Universiteit Leiden
The Netherlands

Opleiding Informatica & Economie

Accessibility through Acceleration: A Fast and Fair Random Forest Algorithm

Sander Hesselink

Supervisors:
António Pereira Barata & Frank Takes

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

19/12/2023

Abstract

Machine learning algorithms have made their way into critical decision-making processes. However, concerns have arisen regarding the bias that these algorithms can exhibit, and the impact these biased decisions can have on human lives. This has lead to an academic discussion about measures of algorithmic fairness. In this thesis, we consider an existing fair random forest algorithm, with the aim to make it more accessible to the scientific community. Specifically, we implement the algorithm in C++ to achieve lower runtimes than the existing code in Python. Our results show that the C++ implementation runs faster than the benchmark Python version. However, the exact ratio depends heavily on the size and nature of the data. At best, our optimized code runs 3 to 4 times faster, at worst, it provides barely any to no speedup at all.

Contents

1 Introduction

Automated decision making (ADM) [1] has become a powerful tool to deal with the colossal amounts of data produced each day. Usually powered by machine learning [2], these algorithms can take input and form the requested output without the need for a human in the loop. This does not necessarily imply that the decisions made by ADM are executed automatically. An algorithms used for automatic recommendations to a human agent might still be considered ADM.
However, machine learning, and thus by extension ADM, knows a number of potential issues. These systems tend to replicate any bias found in the data they are learning from, be it a sample bias or a much more deep rooted cultural or historical bias. The search for solutions to this issue has given rise to the field of algorithmic fairness.

News about bias in algorithms have alarmed many governments, and the regulation of ADM, and more broadly Artificial Intelligence, has become an item on the political agenda. Take for example Article 22 of the EU General Data Protection Regulation (GDPR) [3], which protects EU citizens from being subject to fully automated decision making when this could significantly affect the individual.
On the other hand, many parties in the public sector can stand to benefit greatly from ADM. For example, in the health sector, ADM may assist medical professionals with suggestions for diagnoses or treatment [4]. Sometimes, the advantage can even be attributed to algorithmic fairness itself, as government inspectorates may be able to utilize fair algorithms to be able to execute predictive inspections while correcting for bias [5].

In this thesis, we will consider a specific Fair Random Forest algorithm [6]. This algorithm is currently considered insufficiently accessible due to its long runtimes. Making an effort to solve this, we will port this algorithm from Python, a high-level language, to C++, a language on a much lower level. In theory, manipulating the data on this lower level should lead to a speedup.
We formulate the following problem statement.

How effective is implementing an existing Python Random Forest algorithm in C++ at reducing its runtime?

We further subdivide this into the following research questions:

- **RQ1**: How effective is implementing an existing Python Decision Tree algorithm in C++ at reducing its runtime?
- **RQ2**: How do the characteristics of the training data influence the runtime of a Decision Tree algorithm in Python and C++?
- **RQ3**: How does parallelism affect the runtime of a Random Forest in Python and C++?

1.1 Thesis overview

Section 2 of this thesis contains preliminary concepts and definitions. Section 3 will go over the relevant works in literature. The most important concept from the literature, the Fair Decision Tree Classifier using SCAFF, as well as its implementation, will be discussed in more detail in

Section 4. In Section 5, the methodology for answering the research questions of this thesis will be discussed. Section 6 will cover the setup of the performed experiments, followed by Section 7 detailing the results. Discussion of both the experimental results and the code delivered for this thesis is contained in Section 8. Lastly, the conclusions and suggestions for further research can be found in Section 9.
This thesis was written as part of the Bachelor Informatica & Economie at the Leiden Institute of Advanced Computer Science (LIACS) of Leiden University, and was supervised by dr. A. Pereira Barata and dr. F. W. Takes.
This thesis was written as part of an internship at ID-lab of the the Human Environment and Transport Inspectorate of the Netherlands.

2 Preliminaries

This section will go over a few concepts from the field of machine learning the reader is expected to be familiar with (Section 2.1), as well as introduce certain notations that will be used throughout the thesis (Section 2.2).

2.1 Definitions

- **Decision Tree Classifier**: A machine learning algorithm used for classification tasks. Roughly speaking, the algorithm learns by iteratively splitting the data on a specific value of a specific feature. It generates a score for every possible split, and chooses the one with the highest score. Then, the two resulting segments are split again themselves, and so on, until a stop condition is met, creating a binary tree in the process. The scoring system, also called splitting criterion, and stop conditions may differ per implementation of the algorithm. When predicting from unseen data, the algorithm walks through the tree, and compares the values of the new data to the splitting values, to find the leaf this new input would fall into. Each leaf then corresponds to a class prediction probability, based on the distribution of the samples that ended up in it during the learning process.

- **Random Forest Classifier**: A collection of Decision Tree Classifiers. Each individual tree is only given a part of the data as input to prevent overfitting. The prediction is taken as an average of the prediction of the individual trees. This method of grouping predictors together to achieve better results is also called bagging.

- **AUC**: The area under the receiver operating characteristic curve (abbreviated to ROC AUC, or just AUC), a common performance metric for classifiers. The AUC is valued between 0 and 1, where 0 means the classifier has mislabeled every instance, and 1 means predictions produced by the classifier are perfectly correct. In the case of binary predictions, a fully wrong prediction can be trivially turned into a perfectly correct one by simply flipping the labels. As such, the poorest AUC score a classifier can obtain is 0.5, which indicates that predictions are made at random.

- **Demographic Parity**: A common measure of algorithmic fairness. Demographic parity aims to equalize the distribution of predictions across groups. For example, if the ratio of positive

predictions for one sensitive group is 50%, then this ratio should be 50% for all other groups as well. As a result, the distribution of sensitive groups among those who have received a positive prediction should be equal to the distribution of sensitive groups in the population.

2.2 Notation

- X: the feature space. We use the symbol X for both the input during the learning process, as well as for the input of the predicting process.
- S: the sensitive attributes. The prediction of a fair algorithm should not be influenced by these. This means the algorithm should be incapable of predicting S (i.e. obtain a poor performance score).
- Y: the true class labels. A classifier tries to learn the relationship between X and Y to form a mapping, so it can predict the Y values from an unseen X. In general, Y can be multiclass or multilabel, but in this thesis we only consider binary classification.
- $\hat{Y}$: the predicted class labels returned by the algorithm. In a real classification task, these are predictions, but in an experimental setup, these can be compared to the true Y to obtain performance scores.

Note that we use all of these symbols refer to collections. For example, the feature space, sensitive attribute, true class label, and predicted class label of a single sample i would be X_i, S_i, Y_i, and $\hat{Y}_i$, respectively.

3 Related Work

The works in literature related to this thesis can be split into two categories: Algorithmic Fairness (Section 3.1) and Comparison of Programming Languages (Section 3.2).

3.1 Algorithmic Fairness

In algorithmic fairness literature, the problem is usually a variation of the following:

Given a feature space X, a sensitive attribute S, and a true class label Y, find a classifier that:

(1) Returns a class prediction $\hat{Y}$ with high accuracy

(2) Does not discriminate based on S

In its most abstract form, this problem statement can be applied to any X, Y, and S. However, when discussing fairness, the most common, and perhaps most evocative, use cases are those of classifying human individuals in situations where social or cultural bias is present [7]. Examples of this are college applications [8], hiring decisions [9], and the now infamous case of the COMPAS

recidivism algorithm [10], where an algorithm used by judges in the US to predict whether a defendant would be arrested again was argued to discriminate based on race.

The Fair Decision Tree Algorithm central to this thesis [6] aims to achieve both objectives of the problem by using a splitting criterion which weighs both a performance score and a fairness score. This approach of implementing algorithmic fairness into the learning process is called in-processing [11]. If the performance score and the fairness score are both weighed equally, the algorithm should equalize the distribution of predictions across groups so demographic parity is achieved [12]. Additionally, the algorithm is threshold independent (see Section 4.1), meaning it is an application of strong demographic parity [13].

The last concept from algorithmic fairness literature relevant to this thesis is the performance-fairness trade-off. A fairness-unaware classifier will only optimize for objective (1) of the problem, and therefore any model that deviates from this will by definition have a lower classification performance. In other words, a fair classifier needs to sacrifice some accuracy to be able to gain fairness [14].

3.2 Comparison of Programming Languages

In literature, there are many different works comparing many different programming languages by many different metrics. This section will focus on works comparing Python and C++ in terms of runtime, as this is the comparison relavent to this thesis.

