Università Ca'Foscari Venezia

Master's Degree

in Computer Science

Final Thesis

Efficient implementation of Treant: a robust decision tree learning algorithm

Supervisor:
Ch.mo Prof. Lucchese Claudio

Assistant supervisor:
Abebe Seyum, M.Sc

Graduand:
Girardini Davide
Matricolation number
865919

Anno Accademico 2019/2020

Contents

Acknowledgements

I would like to thank my supervisor, Prof. Claudio Lucchese, and his assistant, Seyum Abebe, for their technical support and very useful suggestions that helped me to bring this work to its final form.

I am grateful to Fernanda for her encouragement in the early stages of this long path toward the degree.

Finally, I must say that without the motivation given by my friends and my family, I do not know if this result would have been possible.

Abstract

The thesis focuses on the optimization of an existing algorithm called Treant for the generation of robust decision trees. Despite its good performance from the machine learning point of view, unfortunately, the code presents some strong limitations when employed with big datasets. The algorithm was originally written in Python, a very good programming language for fast prototyping but, as well as many other interpreted languages, it can lead to poor performances when it is asked to crunch a big amount of data if not supported by appropriated libraries. The code has been translated to the C++ compiled language, it has been parallelized using OpenMP and STL libraries, along with other optimizations regarding the memory management and the choice of third party libraries. A Python module has been generated from the C++ code in order to make use of the very efficient C++ classes in the same fashion as native Python classes. In this way, any user can exploit both the Python flexibility and the C++ efficiency.

Chapter 1

Introduction

The TREANT algorithm [1] presents a novel approach for the construction of decision trees in an hostile environment. The performance of this approach is very good and the generated models has been proved to be quite robust under an attacker able to perform a *white-box* attack. The original code used to test the TREANT algorithm is written in Python, an interpreted language that can be used for fast prototyping. The code counts about 2500 lines.

Python is very powerful because is user-friendly, takes care of the memory management through an internal garbage collector, implements a huge quantities of libraries in the machine learning field: `numpy`, `pandas`, `scipy`, `sklearn` are some of them. Python is an high level language that in few lines of codes can do the job of hundreds of lines of code written in a low level language like C. A good Python developer tries to use as much as possible the packages provided because loops are very expensive. Nevertheless, sometimes it is not possible and the code can became less and less efficient with an increasing complexity.

The original code execution times are very long with big datasets and, in some cases, the usability is really limited. Very often, in the machine learning field, the number of tests on a specific algorithm can be huge and can involve very different configurations. For this reason, big tech companies that employ artificial intelligence in their applications, are investing a big amount of resources in clusters, cloud computing and computational infrastructures.

Unfortunately, the Python implementation does not scale well increasing the size of the training set, the size of features and all the parameters that can increase the computational load. This kind of limitations make the code practically unusable in many cases and difficult to be used to extend its results. This thesis tries to solve this problem re-implementing the TREANT algorithm in C++, a compiled language very suitable for High Performance Computing applications. Many libraries have been implemented in recent years to ease the parallelization of C++

code, both for multi-threading (see OpenMP, Standard Template Library, Boost) and for multi-processing (see MPI library for example). GPU parallelization is also becoming very popular in this field and usually bases its implementation on C-like programming languages like OpenCL, CUDA, and Halide.

Chapter 2 presents an overview of the most popular machine learning techniques that in the last decades have been greatly appreciated and developed by members of the scientific community.

Chapter 3 explains what is the *adversarial machine learning* and how a model can be attacked. This is quite recent research field and the main contributions regard the neural networks. The TREANT algorithm belongs exactly to this category and is supposed to run in an hostile environment. It is able to generate robust models that are able to behave well also if threatened by an external attacker.

The new C++ version of the TREANT algorithm is analysed in the chapter 4 in terms of memory management and scalability thanks to the multi-threading support. The C++ code is validated comparing its results with the original Python code results and then tested with the use of multiple threads. Different types for parallelization has been discussed, on feature level and on tree level when the target is an ensemble model. The advantages and drawbacks of these two techniques has been discussed along with test cases supporting the drawn conclusions. The optimization libraries used by the TREANT algorithm in its Python and C++ versions are discussed and compared inside the unit tests. The speed-up gained reached peaks of `x30` in the sequential version and it has been proved that the code scales well increasing the number of threads employed.

Chapter 4.6 describes how the C++ source code has been wrapped using the `CFFI` library inside a Python module. The middle layer between the two languages is composed by C functions able to call the classes and methods of the generated shared library. In this way a Python user can exploit both the efficiency of the C++ language and the great usability of Python along with all the useful libraries provided by this modern language.

Chapter 2

Machine learning overview

In this chapter four algorithms are analyzed, they can be considered as the most influential in the machine learning and data mining areas [2]: artificial Neural Networks, Support Vector Machines (SVM), k-Nearest Neighbor and Decision Trees. More space is dedicated to the decision tree section because related to the algorithm object of this thesis. The theoretical aspects behind each algorithm are explained as well as the main advantages and the possible drawbacks that can affect their scalability.

2.1 Machine learning at large scale

Large machine learning applications are becoming more and more popular in the recent years mainly because of two key factors: the big size of the dataset involved and the great progresses registered in the development of hardware architectures as well as programming frameworks.

Typically, large distributed storage platforms are used to collect and save large datasets; this implies the need of machine learning algorithms able to use efficiently this stored data.

Efficient and simultaneous processing of these large data volumes are possible thanks to the modern hardware architectures and improved programming frameworks.

In the recent years, a lot of sensors are used to collect a big volume of high dimensional and complex features growing the need of efficient hardware architectures able to process efficiently and, in most of the cases, in a parallel way this amount of data.

Visual object detection autonomous systems and speech recognition are two examples of modern applications that involve this kind of data [3].

Providing an overview of the most effective algorithms can give an idea of how well they can perform in large scale machine learning applications.

2.2 Neural networks

In various relevant applications such as pattern recognition, neural networks proved to be an effective machine learning algorithm outperforming many algorithms such as Support Vector Machines. For this reason they are gaining a strong popularity [4]. A neural network is a particular structure composed by units named neurons. Three different kind of layers usually can be detected in such architectures: the input layer containing the input feature vector; the output layer containing the neural network response and the layers in between that can present a size different from the input and output layers. An example of a neural network is illustrated in Figure 2.1. This particular artificial neural network, called *feed-forward neural network*, the information travels from the input layer to the output, no backward route is allowed.

