Machine learning course – final exercise

Shavit Talman 034234914 github link

1. Algorithm name

FastForest

2. Reference

Yates, Darren, and Md Zahidul Islam. "FastForest: Increasing Random Forest Processing Speed While Maintaining Accuracy." arXiv preprint arXiv:2004.02423 (2020).

3. Motivation for the algorithm

The algorithm aims to decrease the computation time of the Random Forest algorithm, while keeping its performance at the same level. Often, Random Forest achieves high-quality results. Since the authors' research domain is how to employ AI algorithms in mobile devices, they researched how to decrease the Random Forest runtime during training, to allow it to run in devices with limited resources.

In addition, in applications running in clouds that are being charged by the second (as in AWS), decreasing algorithms' computational time will lead to reduced expanses.

Finally, since the training of the trees can be done in parallel (except for sequential ensemble algorithms, as in the Boosting algorithm), the authors concluded that minimizing the training time per tree will be the best venue for decreasing the overall training time.

4. Short Description

4.1. Original Random Forest keynotes

The Random Forest algorithm (**RF**) is an ensemble learning method for classification and regression. It uses several decision trees and combines their results in order to predict its task. The trees are not pruned. In order to avoid overfitting it randomizes the data provided to each of its decision trees with:

1) Bagging – suppose the training samples size is *n*. RF selects a random sample with replacement of size *n* from the training set as an input for each tree in its ensemble
2) Subspacing – suppose the features number is F. RF selects a subset of *k* features, *k*< F, at each split. In scikit-learn the implementation is $k = \sqrt{F}$ or $k = \log_2 F$

4.2. Fast Forest alterations

FastForest (**FF**) uses 3 revised components within the original flow of the Random Forest algorithm, to decrease the training time:

1) Half-Subbagging – the algorithm randomly chooses half the samples for each tree in the ensemble, so the trees are trained with only $\frac{n}{2}$ of the samples, when the training samples size is *n*. as mentioned, Random Forests uses Bagging and selects *n* in random, with replacement
2) Logarithmic Split-Point Sampling – when testing a numerical feature for the best split value, this component examines only log_2 of the number of records in that node plus 1 (with non-missing value for that feature). It first sorts all possible values for the feature and then test them as split candidates in hops, so that no more than log_2 of the number of records in that node plus 1 are considered. In the original Random

Forest algorithm all unique values of the numerical features were considered as candidates for the split.
There is no change for categorical features.

3) Dynamic Restricted Subspacing – when the number of samples reaching a node is $^1/_8$ of the size of the original training set or less then the number of features selected at the split (Subspacing) increases, to allow selecting quality features. In Random Forest the Subspacing value is constant

5. Pseudo-Code

5.1. For training

FastForest – Building the ensemble (fit function)

Input: S – a labeled training set, $S := (x_1, y_1), \ldots, (x_n, y_n)$
T – number of classifiers to train, that is, trees to grow
F – number of features

For t=1 to T **Do**
// the Half-Subbagging
  $S`_t \leftarrow$ random sample of size $\frac{n}{2}$ from S, without replacement
  $M_t \leftarrow$ *buildDecisionTree*($S`_t$, M_t, F)
  FastForest ← *FastForest* + M_t
End For
Return *FastForest*

buildDecisionTree function

Input: $S`$ – a training set passed in the tree build recursion. its size is $|S`|$
M_t – a classifier that is trained (decision tree), passed recursively during the tree build
F – number of features

If $S`$ represents a leaf **Then**
  $M_t = M_t$ + createLeafWithPrediction()
  return M_t
End If
// the Dynamic Restricted Subspacing
If $|S`| < 0.125 \times n$ **Then**
  $k = \log_2\left(F \times \frac{n}{|S`|}\right) + 1$
Else
  $k = \log_2(F) + 1$
End If
$A \leftarrow$ random sample of size k from the F features
bestSplitInformationGain ← ∞, *bestFeature* ← null
For i = 1 to k **Do**
  splitCandidateInformationGain ← findSplit(A_i, $S`$)
  If *splitCandidateInformationGain* > *bestSplitInformationGain* **Then**
    bestSplitInformationGain ← *splitCandidateInformationGain*
    bestFeature ← A_i
  End If
End For
newS ← $S`$ remainder after split according to *bestFeature*
$M_t = M_t$ + buildDecisionTree(*newS*, M_t, F)
Return M_t

Figure 1 FastForest training stage pseudocode

findSplit function

Input: *A* – a feature according to which to split
S` – a training set at this point

28 *bestSplitInformationGain* ← ∞, *bestValueToSplit* ← null
29 **If** *A* is categorical **Then**
30 **return** calculateBestSplitForCategorical(*A*, *S`*)
31 **End If**
// the Logarithmic Split-Point Sampling
32 *sortedUniqueValues* ← find unique values for *A* in *S`* and sort them
33 *valuesToExamine* ← start with *sortedUniqueValues*[0] till *sortedUniqueValues*[last]
In steps of $\frac{sortedUniqueValues[\text{last}] - sortedUniqueValues[0]}{\log_2(|S`|)+1}$
34 **For** *v* in *valuesToExamine* **Do**
35 *informationGain* ← calculateInformationGain(*S`*, *v*)
36 **If** *informationGain* > *bestSplitInformationGain* **Then**
37 *bestSplitInformationGain* ← *informationGain*
38 *bestValueToSplit* ← *v*
39 **End If**
40 **End For**
41 **Return** (*bestValueToSplit, bestSplitInformationGain*)

Figure 2 FastForest training stage pseudocode - continued

5.2. For classifying

FastForest – predicting

Input: *S* – a test set, $S := (x_1, y_1), \dots, (x_q, y_q)$
1 **For** *s* in *S* **Do**
2 *votes* ← [0, … , 0] // votes size is as the number of classes
2 **For** *tree* in *FastForest* **Do**
3 *prediction* ← *tree*.predict(*s*)
4 *votes*[*prediction*] ← *votes*[*prediction*] + 1
5 **End For**
6 $predictions \leftarrow predictions + \text{argmax}(votes)$
7 **End For**
8 **Return** $predictions$

Figure 3 FastForest prediction stage pseudocode

6. Algorithm Explanation

6.1. For training

The explanation refers to Figure 1 and Figure 2.

In line 1 the FastForest creates a loop to grow the ensemble of *T* decision trees.

In line 2 it produces the Half-Subbagging, the training set for the t^{th} tree. It randomly selects $\frac{n}{2}$ samples from the training set and passes it to the function that builds a single decision tree (line 3).

In line 4 the tree that was built, M_t, is appended to the ensemble of trees of the algorithm.

