An implementation and improvement of the ToPs predicting algorithm

Master Thesis

Author: Jordi Piqué Sellés
Supervisor: Ricard Gavaldà Mestre

Universitat Politècnica de Catalunya

Facultat d'informàtica de Barcelona

June 28, 2019

Abstract

In February 2018, J. Yoon, W. R. Zame, and M. van der Schaar presented a paper titled *"ToPs: Ensemble Learning with Trees of Predictors"* [1]. The paper describes a new approach to ensemble learning. They propose to build a tree, but with a predictor associated to each node. The tree implicitly segments the feature space by splitting the dataset recursively and this creates subsets from different regions of this space. Then, the predictors can specialize in learning only the characteristics of these subsets of instances. The overall prediction of the tree for an instance is obtained by aggregating the results of the predictors found along the unique path from the root to a leaf for this instance. The results of their experiments seemed to prove the superiority of ToPs versus other state of the art predictive algorithms.

However, they did not release their implementation of the method, and the paper leaves some aspects unclear.

In this work we present a public implementation of ToPs. This implementation is based on the theory explained in the original paper, and adds new functionality and improved features. Our implementation is finally tested using several datasets and proven not to be as good as other ensemble methods, in contrast to the conclusions from the original paper.

Contents

1. Introduction

Machine Learning (in short, ML) has become one of the hottest topics in the scientific community. It started appearing first in publications related to statistics and computer science research, but in the last years it has become crucial to many different disciplines such as biology or economy. The increase in popularity in cross areas has not been confined only to academic research. ML has become one key tool for solving real problems and many people, companies and institutions use it on a daily basis.

The most popular types of ML algorithms are for supervised learning. These algorithms take a set of observations, called instances, as input data. Each of these instances has a set of features, which describe it, and a target variable, or *label*. In the literature, this set of labeled instances is called the training set. The algorithm is responsible to automatically find a model, given a training set, that can predict the value of the target variable for new, unlabeled, instances. To do so, it tries to find patterns in the training data that correlate with the labels.

The research literature describes hundreds of different algorithms, or methods, for building models from data. The reason is that there is not a single method which is the best in all the areas. Each method has its own pros and cons and the data scientist who use them must know its characteristics. This has been theoretically formalized in the so-called "no-free-lunch theorem" [2]. However, in practice, there is a type of methods that seem to work better than most of the rest. These methods are called *ensemble methods*.

In the framework of ensemble learning we do not build and use a single predictor but a set of them. The idea is that a committee of predictors is more reliable than a single one. The intuition is that one predictor may give a wrong prediction, but the probability that a majority of predictors from the ensemble are wrong is much smaller. To design an ensemble method we have to make two major decisions. The first one is how we will create the set of base predictors. The second one is how we will combine them to make predictions. Depending on what we decide on these aspects we will have a different type of ensemble predictor. In section 2.4 we provide a general explanation about techniques for building ensembles, and in section 3.2 there are examples of widely used ensemble methods.

In this work we present an enhanced implementation of ToPs (Tree of Predictors) and ensemble learning method used for classification [1].

1.1. Brief description of ToPs

ToPs [1] is an ensemble of predictors. This means that it is formed by several base predictors and a method to aggregate their answers, that is, to convert their predictions into a single one). The idea of ensemble learning is not new, as explained in section 1 and developed deeper in section 3.2. Examples of ensemble methods are Random Forest, Adaboost or Stacking. So, if the idea of ensemble learning has been exploited in so many ML algorithms, what gives ToPs that other algorithms do not already have?

The main difference of ToPs with respect to other ensemble methods is its ability to fit different predictors for different regions of the feature space. It does so by growing a tree, like in a standard decision tree builder, splitting the dataset recursively using a variable and a threshold over that variable. Then, it assigns a predictor to each sub-region of the feature space. This is done by simply assigning a predictor to each node of the tree. This is exemplified in the following figure:

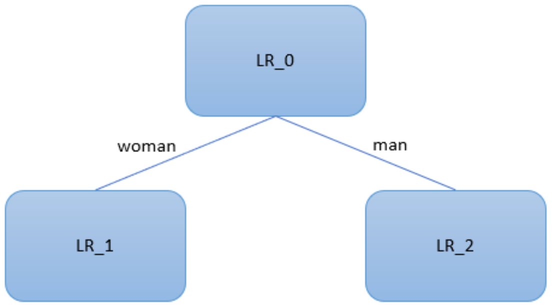
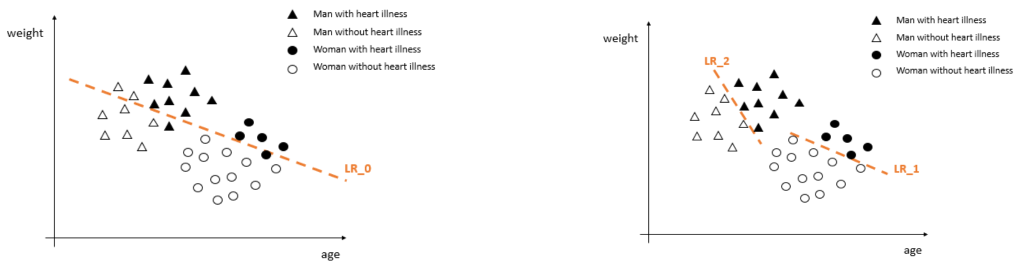


Figure 1: Each node contains an instance of Logistic Regression trained with the data corresponding to it. The parent node contains all the dataset, while the children contain only a sub-region of the feature space. The left one only contains instances corresponding to women while the right one contains instances corresponding to men.

Growing a tree implies each node will receive a rectangular sub-region of the feature space. Implicitly, using this method, we are segmenting the feature space and we are creating bounded regions. Assigning a predictor to each node implies that the predictor will only have to learn a specific region of the feature space, which may be easier and with less interactions than the full one. The consequence is that, with some simple predictors, by specializing them to learn only a small and easier sub-region of the feature space, we can build a very complex predictor.

Figure 2 illustrates the idea of building a complex predictor by assigning different regions of the feature space to some simple base predictors.



(a) (b)

Figure 2: Hand made example. (a) Separating line learned by the Logistic Regression instance associated to the root node in Figure 3 (b) Separating lines learned by the Logistic Regression instances associated to the leaves nodes in Figure 3. In this case, the target variable ”have heart illness” behaves differently depending on the sex. Notice each subtree can fit a better separating line thanks to dealing only with a sub-region of the feature space.

ToPs has another interesting feature, which is automatically selecting the best predictor from a set of base predictors for a region of the feature space. In ML, data-scientists normally spend time trying different predictors in a dataset and selecting the one that performs the best. With tops, this task is done automatically.

Once ToPs has grown atree and assigned a predictor to each node, we have to aggregate somehow these predictors. Here, we will see another difference of ToPs with respect to other ensemble methods. For example, Random Forest is a single ensemble of trees. Every tree is used when we try to classify a new instances because all the trees belong to the same and unique ensemble. In ToPs, when we aggregate the classifiers, we do not create a single ensemble where all models vote on every instance, but one ensemble for each path from the root to a leaf. In practice, this is the same as the number of leaves, because in a tree-like graph there is only a unique path from the root to a given leaf. As a consequence, every path from the root to a leaf defines an ensemble which is created by the predictors found along that path.

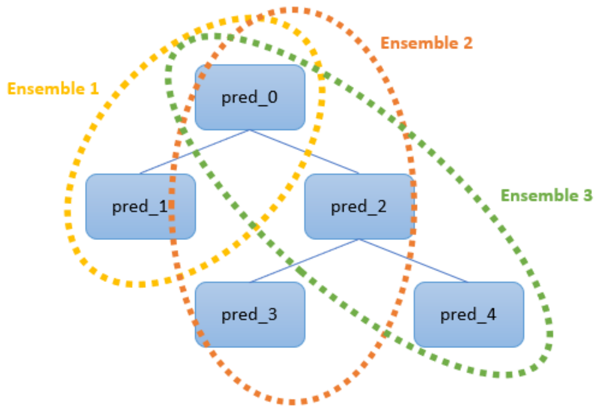


Figure 3: This ToPs tree contains 3 ensembles. The first one is formed by pred_0 and pred_1. The second one is formed by pred_0, pred_2 and pred_3. The third one is formed by pred_0, pred_2 and pred_4. There is no problem using a predictor in more than one ensemble.

Notice one important property of the predictors found along a path. The first ones, which are closer to the root, contain lots of instances, but a more complex feature space. For this reason, they may have high bias but low variance. On the other hand, the ones closer to a leaf contain less instances but a more specialized feature space. For this reason, they may have low bias but high variance. Using all these predictors, with a high diversity in terms of bias and variance, can create a powerful ensemble. Recall that balancing bias vs. variance is one of the main issues in the design and application of ML algorithms.

1.2. Objectives / Motivation

This section explains the reasons why we decided to do that work and the objectives we tried to accomplish.

- **Create a public implementation**

When reading for the first time the original paper of ToPs, one avoidably realizes the potential of that new way of doing ensemble learning. Moreover, the results they present, based on their experiments, seem to support this idea. However, there is not yet any public implementation of ToPs. That means nobody except the authors can take advantage of its potential.

We asked the author Jinsug Yoon for any implementation of the algorithm but his answer was "they were not making it public for strategic reasons".

Given all the potential that seems to have ToPs, we think the world deserves a public implementation for anyone who wants to use it, either for research or for application.

- **Make the intuition of the algorithm easier to understand**

The original paper that explains ToPs is a scientific publication and, as a consequence, it is not intended to be understood for a general public. It is true that anyone needs a certain background to understand a scientific publication, but there is always room for improvement in terms of making the concepts more accessible. This is just one of the objectives of this work. We think that the ideas are not worth if we cannot spread them.

At a first glance, when reading the original paper, the algorithm may seem strange and confusing. In section 1.1 we tried to simplify the idea of the algorithm and gave a good general intuition to the reader. In section 4 we will explain extensively the little tricks that are behind it, in a clear and precise way.

- **Fill the missing information**

In the original paper there are some steps of the algorithm that are not fully explained. Filling these gaps is completely necessary if we want to create an implementation of it and if we want to make ToPs understandable for the public.

- **Make ToPs faster**

ToPs is a time consuming classifier. This is the common behaviour for classifiers that can fit complex models. However, there is always room for improvement.

The ability to fit a dataset using a reasonable amount of time can make the difference between two classifiers, even if the other is worse in terms of accuracy. A fast algorithm is always more appealing for data scientists since the training process can be done in small machines and the time it takes each iteration when developing a data pipeline is shorter.

2. Required concepts

In this section we give the theoretical basis to help the reader understand the full description of ToPs, explained in 4 and 5.

2.1. Tree-based methods

Tree-based methods are a family of ML predictors that produce tree-like structure that divide the feature space in smaller regions. They typically create their tree structure using a recursive procedure. In this procedure, we start with a dataset. The objective is find a way to split this dataset and produce two or more subsets. What they expect is that the behaviour of the instances from a dataset is not equal in all the regions of the feature space. For that reason, segmenting the initial feature space in smaller regions helps to find subspaces where it is easy to describe the behaviour of the target variable.

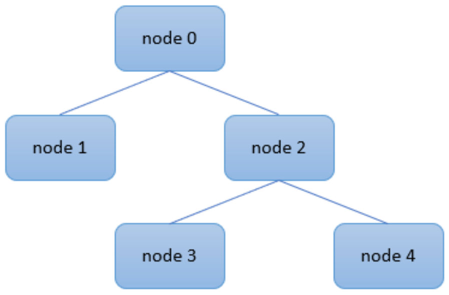


Figure 4: Example of the structure of a tree like predictor

The main three factors that differentiate the tree based methods between them are the way they find splits, how they evaluate the goodness of a split and which criteria they use to stop repeating this recursive procedure.

In section 3.1, we will explain CART (Classification And Regression Tree), one of the most famous algorithms to produce a Decision Tree.

2.2. Comparing two predictors

In ML there exists lots of different predictors that can be used for the same problem. From this set of predictors we need to choose the best one. That's why we need a way to evaluate and compare them. Firstly, we need to decide which metric we will use to measure how close the prediction of a predictor is from the real value. We will call this metric as **goodness function**. We will express a goodness function as:

$$g(y, \hat{y}) \tag{1}$$

Once we have a suitable goodness function, we need to decide how we will estimate the expected value for that metric. The problem is we do not know from which population our data comes from. What we only have is a finite sample. To overcome that problem, we could use the whole dataset for training and for evaluation, but this is not a good idea. It's well known using the same data for evaluation that was used for training gives overestimated values for any goodness function. For example, a predictor that simply memorizes the training set would give no error when predicting any instance of this training set and the goodness function would rank it with a high value. In statistics, this is not what we want. The objective is to predict unseen instances and we want to select the predictor that will be the best in that task.

For that reason it is a must to use instances that were not seen at training time. There are several methods that do so.

- **Train/validation sets**

The easiest technique is to divide the original dataset into two subsets. The first one is used for training the predictors and the second one to evaluate them.

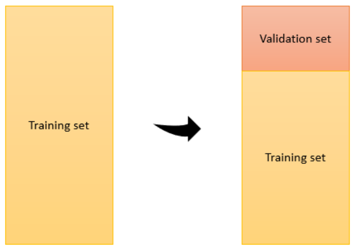


Figure 5: The original dataset is splitted into a training set, used to train the predictors, and a validation set, used to compare their performance

Despite the fact that this technique is very simple and fast, there is a problem. The performance of the predictors may be affected by the way we split the dataset.

- **Cross-validation**

Cross-validation (CV) tries to reduce the variability of the goodness function by averaging several rounds of train/validation. The main concept is that it uses all the instances for the evaluation process. It does so by dividing the dataset into k subsets. Then, it performs k rounds of training/evaluation. At the i^{th} round, the predictor is trained with all the subsets except the i^{th} one and this i^{th} subset is used for evaluation. Finally, it averages the results from each round.

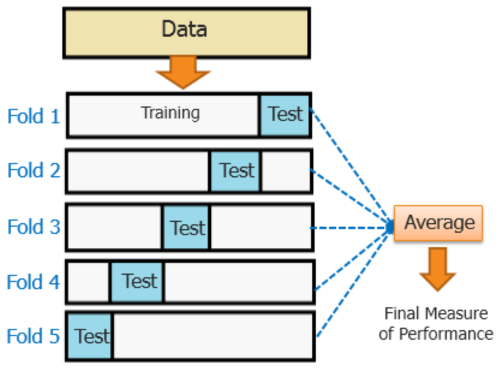


Figure 6: This image represents a 5 fold cross-validation

In this work we will use k-fold CV but there are other variants of CV as Leave One Out Cross Validation (LOOCV). This last case is simply a specific case of the general k-fold CV, where the k is equal to the number of instances.

2.3. Hoeffding/Chernoff bounds

We will use the following probabilistic bounds in a specific point of our version of ToPs. The Hoeffding [3] and the Chernoff [4] bounds give an interval of confidence for the expected value $E[X]$ of a Bernouilli variable, given only a sample of size n. They are widely used in the design of probabilistic algorithms and also often in the design of ML algorithms [5, 6].

Let X be the mean of n samples from a Bernouilli variable. We are interested in computing that the probability the expected value from the population is out of a given interval.

$$Pr\big[|X - E[X]| > \epsilon\big] \tag{2}$$

The Hoeffding bound states the following, for every ϵ:

$$Pr\Big[\big|X - E[X]\big| > \epsilon\Big] \leq 2 \cdot e^{-2\cdot\epsilon^2\cdot n} \tag{3}$$

The Chernoff bound states that for $\epsilon < 1$:

$$Pr\Big[\big|X - E[X]\big| > \epsilon\Big] \leq 2 \cdot e^{-\epsilon^2\cdot n/(3\cdot E[X])} \tag{4}$$

2.4. Ensemble learning

Ensemble learning is a technique in which we use several base predictors instead of only one. The ensemble methods differ between them depending on the way they select the predictors that will be part of the ensemble and the way they aggregate them. Examples of widely used ensemble methods can be found in section 3.2.

2.5. Optimization problems with restrictions

Many problems can be expressed using the following expressions:

$$MAX\Big[f(X)\Big] \qquad \text{subject to} \quad B \cdot X = 0 \tag{5}$$

That means we want to maximize the value of a generic function $f(X)$, where X can represent a vector, but not all the values that X can take are allowed. X must follow a set of linear restrictions. These problems are called non-linear linearly-constrained optimization problems.

We will not enter into the details of the methods that solve these types of problems, but we want to express this problem in a formal way because we will need it later in section 5.1.6.

3. State of the art

3.1. Decision tree

A Decision Tree is a predictive algorithm that works by traversing a tree-like graph, from the root to a leaf, following a path determined by the features of an instance. In other works, at each node, it asks a question about the instance and it chooses the next node based on the answer to this question. To understand it better, we will use the classic example of the Titanic.

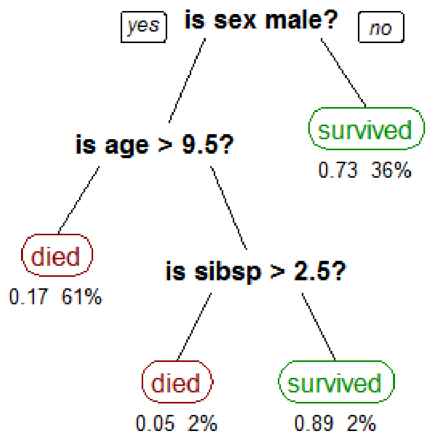


Figure 7: Decision tree that predicts whether a person survived in the Titanic disaster

The last figure represents a tree-like structure to predict whether a person survived or not to the Titanic disaster. As explained before, the decision tree makes some choices using the features that describe a passenger to traverse the tree until it reaches a leaf. Then, it looks at the label this leaf has and this is the prediction it finally makes.

Decision trees have been existing for long, but they started being widely used thanks to the invention of algorithms that could automatically build them from a dataset.

3.1.1. CART

Classification And Regression Trees [7] is one of the first algorithms to build automatically a decision tree from labeled data. It builds the tree using a greedy procedure. At each node, it creates a tentative split for each variable. The splits are always binary and they are created finding the threshold that minimizes the gini impurity of the two new subsets. The selected variable for the final split is the one that minimizes the most the gini impurity.

$$gini(p) = 1 - \sum_{i=1}^{\#classes} p_i^2 \tag{6}$$

We stop the expansion of a node if it contains only one instance or if all the instances that contains belong to the same class. Then, using a validation set or cross-validation the tree is pruned.