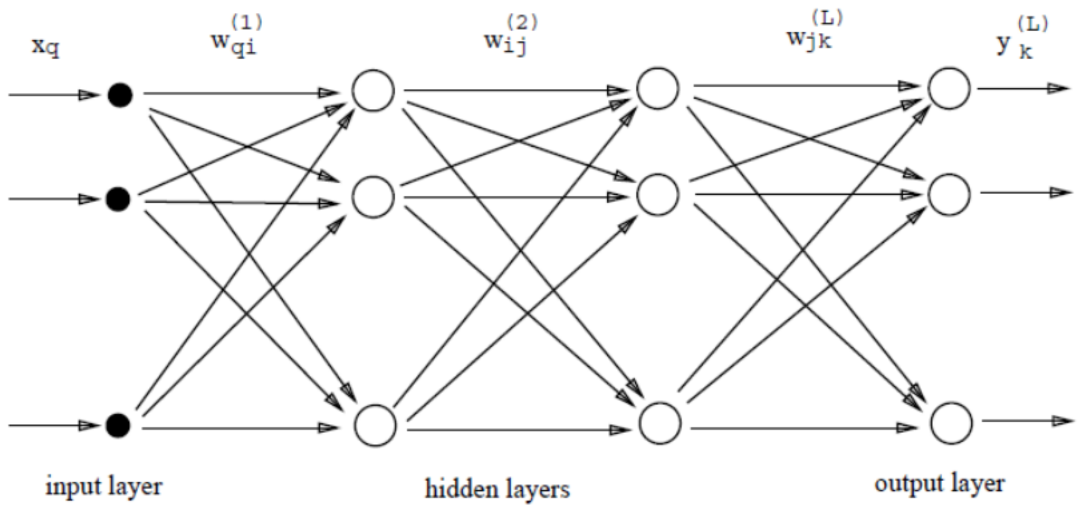

Figure 2.1: Feed-forward neural network: signals can travel only forward, from input to output

The three fundamental characteristics that compose a neural network are: the architecture, the activation functions and the weight of the connections. The network architecture and the functions must be chosen at the beginning and cannot be changed during the training. The performance of the neural network depends mainly on the value of the weights. The weights are refined during the training phase by successive approximations. Many different strategies exist for the neural networks training [5]. The Back Propagation algorithm is a very popular training method [6]. Among the other techniques we can mention the *weight-elimination algorithm* able to automatically infer the network topology and *genetic algorithms* able to derive the network structure [7].

2.2.1 Advantages and drawbacks

The difficult part of a neural network design is guessing the most effective sizes of the hidden layers.

When the number of neurons is too low or too high, the derived system does not generalize well to unseen instances. For example, when too much nodes are used, it may occur the phenomenon called *overfitting* and the desired optimum may not be found at all. Kon and Plaskota [8] studied methods to derive the nearly optimal quantity of neurons. High dimensional features such as images are typically processed well by using neural networks. Unfortunately, neural networks demand a huge amount of computing resources consuming a large amount of processing power and physical memory. Another debated aspect is the difficult neural network comprehensibility for average machine learning users [9].

2.2.2 Scalability

The most used training strategy for neural network training is the back propagation algorithm, for this reason the scalability discussion starts from the analysis of this procedure. Kotsiantis et al. [9] describes the Back Propagation algorithm with six steps:

1. the input layer of the network receives an instance;
2. propagate the information to the output layer;
3. for each neuron, the local error is computed;
4. the weights are tuned in order to minimize the local error;
5. for the error, a penalty is accredited to the previous level of neurons, implicitly more importance is given to the neurons that have an higher weight;
6. another iteration is done where each neuron penalty is used as its error.

The optimal weight configuration is reached through successive adjustments of the weights. I can be shown that the training stage computation time is equal to $O(nW)$ [9], where n is number of training instances and W are the number of weights. So far, the amount of training required for a neural network can not be estimated through an exact formula. The number of needed iterations generally depends on the problem and on the particular architecture chosen. The error in a feed-forward neural network could be equal to $O(1/N)$ as showed by Barron [10]. Furthermore, a rule of thumb exists for the size T of a generic training set [11]:

$$T = O(N/\epsilon) \tag{2.1}$$

where ϵ represents the part of classification errors that is allowed. Because the feed-forward operation is $O(N)$, large parallel computers can be used to train these neural networks and derive the weights, and thereafter copy them to a serial computer for implementation.

2.3 Support vector machine

From the Structural Risk Minimization principle, part of the computational learning theory, has been derived a classification technique called Support Vector Machines (SVM). The main goal of the SVM is to differentiates between units of classes in the training data by means of an optimal classification function. Considering a linearly separable dataset, the optimal classification function is determined constructing a hyperplane which maximizes the margin between two subsets and thus creates the largest possible distance between those [12]. Figure 2.2 illustrates this strategy.

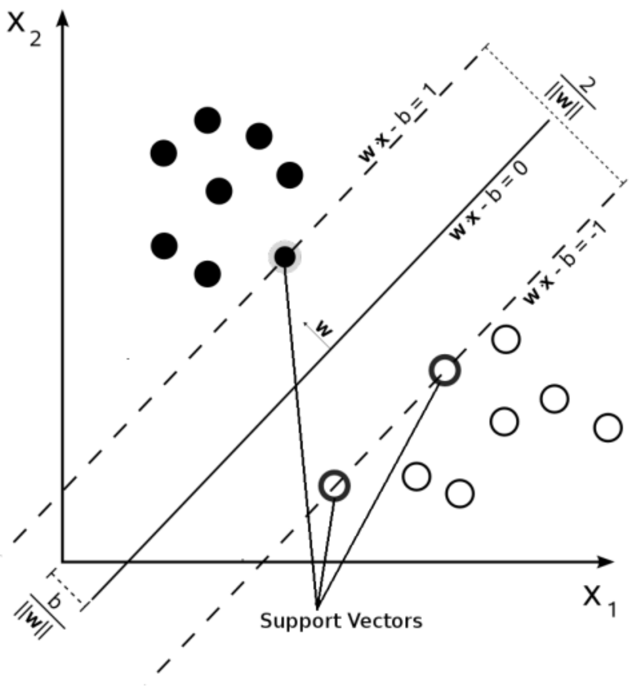

Figure 2.2: Visualization of a 2D support vector machine classifier

From the SVM point of view, the best generalization of the model is reached maximizing the margin and thus the most optimal hyperplane. This yields to the best classification performance for both the training data as well as future data. SVM maximizes the function (2.2) in relation to $\overrightarrow{w}$ and b in order to find the maximum margin hyperplane for the lagrangian L_P:

$$L_P = \frac{1}{2} \|\overrightarrow{w}\| - \sum_{i=1}^{t} \alpha_i \gamma_i (\overrightarrow{w} \cdot \overrightarrow{x}_i + b) + \sum_{i=1}^{t} \alpha_i \tag{2.2}$$

Here, t is the number of training points, α_i are the lagrangian multipliers. The hyperplane is characterized by the vector $\overrightarrow{w}$ and the constant b. Data-points laying on the optimal separating hyperplane are called *support vector points.* The resulting hyperplane is a linear combination of these support vector points, the remaining points are ignored and are not part of the solution. For this reason, the complexity of a SVM is not affected by the number of features present in the training data and makes SVMs very suitable for learning problems that contain a large number of features in comparison to the amount of training instances. Unfortunately, sometimes it is not possible to find a separating hyperplane with the presence of misclassified instances.

