libconform v0.1.0: a Python library for conformal prediction

Jonas Faßbender jonas@fassbender.dev

1. Introduction

This paper introduces the Python library `libconform`, implementing concepts defined in Vovk et al. (2005), namely the conformal prediction framework and Venn prediction for reliable machine learning and predicting with certainty. These algorithms address a weakness of more traditional machine learning algorithms which produce only bare predictions, without their confidence in them/the probability of the prediction, therefore providing no measure of likelihood, desirable and even necessary in many real-world application domains.

The conformal prediction framework is composed of variations of the conformal prediction algorithm, first described in Vovk et al. (1999); Saunders et al. (1999). A conformal predictor provides a measurement of confidence in its predictions. A Venn predictor, on the other hand, provides a multi-probabilistic measurement, making it a probabilistic predictor. Below in the text, Venn predictors are included if only "conformal prediction framework" is written, except stated otherwise.

The conformal prediction framework is applied successfully in many real-world domains, for example face recognition, medical diagnostic and prognostic and network traffic classification (see Balasubramanian et al., 2014, Part 3).

It is build on traditional machine learning algorithms, the so called underlying algorithms (see Papadopoulos et al., 2007), which makes Python the first choice for implementation, since its machine learning libraries are top of the class, still evolving and improving due to the commitment of a great community of developers and researchers.

`libconform`'s aim is to provide an easy to use, but very extensible API for the conformal prediction framework, so developers can use their preferred implementations for the underlying algorithm and can leverage the library, even in this early stage. `libconform v0.1.0` is **not** yet stable; there are still features missing and the API is very likely to change and improve. The library is licensed under the MIT-license and its source code can be downloaded from `https://github.com/jofas/conform`.

This paper combines `libconform`'s documentation with an outline of the implemented algorithms. At the end of each chapter there are notes on the implementation containing general information about the library, descriptions of the internal workings and the API and possible changes in future versions.

Appendix A provides an overview over `libconform`'s API and Appendix B contains examples on how to use the library.

2. Conformal predictors

Like stated in the introduction, this chapter will only outline conformal prediction (CP). For more details see Vovk et al. (2005).

CP—like the name suggests—determines the label(s) of an incoming observation based on how well it/they conform(s) with previous observed examples. Conformal prediction can be used either in the online or the offline—or batch—learning setting. The offline learning setting, compared to online learning, weakens the validity and efficiency—described below in this chapter—of the classifier in favor of computational efficiency (see Vovk et al., 2005, Chapter 4).

Let $\wr z_1, \dots, z_n \int$ be a bag, also called multiset[1], of examples, where each example $z_i \in \mathbf{Z}$ is a tuple $(x_i, y_i); x_i \in \mathbf{X}, y_i \in \mathbf{Y}$. $\mathbf{X}$ is called the observation space and $\mathbf{Y}$ the label space. For this time $\mathbf{Y}$ is considered finite, making the task of prediction a classification task, rather than regression, which will be considered in Chapter 2.2.

A conformal predictor can be defined as a confidence predictor Γ. For this an input $\epsilon \in (0, 1)$, the significance level is needed. $1 - \epsilon$ is called the confidence level. A conformal predictor Γ^ϵ in the online setting is conservatively valid under the exchangeability assumption, which means, as long as exchangeability holds, it makes errors at a frequency of ϵ or less. For more on that refer to Vovk et al. (2005, Chapters 1–4, 7).

CP, in its original setting, produces nested prediction sets. Rather than returning a single label as its prediction, it returns a set of elements $\mathbf{Y}' \in 2^{\mathbf{Y}}$; $2^{\mathbf{Y}}$ being the set of all subsets of $\mathbf{Y}$, including the empty set. The prediction sets are called nested, because, for $\epsilon_1 \geq \epsilon_2$, the prediction set of Γ^{ϵ_1} is a subset of Γ^{ϵ_2} (see Vovk et al., 2005, Chapter 2).

The efficiency of a classifier is the frequency with which it classifies new observations with a single label.

In order to predict the label of a new observation x_{n+1}, set $z_{n+1} := (x_{n+1}, y)$, for each $y \in \mathbf{Y}$ and check how z_{n+1} conforms with the examples of our bag $\wr z_1, \dots, z_n \int$.

This is done with a nonconformity measure $A_{n+1} : \mathbf{Z}^n \times \mathbf{Z} \to \mathbb{R}$. First, z_{n+1} is added to the bag, then A_{n+1} assigns a numerical score to each example in z_i:

$$\alpha_i = A_{n+1}(\wr z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_{n+1} \int, z_i). \tag{1}$$

One can see in this equation that z_i is removed from the bag. It is also possible to compute α_i with z_i in the bag, which means for $A_{n+1} : \mathbf{Z}^{n+1} \times \mathbf{Z} \to \mathbb{R}$ the score is computed as:

$$\alpha_i = A_{n+1}(\wr z_1, \dots, z_{n+1} \int, z_i). \tag{2}$$

[1] It is typical in machine learning to denote this as a data set, even though examples do not have to be unique, making the so called set a multiset. A multiset is not a list, since the ordering of the elements is not important.

Which one is preferable is case-dependent (see Shafer and Vovk, 2008, Chapter 4.2.2).

