Decision Tree & Random Forest from Scratch

Jonas Vander s1005822 and Alexander Zehetmaier s1013644

Radboud University

Submitted: 13 January 2019

Abstract

This paper investigates in building a decision tree and random forest from scratch. First this report introduces the mathematical details of creating a decision tree, its problem space and about previously performed work. Using the knowledge gained, the next section introduces the concept of ensemble algorithms and builds a random forest on top of the programmed 'DecisionTree' class. A real-world dataset about kickstarter projects is introduced on which these two algorithms are tested and reported. The results are compared to the scikit-learn machine learning library. It was nice to see that our algorithms could compete with the equivalent algorithms from scikit-learn with regard to accuracy. Nevertheless scikit-learn fitted twenty-thousand data records about 1000 times faster than the algorithms programmed by ourselves. For reproducibility of these results a Jupyter Notebook is provided in our github repository. Finally we discuss further improvements and possible future directions.

Keywords: Decision Tree, Random Forest, CART Algorithm

1 Introduction

"Random Forests do not get the attention they deserve, in the area of machine learning." PhD Rachel Thomas, co-founder of fast.ai.

The random forest is ranked as one of the best supervised machine learning algorithm when the given data is structured. This is shown by the winning strategies for kaggle competitions [1]. Random forests are often used by experts since it achieves high accuracies when used right, but is also a beginners choice since it forgives specific errors. Slightly inbalanced data does not throw a random forest off the track. The building block of every random forest is a group of decision trees trained on different random samples of the training data with replacement. This emerges from the fact that one decision tree alone inclines to overfit the training data.

Let's dig into the details of how to implement a random forests building block, the decision tree.

2 Decision Tree

A decision tree is a supervised machine learning algorithm which is most efficient on structured data. Since the operations made by the algorithm are easy to interpret a decision tree is considered as a white box algorithm.

2.1 Description

2.1.1 Tree Structure

The implementation was written in a object oriented way, which means that it is possible to speak about the structure of the implemented decision tree by the names of the implemented classes. To begin with, there is the "DecisionTree" class, which besides methods to classify, print and calculate the score, holds onto a reference of its root node. This node is made up by the classes "DTNode", "DTLeaf" and "DTSplit", where "DTNode" is a abstract class, which is extended by each other class. A "DTLeaf" holds onto a label, which can be retrieved when classification takes place. On the other hand, a "DTSplit" holds onto two other instances of "DTNode". This makes the "DTNode" into a recursive structure, which can by itself be represented as a binary tree. The "DTSplit" also holds onto an instance of the class Splitrule, which is a function like object that makes a decision as to in which of its nodes the classification should continue.

2.1.2 Fitting Data

The process of fitting or learning data happens at time of the instantiation of the tree. The new tree object has to be provided with a two dimensional numpy array as for the data, and a one dimensional array as for the labels of the target classes. This data is then used to create the root node.

To create a new node, the "DTNode".fit method is used. This is a static method of the class. It takes the same parameters as the "DecisionTree" as well, but also takes a reference of the tree, in order to keep track of meta data, and a integer indicating the depth of the tree. The outcome of the function is a instance of either "DTLeaf" or "DTSplit".

The subprocessies of fitting a node are:

- Deciding whether to continue to split or not, in order to prune the tree
- Finding the best Splitrule over all attributes
- Checking whether the resulting split is sensible
- Producing the output node and returning it

2.1.3 Pre pruning

When fitting a new node, it is always considered to prune the tree. The criteria used to decide whether the branch should be pruned is taken from the remaining data to be fitted, the remaining class labels, and meta-data about the current node and the tree object.

The considered cases in which the branch should be pruned are:

- The depth has reached the maximum depth.
- There are fewer items that would have to be split, than in a provided threshold. This threshold by standard is at 10, but it can changed by setting the meta-parameter on the tree
- There exists only one unique class label anymore
- In every remaining row, all data is equal

2.1.4 Find Best Split

To find the best split, on each attribute, all possible split values are tested, and compared by their information gain. Then the value with the best information gain is extracted, in form of a Splitrule object. With that one Splitrule exists for each attribute. All these splits are then compared again, by their information gain, and the one with the best result becomes the Splitrule for that particular split.

2.1.5 Information Gain

The information gain is a value that represents how valuable a split can be in terms of the increased purity of parts of the split.

In this case the cart algorithm has been implemented, which meant that the formula that was chosen to calculate the information gain in this case is the gini gain. This is calculated as a weighted average of the gini coefficient of each part of a split.