Zehra et al. [15] report both time and memory usage of Python and C++ implementations of a number of algorithms. They find that in the average case scenario, the C++ implementations are all faster than the Python ones, but the ratio depends heavily on the kind of algorithm. Of the different algorithms the work compares, binary search shows the least difference between Python and C++, with runtimes being almost equal, and bubble sort shows the largest difference, with C++ being about one hundred times faster. The main reason behind the faster runtimes of C++, according to this research, is the language's use of static typing, as opposed to Python's dynamic typing. Other reasons include C++ being a compiled language and it using dynamic memory allocation, as opposed to a garbage collection system.
Arboleda et al. [16] compare runtimes of Python and C++ on matrix operations, specifically in regards to parallelism. The work details generally faster runtimes for C++, except for instances where a native NumPy function can be used. It also reports that the C++ parallel computing library OpenMP generally outperformed Python's multiprocessing module.

Most works in this field do not limit themselves to just Python and C++, however, and instead aim to compare a larger number of languages. Prechelt [17] investigates the performance of a large number of programs submitted by other people in seven different languages, in a sort of puzzle-like programming exercise. The results report a large variety between runtimes for programs written by different people. When limiting our scope to the results for Python and C++, it is noticeable that the fastest C++ program is faster than the fastest Python program, but the slowest Python program runs faster than the slowest C++ program. Other insights of this research include that C++ is significantly faster than Python when it comes to purely loading the data, that C++ programs

are significantly longer than Python programs, and that they generally take more time to develop.

When running on a suite of benchmark tests, Lion et al. [18] find that Python is on average approximately thirty times slower than C++. However, the variance reported in the results is large. For a merge sort algorithm, the Python code is more than one hundred times slower than the C++ code, whereas for a series of server requests, the runtimes are effectively equal.
Pascal and Krzysztof [19] compare the two languages, alongside the language Go, on the n-queens problem. While the work reports that C++ is significantly faster than Python, it also looks at a specific package for Python to speed up execution time, called Numba. The runtime of the Python code using Numba is competitive with C++.
Lastly, some research compares runtimes for algorithms used in specific fields, such as Fourment and Gillings [20] in the field of bioinformatics, and Aruoba and Fernández-Villaverde [21] in macroeconomics. Both works report significantly faster runtimes for C++ compared to Python.

From the works discussed in this section, we can conclude that C++ is generally faster than Python, but the type of algorithm being implemented is a heavy factor in how much speedup it can offer. Critically however, none of the works in scientific literature investigate any machine learning algorithms.

4 Background

In this section, we will inspect the Fair Decision Tree Classifier proposed by Barata et al. [6] in further detail. First, by breaking down the theory and goals behind the splitting criterion (Section 4.1), and then by assessing its implementation in Python (Section 4.2). We will look specifically at the implementation for a single Decision Tree. In the case of a Random Forest, the described actions are performed for each tree in the forest.

4.1 Splitting Criterion AUC For Fairness

Aside from the general algorithmic fairness problem provided in Section 3.1, the splitting criterion proposed in Barata et al. [6] aims to achieve two additional goals: (1) control over the performance-fairness trade-off, and (2) threshold independence. In this section, we will define the SCAFF, and highlight how these goals are achieved.
The formula for the SCAFF is given as follows:

$$SCAFF = (1 - \Theta) \cdot |2 \cdot AUC(Y) - 1| - \Theta \cdot |2 \cdot AUC(S) - 1|$$

Here, the $AUC(Y)$ and the $AUC(S)$ are the AUC scores calculated with respect to both Y and S. Calculating the AUC in respect to Y acts as a measure of classification accuracy, whereas calculating the AUC with respect to S is a metric for fairness. As explained in Section 2, the poorest score a classifier an obtain is not 0, but rather 0.5. Both AUC scores are therefore adjusted for this by multiplying by 2, decrementing by 1, and then taking the absolute value.

In the SCAFF, the AUC is calculated as follows:

$$AUC = \frac{1 + TPR - FPR}{2}$$

Where the TPR and the FPR stand for the true positive rate and the false positive rate, respectively.

The first goal of the SCAFF is to allow the end user to decide over the trade-off between fairness and performance. This is achieved by adding the Θ (Theta) to the formula, a tuneable parameter that represents the importance of the fairness metric compared to the classification metric. A Theta of 0 means that the fairness metric is not considered at all, resulting in a traditional fairness unaware classifier. On the other end, a Theta of 0.5 means the classification metric and fairness metric are both weighed equally, resulting in demographic parity.

In a classification task, the model returns a probability of the class. This can then be turned into a binary prediction by comparing it to a certain threshold. Splitting criteria often also require an internal threshold to obtain a score. The second goal of the SCAFF is to achieve threshold independence, which is to say it should not rely on an internal threshold, to provide the end user more flexibility in deciding the appropriate threshold later on in the process.
To calculate a true positive and false positive rate, necessary for the AUC, a binary prediction is required. Fortunately, the Decision Tree algorithm already splits the data, allowing the algorithm to assign positive predictions to one part of the split, and negative predictions to the other. Since the SCAFF balances the AUC score symmetrically around 0.5, it does not matter which split the positive prediction is assigned to. This way, an internal binary prediction, and thus a split score, can be obtained without the need for an internal threshold.

4.2 Implementation of Python Benchmark

The benchmark Python code is written to be compatible with certain pipelines of the popular Python machine learning package scikit-learn [22]. This means, among other things, that its functionality has been split in two.
The `fit()` function builds a decision tree from the input data, and takes X, Y and S as parameters. In a traditional machine learning setup with a training data set and a test data set, this function should be called on the training set. While the implementation only supports single-label binary input for Y, S can be multilabel and multiclass. To deal with this, the algorithm calculates the AUC of every class of every label of S separately, and returns the highest score of these to be used in the calculation of the SCAFF.
As described in Section 4.1, a classification algorithm can return either a predicted class probability or a binary class prediction $\hat{Y}$. Because of this, the implementation comes with two functions for prediction. The `predict_proba()` function takes only the X as parameter, and returns the predicted class probability for each sample. To obtain $\hat{Y}$, the `predict()` function can be called instead. In a traditional machine learning setup, these functions should be called on the test set, as well as the actual prediction task. When predicting classes in a forest, the option is given to use either the average of the class predictions or the average of the probability predictions of every tree. Internally, the `predict()` function only contains a call to the `predict_proba()` function and a trivial relational comparison of the received probability to a threshold. Due to this, the runtimes of

these two functions are equal on the timescale we measure. For consistency, we will only consider the `predict_proba()` function for the rest of this thesis.

The benchmark Python code was taken from its GitHub repository [23]. It contains several features which were not relevant for this research, such as tree pruning and an additional splitting criterion. For the sake of fair comparison, all experiments were run on a version of the Python code where these were omitted as well.

5 Methodology

This section will discuss a number of implementation details of the C++ code developed for this thesis. We start with the guiding principles that were followed (Section 5.1). Then, we will zoom in on three key differences in implementations between the Python Benchmark code and our C++ code, namely loading the data (Section 5.2), optimizations that have been made (Section 5.3), and the implementation of parallelism (Section 5.4).

5.1 Guiding Principles

The C++ code written for this research is meant as an optimized substitute for the benchmark Python code. For one, this means that it has to fit all of the same requirements. When given the same input, the two implementations should provide the exact same output, save for places were randomness is a factor. To achieve this, it was decided to build the C++ code bottom-up, staying as close to the original Python code as possible, while optimizing where necessary.
In order for the code to be a true substitute, it should still be callable from Python. To achieve this, the pybind11 package was used, which allows C++ code to be compiled as a dynamically linked `.pyd` file. The advantage of this package is that it does not require the code itself to be changed, as opposed to other similar packages, like CPython and Cython. This means that the written C++ code could also be used in a pure C++ environment.
The code developed for this thesis is available on GitHub [24]. Due to the complexity of packaging compiled C++ code, converting the code into an actual Python package falls outside the scope of this thesis.

5.2 Data

In the C++ implementation, the data of both the feature space X and the sensitive attributes S are stored in a custom class called `DataMatrix`. This section will elaborate on the two goals that this class achieves: (1) heterogeneity and (2) slicing.

In Python, containers like lists and NumPy arrays are *heterogeneous*, which means that a single container can store variables of multiple datatypes. This is useful in machine learning, since an feature in a dataset may hold numerical information, in the form of integers or floating point numbers, or categorical information, in the form of booleans or strings. C++, on the other hand, has no true heterogeneous containers the way Python does.

The algorithm we are aiming to implement is able to handle both numerical and categorical information, and thus, a way to represent the heterogeneous data in C++ is required. To achieve this, our code uses the `variant` class, which is part of the C++ standard library. This is a specific implementation of what in programming is often called a union. It can be considered as a container that can store one variable, which can be any one of a number of predefined types. By wrapping this in a more traditional C++ container, like a vector, the impression of a heterogeneous container can be created.
Another advantage of Python, when dealing with machine learning datasets, is that it allows for a wide variety of slicing and bitmasking operations, which are often not available for C++ containers. While there are libraries available with more advanced containers, the choice was made to apply the C++ standard library wherever possible. For this problem, the `valarray` class was used, which is a container that allows for bitmasking and slicing on rows. To be able to also slice columns, the `DataMatrix` class contains a one-dimensional `valarray` structure, and several helper functions using slicing logic to select the right samples.

