Kernel-Based Ensemble Learning in Python

Benjamin Guedj, Bhargav Srinivasa Desikan

information

MDPI

Article

Kernel-Based Ensemble Learning in Python

Benjamin Guedj [1,*,‡], **Bhargav Srinivasa Desikan** [2,‡]

1 Inria and University College London, London WC1E 6BT, UK
2 University of Chicago, Chicago 60637, IL, USA
* Correspondence: benjamin.guedj@inria.fr
‡ Both authors contributed equally to this work.

Received: 17 December 2019; Accepted: 22 January 2020; Published: 25 January 2020

Abstract: We propose a new supervised learning algorithm for classification and regression problems where two or more preliminary predictors are available. We introduce `KernelCobra`, a non-linear learning strategy for combining an arbitrary number of initial predictors. `KernelCobra` builds on the COBRA algorithm introduced by Biau *et al.* [1], which combined estimators based on a notion of proximity of predictions on the training data. While the COBRA algorithm used a binary threshold to declare which training data were close and to be used, we generalise this idea by using a kernel to better encapsulate the proximity information. Such a smoothing kernel provides more representative weights to each of the training points which are used to build the aggregate and final predictor, and `KernelCobra` systematically outperforms the COBRA algorithm. While COBRA is intended for regression, `KernelCobra` deals with classification and regression. `KernelCobra` is included as part of the open source Python package `Pycobra` (0.2.4 and onward), introduced by Guedj and Srinivasa Desikan [2]. Numerical experiments were undertaken to assess the performance (in terms of pure prediction and computational complexity) of `KernelCobra` on real-life and synthetic datasets.

Keywords: machine learning; python; ensemble learning; kernels; open source software

1. Introduction

In the fields of machine learning and statistical learning, ensemble methods consist of combining several estimators (or predictors) to create a new, superior estimator. Ensemble methods (also known as aggregation in the statistical literature) have attracted tremendous interest in recent years, and for a few problems, are considered state-of-the-art techniques, as discussed by Bell and Koren [3]. There is a wide variety of ensemble algorithms (some of which are discussed in Dietterich [4], Giraud [5] and Shalev-Shwartz and Ben-David [6]), with a crushing majority devoted to linear or convex combinations.

In this paper we propose a non-linear way of combining estimators, adding to a streamline of works pioneered by Mojirsheibani [7]. Our method (`KernelCobra`) extends the COBRA (standing for combined regression alternative) algorithm introduced by Biau *et al.* [1]. The COBRA algorithm is motivated by the idea that non-linear, data-dependent techniques can provide flexibility not offered by existing (linear) ensemble methods. By using information of proximity between the training data and predictions on test data, training points are collected to perform the aggregate. The COBRA algorithm selects training points by checking whether the proximity is less than a data dependant threshold ϵ, resulting in a binary decision (either keep the point or discard it). The `KernelCobra` algorithm introduced in the present paper aims to smoothen this data point selection process by introducing a kernel-based method to assigning weights to various points in the collective. The only weights that points could take in the COBRA algorithm were a discrete set between 0 and 1, whereas our smoothed scheme will span real values between 0 and 1. A python implementation of `KernelCobra` is provided in

the python package Pycobra, introduced and described by Guedj and Srinivasa Desikan [2]. We found in numerical experiments that `KernelCobra` consistently outperforms the original COBRA algorithm in a variety of situations.

The paper is organised as follows. Section 2 discusses related work and Section 3 introduces the ideas leading to `KernelCobra`. Section 4 presents the actual implementations of `KernelCobra` in the `pycobra` Python library. Section 5 illustrates the performance (both in prediction accuracy and computational complexity) on real-life and synthetic datasets, along with comparable aggregation techniques. Section 6 presents avenues for future work.

2. Related Work

Our algorithm is inspired by the work of Biau *et al.* [1] which introduced the COBRA algorithm. COBRA itself is inspired by the seminal work by Mojirsheibani [7], where the idea of using consensus between machines to create an aggregate was first discussed. Our algorithm `KernelCobra` is a strict generalisation of COBRA.

In a work parallel to ours, the idea of using the distance between points in the output space is also explored by Fischer and Mougeot [8], where weights are assigned to points based on proximity of the prediction in the output space and the training data. However, the method employed (which will now be referred to as `MixCobra`) also uses the input data while constructing the aggregate. While it is true that more data-dependant information might improve the quality of the aggregate, we argue that in cases with high-dimensional input data, proximity between points will not add much useful information. Computing distance metrics in high dimensions is a computational challenge, which in our view, could undermine the statistical performance (see [9] for discussion). While using both input and output information might provide satisfactory results in lower dimensions, non-linear ensemble learning algorithms arguably perform particularly well in high dimensions, as they are not affected by the dimension of the input space. This edge is lost in the `MixCobra` method.

`KernelCobra` overcomes this problem by only considering proximity of data points in the prediction space, allowing to perform faster calculations. This makes `KernelCobra` a promising candidate for high dimensional learning problems: as a matter of fact, `KernelCobra` is not affected at all by the curse of dimensionality, with the complexity only increasing with the number of preliminary estimators.