2.1.6 Gini Coefficient

The gini coefficient also called gini index is a measure of statistical inequality also denoted as impurity. It is defined as

$$GINI(t) = \sum_{t} p(c|t)^2$$

where t is a node, c is a unique value in the outcome labels, and p(c—t) is the frequency of c in the labels at this node.

2.1.7 Computational Complexity

The worst case time complexity of initializing and fitting a tree is approximately

$$O(n^2 * log(n) * m * 2^d)$$

where n is the amount of data points in the dataset, m is the amount of columns in the data to be fitted, and d is the allowed depth of the tree.

To show how this comes to be, the "fit" method of the "DTNode" class, and many methods used by it have to be considered. For brevity's sake, only the methods that significantly impact the time complexity will be treated here. In the method there are two such methods. These are the best "bestRule" method, of the same class, which has a complexity of

$$O(n^2 * log(n) * m)$$

as well as the potential creation of a "DTSplit" object, in which the method would be called recursively.

The complexity of the "bestRule" method comes to be, because besides other things, it creates a list of "Splitrule" instances, where there is one "Splitrule" for each column. Therefore a rule is created m times.

With the m being explained this leaves us with

$$O(n^2 * log(n))$$

which is the complexity of the initialization of a "Splitrule" object.

The complexity of this function is mostly a result of the frequent use of the numpy function unique. As a subroutine of this function, it sorts an numpy array with the same length as the data points at that point in time. This leads to the $n * log(n)$ part of the equation. This function is used in the function itself directly, but also in the "GINI" method. This method itself is called by the init of Splitrule once for every unique value, which means once for every possible value to split over.

It follows that the run time is not necessarily $n^2 * log(n)$. This is only the case in the worst case scenario, in which every value in the selected column is unique. Nonetheless this clears the $n^2 * log(n)$ component of the complexity.

Lastly the 2^d component remains unexplained. It is caused by the recursive call to the fit function. What is important here is that the recursion at place is double recursion, which is why the base to d is 2.

The time complexity of classifying a data point is

$$O(d)$$

where d is the depth of the tree. This comes to be because one branch of the tree may have to be evaluated up to the branch of the tree.

2.2 Problem Space

A decision tree is capable of taking a decision-making problem and breaks it down into a set of simpler decisions [3].

It is important to recognize that this requires structured data. Structured data has the properties of being easily stored, queried and analyzed, it is usually found in the form of databases.

A decision tree is mostly used to solve supervised machine learning problems, but it can also be used for unsupervised problems like interpretable hierarchical clustering [4].

2.3 Previous Work

The decision tree is already more than 50 years old and had numerous improvements since then [12].

Nevertheless the CART-algorithm implemented in this report is quite young and developed from Breiman et al. in 1984. Instead of using stopping rules, the CART algorithm was the first decision tree which grew large trees and afterwards pruned to the lowest cross-validation estimate.

In 1996 Brown et al. proposed a linear programming solution to split linear combinations of attributes.

3 Random Forest

Decision trees tend to overfit the training data and possibly do not generalize to unseen data records [5]. This property results from the fact that decision trees act greedy when splitting the data records into different partitions.

As a matter of fact decision trees likely find sub-optimal solutions. This problem can be addressed by using decision trees in an ensemble [6]. One way of doing this is to program a random forest, which uses a group of decision trees. Each decision tree is trained on different random sample from the training dataset with replacement (bootstrap).

When predicting an unseen data record all decision trees predict a possible outcome and a majority vote decides the class label. If the random forest is used for regression, the outcome of the different decision trees is averaged.

3.1 Description

3.1.1 Structure

The implementation of the random forest was also written in an object oriented way, which is why the structure can be explained by the classes that it uses. The random forest mainly relies on the implementation of the previously explained "DecisionTree" class. Therefore the only class that needed to be introduced was the "RandomForest" class itself. Each instance of "RandomForest" only needs to keep a reference of a list of "DecisionTree" instances, to be fully functional.

3.1.2 Fitting Data

When a instance of the "RandomForest" class is initiated, it needs to be provided with the data that it will be fitted on. This means a two dimensional numpy array is expected as the data to be fitted, and a one dimensional numpy array for the labels that the data is to be fitted to. It also needs to be provided with a number, which represents the amount of trees that will be generated in this forest.

The fitting process begins by creating samples of the overall data and labels, with which the decision trees that the random forest uses, can be fitted. It follows that as many samples will be created, as there are trees to be created. Each of these samples will have a total size of one third of the total data. Following this, with each sample, a decision tree will be created.

