction.credibility())
```

```
>>> prediction = tcp.predict(test[0].x)
>>> print(prediction.classes(0.1), prediction.classes(0.9))
```

**__init__**(*p*, *eps*)
Initialize the prediction.

**Parameters**

- **p** (`List`) – List of pairs (p-value, class)
- **eps** (`float`) – Default significance level (error rate).

**classes**(*eps=None*)
Compute the set of classes under the default or given *eps* value.

**Parameters** **eps** (`float`) – Significance level (error rate).

**Returns** List of predicted classes.

**verdict**(*ref*)
Conformal classification prediction is correct when the actual class appears among the predicted classes.

**Parameters** **ref** – Reference/actual class

**Returns** True if the prediction with default *eps* is correct.

**confidence**()
Confidence is an efficiency measure of a single prediction.

Computes minimum $eps$ that would still result in a prediction of a single label. $eps = \text{second\_largest}(p_i)$

**Returns** Confidence $1 - eps$.

**Return type** float

**credibility**()
Credibility is an efficiency measure of a single prediction. Small credibility indicates an unusual example.

Computes minimum $eps$ that would result in an empty prediction set. $eps = \max(p_i)$

**Returns** Credibility $eps$.

**Return type** float

**class** `cp.classification.`**ConformalClassifier**(*nc_measure*, *mondrian=False*)
Bases: `cp.base.ConformalPredictor`

Base class for conformal classifiers.

**__init__**(*nc_measure*, *mondrian=False*)
Verify that the nonconformity measure can be used for classification.

**p_values** (*example*)
Extending classes should implement this method to return a list of pairs (p-value, class) for a given example.

Conformal classifier assigns an assumed class value to the given example and computes its nonconformity. P-value is the ratio of more nonconformal (stranger) instances that the given example.

**predict** (*example*, *eps=None*)
Compute a classification prediction object from p-values for a given example and significance level.

**Parameters**

- **example** (*ndarray*) – Attributes array.
- **eps** (*float*) – Default significance level (error rate).

**Returns** Classification prediction object.

**Return type** *PredictionClass*

**__call__** (*example*, *eps*)
Compute predicted classes for a given example and significance level.

**Parameters**

- **example** (*ndarray*) – Attributes array.
- **eps** (*float*) – Significance level (error rate).

**Returns** List of predicted classes.

**class** cp.classification.**TransductiveClassifier** (*nc_measure*, *train=None*, *mondrian=False*)
Bases: *cp.classification.ConformalClassifier*

Transductive classification.

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> tcp = TransductiveClassifier(ProbabilityMargin(NaiveBayesLearner()), train)
>>> print(tcp(test[0].x, 0.1))
```

**__init__** (*nc_measure*, *train=None*, *mondrian=False*)
Initialize transductive classifier with a nonconformity measure and a training set.

Fit the conformal classifier to the training set if present.

**Parameters**

- **nc_measure** (*ClassNC*) – Classification nonconformity measure.
- **train** (*Optional[Table]*) – Table of examples used as a training set.
- **mondrian** (*bool*) – Use a mondrian setting for computing p-values.

**fit** (*train*)
Fit the conformal classifier to the training set and store the domain.

**Parameters** **train** (*Optional[Table]*) – Table of examples used as a training set.

**p_values** (*example*)
Compute p-values for every possible class.

Transductive classifier appends the given example with an assumed class value to the training set and compares its nonconformity against all other instances.

**Parameters** **example** (*ndarray*) – Attributes array.

**Returns** List of pairs (p-value, class)

**class** cp.classification.**InductiveClassifier**(*nc_measure*, *train=None*, *calibrate=None*, *mondrian=False*)

Bases: *cp.classification.ConformalClassifier*

Inductive classification.

**alpha**

Nonconformity scores of the calibration instances. Computed by the *fit()* method.

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> train, calibrate = next(RandomSampler(train, 2, 1))
>>> icp = InductiveClassifier(InverseProbability(LogisticRegressionLearner()),␣
↪train, calibrate)
>>> print(icp(test[0].x, 0.1))
```

**__init__**(*nc_measure*, *train=None*, *calibrate=None*, *mondrian=False*)

Initialize inductive classifier with a nonconformity measure, training set and calibration set. If present, fit the conformal classifier to the training set and compute the nonconformity scores of calibration set.

**Parameters**

- **nc_measure** (ClassNC) – Classification nonconformity measure.
- **train** (*Optional[Table]*) – Table of examples used as a training set.
- **calibrate** (*Optional[Table]*) – Table of examples used as a calibration set.
- **mondrian** (*bool*) – Use a mondrian setting for computing p-values.

**fit**(*train*, *calibrate*)

Fit the conformal classifier to the training set, compute and store nonconformity scores (*alpha*) on the calibration set and store the domain.

**Parameters**

- **train** (*Optional[Table]*) – Table of examples used as a training set.
- **calibrate** (*Optional[Table]*) – Table of examples used as a calibration set.

**p_values**(*example*)

Compute p-values for every possible class.

Inductive classifier assigns an assumed class value to the given example and compares its nonconformity against all other instances in the calibration set.

**Parameters** **example** (*ndarray*) – Attributes array.

**Returns** List of pairs (p-value, class)

**class** cp.classification.**CrossClassifier**(*nc_measure*, *k*, *train=None*, *mondrian=False*)

Bases: *cp.classification.InductiveClassifier*

Cross classification.

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> ccp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 3,
↪train)
>>> print(ccp(test[0].x, 0.1))
```

**__init__**(*nc_measure*, *k*, *train=None*, *mondrian=False*)
Initialize cross classifier with a nonconformity measure, number of folds and training set. If present, fit the conformal classifier to the training set.