3.1.2. Applying a predictor in the leaves

Decision Trees can be combined with other predictors. This is the core of the intuition behind ToPs. One of the first cases where it was done this combination was in [8]. They grew a standard

Decision Tree (with univariate splits) but in the leaves they trained a Naive Bayes classifier. This idea has been further extended, for example in [9], where they used more general learners in the leaves.

3.2. Ensemble methods

Ensemble methods are a family of predictors that take several base learners and combine their predictions. Depending on the way the base learners are created or the way they are combined, there exists different types of ensemble methods.

3.2.1. Bagging

Bagging [10] consists on training the same type of base predictor several times using different subsets of instances sampled with replacement from the original dataset. That means the instances are sampled with uniform probability and they can be repeated.

Bagging is usually applied with Decision Trees as its base learners. It normally improves the performance with respect to a single decision tree, since each tree tends to have high variance but low bias and averaging several trees eliminates this variance.

3.2.2. Random Forests

Random Forests [11] is an example of an ensemble method that uses bagging. However, it decreases further the correlation between the different predictors by adding an extra step of randomization in its training procedure.

Random Forests uses Decision Trees as its base predictor. Each tree is constructed using instances sampled with replacement from the original dataset. A split from a tree is created by selecting randomly a small subset of variables and taking the best one, in terms of reduction of the gini impurity. This is the step that adds more randomness to the trees and helps reduce the correlation between them. Finally, for doing a prediction, it averages all the predictions from the trees.

3.2.3. Boosting

The idea behind boosting is that several weak learners can create a strong one. A weak learner is a predictor that is only slightly correlated with the target variable. In this section we will illustrate how boosting works by explaining Adaboost [12], one of the most famous boosting algorithms.

Adaboost is an iterative algorithm that, at each iteration, trains a predictor and modifies the weight each instance has, depending on if it has been correctly classified or not. At the beginning, all the instances have the same weight ($1/N$). Then it starts iterating. At the beginning of an iteration it trains the predictor and it calculates the error of it. An incorrect predicted instance will increase the error proportionally to the weight associated to it. Then, we give to this predictor a weight (different from the ones the instances have) based on its performance. This weight (associated to the predictor) will be used later to average it and all the other predictors obtained from the different iterations. The last step of an iteration is to update the weights of the instances. That means the incorrectly classified instances will increase their weight.

3.2.4. Stacking

Stacking [13] combines in a more general way the predictions from its set of base learners. Moreover, the set of base learners tends to be formed by very different predictors, in contrast to Random Forests, which uses Decision Trees as its unique type of base learner. The combination of the predictions of the base learners is done by a meta-predictor that learns how to combine the predictions from them. It is normally used Logistic Regression as the meta-predictor. In that case, it takes as features the predictions of each base learner, it combines them linearly and it outputs a prediction which is the one used for the overall ensemble.

Another way to see what stacking does is by imaging that each base predictor creates a new variable, which is the prediction it does. This creation of new variables can be seen as a feature extraction process. Then, the meta-predictor takes as input data these new extracted variables.

4. Tree of Predictors, original proposal

4.1. The algorithm

In the original paper, the authors divide the description of the training process in two major steps. The first one sets the shape of the tree. That means it includes the process of choosing attributes to split the nodes and choosing the most suitable predictor for each one of them. The second converts different sets of predictors into a set of ensembles.

4.1.1. Split process

This process is the one that grows the tree. It is applied recursively. In this section we will explain how a single node is splitted.

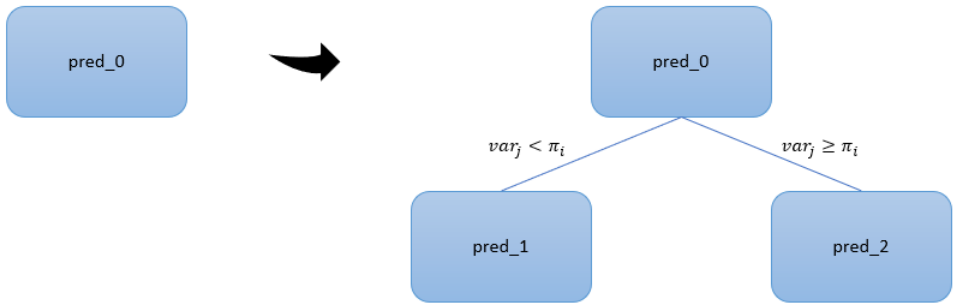


Figure 8: At the left we have a parent node. At the right we have applied the splitting procedure to this node, creating two children. The splitting process finds a variable var_j and a threshold π_i to divide the dataset. Then it chooses a predictor for each new created node.

Naive approach

This step starts assuming the current node contains a subset of the original dataset and a predictor already trained. Then it tries to maximize the following expression:

$$MAX\Big[g(X, var, \pi, pred_1, pred_2)\Big] \tag{7}$$

Where g is the goodness function, X is the original dataset, var is a variable from the dataset, π is a threshold over the variable var that divide the dataset into two halves and $pred_1$ and $pred_2$ are two predictors. That means that it looks for the optimal variable, split point and predictors (all of these factors at the same time) to maximize a goodness function. Later we will dig deeper on the goodness function and how we estimate the generalization value of it. At this moment, the reader only needs to know that the goodness function takes as input the predictions/probabilities from a predictor and assigns them a value. The larger this value is, the better.

To understand better the splitting process, we could imagine the algorithm iterates over all the variables. For each variable, it iterates over all the thresholds that divide the dataset into two halves. For each half, we train all the available predictors. Finally, we choose the variable, threshold and the two predictors that give the best value for our goodness function. That means we end up creating two subsets from the parent dataset with a predictor associated to each of them that maximize the overall goodness function.

Algorithm 1 Split step (naive approach)

```
1: for $j \leftarrow 1, \#vars$ do
2:     for $i \leftarrow 1, \#inst$ do
3:         $X_1, X_2 \leftarrow split\_dataset(X, var_j, \pi_i)$
4:         for $k \leftarrow 1, \#predictors$ do
5:             $g1 \leftarrow g(pred_k, X_1)$
6:         for $k \leftarrow 1, \#predictors$ do
7:             $g2 \leftarrow g(pred_k, X_2)$
```

Performance problems of the naive approach

However, this is not exactly the way the authors propose the implementation of this recursive step. A simple look at it will show that the time complexity is rather large. The main problem is we have three loops. The first one iterates over all the variables. The second one iterates over all the possible ways to split the dataset into two halves using the selected variable. This value is the number of instances. The third one iterates over all predictors and fits each one of them to the corresponding subset previously generated. So, the time complexity is:

$$O(\#vars \cdot \#inst \cdot \sum_{pred \in predictors} O(pred, \#vars, \#inst)) \tag{8}$$

That means, for instance, if we have a single predictor like Logistic Regression, the time complexity for a single recursive step is:

$$O(\#vars^3 \cdot \#inst^2) \tag{9}$$

Assuming the time complexity of Logistic Regression is:

$$O(\#vars^2 \cdot \#inst) \tag{10}$$

Reducing the time complexity of the split process

From the three factors that affect the time complexity, which are $\#vars$, all possible thresholds ($\#inst$) and the implicit time complexity of each predictor, the authors decided to focus on the second one. They propose not to look for all the possible thresholds that divide the dataset into two halves, given a variable. Instead, they propose to use a single threshold (2 bins) for binary variables and 9 thresholds (10 bins) for the numerical ones. With that change, the possible number of thresholds becomes a constant value and the time complexity of the recursive step is now:

$$O(\#vars \cdot \sum_{pred \in predictors} O(pred, \#vars, \#inst)) \tag{11}$$

Choosing the thresholds

Let us focus in more detail on the way the thresholds are picked. As said in the previous paragraph, there are two ways of picking the thresholds depending on the nature of the variables. The authors talk only about binary and numerical variables, so we assume these are the only ones supported by their algorithm. Categorical variables are not directly supported, but they can be transformed to binary ones using one-hot encoding. The binary ones can only take two discrete values. Without loosing generality, we can assume $var_{bin} \in \{0, 1\}$. The threshold chosen is 0.5, but any threshold between 0 and 1 could work because we only need a way to distinguish whether the value of the variable is 0 or 1. For a numerical variable we divide it in 10 percentiles. That means we put 9 thresholds in a way that they produce 10 bins with almost the same number of instances in each bin. We use each of these 9 thresholds to split the dataset into two halves. We do not use them to split the dataset into 10 subsets. We use them to create 9 tentative binary splits.

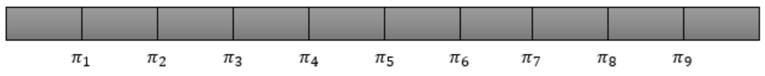


Figure 9: When splitting a numerical variable, we try the 9 thresholds that come when we divide the variable into 10 percentiles. Each of these 9 thresholds is then used to create a tentative binary split

How to compare different predictors: the goodness function

The split process also includes the selection of the best predictor for each new tentative subset. Now it is time to dig deeper on the goodness function and the way its generalization value is calculated.

The goodness function must be something that takes as input value the prediction/probabilities from a predictor and the real values/labels and outputs a value that summarizes how well this predictor did its job. Its objective is to express in a numerical way the discrepancy between the real values/labels associated to a set of instances respect the predicted ones [14].

$$g(y, \hat{y}) \tag{12}$$

If our goodness function is the accuracy, it will tell us how many instances were correctly labeled. Other goodness functions that could be used in ToPs are the AUC or the logarithm of the predicted probability for the true class.

How to compare different predictors: the use of a validation set

The big question, now, is: which data do we use to rank the predictors using the goodness function? In section 2.2 we have presented the problem of comparing two predictors and which techniques can we use to solve it.

Remember that the same data that was used to train the predictor is not suitable for that purpose. That is why the authors divide the original dataset into a training set and a validation set. The predictors are always trained using the instances from the training set and they are scored with the goodness function using instances from the validation set. Since these last instances were not seen at training time, it gives a good estimation about how well is this predictor predicting unseen instances, according to the goodness function.

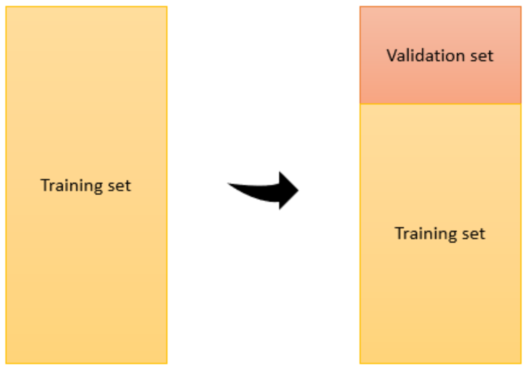


Figure 10: The original dataset is splitted into a training set, used to train the predictors, and a validation set, used to compare their performance

Selecting the predictor for a node

Now, we will continue focusing on the training process of the base predictors. In the last paragraph we said we only use the training dataset to train the base predictors. That's true. But there's also a subtle detail that has not yet been explained.

One could imagine we train each predictor with the instances associated to the tentative subset. That means that we divide our training dataset according to the selected variable and threshold, we create two subsets and we train some predictors using the instances of these subsets.

However, the authors propose a better approach. Instead of using only predictors trained with the instances associated to a tentative new node, they suggest using also predictors trained with the instances from any ancestor node.

When we split a dataset in two, we get two subsets which are smaller than the parent dataset but they have a more restricted feature space. On the one hand, having less instances for training tends to give worse results. With few features, the tendency is to overfit and to have a bad generalization error. On the other hand, a restricted feature space is easier to learn for a predictor than a large space. This is because, when the feature space is more restricted, we have less interactions and the behaviour of the target variable is easier to express. As we split recursively the nodes we get fewer instances with a feature space which is more restricted.

The idea of using predictors trained with the instances of the current node or with the instances of some ancestor may help in the following situation. Imagine we split a node. The left child has less instances than the right child and the right child has a subset of the feature space which is easier to learn. The consequence of this situation is the predictors trained with the instances of the left child give a bad performance, while the predictors trained with the instances of the right child give a good performance. However, the overall performance of these two nodes is worse than the one from their parent node. But if we use in the left child a predictor trained with the instances, for example, from its parent node, we may improve its performance. The associated predictor to the left child is trained with more instances and we may avoid overfitting problems. Now, the overall performance of the two children is better than the one from the parent node, thanks to allowing one of the node to take a predictor trained with a superset of instances.

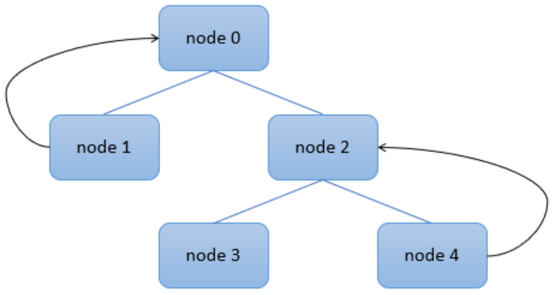


Figure 11: In the following tree, node1 uses a predictor trained with the data of node0 and node4 uses a predictor trained with the data from node2. All the other nodes use predictors trained with their own data

Stopping criteria

Now, it remains to say how to stop repeating recursively the splitting procedure.

At this point, the algorithm has found the feature, threshold and predictors (for each new subset) that maximize the goodness function. What we need to know is whether these two new nodes have a better performance than the parent node in terms of the goodness function. This is the same as comparing two predictors. The first one is the predictor associated to the parent node and the second one is the predictor formed by the binary decision that the split represents plus the predictor associated to the corresponding child.

For example, to predict an instance using the parent node we simply put that instance to its associated predictor and we get its result. On the other hand, to predict an instance using the two recently created children nodes, we first apply a binary decision on that instance to decide in which direction this instance must go, based on the feature and threshold optimized in the split process. Then, we put that instance to the predictor associated to the corresponding child node and we get a prediction.

This is exactly what is done for all the instances of the validation set. These instances are predicted using the parent node and the two children. Finally, the goodness function uses these predictions to estimate the performance of the parent node versus the performance of the children. If the children have a better performance, the split process is applied recursively. Otherwise, it is stopped and the children become leaves.

4.1.2. Aggregation process

In the last section we have seen how a tree is grown according to the methodology described in the original paper of ToPs. After applying the split process, we have a tree with a predictor associated to each node. Now, we need to combine them to output a single prediction for a new instance.

What the authors propose is to create several ensembles. Each ensemble will be composed by all the predictors we find along the path from the root node to a leaf. To create an ensemble we need to have an strategy to combine all the predictions from the different predictors. This combination can be generally expressed with the following equation:

$$\hat{y} = f(\hat{y_1}, \hat{y_2}, ...) \tag{13}$$

The authors restrict the $f()$, so they only allow it to be a linear function. As a result, it has the following shape.

$$\hat{y} = \sum w_i \cdot \hat{y_i} \tag{14}$$

Moreover, the w_i must be positive and they must sum 1.

$$\sum w_i = 0 \tag{15}$$

$$w_i \geq 0 \qquad \forall i \in something \tag{16}$$

Thanks to all these restrictions (linear function plus restrictions over the w_i) we can easily interpret the ensemble. Its output is simply a weighted average of each individual predictor. We are taking every individual predictor we find along a path from the root to a leaf and we are weighting it. This is the aggregation method they propose.

Notice that we have a set of weights for each path from the root to the leaves. This is the same as the number of leaves, since a tree has only one unique path from the root to a leaf. That means the weights are associated to a path, so a node can have several weights if it belong to different paths. The extreme case is the root, which has as many different weights as leaves has the tree, because to reach any leaf we must pass through the root.

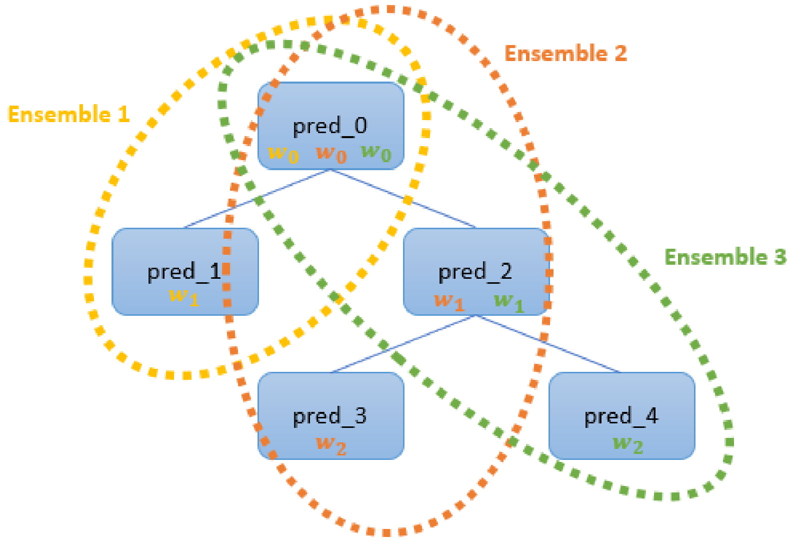


Figure 12: Each ensemble in ToPs is formed by weighting the predictors along a path from the root to a leaf. A predictor, if several paths pass through it, will belong at the same time to different ensembles and, as a consequence, will have several weights, one for each ensemble.

The question now is: how do we set the values for the w_i? In the splitting process we wanted to maximize the goodness function, so now it seems reasonable to choose weights such that they maximize the goodness function for the ensemble.

$$MAX\Big[g(y, \hat{y})\Big] \tag{17}$$

The problem now is not only limited to solving the maximization problem expressed in the last formula. We must decide which data we will use for it. Reusing the training set does not seem fair. Predictors that overfit will have an overvalued weight. Reusing the validation set from the splitting process is not fair too. The predictors have already seen these validation data and we have chosen them according to their performance using it, so it is not a good choice.

The solution from the original paper is to create a second validation set, which will be used only to solve this maximization problem. As a result, the original dataset will be divided in a training set, a validation set 1 (V1) and a validation set 2 (V2). The use of the validation set 2 in this step will give non-biased predictions and the weights will be set to "fair" values.

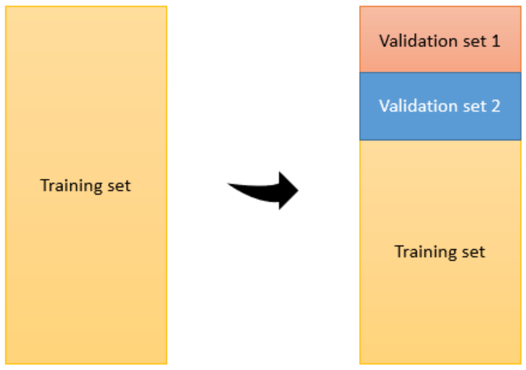


Figure 13: At the end, the original dataset is divided into a training set, a validation set 1 (used for the splitting process) and a validation set 2 (used for the aggregation of the predictors)

4.1.3. Making a prediction

Making a prediction is the easiest step of ToPs. We simply have to put an instance in the root node and let it reach a leaf according to the tree structure and the value of its features. Then, we use the predictions made at each visited node and we aggregate them using their corresponding weights.