3.1.3 Computational Complexity

The worst case run time complexity of the initialization of a random forest is

$$O(n^2 * log(n) * m * 2^d * t)$$

where n is a third of the total data points in the dataset, m is the amount of columns in the data to be fitted, d is the allowed depth of the tree and t is the amount of trees, chosen to be in the forest.

It should be apparent that the only changes to the run time complexity of the decision tree lie in the parameter n, which is now only a third of the data, since each sample that is used to create a tree only has a third of the data points, and in the new parameter t, which just introduces a number for how often the fitting of a tree has to be performed.

The worst case run time complexity of classifying a data point is

$$O(t * log(t) + t * d)$$

where t is the amount of trees in the forest, and d is the allowed depth of the trees.

This comes to be because first, each tree has to be evaluated to get a total of t labels. Each tree evaluation will take d steps. This leads to the $t * d$ component of the expression. Following that the numpy unique function is used on the list of labels, to find the most frequent label, which takes $t * log(t)$ steps. Since it is depends on the parameters of the forest if the $t * log(t)$ component or the $t * d$ component will have the highest impact on run time, both are included in the expression.

3.2 Problem Space

Since a random forest is based on multiple decision trees the problems it solves are quite similar to the problems solved by a decision tree described in section 2.2.

Random forests are again mostly used on structured supervised data. Nevertheless it is even possible to implement an unsupervised random forest predictor [7].

3.3 Previous Work

1995 the first proposal about fitting multiple decision trees to selected subspaces of the feature space was released [13].

Anyhow the random forest needed a few more years to become attractive for the research community through a demonstration from Breiman. He argues that it is easy to execute it in parallel, faster than bagging and at least as accurate as adaboost.

Researach in 2012 showed that genetic algorithms could boost the performance of random forests even further [14].

4 Dataset

The dataset used for this report is about kickstarter projects, structured and free available on Kaggle [2]. It consists of 15 columns and has more than 375000 records.

One of the columns is called 'state' and describes if a submitted project was a success or failure. This is also the dependent variable and predicted by the classification algorithms described above. About 12% of the data records did not have any meaningful entry for the 'state' variable and were removed immediately. The value 'failure' appears about 60% as class label in the remaining data whilst the value 'success' only appears about 40%. We assume that this bit of imbalanced class labels does not affect the accuracy of our classifiers to a high degree. Especially decision trees and random forests are quite robust in this regard [10].

Since it would take too long to train the algorithms on the whole dataset a random sample of twenty-thousand records was selected. Furthermore the columns 'ID', 'name', 'deadline' and 'launched' were dropped. 'ID' was removed since there should not be any effect on the dependent variable 'state'. The attribute 'name' may would have had an effect on the 'state' variable, but it would be too complicated to convert this variable, so that a decision tree could make sense out of it. The attributes 'deadline' and 'launched' was removed since it were time-series variables and are also too complicated to transform.

Decision trees and random forests need numerical values for training and evaluation [6]. So the last step of the preprocessing process was to transform the non-numerical attributes to numerical ones. Envolved were the columns: 'state', 'category', 'main_category', 'currency' and 'country'

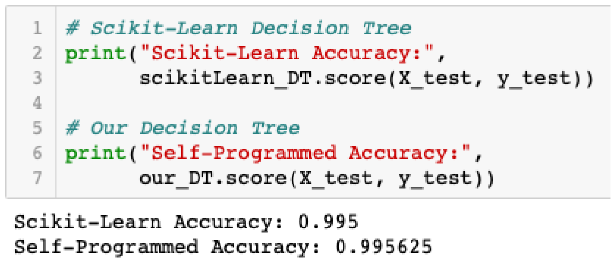
The data is split into a train, validation and test dataset, since the amount of data for training and evaluation is enough to not use cross-validation [11].

5 Results

5.1 Decision Tree Accuracy

This section compares the accuracy of the self programmed decision tree with the accuracy of the decision tree classifier from the scikit-learn machine learning library.