**Parameters**

- **nc_measure** (`ClassNC`) – Classification nonconformity measure.
- **k** (*int*) – Number of folds.
- **train** (*Optional[Table]*) – Table of examples used as a training set.
- **mondrian** (*bool*) – Use a mondrian setting for computing p-values.

**fit**(*train*)
Fit the cross classifier to the training set. Split the training set into k folds for use as training and calibration set with an inductive classifier. Concatenate the computed nonconformity scores and store them (*InductiveClassifier.alpha*).

**Parameters** **train** (*Table*) – Table of examples used as a training set.

**class** cp.classification.**LOOClassifier**(*nc_measure*, *train=None*, *mondrian=False*)
Bases: *cp.classification.CrossClassifier*

Leave-one-out classifier is a cross conformal classifier with the number of folds equal to the size of the training set.

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> loocp = LOOClassifier(InverseProbability(LogisticRegressionLearner()), train)
>>> print(loocp(test[0].x, 0.1))
```

**__init__**(*nc_measure*, *train=None*, *mondrian=False*)

**fit**(*train*)

## 2.3 cp.evaluation

Evaluation module contains methods for evaluation of conformal predictors.

Function *run()* produces Results of an appropriate type by using a Sampler on a given data set to split it into a training and testing set.

Structure:

- **Sampler (sampling methods)**
  - *RandomSampler*
  - *CrossSampler*

  - `LOOSampler`
- **Results (evaluation results)**
  - `ResultsClass`
  - `ResultsRegr`
- **Evaluation methods**
  - `run()`
  - `run_train_test()`
  - `calibration_plot()`

**class** `cp.evaluation.`**`Sampler`**(*data*)

Bases: `object`

Base class for various data sampling/splitting methods.

**`data`**

*Table* – Data set for sampling.

**`n`**

*int* – Size of the data set.

### Examples

```
>>> s = CrossSampler(Table('iris'), 4)
>>> for train, test in s.repeat(3):
>>>     print(train)
```

**`__init__`**(*data*)

Initialize the data set.

**`__iter__`**()

**`__next__`**()

Extending samplers should implement the __next__ method to return the selected and remaining part of the data.

**`repeat`**(*rep=1*)

Repeat sampling several times.

**class** `cp.evaluation.`**`RandomSampler`**(*data, a, b*)

Bases: `cp.evaluation.Sampler`

Randomly samples a subset of data in proportion a:b.

**`k`**

*float* – Size of the selected subset.

### Examples

```
>>> s = RandomSampler(Table('iris'), 3, 2)
>>> train, test = next(s)
```

**`__init__`**(*data, a, b*)

Initialize the data set and the size of the desired selection.

**`__iter__`**()
Return a special iterator over a single split of data.

**`__next__`**()
Splits the data based on a random permutation.

**class** `cp.evaluation.`**`CrossSampler`**(*data*, *k*)
Bases: *`cp.evaluation.Sampler`*

Sample the data in *`k`* folds. Shuffle the data before determining the folds.

**`k`**
*int* – Number of folds.

### Examples

```
>>> s = CrossSampler(Table('iris'), 4)
>>> for train, test in s:
>>>     print(train)
```

**`__init__`**(*data*, *k*)

**`__next__`**()
Compute the next fold. Initializes a new k-fold split on each repetition of the entire sampling procedure.

**class** `cp.evaluation.`**`LOOSampler`**(*data*)
Bases: *`cp.evaluation.CrossSampler`*

Leave-One-Out sampler is a cross sampler with the number of folds equal to the size of the data set.

### Examples

```
>>> s = LOOSampler(Table('iris'))
>>> for train, test in s:
>>>     print(len(test))
```

**`__init__`**(*data*)

**class** `cp.evaluation.`**`Results`**
Bases: `object`

Contains results of an evaluation of a conformal predictor returned by the *`run()`* function.

### Examples

```
>>> cp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 5)
>>> r = run(cp, 0.1, RandomSampler(Table('iris'), 2, 1))
>>> print(r.accuracy())
```

**`__init__`**()

**`add`**(*pred*, *ref*)
Add a new predicted and corresponding reference value.

**`concatenate`**(*r*)
Concatenate another set of results.

**accuracy**()

Compute the accuracy of the predictor averaging verdicts of individual predictions. This is the fraction of instances that contain the actual/reference class among the predicted ones for classification and the fraction of instances that contain the actual value within the predicted range for regression.

**time**()

**class** cp.evaluation.**ResultsClass**

Bases: *cp.evaluation.Results*

Results of evaluating a conformal classifier. Provides classification specific efficiency measures.

### Examples