In a recent work, the original COBRA algorithm (as implemented by the `pycobra` Python library; see Guedj and Srinivasa Desikan [2]) was successfully adapted by Guedj and Rengot [10] to perform image denoising. The authors report that the COBRA-based denoising algorithm significantly outperforms most state-of-the-art denoising algorithms on a benchmark dataset, calling for the broadcasting of non-linear ensemble methods in computer vision and image processing communities.

3. KernelCobra: A Kernelized Version of COBRA

Throughout this section, assume that we are given a training sample $D_n = (\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)$ of i.i.d. copies of $(\mathbf{X}, Y) \in \mathbb{R}^d \times \mathbb{R}$ (with the notation $\mathbf{X} = (X_1, \ldots, X_d)$). Assume that $\mathbb{E}Y^2 < \infty$. The space $\mathbb{R}^d$ is equipped with the standard euclidean metric. Our goal is to consistently estimate the regression function $r^\star(\mathbf{x}) = \mathbb{E}[Y|\mathbf{X} = \mathbf{x}]$, for some new query point $\mathbf{x} \in R^d$, using the data D_n.

To begin with, the original data set D_n is split into two data sequences $D_k = (\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_k, Y_k)$ and $D_\ell = (\mathbf{X}_{k+1}, Y_{k+1}), \ldots, (\mathbf{X}_n, Y_n)$, with $\ell = n - k \geq 1$. For ease of notation, the elements of D_ℓ are renamed $(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_l, Y_l)$, similar to the notation used by Biau *et al.* [1].

Now, suppose that we are given a collection of $M \geq 1$ competing estimators (referred to as machines from now on) $r_{k,1}, \ldots, r_{k,M}$ to estimate $r^\star$. These preliminary machines are assumed to be generated using only the first sub-sample D_k. In all practical scenarios, machines can be any machine learning algorithm, from classical linear regression all the way up to a deep neural network, including naive Bayes, decision trees, penalised regression, random forest, k-nearest neighbours and so on. These machines have no restrictions in their nature: they can be parametric or nonparametric. The only

condition is that each of these machines $m = 1, \ldots, M$ is able to provide an estimation $r_{k,m}(\mathbf{x})$ of $r^\star(\mathbf{x})$ on the basis of D_k alone. Let us stress here that the number of machines M is fixed.

Let us recall the COBRA algorithm, introduced by Biau *et al.* [1]: for any query point $\mathbf{x} \in \mathbb{R}^d$, the aggregated estimator COBRA is given by

$$T_n(\mathbf{r}_k(\mathbf{x})) = \sum_{i=1}^{\ell} W_{n,i}(\mathbf{x}) Y_i, \tag{1}$$

where the random weights are given by Biau *et al.* [1, Equation 2.1]

$$W_{n,i}(\mathbf{x}) = \frac{\mathbf{1}_{\cap_{m=1}^{M}\{|r_{k,m}(\mathbf{X}_i) - r_{k,m}(\mathbf{x})| \leq \epsilon\}}}{\sum_{j=1}^{\ell} \mathbf{1}_{\cap_{m=1}^{M}\{|r_{k,m}(\mathbf{X}_j) - r_{k,m}(\mathbf{x})| \leq \epsilon\}}}, \tag{2}$$

where $\mathbf{1}_A$ denotes the indicator function of a set A, given by

$$\mathbf{1}_A : \mathbf{x} \mapsto \begin{cases} 1 & \text{if } \mathbf{x} \in A, \\ 0 & \text{if not.} \end{cases}$$

This aggregation strategy operates in a nonlinear way with respect to the preliminary machines, by aggregating outputs only.

The (possibly data-dependent) threshold parameter $\epsilon > 0$ serves to discard points from the initial sample for which at least one prediction made by a machine is considered too far from the prediction made on the new query point: this then sets this point's weight to zero. This threshold parameter needs to be finely tuned (using cross-validation in the initial COBRA algorithm). One motivation for introducing `KernelCobra` is to attenuate this dependence on a problem-dependent threshold parameter.

In addition, the rather bumpy behaviour of the weights in (2) (they can only take finite values in the set $\{0, 1/\ell, 1/(\ell-1), \ldots, 1\}$) only allows for restricted expressivity. Another motivation for introducing `KernelCobra` is to allow for continuous values in $(0, 1)$, and hence, it increases flexibility. Last but not least, the more generic formulation of `KernelCobra` will yield better performance than COBRA's, as illustrated in Section 5.

As a gentle start, we now introduce a version of `KernelCobra` with the Euclidean distance d_ε and an exponential form of the weights—these will be eventually generalised.

Given the collection of basic machines $\mathbf{r}_k = (r_{k,1}, \ldots, r_{k,M})$, we define the aggregated estimator for any $\mathbf{x} \in \mathbb{R}^d$ as in (1),

$$T_n(\mathbf{r}_k(\mathbf{x})) = \sum_{i=1}^{\ell} W_{n,i}(x) Y_i, \tag{3}$$

where the random weights $W_{n,i}(\mathbf{x})$ are now given by

$$W_{n,i}(\mathbf{x}) = \frac{\exp\left\{-\lambda \sum_{m=1}^{M} d_\varepsilon(r_{k,m}(\mathbf{X}_i), r_{k,m}(\mathbf{x}))\right\}}{\sum_{j=1}^{\ell} \exp\left\{-\lambda \sum_{m=1}^{M} d_\varepsilon(r_{k,m}(\mathbf{X}_j), r_{k,m}(\mathbf{x}))\right\}}. \tag{4}$$

The hyperparameter $\lambda > 0$ acts as a temperature parameter, to adjust the level of fit to data, and will be optimised in numerical experiments using cross-validation. Let us stress here that $d_\varepsilon(a, b)$ denotes the Euclidean distance between any two points $a, b \in \mathbb{R}$. In (4), this serves as a way to measure the proximity or coherence between predictions on training data and predictions made for the new query point, across all machines.