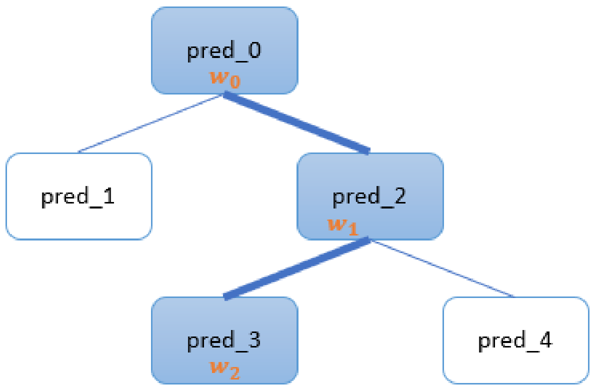


Figure 14: In that case, the instance enters in the node0, goes to the node2 and finally finishes its path in the node3. The predicted class probabilities for that instance is the class probabilities of the predictors found along this path multiplied by the corresponding weights ($w_0 \cdot \hat{y}_0 + w_1 \cdot \hat{y}_1 + w_2 \cdot \hat{y}_2$)

4.2. Missing information

Although it may seem the ToPs algorithm is explained well enough for anybody that may want implement it from scratch, there are a few missing details.

- **How do we avoid nodes with too few validation instances?**

What does it mean avoiding nodes with too few validation instances? Well, as we split nodes and grow the tree we end up having nodes with a dataset associated to them that may be small. As a consequence, the instances that belong to V1 or V2 (validation sets) may be only a few.

The success of ToPs is mainly a matter of choosing good predictors for the nodes and setting good values for the weights. If we are able to put the best predictor in each node and aggregate all these predictors in the wisest way, then we will have an extremely good overall predictor. The way we choose one predictor or another in a node is using the V1 data and the way we set the weights is using the V2 data. We trust these validation data to approximate the generalization error of the predictors. The confidence interval of this approximation is a function of the amount of data used to estimate it. When the number of instances is low, then we may fail to give a good estimation. The result is that we cannot know which is the best predictor nor we cannot aggregate them fairly.

However, in the original paper they do not talk about the problems related when having too few validation instances. The only criteria to stop the splitting procedure is when no children improve the parent node in terms of the goodness function. This criteria is not enough and we think we should also take into account the amount of validation instances.

- **How to choose the size of the validation sets?**

The size of the validation sets is an important hyperparameter of ToPs. However, the authors simply say that in their experiments they used 15% of instances for the V1 set and 10% of instances for the V2 set. They don't justify these numbers nor they suggest any heuristic to choose them.

We think the algorithm should have some logic to set automatically the proportion of data used as validation sets. Or at least, the paper should give some tips to the user to set these values. Balancing wisely size of the training set and validation sets is one of the keys to the success of ToPs. ToPs is simply an ensemble and it will perform well if each individual predictor is a good one and if they are aggregated wisely. If these sizes are not well balanced and any of these 3 sets (training, V1, V2) is too small, ToPs will fail. The next paragraphs explain why.

Predictors have different learning curves. That means the function that describes the performance of a predictor based on the amount of training data may be different depending on the predictor. Now, imagine the training set is too small. In this situation we may choose a simple predictor, which performs regularly with a few data, instead of choosing a more sophisticated predictor, which performs bad with a few instances but is extremely good with a large training set. This example has shown how a small training set may end up selecting not the best predictor.

When we create a new node, we have to assign to it the best predictor. If we cannot estimate with enough precision the true value of the goodness function for each of these predictors, then we don't have a clear way to choose the best one. The consequence is that we may end up choosing a bad predictor, which will not help in the performance of the overall ensemble. This example has shown how a small V1 set does mistakes when choosing a good predictor.

Once we have all the predictors chosen for each node, we have to aggregate them (create an ensemble). As shown before, the weights we give to each predictor are related to their performance on the V2 set. In this case, a small V2 set may give bad estimations of the optimal values for the weights.

- **Justification about how to split a numerical variable**

In the original paper, the authors say they divide a numerical variable in 10 percentiles of almost equal size. Then, they use the 9 thresholds that define these percentiles to try 9 different splits over the dataset, using the current variable.

The way we divide the feature space is also a key point of the algorithm. Finding regions of this space which contain enough instances and are easy to learn will improve dramatically the performance of ToPs.

However, the authors don't justify their proposal. They don't give any intuition about why this method should give good enough subspaces from the original feature space. Moreover, they don't take into account the computational resources spent trying 9 different splits for each numerical variable.

- **How do they solve the optimization problem (aggregation) when the goodness function is the accuracy?**

As explained before, the predictors we find from the root to a leaf are aggregated using a weighted average. That means we multiply the prediction of each predictor by a weight and then we sum all these results. The values of the weights try to maximize the goodness function for the ensemble. The problem is that the authors simply write in a general way the formula they want to optimize, without saying how to solve this optimization problem.

For smooth goodness functions, this problem can be solved easily using some package that implements optimizers for smooth functions with linear restrictions. However, if we use the accuracy as the goodness function, this problem is not straightforward, because the target function is no longer smooth.

4.3. Areas of Improvement / Limitations

In the description of ToPs, the authors show a satisfactory method to build a new and different ensemble predictor. They identify several problems that need to be solved in order to have a successful predictor. Some of this problems are how to use the initial dataset to train and evaluate the predictors or how to improve the performance of a split by reusing some of the predictors from the ancestors. All this tricks and more, explained in section 4.1.1, give a strong potential for ToPs.

However, there is always room for improvement. The next paragraphs focus their attention to some areas that could be improved or that may be problematic.

- **Ability to handle directly categorical variables**

As explained in section 4.1.1 the algorithm can only handle numerical variables. It uses numerical thresholds to do the splits and, for this reason, the variables must be numerical. We also explained that this is not a big issue, since a categorical variable can easily be represented using numerical variables if we one-hot encode it.

However, there are techniques to handle directly the categorical variables without the need to one-hot encode them. This may give other ways to split the feature space. For example, creating a new node for each different category of the variable is a simple way of handling it directly. This method would reduce the depth of the tree and would also reduce faster the size of the dataset as we travel deeper. Is this better than the current way of handling categorical variables? To answer this question, an implementation of that method and some experiments are needed. Maybe this technique would be proven to be worse than the current one, but it is for sure that the current one is not the best one.

Another idea to try when handling categorical variables would be the one proposed by the authors of Catboost [15].

- **New techniques to obtain splits from numerical variables**

This subsection follows the same line as the last one. The way we split the data determines the shape of the tree and this factor, at the same time, determines the data we will use to train and evaluate the predictors. For this reason, we want to point that it may be worth to investigate other ways to split the variables. Now, we will focus on the numerical ones.

In the original paper the only technique to split a numerical variable that is mentioned is to try 9 different thresholds. These thresholds are set in a way that they divide the dataset into 10 percentiles of almost equal size. It is true that by trying 9 different thresholds, at least one may lay close to the optimal threshold, locally. However, this comes at the expense of computational resources. From these 9 ways to split the dataset using a single numerical variable, we only end up keeping one. The other 8 are completely lost.

The structure of the dataset is never looked when doing a split with a numerical variable. We think it may be worth to take into account the internal structure of this variable or the values the target variable takes. Using a method that takes into account at least one of these two factors may end up finding better thresholds with only one shot.

A clustering algorithm could be used to automatically find a single threshold (two clusters) to split the dataset by looking at the internal structure of the variable. The reduction of the gini impurity or entropy could also be used to find a split, in the same way they are used in CART, explained in section 3.1.

- **Criteria to stop growing the tree**

As pointed in section 4.2 the original paper does not mention any technique to determine that there are not enogugh instances for validation to continue growing the tree.

Apart from the problems in accuracy derived by having a small set of validation instances, growing the tree always comes at a cost in computational resources. If the growth is not necessary, why do we have to waste time and memory?

That means, from both the accuracy point of view and the efficiency one, a good stopping criteria is needed for ToPs.

- **Compute the time complexity for the mean case**

In the original paper, the authors analyze the time complexity of ToPs for the splitting procedure and they give an upper bound using the asymptotic notation:

$$O(\#vars \cdot \#inst^2 \cdot \sum_{pred \in predictors} O(pred, \#vars, \#inst)) \tag{18}$$

However, their bound is very pessimistic. To deduce it, they make very conservative assumptions, so the formula does not represent the computational time cost of ToPs in practice.

Later, in section 5.3 we will show how the authors deduce the last formula and how we can get a more optimistic bound for the worst case. Finally, we will show an approximation for the mean case, which will try to reflect its complexity for a standard execution.

- **Reduce the execution time**

Even if the analysis mentioned above is pessimistic, ToPs is a complex classifier, and training time is often a limiting factor when building a classifier on a large dataset.

In big datasets, using ToPs as it is described in the original paper, is completely impractical. In other datasets, it may achieve to finish the training process but it may not pay off the time spent. Simpler and faster methods may do almost equal in terms of accuracy.

One observation is that ToPs discards a large amount of computations. For example, when we split a node, all the tentative splits we create, with all the predictors trained in the tentative children, are discarded except one. That means it is not very efficient if we look at the work it does and we compare it to the work it really uses.

- **Increase the number of datasets used to test the algorithm**

When determining if a new predicting algorithm is good or not we run it in a battery of datasets and we compare its performance again other well established predictors. In the original paper they only use 4 datasets for their experiments and only 3 of them are well-known datasets with public performance numbers using other predictors.

4 datasets are not enough to determine if a predicting algorithm is really good or not. In a formal experimental setup, more than 4 datasets should be used, with varying sizes in terms of instances and variables.

For this reason, we think ToPs is not proven empirically to be better than other state of the art predictive algorithms.

- **Interpretation of the resulting tree**

The authors argue that ToPs can be interpreted, but the reality is they only can plot the shape of the resulting tree. One thing is seeing a tree-like plot and the other one is understanding why and how an instance is predicted to one class or the other.

The structure of the tree is easy to represent, but not an ensemble. Because ToPs is actually a set of ensembles, there is no straightforward interpretation or understanding of what it has learned.

5. Our proposal: A revised Tree of Predictors algorithm

5.1. Problems that we address

5.1.1. Splits for categorical variables

Our implementation of ToPs supports both one-hot encoded categorical variables and raw categorical variables. The way we treat the raw categorical variables when doing the split is the one we suggested in 5.1.1. This means that we create as many tentative children as different categories in the variable.

This feature is totally available for the user that wants to use our implememtation of ToPs. However, in the experiments, we have not tested it in terms of performance and execution time.

5.1.2. Splits for numerical variables

In the case of numerical variables we have implemented the splits as described in the original paper and explained in section 4.1.1. However, we have implemented a new alternative, as suggested in section 5.1.2.

This new alternative consist of finding a single threshold using the reduction in gini impurity. Our algorithm finds the threshold that maximizes the reduction of gini impurity for the children and this threshold is the one we apply to split the dataset. The theoretical advantage of this method is that it tries to find a more meaningful threshold, by using the information of the labels of the instances instead of picking a threshold without looking at the dataset. In principle, this would add more information to the tree and would provide better splits. The practical advantage is that we just try one threshold for each variable instead of 9.

5.1.3. Retrain process

The retrain process consists of training again a predictor with the training and validation instances, to improve its performance, just after the two sets (training and validation) have been used to estimate the generalization value for the goodness function. In our implementation of ToPs, the retrain process is applied in both the splitting phase and the aggregation phase.

In our splitting phase, we first select the best triple feature-threshold-predictors using the train and V1 sets or using CV, as it is explained later in section 5.1.4. Once we have expanded the tree by adding the two children as new nodes, we train again the predictors of each child with all the instances used in this step (all the ones we have at the corresponding node except the ones that belong to the V2 set).

In our aggregation phase, we first use the V2 set to evaluate our predictors and set the weights and, finally, we train again each predictor with all the available instances each node has (train + V1 + V2).

Thanks to this retrain process, we can improve the strength of each individual predictor after having set the shape of the tree.

5.1.4. The validation set 1 (V1)

The importance of the V1 set is crucial for achieving a good performance. It is the set that is used to choose the best triple feature-threshold-predictors. From these three factors, the first two determine the shape of the tree and the last one determines which predictors we will have in our ensemble.

In section 4.2 we pointed out that in the original paper there is no clue about the desirable size of V1. We also remarked how a small size for the V1 may lead to very bad performance for ToPs. This is the reason why this implementation of the ToPs algorithm should have some alternative to simply use some fixed and hand-crafted percentage of the initial dataset as V1.

Our implementation supports using a fixed proportion of instances as V1 and using cross-validation. The idea is very simple but powerful. Imagine in a node we have a dataset with 900 instances devoted to training and 100 instances to validation. If we use 10-fold CV instead of using a single validation set we will have 900 instances for training for each fold and a total of 1000 instances for validation, taking into account all the folds $(10folds \cdot 100instances/fold)$. In that case, we have multiplied by a factor of 10 the number of instances used to judge the performance of the triple feature-threshold-predictors.

If we use CV, apart from using a single V1 set, we can increase the amount of instances used for validation without compromising the size of the training set or the V2 set. However, this technique does not come for free, but it is at the expense of the computational resources. In the previous example, the increase of the instances used for V1 comes with an increase of a factor of 10 for the computational time, because we are training the predictors 10 times, one for each fold.

5.1.5. The criteria to continue splitting

In large datasets, at the beginning, even with a small percentage of instances devoted to the V1, it is enough to estimate well the performance of the predictors. The problem is that, eventually, as the tree grows, the dataset is splitted several times and it becomes smaller for the deep nodes. If we start with $\#V1$ instances for validation, after applying recursively the split process several times we will have $\#V1/2^d$ instances in the nodes of depth d, if we assume the splits divide the dataset in two subsets of equal size. With only a few V1 instances, it is more difficult to compare two predictors with enough confidence. The original paper says we must continue splitting the nodes until we cannot improve more the goodness function. It may happen, though, that we may accept a split even when the two children do not improve the generalization value for the goodness function with respect to their parent.

To avoid this situation, the stopping criteria for the split process should not only take into account if the children are better than their parent in terms of the value given by the goodness function. To really accept a split, this difference must be significant. That means, with probability β, the true value of the goodness function for the children is higher than the one corresponding to their parent.

Firstly, we need a way to compute the probability that the true value of the goodness function for a given predictor is within an interval. To do so, we will use the Hoeffding and Chernoff bounds. However, these techniques assume a bounded random variable. That means our goodness function must output values within a finite interval. For example, the log-likelihood is not suitable for estimating a confidence interval using Hoeffding or Chernoff since the values that returns are within an infinite interval, $(-\infty, 0)$.

In our implementation of ToPs, the use of the Hoeffding and Chernoff bounds implies also the use of the accuracy as the goodness function when splitting the tree. The accuracy simply maps an instance to a value of 0, if this instance is miss-classified, or it maps it to a value of 1, if it is correctly classified. Notice that this means the possible outputs for the accuracy are $0, 1$, so they are within a finite interval.

Now we will take as our random variable if an instance is correctly classified or not. We can assume our random variable is a Bernoulli variable and it has probability p, being p the true value for the accuracy. Our objective is to calculate the true mean value of the population where this random variable comes from. This is the same as knowing the generalization value of the accuracy for a given predictor and population of instances.

As shown in section 2.3 the probability that the mean value of a Bernouilli variable is out of an interval of length ϵ, centered around the empirical mean, given the Hoeffding bound is:

$$D < 2 \cdot exp(-2 \cdot \epsilon^2 \cdot n) \tag{19}$$

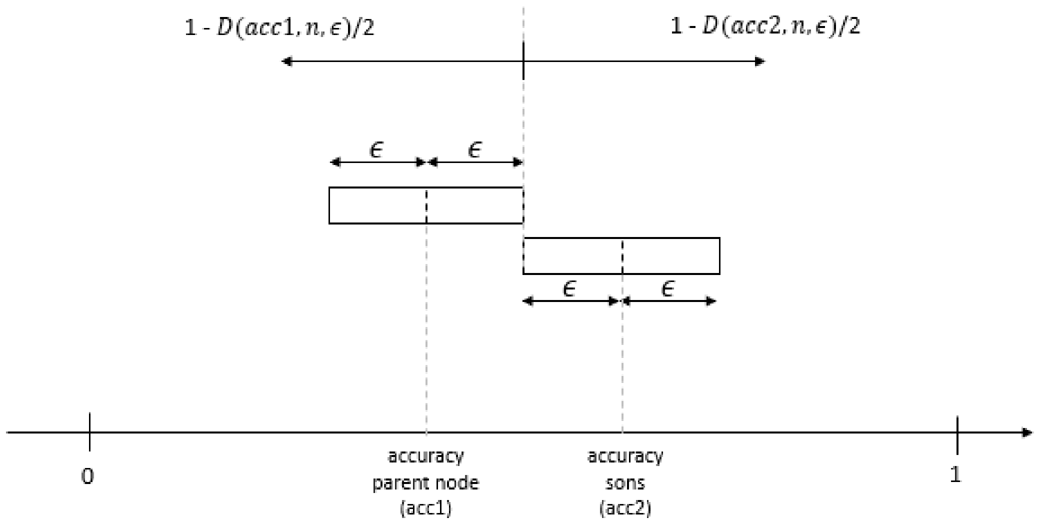
And for the Chernoff bound:

$$D < 2 \cdot exp(-\epsilon^2 \cdot n/(3 \cdot E[X])) \tag{20}$$

Notice that when $E[X]$ is around 0.5 the Hoeffding bound gives a better approximation than the Chernoff bound. However, the Chernoff bound is better when $E[X]$ is close to 0. This is because the Chernoff bound depends on $E[X]$ while the Hoeffding bound does not depend on it.

The typical accuracy of a predictor does not tend to be close to 0. If so, this would be a really bad predictor and we should think using another one. What we expect is having a value for the accuracy far from 0. This seems to be bad news for the Chernoff bound. However, we can transform our initial Bernouilli variable (accuracy) into $1 - accuracy$. In that case, values close to 1 become close to 0 and the variable remains being Bernouilli. Thanks to that trick, we can use the Chernoff bound when our expected accuracy is close to 1 and this helps us to calculate a better bound.

Once we know how to calculate a confidence interval for a Bernouilli variable, we simply have to calculate the empirical accuracy for the parent node and for the children and play with the previous equations to obtain a probability. The following figure and the next explanation will help the reader to understand how to calculate a lower bound for the probability that the children are better than the parent.