The next lines describe the *buildDecisionTree* function: it is a recursive function that grows the decision tree. Its input is the training data remained at that stage, the tree grown so far and the total number of features in the set.

In line 7 the algorithm checks if it reached a stopping criterion for the recursion – whether the training set represents a leaf. This can be true for several reasons, among which –

1) The training set is pure and has samples of a single class only
2) The max_depth limitation of the tree is reached
3) The minimal number of samples for node condition is violated

The algorithm then creates a leaf, sets the relevant prediction for that leaf and returns the tree grown so far (lines 8-9).

Otherwise, in lines 11-12, the Subspacing size is calculated – how many features should be considered for the current split. To allow a quality split, the algorithm performs a Dynamic Restricted Subspacing and allows larger number of features when the number of samples in the node is less than $1/8$ of the size of the original training set. In that case the Subspacing size (k) will be

$$\mathrm{k} = \log_2\left(F \times \frac{n}{|S`|}\right) + 1\,,$$

where F is the total number of features, n is the total number of samples in the training set and $|S`|$ is the number of samples that reached that node. If the condition is not met then the Subspacing size will be

$$k = \log_2(F) + 1$$ (lines 13-14).

In line 16 the algorithm selects this number of features in random, and then loops on them and examines each one to determine whether splitting the tree on that feature will result with the best information gain (lines 17-20). The calculation is done with the *findSplit* function described below. If the split improves the current best information gain, the algorithm saves the information gain obtained thus far and the best feature and value to split on (lines 21-22). Once all the features were considered and the best feature and split point discovered, the algorithm creates the new training set according to the split and call the function again with the new parameters.

The *findSplit* pseudo code is in lines 28-41. The input for the function is the candidate feature and the training set at that point. Firstly, the algorithm checks whether the feature in the input is categorical (line 29). If so, then the original code of the Random Forest is activated (line 30). Otherwise, the algorithm performs the Logarithmic Split-Point Sampling:

Instead of going over all possible unique values of a numerical feature for a split, the algorithm considers only $\log_2$ of the number of records with non-missing values reaching that node:

$$\mathrm{hopsNum} = \log_2(|S`|) + 1$$

then, the unique values for the feature are sorted and the algorithm builds the array of potential values to be considered for the split: the range of the feature's values are divided by the number of hops, and the values are calculated is steps of –

$$step = \frac{sortedUniqueValues[\text{last}] - sortedUniqueValues[0]}{\log_2(|S`|) + 1}$$ (lines 32-33)

The algorithm calculates the information gain obtained from splitting on each of these values and returns the best value and information gain obtained (lines 34-41).

6.2. For classifying

The FastForest algorithm didn't enhanced the flow of the Random Forest for classifying. The pseudocode of the algorithms is described in Figure 3.

The input for the function is the test set. The algorithm loops on the samples in the set and activates the *T* decision trees *predict* function on them. It combines all the *T* predictions and uses a majority vote to output the predicted class per sample in the test set.

7. Illustration

In this section I illustrate the growing of one tree by the FastForest algorithm. The settings of the algorithm are:

1. Number of trees to grow (*T*) = 3 (only one tree will be demonstrated though)
2. Max depth of the tree = 3
3. Min number of samples in a leaf = 2

I created a synthetic dataset to be used as the training set, based on a simplified version of the *iris* dataset (Table 1). I'll use the entire dataset for the illustration and will not divide it into train and test sets.

#	f1	f2	f3	f4	class
0	**-1.2**	0.1	-1.2	-1.3	0
1	**-0.5**	0.7	-1.2	-1.0	0
2	**-0.7**	2.4	-1.2	-1.4	0
3	**-0.4**	2.6	-1.3	-1.3	0
4	**-1.1**	0.1	-1.2	-1.4	0
5	**-1.0**	0.3	-1.4	-1.3	0
6	**-0.4**	1.0	-1.4	-1.3	0
7	**-1.1**	0.1	-1.2	-1.4	0
8	**1.3**	0.3	0.5	0.2	1
9	**0.6**	0.3	0.4	0.3	1
10	**1.2**	0.1	0.6	0.3	1
11	**-0.4**	-1.7	0.1	0.1	1
12	**0.7**	-0.5	0.4	0.3	1
13	**-0.1**	-0.5	0.4	0.1	1
14	**0.5**	0.5	0.59	0.5	1
15	**-1.1**	-0.5	-0.2	-0.2	1
16	**0.5**	0.3	1.2	1.7	2
17	**-0.1**	-0.8	0.6	0.3	2
18	**1.5**	-0.1	1.2	1.7	2

#	f1	f2	f3	f4	class
19	**0.5**	-0.3	1.2	0.2	2
20	**0.7**	-0.1	1.2	1.7	2
21	**2.1**	-0.1	0.6	1.7	2
22	**-1.1**	-1.2	0.4	0.1	2
23	**1.7**	0.3	0.4	0.2	2

Table 1 A synthetic dataset for the illustration of the FastForest algorithm

Figure 4 illustrates the first step in the algorithm (line 2 in Figure 1). The dataset consists of 24 samples and the algorithm selects half of them in random (Half-Subbagging), per tree. So, the input per tree is a dataset with 12 samples only. We will concentrate on the growing of tree 2, in the first 2 calls for the *buildDecisionTree* unction. The Half-Subbagging result of the illustration example is presented in Table 2.