```
>>> cp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 5)
>>> r = run(cp, 0.1, RandomSampler(Table('iris'), 2, 1))
>>> print(r.singleton_criterion())
```

**accuracy**(*class_value=None*)

Compute accuracy for test instances with a given class value. If this parameter is not given, compute accuracy over all instances, regardless of their class.

**confidence**()

Average confidence of predictions.

**credibility**()

Average credibility of predictions.

**confusion**(*actual*, *predicted*)

Compute the number of singleton predictions of class *predicted* when the actual class is *actual*.

### Examples

Drawing a confusion matrix.

```
>>> data = Table('iris')
>>> cp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 3)
>>> r = run(cp, 0.1, RandomSampler(data, 2, 1))
>>> values = data.domain.class_var.values
>>> form = '{: >20}'*(len(values)+1)
>>> print(form.format('actual\predicted', *values))
>>> for a in values:
>>>     c = [r.confusion(a, p) for p in values]
>>>     print(('{: >20}'*(len(c)+1)).format(a, *c))
    actual\predicted         Iris-setosa     Iris-versicolor      Iris-
↪virginica
         Iris-setosa                  18                   0                   ␣
↪ 0
     Iris-versicolor                   0                  14                   ␣
↪ 4
      Iris-virginica                   0                   0                   ␣
↪12
```

**multiple_criterion**()

Number of cases with multiple predicted classes.

**singleton_criterion**()
Number of cases with a single predicted class.

**empty_criterion**()
Number of cases with no predicted classes.

**singleton_correct**()
Fraction of singleton predictions that are correct.

**class** cp.evaluation.**ResultsRegr**
Bases: *cp.evaluation.Results*

Results of evaluating a conformal regressor. Provides regression specific efficiency measures.

### Examples

```
>>> ir = InductiveRegressor(AbsErrorKNN(Euclidean, 10, average=True))
>>> r = run(ir, 0.1, RandomSampler(Table('housing'), 2, 1))
>>> print(r.interdecile_range())
```

**widths**()

**median_range**()
Median width of predicted ranges.

**mean_range**()
Mean width of predicted ranges.

**std_dev**()
Standard deviation of widths of predicted ranges.

**interdecile_range**()
Difference between the first and ninth decile of widths of predicted ranges.

**interdecile_mean**()
Mean width discarding the smallest and largest 10% of widths of predicted ranges.

cp.evaluation.**run**(*cp*, *eps*, *sampler*, *rep=1*)
Run method is used to repeat an experiment one or more times with different splits of the dataset into a training and testing set. The splits are defined by the provided sampler. The conformal predictor itself might further split the testing set internally for its computations (e.g. inductive or cross predictors).

Run the conformal predictor *cp* on the datasets defined by the provided sampler and number of repetitions and construct the results. Fit the conformal predictor on each training set returned by the sampler and evaluate it on the corresponding test set. Inductive conformal predictors use one third of the training set (random subset) for calibration.

For more control over the exact datasets used for training, testing and calibration see *run_train_test()*.

**Returns** *ResultsClass* or *ResultsRegr*

### Examples

```
>>> cp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 5)
>>> r = run(cp, 0.1, CrossSampler(Table('iris'), 4), rep=3)
>>> print(r.accuracy(), r.empty_criterion())
```

The above example uses a *CrossSampler* to define training and testing datasets. Each fold is used as the test set and the rest as a training set. The entire process is repeated three times with different fold splits and results in 3*n predictions, where n is the size of the dataset.

cp.evaluation.**run_train_test**(*cp*, *eps*, *train*, *test*, *calibrate=None*)

Fits the conformal predictor *cp* on the training dataset and evaluates it on the testing set. Inductive conformal predictors use the provided calibration set or default to extracting one third of the training set (random subset) for calibration.

**Returns** *ResultsClass* or *ResultsRegr*

### Examples

```
>>> tab = Table('iris')
>>> cp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 4)
>>> r = run_train_test(cp, 0.1, tab[:100], tab[100:])
>>> print(r.accuracy(), r.singleton_criterion())
```

cp.evaluation.**calibration_plot**(*cp*, *data*, *k=11*, *rep=1*, *title='calibration plot'*, *fname='cplot.png'*)

Draw and save a calibration plot by evaluating the conformal predictor (*cp*) with different significance values *eps* on random train/test splits. Repeat the experiment *rep* times.

### Examples