This form (4) is more smooth than the form introduced in the COBRA algorithm [1] and is reminiscent of exponential weights. The aggregated estimator in (1) with weights defined in (4) is called `KernelCobra`.

A more generic form is given by

$$W_{n,i}(\mathbf{x}) = \frac{\sum_{m=1}^{M} K(r_{k,m}(\mathbf{X}_i), r_{k,m}(\mathbf{x}))}{\sum_{j=1}^{\ell}\sum_{m=1}^{M} K(r_{k,m}(\mathbf{X}_j), r_{k,m}(\mathbf{x}))}, \tag{5}$$

where K denotes a kernel used to capture the proximity between predictions on training and query data, across machines. The aggregated estimator in (1) with weights defined in (5) is called general `KernelCobra`.

Rather than a threshold to keep or discard data point i in the weights (as with the initial COBRA algorithms with weights defined in Equation (2)), its influence is now always considered, by a measure of how preliminary machines predict outcomes for the new query point which are close to the predictions made for point i. In other words, a data point i will have more influence on the aggregated estimator (its weight will be higher) if machines predict similar outcomes for i and the new query point. Let us stress again here that `KernelCobra`, as the initial COBRA algorithm, aggregates machines in a non-linear way: the aggregated estimator in (1) is a weighted combination of *observed outputs* Y_is, *not* of initial machines (which serve to build the weights). As such, it is fairly different from most aggregation schemes which form linear combinations of machines' outcomes.

Note also that computing the weights defined in (4) and (5) involves elementary computations over *scalars* (each machine's prediction over the training sample and the new query point) rather than d-dimensional vectors. As highlighted above, both versions of `KernelCobra` avoid the curse of dimensionality.

General `KernelCobra` allows for the use of any kernel which might be preferred by practitioners—it is the generic version of our algorithm. In practice, we have found that the `KernelCobra` defined with weights in (4) provides interesting empirical results, and is more interpretable. We thus provide both versions, as they express a trade-off between generality and ease of interpretation and use.

The remainder of this section is devoted to two interesting byproducts of our approach, to the unsupervised setting and for classification.

3.1. The Unsupervised Setting

As COBRA and `KernelCobra` are non-linear aggregation methods, the final estimator is a weighted combination of observed outputs Y_is. We can turn our approach to a more classical linear aggregation scheme, to the notable point that none of the approaches depend on Y_is, allowing us to consider the unsupervised setting. This differs from classical linear or convex aggregation methods such as exponential weights: the weights depend on a measure of performance such as an empirical risk, which will involve Y_is.

We can now throw away all Y_js for $j = 1, \ldots, \ell$, and we propose the following estimator for any new query point $\mathbf{x} \in \mathbb{R}^d$:

$$T_n(\mathbf{r}_k(\mathbf{x})) = \sum_{i=1}^{\ell} W_{n,i}(\mathbf{x}) \sum_{m=1}^{M} r_{k,m}(\mathbf{X}_i)\widetilde{W}_{n,m}. \tag{6}$$

Our first set of weights $(W_{n,i}(\mathbf{x}))_{i=1}^{\ell}$ is given by (4) or (5), and serves to weigh data points. Our second set of weights $(\widetilde{W}_{n,m})_{m=1}^{M}$ used to aggregate the predictions of each machine, can be any sequence of weights summing up to 1, and serves to weigh machines.

In other words, once the machines have been trained (either in a supervised setting using the outputs in subsample D_k, or in an unsupervised setting by discarding all outputs across the dataset D), the estimator defined in (6) no longer needs outputs from the second half of the dataset D_ℓ, thereby extending to semi-supervised and unsupervised settings, further illustrating the flexibility of our approach.

3.2. Classification

Non-linear aggregation of classifiers has been studied by Mojirsheibani [7] and Mojirsheibani [11] (where a kernel is also used to smoothen the point selection process). The papers Mojirsheibani [12] and Balakrishnan and Mojirsheibani [13] focus on using the misclassification error to build the aggregate. Here we provide a simple extension of our approach to classification.

For binary classification ($\mathcal{Y} = \{0, 1\}$), the combined classifier is given by

$$C_n(\mathbf{x}) = \begin{cases} 1, & \text{if} \quad \sum_{i=1}^{\ell} Y_i W_{n,i}(\mathbf{x}) \geq \frac{1}{2}, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

The weights can be chosen as (4) or (5).