1 - $D(acc1, n, \epsilon)/2$
1 - $D(acc2, n, \epsilon)/2$
ϵ
ϵ
ϵ
ϵ
0
accuracy parent node (acc1)
accuracy sons (acc2)
1

Figure 15: Graphical representation of the confidence intervals and probabilities for the true value of the accuracy for the parent node and its children

Imagine the empirical accuracy of the parent node is $acc1$ and for the children, $acc2$. Then, we extend our confidence interval for both accuracies until the two boxes intersect. That means the ϵ, the deviation from our empirical value, is $(acc2 - acc1)/2$. Using this epsilon, for each accuracy, we can calculate if its true value is out of our box. Let $D(acc, n, \epsilon)$ be the probability that the true value is out of the box. $D(acc, n, \epsilon)/2$ is the probability the true value is further than one of the limits. For example, it can be used to calculate the probability that the true value for the children is less than $acc2 - \epsilon$ or the probability that the true value for the parent is greater than $acc1 + \epsilon$. The children are better than the parent if the true value for the parent is less than $acc1 + \epsilon$, which has a probability of $pr \geq 1 - D(acc1, n, \epsilon)/2$ and the true value for the children is greater than $acc2 + \epsilon$, which has a probability of $pr \geq 1 - D(acc2, n, \epsilon)/2$. So the probability the children are better than the parent is:

$$pr \geq (1 - D(acc1, n, \epsilon)/2) \cdot (1 - D(acc2, n, \epsilon)/2) \tag{21}$$

There is a more straightforward case and it is when the $acc2$, from the children, is lower than $acc1$. In that situation, we simply say the children are not better than their parent.

When we use the Chernoff bound, since we do not know in advance which is the real $E[X]$, we assume the worst case, which is $E[X] = acc - \epsilon$.

5.1.6. Aggregating the classifiers

Remember that in the aggregation process we want to convert a set of predictors into a single one by doing a weighted average on their predictions. The weights must maximize the goodness function for the ensemble.

$$MAX\Big[g(y, \hat{y}, w)\Big] \tag{22}$$

As shown in section 4.3 the aggregation process is not completely explained in the original paper. They only mention the formula that has to be maximized, but they don't give any clue about how to solve it or which goodness functions are suitable to be used in the aggregation process.

- **How to solve the optimization problem**

The maximization of the goodness function for the ensemble is not a straightforward problem. The difficult part is dealing with the restrictions. Without the restrictions it would be a simple non-linear non-constrained optimization problem. This kind of problems are easily solved using gradient descent, for example. When we add some constraints, then the difficulty of the problem increases. In our case we have two linear constraints, so our problem is a non-linear linearly-constrained optimization problem.

Luckily, the `scipy` package contains a minimization function called `minimize` that can handle non-linear target functions with linear constraints, which is our case.

- **Suitable goodness functions for the aggregation**

Not any type of goodness function is appropriate for being used in the aggregation process. The major requirement is that the goodness function should be smooth, to make the optimization problem easier.

For example, the accuracy is not a smooth goodness function. An instance can be correctly classified or not. There is nothing in between. So, when we calculate the accuracy using a set of instances, the possible values we may have are:

$$acc = \frac{x}{\#inst} \qquad x \leq \#inst \quad x \in \mathbb{N} \tag{23}$$

where x is the number of instances correctly classified. In the ensemble x depends on w (the weights). The problem is the derivative of x respect to w is not well defined. The consequence is we cannot calculate the gradient of the goodness function with respect to w and we cannot appply gradient descent methods.

An alternative could be using a smooth function that approximates the accuracy function.

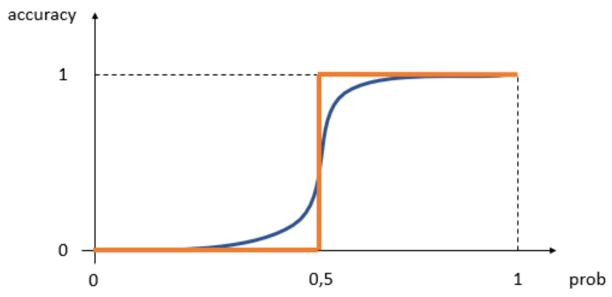


Figure 16: In orange, the accuracy function for a positive class. In blue, a smooth approximation of the accuracy function.

Now, we have a smooth function and this allows us to calculate its gradient. However, the function represented in the previous figure is not convex. Non-convex functions may get trapped in local optimum points.

At this point, we have shown that the goodness function used in the aggregation step must be smooth and it's highly advisable to be convex. However, these requirements are necessary but not sufficient.

For example, a user may decide to use the following goodness function:

$$\sum prob^{+} \tag{24}$$

That means he wants to maximize the sum of the probabilities for the positive class (assuming we are in a binary classification problem).

$$\sum_{i=1}^{\#pred} w_i \cdot \sum_{j=1}^{\#inst} prob^{+}_{i,j} \tag{25}$$

If we represent the second sum as a constant value c_i and we unroll the first sum, the last equation becomes:

$$w_1 \cdot c_1 + w_2 \cdot c_2 + ... + w_n \cdot c_n \tag{26}$$

It is trivial to see that the last expression is maximized giving a value of 1 to the weight associated to the highest c_i and giving a value of 0 to the remaining ones. The result is we will only use one predictor in the ensemble and we will discard the other ones so, in fact, we will not have an ensemble, but a single predictor.

Given all the requirements that must follow the goodness function for the aggregation process, we decided to not parameterize it, but use always a good and fixed goodness function. The candidate we finally chose was the log-likelihood. It's smooth, convex and gives solutions that use all the predictors in the ensemble.

$$\sum_{i=1}^{\#pred} w_i \cdot log\Big[\sum_{j=1}^{\#inst} prob^{+}_{i,j}\Big] \tag{27}$$

5.2. Implementation

The ToPs algorithm has been implemented in Python. There are two main reasons that sustain this decision.

- We want to make the algorithm available to the public. Python is one of the most popular programming languages among data scientists, so an implementation of a ML algorithm in Python will be easier to use and more accessible.
- ToPs is a meta-learner, so it needs base predictor learners in order to be able to run. It is desirable to program it in a language in which many other ML algorithms are written, as is the case of Python.

Our implementation of ToPs is compatible with the `scikit-learn` library. This library is one of the most popular and used libraries for ML. It includes hundreds of tools and algorithms that go from preprocessing data, to supervised and unsupervised learning methods. The compatibility of ToPs with `scikit-learn` comes from two factors.

The first one is the ToPs algorithm itself implements the public interface of a general supervised algorithm. That means we can use ToPs at any place in `scikit-learn` that accepts a supervised ML algorithm. For example, in an already created predictive pipeline we can substitute the current supervised method, like Random Forest, by our implementation of ToPs, without any major change.

The second one is that ToPs accepts as a base predictor any classifier that also implements the public interface defined in `scikit-learn`. Thanks to all the predictors that comes in this library (and a lot more that can come from external repos) our implementation of ToPs can potentially use a huge amount of base predictors.

Our implementation can be found in the following github repo:
https://github.com/jordipiqueselles/tops

5.3. Time complexity

In this section we will enter in more details about the time complexity of our implementation of ToPs.

Remember that ToPs performs two phases, the splitting, or tree creation, process and the aggregation process, so the time complexity of ToPs is the sum of this two steps. Let $t(split)$ and $t(aggr)$ be the time complexity of the splitting and aggregation process, respectively.

$$O(t(split) + t(aggr)) \tag{28}$$

We will mainly focus on the complexity of the split process, which is the one analyzed in the original paper $t(split)$. To do so, as we have done in section 4.1.1, we will start by calculating the cost of a single recursive step. To do a single split we have to select a variable, sort it, pick a threshold (in our case we have a maximum of 9 possible thresholds), create two children and train each predictor with the data of the children. Let N be the number of instances and D the number of variables. The cost of sorting is $O(N \cdot log(N)$. The cost of dividing the dataset in two subsets is $O(N \cdot D)$. The cost of training all the predictors is $O(\sum t(pred))$, but this expression can be simplified if we take the cost of the slowest predictor, $O(t(pred_{max})$. So the cost of a single recursive step is:

$$O(N \cdot log(N) + N \cdot D + t(pred_{max})) \tag{29}$$

In the original paper the authors assume the worst case happens when each split creates a subset with a single instance and another will all the remaining ones. For this reason, they assume the recursive step can be performed at most N times. They also assume the cost of a single recursive split is the same at any place of the tree. As a result of these assumptions, the bound of the time complexity is:

$$O(N \cdot (N \cdot log(N) + N \cdot D + t(pred_{max}))) \tag{30}$$

However, we can improve the previous bound.

The cost of splitting two recently created children is lower or equal to the cost of splitting the single parent node. This is because, although we have 2 nodes instead of 1, we have less instances at each node compared to the parent and the cost of the three factors of the recursive step depends, at least, linearly with N. That means the total cost of the splitting procedure is equal or less to the cost of splitting the root multiplied by the depth of the tree

$$O(depth \cdot (N \cdot log(N) + N \cdot D + t(pred_{max}))) \tag{31}$$

Now, let us focus on the maximum depth of the tree. If all the variables are numerical, at each recursive step any of the two created children will have a 10% less of instances, because we can only use 9 possible thresholds evenly distributed. For that reason, the maximum depth using a dataset with numerical variables is $log_9(N)$, so the total time complexity for that case is:

$$O(log(N) \cdot (N \cdot log(N) + N \cdot D + t(pred_{max}))) \tag{32}$$

If all the variables are binary, then the maximum depth is $min(N, D)$, so the total time complexity for that case is:

$$O(min(N, D) \cdot (N \cdot log(N) + N \cdot D + t(pred_{max}))) \tag{33}$$

If we have both numerical and binary variables:

$$O((log(N) + min(N, D_{bin})) \cdot (N \cdot log(N) + N \cdot D + t(pred_{max}))) \tag{34}$$

6. Experimental setup

6.1. Datasets

In our experiments we have used 18 datasets. All these datasets contain more than 1000 instances and have different number of variables and classes. From these 18 datasets, two of them are the same they used in the original paper (`bank_marketing, online_news_popularity`). `mnist` was also used in the original paper, but we had to discard it because our implementation of ToPs was to slow to handle it. The last mentioned datasets can be downloaded from the UCI repository [16] and the other ones are available with the Weka distribution [17]. They are also available in the github repo where we have the ToPs implementation: https://github.com/jordipiqueselles/tops

In our first experiments we saw that ToPs does not give any advantage with respect to their base learners when the datasets are too small. This is because in small datasets, most of the times we cannot do a single split because the number of instances becomes so small that the predictors overfit easily. For that reason all the datasets we used have at least 1000 instances.

The next table shows the datasets we used with their number of instances, features and classes.

dataset	#inst	#feat	#classes
car.arff	1728	21	4
cmc.arff	1473	24	3
credit.g.arff	1000	61	2
hypothyroid.arff	3770	53	3
kr.vs.kp.arff	3196	73	2
letter.arff	20000	16	26
mushroom.arff	8124	117	2
nursery.arff	12958	27	4
optdigits.arff	5620	64	10
page.blocks.arff	5473	10	5
pendigits.arff	10992	16	10
segment.arff	2310	19	7
sick.arff	3772	53	2
solar.flare2.arff	1066	42	6
spambase.arff	4601	57	2
splice.arff	3190	3465	3
waveform5000.arff	5000	40	3
bank_marketing.csv	41188	62	2
online_news_popularity.csv	39644	58	2

Table 1: Datasets used for the experiments

6.2. Execution context

The executions have been done in a laptop with a processor Intel i5-8250U (8 logical cores), 8 GB of RAM and Windows 10. We used Python3.7 from the Anaconda environment.

To extract the performance metrics we executed a 8-fold CV. The reason to do the 8-fold CV instead of the standard 10-fold CV is time efficiency. Our processor has 8 cores, so the 8 folds can be executed in parallel, simultaneously. If we added a single more fold, then the execution time for an entire CV round would double. We would have liked to repeat 5 times every 8-fold CV round, to have a better estimation of the performance metrics and calculate their variability between different rounds. However, our computational resources did not make it possible.

For the different instances of ToPs that we have tested, we have fixed some common parameters. We have used the 15% of the instances as V1 and the 10% of the instances as V2, in the same way the experiments are done in the original paper. We have set to 20 the minimum number of validation instances a node can have. This value avoids having nodes with no validation instances, but it will not impact much on the growing process (the tree will be able to expand itself at full), since we want to be as close as possible to the original implementation. For the goodness function for the splitting process we have used the accuracy.

6.3. Performance metrics

These are the performance metrics we wanted to extract from our executions:

- Accuracy
- AUC
- Precision
- Recall
- Execution time

From all these metrics the one in which we put more focus is the accuracy.

6.4. Experiments

In our experiments, firstly, we wanted to test the performance of our implementation of ToPs using the configuration parameters that make it the most similar to the one explained in the original paper (section 7.1). Then, we wanted to test two of our possible modifications with respect to the original version. The first one is accepting a split only if the children actually outperform the parent node with probability at least 80% (section 7.2). The second one is using the point that minimizes the gini impurity as the threshold to split a variable (section 7.3).

In the three cases we have used the following base predictors: Logistic Regression (LR), Decision Tree (DT) and Gaussian Naive Bayes (NB). Using these base predictors we have created 4 instances of ToPs: ToPs(LR), ToPs(DT), ToPs(NB) and ToPs(LR, DT, NB).

In section 7.1, what we want to see is how ToPs can improve a single predictor. In principle, thanks to nature of ToPs, ToPs(*pred*) should be at least equal than *pred*, being *pred* any predictor. The reason is ToPs can simply create a tree with a single root node if it sees the performance cannot be improved splitting this first node. Then, we want to check if using several base predictors gives an instance of ToPs better than any that uses only one of these base predictors.

In sections 7.2 and 7.3 we will see how our modifications affect the performance in terms of accuracy and execution time.

Apart from comparing the performance of ToPs versus its base predictors, we wanted to test it against another ensemble method. That is why, in section 7.4, we compare ToPs against Random Forest.

Finally, in section 7.5 we will analyze the execution time in more detail for a single dataset, to see the steps of the algorithm that are the top offenders.

7. Results

7.1. ToPs ”original”

In this set of experiments we tested ToPs with the set of parameters that make it the most similar to the description in the original paper.

In the figures 17, 18 and 19 we can see the difference of accuracy for ToPs and its corresponding base predictor. From these figures we can extract two main conclusions:

- ToPs is not worse than its base predictor
- ToPs is either equal (in terms of accuracy) to its base predictor or it is clearly superior.

7.1.1. LR vs ToPs(LR)

ToPs(LR) has a statistically significant increase in accuracy respect to LR. The mean of that increase is 3%. Notice the pattern that follows the difference in accuracy between ToPs(LR) and LR, represented in figure 17. More or less, half of the datasets have the same performance using the two predictors and the other half have a clearly better performance when using ToPs(LR). In the cases where the performance is similar this is, mainly, because no split increases the accuracy. In the other cases it is because a general function can be approximated by several linear functions defined at different regions. When ToPs can identify these regions, it can fit several instances of LR to approximate the main function. The figure 2b, in section 1.1 gives a hint to understand that intuition.

If we look at the execution time we can see the increase in accuracy is not for free. ToPs(LR) is 3 orders of magnitude greater than LR in terms of execution time and this is a major concern for the futur users of ToPs.

predictor	accuracy	auc	precision	recall	time
LR	0.866	0.947	0.862	0.866	5.435
ToPs(LR)	0.894	0.963	0.892	0.894	1146.931

Table 2: Mean of the performance metrics for LR vs ToPs(LR)

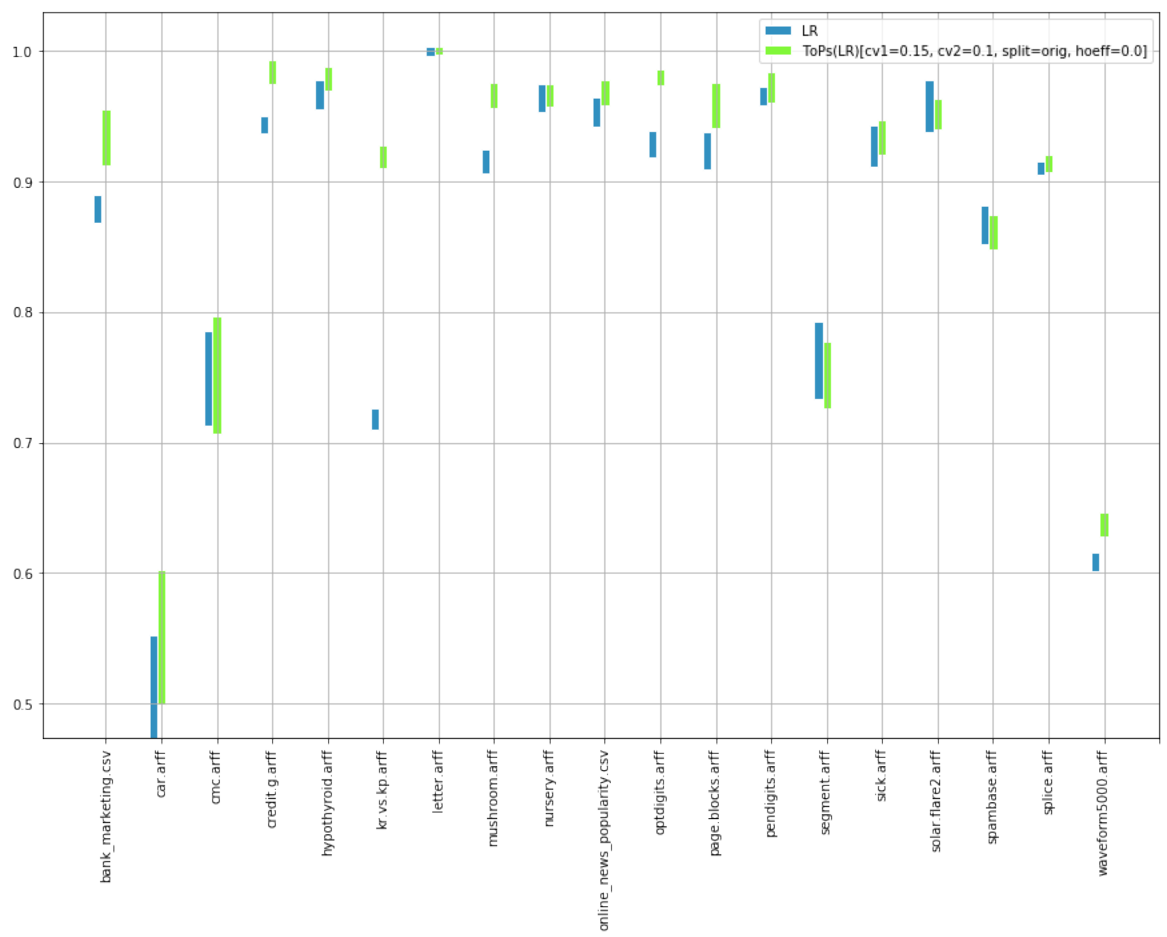


Figure 17: Accuracy for each dataset for LR vs ToPs(LR)