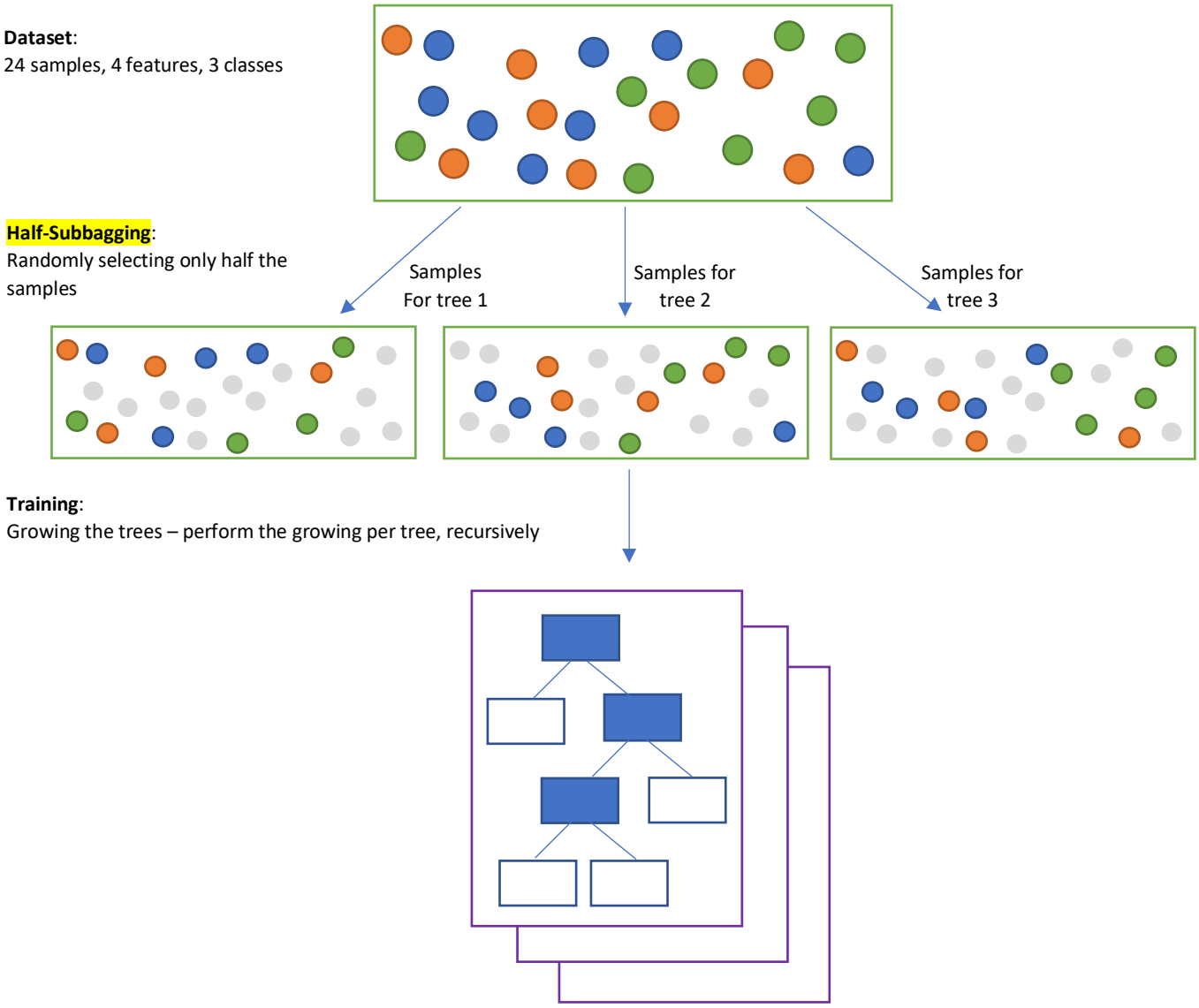


Figure 4 FastForest illustration – for the example with a dataset that has 24 samples, 4 features and 3 classes. Performing Half-Subbagging on the input and growing 3 trees

#	f1	f2	f3	f4	class
0	-1.2	0.1	-1.2	-1.3	0
1	-0.5	0.7	-1.2	-1.0	0
2	-0.7	2.4	-1.2	-1.4	0
3	-0.4	2.6	-1.3	-1.3	0
4	-1.1	0.1	-1.2	-1.4	0
5	-1.0	0.3	-1.4	-1.3	0
6	-0.4	1.0	-1.4	-1.3	0
7	-1.1	0.1	-1.2	-1.4	0
8	1.3	0.3	0.5	0.2	1
9	0.6	0.3	0.4	0.3	1
10	1.2	0.1	0.6	0.3	1
11	-0.4	-1.7	0.1	0.1	1
12	0.7	-0.5	0.4	0.3	1
13	-0.1	-0.5	0.4	0.1	1
14	0.5	0.5	0.59	0.5	1
15	-1.1	-0.5	-0.2	-0.2	1
16	0.5	0.3	1.2	1.7	2
17	-0.1	-0.8	0.6	0.3	2
18	1.5	-0.1	1.2	1.7	2
19	0.5	-0.3	1.2	0.2	2
20	0.7	-0.1	1.2	1.7	2
21	2.1	-0.1	0.6	1.7	2
22	-1.1	-1.2	0.4	0.1	2
23	1.7	0.3	0.4	0.2	2

Table 2 the Half-Subbagging for tree 2. The selected rows from the original dataset are highlighted in yellow

Once the Half-Subbagging is completed, the algorithm starts growing the tree. The growing is done recursively and stops once all the data is classified at the leaves. The procedure is illustrated in Figure 5. So, in the first call to the *buildDecisionTree* function (line 7 in Figure 1), the algorithm checks if the input data can be classified as a leaf. We have 12 samples and they do not meet the conditions for being a leaf – they are not pure, nor the *max_depth* limitation or the *min_samples_per_leaf* condition are met.

So, continuing to line 11, the algorithm calculates the Subspacing number: the number of features to select in random from the 4 features in total. Since we are in the first call, the number of samples equals to the total number of samples and the algorithm selects

$$k = \log_2(4) + 1 \rightarrow 3 \text{ features (line 14)}$$

Suppose f_1, f_2, f_3 were randomly drawn. Then, the algorithm loops over the 3 selected features, and for each finds the best value to split the data on (the *findSplit* function). In a greedy manner, once it discovers the feature and value that have the best information gain, it splits the data according to them and calls itself again with the new data (line 26). The information gain is calculated with the entropy formula.

In our dataset all the features are numerical. The *findSplit* function, as presented in the algorithm in Figure 2, first sorts all the values and calculates the split point candidates according to the Logarithmic Split-Point Sampling component. For f_1, for example, the unique values are: [-1.1 -0.5 -0.4 -0.1 0.6 0.7 1.5], total of 7 different values.

The Logarithmic Split-Point Sampling component takes these values and considers only $\log_2(|S`|)$, where $|S`|$ is the number of samples that reached that node, as candidate split values. It takes the first value in the sorted unique values array of the feature and adds to it the hop, which is

$$\frac{last\ sorted\ value - first\ sorted\ value}{\log_2(records\ number)+1} = \frac{1.5-(-1.1)}{\log_2(12)+1} = \frac{2.6}{4.6} = 0.52.$$

So, the split candidates for f_1 are [-1.1, -0.58, -0.06, 0.46, 0.98], only 5 values.

Then, the algorithm calculates the information gain for splitting according to each of the candidate values and returns the best feature and value to split on. The tables below display the search for the best feature and value in this first call to *buildDecisionTree* function.