```
>>> cp = CrossClassifier(InverseProbability(LogisticRegressionLearner()), 5)
>>> calibration_plot(cp, Table('iris'))
```

## 2.4 cp.nonconformity

Nonconformity module contains nonconformity scores for classification and regression.

Structure:

- **ClassNC (classification scores)**
  - **ClassModelNC (model based)** *InverseProbability*, *ProbabilityMargin*, *SVMDistance*, *LOOClassNC*
  - **ClassNearestNeighboursNC (nearest neighbours based)** *KNNDistance*, *KNNFraction*
- **RegrNC (regression scores)**
  - **RegrModelNC (model based)** *AbsError*, *AbsErrorRF* *AbsErrorNormalized*, *LOORegrNC*, *ErrorModelNC*
  - **RegrNearestNeighboursNC (nearest neighbours based)** *AbsErrorKNN*, *AvgErrorKNN*

**class** cp.nonconformity.**ClassNC**

Bases: object

Base class for classification nonconformity scores.

Extending classes should implement *fit()* and *nonconformity()* methods.

**fit** (*data*)
Process the data used for later calculation of nonconformities.

**Parameters** **data** (*Table*) – Data set.

**nonconformity** (*instance*)
Compute the nonconformity score of the given *instance*.

**class** cp.nonconformity.**ClassModelNC** (*classifier*)
Bases: *cp.nonconformity.ClassNC*

Base class for classification nonconformity scores that are based on an underlying classifier.

Extending classes should implement *ClassNC.nonconformity()* method.

**learner**
Untrained underlying classifier.

**model**
Trained underlying classifier.

**__init__** (*classifier*)
Store the provided classifier as *learner*.

**fit** (*data*)
Train the underlying classifier on provided data and store the trained model.

**class** cp.nonconformity.**InverseProbability** (*classifier*)
Bases: *cp.nonconformity.ClassModelNC*

Inverse probability nonconformity score returns $1 - p$, where $p$ is the probability assigned to the actual class by the underlying classification model (*ClassModelNC.model*).

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> tp = TransductiveClassifier(InverseProbability(NaiveBayesLearner()), train)
>>> print(tp(test[0].x, 0.1))
```

**nonconformity** (*instance*)

**class** cp.nonconformity.**ProbabilityMargin** (*classifier*)
Bases: *cp.nonconformity.ClassModelNC*

Probability margin nonconformity score measures the difference $d_p$ between the predicted probability of the actual class and the largest probability corresponding to some other class. To put the values on scale from 0 to 1, the nonconformity function returns $(1 - d_p)/2$.

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> tp = TransductiveClassifier(ProbabilityMargin(LogisticRegressionLearner()),␣
↪train)
>>> print(tp(test[0].x, 0.1))
```

**nonconformity** (*instance*)

**class** `cp.nonconformity.`**`SVMDistance`**(*classifier*)

Bases: *`cp.nonconformity.ClassNC`*

SVMDistance nonconformity score measures the distance from the SVM's decision boundary. The score depends on the distance and the side of the decision boundary that the example lies on. Examples that lie on the correct side of the decision boundary and would therefore result in a correct prediction using the SVM classifier have a nonconformity score less than 1, while the incorrectly predicted examples have a score more than 1.

$$nc = \begin{cases} \frac{1}{1+d} & \text{correct} \\ 1+d & \text{incorrect} \end{cases}$$

The provided SVM classifier must be a sklearn's SVM classifier (SVC, LinearSVC, NuSVC) providing the decision_function() which computes the distance to decision boundary. This nonconformity works only for binary classification problems.

### Examples

```
>>> from sklearn.svm import SVC
>>> train, test = next(LOOSampler(Table('titanic')))
>>> train, calibrate = next(RandomSampler(train, 2, 1))
>>> icp = InductiveClassifier(SVMDistance(SVC()), train, calibrate)
>>> print(icp(test[0].x, 0.1))
```

**`__init__`**(*classifier*)

**`fit`**(*data*)

**`nonconformity`**(*instance*)

**class** `cp.nonconformity.`**`NearestNeighbours`**(*distance*, *k=1*)

Bases: `object`

Base class for nonconformity measures based on nearest neighbours.

**`distance`**

Distance measure.

**`k`**

*int* – Number of nearest neighbours.

**`__init__`**(*distance*, *k=1*)

Store the distance measure and the number of neighbours.

**`fit`**(*data*)

Store the data for finding nearest neighbours.

**`neighbours`**(*instance*)

Compute distances to all other data instances using the distance measure (*`distance`*).

Excludes data instances that are equal to the provided *instance*.

**Returns** List of pairs (distance, instance) in increasing order of distances.

**class** `cp.nonconformity.`**`ClassNearestNeighboursNC`**(*distance*, *k=1*)

Bases: *`cp.nonconformity.NearestNeighbours`*, *`cp.nonconformity.ClassNC`*

Base class for nearest neighbrours based classification nonconformity scores.

**class** `cp.nonconformity.`**`KNNDistance`**(*distance*, *k=1*)

Bases: *`cp.nonconformity.ClassNearestNeighboursNC`*

Computes the sum of distances to k nearest neighbours of the same class as the given instance and the sum of distances to k nearest neighbours of other classes. Returns their ratio.

### Examples