Non-linearly separable data are very common in real world problems, in such cases no hyperplane correctly dividing the training set can be found. In order to overcome this problem, a possible solution is outlining the data onto an altered feature space. It is possible to demonstrate that increasing the feature space dimensionality any regular training set can be divisible [9].

2.3.1 Advantages and drawbacks

Among the many advantages that SVM provides, we can mention that the algorithm is based on an established theory, it needs only tens of training specimens and it is not affected by the feature space dimensionality [9]. However, the learning strategy implements relatively complex training and categorization algorithms as well as high memory and time utilization during training and classification phases.

2.3.2 Scalability

The solution of an n-dimensional Quadratic Programming problem (QP) is involved in any SVM training phase, where n represents the number of training instances. Large matrix operations and time consuming numerical calculations are usually implemented in standard QP methods to solve this kind of problem. Dealing with large scale applications, this method is very inefficient and not practical, unfortunately this is a well known drawback of the SVM method. Methods that can solve the QP problem relatively quickly and efficiently can be found in the scientific literature, for example the Sequential Minimal Optimization (SMO) [13]. This method divides the QP problem into QP sub-problems and solves the SVM quadratic programming problem without the use of numerical QP optimization steps or extra matrix storage. A novel SVM approach finds an estimation of a least surrounding sphere of a group of items [2]. Intrinsically, SVM methods solve binary classification problems; so, dealing with a multi-class problem, usually the strategy used is dividing the problem into a group of various classification challenges.

2.4 k-Nearest Neighbor

The main goal of the *k-Nearest Neighbor* classification algorithm, or *kNN*, is to find a group (also called *cluster*) of k objects that has the closest proximity to a test object, then a unique label is assigned considering the prevalence of a class in the closest proximity. The main three components of this algorithm are here summarized: a group of labeled objects; a proximity metric; and the number k of nearest neighbours [2]. Usually, the "Euclidian Distance" is used, this is a very popular proximity metric used for kNN classification. This metric is explained by the following formula [9]:

$$D(x, y) = \left(\sum_{i=1}^{m} (x_i - y_i)^2 \right)^{1/2} \tag{2.3}$$

The "Euclidian Distance" is only one of the possible metrics used to define a distance between instances of a dataset. Other examples include the Minkowsky, Camberra or Chebychev metrics [14], even if often, in order to achieve more accurate results altering the voting influence, weighing strategies are used.

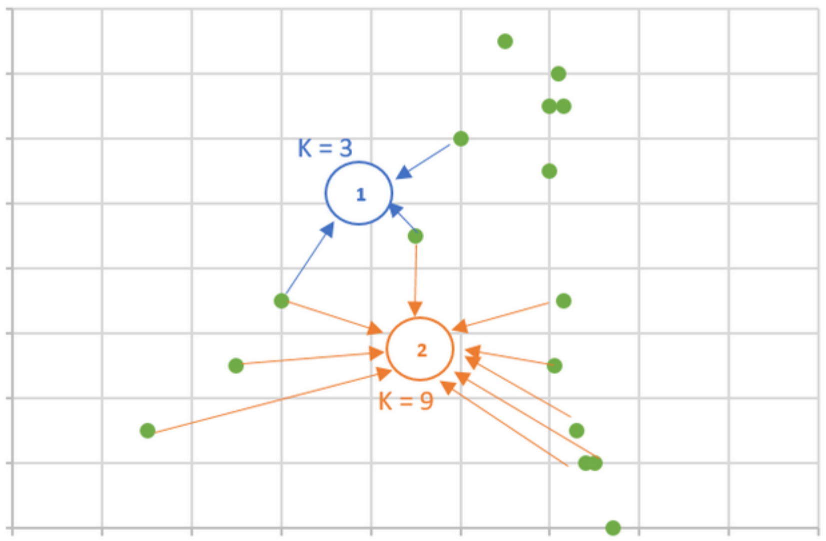

Figure 2.3: Visualization of the k nearest neighbors of two centroids in the case of k = 3 and k = 9

Equation (2.4) is used to get the majority class and categorize the test objects once the k-Nearest-Neighbor list is acquired:

$$Majority\ Voting : Y' = argmax \sum_{(x_i,y_i)\in D_z} I(v = y_i) \tag{2.4}$$

where v represents the class label, y_i represents the *i-th* nearest neighbor class label and $I(\cdot)$ is the indicator functions that returns value one for a valid argument or zero for an invalid argument [2].

2.4.1 Advantages and drawbacks

The logic behind kNN algorithms can be easily understood and implemented. Despite its simplicity, the algorithm still performs well for many cases, for example for multi-modal classes. However, structural problems can influence the algorithm behavior. Kotsiantis et al. [9] summarizes those in three points:

1. the algorithm can require too much memory especially for large datasets;
2. the choice of the similarity function can change a lot the results;
3. an approved method for selecting the k factor is not existing.

2.4.2 Scalability

kNN classification is one of the unsupervised learning algorithms and the training phase does not exist, instead all feature values are stored in memory. Compared with other machine learning algorithms like SVMs or decision trees, kNN classifiers are considered slow learners. Although the model creation is computationally inexpensive, the classification phase is expensive from the computational point of view because each k-nearest neighbor needs to be labelled. This requires calculating the distance based on the chosen metric of the unlabeled instance to all instances in the labeled set. For this reason, all the dataset must be stored in memory, and the computation can be extremely expensive, especially with a great number of instances. Many methods has been discussed in the past in order to deal with this problem, such as the inverted index join algorithm [15], one of the fastest algorithms for producing exact kNN graphs. The drawback of this algorithm is its exponentially growing execution time. The greedy filtering algorithm by Park et al. [16] is one example of the recent algorithms recently developed that guarantee a time complexity of $O(n)$.