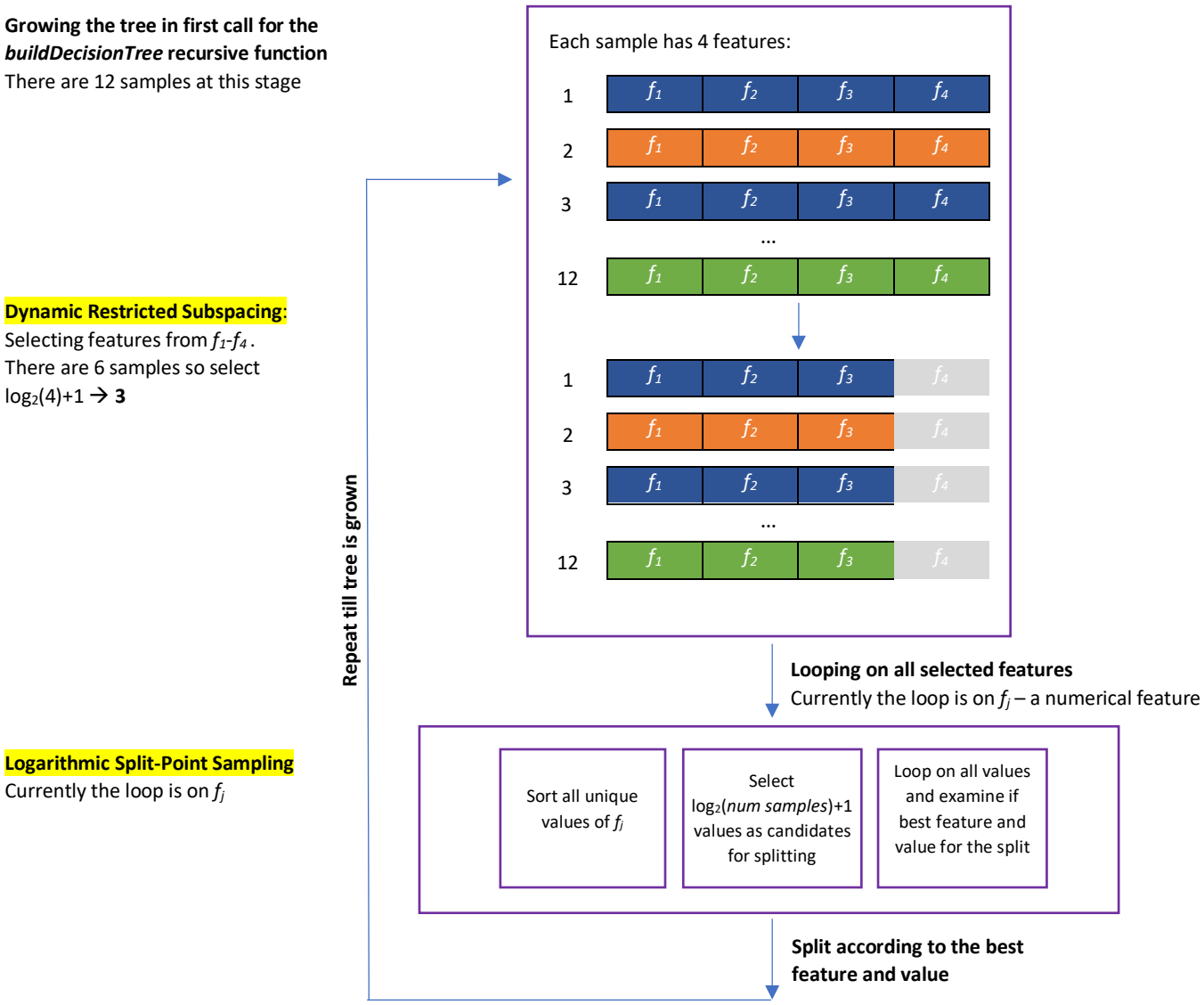


Figure 5 FastForest illustration example continued. It is the first call for the buildDecisionTree *function and the input consists of 12 samples. Each sample has 4 features. The algorithm builds the decision tree by selecting the features (Dynamic Restricted Subspacing) and finding the best feature and values to split the tree on. It repeats this procedure till all the dataset is classified in the tree's leaves and returns the final outcome to the* FastForest *object*

the best feature, value and entropy is highlighted in blue in the tables below:

First call for the *buildDecisionTree* function - Candidate split feature: f_1				
#	Candidate split value	Best gain thus afar	Split entropy	Data indices split (from Table 2)
1	-1.1	∞	1.585	[4,15,22] [1,3,6,9,11,12,17,18,20]
2	-0.58	1.585	1.585	[4,15,22] [1,3,6,9,11,12,17,18,20]
3	-0.06	1.585	1.333	[1,3,4,6,11,15,17,22] [9,12,18,20]
4	0.46	1.333	1.333	[1,3,4,6,11,15,17,22] [9,12,18,20]
5	0.98	1.333	1.442	[1,3,4,6,9,11,12,15,17,20,22] [18]

The unique values for f_2 are:

[-1.7 -1.2 -0.8 -0.5 -0.1 0.1 0.3 0.7 1. 2.6]

and the output of the Logarithmic Split-Point Sampling component for them is:

[-1.7, -0.84, 0.02, 0.88, 1.74]

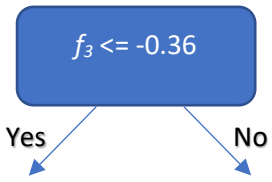
First call for the *buildDecisionTree* function - Candidate split feature: f_2				
#	Candidate split value	Best gain thus afar	Split entropy	Data indices split (from Table 2)
1	-1.7	1.333 (of feature f_1)	1.441	[11] [1,3,4,6,9,12,15,17,18,20,22]
2	-0.84	1.333	1.476	[11,22] [1,3,4,6,9,12,15,17,18,20]
3	-0.02	1.333	0.875	[11,12,15,17,18,20,22] [1,3,4,6,9]
4	0.88	0.875	1.268	[1,4,9,11,12,15,17,18,20,22] [3,6]
5	1.74	0.875	1.441	[1,4,6,9,11,12,15,17,18,20,22] [3]

And for f_3:

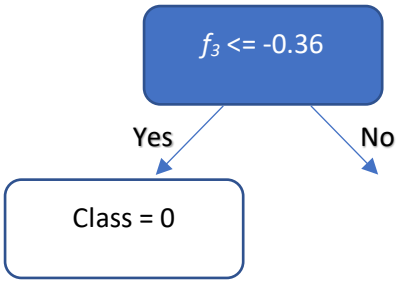
First call for the *buildDecisionTree* function - Candidate split feature: f_3				
#	Candidate split value	Best gain thus afar	Split entropy	Data indices split (from Table 2)
1	-1.4	0.875 (of feature f_2)	1.441	[4] [1,3,6,9,11,12,15,17,18,20,22]
2	-0.88	0.875	0.667	[1,3,4,6] [9,11,12,15,17,18,20,22]
3	-0.36	0.667	0.667	[1,3,4,6] [9,11,12,15,17,18,20,22]
4	0.16	0.667	0.918	[1,3,4,6,11,15] [9, 12,17,18,20,22]
5	0.68	0.667	1.268	[1,3,4,6,9,11,12,15,17,22] [18,20]

The output of the first call is to split on f_3 with value -0.36. and now the function calls itself recursively with the split data indices: Indices [1,3,4,6] creates the left sub-tree and [9,11,12,15,17,18,20,22] creates the right sub-tree.