α_i is called nonconformity score. The nonconformity score can now be used to compute the p-value for z_{n+1}, which is the fraction of examples from the bag which are at least as nonconforming as z_{n+1}:

$$\frac{|\{i = 1, \dots, n+1 : \alpha_i \geq \alpha_{n+1}\}|}{n+1}. \tag{3}$$

Another way to determine the p-value is through smoothing, in which case the nonconformity scores equal to α_{n+1} are multiplied by a random value τ_{n+1}:

$$\frac{|\{i = 1, \dots, n+1 : \alpha_i > \alpha_{n+1}\}| + \tau_{n+1}|\{i = 1, \dots, n+1 : \alpha_i = \alpha_{n+1}\}|}{n+1}. \tag{4}$$

A conformal predictor using the smoothed p-value is called a smoothed conformal predictor and is exactly valid under exchangeability in the online setting, which means it makes errors at a rate exactly ϵ (see Vovk et al., 2005, Chapter 2). If the p-value of z_{n+1} is larger than ϵ, y is added to the prediction set.

Algorithm 1 : Conformal predictor $\Gamma^{\epsilon}(\wr z_1, \dots, z_n \wr, x_{n+1})$

```
for all $y \in \mathbf{Y}$ do
    set $z_{n+1} := (x_{n+1}, y)$ and add it to the bag
    for all $i = 1, \dots, n+1$ do
        compute $\alpha_i$ with (1) or (2)
    end for
    set $p_y$ with (3) or (4)
    if $p_y > \epsilon$ then
        add $p_y$ to prediction set
    end if
end for
return prediction set
```

Notes on the implementation: `libconform` provides the `CP` class for creating conformal prediction classifiers. `libconform`'s classifier classes provide quite equal APIs, only with minor variations. The API of the classifier classes is comparable to major machine learning libraries like sklearn or keras (see Buitinck et al., 2013; Chollet et al., 2015).

It is common in machine learning to split the learning task in two distinct operations, first a training—or fit—operation on a bag of examples and then a predict operation on new observations. `libconform`'s classifier classes follow this style, providing a `train` and a `predict` method.

While this split in training and predicting is common for inductive classifiers, which first derive a prediction rule, or decision surface, from the training bag and then predict unseen examples inductively based on that rule, it is not really the way CP works. CP was designed to be transductive, not inductive, which means rather than to generate a prediction rule, it uses all previous seen examples to classify a new observation, making the training step unnecessary (see Algorithm 1). While the transductive setting is more elegant than the inductive setting, it is computationally very expensive and not feasible for larger bags of examples and for underlying algorithms—discussed in Chapter 2.1—which have a computationally complex training phase (see Papadopoulos et al., 2007; Vovk et al., 2005, Chapter 1).

`libconform`'s aim is to be—one day—ready for production, where, for some application domains, the time complexity of predicting a new observation is crucial, while the time complexity of the training phase is—in a certain range—not as important. Therefore `libconform`'s `CP` class tries to minimize the time complexity of its `predict` method. Instead of adding z_{n+1} to the bag and then computing α_i for each example in the bag during prediction, it computes $\alpha_1, \dots, \alpha_n$ during training and only computes α_{n+1} in `predict` (see Algorithm 1, lines 3–5). Therefore—not adding z_{n+1} to the bag—it currently computes the nonconformity scores based on A_n instead of A_{n+1}.

Arguably `CP` does not implement the conformal prediction algorithm in its original form, being currently rather a special case of inductive conformal prediction, where the calibration set is equal to the whole bag of examples previously witnessed, instead of a subset (see Chapter 3). It is possible that `CP` will change to being the implementation of the original conformal prediction algorithm in a future version, or simply providing an extra method for the computationally more demanding original online learning setting Vovk et al. (see 2005, Chapter 2).

`CP` takes an instance of a nonconformity measure A and a sequence of $\epsilon_1, \dots, \epsilon_g$ as its arguments during initialization, therefore being the implementation of $\Gamma^{\epsilon_1}, \dots, \Gamma^{\epsilon_g}$.

It also provides to extra utility methods for validation, `score` and `score_online`, which generate metrics for the conformal predictors $\Gamma^{\epsilon_1}, \dots, \Gamma^{\epsilon_g}$. The most important of those metrics are the error rates $Err_1, \dots, Err_g$. It the error rate $Err_i \leq \epsilon_i$ over the bag of examples provided to `score`/`score_online` than Γ^{ϵ_i} was valid on the bag.

`score_online` adds an example, after it was predicted, to the training bag and calls `train`, using $\Gamma^{\epsilon_1}, \dots, \Gamma^{\epsilon_g}$ in the online learning setting.

`CP` also provides a method for another setting of conformal prediction, this one not based on a significance level ϵ: `predict_best`. `predict_best` always returns a single label, the one with the highest p-value and optionally also its significance level. The significance level is the second highest p-value, since a label is added to the prediction set—in the original setting—if its p-value is greater than ϵ (see Papadopoulos et al., 2007).

2.1 Nonconformity measures based on underlying algorithms

Previously nonconformity measures were only described as any function $A : \mathbf{Z}^* \times \mathbf{Z} \to \mathbb{R}$, $\mathbf{Z}^*$ being any possible bag of examples from $\mathbf{Z}$. This chapter will make a more concrete description on what nonconformity measures are and how they use underlying traditional machine learning algorithms.