5.3 Optimizations

Of the optimizations that were made to the C++ code, there are two that stand out in particular, and will be discussed in the following sections. These are the encoding of string data types (Section 5.3.1), and the simplification of calculations (Section 5.3.2).

5.3.1 Encoding

To be able to perform calculations on categorical data in the form of strings, the benchmark Python code uses a one-hot encoding function from scikit-learn [22]. While one-hot encoding is certainly possible in C++, our manual implementation could not compete with scikit-learn in terms of runtime. Additionally, one-hot encoding decreases the number of values, but in return increases the number of features, which was found to further increase runtime in C++. For these reasons, our implementation uses a hash encoder. This is computationally a lot cheaper than one-hot encoding, and only changes the data entries themselves, not the shape of the data as a whole.
While hashing does represent categorical data as numbers, their numerical order does not have any meaning. This means the C++ code needs to store which columns of the data contain categorical information, and treat these slightly differently when calculating split scores.

Interestingly, the algorithm itself already contains a form of encoding, which makes it able to fit on string datatypes directly. However, in C++, our computations run slower on string datatypes than on numerical values, so while hash encoding is not strictly necessary, it is preferred. This does allow us to introduce an additional boolean parameter, called `hash_values`. By setting this to false, the user can turn off the hashing of the data, causing a slightly longer runtime. However, this has the benefit that the tree itself becomes human-interpretable. By including this parameter, we therefore allow the end user to decide over this trade-off between speed and explainability.

5.3.2 Calculations

The benchmark Python code uses NumPy functions for calculations wherever possible. While substitutes for a number of these functions are available in C++, either as functions of the `valarray`

class or in the `algorithms` library, it was found that these were inflexible, and the workarounds necessary often lacked the level of optimization NumPy possesses. Because of this, the calculations for the split scores were decomposed, and substituted with simpler arithmetic wherever possible.

Listing 1 displays a code snippet for the calculation of the AUC of Y in Python. It shows how the true positive and false positive rates are calculated in one line each by chaining a number of NumPy functions together. Listing 2 shows how the same calculations have been implemented in C++. Here, the different parts of the chain have been mostly separated. This allows for the variables `neg_sum` and `total_neg`, which together make up the false positive rate, to be calculated with arithmetic, instead of the functions necessary for their positive counterparts. The downside of the decomposition is the larger number of intermediate variables necessary. This is handled by wrapping the whole calculation inside a lambda function, ensuring a small scope for these variables.

```
tpr_y = (y & split_indexes).sum() / (y & total_indexes).sum()
fpr_y = (y_flipped & split_indexes).sum() / (y_flipped & total_indexes).sum()
auc_y = (1 + tpr_y - fpr_y) / 2
auc_y = max(auc_y, 1 - auc_y)
```

Listing 1: Calculation of AUC(Y) in Python. Some variable names have been changed to improve out-of-context readability.

```
const float auc_y = [&]() {
    const float pos_sum = boolsum(y & split_indexes);
    const float neg_sum = splitsize - pos_sum;
    const float total_pos = boolsum(y & total_indexes);
    const float total_neg = totalsize - total_pos;

    const float tpr_y = pos_sum / total_pos;
    const float fpr_y = neg_sum / total_neg;

    const float auc = (1.0 + tpr_y - fpr_y) / 2.0;
    return max(auc, 1.0 - auc);
}();
```

Listing 2: Calculation of AUC(Y) in C++. Some variable names have been changed to improve out-of-context readability.

For the sake of comparison, two additional versions of the Python benchmark code were created, with each one implementing one of the optimizations mentioned above. This leads to a total of four different tree implementations: a Python benchmark version, an optimized C++ version, and two partially optimized Python versions. Note that, unlike in C++, optimizing the encoding in Python did not mean using hash encoding, but rather taking the additional encoding out altogether.

5.4 Parallelism

Since the trees in a Random Forest are independent from each other, they can all fit at the same time. As such, the functions of a Random Forest can be easily parallelized. However, Python and C++ implement parallel computing differently.

The most common parallelization method in programming is multithreading, which is supported by multiple libraries in C++. For this project, the basic functionality of the standard C++ concurrency support library was sufficient. The core idea is that the program creates a number of threads, which can execute code simultaneously and all share the same memory. This can lead to problems if multiple threads try to access the same variable. To deal with this, a lock, sometimes called a mutex, can be used to ensure that a variable is only accessed by one thread at a time.
While Python also provides the ability to create threads, these cannot be used for parallelization. This is because of the Global Interpreter Lock, a lock on the interpreter itself, which ensures that only one thread can execute code at a time. True parallelization in Python is instead achieved through the `multiprocessing` module, allowing the creation of subprocesses, which each have their own memory. Consequently, these subprocesses don't suffer from memory issues, and can work around the Global Interpreter Lock. The drawback attached to this additional memory allocation is that a subprocess generally takes longer to create than a thread.
The benchmark Python code uses this `multiprocessing` module to parallelize the training and prediction of the Python trees. For this research, we have created two different implementations of the Random Forest; one in pure C++, which uses threads, and one hybrid implementation, which uses our trees in C++, but calls these from Python subprocesses.

6 Setup

This section will discuss the technical specifications of the experiments (Section 6.1), the different kinds of data used (Section 6.2), and the different variables we are interested in assessing (Section 6.3).

6.1 Technical Specifications

All of the experiments were run on a Lenovo IdeaPad 5 with Windows 10 and 16 GB of RAM. This device has an Intel i7-1065G7 processor with 4 cores, 8 threads and a maximum frequency of 3.90 GHz. The C++ code was compiled with the Intel C++ compiler version 19.2, as this compiler is known to have effective optimizations for the `valarray` class. On the Python side, a CPython 3.10.12 interpreter was used.
All time measurements are done in Python, using the `perf_counter()` function from the `time` module, while calling the C++ code using pybind11 as described in Section 5.1. All tree and forest classes are initialized using default parameters, unless specified otherwise.

6.2 Data

In the experiments, we will be using two kinds of data: synthetic datasets (Section 6.2.1) and fairness benchmark datasets (Section 6.2.2).

6.2.1 Synthetic Datasets

To be able to explore the different parameters in a controlled environment, we use artificial numerical datasets, generated by the sci_kit-learn `make_classification()` function [22]. For each experiment, we generate five datasets with different random states, to avoid results biased towards a specific

dataset. When we are interested in exactness of individual results, rather than the overall trend, we also employ 5-fold cross-validation on each of these datasets to achieve higher accuracy.
We limit the synthetic datasets by only using a singular binary S. This allows us to introduce the additional parameter S *shift*, which represents how strongly S and Y are correlated. Specifically, since the sci_kit-learn function only returns an X and a Y, we create S as a mutation of Y, where each label is given a chance of S shift to be flipped. Hence, an S shift of 0 means that Y and S are the same, and a value of 0.5 means that S is entirely random, and not correlated to Y at all. The intuition is that a lower value for S shift leads to a more difficult classification task.

6.2.2 Real World Datasets

To verify our results on the synthetic datasets, we investigate runtime comparisons on five real world fairness benchmark datasets, taken from a collection gathered by Pessach and Shmuelli [11]. These datasets are described in Table 1. The source paper [11] contains a more detailed description of these datasets. Note however that the number of samples might not always match, as the datasets had to be preprocessed for missingness.

Name (Abbreviation)	Number of samples	Number of features	Sensitive Attributes
Adult (A)	30,913	10	2 binary, 1 ternary
Dutch Census (DC)	37,119	9	1 binary, 1 ternary
German Credit (GC)	982	18	3 binary
Bank Marketing (BM)	45,203	14	1 binary, 1 ternary
College Admissions (CA)	20,510	9	2 binary

Table 1: Description of the real world benchmark datasets used in the experiments.

6.3 Variables

This section will go over the variables we will consider for assessing the validity of our implementation (Section 6.3.1), the runtimes on Decision Trees (Section 6.3.2), and the runtimes on Random Forests (Section 6.3.3).

6.3.1 Quality Assurance

As discussed in Section 5.1, the C++ algorithm should give the same output as the Python code. We validate this by comparing the predictions produced by the different implementations for both trees and forests.
Secondly, to verify the proper implementation of the hyperparameters, we look at the runtimes for two C++ trees with different hyperparameters on the same datasets. Specifically, we assess one tree initialized with default hyperparameters, and one tree initialized with hyperparameters set to overfit as much as possible. A tree that performs every single possible split should make $n \log(n)$ splits, so we expect to see this function when we graph the runtimes.