At this stage the algorithm found the root of the tree and it is composed of –



Calling *buildDecisionTree* with the [1,3,4,6] indices creates a leaf node – the data is pure and classifies the samples reaching this node as class 0. The tree now was added a leaf:



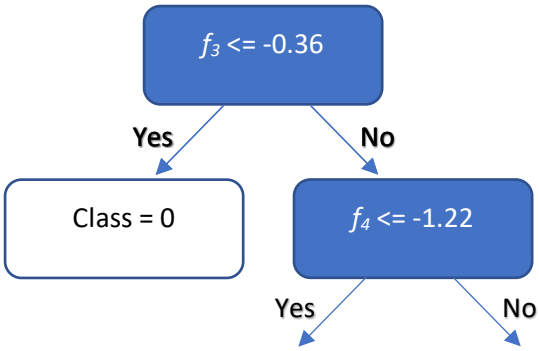
At this stage the Dynamic Restricted Subspacing takes place again for the branch to the right, and this time features f_1, f_2, f_4 are selected. The calculations for the right sub-tree with the [9,11,12,15,17,18,20,22] indices are:

Building left sub-tree with the *buildDecisionTree* function - Candidate split feature: f_1				
#	Candidate split value	Best gain thus afar	Split entropy	Data indices split (from Table 2)
1	-1.1	∞	1.0	[15,22] [9,11,12,17,18,20]
2	-0.45	1.0	1.0	[15,22] [9,11,12,17,18,20]
3	0.2	1.0	1.0	[15,17,22] [9,11,12,18,20]
4	0.85	1.0	0.862	[] []

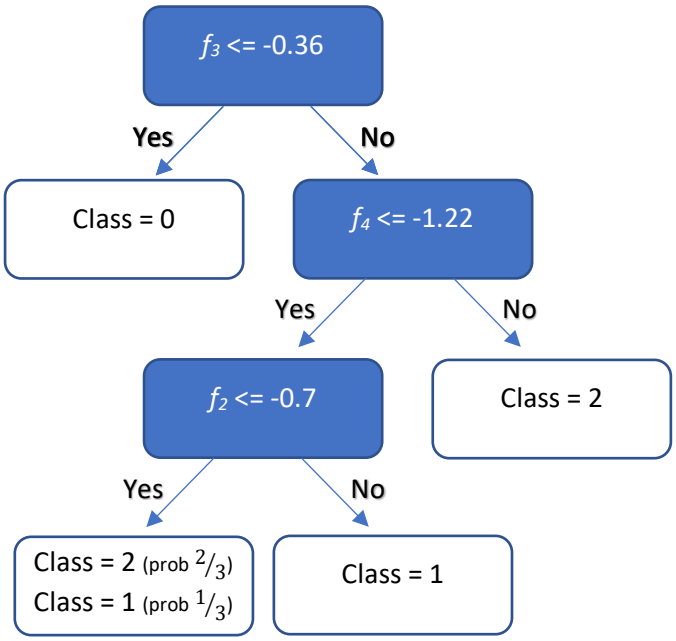
Building left sub-tree with the *buildDecisionTree* function - Candidate split feature: f_2				
#	Candidate split value	Best gain thus afar	Split entropy	Data indices split (from Table 2)
1	-1.7	0.862	0.862	[11] [9,12,15,17,18,20,22]
2	1.2	0.862	1.0	[11.22] [9,12,15,17,18,20]
3	-0.7	0.862	0.951	[11,17,22] [9,12,15,18,20]
4	-0.2	0.862	0.951	[11, 12,15,17,22] [9,18,20]

Building left sub-tree with the *buildDecisionTree* function - Candidate split feature: f_4				
#	Candidate split value	Best gain thus afar	Split entropy	Data indices split (from Table 2)
1	-0.2	∞	0.862	[12] [9,11,15,17,18,20,22]
2	0.27	0.862	0.951	[11,15,22] [9,12,17,18,20]
3	0.75	0.862	0.689	[9,11,12,15,17,22] [18,20]
4	1.22	0.689	0.689	[9,11,12,15,17,22] [18,20]

So, f_4 is selected and the tree is proceeding with –



The procedure is repeated until all the data is divided into the tree's leaves or the limitation for the tree's depth or number of samples in the leaf are met. The final outcome for this illustration is:



As can be seen, the tree stops at depth 3 and one of the leaves is not pure.

8. Strengths

1. FastForest is an optimization for Random Forest. It optimizes the processing time required for training the forest's estimators and in the paper the authors achieved an overall tested speed gain of over 24% compared with Random Forest on 45 datasets (when running on Windows platform).
2. It was also compared with 5 other ensemble algorithms: SysFor, Random Subspace, ForestPA, Bagging and Random Committee, and was found to be faster than them as well.
3. FastForest was also tested on Android smartphones, on 30 datasets, and outpaced Random Forest on all of them.
4. FastForest performance was found to be as good as Random Forest on the datasets tested.

9. Drawbacks

1. FastForest is composed of 3 optimization components: the Half Subbagging, the Logarithmic Split-Point Sampling and the Dynamic Restricted Subspacing, aiming to

reduce the processing time. The Logarithmic Split-Point Sampling and the Dynamic Restricted Subspacing component is relevant only for numerical features, so when a dataset has mainly categorical features then its contribution to the overall result is very limited.