Let $D : \mathbf{Z}^* \times \mathbf{X} \to \hat{\mathbf{Y}}$ be a traditional machine learning algorithm. $\hat{\mathbf{Y}}$ must not be equal to $\mathbf{Y}$. Furthermore there exists a discrepancy measure $\Delta : \mathbf{Y} \times \hat{\mathbf{Y}} \to \mathbb{R}$. For a concrete bag $\wr z_1, \dots, z_n \int$, $D_{\wr z_1,\dots,z_n \int}$ would be the instance of D trained on the bag, generating a decision surface based on it. $D_{\wr z_1,\dots,z_n \int}(x)$ would return the label $\hat{y}$ for x. Now we can define the nonconformity score α for $z := (x, y)$ from the nonconformity measure A_n as:

$$\alpha = A_n(\wr z_1, \dots, z_n \int, z) = \Delta\big(y, D_{\wr z_1,\dots,z_n \int}(x)\big),$$

or rather with removed example for any α_i, $i = 1, \dots, n$:

$$\alpha_i = A_n(\wr z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_n \int, z_i) = \Delta\big(y_i, D_{\wr z_1,\dots,z_{i-1},z_{i+1},\dots,z_n \int}(x_i)\big).$$

Especially the second equation can be computationally very complex since it requires to refit D for each $i = 1, \dots, n$. In general it is not very natural to use a inductive decision surface D within the transductive framework of CP.

A popular nonconformity measure is based on the nearest neighbor method (see Vovk et al., 2005; Shafer and Vovk, 2008; Balasubramanian et al., 2014; Smirnov et al., 2009). The general description for the k-nearest neighbor method can be found in Smirnov et al. (2009), the other articles/books describe the nonconformity measure based on the 1-nearest neighbor method for $z := (x, y)$ as:

$$A_n(\wr z_1, \dots, z_n \int, z) = \frac{\min_{i=1,\dots,n: y_i = y} d(x, x_i)}{\min_{i=1,\dots,n: y_i \neq y} d(x, x_i)},$$

d being a distance measure, for example the Euclidean distance. It should be noted that A_n based on the 1-nearest neighbor method for $A_n(\wr z_1, \dots, z_n \int, z_i), i = 1, \dots, n$ requires the removal of z_i from the bag, since otherwise the smallest distance for $y_i = y_j, j = 1, \dots, n$ would always be 0 resulting in worthless nonconformity scores.

The more general nonconformity measure based on the k-nearest neighbor method can be described as:

$$A_n(\wr z_1, \dots, z_n \int, z) = \frac{d_k^y}{d_k^{-y}},$$

d_k being the sum of the k smallest distances to x, $-y$ being all the examples where $y \neq y_i, i = 1, \dots, n$.

Notes on the implementation: `libconform` tries again to be as extensible as possible, providing a way for developers to define their own nonconformity measures. For nonconformity measures `libconform` provides a module `ncs` containing predefined nonconformity measures and a base class for inheritance called `NCSBase`.

Predefined are currently the k-nearest neighbor method, one based on a decision tree (see Vovk et al., 2005, Chapter 4) and one based on neural networks (see Papadopoulos et al., 2007; Vovk et al., 2005, Chapter 4). The k-nearest neighbor method and the decision tree are based on the sklearn library (see Buitinck et al., 2013).

Nonconformity measures are classes inheriting from `NCSBase` and have to provide an interface with three methods: `train`, `scores` and `score`.

`train`: $\mathbf{X}^n \times \mathbf{Y}^n$ is for fitting the underlying algorithm D to a bag of examples.

`scores`: $\mathbf{X}^m \times \mathbf{Y}^m \times \mathit{bool} \to \mathbb{R}^m$ returning the scores for a bag of examples. The *bool* value provided as a parameter tells the nonconformity measure if the bag is equal to the bag provided to `train`, making it possible to implement (1), rather than (2). The CP-implementation passes the same bag to `train` and `scores`, while the inductive conformal prediction implementation (see Chapter 3) passes another bag—the so called calibration set—as a parameter to `scores`.

`score`: $\mathbf{X} \times \mathbf{Y}^{|\mathbf{Y}|} \to \mathbb{R}^{|\mathbf{Y}|}$ is for returning the scores of an example x and each $y \in \mathbf{Y}$ combined as $z := (x, y)$.

2.2 Conformal predictor for regression: ridge regression confidence machine

The ridge regression confidence machine algorithm is described in Nouretdinov et al. (2001); Vovk et al. (2005, Chapter 2.3). In this chapter, unlike in the previous and following chapters, $\mathbf{Y}$ will be $\mathbb{R}$, making the prediction a regression problem instead of classification.

Algorithm 1 is not feasible for regression, since $\mathbf{Y}$ is now infinite and we would need to test for each $y \in \mathbf{Y}$ if it is in the prediction set or not. Instead the ridge regression confidence machine (RRCM) algorithm offers a different approach, returning prediction intervals instead of prediction sets.