The last validation necessary is that of S shift. In Section 6.2 we asserted that a lower value of S shift should lead to a more difficult classification task, and in turn a lower classification score. To verify this, we compute the AUC for the predictions made by the Decision Tree for both an S shift of 0.1 and an S shift of 0.5.

6.3.2 Decision Trees

To answer our first and second research questions, we will examine the runtimes of both the `fit()` and `predict_proba()` functions of the different Decision Tree implementations discussed in Section 5.3. We explore how a number of different variables can affect these runtimes. Specifically, we will investigate the effects of the following parameters:

- **Number of samples**: Real world datasets often have large numbers of samples, so it is valuable to compare how the different implementations fare. The use of synthetic datasets allows us to generate datasets with any number of samples.
- **Number of features**: Real world datasets often have large numbers of features, so it is valuable to compare how the different implementations fare. The use of synthetic datasets allows us to generate datasets with any number of features.
- **Number of unique values per feature**: More unique values leads to more possible splits for the algorithm to consider, and presumably a longer runtime. The trees and forest both employ a binning technique to be able to reduce runtime. Therefore, the Number of unique samples per feature can be adjusted by tuning the `n_bins` hyperparameter.
- ***S* shift**: The S shift parameter is an artifact of the synthetic datasets and should correlate with the difficulty of the classification task.

6.3.3 Random Forests

To answer our third research question, we will examine the runtimes of both the `fit()` and `predict_proba()` functions of the different Random Forest implementations discussed in Section 5.4. Here, we will vary the following parameters:

- **Number of samples**: Real world datasets often have large numbers of samples, so it is valuable to compare how the different implementations fare. The use of synthetic datasets allows us to generate datasets with any number of samples.
- **Number of trees per forest** (estimators): Random Forests can vary in how many individual trees they contain. Real world application may utilize upwards of hundreds of trees, so it valuable to compare how the different implementations fare.
- **Number of threads** (jobs): We assess the runtimes of forests both with and without parallelism to investigate its effects. Specifically, we compare a forest utilizing one thread with a forest utilizing all available threads.

7 Results

This section will discuss the results of the experiments described in the previous section. This is split into four parts: Quality Assurance (Section 7.1), results for Decision Trees (Section 7.2), results for Random Forests (Section 7.3), and lastly, results on the real world benchmark datasets (Section 7.4).

7.1 Quality Assurance

When comparing the predictions, we take the predictions by the Python Benchmark as a baseline to compare the predictions of the other implementations to. Table 2 shows the results in two columns: the amount of class predictions that the implementation predicted differently from the Python Benchmark, and the mean difference of the probability predictions.

Implementation	Number of inverted `predict()`	Mean difference `predict_proba()`
Python Benchmark	–	–
Python Encoding Optimization	9	0.00
Python Calculation Optimization	0	0
C++ optimized	10	0.00

Table 2: Prediction differences of the different tree implementations, with the Python Benchmark predictions as a baseline. Predictions were made in the form of 5-fold cross-validation on 5 synthetic datasets of 1000 samples, summing up to a total of 5000 predictions.

The table shows that all versions have very similar prediction, but only the prediction of the Python Calculation Optimization variant perfectly matches the predictions of the Python Benchmark algorithm. The difference in predictions for the other two implementations can however be explained by the factor of randomness. Specifically, the difference occurs when there are two or more possible best splits with the same score. In this case, the algorithm will choose the one it has evaluated last. However, encoding the features may change their order, and as such, only the implementation that uses the exact same encoding as the benchmark produces the exact same results.

Since randomness is a considerably larger factor in Random Forests, we have to set a baseline deviation when comparing predictions. For this, we use the difference in predictions between two Python forests with different random states. The results of comparing forest predictions can be found in Table 3.

Comparison	Number of inverted `predict()`	Mean difference `predict_proba()`
Python Random States	217	0.01
Python – C++	219	0.01
Python – Hybrid	228	0.01

Table 3: Comparisons of prediction of the different forest implementations. Predictions were made in the form of 5-fold cross-validation on 5 synthetic datasets of 1000 samples, summing up to a total of 5000 predictions.

The table shows that while the inter-Python comparison has the least amount of inverted class predictions, the other two comparisons are not significantly higher. As such, the mean differences in probability predictions are indistinguishable from each other.

Figure 1 shows the runtimes for two trees with different hyperparameters.

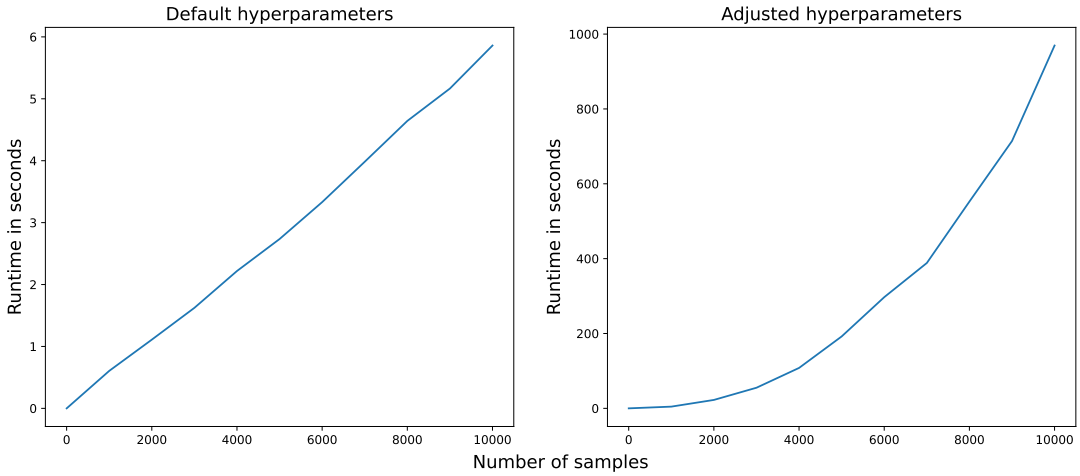


Figure 1: The runtimes of fitting and predicting two different trees on the same synthetic dataset. The adjusted hyperparameters are set to build the largest possible tree, i.e. overfit as much as possible. The dataset used contains 10 features and applied an S shift of 0.3.

While the left graph seems linear, the right graph shows a shape of $n \log(n)$, which lines up with the expectations expressed in Section 6.3.1. Note however not only the shape of the graphs, but also the disparity between the ranges of the vertical axes, which showcases the large impact hyperparameters can have on the runtime.

The results of the validation of S shift are shown in Table 4.

	Number of samples			
Number of features	**1000**		**10000**	
10	S **shift = 0.1**	S **shift = 0.5**	S **shift = 0.1**	S **shift = 0.5**
	0.602	0.688	0.691	0.747
100	S **shift = 0.1**	S **shift = 0.5**	S **shift = 0.1**	S **shift = 0.5**
	0.505	0.516	0.506	0.558

Table 4: AUC scores of the predictions by the fair tree algorithm for different values for S shift. Every cell is the mean value of 25 scores, in the form of 5-fold cross-validation over 5 synthetic datasets.

This table confirms the intuition we have expressed. Every single cell with an S shift of 0.1 shows a lower AUC score than their counterpart with an S shift of 0.5. The variation between results of

different numbers of samples and features can be attributed to the hyperparameters of the Decision Tree class.

7.2 Decision Trees

Table 5 shows the execution times of the different implementations explained in Section 5.3.

Implementation	Total time (Standard Deviation)	`fit()`	`predict_proba()`
Python Benchmark	18.5s (1.2)	18.5s	0.003s
Python Encoding Optimization	17.9s (0.2)	17.9s	0.004s
Python Calculation Optimization	14.6s (0.2)	14.6s	0.002s
C++ Optimized	7.71s (0.2)	7.70s	0.012s

Table 5: Runtimes for different Decision Tree implementations. Values are the mean of 5-fold cross-validation on 5 synthetic datasets, each with 5000 samples, 50 features and an S shift of 0.1.

In the total column, each row shows a lower time than the one above it. The Python benchmark is the slowest implementation, followed by the two partially optimized Python versions, and the C++ shows the fastest time. As briefly mentioned in Section 6.2, the synthetic datasets we use for these experiments only contain numerical values. As such, the Python version with optimized encoding of string variables is effectively equivalent to the benchmark version. Interestingly, this is not reflected in the results, as the Python Encoding Optimization variant shows lower runtimes than the benchmark code.
Comparing the total execution time with the runtimes of the individual functions shows that nearly all of the total time is taken up by the `fit()` function, and as such, the pattern in this column is the same as for the total time.
For predictions, the runtime for C++ is about 4 to 5 times higher than for any of the Python implementations. However, this is still less than a tenth of a second, and less than a hundredth of the prediction time.
While the partially optimized Python versions show some improvements over the benchmark Python code, these results are not the focus of this thesis. Because of this, the rest of this subsection will focus only on the benchmark Python code and the optimized C++ code. The partially optimized versions will appear again in Section 7.4.