2. The Logarithmic Split-Point Sampling is not considering the distribution of the unique values of the numerical values and rather assumes a uniform distribution. It divides the range of the values in $\log_2$(number of samples in the node) sections and splits the samples accordingly. If the distribution is very skewed then it will miss out many worthy potential candidates. For example: if the unique values of a feature are [0.01, 0.02, 0.03, 0.04, 0.05, 10] and 8 samples reached the node then the hop value will be $\frac{10-0.1}{\log_2 8+1} \approx 2.5$ and the algorithm will consider these values as split candidates: [2.6, 5.1, 7.6], resulting always in the same split outcome: [0.01, 0.02, 0.03, 0.04, 0.05] and [10]. If the data is normalized then it won't pose as a problem though.
3. For very simple datasets, the FastForest algorithm may take longer time for the training of the estimators. Due to the Dynamic Restricted Subspacing component, it may consider more features for splits when compared with the Random Forest algorithm. However, when the dataset is simple then the training time is short at any case.
4. When the dataset is extremely imbalanced than the Half-Subbagging may result with training or test sets without some of the classes. Using stratification for the Half-Subbagging does not always help, since the split to train and teat sets may already resulted is too small a number of the minority classes.

10. Experimental Results

I conducted the experiments on several computers in parallel, on the Windows platform. The protocol of the experiments–

1. Loop on all datasets and –
    a. Read the dataset's csv file
    b. Perform minimal and generic preprocessing for the data (more details in section 10.1.1)
    c. Create 10-Fold CV for splitting for training and test sets (section 10.1.2). For FastForest and Random Forest do -
        i. Perform 3-Fold CV on the train set to tune the hyper-parameters for each fold created in step 3. Above (section 10.1.3)
        ii. Create the estimator with the best parameters and predict the class for the test set with the best hyper-parameters
        iii. Calculate performance measurements (results in 10.2.1)
2. Test if the differences between FastForest and Random Forest results are statistically significant (section 10.2.2)
3. Build a Meta-learning model with XGBoost and calculate the features importance measures (section 10.2.3)

10.1 Implementation notes

The paper's authors did not publish their code for FastForest. My original intention was to derive the scikit-learn package implementation of Random Forest and to override the relevant parts with my implementation of the FastForest components.

Random Forest code is written in scikit-learn package in Python and its estimators are written in Cython. I was able to change the Python code, but could not make the estimators

code to cythonized. As a result, I implemented a Decision Tree Classifier from scratch, in python, based on examples I found online ([1], [2], [3]). I implemented binary decision trees, as in scikit-learn, and for calculating the information gain I chose the entropy measure. In addition, the grown trees were not pruned, as in the original Random Forest paper.

Since my code is written in Python and not in Cython, and is not executed in parallel, then it is slower than Scikit-learn implementation and should be run on several computers for a couple of days in order to collect all the results.

10.1.1 Datasets preprocessing

The preprocessing was done automatically per dataset and was kept as general and generic as possible.

The classification column (class) was set as the last column of the datasets, except for the *solar-flare* dataset. The class column appears there as the first column, and I discovered it while analyzing the results from this dataset, and did the change to improve the results.

I filled in missing values with a unique value. I preferred not to drop samples since most datasets are small. I also performed one hot encoding for categorical features and did label encoding for the class column.

Lastly, several datasets have only 1-4 samples of certain class values. I marked these values as extreme minority classes and had several venues for handling them –

1. Find more data – I couldn't find additional relevant datasets online to enrich my data and increase the number of the extreme minority classes
2. Perform random oversampling or undersampling – most datasets with this issue are quite small, so undersampling is irrelevant. Oversampling is also problematic since there are only 1-4 samples for the process
3. Remove the extreme minority classes – I chose that venue and dropped the samples with extreme minority classes from the datasets

10.1.2 Data split

As instructed, I used 10-fold CV for the splits. Since some dataset were imbalanced I did the splits with stratification, unless the number of the least populated class value was less than the number of folds (10). This led to some errors in scikit-learn, when the test set had classes that did not appear in the training set. I preferred to keep the 10-fold CV, as instructed, and not minimize it to match the datasets. Instead, I removed "redundant" classes when such contradictions arose.

In addition, one dataset, *lenses*, was very small, with 24 samples in total, and imbalanced. Consequently, the 10-folds CV failed for it and so I performed only 4-folds instead.

10.1.3 Hyper parameters tuning

I used the RandomizedSearchCV for tuning the algorithms hyper-parameters, as instructed.

For FastForest and Random Forest I tuned –

1. n_estimators – the number of trees for the forest to grow. I ran the experiments on several "regular" PCs so I set the options as 5, 15 and 60 trees, to limit the runtime.
2. max_depth – the maximal depth of the Decision Tree, 5, 8 or 15. These values should allow growing satisfactory binary trees

3. min_samples_leaf – the minimal number of samples in a leaf node in the Decision Tree. If the number of samples reaching a node were less than this value then a leaf node was created. The options were 1, 2, 4

For Random Forest I also tuned –

1. bootstrap – whether to perform bootstrapping or not. For FastForest I always chose half the samples, as required in the algorithm description
2. max_features – the number of features to consider for splits (Subspacing). The options were:
    a. squared root of the number of features in the dataset
    b. $\log_2$ of the number of features in the dataset

    for FastForest I used the Dynamic Restricted Subspacing component, as described in section 6.1.

I chose these 5 parameters since I expected them to have the most meaningful impact on the results. Since I ran the experiments on standard desktop computers I chose the options per parameter to allow the algorithms reasonable performance but with reasonable runtime as well.

10.2 The results

10.2.1 Performance comparison between FastForest and Random Forest

The performance was measured according to several metrics mentioned in the exercise instructions. I used functions implemented in the scikit-learn package, or implemented them myself, specifically –

1. Accuracy – computed with the accuracy_score function
2. TPR – I implemented the calculation by calculating the confusion matrix and FP, FN, TP and TN. Then I calculated the average of the $\frac{TP}{TP+FN}$. When $TP + FN = 0$ then my function returned NaN and the value is missing in the results
3. FPR – similarly to the calculation of TPR, I calculated FPR myself to be $\frac{FP}{FP+TN}$
4. Precision – I used the average_precision_score function. For multi-classification problems I first binarized the classes and calculated the function in a one-vs-rest manner. Then I calculated the average for the results of all the classes.
5. AUC – I used the roc_auc_score. For the multi-classification problem I added the *multi_class* parameter as "ovr" (one-vs-rest) and the *average* parameter as "weighted", since many datasets were imbalanced
6. Precision-Recall score – I used the average_precision_score function with the *average* parameter as "weighted"
7. Training time – measured in seconds. I first tuned the hyper parameters and only then measured the training time for the best results. The search space for the parameters tuning per algorithm was different so it was excluded from the training time calculation
8. Inference time – measured in seconds and normalized for 1000 samples

The average of each of the performance measurements is in Table 3. We can see the average result per measurement and then its standard deviation in parenthesizes.