We also provide a combined classifier for the multi-class setting: let us assume that $\mathcal{Y}$ is a finite discrete set of classes,

$$C_n(\mathbf{x}) = \arg\max_{k \in \mathcal{Y}} \sum_{i=1}^{\ell} \mathbf{1}_{\{Y_i = k\}} W_{n,i}(\mathbf{x}). \tag{8}$$

To conclude this section, let us mention that Biau *et al.* [1, Theorem 2.1] proved that the combined estimator with weights chosen as in the initial COBRA algorithm (2) enjoys an oracle guarantee: the average quadratic loss of the estimator is upper bounded by the best (lowest) quadratic loss of the machines up to a remainder term of magnitude $\mathcal{O}(\ell^{-\frac{2}{M+2}})$. This result is remarkable, as it does not involve the ambient dimension d but rather the (fixed) number of machines M. We focus in the present paper on the introduction of `KernelCobra` and its variants, and its implementation in Python (detailed in the next section). We leave for a future work the extension of Biau *et al.* [1]'s theoretical results.

4. Implementation

All new algorithms described in the present paper are implemented in the Python library `pycobra` (from version 0.2.4 and onward); we refer to Guedj and Srinivasa Desikan [2] for more details.

The python library `pycobra` can be installed via `pip` using the command `pip install pycobra`. The PyPi page for `pycobra` is https://pypi.org/project/pycobra/. The code for `pycobra` is open source and can be found on GitHub at https://github.com/bhargavvader/pycobra. The documentation for `pycobra` is hosted at https://modal.lille.inria.fr/pycobra/.

We describe the general `KernelCobra` algorithm in Algorithm 1.

Algorithm 1: General `KernelCobra`

```
Data: input vector X, Kernel, [Kernel Parameters], basic-machines,
      training-set-responses, training-set
# training-set is the set composed of all data_point and the responses.
# training-set-responses is the set composed of the responses.
Result: prediction Y
weights = [] ;
# weights is a list of size ℓ with each index mapping to information of proximity of a data point;
for machine j in basic-machines do
    pred = basic-machines[j](X)
    # where basic-machines[j](X) denotes the prediction made by machine j at point x;
    for index,vector in training-set-responses do
        weights[index] += Kernel(pred, basic-machines[j](vector)) ;
    end
end
weights = weights / sum(weights) ;
result = training-set-responses * weights ;
```

`KernelCobra` is implemented as part of the `KernelCobra` class in the `pycobra` package. The estimator is `scikit-learn` compatible (see Pedregosa *et al.* [14]), and works similarly to the other estimators provided in the `pycobra` package. The only hyperparameter accepted in creating the object is a `random state` object.

The `pred` method implements the algorithm described in Algorithm 1, and the `predict` method serves as a wrapper for the `pred` method to ensure it is `scikit-learn` compatible. It should be noted that the `predict` method can be customised to pass any user-defined kernel (along with parameters), as suggested by (5). The default behaviour of the `predict` method is set to use the weights defined in (4).

Similarly to the other estimators provided in `pycobra`, `KernelCobra` can be used with the `Diagnostics` and `Visualisation` classes, which are used for debugging and visualising the model. Since it abides the `scikit-learn` ecosystem, one can use either `GridSearchCV` or the `Diagnostics` class to tune the parameters for `KernelCobra` (such as the temperature parameter).

The default regression machines used for `KernelCobra` are the `scikit-learn` implementations of lasso, random forest, decision trees and ridge regression. This is merely an editorial choice to have the algorithm up and ready immediately, but let us stress here that one can provide *any* estimator using the `load_machine` method, with the only constraints being that it must be trained on D_k, and that it has a valid `predict` method.

We also provide the pseudo-code for the variant of `KernelCobra` in semi-supervised or unsupervised settings defined by (6) (Algorithm 2), along with the variant for multi-class classification defined by (8) (Algorithm 3).

Algorithm 2: `KernelCobra` in the unsupervised setting

Data: input vector **X**, Kernel, `[Kernel Parameters]`, `basic-machines`, `training-set-responses`, `training-set`
`training-set` is the set composed of all `data_point` and the responses.
`training-set-responses` is the set composed of the responses.
Result: prediction **Y**
`weights-points` = [] ;
# *`weights-points` is a list of size l with each index mapping to information of proximity of a data point;*
for *`machine` j in `basic-machines`* **do**
  `pred` = `basic-machines`[j](**X**)
  # *where* `basic-machines`[j](**X**) *denotes the prediction made by machine j at point* **X**;
  for *`index,vector` in `training-set-responses`* **do**
    weights-points[index] += Kernel(pred, `basic-machines[j](vector)`) ;
  end
end
`machine-predictions` is a list mapping each machine and it's prediction of x_i ;
`weights-machines` is a list mapping each machine and it's weight which must sum to 1 ;
weights-points = weights-points / sum(weights-points) ;
machine-predictions = weights-machines * machine-predictions ;
results = machine-predictions * weights-points ;

Algorithm 3: `KernelCobra` for classification

Data: input vector **X**, `basic-machines`, `training-set-responses`, `training-set`
`machine-predictions`
`training-set` is the set composed of all `data_point` and the responses.
`training-set-responses` is the set composed of the responses.
`machine-predictions` is the dictionary mapping the constituent machines and their predictions on the training-set
Result: prediction **Y**
`machine-set` = [] ;
# *`machine-set` is a dictionary which stores the label predicted for each point in the training set;*
for *`machine` j in `basic-machines`* **do**
  `pred` = basic-machines[j](**X**)
  # *where `basic-machines[j]`(**X**) denotes the prediction made by machine j at point* **X**;
  for *`index` in `training-set-responses`* **do**
    if *machine-predictions[machine][index] == pred* **then**
      add index to `machine-set`[`machine`]
    end
  end
end
return the majority vote on `machine-set`, as defined by (8) ;

To conclude this section, let us mention that the complexity of all presented algorithms is $\mathcal{O}(M\ell)$ as we loop over all data points in the subsample D_ℓ and over all machines.