Out of the variables described in Section 6.3.2, the ones we will investigate first are the number of samples and the number of features, which will be plot as the two axes of a heatmap. Figure 2 and Figure 3 both contain one such heatmap for the execution times of each language, as well as a heatmap for the difference, and a plot for the relative speedup. The two figures show experiments with the same conditions, except for the value of S shift, which is 0.1 in Figure 2 and 0.5 in Figure 3.

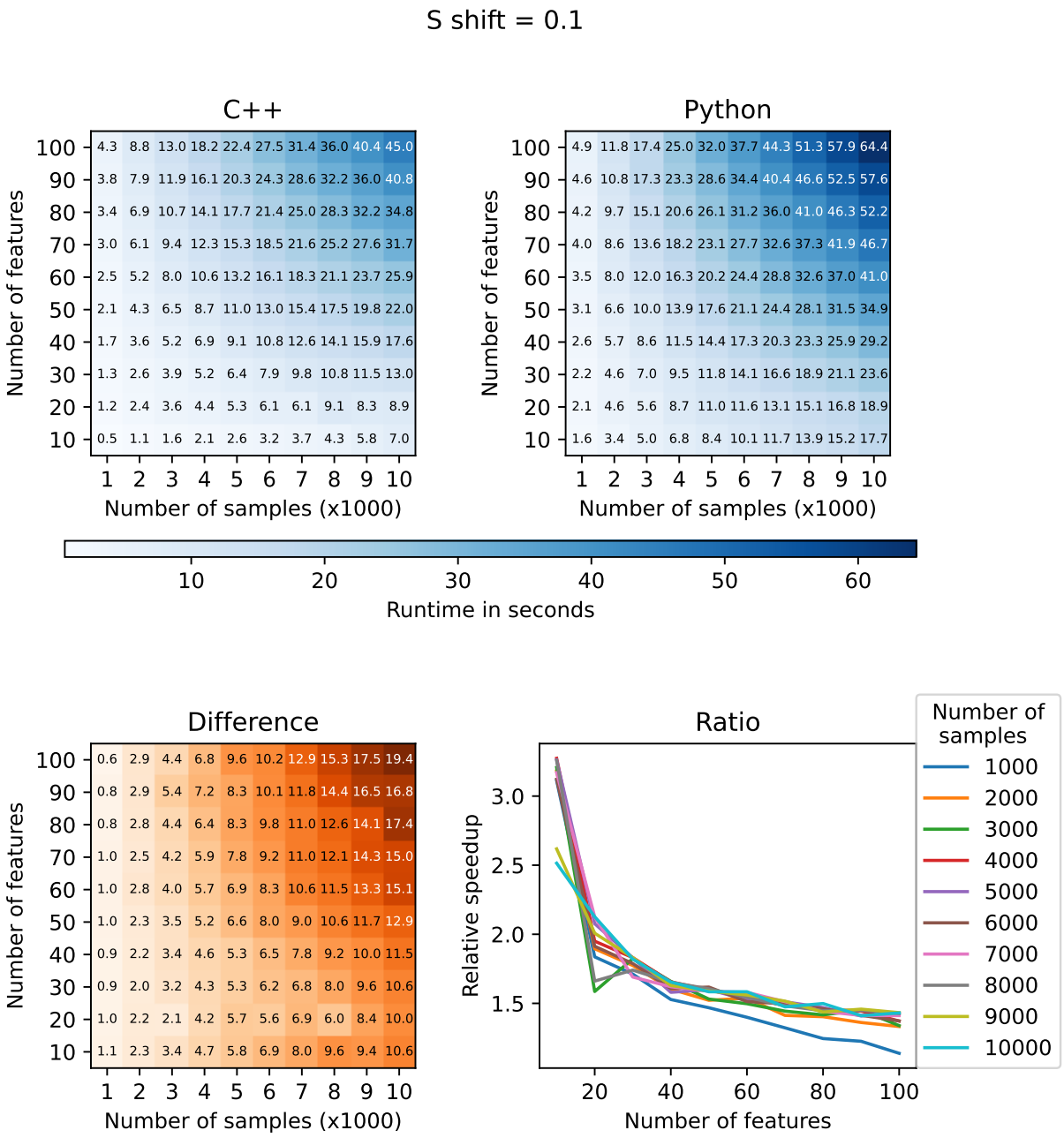


Figure 2: Execution times for the Decision Tree class, with S shift set to 0.1. The values are acquired by running the `fit()` and `predict_proba()` functions consecutively. Each cell is the mean of 5 synthetic datasets, with a single 80/20 train-test split per dataset.

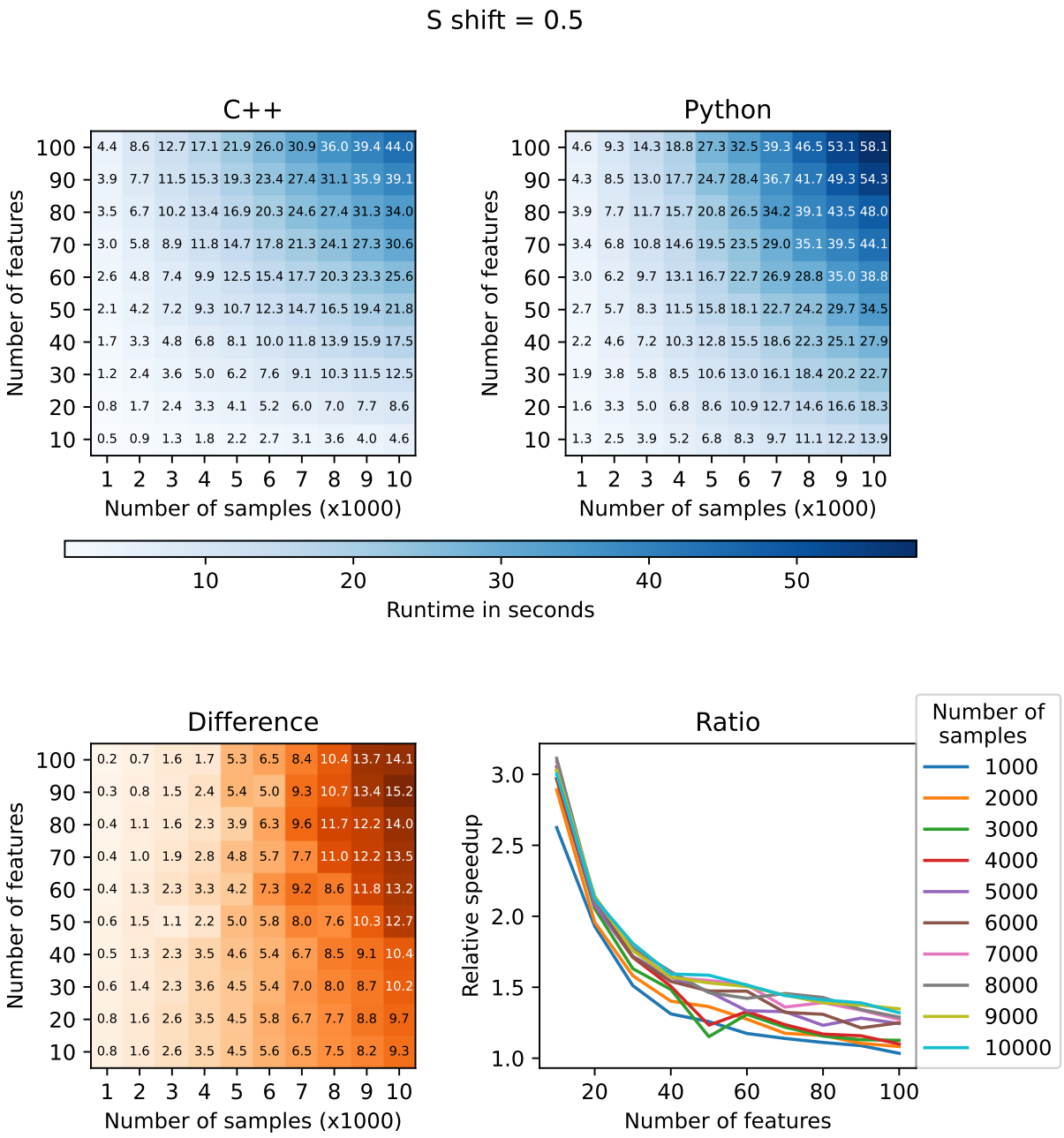


Figure 3: Execution times for the Decision Tree class, with S shift set to 0.5. The values are acquired by running the `fit()` and `predict_proba()` functions consecutively. Each cell is the mean of 5 synthetic datasets, with a single 80/20 train-test split per dataset.