```
>>> from Orange.distance import Euclidean
>>> train, test = next(LOOSampler(Table('iris')))
>>> cp = CrossClassifier(KNNDistance(Euclidean, 10), 2, train)
>>> print(cp(test[0].x, 0.1))
```

**nonconformity**(*instance*)

**class** cp.nonconformity.**KNNFraction**(*distance*, *k=1*, *weighted=False*)
Bases: *cp.nonconformity.ClassNearestNeighboursNC*

Computes the k nearest neighbours of the given instance. Returns the fraction of instances of the same class as the given instance within its k nearest neighbours.

Weighted version uses weights $1/d_i$ based on distances instead of simply counting the instances. Non-weighted version is equivalent to using a value 1 for all weights.

### Examples

```
>>> train, test = next(LOOSampler(Table('iris')))
>>> cp = CrossClassifier(KNNFraction(Euclidean, 10, weighted=True), 2, train)
>>> print(cp(test[0].x, 0.1))
```

**__init__**(*distance*, *k=1*, *weighted=False*)

**nonconformity**(*instance*)

**class** cp.nonconformity.**LOOClassNC**(*classifier*, *distance*, *k*, *relative=True*, *include=False*, *neighbourhood='fixed'*)
Bases: *cp.nonconformity.NearestNeighbours*, *cp.nonconformity.ClassNC*

$$nc = error + (1 - p) \quad \text{or} \quad nc = \frac{1 - p}{error}$$

$p$ ... probability of actual class predicted from $N_k(z^*)$ - k nearest neighbours of the instance $z^*$

The first nonconformity score is used when the parameter `relative` is set to *False* and the second one when it is set to *True*.

$$error = \frac{\sum_{z_i \in N_k(z^*)} w_i (1 - p_i)}{\sum_{z_i \in N_k(z^*)} w_i}, \quad w_i = \frac{1}{d(x^*, x_i)}$$

$p_i$ ... probability of actual class predicted from $N_k(z') \setminus z_i$ or $N_k(z') \setminus z_i \cup z^*$ if the parameter `include` is set to *True*. $z'$ is $z^*$ if the `neighbourhood` parameter is '*fixed*' and $z_i$ if it's '*variable*'.

**__init__**(*classifier*, *distance*, *k*, *relative=True*, *include=False*, *neighbourhood='fixed'*)
Initialize the parameters.

**fit**(*data*)
Store the data for finding nearest neighbours and initialize cache.

**get_neighbourhood**(*inst*)
Construct an Orange data Table consisting of instance's k nearest neighbours. Cache the results for later calls with the same instance.

**error** (*inst*, *neighbours*)
Compute the average weighted probability prediction error for predicting the actual class of each neighbour from the other ones. Include the new example among the neighbours if the parameter `include` is True.

**nonconformity** (*inst*)

**class** `cp.nonconformity.`**RegrNC**
Bases: `object`

Base class for regression nonconformity scores.

Extending classes should implement *fit()*, *nonconformity()* and *predict()* methods.

**fit** (*data*)
Process the data used for later calculation of nonconformities.

**Parameters** **data** (*Table*) – Data set.

**nonconformity** (*instance*)
Compute the nonconformity score of the given *instance*.

**predict** (*inst*, *nc*)
Compute the inverse of the nonconformity score. Determine a range of values for which the nonconformity of the given *instance* does not exceed *nc*.

**class** `cp.nonconformity.`**RegrModelNC** (*classifier*)
Bases: *cp.nonconformity.RegrNC*

Base class for regression nonconformity scores that are based on an underlying classifier.

Extending classes should implement *RegrNC.nonconformity()* and *RegrNC.predict()* methods.

**learner**
Untrained underlying classifier.

**model**
Trained underlying classifier.

**__init__** (*classifier*)
Store the provided classifier as *learner*.

**fit** (*data*)
Train the underlying classifier on provided data and store the trained model.

**class** `cp.nonconformity.`**AbsError** (*classifier*)
Bases: *cp.nonconformity.RegrModelNC*

Absolute error nonconformity score returns the absolute difference between the predicted value ($\hat{y}$) by the underlying *RegrModelNC.model* and the actual value ($y^*$).

$$nc = |\hat{y} - y^*|$$

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> cr = CrossRegressor(AbsError(LinearRegressionLearner()), 2, train)
>>> print(cr(test[0].x, 0.1))
```

**nonconformity** (*instance*)

**predict** (*inst*, *nc*)

**class** `cp.nonconformity.`**`AbsErrorRF`**(*classifier*, *rf*, *beta=0.5*)
Bases: *cp.nonconformity.RegrModelNC*

AbsErrorRF is based on an underlying regressor and a random forest. The prediction errors of regressor are used as nonconformity scores and are normalized by the standard deviation of predictions coming from individual trees in the forest.