2.5 Decision trees

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be represented as sets of if-then rules to improve human readability. These learning methods are among the most popular of inductive inference algorithms and have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

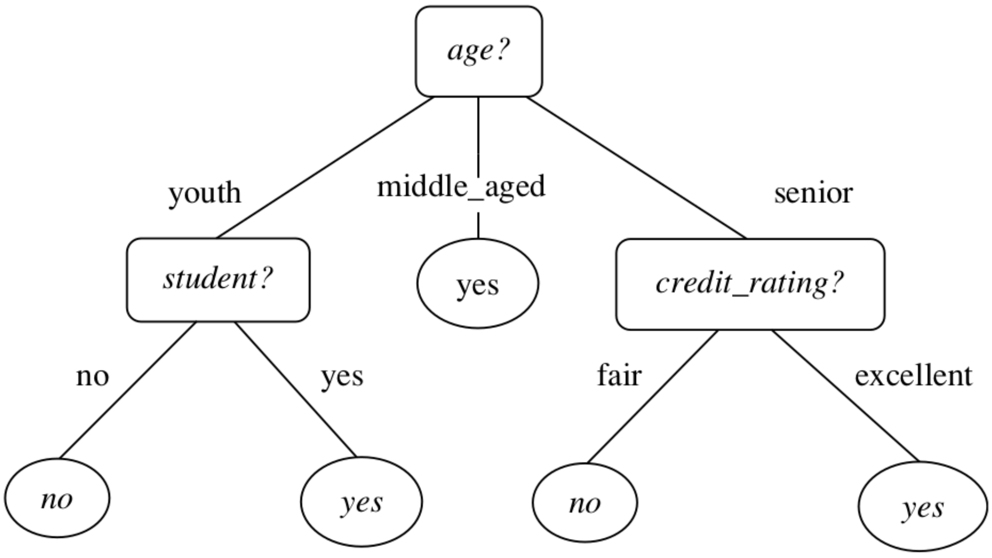

Figure 2.4: A decision tree for the concept *buys_computer*, indicating whether an *WorldElectronics* customer is likely to purchase a computer. Tests on attributes are represented by internal nodes (non-leaf). The two classes (either *buys_computer* = yes or *buys_computer* = no) are represented by the leaf nodes.

Figure 2.4 shows a typical decision tree that represents the concept *Buys computer*. The goal of the model is predicting whether a customer at *WorldElectronics* is likely to purchase a computer. Rectangles denote internal nodes, whereas ovals denote leaf nodes. Some decision tree algorithms produce only binary trees (where each internal node branches to exactly two other nodes), whereas others can produce non-binary trees.

For example the instance:

$$(age = youth, \quad student = no, \quad credit\_rating = excellent)$$

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that *Buys computer = no*).

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions.

For example, the decision tree is shown in Figure 2.4 corresponds to the expression:

$$\begin{array}{rc} & (age = youth \wedge student = yes) \\ \vee & (age = middle\_aged) \\ \vee & (age = senior \wedge credit\_rating = excellent) \end{array}$$

2.5.1 Bootstrapping

Bootstrapping is any test or metric that uses random sampling with replacement. It falls under the broader class of resampling methods. Bootstrapping assigns measures of accuracy like for example bias, variance, confidence intervals, prediction error, to sample estimates. The estimation of the sampling distribution of almost any statistic using random sampling methods can be reached using this technique.

The bootstrap approach can be applied in machine learning applications involving a set of trained models, like the ensemble models. In this case the training records are sampled with replacement; i.e., a record already chosen for training is put back into the original pool of records so that it is equally likely to be redrawn. If the original data has N records, it can be shown that, on average, a bootstrap sample of size N contains about 63.2% of the records in the original data. This approximation follows from the fact that the probability a record is chosen by a bootstrap sample is $1 - (1 - 1/N)^N$. When N is sufficiently large, the probability asymptotically approaches $1 - e^{-1} = 0.632$. Records that are not included in the bootstrap sample become part of the test set. The model induced from the training set is then applied to the test set to obtain an estimate of the accuracy of the bootstrap sample, ϵ_i. The sampling procedure is then repeated b times to generate b bootstrap samples used to train the rest of the ensemble model.

2.5.2 Bagging predictors

Bootstrap aggregating, also called bagging (from **b**ootstrap **agg**regat**ing**), is a machine learning ensemble meta-algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. It also reduces variance and helps to avoid overfitting. Although it is usually applied to decision tree methods, it can be used with any type of method.

Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. The aggregation averages over the versions when predicting a numerical outcome and does a plurality vote when predicting a class. The multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets. Tests on real and simulated data sets using classification and regression trees and subset selection in linear regression show that bagging can give substantial gains in accuracy. The vital element is the instability of the prediction method. If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.

2.5.3 Random forests

A random forest is an ensemble (i.e., a collection) of fully grown decision trees. Random forests are often used dealing with very large training datasets and a very large number of input variables. A random forest model is typically composed by tens or hundreds of decision trees. The generalisation error rate from random forests tends to compare favourably to boosting approaches, yet the approach tends to be more robust to noise in the training dataset, and so seems to be a very stable model builder, mildly affected by the noise in a dataset that single decision tree induction usually presents. Recent studies demonstrated that the random forest model builder is very competitive with nonlinear classifiers such as artificial neural nets and support vector machines. However, performance often depends on dataset configuration and so it remains useful to try a suite of approaches. Each decision tree is built from a bag of the training dataset, using what is called *replacement* in performing this sampling. During the construction of a decision tree, at each node, the features to be considered are randomly sampled. This idea introduces a variation among the trees by projecting the training data into a randomly chosen subspace before fitting each tree or each node. The construction of each decision tree is not stopped until the model reaches its maximum size (no pruning performed). Together, the resulting decision tree models of the forest represent the final aggregated model where each decision tree votes for the result, and the final outcome is decided by the majority. It is worth to remind that for a regression model the result is the average value over the ensemble of regression trees. Different options can be tuned during the random forest construction: the number of trees to build, the training dataset sample size to use for building each decision tree, and the number of variables to randomly select when considering how to partition the training dataset at each node. A final report can also present the input variables that are actually most important in determining the values of the output variable. Avoiding the pruning, i.e. building each decision tree to its maximal depth, hopefully we can generate a model that is less biased. The RF natural randomness in the dataset selection and in the variable selection delivers considerable robustness to noise, outliers, and over-fitting, when compared to a single tree classifier.

This randomness has another advantage: it also delivers substantial computational efficiencies. In building a single decision tree we may select a random subset of the training dataset. Moreover, at each node in the process of building the decision tree, only a small fraction of all of the available variables are considered for determining the best dataset partition. For this reason, the computational requirement is substantially reduced.