5. Numerical Experiments

We have conducted numerical experiments to assess the merits of `KernelCobra` in terms of statistical performance, and computational cost. We compare pythonic implementations of `KernelCobra`, `MixCobra`, the original COBRA algorithm as implemented by `pycobra` and the default `scikit-learn` machines used to create our aggregate.

We test our method on four synthetic data-sets and two real world data-sets, and report statistical accuracy and CPU-timing. The synthetic datasets are generated using `scikit-learn`'s `make-regression`, `make-friedman1` and `make-sparse-uncorrelated` functions. The two real world datasets are the Boston Housing dataset, and the Diabetes dataset.

Table 1 wraps up our results for statistical accuracy and establishes `KernelCobra` as a promising new kernel-based ensemble learning algorithm. `KernelCobra` achieves the smallest mean error on four datasets (Gaussian, Sparse, Boston, Friedman) out of six, with the ridge algorithm and MixCobra winning in the other two. Out of these six classical datasets, `KernelCobra` systematically outperforms the initial COBRA algorithm.

	Gaussian	Sparse	Diabetes	Boston	Linear	Friedman
random-forest	12,266.640297 (1386.2011)	3.35474 (0.3062)	2924.12121 (415.4779)	18.47003 (4.0244)	0.116743 (0.0142)	5.862687 (0.706)
ridge	491.466644 (201.110142)	1.23882 (0.0311)	**2058.08145** (127.6948)	13.907375 (2.2957)	0.165907 (0.0101)	6.631595 (0.2399)
svm	1699.722724 (441.8619)	1.129673 (0.0421)	8984.301249 (236.8372)	74.682848 (114.9571)	0.178525 (0.0155)	7.099232 (0.3586)
tree	22,324.209936 (3309.8819)	6.304297 (0.9771)	5795.58075 (1251.3533)	32.505575 (14.2624)	0.185554 (0.0246)	11.136161 (1.73)
Cobra	1606.830549 (651.2418)	1.951787 (0.5274)	2506.113231 (440.1539)	16.590891 (8.0838)	0.12352 (0.0109)	5.681025 (1.3613)
KernelCobra	**488.141132** (189.9921)	**1.11758** (0.1324)	2238.88967 (1046.0271)	**12.789762** (9.3802)	0.113702 (0.0089)	**4.844789** (0.5911)
MixCobra	683.645028 (196.7856)	1.419663 (0.1292)	2762.95792 (512.6755)	16.228564 (12.7125)	**0.104243** (0.0104)	5.068543 (0.6058)

Table 1. For each estimator and each dataset (names point to datasets URLs), we report the mean root mean square error (RMSE—along with standard deviation) over 100 independent runs. Bold numbers indicate the best method for each dataset.

Figure 1 compares the computational costs of the original COBRA, `MixCobra` and `KernelCobra`. As both COBRA and `KernelCobra` do not use the input data, they do not suffer from an increase of data dimensionality and significantly outperform `MixCobra`. A salient fact is that the computational complexity of `KernelCobra` does not increase with the number of features.

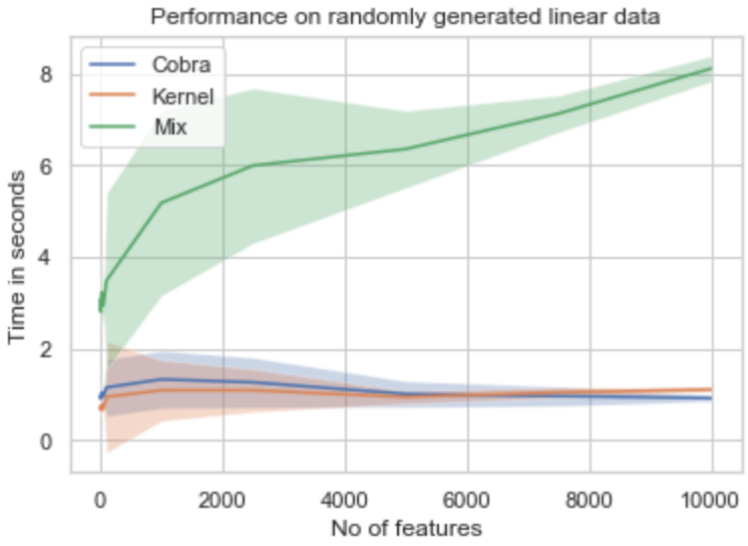
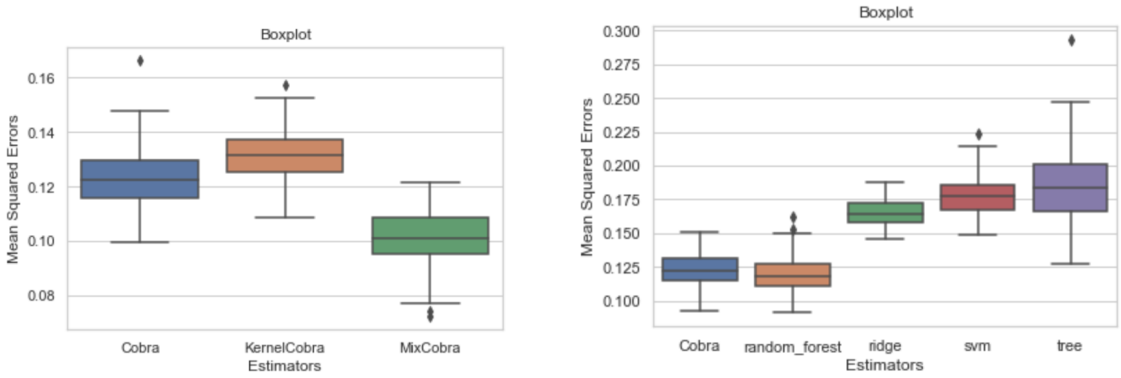