$$nc = \frac{|\hat{y} - y^*|}{\sigma_{RF} + \beta}$$

### Examples

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> icr = InductiveRegressor(AbsErrorRF(RandomForestRegressionLearner(),␣
↪RandomForestRegressor()))
>>> r = run(icr, 0.1, CrossSampler(Table('housing'), 10))
>>> print(r.accuracy(), r.median_range(), r.interdecile_mean())
```

**`__init__`**(*classifier*, *rf*, *beta=0.5*)
Store the classifier and beta parameter.

**`fit`**(*data*)
Train the underlying classifier on provided data and store the trained model.

**`norm`**(*inst*)
Normalization factor is equal to the standard deviation of predictions from trees in a random forest plus a constant term `beta`.

**`nonconformity`**(*inst*)

**`predict`**(*inst*, *nc*)

**class** `cp.nonconformity.`**`ErrorModelNC`**(*classifier*, *error_classifier*, *beta=0.5*, *loo=False*)
Bases: *cp.nonconformity.RegrModelNC*

ErrorModelNC is based on two underlying regressors. The first one is trained to predict the value while the second one is used for predicting logarithms of the errors made by the first one.

H. Papadopoulos and H. Haralambous. *Reliable prediction intervals with regression neural networks*. Neural Networks (2011).

$$nc = \frac{|\hat{y} - y^*|}{\exp(\mu) - 1 + \beta}$$

$\mu$ ... prediction for the value of $\log(|\hat{y} - y^*| + 1)$ returned by the second regressor

Parameter `loo` determines whether to use a leave-one-out schema for building the training set of errors for the second regressor or not.

### Examples

```
>>> nc = ErrorModelNC(SVRLearner(), LinearRegressionLearner())
>>> icr = InductiveRegressor(nc)
>>> r = run(icr, 0.1, CrossSampler(Table('housing'), 10))
>>> print(r.accuracy(), r.median_range(), r.interdecile_mean())
```

**`__init__`**(*classifier*, *error_classifier*, *beta=0.5*, *loo=False*)

**fit**(*data*)

**nonconformity**(*inst*)

**predict**(*inst*, *nc*)

**class** cp.nonconformity.**ExperimentalNC**(*rf*)
Bases: *cp.nonconformity.RegrModelNC*

**__init__**(*rf*)

**fit**(*data*)

**norm**(*inst*)

**nonconformity**(*inst*)

**predict**(*inst*, *nc*)

**class** cp.nonconformity.**AbsErrorNormalized**(*classifier*, *distance*, *k*, *gamma=0.5*, *rho=0.5*, *exp=True*, *rf=None*)
Bases: *cp.nonconformity.RegrModelNC*, *cp.nonconformity.NearestNeighbours*

Normalized absolute error prediction uses an underlying regression model to predict the value, which is then normalized by the distance and variance of the nearest neighbours.

H. Papadopoulos, V. Vovk and A. Gammerman. *Regression Conformal Prediction with Nearest Neighbours*. Journal of Artificial Intelligence Research (2011).

$$nc = \frac{|\hat{y} - y^*|}{\exp(\gamma\lambda^*) + \exp(\rho\xi^*)} \quad \text{or} \quad nc = \frac{|\hat{y} - y^*|}{\gamma + \lambda^* + \xi^*}$$

The first nonconformity score is used when the parameter `exp` is set to *True* and the second one when it is set to *False*.

$$\lambda^* = \frac{d_k(z^*)}{median(\{d_k(z), z \in T\})}, \quad d_k(z) = \sum_{z_i \in N_k(z)} distance(x, x_i)$$

$$\xi^* = \frac{\sigma_k(z^*)}{median(\{\sigma_k(z), z \in T\})}, \quad \sigma_k(z) = \sqrt{\frac{1}{k} \sum_{z_i \in N_k(z)} (y_i - \bar{y})}$$

Parameter `rf` enables the use of a random forest for computing the standard deviation of predictions instead of the nearest neighbours.

**__init__**(*classifier*, *distance*, *k*, *gamma=0.5*, *rho=0.5*, *exp=True*, *rf=None*)
Initialize the parameters.

**fit**(*data*)
Train the underlying model and precompute medians for nonconformity scores.

**_d**(*inst*)
Sum of distances to nearest neighbours.

**_lambda**(*inst*)
Normalized distance measure.

**_sigma**(*inst*)
Standard deviation of y values. This comes either from the nearest neighbours or from the predictions of individual trees in a random forest if the `rf` is provided.

**_xi**(*inst*)
Normalized variance measure.

**norm** (*inst*)
Compute the normalization factor.

**nonconformity** (*inst*)

**predict** (*inst*, *nc*)

**class** `cp.nonconformity.`**LOORegrNC** (*classifier*, *distance*, *k*, *relative=True*, *include=False*, *neighbourhood='fixed'*)
Bases: `cp.nonconformity.NearestNeighbours`, `cp.nonconformity.RegrNC`

$$nc = error + |\hat{y} - y^*| \quad \text{or} \quad nc = \frac{|\hat{y} - y^*|}{error}$$

$\hat{y}$ ... value predicted from $N_k(z^*)$

The first nonconformity score is used when the parameter `relative` is set to *False* and the second one when it is set to *True*.

$$error = \frac{\sum_{z_i \in N_k(z^*)} w_i |\hat{y_i} - y_i|}{\sum_{z_i \in N_k(z^*)} w_i}, \quad w_i = \frac{1}{d(x^*, x_i)}$$

$\hat{y_i}$ ... value predicted from $N_k(z') \setminus z_i$ or $N_k(z') \setminus z_i \cup z^*$ if the parameter `include` is set to *True*. $z'$ is $z^*$ if the `neighbourhood` parameter is '*fixed*' and $z_i$ if it's '*variable*'.

**__init__** (*classifier*, *distance*, *k*, *relative=True*, *include=False*, *neighbourhood='fixed'*)
Initialize the parameters.

**fit** (*data*)
Store the data for finding nearest neighbours and initialize cache.

**get_neighbourhood** (*inst*)
Construct an Orange data Table consisting of instance's k nearest neighbours. Cache the results for later calls with the same instance.

**error** (*inst*, *neighbours*)
Compute the average weighted error for predicting the value of each neighbour from the other ones. Include the new example among the neighbours if the parameter `include` is True.

**nonconformity** (*inst*)

**predict** (*inst*, *nc*)

**class** `cp.nonconformity.`**RegrNearestNeighboursNC** (*distance*, *k=1*)
Bases: `cp.nonconformity.NearestNeighbours`, `cp.nonconformity.RegrNC`

Base class for nearest neighbours based regression nonconformity scores.

**class** `cp.nonconformity.`**AbsErrorKNN** (*distance*, *k*, *average=False*, *variance=False*)
Bases: `cp.nonconformity.RegrNearestNeighboursNC`

Absolute error of k nearest neighbours computes the average value of the k nearest neighbours and returns an absolute difference between this average ($y_k$) and the actual value ($y^*$).

$$\bar{y} = 1/k \sum_{N_k(x^*)} y_i$$
$$nc = |\bar{y} - y^*|$$

Weighted version can normalize by average and/or variance.

$$nc = \frac{|\bar{y} - y^*|}{\bar{y} \cdot y_\sigma}$$

**average**
*bool* – Normalize by average.

**variance**
*bool* – Normalize by variance.

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> cr = CrossRegressor(AbsErrorKNN(Euclidean, 10, average=True), 2, train)
>>> print(cr(test[0].x, 0.1))
```