7.1.2. DT vs ToPs(DT)

In the case of DT vs ToPs(DT) there's no difference in the mean accuracy (and no difference for the rest of the performance metrics). Looking at figure 18 we can see this nonexistent difference comes from the fact that all the datasets perform equal. This phenomenon is not strange since both ToPs and DT split the dataset as a way to grow. The result is that the different instances of DT in ToPs are quite similar to each other and this gives a correlated set of predictors for the ensembles.

predictor	accuracy	auc	precision	recall	time
DT	0.873	0.914	0.873	0.873	3.084
ToPs(DT)	0.872	0.931	0.872	0.872	574.418

Table 3: Mean of the performance metrics for DT vs ToPs(DT)

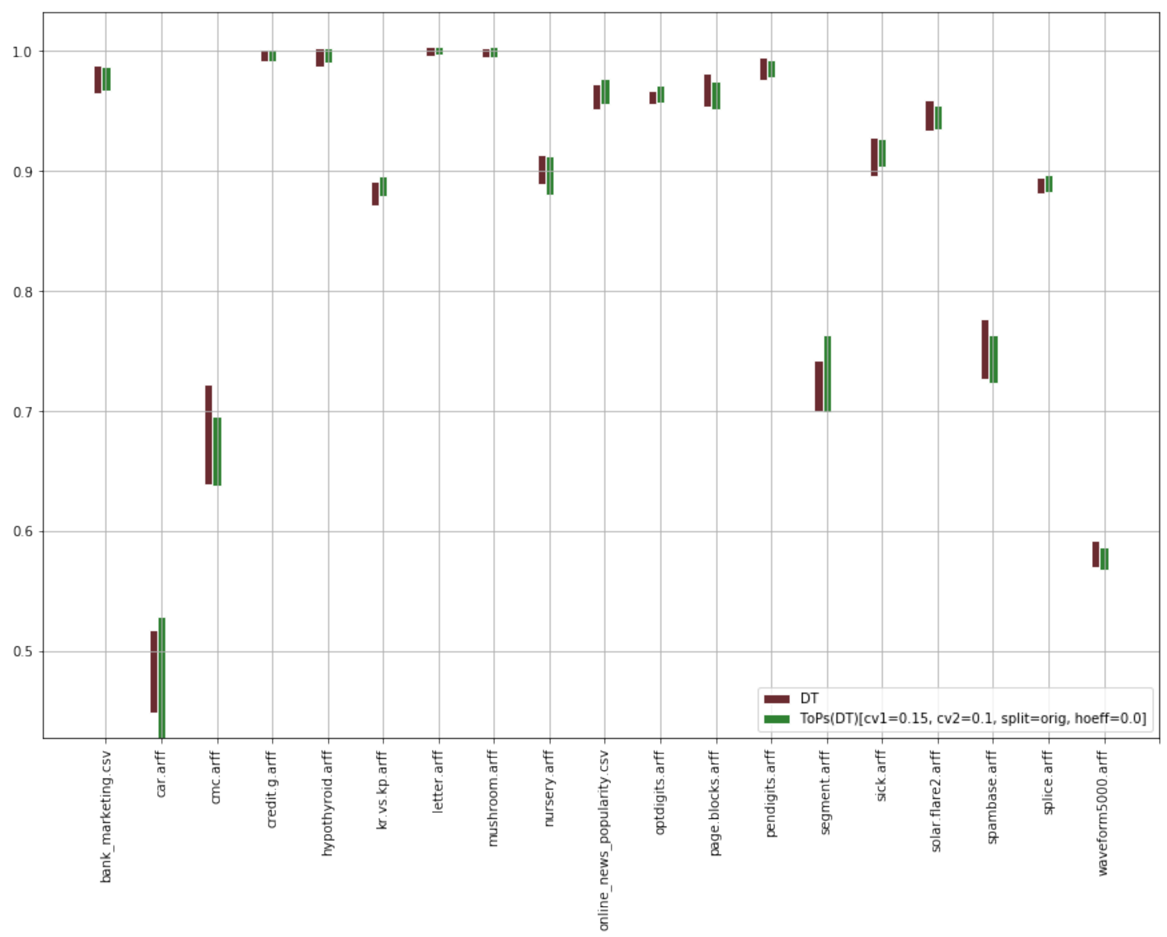


Figure 18: Accuracy for each dataset for DT vs ToPs(DT)

7.1.3. NB vs ToPs(NB)

In the case of NB vs ToPs(NB) the improvement of ToPs is considerable. Only 4 datasets have the same performance using the two predictors. In the others, the superiority of ToPs is significant and, in most of them, the difference is huge. The difference between the mean accuracy of the two predictors over all the datasets is 15%. That means NB can take advantage of a segmented feature space. The idea of using a decision like predictor combined with NB is not new, as explained in section 3.1.2.

predictor	accuracy	auc	precision	recall	time
NB	0.682	0.889	0.797	0.682	2.939
ToPs(NB)	0.838	0.941	0.856	0.838	244.042

Table 4: Mean of the performance metrics for NB vs ToPs(NB)

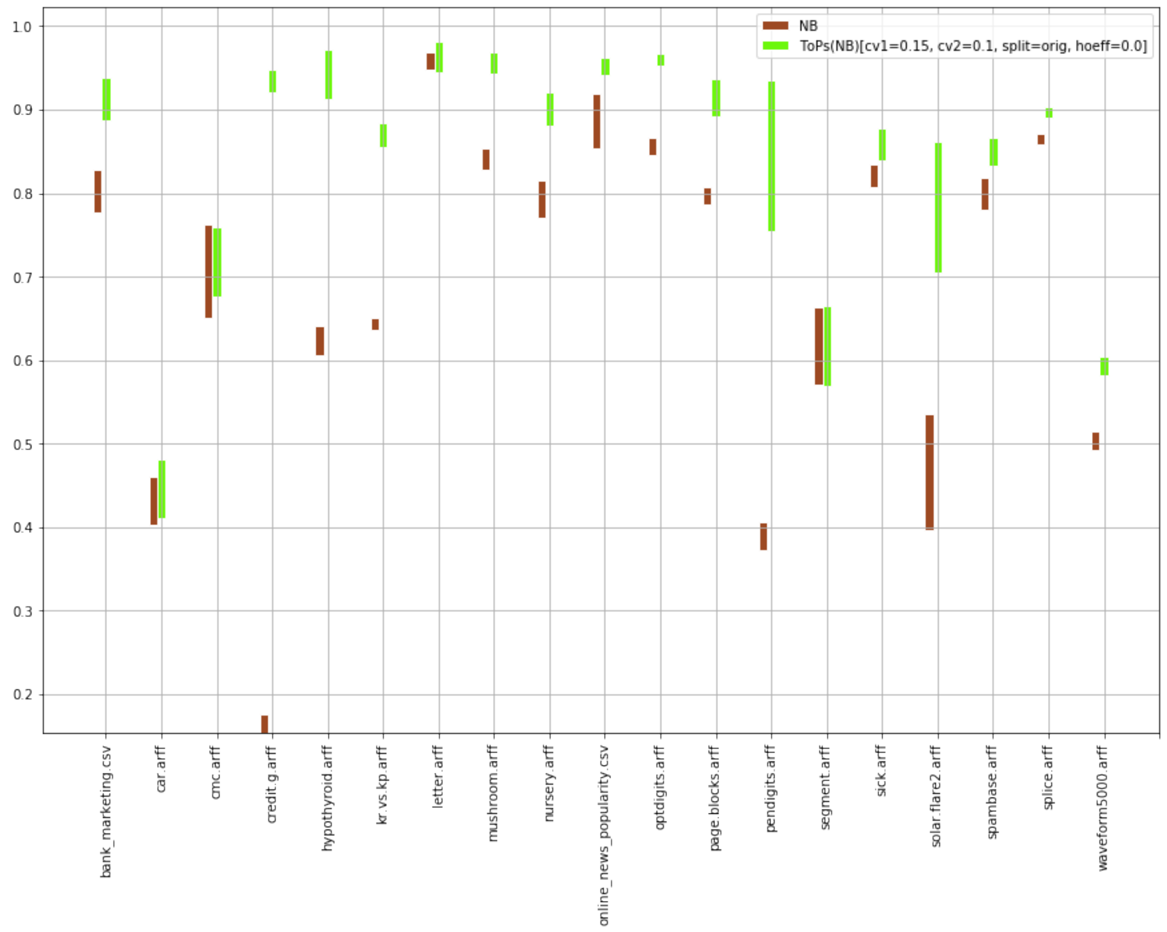


Figure 19: Accuracy for each dataset for NB vs ToPs(NB)

7.1.4. ToPs(LR, DT, NB) vs ToPs(LR), ToPs(DT), ToPs(NB)

The results of these experiments were not the ones we expected. It seemed that using our three base predictors in ToPs would outperform any instance of ToPs with a single base predictor, but it is not the case. Our experiments show that ToPs(LR, DT, NB) is not better in terms of the mean performance metrics than ToPs(LR), the best instance of ToPs with a single base predictor. If we look carefully at figure 20 we can see that in most of the cases ToPs(LR, DT, NB) performs equal than ToPs(LR). In *bank_marketing.csv* ToPs(LR, DT, NB) is slightly better than ToPs(LR), because it takes advantage of having a DT, but in other cases like *kr.vs.kp.arff* ToPs(LR, DT, NB) is worse than ToPs(LR). One reason that could explain this behaviour is the greedy approach ToPs uses. Taking the best split (locally) in a node does not always produce the optimal tree. A similar phenomena happens with the traditional Decision Tree.

predictor	accuracy	auc	precision	recall	time
ToPs(LR)	0.894	0.963	0.892	0.894	1146.931
ToPs(DT)	0.872	0.931	0.872	0.872	574.418
ToPs(NB)	0.838	0.941	0.856	0.838	244.042
ToPs(LR, DT, NB)	0.895	0.951	0.909	0.911	1348.060

Table 5: Mean of the performance metrics for ToPs(LR), ToPs(DT), ToPs(NB) and ToPs(LR, DT, NB)

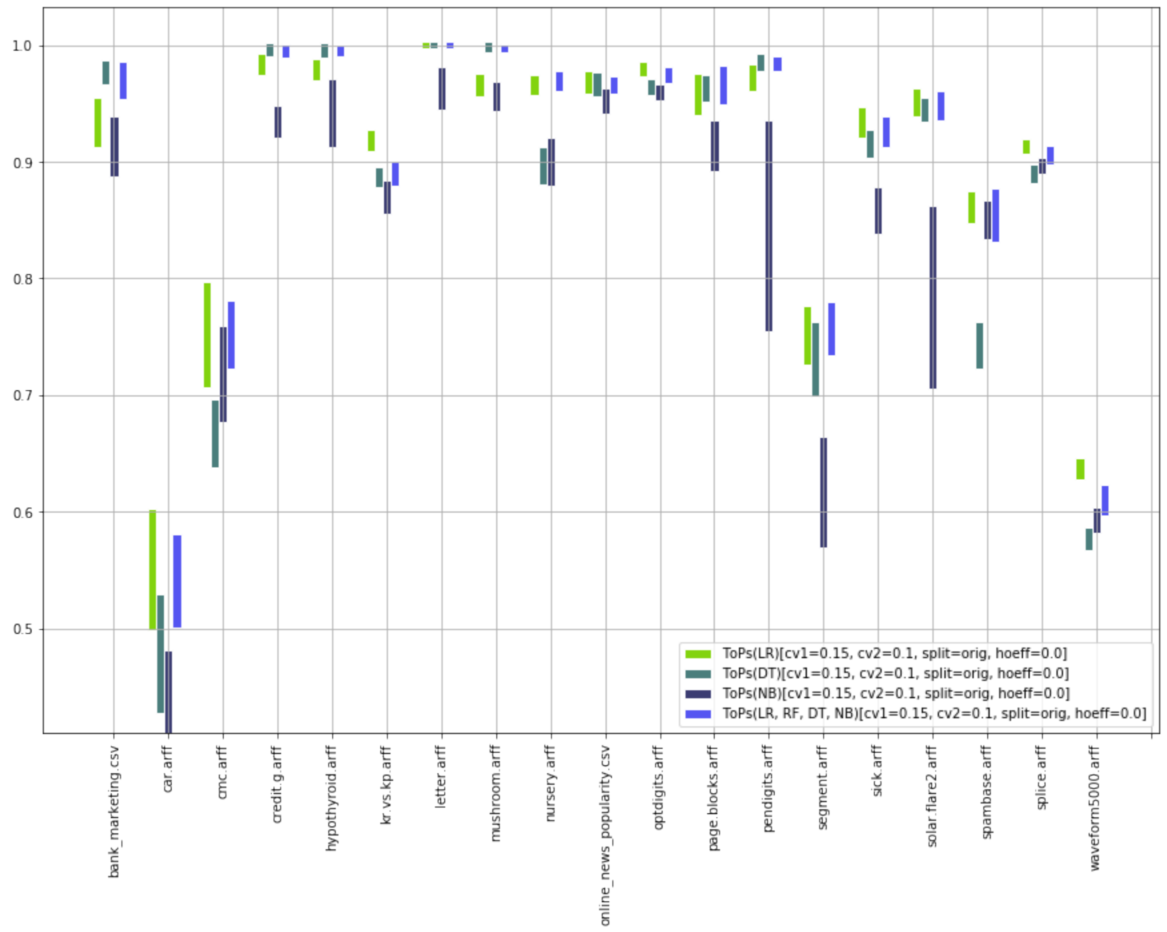


Figure 20: Accuracy for each dataset for ToPs(LR), ToPs(DT), ToPs(NB) and ToPs(LR, DT, NB)

7.2. ToPs $pr > 80\%$

The experiments in this section show how ToPs behaves, using a control mechanism over its splits, vs the "original" version of ToPs. This control mechanism consists of accepting a split only if the probability that the two children are better than their parent node is greater than 80%.

The results are very similar for all the instances of ToPs tested, so we will analyze them here. The use of $pr > 80\%$ does not improve ToPs from the point of view of the accuracy, but it may decrease it a little bit. However, having an explicit control of the growth of the tree helps in reducing the execution time. This is because we avoid continue splitting nodes that we think will not improve the overall performance. Probably, there exists a lower value of this probability that brings the same performance metrics than the original ToPs but maintains a decrease in the execution time.

What we did not expect was to see a similar variability in the accuracy of each dataset. We thought that a full grown tree, because of the fact that in the deep nodes we have less validation instances, would have some variance. If we controlled the depth of the tree we thought the tree would have less variance. However, our results do not support this theory.

7.2.1. ToPs(LR) vs ToPs_pr_80%(LR)

predictor	accuracy	auc	precision	recall	time
ToPs(LR)	0.894	0.963	0.892	0.894	1146.931
ToPs_pr_80%(LR)	0.885	0.961	0.883	0.885	560.413

Table 6: Mean of the performance metrics for ToPs(LR) vs ToPs_pr_80%(LR)

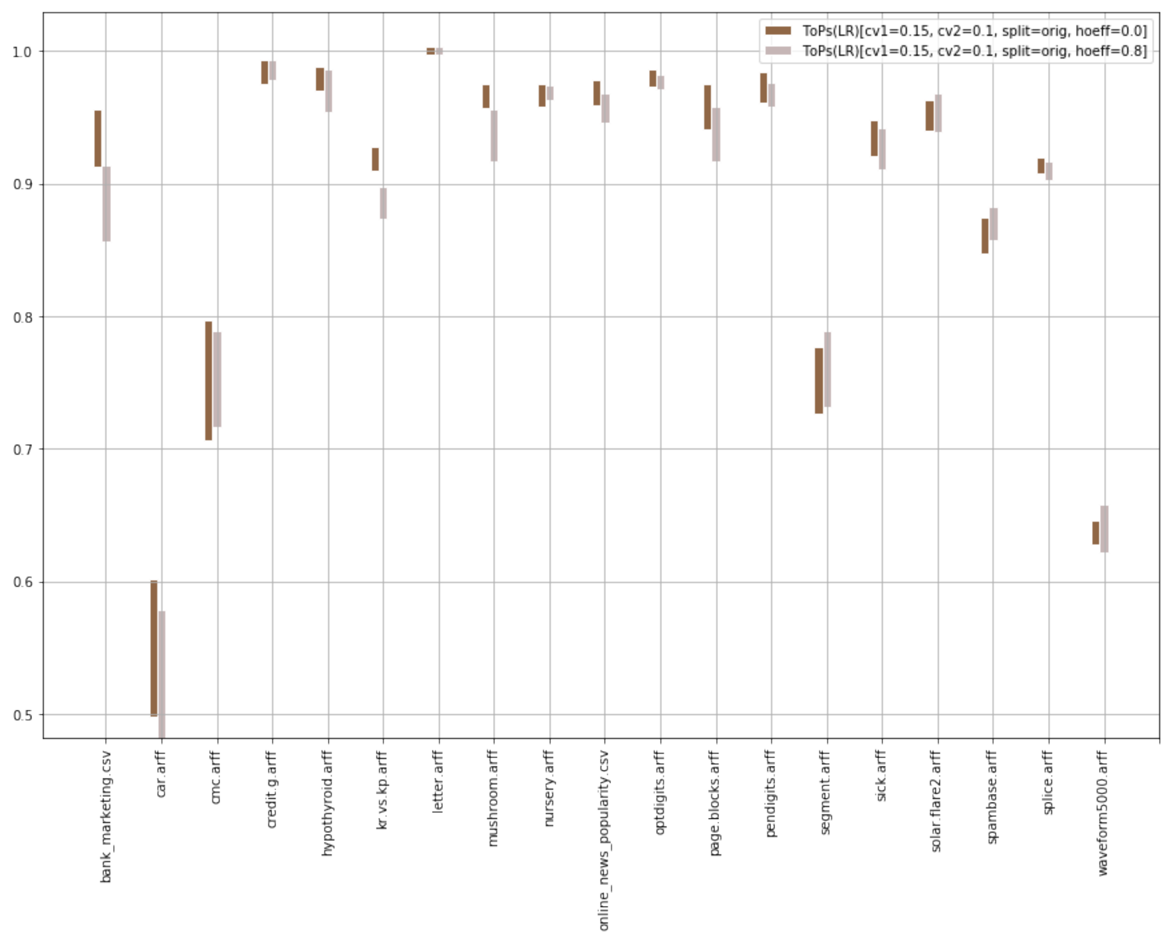


Figure 21: Accuracy for each dataset for ToPs(LR) vs ToPs_pr_80%(LR)