Even though RRCM has ridge regression in its name, it can be used with other underlying algorithms, like nearest neighbor regression. For more on ridge regression and its special case linear regression refer to e.g. Hastie et al. (2009, Chapter 3).

Let $\wr z_1, \dots, z_n \int$ be our bag of examples, let $z_{n+1} := (x_{n+1}, y)$ be the observation we want to predict and let $D_{\wr z_1, \dots, z_{n+1} \int}$ be an underlying regression algorithm. Previously nonconformity scores were treated as constants, now we treat them as functions, since y is now an unknown variable: $\alpha_i(y) = |a_i + b_i y|, i = 1, \dots, n+1$. a_i and b_i are provided by the underlying regression algorithm. Each b_i is always positive, if not a_i and b_i are multiplied with -1.

Now we can compute the set of y's which p-values are exceeding a significance level ϵ. Let $S_i = \{y : |a_i + b_i y| \geq |a_{n+1} + b_{n+1} y|\}, i = 1, \dots, n$. Each S_i looks like:

$$S_i = \begin{cases} [u_i, v_i] & \text{if } b_{n+1} > b_i \\ (-\infty, u_i] \cup [v_i, \infty) & \text{if } b_{n+1} < b_i \\ [u_i, \infty) & \text{if } b_{n+1} = b_i > 0 \text{ and } a_{n+1} < a_i \\ (-\infty, u_i] & \text{if } b_{n+1} = b_i > 0 \text{ and } a_{n+1} > a_i \\ \mathbb{R} & \text{if } b_{n+1} = b_i = 0 \text{ and } |a_{n+1}| \leq |a_i| \\ \emptyset & \text{if } b_{n+1} = b_i = 0 \text{ and } |a_{n+1}| > |a_i| \end{cases},$$

so each S_i is either an interval, a point (a special interval), the union of two rays, a ray, the real line or empty. u_i and v_i are either the minimum/maximum of $-\frac{a_i - a_{n+1}}{b_i - b_{n+1}}$ and $-\frac{a_i + a_{n+1}}{b_i + b_{n+1}}$, if $b_{n+1} \neq b_i$ or $u_i = v_i = -\frac{a_i + a_n}{2b_i}$, if $b_{n+1} = b_i > 0$ The p-value can only change at u_i or v_i. Therefore all u_i and v_i are sorted in ascending order generating the sequence $s_1, \dots, s_m$ plus two more s-values, $s_0 = -\infty$, $s_{m+1} = \infty$. The p-value is constant on any interval $(s_i, s_{i+1}), i = 0, \dots, m$ from the sorted set (see Nouretdinov et al., 2001).

After that N and M are computed from the sequence. $N_j, j = 0, \dots, m$ for the interval (s_j, s_{j+1}) is the count of $S_i : (s_j, s_{j+1}) \subseteq S_i, i = 1, \dots, n$. $M_j, j = 1, \dots, m$, on the other hand, does the same count only for single s_j: $S_i : s_j \in S_i, i = 1, \dots, n$.

For a given significance level ϵ the prediction interval is the union of intervals from N and points from M for which $\frac{N_j}{n+1} > \epsilon$ or $\frac{M_j}{n+1} > \epsilon$, respectively (see Vovk et al., 2005, Chapter 2.3).

In Nouretdinov et al. (2001) it is stated that there could be holes in the prediction interval, which means the the RRCM would return more than a single prediction interval. According to the authors these holes rarely show in empirical tests. The authors therefore remark that the RRCM can just remove those holes—therefore returning a single interval—by simply returning the convex hull of the prediction intervals.

Notes on the implementation: The ridge regression confidence machine is implemented as a class `RRCM`. It provides the same API as `CP`. It implements a computationally less complex prediction method than the one described above. While the RRCM described above runs at $\mathcal{O}(n^2)$, `RRCM` takes only $\mathcal{O}(n \log n)$, because it does not compute N and M directly but instead

$$N'_j = \begin{cases} N_j - N_{j-1} & \text{if } j = 0, \dots, m \\ 0 & \text{if } j = -1 \end{cases}$$

and

$$M'_j = \begin{cases} M_j - M_{j-1} & \text{if } j = 1, \dots, m \\ 0 & \text{if } j = 0 \end{cases},$$

which takes only $\mathcal{O}(n)$, making the sorting the u_i's and v_i's the most complex task (see Vovk et al., 2005, Chapter 2.3).

The `RRCM` implementation takes a flag during its initialization for dealing with the holes described above, so developers can choose if they want possibly more than one prediction interval or the convex hull.

`RRCM` is based on underlying regression algorithms providing it with its a_i and b_i. Currently the library provides one of these regression algorithms, based on the k-nearest neighbor method. a_i is the difference between y_i and the average of the labels of its k-nearest neighbors. y_{n+1} is set to 0, therefore the k-nearest neighbor method returns the negated average of the labels of the k-nearest neighbors of x_{n+1} as a_{n+1}. For $b_i, i = 1, \dots, n$ it returns 0, for b_{n+1} it returns 1.