We assumed that the accuracy of our decision tree is somewhat comparable to the accuracy of the decision tree provided from scikit-learn, since we used the same specifications as described in the scikit-learn DecisionTreeClassifier documentation [8]. This assumption was correct and our classifier even slightly outperformed the accuracy of scikit-learn's decision tree on the untouched test dataset. This can be seen in Figure 1.



```
# Scikit-Learn Decision Tree
print("Scikit-Learn Accuracy:",
      scikitLearn_DT.score(X_test, y_test))

# Our Decision Tree
print("Self-Programmed Accuracy:",
      our_DT.score(X_test, y_test))
```

```
Scikit-Learn Accuracy: 0.995
Self-Programmed Accuracy: 0.995625
```

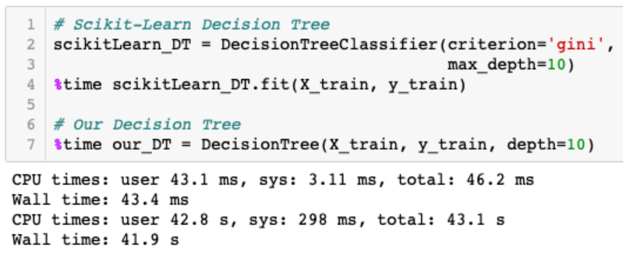
Figure 1: 99.5% versus 99.56% accuracy

5.2 Decision Tree Runtime

This section compares the runtime of the self programmed decision tree with the runtime of the decision tree classifier from the scikit-learn machine learning library.

It was not a surprise that scikti learn was more runtime-efficient, since it is regularly updated and programmed from the top researchers in the field. It definitely was a surprise that it was 1000 times more efficient. The decision tree programmed by ourselves needed

42.9 seconds to fit twenty-thousand records whilst the decision tree provided from scikit-learn only needed 43.1 milliseconds. This can be seen in Figure 2.



```
# Scikit-Learn Decision Tree
scikitLearn_DT = DecisionTreeClassifier(criterion='gini',
                                        max_depth=10)
%time scikitLearn_DT.fit(X_train, y_train)

# Our Decision Tree
%time our_DT = DecisionTree(X_train, y_train, depth=10)
```

```
CPU times: user 43.1 ms, sys: 3.11 ms, total: 46.2 ms
Wall time: 43.4 ms
CPU times: user 42.8 s, sys: 298 ms, total: 43.1 s
Wall time: 41.9 s
```

Figure 2: 43.1ms versus 42900ms runtime

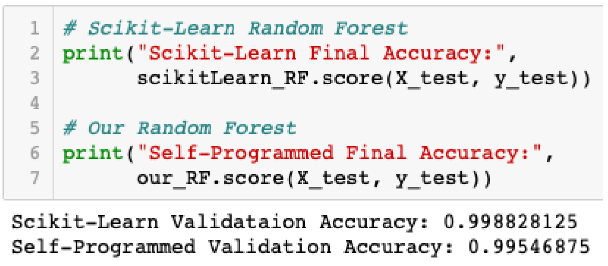
Further profiling allowed a closer look into the different process-times and showed that most of the runtime (14 seconds) is caused by the 'Splitrule' class. It seems like 'numpy.ndarrays' objects are hard to sort and is somehow better handled by scikit-learn. Lots of time is also caused by the functions attempt_split() from the class 'Splitrule' and unique() from numpy.

5.3 Random Forest Accuracy

This section compares the accuracy of the self programmed random forest with the accuracy of the random forest classifier from the scikit-learn machine learning library.

We assumed that the accuracy of our algorithm to be somewhat comparable to the accuracy of the random forest provided from scikit-learn. We did not expect it to exceed the accuracy of the scikit-learn algorithm since we used slightly simplified specifications as described in the scikit-learn RandomForestClassifier documentation [9]. For instance does the self programmed random forest give every tree an equal weight, whilst scikit-learn's algorithm has some additional weights for each decision tree. Our algorithm also strictly uses random 30% of the training data for each decision tree.

The assumption that we manage a comparable accuracy was correct and our classifier was slightly outperformed by the accuracy of scikit-learn's random forest. This was tested on the untouched test dataset and can be seen in Figure 3.



```
# Scikit-Learn Random Forest
print("Scikit-Learn Final Accuracy:",
      scikitLearn_RF.score(X_test, y_test))

# Our Random Forest
print("Self-Programmed Final Accuracy:",
      our_RF.score(X_test, y_test))
```

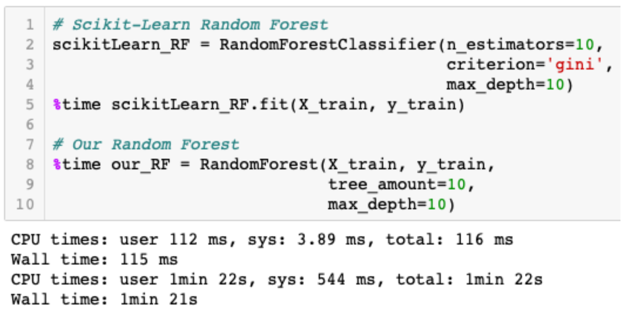
```
Scikit-Learn Validataion Accuracy: 0.998828125
Self-Programmed Validation Accuracy: 0.99546875
```

Figure 3: 99.88% versus 99.55% accuracy

5.3.1 Random Forest Runtime

This section compares the runtime of the self programmed random forest with the runtime of the random forest classifier from the scikit-learn machine learning library.