Figure 1. CPU timing for building the initial COBRA, `MixCobra` and `KernelCobra` estimators. Each line is an average over 100 independent runs, and shaded areas show standard deviations.

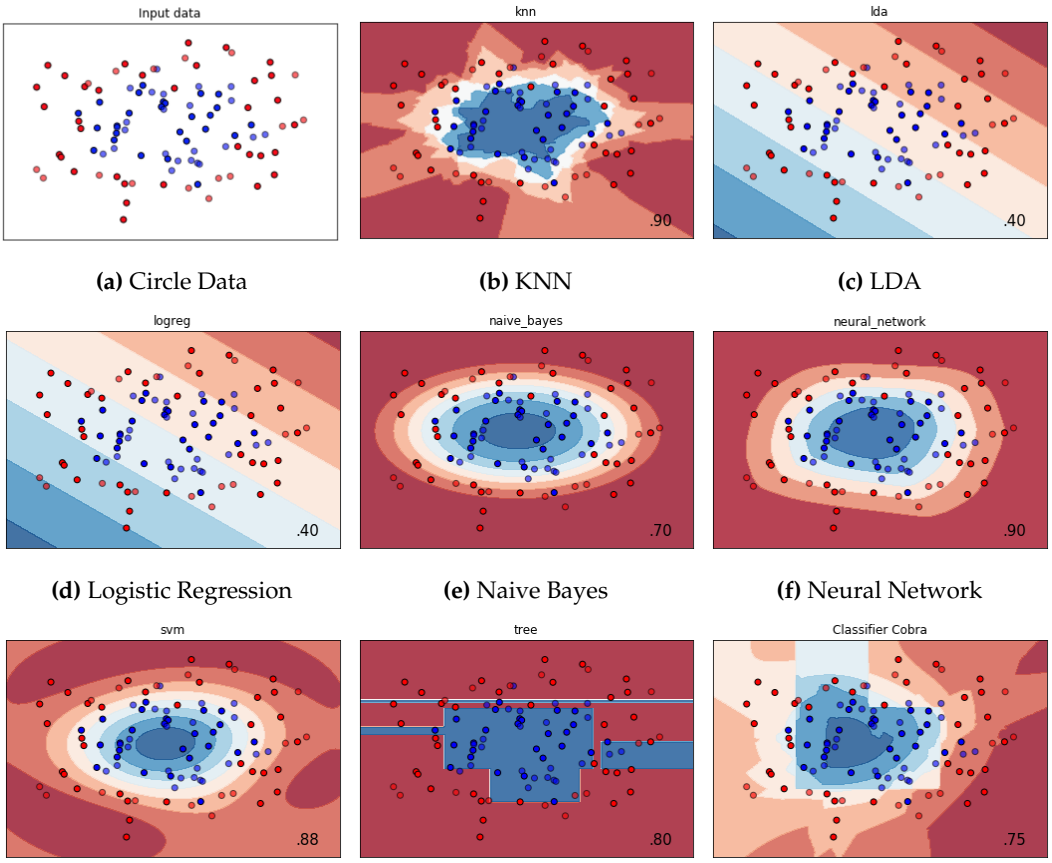
The `pycobra` package also offers a visualisation suite which gives QQ-plots, boxplots of errors and comparisons between the predictions of machines and the aggregate, along with the true values. We report a sample of those outputs in Figure 2.



(a) COBRA, `MixCobra`, `KernelCobra`. **(b)** COBRA vs. basic machines.

Figure 2. Boxplot of errors over 100 independent runs.

Last but not least, we provide a sample of decision boundaries for the classification variant of `KernelCobra` on three datasets, in Figure 3, Figure 4 and Figure 5. These three datasets are `scikit-learn` generic datasets for classification—`linearly-separable`, `make-moons` and `make-circles`. The nature of these datasets provides us a way to visualise how `ClassifierCobra` classifies with regard to the default classifiers used.