For developing underlying regression scorers there exists a base class for inheritance called `NCSBaseRegressor`. It provides a comparable API to `NCSBase`—the base class for nonconformity measures—described in the previous chapter. Like `NCSBase` the API contains a `train` method for training the underlying algorithm. Instead of `scores` and `score` it has `coeffs` and `coeffs_n`. The first returns for a bag two vectors of coefficients a_i and b_i for each element in the bag. `coeffs_n` returns the coefficients for the observation that needs to be predicted, in this chapter $z_{n+1} := (x_{n+1}, y)$.

3. Inductive conformal predictors

Suppose we have an underlying inductive machine learning algorithm D as our nonconformity measure and a bag $\wr z_1, \dots, z_n \int$ of examples. If we want to use D as the nonconformity measure we need to fit it to our bag: $D_{\wr z_1, \dots, z_n \int}$. For some D this can be a quite time consuming task and in general is not a very aesthetic thing to do in our transductive setting from the previous chapter, since—if we would want to predict a new observation x_{n+1}—we would need to refit D for each $y \in \mathbf{Y}$, because we compute our nonconformity scores adding $z_{n+1} := (x_{n+1}, y)$ to the bag and refitting D with it (see Algorithm 1, lines 2–5). Even worse, if we would use (1) instead of (2) we would need to refit D for each bag $\wr z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_{n+1} \int, i = 1, \dots, n$.

There exists a natural derivation from the transductive setting of conformal prediction to the inductive setting called inductive conformal prediction (ICP). ICP works more natural with nonconformity measures relying on inductive machine learning algorithms as the underlying algorithm (see Vovk et al., 2005, Chapter 4).

ICP is computationally less complex than CP, to the cost of the classifier's validity and efficiency (see Vovk et al., 2005, Chapter 4).

Suppose, again, we have our bag of examples $\wr z_1, \dots, z_n \int$. ICP now splits this bag at a point $m < n$ into two bags, the so called training set $\wr z_1, \dots, z_m \int$ and the calibration set $\wr z_{m+1}, \dots, z_n \int$.

With the training set the underlying algorithm is trained generating $D_{\wr z_1,\dots,z_m \wr}$. For each example in the calibration set the nonconformity score α_i gets computed:

$$\alpha_i = \Delta(y_i, D_{\wr z_1,\dots,z_m \wr}(x_i)), i = m+1,\dots,n. \tag{5}$$

Now, for an incoming example $x_g, g \in [n+1, \infty)$ set $z_g := (x_g, y)$ for each $y \in \mathbf{Y}$ and compute the nonconformity score α_g like (5). The p-value of z_g is:

$$\frac{|\{i = m+1,\dots,n : \alpha_i \geq \alpha_g\}|}{n - m + 1}$$

(see Papadopoulos et al., 2007).

The huge costs of fitting D repetitively are now reduced to fitting D only once. More elaborate update cycles—called teaching schedules—where m is changing after certain events and how they impact the validity of the classifier can be found in Vovk et al. (2005, Chapters 4.3, 4.4).

Notes on the implementation: `ICP` is the class implemention inductive conformal prediction. It provides the same API as `CP`, except `score_online`. It has an additional method `calibrate` for generating the nonconformity scores for a the calibration set.

It works with the same nonconformity measures (instances of classes inheriting from `NCSBase`) as does `CP`.

Currently the nonconformity scores from the calibration set are saved internally as a vector. In future releases this will change to an optimized data structure for searching, e.g. a red-black tree.

4. Mondrian or conditional (inductive) conformal predictors

The property of validity under the exchangeablity assumption can be further optimized with mondrian or conditional (inductive) conformal prediction (MCP). In Vovk et al. (2005, Chapter 4.5) this form of conformal prediction is called mondrian, in Balasubramanian et al. (2014, Chapter 2) it is called conditional.

An example from Vovk et al. (2005, Chapter 4.5) makes it clear why the stronger form of validity provided by MCP can be important for some real-world application domains. The authors tested a 1-nearest neighbor based smoothed conformal predictor with the significance level $\epsilon = 0.05$ on the USPS data set. The USPS data set contains 9298 images of handwritten digits. The observations are a 16×16 matrix where each cell is in the interval of $(-1, 1)$ and the labels obviously are 0 to 9 (see LeCun et al., 1989). They found out, that while overall the validity (the error frequency was nearly equal to $\epsilon = 0.05$) held, the smoothed conformal predictor had an error rate of 11.7% on examples with the label "5". The smoothed conformal predictor masked its bad performance on examples with the label "5" simply by predicting other labels with an error rate of less than $\epsilon = 0.05$, e.g. for the label "0" the error rate was below 0.01 (see Vovk et al., 2005, Chapter 4.5).

The idea of MCP is to partition the examples into a discrete and finite set $\mathbf{K}$ of categories $k_i \in \mathbf{K}$ and to achieve conditional validity in each category. For the partitioning a measurable function called a taxonomy is used. In Vovk et al. (2005, Chapter 4.5) the taxonomy is called mondrian taxonomy and is defined as:

$$\kappa : \mathbb{N} \times \mathbf{Z} \to \mathbf{K},$$

in Balasubramanian et al. (2014, Chapter 2) the taxonomy is called a n-taxonomy:

$$K_n : \mathbf{Z}^n \to \mathbf{K}^n.$$

The mondrian taxonomy κ takes the index i of an example z_i from a sequence $z_1, \dots, z_n$ and z_i as its input and maps it to a category while the n-taxonomy K_n takes a sequence of examples with a size of n and maps it to a sequence of categories with size n. K_n is more flexible than κ since it is possible to make the decision which category an example from the sequence should be in based on the other examples from the sequence.