**__init__**(*distance, k, average=False, variance=False*)
Initialize the distance measure, number of nearest neighbours to consider and whether to normalize by average and by variance.

**stats**(*instance*)
Computes mean and standard deviation of values within the k nearest neighbours.

**norm**(*avg, std*)
Compute the normalization factor according to the chosen properties.

**nonconformity**(*instance*)

**predict**(*inst, nc*)

**class** cp.nonconformity.**AvgErrorKNN**(*distance, k=1*)
Bases: *cp.nonconformity.RegrNearestNeighboursNC*

Average error of k nearest neighbours computes the average absolute error of the actual value ($y^*$) compared to the k nearest neighbours ($y_i$).

$$nc = 1/k \sum_{N_k(x^*)} |y^* - y_i|$$

**Note:** There might be no suitable *y* values for the required significance level at the time of prediction. In such cases, the predicted range is [nan, nan].

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> cr = CrossRegressor(AvgErrorKNN(Euclidean, 10), 2, train)
>>> print(cr(test[0].x, 0.1))
```

**avg_abs**(*y, ys*)

**avg_abs_inv**(*nc, ys*)

**nonconformity**(*instance*)

**predict**(*inst, nc*)

## 2.5 cp.regression

Regression module contains methods for conformal regression.

Conformal regressors predict a range of values (not always a single value) under a given significance level (error rate). Every regressors works in combination with a nonconformity measure and on average predicts the correct value with the given error rate. Lower error rates result in narrower ranges of predicted values.

Structure:

- **ConformalRegressor**
  - Inductive (*InductiveRegressor*)
  - Cross (*CrossRegressor*)

**class** cp.regression.**PredictionRegr**(*lo*, *hi*)

Bases: object

Conformal regression prediction object, which is produced by the *ConformalRegressor.predict()* method.

**lo**

*float* – Lowest value of the predicted range.

**hi**

*float* – Highest value of the predicted range.

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> ccr = CrossRegressor(AbsError(LinearRegressionLearner()), 5, train)
>>> prediction = ccr.predict(test[0].x, 0.1)
>>> print(prediction.width())
```

**__init__**(*lo*, *hi*)

Initialize the prediction.

**Parameters**

- **lo** (*float*) – Lowest value of the predicted range.
- **hi** (*float*) – Highest value of the predicted range.

**range**()

Predicted range: *lo*, *hi*.

**verdict**(*ref*)

Conformal regression prediction is correct when the actual value appears in the predicted range.

**Parameters** **ref** – Reference/actual value

**Returns** True if the prediction is correct.

**width**()

Width of the predicted range: *hi* - *lo*.