In summary, a random forest model is a good choice for model building for a number of reasons. First, just like decision trees, very little, if any, pre-processing of the data needs to be performed. The data does not need to be normalised and the approach is resilient to outliers. Second, if we have many input variables, we generally do not need to do any variable selection before we begin model building. The random forest model builder is able to target the most useful variables. Thirdly, because many trees are built and there are two levels of randomness and each tree is effectively an independent model, the model builder tends not to overfit to the training dataset.

2.5.4 Advantages and drawbacks

Decision tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions.

Comparative studies focused on performance demonstrated that decision tree algorithms possess a very good combination of classification speed and error rate if compared with other machine learning algorithms [17].

An improved version of the C4.5, the EC4.5, can calculate identical decision trees gain of up to 500% [18].

The construction of decision tree classifiers does not require any domain knowledge or parameter setting, for this reason it is appropriate for exploratory knowledge discovery. Decision trees are not limited by the data space dimensionality. The tree form makes intuitive and generally easy to assimilate by humans the representation of the acquired knowledge. The two main steps of the decision tree induction, learning and classification, are usually simple and fast. In general, decision tree classifiers are considered to be machine learning models having a good accuracy. However, the data at hand can determine a successful use of the model. Many application areas such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology have employed decision tree induction algorithms for classification, they are also the basis of several commercial rule induction systems.

2.5.5 Scalability

There are many examples of tree inducing algorithms that split the feature space with hyper-planes that are parallel to an axis. This kind of strategy generates the so called *univariate* decision trees. Split at each non-terminal node usually involves a single feature. Building an *univariate* decision tree has a time complexity of $O(d \cdot f \cdot Nlog(N))$, where N represents the total number of instances, f is the number of features and d the number of tree nodes [19]. Data mining applications that having large datasets can build univariate decision trees in reasonable time, good performance can be achieved by parallelizing these algorithms.

Olcay and Onur [19] proposed side-by-side applications of the C4.5 algorithm through three different methods:

- feature based parallelization;
- node based parallelization;
- data based parallelization;

Following their study, on datasets with high dimensionality, feature based parallelization seems to be, theoretically, the most effective reaching high speedups, whereas data based parallelization and node based parallelization reach their best performance respectively on large datasets and on datasets that contain trees with a high number of nodes.

Perfect load balancing is a mandatory requirement in all of these cases. As confirmed by the test cases discussed in chapter 4.3, they observed that the speedup depends significantly on the load distribution amongst processors.

Other kind of parallel strategies are described in [20] where the authors employed graphics cards to generate Random Forests (RF) and Extremely Randomized Trees (ERT); or in [21] where the authors described *PLANET*: a scalable distributed framework for learning tree models over large datasets with a *MapReduce* strategy.

2.6 Summary

Machine learning algorithms try to solve mainly two kind of problems: classification problems and regression problems. While models for regression problems predict the value of a numerical variable providing as output a number, the classification models, also called *classifiers*, predict categorical (discrete, unordered) class labels. The machine learning algorithms presented in this chapter are applied mainly as classifiers, but, even after an overview, it is usually difficult to say which is the best, in general.

Most popular machine learning algorithms are discussed in several comparison studies in the recent literature. Easily interpreted decision trees, like the ones generated by C4.5, contain a good mixture of error rate and calculation time. However, C4.5 has a lower performance if compared to SVM when there are multi-dimensions and continuous features [9]. A subset of the C4.5 and kNN (among other algorithms) has been compared with SVM by Kotsiantis and Miyamoto et al., in those experiments SVM performed significantly better than C4.5 and kNN [22].

Choosing the right size of the hidden layer is the main issue when working with Neural Networks. A poor generalization can be caused by an underestimation of this layer, whereas overfitting can be due to an overestimation [2].

Comparison experiments performed by Royas-Bello et al. and Aruna et al. between SVM and Neural Networks using multiple datasets concluded that SVM can reach a higher accuracy than Neural Networks [23][24].

Despite its simplicity, the kNN algorithm must store all feature values in memory. However, even if it lacks a training phase, there are multiple issues that can affect the performance of this algorithm. According to Kotsiantis et al. and Wu et al. kNN requires large storage space, its results are sensitive to noise and irrelevant features, the choice of k is not principled and the choice of the metric used to calculate the distances can significantly affect the model [9][2]. In a similar way, neural network training can be very inefficient due to the presence of irrelevant features.

It seems that better performance are achieved by SVMs and Neural networks working with multi-dimensions or continuous features; as opposed, working with categorical or discrete features, logic-based systems should be preferred. On the other hand, when working with multi-collinearity and an existing input-output relation, NNs and SVMs achieve better performance.

Diagonal partitioning can affect negatively decision trees models. Noise can seriously affect kNN classifiers, in contrast to decision trees which are very tolerant in this case.

Concluding this overview, when facing a classification problem, it seems that the best approach to choose the best algorithm is performing a benchmark of the candidate algorithms and selecting the most promising one for the specific application.

Chapter 3

Adversarial machine learning

This thesis focuses on decision trees trained in an hostile environment, for this reason the adversarial learning process needs to be explained with some further details. This chapter presents some examples of attacks that can be performed on two very popular machine learning (ML) models: deep neural networks and decision trees. When a machine learning system is designed to ensure system security, such as in spam filtering and intrusion detection, everybody acknowledges the need of training models resilient to adversarial manipulations [25], [26]. Yet, the same considerations can be drawn for other critical application scenarios in which machine learning is now employed, where adversaries may cause severe system malfunctioning or faults.

In the scientific community there is a growing concern about machine learning algorithm potential to reproduce discrimination against a particular group of people based on sensitive characteristics such as gender, race, religion or other. In particular, algorithms trained on biased data are prone to learn, perpetuate or even reinforce these biases.

Recently, many episodes of this nature have been documented. For example, an algorithmic model used to generate predictions of criminal recidivism in the United States of America (COMPAS) discriminated against black defendants [27]. Also, discrimination based on race and gender could be demonstrated for targeted and automated online advertising on employment opportunities [28].

Taking an example regarding the financial sector, consider a machine learning model which employed by a bank to grant loans to inquiring customers: a malicious user may try to undermine the model qualifying himself for a loan. Unfortunately, traditional machine learning algorithms proved week defensive strategies to a wide range of attacks, and in particular to evasion attacks, i.e., carefully crafted perturbations of test inputs that manipulates the model and can generate prediction errors.