	Accuracy	TPR	FPR	Precision	AUC	PR-Curve	Training Time	Inference Time
Fast Forest	**0.7402** (0.1939)	**0.6569** (0.2249)	0.2097 (0.1743)	**0.718** (0.2254)	**0.8411** (0.1633)	**0.7033** (0.2416)	**1.6518** (2.6053)	0.6357 (0.5289)
Random Forest	0.6928 (0.2065)	0.5996 (0.2284)	**0.2466** (0.1785)	0.6489 (0.249)	0.7838 (0.1841)	0.6421 (0.2516)	21.0624 (66.5051)	**0.4877** (0.4818)

Table 3 Average and std results for the performance measurements per algorithm for the 150 datasets tested

The complete results, per dataset and cross validation run, are in an accompanying file – results_with_avg.csv (attaching it here makes the document too long).

We can see that in all the measurements but the inference time, FastForest achieved better average results than the Random Forest algorithm. The measurement in focus, the training time, is significantly lower for FastForest than for Random Forest, **1.65** seconds vs. **21.06** seconds.

10.2.2 Statistical significance

I performed the Mann–Whitney U test to examine the significance of the differences between the results of the two algorithms, as implemented in the Scipy package. The tests' results are summarized in Table 4.

Parameter	FastForest average value	Random Forest average value	Statistic	p-value
Training time	**1.6518**	21.0624	1029040.5	0.0001
Accuracy	**0.7402**	0.6928	961436.5	<0.0001
AUC	**0.8411**	0.7838	900127.5	<0.0001
Inference time	0.6357	**0.4877**	893439.0	<0.0001

Table 4 Mann–Whitney U test results for the differences between FF and RF

As in the original paper, the training time when applying the FastForest algorithm is statistically significantly lower than the training time of the Random forest algorithm for the 150 datasets tested (p-value 0.0001). It takes on average 19.4 seconds less to train the algorithm with FastForest, which is 92% faster.

The accuracy and AUC differences also turned out to be statistically significant (p-value < 0.0001). in the paper FastForest frequently exceeded Random Forest for classification accuracy as well. It implies that FastForest is overfitting less than Random Forest thanks to its three revised randomizing components.

Interestingly, the inference time gap is statistically significant as well (p-value < 0.0001). on average, it took the FastForest algorithm to infer the class more slowly than Random Forest. When analyzing the hyper-parameters tuned per algorithm (see Table 5), we understand that the reason lies in them: the best hyper-parameters found for FastForest are slightly more complex than the ones found for Random Forest, which resulted in longer inference time.

Parameter	FastForest average value	Random Forest average value
n_estimators	**22.326**	19.9632
min_samples_leaf	**1.9933**	1.9692
Max_depth	**8.1827**	7.7129

Table 5 Hyper-parameters average value per algorithm

10.2.3 Meta-learning model

In order to train a model to predict which algorithm will prevail based on the meta features of the dataset, I created a data frame based on the meta-features file provided with –

1) Each row was duplicated, for FastForest and Random Forest
2) The 'Best AUC' column was added, for the classification. It was populated but either 1 or 0 per row, depending if that row's algorithm had the best AUC for the row's dataset

Then I ran the XGBoost algorithm on the new data frame with Leave-One-Dataset-Out CV protocol. The results of all the runs are in the accompanying file meta_results_with_avg.csv, and their average is in Table 5.

Accuracy	TPR	FPR	Precision	AUC	PR-Curve	Training Time	Inference Time
0.7533 (0.4325)	0.7533 (0.4325)	0.2467 (0.4325)	0.7533 (0.4325)	0.7533 (0.4325)	0.8767 (0.2163)	0.0729 (0.0269)	0.4999 (0.1636)

Table 6 Average results of the meta-learning model

We can see that the model's accuracy and AUC were 0.7533, meaning that the model could predict which algorithm will yield the best results for more than three quarters of the datasets.

The importance of the 10 most influential meta-features is presented below for weight (Figure 6), gain (Figure 7) and cover (Figure 8). The complete importance results per type are in the accompanying file importance.csv.