7.2.2. ToPs(DT) vs ToPs_pr_80%(DT)

predictor	accuracy	auc	precision	recall	time
ToPs(DT)	0.872	0.931	0.872	0.872	574.418
ToPs_pr_80%(DT)	0.871	0.922	0.872	0.871	250.162

Table 7: Mean of the performance metrics for ToPs(DT) vs ToPs_pr_80%(DT)

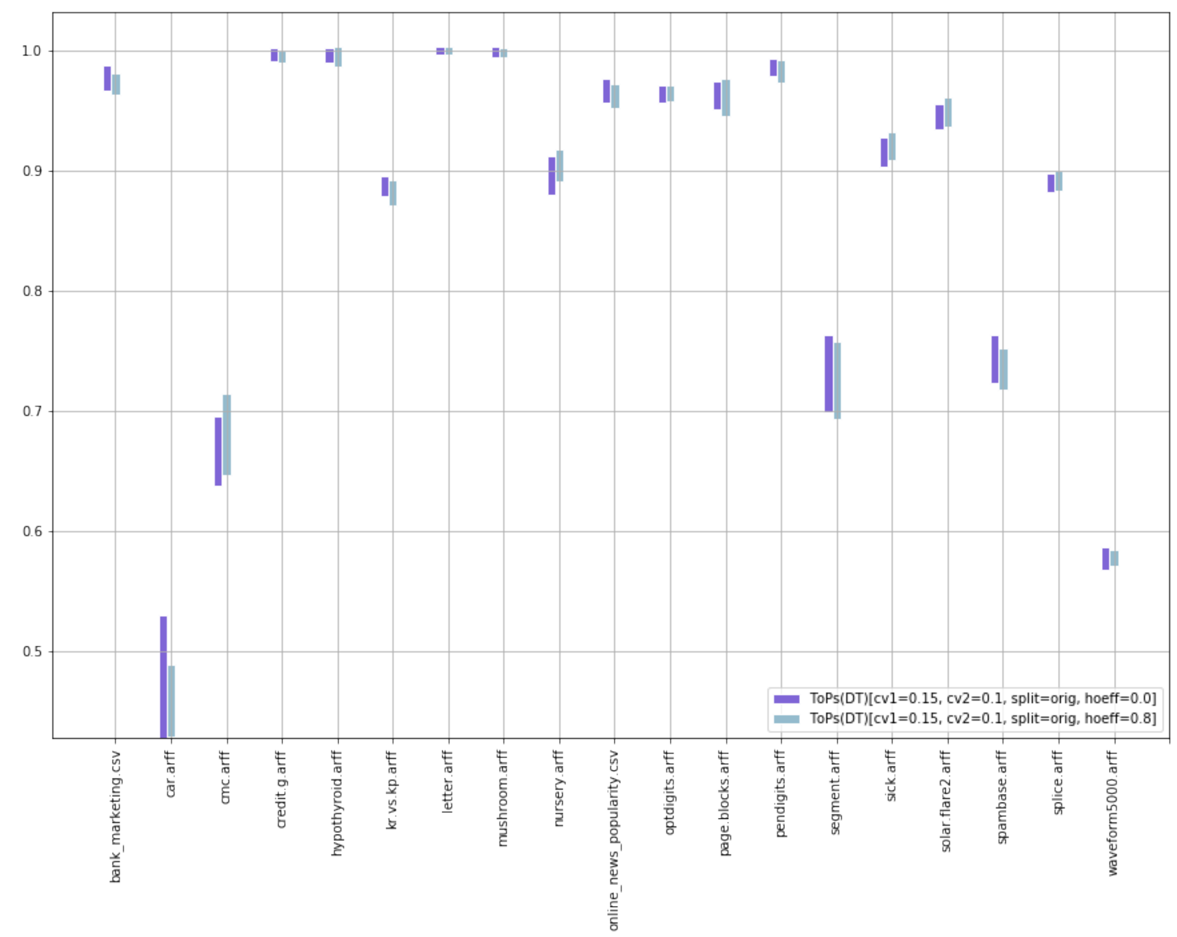


Figure 22: Accuracy for each dataset for ToPs(DT) vs ToPs_pr_80%(DT)

7.2.3. ToPs(NB) vs ToPs_pr_80%(NB)

predictor	accuracy	auc	precision	recall	time
ToPs(NB)	0.838	0.941	0.856	0.838	244.042
ToPs_pr_80%(NB)	0.817	0.938	0.843	0.817	192.786

Table 8: Mean of the performance metrics for ToPs(NB) vs ToPs_pr_80%(NB)

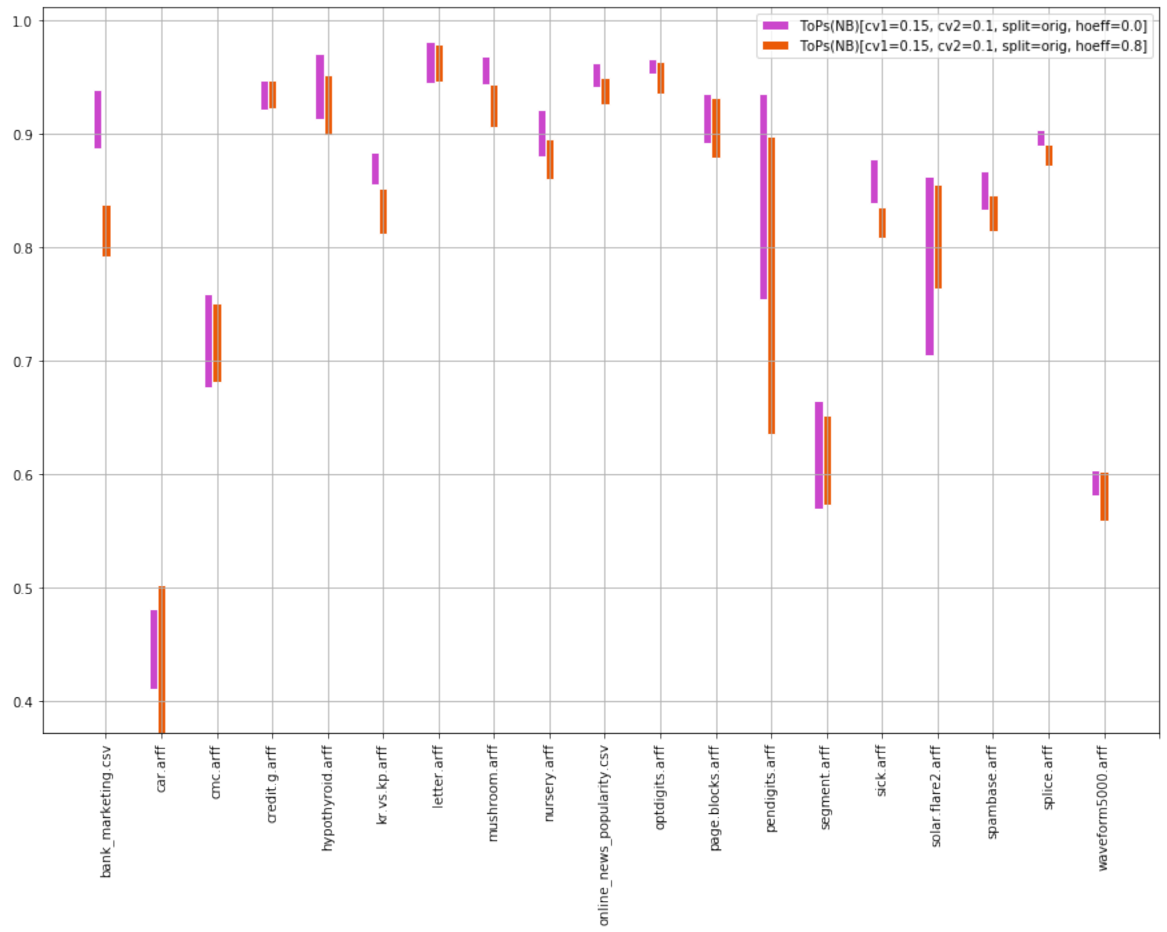


Figure 23: Accuracy for each dataset for ToPs(DT) vs ToPs_pr_80%(DT)

7.2.4. ToPs(LR, DT, NB) vs ToPs_pr_80%(LR, DT, NB)

predictor	accuracy	auc	precision	recall	time
ToPs(LR, RF, DT, NB)	0.895	0.951	0.909	0.911	1348.060
ToPs_pr_80%(LR, DT, NB)	0.896	0.949	0.895	0.896	784.855

Table 9: Mean of the performance metrics for ToPs(LR, DT, NB) vs ToPs_pr_80%(LR, DT, NB)

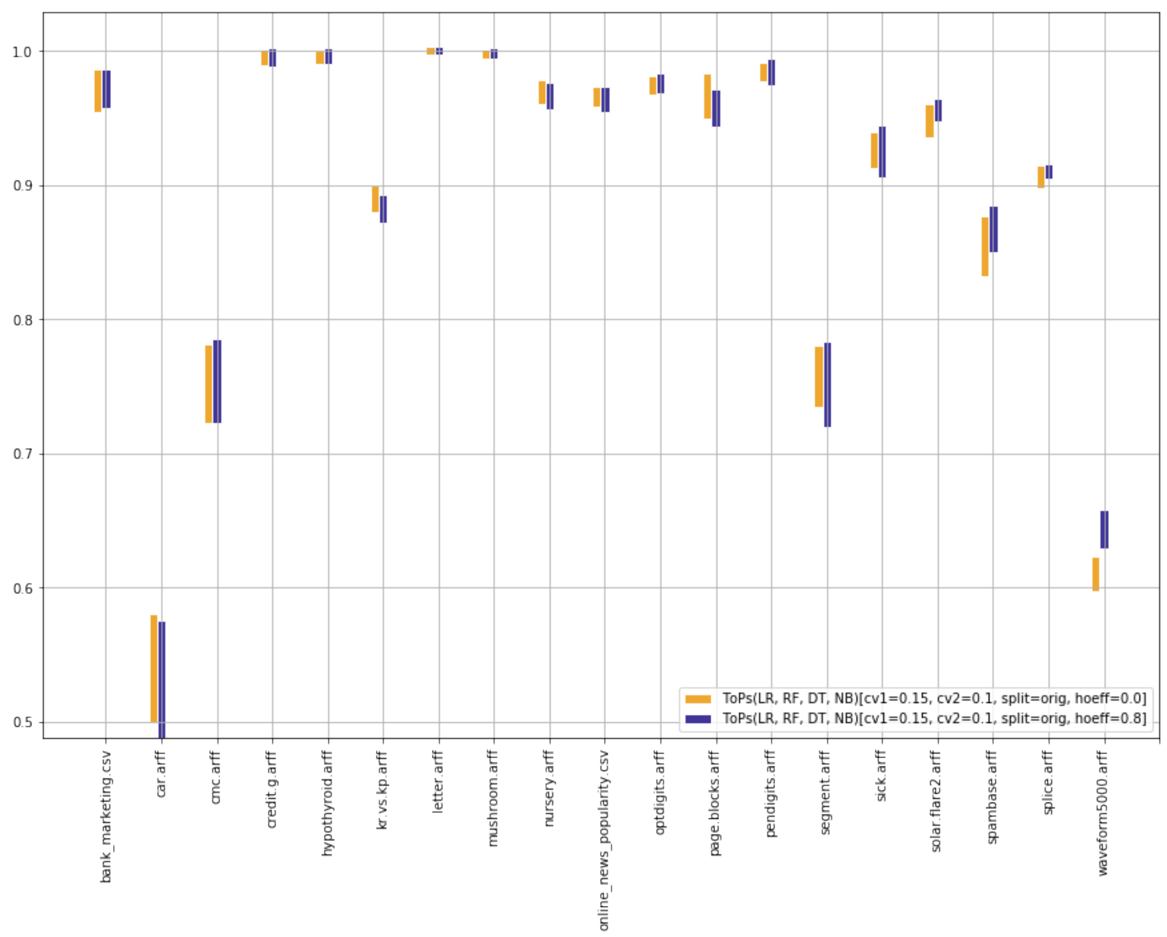


Figure 24: Accuracy for each dataset for ToPs(LR, DT, NB) vs ToPs_pr_80%(LR, DT, NB)

7.3. ToPs split gini impurity

In these experiments we wanted to see the effect of choosing only one threshold per variable but applying some kind of heuristic. We decided to use the reduction of gini impurity as our heuristic. The results are very consistent accross the different instances of ToPs we used, so we will analyze them here.

In this case, we can clearly say that we have improved our first implementation, based on the original paper. The performance metrics are not worse, but the execution time is dramatically reduced, even more than using ToPs_pr_80%. The reason why it is faster is different from the one explained in section 7.2. Here, we don't have any specific control over the growth of the tree, but we avoid some calculations by only trying one split per variable, instead of 9. The intelligence that the gini score gives to the algorithm avoids the brute force approach of the original version of ToPs.

7.3.1. ToPs(LR) vs ToPs_gini(LR)

predictor	accuracy	auc	precision	recall	time
ToPs(LR)	0.894	0.963	0.892	0.894	1146.931
ToPs_gini(LR)	0.891	0.964	0.889	0.891	384.857

Table 10: Mean of the performance metrics for ToPs(LR) vs ToPs_gini(LR)

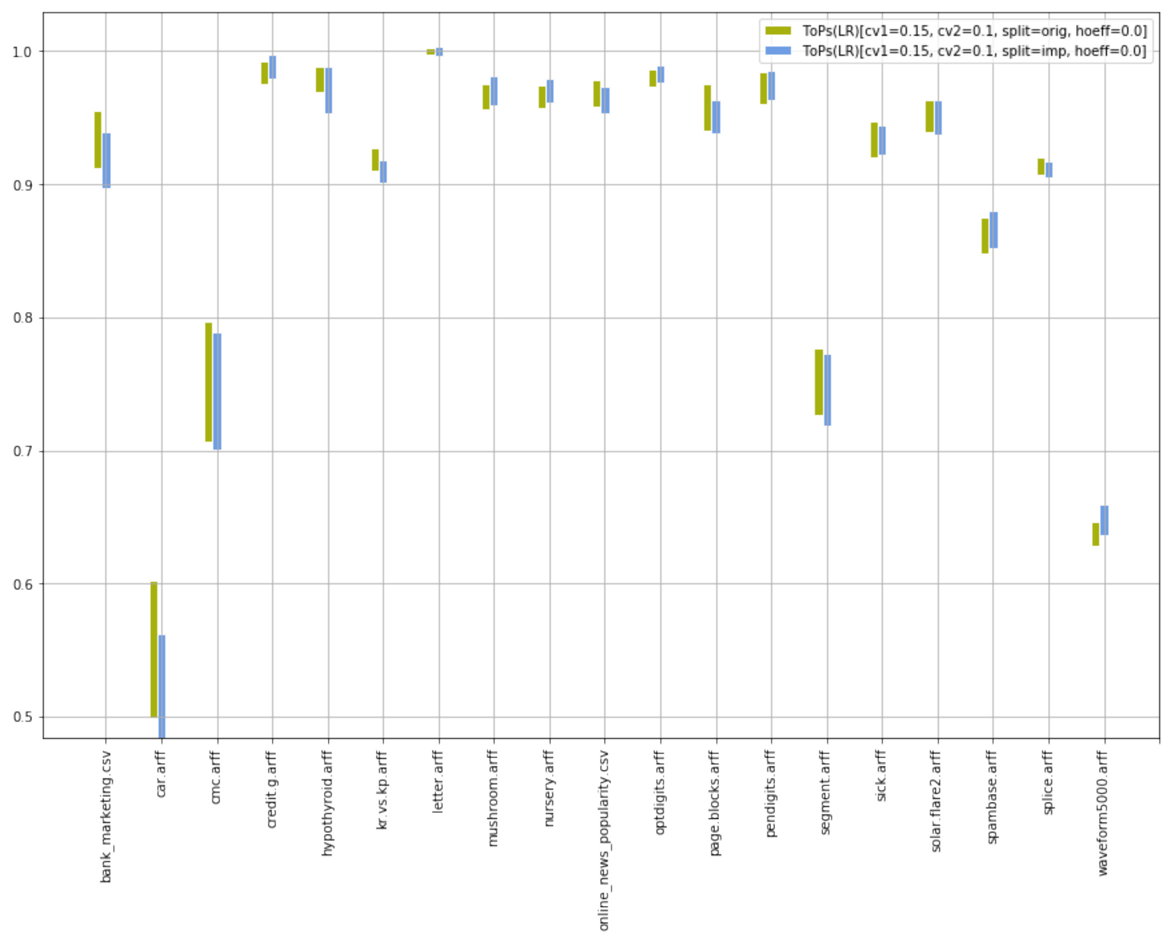


Figure 25: Accuracy for each dataset for ToPs(LR) vs ToPs_gini(LR)

7.3.2. ToPs(DT) vs ToPs_gini(DT)

predictor	accuracy	auc	precision	recall	time
ToPs(DT)	0.872	0.931	0.872	0.872	574.418
ToPs_gini(DT)	0.872	0.925	0.872	0.872	444.889

Table 11: Mean of the performance metrics for ToPs(DT) vs ToPs_gini(DT)