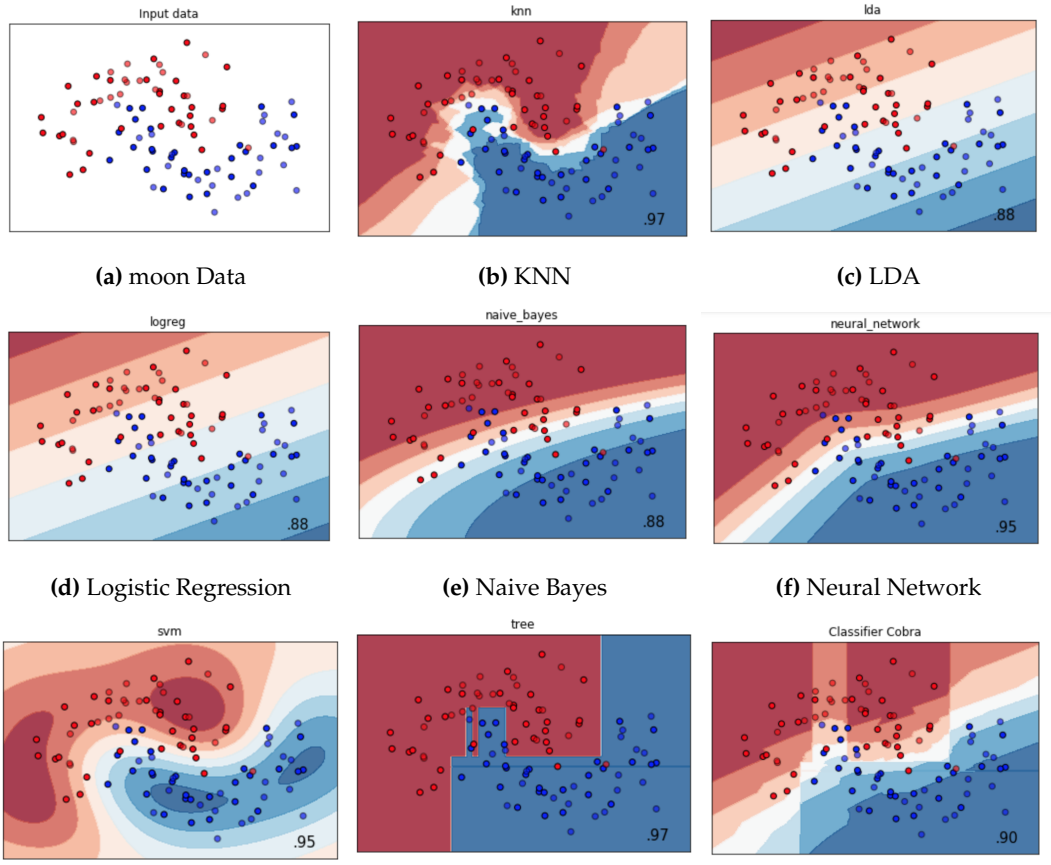


(a) Circle Data **(b)** KNN **(c)** LDA

(d) Logistic Regression **(e)** Naive Bayes **(f)** Neural Network

(g) Support Vector Machine **(h)** Decision Tree **(i)** `KernelCobra`

Figure 3. Decision boundaries of base classifiers and `KernelCobra` on the circle dataset.