The K-conditional p-value for an example z_{n+1} and a bag $\wr z_1, \dots, z_n \int$ is now defined for $i = 1, \dots, n+1$ as:

$$\frac{|\{i : K_i = K_{n+1} \ \& \ \alpha_i \geq \alpha_{n+1}\}|}{|\{i : K_i = K_{n+1}\}|}. \tag{6}$$

The smoothed version would be:

$$\frac{|\{i : K_i = K_{n+1} \ \& \ \alpha_i > \alpha_{n+1}\}| + \tau_{n+1}|\{i : K_i = K_{n+1} \ \& \ \alpha_i = \alpha_{n+1}\}|}{|\{i : K_i = K_{n+1}\}|}. \tag{7}$$

(6) and (7) are the same for κ if K is substituted with κ.

A MCP classifier is category-wise valid under the exchangeability assumption (see Vovk et al., 2005; Balasubramanian et al., 2014).

Notes on the implementation: There is no direct implementation for MCP, `libconform` rather leverages the fact that CP and ICP are just a special form of mondrian (inductive) conformal prediction, where $|\mathbf{K}| = 1$, which means all examples are in the same category. `CP` and `ICP` can take an argument during initialization called `mondrian_taxonomy`. Currently `mondrian_taxonomy` is a function—or a Python callable rather—which takes one example as its input and returns the category, basically a 1-taxonomy K_1 where the single example can only be looked at without context.

In practice a single example often is more information than really needed. Often just the label of the example is important, making the MCP based on this $\mathbf{K}$ a label conditional (inductive) conformal predictor (see Balasubramanian et al., 2014, Chapter 2).

mondrian_taxonomoy will change in a future version to K_n for more flexibility.

Vovk et al. (2005, Chapter 4.5) defines mondrian nonconformity measures

$$A : \mathbf{K}^* \times \mathbf{Z}^* \times \mathbf{K} \times \mathbf{Z} \to \mathbb{R},$$

which add the category to each example in order to compute the nonconformity scores. Currently libconform does not have an API for mondrian nonconformity measures, which could change in future releases.

5. Multi-probabilistic prediction: Venn predictors

In the previous chapters we measured the likelihood of a prediction based on p-values. Even though they produce valid confidence predictions, the use of p-values is controversial and they have disadvantages compared to the probability of a prediction, namely that they are harder to reason about and that they are often confused with probabilities (see Vovk et al., 2005, Chapter 6.3).

The main negative property of probabilistic prediction is the fact that it is impossible to estimate true probabilities—under the unconstrained randomness assumption—from a finite bag of examples, if the objects of the bag do not precisely repeat themselves (see Vovk et al., 2005, Chapter 5).

To bypass this property and to achieve a notion of validity, Venn predictors produce a set of probability distributions $\{P_y | y \in \mathbf{Y}\}$, $|\mathbf{Y}| < \infty$ as their predictions, for which reason they are called multi-probabilistic predictors (see Balasubramanian et al., 2014, Chapter 2.8).

There are two definitions of validity for a Venn predictor, the stronger form of validity being that Venn predictors are "well-calibrated" (see Vovk et al., 2005, Chapter 6), while the weaker form of validity states that—under the unconstrained randomness assumption—a Venn predictor's prediction is guaranteed to contain the conditional probability in its multi-probabilistic prediction with regard to the true probability distribution generating the examples (see Balasubramanian et al., 2014, Chapter 2.8).

A Venn predictor is based on a Venn taxonomy V_n equal to the n-taxonomy described in the previous chapter. Now, suppose we want to predict the probability distribution for $z_{n+1} := (x_{n+1}, y); y \in \mathbf{Y}$. We first determine the category $K \in \mathbf{K}$ of z_{n+1} with V_{n+1} and then look at the frequency of y in this particular category to generate the probability distribution:

$$P_y = \frac{|\{(x_i, y_i) \in K : y_i = y\}|}{|K|}.$$

K is not empty since it at least contains z_{n+1}. This is done for each $y \in \mathbf{Y}$ generating the set of probability distributions $\{P_y | y \in \mathbf{Y}\}$ which is returned as the multi-probability prediction. The label with the highest probability is the predicted label.

An example for such a Venn predictor is described in Vovk et al. (2005, Chapter 6.3). It is based on a Venn taxonomy using the 1-nearest neighbor method to map an example to a category. In this case the Venn taxonomy returns the label of the nearest neighbor as the category.

The Venn predictor generates a matrix $M : |\mathbf{Y}|\times|\mathbf{Y}|$. Each example $z_{n+1} := (x_{n+1}, y); y \in \mathbf{Y}$ is mapped to a row. Each column contains the frequency of $y_i \in \mathbf{Y}$ of all examples in the same category as z_{n+1}.

The quality of a column is its minimum entry. Now select the best column M_{best} (with the highest quality) an return the label of the cell with the highest frequency as the label prediction and the column as the multi-probability prediction $\{P_y|y \in \mathbf{Y}\}$ (see Vovk et al., 2005, Chapter 6.3).