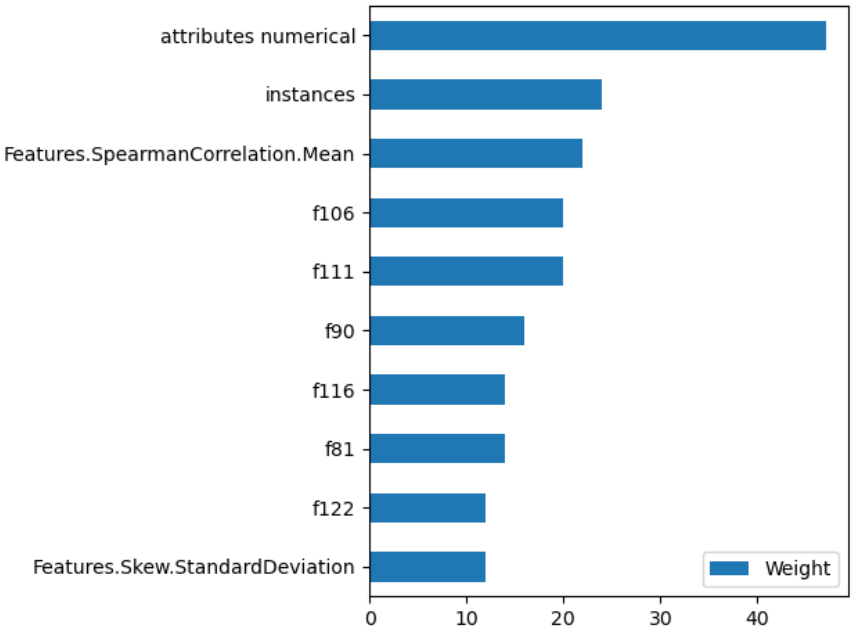


Figure 6 Ten most important features according to weight

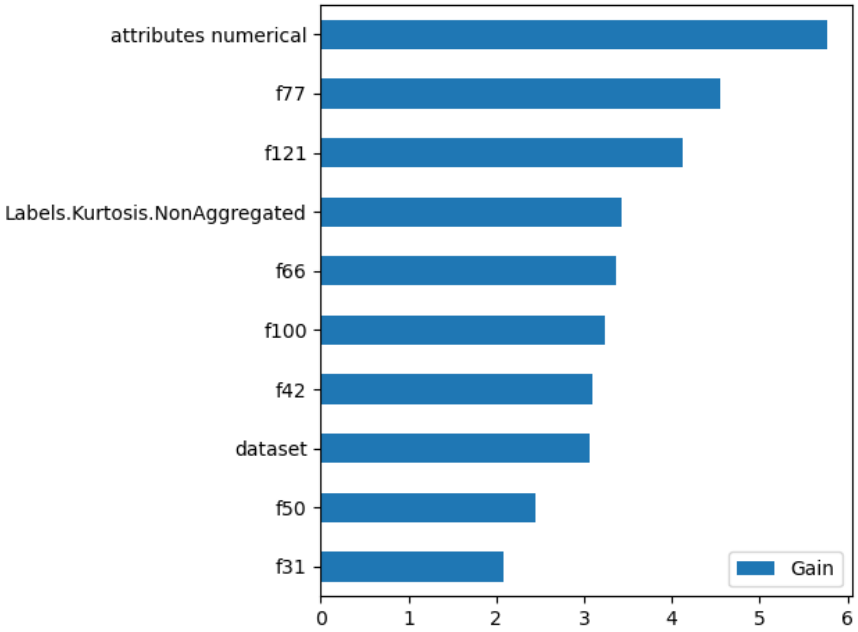


Figure 7 Ten most important features according to gain

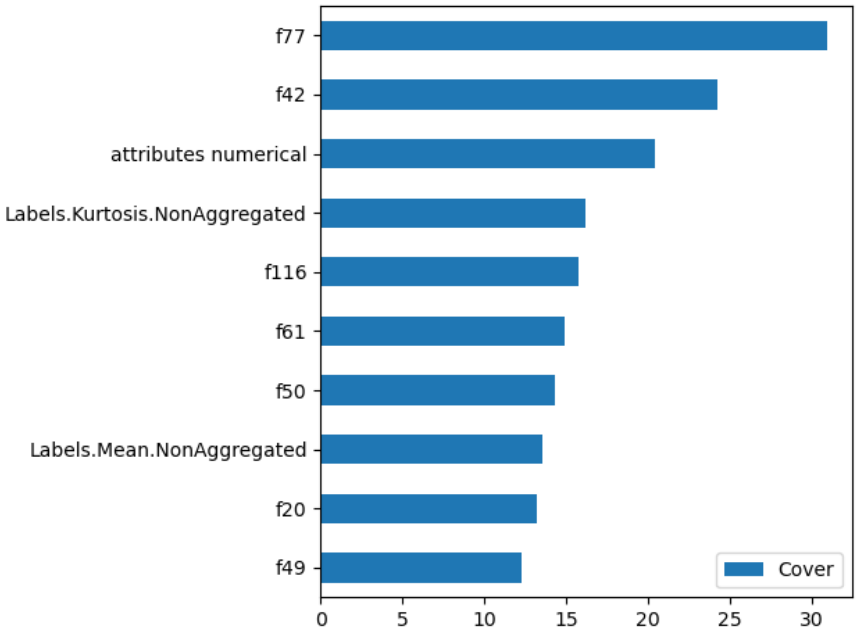


Figure 8 Ten most important features according to cover

As expected, we see different results per importance type. When considering the features' weight, that is, the relative number the feature was used to split the trees in the model, the *attribute numerical* and the *instances* meta-features was the most important.

For the gain type, the *attribute numerical* and the *f77* meta-features improved the accuracy the most, and for the cover type, *f77* and *f42* meta-features had the largest number of samples concerned by them.

SHAP summary plot is presented in Figure 9. Most features had a mild effect on the prediction except for the *attributes numerical* meta-feature. When there was a large number of numerical features in the dataset is contributed to the prediction and vice versa.

Figure 9 SHAP features importance summary plot

11. Conclusions

In this exercise I investigated the FastForest algorithm. I implemented it according to the ideas presented in its paper and tested it on 150 classification problems. Then I compared its performance with the Random Forest algorithm. Lastly, I created a meta model for predicting when to use the algorithm according to datasets meta-features.

The purpose of the FastForest algorithm was to run faster than Random Forest while keeping the same level of accuracy. My experiments achieved this goal – it was statistically significantly faster than Random Forest and on average outpaced Random Forest by 92%.

FastForest AUC metric also outperformed Random Forest results by 7.3% and the accuracy by 6.8% (statistically significant differences).

However, FastForest inference time was statistically significantly slower than Random Forest. This was not mentioned in the original paper. When considering the complexity of the trees created per algorithm, we can see that the number of estimators and the max tree depth were tuned to be slightly larger for FastForest. This can explain the results and to raise a question – whether FastForest creates more complex trees on average due to its revised components.

FastForest achieved the best results when most of the features in the dataset were numerical and had many unique values. In these cases, it was several degrees faster than Random Forest. For example, the *mfeat-karhunen* dataset has 64 numerical features with many unique values (1993-1994 per feature). Consequently, FastForest average training time for this dataset was 6.52 seconds while Random Forest's time was 352.43 seconds.

On contrary, for small datasets, with categorical features, the training time for FastForest was worse than the one for Random Forest. For example, dataset *semeion* has 256 binary features. FastForest average training time was 8.21 seconds while Random Forest finished the task by merely 0.23 seconds on average.

There are some open questions for future research –

1. Investigate why the hyper-parameters tuning for FastForest grows trees that are slightly more complex than Random Forest's on the same dataset
2. Adapt the Logarithmic Split-Point Sampling component to not uniform distributions
3. Optimize the algorithm for datasets with many categorical features
4. Compare it with additional algorithms

12. Citations

The paper was published on 2020 and was not yet cited by others.

However, the paper's authors performed similar earlier work in [4]. There, they worked on a single Decision Tree and also aspired to decrease the processing time of the tree. When encountering a numerical feature as a candidate for the tree split, they limit the number of values to examine (the Split Point Sampling, or SPS) to up to 20, distributed evenly on the feature's values range. They empirically showed that this method, called SPAARC (Split-Point And Attribute Reduced Classifier), decreased the training time whilst maintaining the tree's performance similar to a regular Decision Tree.

The FastForest was created to examine SPAARC in an ensemble and to improve it by changing the arbitrary Split Point Sampling value of 20 to a parameter with relevance to the data: The Logarithmic Split-Point Sampling in FastForest takes into account the number of records in each split. It keeps the number of split candidates low when there is a large number of records, and increases the number when the records count decreases.

13. Additional references

[1] Brownlee, J., How to Implement Random Forest From Scratch in Python (2016)

[2] Kumar, V., Random forests and decision trees from scratch in python (2018)

[3] Mantey, S., Coding a Decision Tree from Scratch (Python) (2020)

[4] Yates D., Islam M.Z., Gao J. (2019) SPAARC: A Fast Decision Tree Algorithm. In: Islam R. et al. (eds) Data Mining. AusDM 2018. Communications in Computer and Information Science, vol 996. Springer, Singapore