In this context, the EU introduced the General Data Protection Regulation

(GDPR). This legislation, dated May 2018, represents one of the most important changes in the regulation of data privacy in the last 20 years. It strictly regulates the collection and use of sensitive personal data. The main purpose is obtaining non-discriminatory algorithms, it rules in Article 9(1): "Processing of personal data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation shall be prohibited" [29]. The common practice, today, is to remove protected attributes from the data set. This concept is known as "fairness through unawareness".

While this approach seems to be viable when using conventional, deterministic algorithms with a manageable quantity of data, it is not compliant for machine learning algorithms trained on "big data." Here, unexpected links to sensitive information can rise from complex correlations in the data. Unfortunately, there are cases where presumably non-sensitive attributes, can serve as substitutes or proxies for protected attributes.

For this reason, on one side the data scientists try to optimize the performance of a machine learning model, but on the other they are challenged to determine whether the model output predictions are discriminatory, and how can be mitigated such unwanted bias. Many bias mitigation strategies for ML have been proposed in recent years; however, most of them regard neural networks. Ensemble methods combining several decision tree classifiers have proven to be very efficient for various applications. Therefore, actuaries and data scientists prefer, for tabular data sets, the use of gradient tree boosting over neural networks due to its generally higher accuracy rates.

3.1 Modelling an hostile environment

A game between two players, an attacker and a defender, can be used to model secure learning systems: the data are manipulated by the attacker that tries to fool the learning algorithm chosen by the defender. Let us calling H the learning algorithm and the two strategies implemented by the attacker $A^{(train)}$ and $A^{(eval)}$. The resulting game has the following players and phases:

1. **Defender**: in order to select hypotheses based on the observed data, he chooses the learning algorithm H ;

2. **Attacker**: can have a potential knowledge of H and chooses attack procedures $A^{(train)}$ and $A^{(eval)}$

3. Learning phase:

- Get the dataset $\mathbb{D}^{(train)}$ affected by $A^{(train)}$;
- Learn hypothesis: $f \longleftarrow H(\mathbb{D}^{(train)})$

4. Evaluation phase:

- Get the dataset $\mathbb{D}^{(eval)}$ affected by $A^{(eval)}$;
- Each prediction $f(x)$ is compared with y for each pair $(x, y) \in \mathbb{D}^{(eval)}$

Figure 3.1 shows these steps where the fundamental interactions between the defender and attacker in choosing H, $A^{(train)}$, and $A^{(eval)}$ take place. H is chosen by the defender to select hypotheses that predict with a good accuracy without considering the possible attacks generated by $A^{(train)}$ and $A^{(eval)}$, in the meanwhile the attacker tries to produce poor predictions choosing the most aggressive $A^{(train)}$ and $A^{(eval)}$.

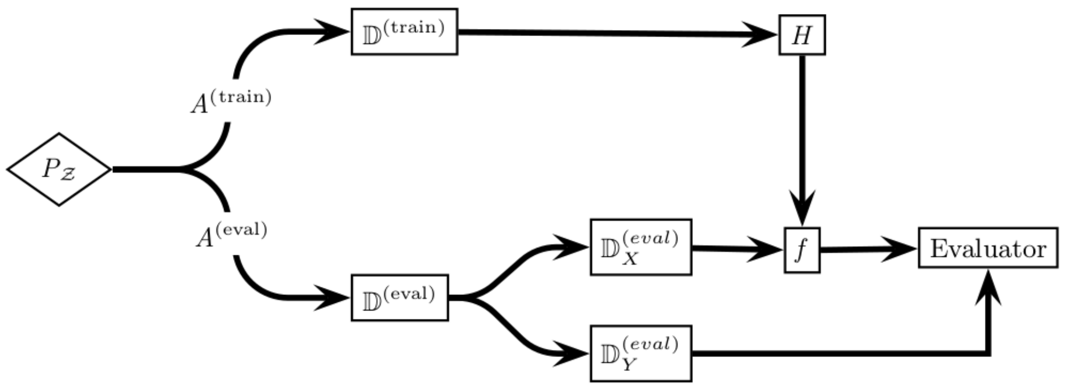

Figure 3.1: Diagram of an attack against a learning system where $P_{\mathcal{Z}}$ is the data's true distribution, $A^{(train)}$ and $A^{(eval)}$ are adversary's attack procedures, $\mathbb{D}^{(train)}$ and $\mathbb{D}^{(eval)}$ are the training and test datasets, H is the learning algorithm, and f is the hypothesis it learns from the training data. The hypothesis is evaluated on the test data by comparing its prediction $f(x)$ to the true label y for each $(x, y) \in \mathbb{D}^{(eval)}$

Influence, security violation and specificity are characteristics typically used to categorize a generic attack.

For example, the structure of the game and the legal moves that each player can make are determined by the INFLUENCE axis.

Exploratory attacks are not applied during the training phase, but they use other techniques, like, for example, probing the detector to discover information about it or its training data. When this kind of attacks are present, the procedure $A^{(train)}$ is not used and thereby the attacker only influences $\mathbb{D}^{(eval)}$.

On the other hand, the attacker may has indirect influence on f through his choice of $A^{(train)}$ when he uses *causative* attacks that are active during the training process. The choice of the instances the adversary would like to have misclassified during the evaluation phase is determined by the SECURITY VIOLATION and SPECIFICITY axes of the taxonomy.

In an *integrity* attack the focus is moved to false negatives, for this reason the attacker will use $A^{(train)}$ and/or $A^{(eval)}$ to find false negatives in the dataset, whereas the creation of false positives can take place in an *availability* attack. A *targeted* attack involves the predictions for a small number of instances, while an *indiscriminate* attacker employs its resources to a wide range of instances.

3.2 Attacker knowledge

A very important aspect regarding the attacker nature is the amount of information accessible and possibly used to fool the learning system. Generally, three components of of the learner are known to the adversary: its learning algorithm, its feature space and its data.

Not only the attacker's capabilities are a key factor to build a reasonable model, but also a critical aspect is the attacker's information.

The *Kerckhoffs' Principle* [30] can be taken as a relevant guideline for assumptions about the adversary: i.e., secure learning systems should not be based on unrealistic expectations of secrecy. Rely on secrets can be very dangerous because, in case of information leakage, the entire system can be suddenly comprised. I has been proved that robust security systems marginally rely on assumptions about what can be kept realistically secret from a potential attack.

On the other hand, giving to the attacker an unrealistic degree of information, can yield to an over pessimistic model; e.g., optimal attacks can be crafted by an omnipotent adversary who completely knows the algorithm, feature space, and data.

Thus, it is important to estimate adequately the real possibilities of the attacker, what is its knowledge of the learning algorithm and the side effects that this information can cause.