In the figures on the previous two pages, both languages show execution times that increase with both number of samples and number of columns. The complexity twoards both variables would appear linear, but the difference and ratio graphs show us that this is not the case. Specifically, the ratio graph shows a convex, decreasing curve, where a smaller amount of features lead to a much larger relative speedup. While the number of samples does not show a relationship as clear, we can generally say that a larger amount of samples leads to a larger speedup.
Comparing the ratio graphs of the two figures, we can see a number of differences. Not only does the ratio graph in Figure 2 lie slightly higher on the vertical axis compared to Figure 3, the lines are also bundled more closely together, with the exception of the line of 1000 samples. This means that a lower S shift leads to a relatively faster runtime for the C++ version, as well as less influence from the number of samples.

Investigating this further, Figure 4 shows the execution times and ratio for different values of S shift.

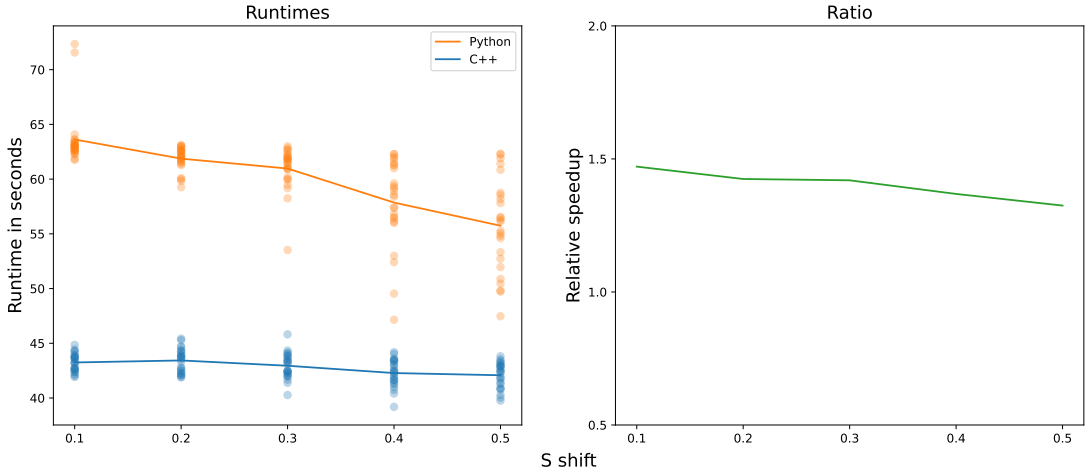


Figure 4: Execution times of both Python and C++ tree implementations for different values of S shift. Run with 5-fold cross-validation on 5 synthetic datasets with 100 features and 10000 samples. The dots are the individual results, the lines represent the mean.

When looking at the means, we see the same pattern as we saw when comparing Figure 2 and Figure 3, interpolated for other values of S shift. The increased difficulty of a lower value of S shift seems to affect the Python implementation more than the C++ code, as evidenced by the stronger downward slope, which in turn leads to a slightly downward sloping ratio curve.
The individual results in the figure add some nuance. While the range of the results for the C++ implementation changes very little, the results for the Python version seem to spread out more as S shift increases. Specifically, the maximum value seems to stay the same, but the range fans out downwards.

Figure 5 shows the runtimes and ratio for different amounts of unique values per feature.

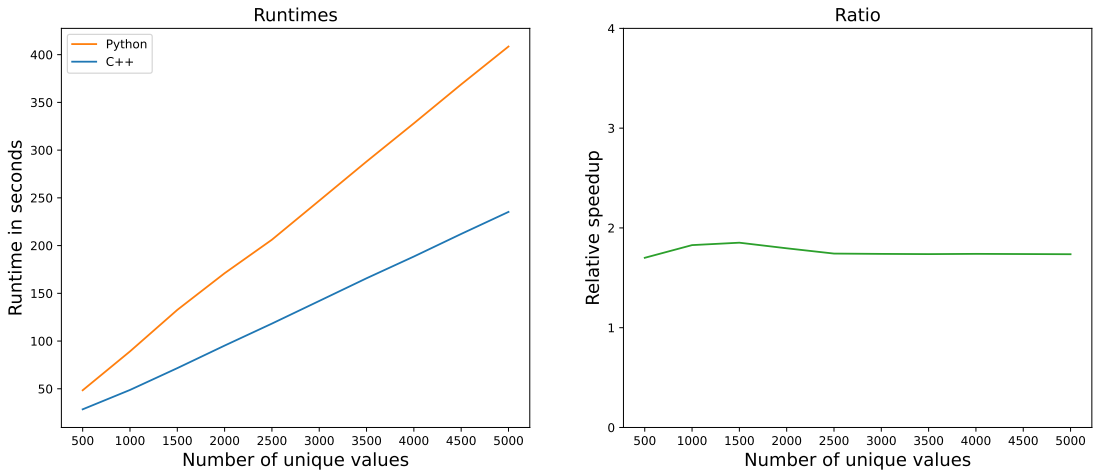
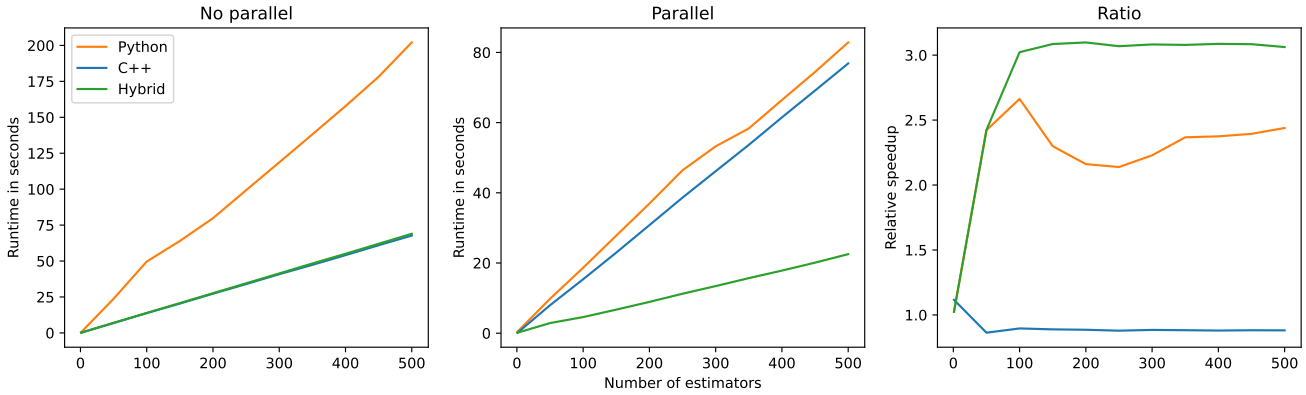


Figure 5: Runtimes of the Decision Tree `fit()` function for different numbers of unique samples per feature. Results are averaged over 5 synthetic datasets with 50 features, 5000 samples and S shift set to 0.1. Since there is no prediction in this experiment, the trees are fit on the entire dataset.

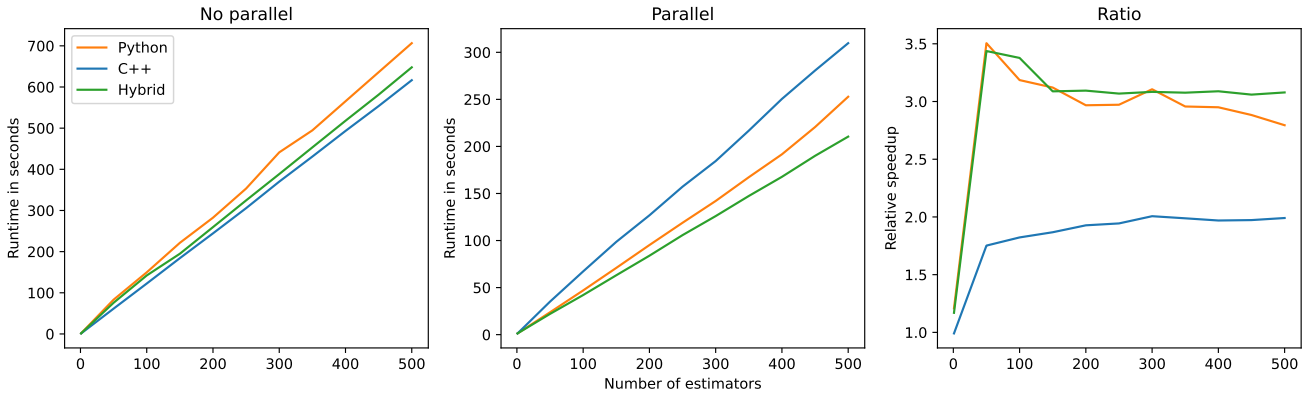
In the figure, both languages show a linear function with a constant slope. This means that while runtimes increase for both languages, the ratio between the implementations stays constant. In this case, the ratio is approximately 1.75, which roughly corresponds with the results for this configuration present in Figure 2.