**class** cp.regression.**ConformalRegressor**(*nc_measure*)

Bases: *cp.base.ConformalPredictor*

Base class for conformal regression.

**__init__**(*nc_measure*)
Verify that the nonconformity measure can be used for regression.

**predict**(*example*, *eps*)
Compute a regression prediction object for a given example and significance level.

Function determines what is the `eps`-th lowest nonconformity score and computes the range of values that would result in a lower or equal nonconformity. This inverse of the nonconformity score is computed by the nonconformity measure's `cp.nonconformity.RegrNC.predict()` function.

**Parameters**

- **example** (*ndarray*) – Attributes array.
- **eps** (*float*) – Default significance level (error rate).

**Returns** Regression prediction object.

**Return type** *PredictionRegr*

**__call__**(*example*, *eps*)
Compute predicted range for a given example and significance level.

**Parameters**

- **example** (*ndarray*) – Attributes array.
- **eps** (*float*) – Significance level (error rate).

**Returns** Predicted range as a pair (*PredictionRegr.lo*, *PredictionRegr.hi*)

**class** cp.regression.**TransductiveRegressor**(*nc_measure*)
Bases: `cp.regression.ConformalRegressor`

Transductive regression. TODO

**class** cp.regression.**InductiveRegressor**(*nc_measure*, *train=None*, *calibrate=None*)
Bases: `cp.regression.ConformalRegressor`

Inductive regression.

**alpha**
Nonconformity scores of the calibration instances. Computed by the `fit()` method. Must be *sorted* in increasing order.

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> train, calibrate = next(RandomSampler(train, 2, 1))
>>> icr = InductiveRegressor(AbsError(LinearRegressionLearner()), train,␣
↪calibrate)
>>> print(icr(test[0].x, 0.1))
```

**__init__**(*nc_measure*, *train=None*, *calibrate=None*)
Initialize inductive regressor with a nonconformity measure, training set and calibration set. If present, fit the conformal regressor to the training set and compute the nonconformity scores of calibration set.

**Parameters**

- **nc_measure** (RegrNC) – Regression nonconformity measure.
- **train** (*Optional[Table]*) – Table of examples used as a training set.
- **calibrate** (*Optional[Table]*) – Table of examples used as a calibration set.

**fit** (*train*, *calibrate*)
Fit the conformal regressor to the training set, compute and store sorted nonconformity scores (`alpha`) on the calibration set and store the domain.

**Parameters**

- **train** (`Optional[Table]`) – Table of examples used as a training set.
- **calibrate** (`Optional[Table]`) – Table of examples used as a calibration set.

**class** `cp.regression.`**CrossRegressor** (*nc_measure*, *k*, *train=None*)
Bases: `cp.regression.InductiveRegressor`

Cross regression.

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> ccr = CrossRegressor(AbsError(LinearRegressionLearner()), 4, train)
>>> print(ccr(test[0].x, 0.1))
```

**__init__** (*nc_measure*, *k*, *train=None*)
Initialize cross regressor with a nonconformity measure, number of folds and training set. If present, fit the conformal regressor to the training set.

**Parameters**

- **nc_measure** (`RegrNC`) – Regression nonconformity measure.
- **k** (`int`) – Number of folds.
- **train** (`Optional[Table]`) – Table of examples used as a training set.

**fit** (*train*)
Fit the cross regressor to the training set. Split the training set into k folds for use as training and calibration set with an inductive regressor. Concatenate the computed nonconformity scores and store them (`InductiveRegressor.alpha`).

**Parameters** **train** (`Table`) – Table of examples used as a training set.

**class** `cp.regression.`**LOORegressor** (*nc_measure*, *train=None*)
Bases: `cp.regression.CrossRegressor`

Leave-one-out regressor is a cross conformal regressor with the number of folds equal to the size of the training set.

### Examples

```
>>> train, test = next(LOOSampler(Table('housing')))
>>> ccr = LOORegressor(AbsError(LinearRegressionLearner()), train)
>>> print(ccr(test[0].x, 0.1))
```

**__init__** (*nc_measure*, *train=None*)

**fit** (*train*)

## 2.6 cp.utils

Utils module contains various utility functions that are used in different parts of the conformal prediction library.

`cp.utils.`**`get_instance`**`(`*data*, *i*`)`
Extract a single instance from data as a test instance and return the remainder as a training set.

`cp.utils.`**`split_data`**`(`*data*, *a*, *b*`)`
"Split data in approximate ratio a:b.

`cp.utils.`**`shuffle_data`**`(`*data*`)`
Randomly shuffle data instances.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

Bibliography

[Vovk08] Glenn Shafer, Vladimir Vovk. Tutorial on Conformal Predictions. *Journal of Machine Learning Research* 9 (2008) 371-421

[Shafer05] Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*. Springer, New York, 2005.

# Python Module Index

## C

## Symbols



## A



## C

## D



## E



## F



## G



## H



## I



## K



## L



## M



## N

## P



## R



## S



## T



## V



## W