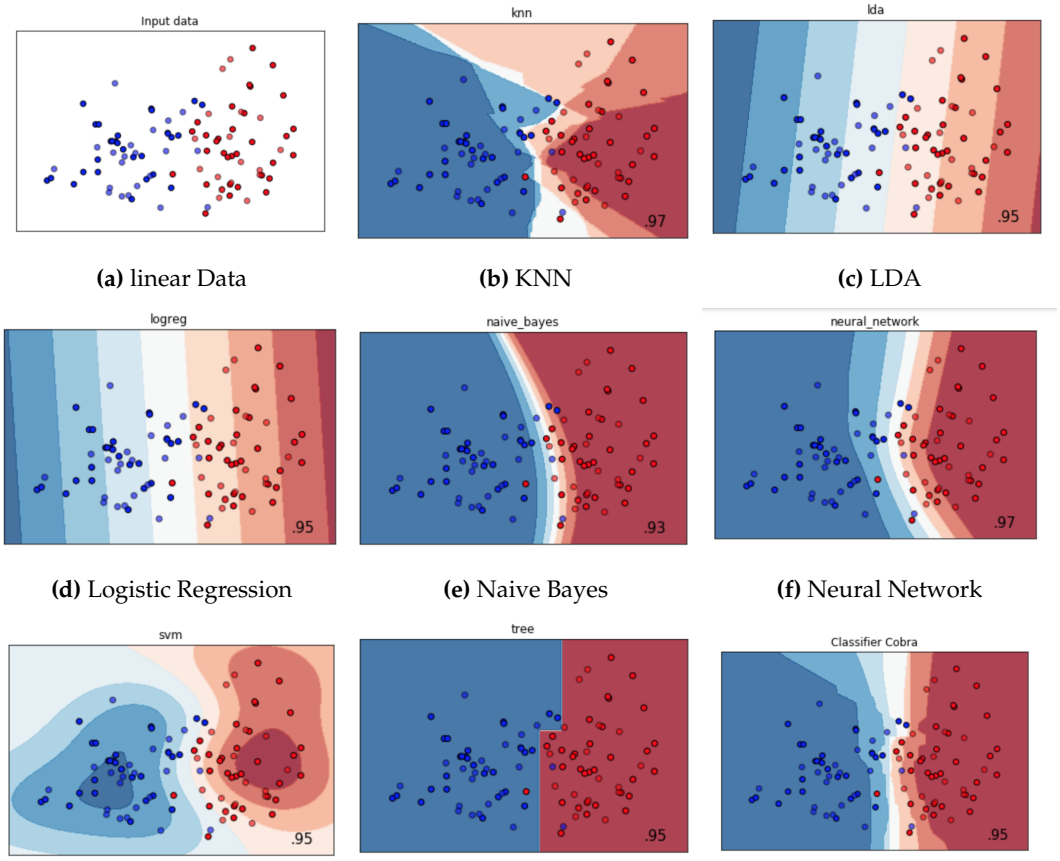


(a) moon Data **(b)** KNN **(c)** LDA

(d) Logistic Regression **(e)** Naive Bayes **(f)** Neural Network

(g) Support Vector Machine **(h)** Decision Tree **(i)** `KernelCobra`

Figure 4. Decision boundaries of base classifiers and `KernelCobra` on the moon dataset.



(a) linear Data **(b)** KNN **(c)** LDA

(d) Logistic Regression **(e)** Naive Bayes **(f)** Neural Network

(g) Support Vector Machine **(h)** Decision Tree **(i)** `KernelCobra`

Figure 5. Decision boundaries of base classifiers and `KernelCobra` on the linear dataset.

Some notes about the nature of the experiments and the performance: `KernelCobra` is the best performing machine for four out of six datasets. These values were achieved using an optimally derived bandwidth parameter for that dataset. This was calculated using the `optimal-kernelbandwidth` function in the `Diagnostics` class of the `pycobra` package. The default bandwidth values do not perform as well, and if we further fine-tune the bandwidth value, we would get potentially better results. `MixCobra` has similar tunable parameters which affect its performance, but takes significantly longer, as there are three parameters to tune. We used the default range of parameters to test before choosing optimal parameters for both `KernelCobra` and `MixCobra` in the results displayed.

When considering both the CPU timing to find optimal parameters and the statistical performance, `KernelCobra` outperforms the initial COBRA algorithm.

6. Conclusion and Future Work

The COBRA algorithm from Biau *et al.* [1] established a nonlinear aggregation method with theoretical guarantees and competitive numerical performance. In the present paper, we introduced a generalisation of the COBRA algorithm—called `KernelCobra`—to make it more flexible and user-friendly, and increase its performance on a wider range of problems (classification and regression—supervised, semi-supervised or unsupervised). `KernelCobra` delivers a kernel-based ensemble learning algorithm which is versatile, computationally cheap and flexible. All variants of `KernelCobra` ship as part of the `pycobra` Python library introduced by Guedj and Srinivasa Desikan [2] (from version 0.2.4), and are designed to be used in a `scikit-learn` environment. We will conduct, in future work, a theoretical analysis of the kernelised COBRA algorithm to complete the theory provided by Biau *et al.* [1].

Author Contributions: Both authors contributed equally to this work.

Funding: A substantial fraction of this work was carried out while both authors were affiliated to Inria, Lille—Nord Europe research centre, Modal project-team.

Acknowledgments: The authors are grateful to Sebastian Raschka for pointing out to the call for papers for this special issue of *Information*.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Biau, G.; Fischer, A.; Guedj, B.; Malley, J.D. COBRA: A combined regression strategy. *J. Multivar. Anal.* **2016**, *146*, 18–28.
2. Guedj, B.; Srinivasa Desikan, B. Pycobra: A Python Toolbox for Ensemble Learning and Visualisation. *J. Mach. Learn. Res.* **2018**, *18*, 1–5.
3. Bell, R.M.; Koren, Y. Lessons from the Netflix prize challenge. *ACM SIGKDD Explor. Newsl.* **2007**, *9*, 75–79.
4. Dietterich, T.G. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer: Berlin, Germany, 2000; pp. 1–15.
5. Giraud, C. *Introduction to high-dimensional statistics*. CRC Press: Boca Raton, FL, USA, 2014.
6. Shalev-Shwartz, S.; Ben-David, S. *Understanding machine learning: From theory to algorithms*. Cambridge university press: Cambridge, UK, 2014.
7. Mojirsheibani, M. Combining classifiers via discretization. *J. Am. Stat. Assoc.* **1999**, *94*, 600–609.
8. Fischer, A.; Mougeot, M. Aggregation using input–output trade-off. *J. Stat. Plan. Inference* **2019**, *200*, 1–19.
9. Steinbach, M.; Ertöz, L.; Kumar, V. The challenges of clustering high dimensional data. In *New directions in statistical physics*; Springer: Berlin, Germany, 2004; pp. 273–309.
10. Guedj, B.; Rengot, J. Non-linear aggregation of filters to improve image denoising. *arXiv* **2019**, arXiv:1904.00865.
11. Mojirsheibani, M. A kernel-based combined classification rule. *Stat. probab. lett.* **2000**, *48*, 411–419.
12. Mojirsheibani, M. An almost surely optimal combined classification rule. *J. multivar. anal.* **2002**, *81*, 28–46.
13. Balakrishnan, N.; Mojirsheibani, M. A simple method for combining estimates to improve the overall error rates in classification. *Comput. Stat.* **2015**, *30*, 1033–1049.
14. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, É. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.