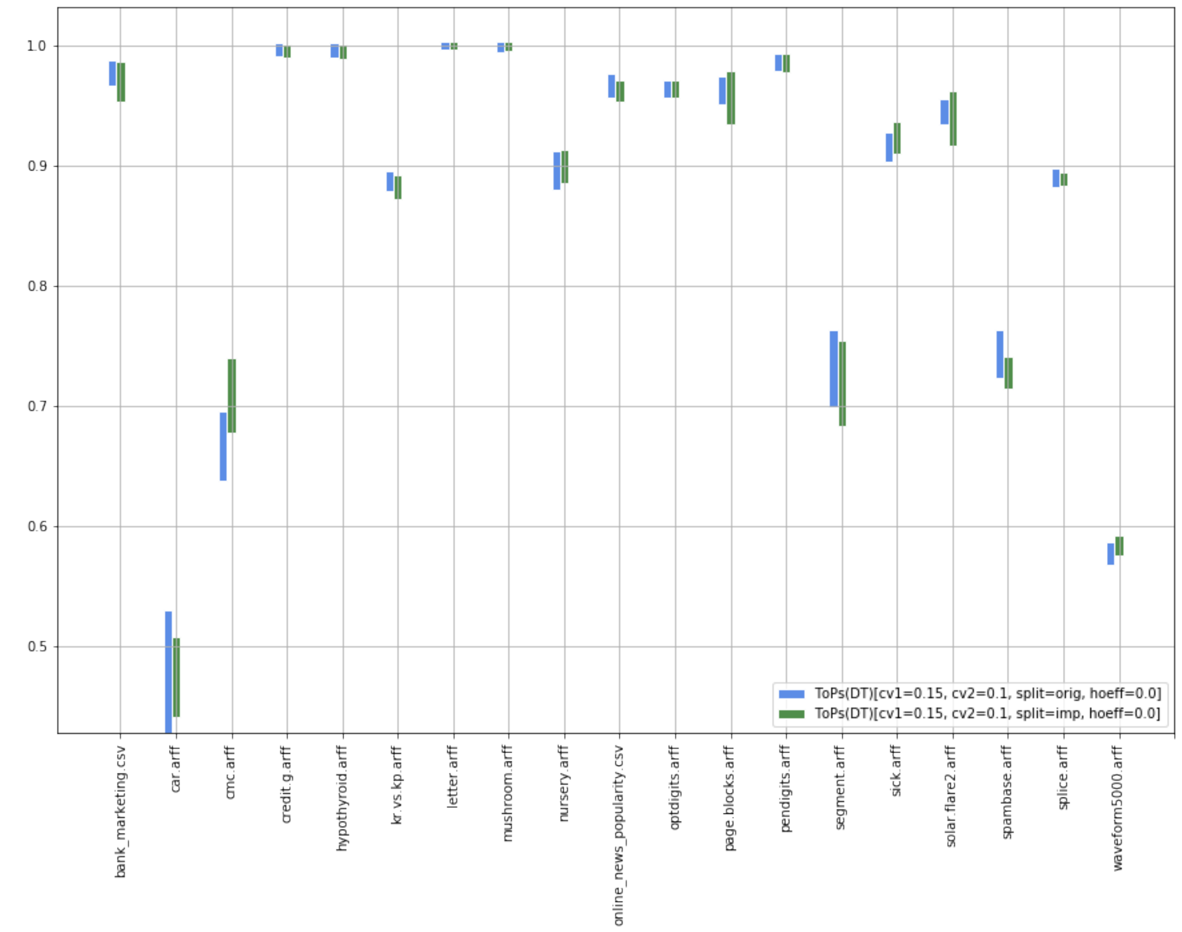


Figure 26: Accuracy for each dataset for ToPs(DT) vs ToPs_gini(DT)

7.3.3. ToPs(NB) vs ToPs_gini(NB)

	accuracy	auc	precision	recall	time
ToPs(NB)	0.838	0.941	0.856	0.838	244.042
ToPs_gini(NB)	0.839	0.942	0.855	0.839	225.992

Table 12: Mean of the performance metrics for ToPs(NB) vs ToPs_gini(NB)

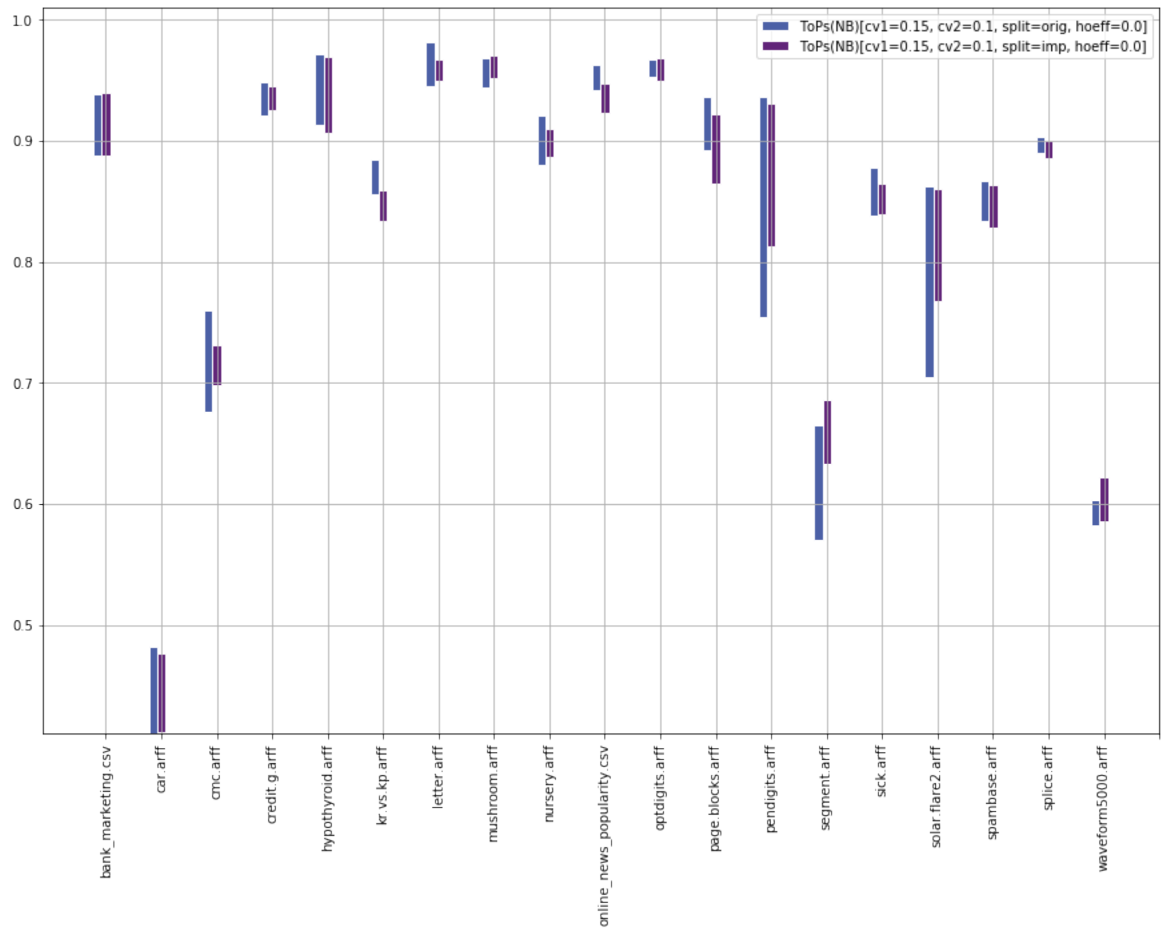


Figure 27: Accuracy for each dataset for ToPs(NB) vs ToPs_gini(NB)

7.3.4. ToPs(LR, DT, NB) vs ToPs_gini(LR, DT, NB)

predictor	accuracy	auc	precision	recall	time
ToPs(LR, RF, DT, NB)	0.895	0.951	0.909	0.911	1348.060
ToPs_gini(LR, DT, NB)	0.893	0.949	0.890	0.891	647.614

Table 13: Mean of the performance metrics for ToPs(LR, DT, NB) vs ToPs_gini(LR, DT, NB)

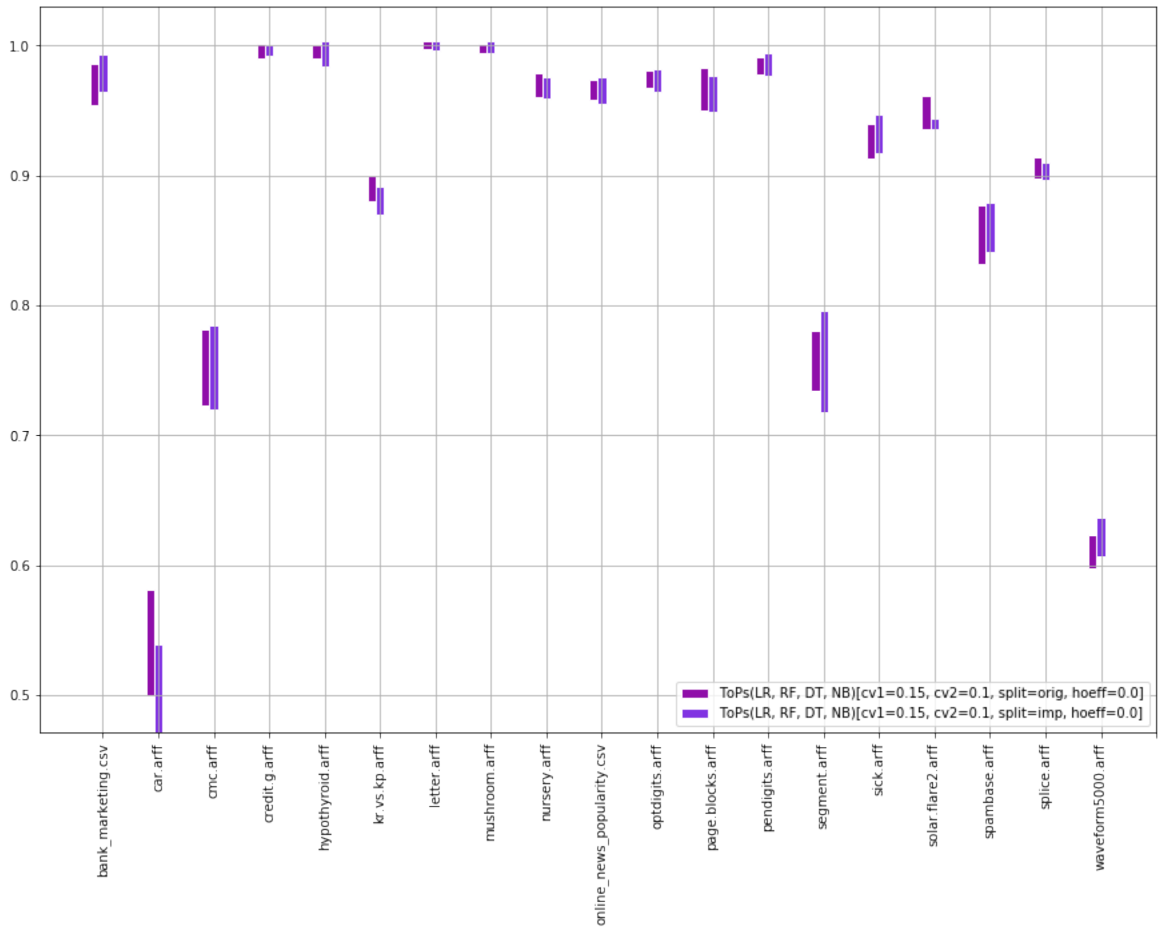


Figure 28: Accuracy for each dataset for ToPs(LR, DT, NB) vs ToPs_gini(LR, DT, NB)

7.4. ToPs vs Random Forest

It is easy for ToPs to outperform a base learner as Logistic Regression or Naive Bayes, because it is an ensemble method and it takes advantage of having a committee of predictors instead of only one. For this reason, the question that a data scientist may have is: should we use ToPs instead of another well established ensemble method? To answer this question, we run all the datasets with an instance of Random Forest, which had 500 trees, and we compared its performance with the best instance of ToPs that we tested, which is ToPs_gini(LR).

The results of our experiments prove that ToPs is not yet prepared to become a real competitor of Random Forest. The following table show that the mean value of the goodness metrics is slightly better for Random Forest. If we look the accuracy of each dataset, one by one, we see that there is no case where ToPs is significantly better than Random Forests. Moreover, the execution time of ToPs is one order of magnitude higher than Random Forest. The consequence of all these observations is no data scientist would use ToPs instead of another ensemble method as Random Forest in any of the evaluated datasets.

predictor	accuracy	auc	precision	recall	time
RF	0.903	0.963	0.902	0.903	20.162
ToPs_gini(LR)[cv1=0.15, cv2=0.1, split=imp, hoeff=0.0]	0.891	0.964	0.889	0.891	384.857

Table 14: Mean of the performance metrics for RF vs ToPs_gini(LR)

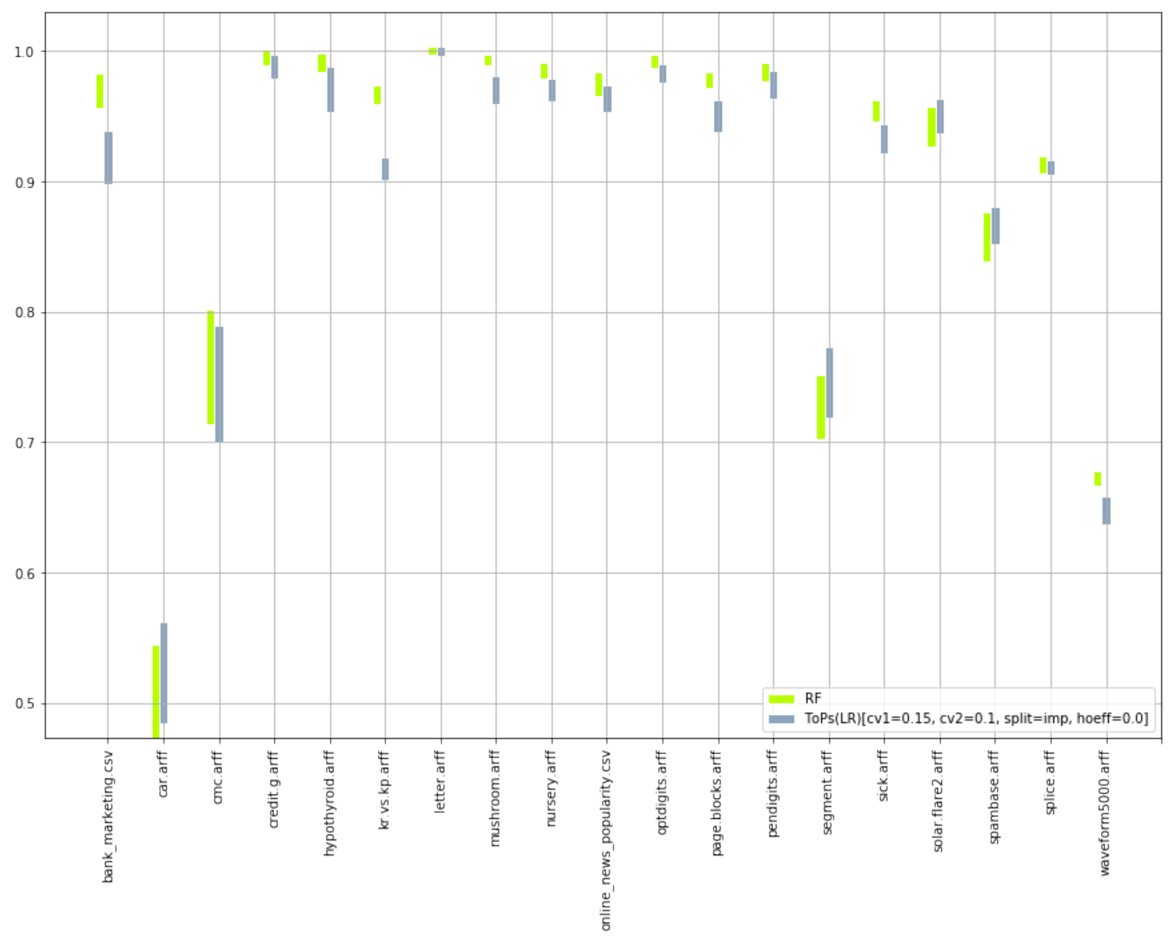


Figure 29: Accuracy for each dataset for RF vs ToPs_gini(LR)

7.5. Detailed view of the execution time

For this experiment we used the dataset *bank_marketing* and we splitted it into a training set (70%) and a validation set (30%). The ToPs instance we used was ToPs(LR) with the same parameters used in 7.1, the ones that are more similar to the original description. Previously, we instrumented the code with several timers, to know exactly how much time each step consumes. The objective was to discover the bottlenecks of the algorithm.

What we can see is that the process of training the base predictors is the top offender. The table 15 contains the exact number for the steps represented in figure 30 and it shows this step consumes 85.60% of the total time spent.

From this experiment we learned that the main point to focus for improving the execution time is the training process for the base predictors. That means it is better to use simple and fast base predictors and we should try to reduce the number of tentative splits that we do at each node. The positive effect of reducing the number of tentative splits has already been seen in section 7.3. Because the gini impurity split only tries one threshold for each variable instead of 9, it reduces the number of tentative splits and, thus, the number of times a base predictor is trained.

Task	Time (s)	Percentage
Train ToPs	147.4	98.27%
Preprocess initial data	0.3	0.20%
Split (grow tree)	147.1	98.07%
Generate thresholds	0.5	0.33%
Pass data to tentative children	3.7	2.47%
Find best predictor for children	139.4	92.93%
Train base predictors	128.4	85.60%
Evaluate goodness (V1)	7.8	5.20%
Aggregate	2.1	1.40%
Predict	2.5	1.67%

Table 15: Time consumed for the different steps of out implementation of ToPs, using the dataset *bank_marketing.csv*

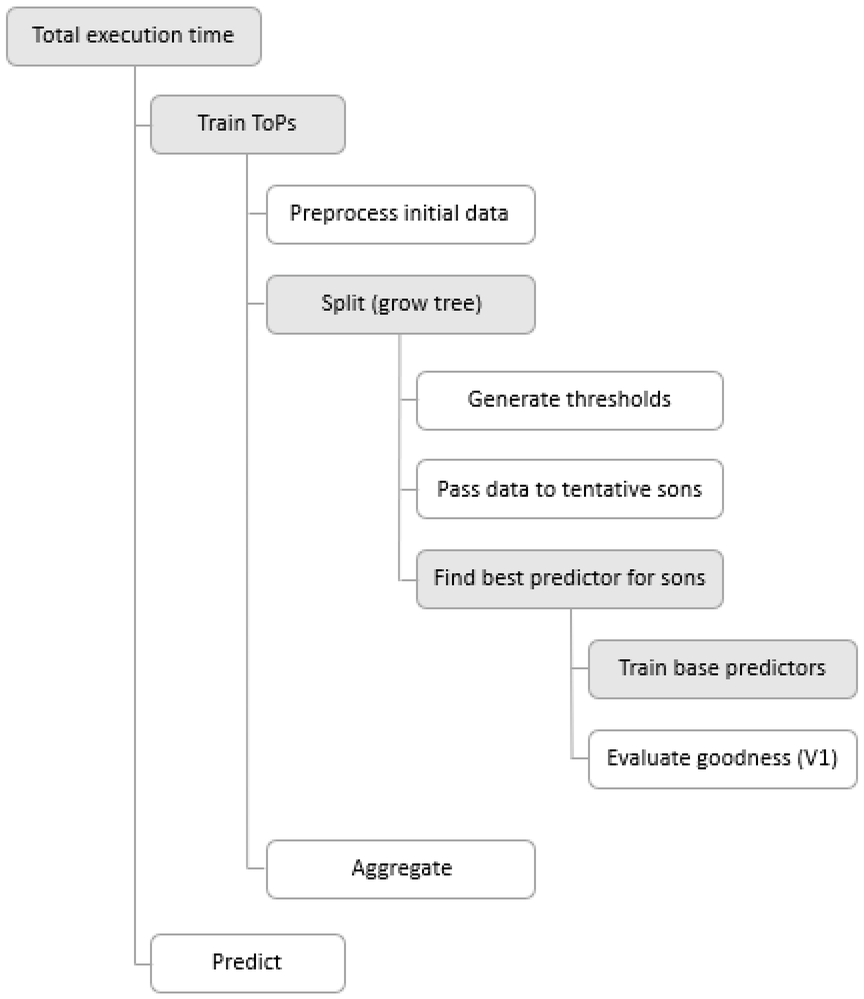


Figure 30: Diagram of the steps differents steps of our implementation of ToPs

8. Conclusions

In this thesis we have shown the theoretical foundations of ToPs (a tree of predictors), we have created a public implementation of it and we have evaluated its performance.