A good compromise is assuming that the attacker has complete knowledge of the training algorithm, and, in addition, partial or complete information about the training set. For example, the attacker could eavesdrop on all network traffic over a specific time period that involves the sampling of the learner training data. In the following, we consider different degrees of the adversary's knowledge and we evaluate the attack efficiency based on different sources of potential information.

White-Box Attacks: in this case the attacker is assumed to know everything about the model, the training algorithm, and training data distribution. Hence, the underlying training strategy and parameters are exposed to the adversary.

Black-Box Attacks: the attacker have a very limited knowledge of the targeted model or algorithm and can not access it's parameters. The attacker can query the model to observe what kind of outputs it gives. In the hard-label black box attack scenario the attacker is not able to query the model, and perturbation to craft adversarial samples is done without it.

Different attack types can be generated changing of the following conditions: when the attack takes place (i.e, before or after learning process); which is its influence (if *causative* introduces vulnerabilities at the beginning, if *exploratory* introduces vulnerabilities after training). Based on that, we can distinguish two fundamental types of attack.

Poisoning attack: takes place before the learning process starts by modifying a part of training data in order to disrupt the process and make it learn what it should not learn, hence reduce the performance and accuracy of the trained model. The bad samples can be labeled incorrectly in the training domain (either it is targeted or indiscriminate) or in a different domain. One of the challenges of targeting training data is that it is difficult to get their real distribution since data are usually protected. Hence, poisoning attack exploit the vulnerability of machine learning models emerging from re-training which corrupt the process by meddling. Other methods of poisoning attack are label modification in which the adversary modify the label for arbitrary instances, and also if the adversary has access to the training data it is possible to poison by modifying the data before it is used for training, and if the adversary has the ability to augment data it can inject malicious instances into the training set.

Evasion attack: is the most known type of attack where the attacker craft adversarial samples that look like normal data instances which forces wrong prediction (wrong labelling). In this attack adversarial samples are crafted at test time to evade detection and exploit the vulnerabilities of trained model. Hence, the model misclassifies it as a legitimate instance. This attack does not assume any influence over the training data. In evasion attack, given a generic instance x, the target is to add the smallest perturbation δ which generates an adversarial sample $\bar{x} = x + \delta$ that is able to evade the model into an inaccurate prediction. A typical example in *Computer Vision* is changing some image pixel before uploading and fool the correct model labelling.

3.3 Trick neural networks

In the recent years, a lot of attacks examples related to deep neural networks has been discovered, probably because this research field is still young and the learning process of a neural network is still difficult to understand at a human eye and the tests, sometimes can lead to unexpected results.

The adjective "deep" in deep learning (also known as *deep structured learning*) comes from the use of multiple layers in the network. Early work showed that a linear perceptron cannot be a universal classifier, and then that a network with a non-polynomial activation function with one hidden layer of unbounded width can on the other hand so be. Deep learning is a modern variation which is concerned with an unbounded number of layers of bounded size, which permits practical application and optimized implementation, while retaining theoretical universality under mild conditions. In deep learning the layers are also permitted to be heterogeneous and to deviate widely from biologically informed connectionist models, for the sake of efficiency, trainability and understandability, whence the *structured* connotation.

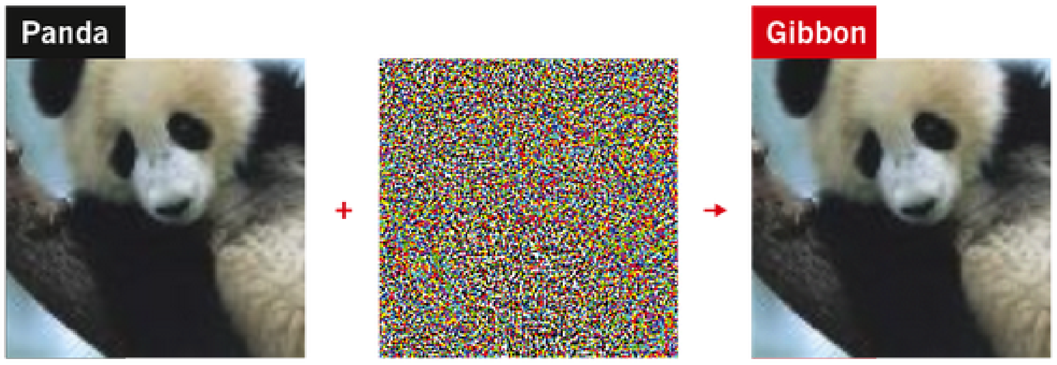

Figure 3.2: Adding carefully crafted noise to a picture can create a new image that people would see as identical, but which a DNN sees as utterly different.

State-of-the-art deep neural networks have achieved impressive results on many image classification tasks. However, these same architectures have been shown to be unstable to small, well sought, perturbations of the images. Despite the importance of this phenomenon, no effective methods have been proposed to accurately compute the robustness of state-of-the-art deep classifiers to such perturbations on large scale datasets.

A first big reality check on the DNN classification performance came in 2013, when Google researcher Christian Szegedy and his colleagues posted a preprint called "Intriguing properties of neural networks". The researchers showed that it was possible to take an image - of a lion, for example - that a DNN could identify and, by altering a few pixels, convince the machine that it represented something different, such as a library (see Figure 3.2 for another example). The team called the analyzed images *adversarial examples.*

Surprisingly, DNNs can see things that were not there, such as a penguin in a

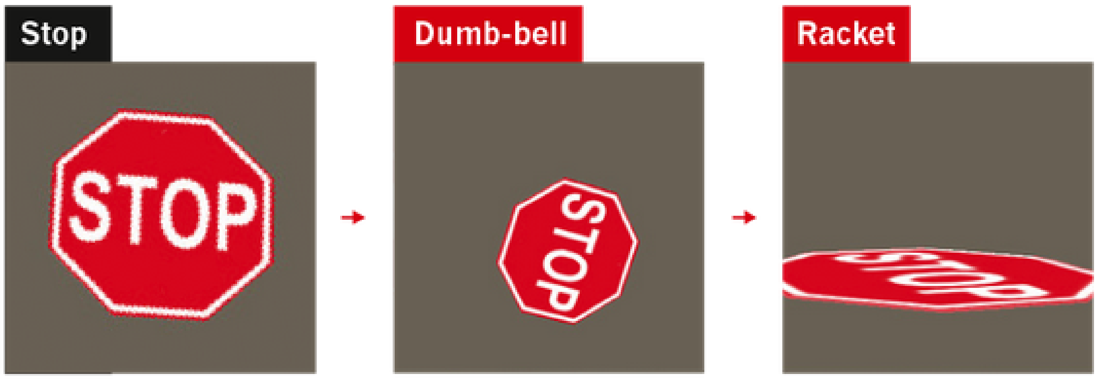

Figure 3.3: DNNs can be confused by object rotation, probably because they are too different from the types of image used during the training phase