7.3 Random Forests

Figure 6 shows the runtimes of the `fit()` function of the Random Forest class under different circumstances. In the figure, we mention two different configurations. "No parallel" refers to the situation where only a single thread or subprocess is used, "Parallel" refers to the situation where the program uses all available multiprocessing resources, which on the device the test were run on is 8 threads. Note that the ratio graph is this figure does not compare different implementations, but rather the relative speedup the implementation gains from parallelization.



(a) 1000 samples

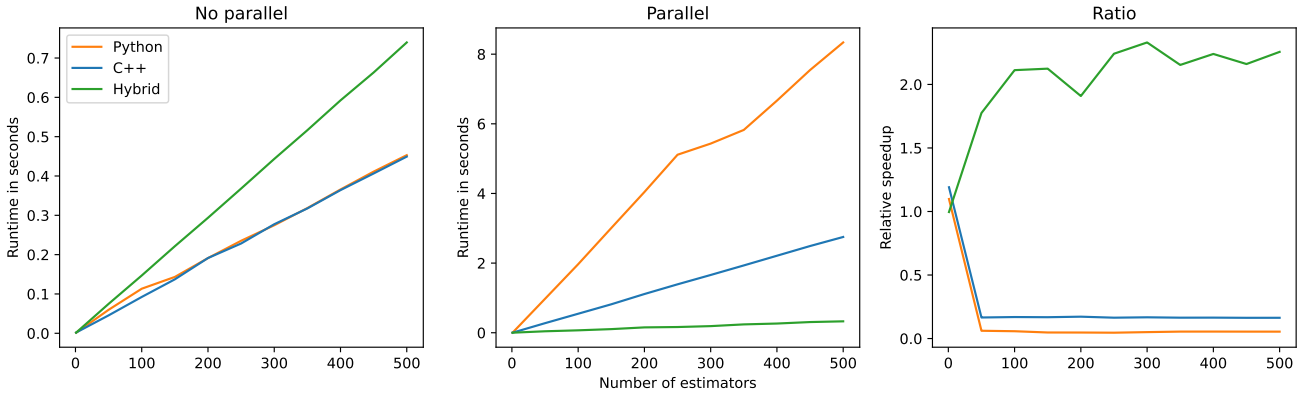


(b) 10000 samples

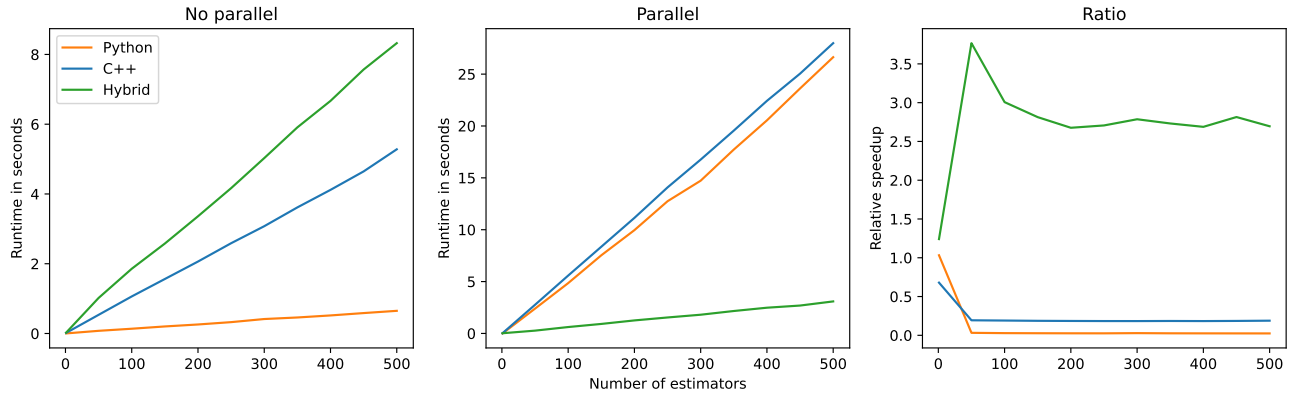
Figure 6: Runtimes and comparison of the the runtime of the `fit()` function of the Random Forest class. The function was run on on 5 synthetic datasets with 10 columns and an S shift of 0.1, without any train-test split.

Without parallelization, the C++ version and the Hybrid version are effectively equal, and as such, in the leftmost graphs, the two lines practically overlap. At 1000 samples, these versions show considerably faster times than the Python Benchmark version, but unlike with individual trees, this speedup diminishes as the sample size increases to 10000.
With parallelization, the relationship between the Python Benchmark and Hybrid version is roughly the same as without, but the pure C++ version shows a tilt upwards. At 1000 samples, this means C++ approaches the Python line, and at 10000 samples, the benchmark is even exceeded in runtime. The ratio curves on the right reveal additional information. For one, the relative speedup that parallelization can provide is limited at lower amounts of estimators, so the curves only stabilize after around 100 estimators. Secondly, the pure C++ version shows substantially less speedup from parallelization than the other implementations. At 1000 samples, the curve even goes below 1, meaning the parallelization actually slows down the execution. Also, note that the peak relative speedup any implementation is able to achieve is around 3.5, not even half of the amount of available threads.

Figure 7 shows the runtimes for the `predict_proba()` function of the Random Forest classes.



(a) 1000 samples



(b) 10000 samples

Figure 7: Runtimes and comparison of the the runtime of the `predict_proba()` function of the Random Forest class. The function was run on on 5 synthetic datasets with 10 columns and an S shift of 0.1, without any train-test split.

Like the `fit()` function, the prediction of the C++ version and the Hybrid version differ mostly in their implementation of parallelism. However, their curves do not overlap in the leftmost graphs, unlike in Figure 6. This might be explained by the much smaller range on the vertical axes, which would amplify the minute differences. At 10000 samples, the Python version runs about 4 times faster than the C++ version, which corresponds with the results in Table 5. At 1000 samples however, these two lines overlap.
With parallelization, the Hybrid version is substantially faster than the other two implementations, despite it being the slowest without. Note however that at 10000 samples, the hybrid prediction with parallelization still takes more time than the Python Benchmark prediction without.
These observations are reflected in the ratio graphs. The `predict_proba()` function of the Hybrid implementation is able to achieve a relative speedup not dissimilar to the `fit()` function, but the other two implementations fall far below 1.

7.4 Real world datasets

Table 6 shows the execution times of the different Decision Tree implementations when run on the different real world datasets outlined in Section 6.2.2.

Implementation	A	DC	GC	BM	CA
Python Benchmark	31.8s	2.28s	0.663s	51.0s	15.5s
Python Encoding Optimization	27.9s	2.51s	0.685s	51.1s	15.5s
Python Calculation Optimization	25.1s	2.06s	0.572s	46.3s	14.8s
C++ Optimized	7.47s	1.58s	0.161s	10.9s	2.85s

Table 6: Execution times for different Decision Tree implementations on real world datasets. Values are the mean of 5-fold cross validation. While the times shown are the sum of the fitting time and the prediction time, this is effectively equal to just the fitting time.

The result in this table are in line with our results on the artificial datasets, with almost each row showing a lower value than the one above it. The exception to this is the Python Encoding Optimization, which is either equal to or even slightly slower than the benchmark, save for the first column. Interestingly, this means the encoding optimization is slower for the Dutch Census dataset. This dataset consists solely of string datatypes, which, in theory, should give the implementation optimizing the encoding of string data an advantage. However, it seems that one-hot encoding reshaping the data might have a positive effect here, as evidenced before in Figure 2 and Figure 3.

Table 7 shows the results of the different Random Forest implementation on the real world datasets.

Implementation	A	DC	GC	BM	CA
Python Forest	241s	371s	4.86s	371s	178s
C++ Forest	200s	201s	9.72s	305s	94s
Hybrid Forest	112s	47s	2.56s	156s	51s

Table 7: Execution times for different Random Forest implementations on real world datasets. All forests in this table use 100 estimators and have parallelism enabled. Values are the mean of 5-fold cross validation. The times shown are the sum of the fitting time and the prediction time.

The results in this table show similarities to our results on synthetic datasets. In most columns, the Hybrid implementation is the fastest, followed by the pure C++ version, and the Python version shows the highest time, akin to Figure 6(a). The exception is the German Credit dataset, where the C++ variant is the slowest, like in Figure 6(b). This seems to imply that this dataset has the highest amount of samples, as this is what separates the two sub-figures. However, Table 1 shows us that this dataset has by far the least amount of samples, and the cause is most likely the large amount of features instead.

8 Discussion

The discussion of this work will be split into two parts: a discussion of the experimental results (Section 8.1) and a discussion of the implementation itself (Section 8.2).