Actually, in the example in Vovk et al. (2005, Chapter 6.3), instead of returning $\{P_y|y \in \mathbf{Y}\}$ the Venn predictor returns the interval of the convex hull of the multi-probability prediction: $[min\ M_{best},\ max\ M_{best}]$. This is called the probability interval. The complementary interval $[1 - max\ M_{best},\ 1 - min\ M_{best}]$ is called the error probability interval.

Notes on the implementation: Venn prediction—like described above—is implemented as `Venn`. It again implements the same API as `CP` does. The only difference is that `Venn`'s `predict` method takes a flag `proba`. If `proba` is off only the label prediction is returned. On the other hand, it `proba` is set, the label prediction and the error probability interval is returned.

Currently Venn taxonomies have their own module `vtx`. A Venn taxonomy is an instance of a class that inherits from `VTXBase`, the same design like `NCSBase` and `NCSBase-Regressor` (see Chapter 2). Once the MCP implementation from the previous chapter moves from 1-taxonomies K_1 to n-taxonomies K_n, Venn taxonomies and n-taxonomies will be combined—since they are equal—and `libconform` will provide a new API for both together.

6. Meta-conformal predictors

For understanding how meta-conformal predictors achieve reliability we have to introduce the concept of abstention and abstaining classifiers. An abstaining classifier uses a measure of uncertainty and if the uncertainty of a new observation is too high the classifier does not give a prediction, it rejects it (see Vanderlooy et al., 2009).

CP is easily modified to an abstaining classifier. We could simply predict the label with the highest p-value and if its significance level (the second highest p-value, see Chapter 2) is above a certain threshold, the classifier returns the label, otherwise it rejects the observation.

		True class	
		positive	negative
Predicted class	positive	True Positive (TP)	False Positive (FP)
	negative	False Negative (FN)	True Negative (TN)
	rejected	Rejected Positive (RP)	Rejected Negative (RN)

Figure 1: Confusion matrix for binary abstention classifiers.

Meta-conformal predictors are described in Smirnov et al. (2009). The authors argue, that if there exists a classifier B we would need to construct a nonconformity measure based on B; not an easy task, since there exists no approach for doing so in general.

Therefore they introduce the method of using a conformal predictor with an already established nonconformity measure M as a meta-classifier in combination with B, so we can add a certainty measure to a prediction otherwise without any likelihood indicator. *B:M* is called the combined classifier.

The meta-classifier is a binary classifier that predicts new observations based on the meta data of the base classifier B. The labels of the meta data are either 0—the negative meta class—if B's prediction for an observation x_i was wrong or 1—the positive meta class—if it was correct.

Our meta-conformal predictor M should use a certainty measure in order to decide for a new observation, if the prediction of the base classifier B is trustworthy enough to return. Since M is trained on our meta data with the positive and negative meta class, M generates two p-values—one for each class—p_p, the positive p-value and p_n, the negative p-value. We can use both p-values to convert *B:M* to a scoring classifier with a score ratio $\frac{p_p}{p_n}$. Now we only need to define the reliability threshold T. If the score ratio generated by M for a new observation is greater than T we say the prediction of B is trustworthy and is returned, otherwise *B:M* abstains from making the prediction (see Smirnov et al., 2009).

Metrics for the performance of a binary classifier can be given by a confusion matrix (see Figure 1). For a test set the confusion matrix counts the predicted examples and maps them—depending on the true class and the predicted class—to its entries.

From the confusion matrix we can derive interesting metrics for the binary classifier, the most important being the accuracy A, the precision rate P:

$$A = \frac{TP + TN}{TP + TN + FP + FN}, \ P = \frac{TP}{TP + FP},$$

the true positive rate TPr and the false positive rate FPr:

$$TPr = \frac{TP}{TP + FN}, \; FPr = \frac{FP}{FP + TN}.$$

For a binary abstention classifier we can also add the rejection rate

$$R = \frac{RP + RN}{RP + RU + TP + TN + FP + FN}.$$

In order to construct a combined classifier $B{:}M$ we first need the meta data. For that we use the k-fold method normally used as a cross-validation technique (see Hastie et al., 2009, Chapter 7.10).

Let $\lbag z_1, \ldots, z_n \rbag, \; z_i \in \mathbf{Z} : \mathbf{X} \times \mathbf{Y}$ be our training set. We split the training set in k roughly equal sized partitions. For each partition: take the partition as test set, combine the others to a training set and fit B to it. Let B predict on the test set which then generates a partition of the meta data $\lbag z'_i, \ldots, z'_{i+l} \rbag, \; z'_i \in \mathbf{Z}' : \mathbf{X} \times \{0, 1\}$ (see Algorithm 2).

After we are through with Algorithm 2 and we have generated our meta data $\lbag z'_1, \ldots, z'_n \rbag$, we potentially could train M to it, but we still have to define the reliability threshold T.

Having a target accuracy A_t Smirnov et al. (2009) proposes to use a ROC isometrics approach defined in Vanderlooy et al. (2009) to set T based on A_t. In the case where we have a combined classifier $B{:}M$ A_t equals the precision rate P_M of M (see Smirnov et al., 2009).