Again it was not a surprise that scikti learn was more runtime-efficient, and after testing our decision tree we expected the random forest to be about the same magnitude slower. The random forest programmed by ourselves needed 81 seconds to fit twenty-thousand records whilst the decision tree provided from scikit-learn only needed 115 milliseconds. This can be seen in Figure 4.



```
# Scikit-Learn Random Forest
scikitLearn_RF = RandomForestClassifier(n_estimators=10,
                                        criterion='gini',
                                        max_depth=10)
%time scikitLearn_RF.fit(X_train, y_train)

# Our Random Forest
%time our_RF = RandomForest(X_train, y_train,
                            tree_amount=10,
                            max_depth=10)
```

```
CPU times: user 112 ms, sys: 3.89 ms, total: 116 ms
Wall time: 115 ms
CPU times: user 1min 22s, sys: 544 ms, total: 1min 22s
Wall time: 1min 21s
```

Figure 4: 115ms versus 81000ms runtime

Further profiling allowed a closer look into the different process-times and showed that most of the runtime is caused by the same three functions as with our decision tree in section 5.1.2. Sorting 'numpy.ndarrays', the functions attempt_split() from the class 'Splitrule' and unique() from numpy need all about seventeen seconds in the training process.

6 Discussion

It is not quite sure to us why our random forest could not outperform the accuracy of our decision tree on the test dataset. One possible explanation we thought of could be that our decision tree did not overfit the training data. So a random forest could not improve the accuracy since it tries to generalize the overfitted prediction. Nevertheless we are welcoming all possible explanations and different conclusions about this problem.

Possible future direction would probably start with optimizing the time intensive parts described in the runtime sections. Furthermore an inclusion of the time series attributes we dropped out when preprocessing the data could yield to even better results. Then someone also have to take care of splitting the train, validation and test set appropriately.

Word embedding could interpret the 'name' attribute, so that maybe some correlation with the naming and the 'state' variable can be detected. Someone may also be interested in the strongest attributes for predicting the 'state' variable, in that case compare the variances explained by each attribute.

Last but definitely not least someone could try to use the pytorch library instead of numpy since it has GPU support and could run many more decision trees in parallel.

7 Conclusions

It was nice to see how optimized modern machine learning libraries as scikit-learn are. In our opinion it is not imaginable how someone can possibly program a decision tree which fits 20000 data records with 10 attributes in less than 100ms. Additionally it easily exceeds an accuracy of 99%.

On the other hand it was nice to see that our self programmed algorithms could compete with scikit-learns algorithms with regard to accuracy.

References

[1] https://www.kdnuggets.com/2015/12/harasymiv-lessons-kaggle-machine-learning.html

[2] https://www.kaggle.com/kemical/kickstarter-projects#ks-projects-201801.csv

[3] A Survay to Decision Tree Classifier Methodology, Authors: S. Rasoul Safavian and David Landgrebe

[4] Interpretable Hierarchical Clustering by Constructing an Unsupervised Decision Tree, Authors: Jayanta Basak and Raghu Krishnapuram

[5] Topological Models for Prediction of Pharmacokinetic Parameters of Cephalosporins using Random Forest, Decision Tree and Moving Average Analysis, Authors: Harish Dureja, Sunil Gupta and Anil Kumar Madan

[6] Classification and Regression by randomForest, Authors: Andy Liaw and Matthew Wiener

[7] Unsupervised Learning With Random Forest Predictors, Authors: Tao Shi and Steve Horvath

[8] https://scikit-learn.org/stable/modules/generated/sklearn.tree.Deci

[9] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble

[10] An Empirical Study of Learning from Imbalanced Data Using Random Forest, Authors: Taghi M. Khoshgoftaar, Moiz Golawala and Jason Van Hulse

[11] http://guidetodatamining.com

[12] Fifty Years of Classification and Regression Trees, Author: Wei-Yin Loh

[13] Random forests: from early developments to recent advancements, Authors: Khaled Fawagreh, Mohamed Medhat Gaber and Eyad Elyan

[14] Towards self-optimised random forests, Authors: Bader-El-Den, M., & Gaber, M. (2012, November 1215). In T. Huang, Z. Zeng, C. Li, & C.-S.