8.1 Results

Due to the large amount of variables influencing the runtimes of every implementation, it is difficult to obtain a complete understanding of all the exact effects, much less provide an abridged overview to the reader. As such, we are aware that the analysis provided in Section 7 has limitations, of which we will name three:

1. **Hyperparameters**: The tree and forest classes both have more hyperparameters than we were able to explore in the analysis. The most notable of these being the orthogonality parameter, explained as Θ in Section 4.1.

2. **Values**: For the parameters we were able to explore, we have had to limit ourselves to looking at specific values or at most a specific range, which may not always provide a full overview. For example, when comparing forests on parallelism, we have limited ourselves to investigating the difference between only using 1 thread and using all available threads, disregarding any number in between.

3. **Inter-variable effects**: During preliminary testing, our results indicated that certain parameters may not only affect the execution time of the code directly, but also influence the effects of other parameters. This increases the already large amount of effects to consider exponentially, and was therefore left largely unexplored. The only inter-variable effects we investigated are those between the number of features and the number of variables (Figure 2 and Figure 3).

The aim of this thesis is to take an existing Python algorithm, and see how effective a C++ implementation is at reducing its runtime. However, the results of this thesis are not meant to be interpreted as an objective comparison between Python and C++, unlike the works discussed in Section 3.2. There are several reasons for this, of which we will name two:

1. When calling the compiled C++ code from Python, some time is needed to copy the function parameters and convert them to data C++ can parse. In most cases, this overhead is negligible, but in the case of machine learning, where function parameters often consist of large datasets, this can have a noticeable impact.

2. The benchmark Python implementation uses the NumPy package for a lot of its calculations, which is a heavily optimized package, largely implemented in C. Depending on the type of comparison being made and the personal convictions of the people performing them, this may or may not be regarded as 'pure' Python code. However, because our experiments are not meant to compare Python and C++ as a whole, we do not have to entertain this question; we have simply taken the benchmark Python code as is, and tried to improve upon it.

8.2 Implementation

Before this project, the programmer working on the code was familiar with Python, but they did not have a significant amount of experience with C++. More concretely, C++ syntax and concepts

were known, but coding standards and best practices were only learned during the work. While the results show that the C++ implementation is an overall improvement over the existing Python code in terms of runtime, it is possible, if not likely, that certain opportunities for optimization were missed.
In fact, certain parts of the code were deemed to have room for improvement, but not a high priority. These parts were often implemented as a naive first version, meant to be optimized later. However, when profiling the code showed that these parts were not taking up a large part of the runtime, the focus was directed elsewhere, and these optimizations were never performed.
Another opportunity for improvements concerns parallelism. When analyzing the effects of parallelism on the different forest implementations, we are able to conclude that some implementations show negative effects from parallelization, but we are unable to explain why. Specifically the very low speedup of the C++ implementation is most likely due to a fault in the code that should be fixable. This was extensively investigated, but unfortunately, due to the complexity of debugging multithreaded programs, the cause was not found.

9 Conclusions and Further Research

In this thesis, we have taken an existing Random Forest Classifier in Python, and implemented it in C++. The algorithm uses a fairness constraint in its splitting criterion, enabled by the input of a sensitive attribute. The resulting prediction should then be made independent of this sensitive attribute.
In our experiments, we have looked at a number of variables that effect the runtime of our code, such as number of samples, number of features, and a variable we named S shift, representing the added complexity of fair classification. We have explored the influence of these variables on the Random Forests, as well as the Decision Trees they consist of. Our results show that our C++ implementation is generally faster than the Benchmark Python version, but the exact ratio varies heavily. At best, our optimized code runs 3 to 4 times faster, at worst, it provides barely any to no speedup at all. When the forest employs parallelism, the pure C++ code can even be slower than the Python code. In this case, the best option appears to be a Hybrid solution.

When comparing Python and C++ in this thesis, we have looked purely at runtime. However, time is not the only factor of a program that can be optimized. Many of the works in literature in this field therefore include analysis of memory usage alongside the runtime. A next step could thus be to analyze the memory usage of the different implementations created in this thesis, and optimizing for this factor too.
In terms of implementation, there is still room for further work as well. As indicated in Section 8.2, there are certain parts of the code which do not take up a lot of the runtime, but could still be optimized further. Additionally, while the code developed for this research is publicly available [24], setting up the right environment still requires a good deal of effort from the user before it can be run. Future work could therefore address converting the code into a more practical Python package.

References

[1] Araujo, T., Kruikemeier, S. & De Vreese, C. H. In AI we trust? Perceptions about automated decision-making by artificial intelligence. *AI & SOCIETY*, 35(3):611–623, January 2020. DOI: 10.1007/s00146-019-00931-w.

[2] P.B. De Laat. Algorithmic Decision-Making based on Machine learning from big data: Can transparency restore accountability? *Philosophy & Technology*, 31(4):525–541, 11 2017. DOI: 10.1007/s13347-017-0293-z.

[3] The European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), April 2014.

[4] Dilsizian, S. E. & Siegel, E. L. Artificial Intelligence in medicine and cardiac imaging: Harnessing big data and advanced computing to provide personalized medical diagnosis and treatment. *Current Cardiology Reports*, 16(1), December 2013. DOI: 10.1007/s11886-013-0441-8.

[5] De Bruin, G. J., Barata, A. P., Van Den Herik, H.J., Takes, F. W. & Veenman, C. J. Fair automated assessment of noncompliance in cargo ship networks. *EPJ Data Science*, 11(1), March 2022. DOI: 10.1140/epjds/s13688-022-00326-w.

[6] Barata, A. P., Takes, F. W., Van Den Herik, H. J. & Veenman, C. J. Fair tree classifier using strong demographic parity. *Machine Learning*, August 2023. DOI: 10.1007/s10994-023-06376-z.

[7] Akter, S., Dwivedi, Y. K., Sajib, S., Biswas, K., Bandara, R. & Michael, K. Algorithmic bias in machine learning-based marketing models. *Journal of Business Research*, 144:201–216, May 2022. DOI: 10.1016/j.jbusres.2022.01.083.

[8] Kleinberg, J., Ludwig, J., Mullainathan, S. & Rambachan, A. Algorithmic fairness. *AEA papers and proceedings*, 108:22–27, May 2018. DOI: 10.1257/pandp.20181018.

[9] Binns, R. On the apparent conflict between individual and group fairness. *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, January 2020. DOI: 10.1145/3351095.3372864.

[10] Mattu, J., Angwin, J., Larson, L. & Kirchner, S. Machine bias. *ProPublica*, May 2016.

[11] Pessach, D. & Shmueli, E. Algorithmic Fairness, February 2023. DOI: 10.1007/978-3-031-24628-9_37.

[12] Dwork, C., Hardt, M., Pitassi, T., Reingold, O. & Zemel, R. S. Fairness through awareness. *Proceedings of the 2012 Conference on Innovations in Theoretical Computer Science*, January 2012. DOI: 10.1145/2090236.2090255.

[13] Jiang, R., Pacchiano, A., Stepleton, T., Jiang, H., & Chiappa, S. Wasserstein fair classification. In Ryan P. Adams and Vibhav Gogate, editors, *Proceedings of the 2020 Conference on Uncertainty in Artificial Intelligence*, volume 115 of *Proceedings of Machine Learning Research*, pages 862–872. PMLR, July 2020.

[14] Kleinberg, J., Mullainathan, S. & Raghavan, M. Inherent Trade-Offs in the fair determination of risk scores. *arXiv (Cornell University)*, September 2016. DOI: 10.48550/arxiv.1609.05807.

[15] Zehra, F., Javed, M., Khan, D., & Pasha, M. Comparative Analysis of C++ and Python in Terms of Memory and Time. *Preprints*, December 2020. DOI: 10.20944/preprints202012.0516.v1.

[16] Arboleda, F. J. M., Arias, M. R. & Riveros, J. A. H. Performance of Parallelism in Python and C++. *IAENG International Journal of Computer Science*, 50(2), June 2023.

[17] Prechelt, L. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000. DOI: 10.1109/2.876288.

[18] Lion, D., Chiu, A., Stumm, M., & Yuan, D. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster? In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 835–852, Carlsbad, CA, July 2022. USENIX Association.

[19] Fua, P. & Lis K. Comparing Python, Go, and C++ on the N-Queens Problem. *CoRR*, abs/2001.02491, January 2020.

[20] Fourment, M. & Gillings, M. R. A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9(1), February 2008. DOI: 10.1186/1471-2105-9-82.

[21] Aruoba, S. B. & Fernández-Villaverde, J. A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58:265–273, September 2015. DOI: 10.1016/j.jedc.2015.05.009.

[22] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, November 2011.

[23] Barata, A. P. Fair Tree Classifier, 2021. https://github.com/pereirabarataap/fair_tree_classifier (visited on 19/12/2023).

[24] Hesselink, S. FDTCPlusPlus, 2023. https://github.com/SanderHesselink/FDTCPlusPlus (visited on 19/12/2023).