Defining T based on P_M is done in five steps:

1. use the same k-fold algorithm used to generate the meta data—with M instead of B—to generate a set of scoring ratios $\frac{p_p}{p_n}$.

2. construct the ROC curve based on the TPr and FPr of M which indirectly maps to the scoring ratios. For more information about ROC curves see e.g. Fawcett (2006).

3. abstract the convex hull ROCCH of the ROC curve.

4. construct the iso-precision line from the equation $TPr = \frac{P_M}{1-P_M} \frac{N}{P} FPr$, N being the number of negative meta instances, P being the number of positive meta instances. This line represents all classifiers with a target precision rate of P_M. The classifier on the line which is abstaining the least is determined in the next step.

5. set T as the scoring ratio $\frac{p_p}{p_n}$ at the intersection of the ROCCH and the iso-precision line.

After we have determined T, B is trained on the whole training set $\lbag z_1, \ldots, z_n \rbag$, M on the whole meta data set $\lbag z'_1, \ldots, z'_n \rbag$ and the training operation of $B{:}M$ is over.

Algorithm 2 : k-fold method for meta data generation

```
Input:
  $B$: a classifier,
  bag: a bag of examples $\Lbag z_1, \dots, z_n \Rbag$,
  $k$: the amount of partitions
Output:
  meta-data: a bag of examples $\Lbag z'_1, \dots, z'_n \Rbag$
split bag into $k$ roughly equal sized partitions $\text{bag}_1, \dots, \text{bag}_k$
for all $\text{bag}_i, i = 1, \dots, k$ do
    combine all bags $\neq \text{bag}_i$ to the training set
    train $B$ with the training set
    let $B$ predict examples in $\text{Bag}_i$
    for all $(x_j, y_j) \in \text{bag}_i$ do
        add element to meta data: $\left( x_j, y'_j := \begin{cases} 0 & \text{if prediction of } B \text{ for } x_j \neq y_j \\ 1 & \text{if prediction of } B \text{ for } x_j = y_j \end{cases} \right)$
    end for
end for
return meta data
```

Notes on the implementation: `libconform` provides the `Meta` class for combined classifiers. In order to offer the most flexibility for developers `Meta` only takes two interfaces to each classifier B and M, one being a function used for training the classifier, the other being for predicting. This way `Meta` does not take the classifiers itself, so it can be used with any other library implementing B. The interfaces to M could change in future versions, so `Meta` uses M directly. Currently this is not the case since using conformal prediction and inductive conformal prediction takes different approaches.

The ROCCH is constructed using the scipy library's implementation (based on the qhull library) of the Quickhull algorithm (see Jones et al., 2001–2019; Barber et al., 1996).

7. Conclusion

Appendices

A. API reference

B. Examples

References

Vineeth Balasubramanian, Shen-Shyang Ho, and Vladimir Vovk. *Conformal Prediction for Reliable Machine Learning: Theory, Adaptations and Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, first edition, 2014. ISBN 0123985374, 9780123985378.

C. Bradford Barber, David P. Dobkin, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996. ISSN 0098-3500. doi: 10.1145/235815.235821. URL `http://doi.acm.org/10.1145/235815.235821`.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013.

François Chollet et al. Keras. `https://keras.io`, 2015.

Tom Fawcett. An introduction to roc analysis. *Pattern Recogn. Lett.*, 27(8):861–874, June 2006. ISSN 0167-8655. doi: 10.1016/j.patrec.2005.10.010. URL `http://dx.doi.org/10.1016/j.patrec.2005.10.010`.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer, second edition, 2009.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–2019. URL `http://www.scipy.org/`. [Online; accessed 14.06.2019].

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.4.541. URL `http://dx.doi.org/10.1162/neco.1989.1.4.541`.

Ilia Nouretdinov, Tom Melluish, and Vladimir Vovk. Ridge regression confidence machine. In *In Proceedings of the Eighteenth International Conference on Machine Learning*, pages 385–392. Morgan Kaufmann, 2001.

Harris Papadopoulos, Vladimir Vovk, and Alex Gammerman. Conformal prediction with neural networks. volume 2, pages 388 – 395, 11 2007. ISBN 978-0-7695-3015-4. doi: 10.1109/ICTAI.2007.47.

Craig Saunders, Alex Gammerman, and Vladimir Vovk. Transduction with confidence and credibility. 1999.

Glenn Shafer and Vladimir Vovk. A tutorial on conformal prediction. *Journal of Machine Learning Research*, 9:371–421, jun 2008. ISSN 1532-4435.

Evgueni Smirnov, Georgi Nalbantovi, and A. M. Kaptein. Meta-conformity approach to reliable classification. *Intelligent Data Analysis*, 13, 01 2009. doi: 10.3233/IDA-2009-0400.

Stijn Vanderlooy, Ida G. Sprinkhuizen-Kuyper, Evgueni N. Smirnov, and H. Jaap van den Herik. The roc isometrics approach to construct reliable classifiers. *Intell. Data Anal.*, 13(1):3–37, January 2009. ISSN 1088-467X. URL `http://dl.acm.org/citation.cfm?id=1551758.1551760`.

Vladimir Vovk, Alex Gammerman, and Craig Saunders. Machine-learning applications of algorithmic randomness. 1999.

Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*. Springer, 2005.