# Machine learning course – final exercise

Shavit Talman 034234914

## Algorithm name

FastForest

## Reference

Yates, Darren, and Md Zahidul Islam. "[FastForest: Increasing Random Forest Processing Speed While Maintaining Accuracy](https://arxiv.org/pdf/2004.02423.pdf)." arXiv preprint arXiv:2004.02423 (2020).

## 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.

## Short Description

### Original Random Forest keynotes

The [Random Forest algorithm](https://en.wikipedia.org/wiki/Random_forest) (**RF**) is an ensemble learning method for classification and regression. It uses many 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:

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
2. Subspacing – suppose the feature number is *m*. RF select a subset *k* features, *k*<*m* at each split. In scikit-learn implementation or

### 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 of the samples, when the training samples size is *n*.
2. Logarithmic Split-Point Sampling – when testing a numeric feature for the best split value, the method examines only Log2 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 in hops so that no more than Log2 of the number of records in that node plus 1 are considered.

There is no change for nominal features.

1. 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 increases, to allow selecting quality features.

## Pseudo-Code

### For training

Figure 1 FastForest training stage pseudocode

**FastForest – Building the ensemble**

**Input**: *S* – a labeled training set,

*T* – number of classifiers to train

– number of features

1 **For** *t*=1 to *T* **Do**

2 *S`t*  random sample with size from *S*, with replacement

3 *Mt* *buildDecisionTree*(*S`t*, *Mt*, )

4 *FastForest* *FastForest* + *Mt*

5 **End For**

6 **Return** *FastForest*

**buildDecisionTree function**

**Input**: *S`* – a training set passed in the tree build recursion. its size is |*S`*|

*Mt* – a classifier that was trained (decision tree), passes recursively during tree build

– number of features

7 **If** *S`* represents a leaf **Then**

8 *Mt* += createLeafWithPrediction()

9 **return** *Mt*

10 **End If**

// calculates the size of the Subspacing, the number of features selected from *F* at random

11 **If** |*S`*| < 0.125*n* **Then**

12

13 **Else**

14

15 **End If**

16 *A* select features from in random

17 *bestSplitInformationGain* ∞, *bestFeature* null

18 **For** = 1 to **Do**

19 *splitCandidateInformationGain* findSplit(*Ai*, *S`*)

20 **If** *splitCandidateInformationGain* > *bestSplitInformationGain* **Then**

21 *bestSplitInformationGain* *splitCandidateInformationGain*

22 *bestFeature* *Ai*

23 **End If**

24 **End For**

25 *newS* *S`* remainder after split according to *bestFeature*

26 *Mt* += buildDecisionTree(*newS*, *Mt*, )

26 **Return** *Mt*

Figure 2 FastForest training stage pseudocode - continued

**findSplit function**

**Input**: *A* – a feature according to which to split

*S`* – a training set at this point

27 *bestSplitInformationGain* ∞, *bestValueToSplit* null

28 **If** *A* is nominal **Then**

29 **return** (*bestValueforSplit*, *informationGain*)

30 **End If**

31 *sortedUniqueValues* find unique values for *A* in *S`* and sort them

32 *valuesToExamine* start with *sortedUniqueValues*[0] till *sortedUniqueValues*[last]

In steps of

33 **For** *v* in *valuesToExamine* **Do**

34 *informationGain* calculateInformationGain(I, *S`*, *v*)

35 **If** *informationGain* > *bestSplitInformationGain* **Then**

36 *bestSplitInformationGain informationGain*

37 *bestValueToSplit v*

38 **End If**

39 **End For**

40 **Return** (*bestValueToSplit, bestSplitInformationGain*)

### For classifying

Figure 3 FastForest prediction stage pseudocode

**FastForest – predicting**

**Input**: *S* – a test set,

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

7 **End For**

8 **Return**

## Algorithm Explanation

### For training

The explanation refers to figure [1](#_For_training) and [2](#_For_training).

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

In line 2 it produces the training set for the *t*th tree. It randomly selects samples from the training set and pass it to the function that build a single decision tree (line 3).

In line 4 the tree that was built, *Mt*, 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 and the total number of features in the set.

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

1. The training set is pure and has sample of only 1 class in it
2. The max\_depth of the tree is achieved
3. The minimal number of samples for node 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 line 11,

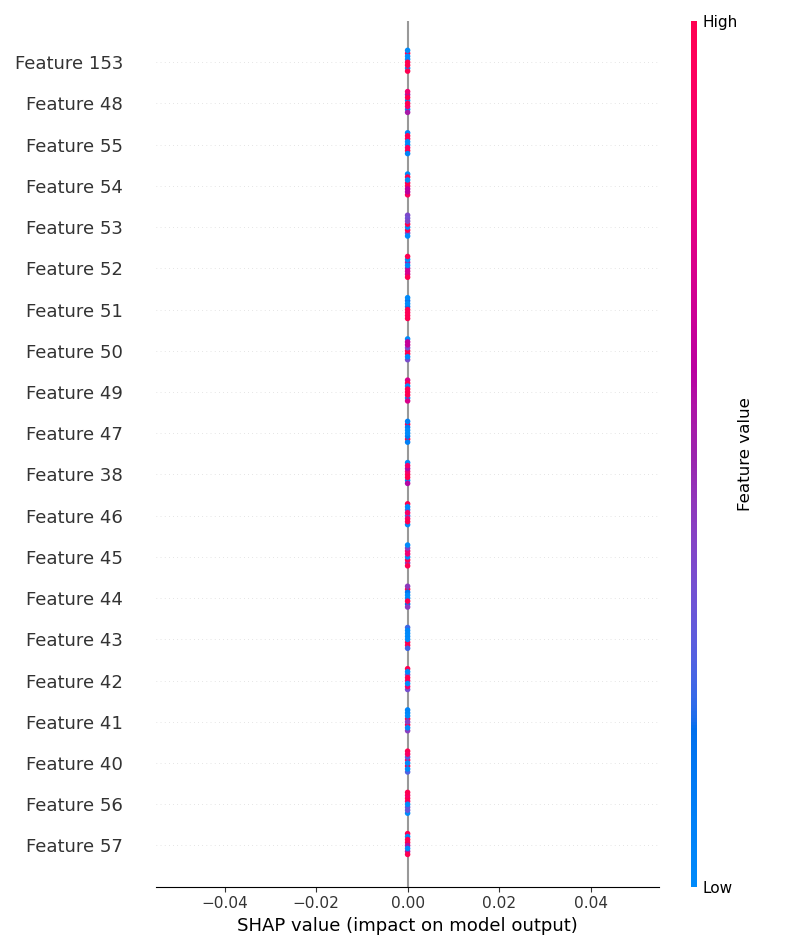
## Illustration

## Strengths

## Drawbacks

## Experimental Results

## Conclusions



## Citations

1. <https://arxiv.org/pdf/2004.02423.pdf> - the paper
2. <https://towardsdatascience.com/random-forests-and-decision-trees-from-scratch-in-python-3e4fa5ae4249> - building a tree from scratch in Python
3. <https://machinelearningmastery.com/implement-random-forest-scratch-python/> - another example for building a tree from scratch in Python
4. <https://www.sebastian-mantey.com/code-blog/coding-a-decision-tree-from-scratch-python-p4-helper-functions-3> - another

\*\* ovo is less prone to imbalance in dataset and ovr is fast 🡪 use ovo in our calculations since some of the datasets are imbalanced