Our implementation improves some aspects of the original paper:

- We can set a limit to the minimum number of validation instances a node can have.
- We can use CV instead of the V1 set.
- We can handle directly categorical variables.
- We can use the reduction of the gini impurity as a method to find a threshold in a numerical variable.
- We have a way to know whether a split improves significantly the goodness function with respect to the parent node (Hoeffding/Chernoff).

We have also given a better bound of the time complexity for the whole split process.

In our experiments we have seen the pros and the cons of our implementation of ToPs. ToPs has been proven to be at least better than its base predictors, thanks to the ability to train predictors in different regions of the feature space and the ability to create ensembles using these predictors. Logistic Regression and, especially, Gaussian Naive Bayes are the predictors that show a better improve when they are combined with ToPs. However, the superiority of ToPs versus its base predictors does not come for free, but at the expense of the execution time. Using a probability of 80% when deciding if the children are better than the parent node in a tentative split, improves the execution time, since the nodes are less expanded, but it also decreases the accuracy. The use of the gini impurity for finding a threshold improves further the execution time, because it only tries one threshold per variable instead of 9, and it maintains the accuracy.

In the experiments we have also tested ToPs versus Random Forest, a widely used ensemble method. The results were not favorable to ToPs. Random Forest was equal or better in terms of accuracy in all the datasets with a mean execution time 1 order of magnitude less than ToPs.

The last conclusion of this work is that ToPs is not yet a top competitor among ML algorithms, at least the implementation we created. There are faster and more accurate algorithms, like Random Forest. For this reason, in the next section, we suggest several improvements that could give ToPs the potential to become a strong state-of-the-art predictor.

9. Future work

9.1. Improve the execution time

As we have seen in our experiments and remarked in the conclusions, the major concern of ToPs is its execution time. The algorithm is complex an it requires datasets with a minimum number of instances around 1000.

However, the main drawback regards not the number of instances but the number of variables. In section 4.1.1 we have seen the split process and how the number of variables impacts the cost of this process. For that reason, the main focus of our future investigation when trying to improve the execution time is about reducing the number of candidate variables in a split.

9.1.1. Avoid correlated variables

A variable in a dataset gives a way to sort it. It imposes an order between the different instances. That means some of them will be close in that new order while other ones will be far away.

Two high correlated variables will give similar orders in the dataset. The consequence is that instances that are close using the order that gives the first variable will probably be close using the order that gives the second one. The same happens for the instances that are far away. As a result, the threshold we can use to split the dataset using the first variable will probably give a very similar split using the second variable. For this reason, the first subset created using the first variable will contain almost the same instances as the first subset created using the second variable, and the same for the second subsets.

The success of ToPs is finding a good way to divide the feature space. Its approach to this target is by trying different ways to divide it. Notice that high correlated variables, since they give similar splits, they do not help to explore new promising ways to divide the feature space. Their splits are redundant. For this reason is not worth to do a split with a variable if we have already tried another variable highly correlated with the first one.

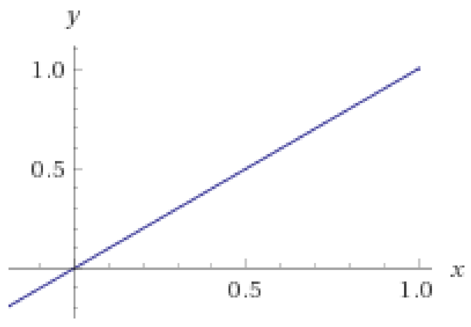
Until now, we have talked about correlated variables, but it is more precise to talk about the order that they impose on the variables, because what really matters in a split is how the variable sorts the dataset. To illustrate the slight difference between correlated variables and similar orders we can take a look at the following functions and think they are variable generators assuming $x \in \mathbb{R}^+$.

$$f_1(x) = x \tag{35}$$

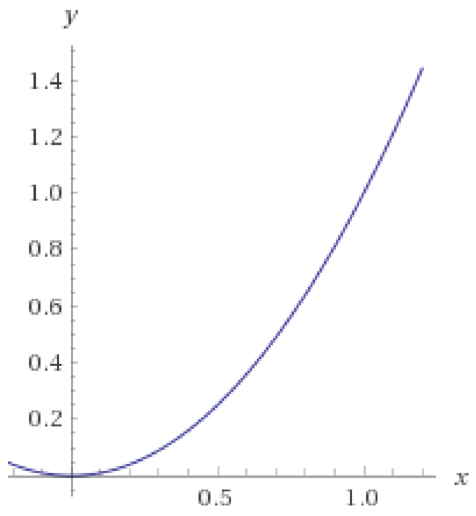
$$f_2(x) = x \tag{36}$$

$$f_2(x) = x^2 \tag{37}$$

The correlation of f_1 and f_2 is 1 and their order is the same. However, the correlation between f_1 and f_3 is slightly less than one but their order is exactly the same.



(a)



(b)

Figure 31: (a) Representation of f_1 or f_2. (b) Representation of f_3. The two plots don't have a perfect correlation but a perfect order.

The summary of this subsection is we should select a subset of variables for the split in which no pair of variables has a very similar order. The variables that are not in this subset can be discarded because they will not add new ways to split the feature space.

9.1.2. Lower the dimension of the feature space

A high dimensional feature space is completely unmanageable for ToPs. However, there exists techniques to lower the feature space. What they have in common is that they try to represent the maximum amount of information from the original space but using a smaller number of variables.

The most famous tool to do so is PCA. PCA not only reduces the dimensionality of the original space but also gives uncorrelated variables, because the components it extracts are orthogonal between them. That means these new components (variables), apart from being less numerous than the original ones, are not redundant (not correlated).

9.1.3. Select a promising subset of variables

ToPs could have some heuristic to select automatically a subset of variables which seems to be promising for splitting the dataset. This is also a feature selection technique. However, in that case, we are not caring about correlations between the variables. What we actually search is variables that may give splits with a better performance. Of course, this process could be applied after selecting only a subset of uncorrelated variables, following the idea explained in 9.1.1.

We suggest taking the top-k variables with highest reduction in gini impurity (or entropy). Our intuition is that the variables with highest ability to reduce the gini impurity will add more information to ToPs and, as a consequence, they seem to be more useful when trying new splits.

9.1.4. Select a random subset of variables

Instead of using any heuristic to select a subset of variables for a split we could select them randomly. In principle, it seems worse than the previous suggested technique, but it has the advantage of a negligible computational time.

From a theoretical point of view, a random subset of variables may be enough to obtain an acceptable split. It is true that using all the variables the split will be better (or at least the same) from the point of view of the goodness function estimated using the validation set 1. However, we should think if this increase pays off, taking into account the computational resources we have to spend (remember that trying lots of eventual splits is very expensive). Moreover, sometimes the increase in the value of the goodness function is due to an overfit over the validation set 1 and it does not represent a real increase of its generalization value.

9.1.5. Random Forest of ToPs

ToPs is a tree and, in the same way Decision Trees can create a Random Forest, ToPs could create a Random Forest of ToPs. The idea is exactly the same applied to convert a Decision Tree into a Random Forest.

Bagging would be used to select the instances that would be used to train an instance of ToPs. Then, for the splitting process, instead of trying all the variables, we would select at random only a small subset of them. Finally, a prediction would be done taking the probabilities that outputs each instance of ToPs and giving to them the same weight $(1/n)$.

Using this idea we also follow the path to reduce the number of variables used at each split. It is true that we have to create more ToPs instances, but the time to create each one is reduced considerably. Apart from the benefits in terms of the execution time, we could also extract new meaningful information about the dataset. For example, we could try to apply an adapted version of the feature importance metric that is extracted from Random Forest.

9.1.6. Parallelize the current implementation

Nowadays, all the processors have several cores. However, our implementation is sequential. That means, when we train an instance of ToPs, we are not using most of the computational power of a PC because only a core is used.

The success of the future parallel ToPs implementation will depend on the ability of the programmer to define a good granularity and distribute evenly the work among the different cores.

9.2. Improve accuracy

The performance in terms of the accuracy is always a factor that pays off to improve in a predictor. We have seen that ToPs gives a better performance that its base predictors, but it still fails to outperform other ensemble methods as Random Forest.

9.2.1. Find uncorrelated predictors

One factor that affects considerably the performance of an ensemble is its ability to generate or select a subset of predictors which are not correlated between them.

Correlated predictors do not make a good ensemble. Imagine we have a single predictor and we want to improve its accuracy. Someone comes and tells us to make several copies of this predictor and create an ensemble. We should know this technique is useless, since all the predictors in the ensemble will predict the same. The success of an ensemble is due to having different uncorrelated predictors that can cover between them their weak points.

Notice that ToPs ignores completely whether a new predictor is correlated or not with its ancestors. It only looks at its performance measured in terms of the goodness function. An ensemble is team work and its success depends on how well their members complement each other. For that reason, we think a new predictor should be judged depending on how well it integrates itself in its corresponding ensemble.

Our proposal is the following. For each new trained predictor we apply the aggregation function over itself and all its ancestor predictors. Then, we evaluate the new tentative ensemble using our goodness function and the validation set 2. We finally take the predictor that adds more performance to the ensemble. Notice that, with this technique, we no longer need the validation set 1, so its instances can be used for the training set or the validation set 2. Increasing the size of these two sets is always good, as pointed in section 4.3. The only problem is, with this technique, we would grow too much the tree, because adding a new predictor in an ensemble always gives an equal or better result (with respect to the validation set 2) than the previous version of the ensemble. However, this does not mean the ensemble generalizes better. To limit the growth of the tree we could use L1 regularization.

9.2.2. Reuse the discarded splits

As pointed in section 4.3 ToPs spends a lot of computational resources training predictors for nodes that are finally discarded. The result is that this computational effort is wasted. It is true that it gives information about whether this split is worth or not, but in the negative case, when we have this information, we never reuse the discarded nodes.

However, we could preserve a selection of these discarded nodes to increase the amount of predictors in our ensemble. The following image illustrates the idea we want to transmit.

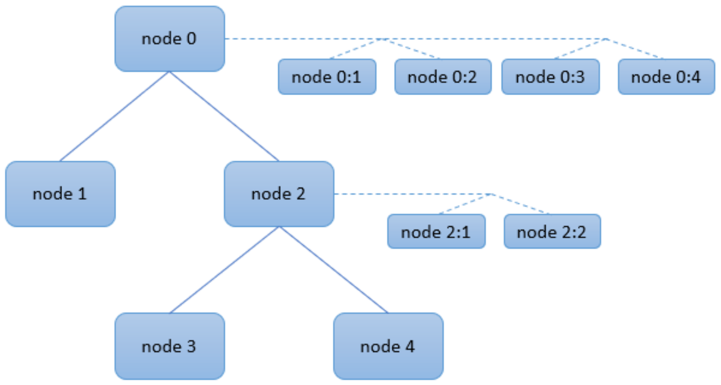


Figure 32: Structure of the tree with the extra children

The main tree structure remains the same. What we add are some extra children to each non-terminal node. These extra children are a selection of the tentative children created for a node during its splitting process. All the extra children are no further splitted, so they remain as leaves.

For making a prediction, we use the ensemble generated by all the predictors found along the path from the root to the corresponding leaf plus all the corresponding predictors from the corresponding extra children attached to the non-terminal nodes.

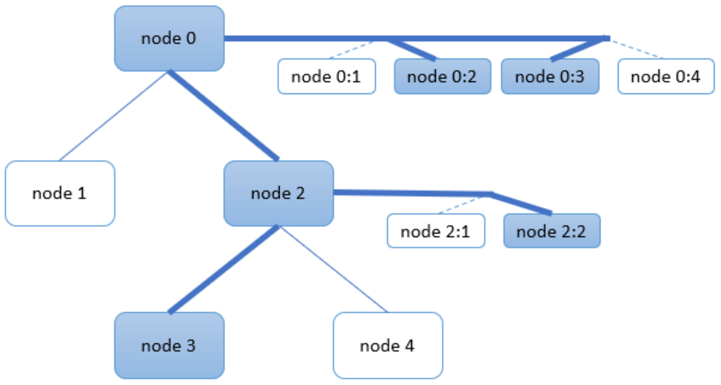


Figure 33: In that case, the prediction is done using the predictors from node0, node0:2, node0:3, node2, node2:2 and node 3

9.3. ToPs regressor

In the original paper and in the current one we have only talked about ToPs as a classifier. The prediction problems that it can handle in the current implementation are classification ones, those where we have to decide in which class an instance belongs. Regression problems, the ones where the target variable is a quantity, are not currently supported. However, once we have a clear understanding of the theory behind ToPs and a skeleton of its implementation is quite straightforward to adapt it for regression problems.

From a theoretical point of view there are only two changes needed. The first one is to use base classifiers that can handle regression problems. The second one is having a goodness function that can work with real and predicted quantities instead of real labels and predicted probabilities. For example, instead of using the logarithm of the probability for the true class as the goodness function we could use the squared difference between the real and the predicted values.

Notice that all the other steps involved in the training process remain the same. For the splitting process we have to go through all the variables, try different thresholds, create tentative children and associate predictors to them. At the point where we have to compare the performance of the different base regression predictors is where we have a different goodness function, as explained in the previous paragraph. We even can use the Hoeffding/Chernoff criteria if our goodness function gives bounded values. For the aggregation of the predictors we need another time a goodness function for regression. The properties that should have this goodness function (smooth, non local optima...) are the same ones that are required for the ToPs-classification. The explanation of these properties and why the goodness function used for the aggregation process should follow them are in section 4.1.2.

The overall tree is again a tree-like structure, with a predictor associated to each node and with a set of weights associated to each path from the root to a leaf that define an ensemble.

9.4. Ability to handle missing values

Collecting real data is not an easy task and real datasets are not always full and clean. Sometimes we find instances with missing values for some of their variables. To solve the problem that represents having a dataset with missing values there are some techniques to fill these values. Examples of them are using the mean of the instances that have a defined value for that variable or using chained equations [18]. Another solution is to do nothing with the dataset and simply use a predictive algorithm that can swallow datasets with missing values. Decision trees and ensemble methods made of them, as Random Forest, are examples of such algorithms.

ToPs could be transformed to handle missing values. There are only two factors that have to be modified. The first one is forcing the use of base predictors that can handle missing values by their own. The second one is being able to split a variable even when there are missing values. From these two problems, the second is the one that presents the biggest challenge. However, can borrow the solution decision trees and random forest use to handle missing values in their splits [7] [11].

9.5. Support for sparse data

The instances of some datasets contain only a small fraction of non-zero values for their variables. These datasets are called sparse. Sparse datasets are normally stored using an index and a value for each non-zero variable. The variables that do not have and index and value associated to them are assumed to be 0. This way to encode a sparse dataset helps to reduce the memory usage. The problem is it is more difficult to manipulate the data. Algorithms that assume the dataset is encoded as a full matrix will no longer work with sparse data. Having the ability to handle sparse datasets would give to ToPs a major versatility and usefulness, since sparse datasets are common in applications as recommendation systems [19].

References

[1] W. R. Z. Jinsung Yoon, M. van der Schaar, Tops: Ensemble learning with trees of predictors, IEEE Transactions on Signal Processing 66 (8) (2018) 2141—-2152.

[2] Wikipedia, No free lunch theorem (2019).
URL `https://en.wikipedia.org/wiki/No_free_lunch_theorem`

[3] Wikipedia, Hoeffding's inequality (2019).
URL `https://en.wikipedia.org/wiki/Hoeffding%27s_inequality`

[4] Wikipedia, Chernoff bound (2019).
URL `https://en.wikipedia.org/wiki/Chernoff_bound`

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, 3rd Edition, The MIT Press, 2009.

[6] M. J. Kearns, U. V. Vazirani, An Introduction to Computational Learning Theory, MIT Press, Cambridge, MA, USA, 1994.

[7] C. J. S. L. Breiman, J. Friedman, R. A. Olshen, Classification and regression trees, Taylor & Francis, 1984.

[8] R. Kohavi, Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid., in: Kdd, Vol. 96, Citeseer, 1996, pp. 202–207.

[9] L. Torgo, Functional models for regression tree leaves, in: ICML, Vol. 97, Citeseer, 1997, pp. 385–393.

[10] L. Breiman, Bagging predictors, Machine learning 24 (2) (1996) 123–140.

[11] L. Breiman, Random forests, Machine learning 45 (1) (2001) 5–32.

[12] Y. Freund, R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, Journal of computer and system sciences 55 (1) (1997) 119–139.

[13] P. Smyth, D. Wolpert, Linearly combining density estimators via stacking, Machine Learning 36 (1-2) (1999) 59–83.

[14] Wikipedia, Goodness of fit (2019).
URL `https://en.wikipedia.org/wiki/Goodness_of_fit`

[15] V. Dorogush, A. V.; Ershov, A. Gulin, Catboost: gradient boosting with categorical features support, in: Workshop on ML Systems at NIPS, 2017.

[16] M. Lichman, Uci machine learning repository (2018).
URL `https://www.cs.waikato.ac.nz/ml/weka/datasets.html`

[17] U. of Waikato, Weka (2013).
URL `http://archive.ics.uci.edu/ml`

[18] P. Royston, I. R. White, et al., Multiple imputation by chained equations (mice): implementation in $^{\circ}$stata, J Stat Softw 45 (4) (2011) 1–20.

[19] A. Popescul, D. M. Pennock, S. Lawrence, Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments, in: Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., 2001, pp. 437–444.