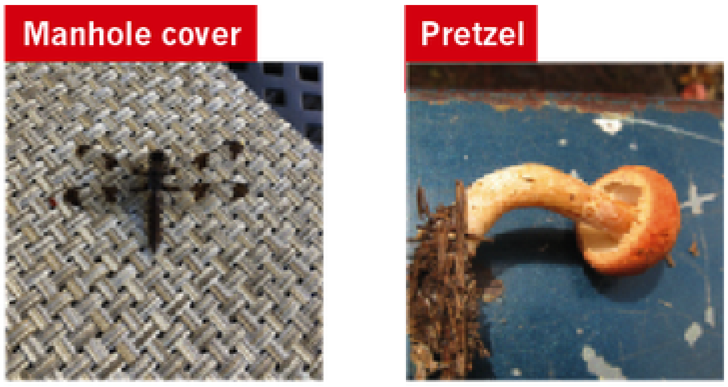

Figure 3.4: Natural images can sometimes trick a DNN, because it might focus on picture colour, texture or background rather than selecting the salient features a human would recognize

pattern of wavy lines as demonstrated by Clune and Anh Nguyen, together with Jason Yosinski.

The following are the considerations of Yoshua Bengio (University of Montreal in Canada), a pioneer of deep learning: "Anybody who has played with machine learning knows these systems make stupid mistakes once in a while. What was a surprise was the type of mistake", he says. "That was pretty striking. It's a type of mistake we would not have imagined would happen".

New types of mistake have been recorded in this field. Again, Nguyen [31] showed that a simple rotation in an image was sufficient to throw off some of the best image classifiers around. According to a recent study by Hendrycks and his colleagues [32], even unadulterated, natural images can still trick state-of-the-art classifiers into making unpredictable gaffes, such as identifying a mushroom as a pretzel or a dragonfly as a manhole cover (see Figures 3.3 and 3.4). These are only few of the numerous examples of DNN fooling processes that can be found in the current scientific literature. In practice for tabular data sets, actuaries and data scientists prefer the use of gradient tree boosting over neural networks due to its generally higher accuracy rates.

3.4 Trick decision trees

To date, research on evasion attacks has mostly focused on linear classifiers and, more recently, on deep neural networks. As described in section 2.5, decision trees are interpretable models, yielding to predictions which are human-understandable in terms of syntactic checks over domain features; this is particularly appealing from the security point of view.

Unfortunately, despite their success, only limited attention have been given to decision tree ensembles by the security and machine learning communities so far, yielding a sub-optimal state of the art for adversarial learning techniques.

The perturbation cost, in most of the attacks, is calculated as a measure of distance. The following three metrics are well known in adversarial learning research community:

1. $\ell_0 - norm$: captures localized perturbations with arbitrary magnitude. For example, attacks using $\ell_0 - norm$ on image minimize the number of modified pixels.

2. $\ell_2 - norm$: attacks which calculate their cost using $\ell_2 - norm$, first calculate the sum of squared error between adversarial and benign samples and then minimize it.

3. $\ell_\infty - norm$: this measurement is the simplest one and aim to minimize the amount of perturbation which can be applied to generate adversarial samples.

Models that implement counter-measures to the threats of on attacker are usually classified as *robust models*. Figure 3.5 explains how the best accuracy should not be confused with the best robustness.

In the upper figure a set of 10 points is easily separated with a horizontal split on feature $x^{(2)}$. The accuracy of this split is 0.8. In the middle figure the high accuracy horizontal split cannot separate the ℓ_∞ balls around the data points and thus an adversary can perturb any example $\mathbf{x}_i$ within the indicated ℓ_∞ ball to mislead the model. The worst case accuracy under adversarial perturbations is 0 if all points are perturbed within the square boxes (ℓ_∞ norm bounded noise). The lower figure shows a more robust split on feature $x^{(1)}$. The accuracy of this split is 0.7 under all possible perturbations within the same size ℓ_∞ norm bounded noise

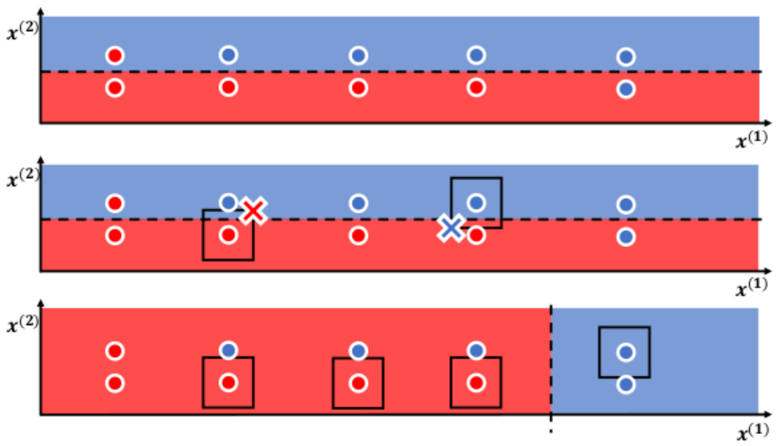

Figure 3.5: A simple example illustrating how robust splitting works: a) best splitting; b) best splitting under attack; c) best robust split.

(square boxes).

An early attack algorithm designed for single decision trees has been proposed by Papernot et al. [33], based on greedy search. To find an adversarial example, this method searches the neighborhood of the leaf which produces the original prediction, and finds another leaf labeled as a different class by considering the path from the original leaf to the target leaf, and changing the feature values accordingly to result in misclassification.

Zhang et al. [34] introduced an attacking strategy specifically designed to voting ensembles of binary decision trees with binary valued feature variables. Their approach depends on input and output analysis of the targeted system. They simplified the optimal evasion attack in [35] to a reduced and less costly strategy of $0-1$ Integer Linear Programming (ILP). Their optimal evasion problem became the minimization of ℓ_0 distance formulated in terms of $0-1$ ILP with constraints. The constraints ensure any successful attack will change the label (sign) of the output; ensure that there is one and only one decision making path; even with the introduction of attacks, the method preserves the semantics of binary decision tree. To complement their optimal evasion problem solution, Zhang et al. also introduced an heuristic evasion algorithm which finds the most important features which appears most frequently and closer to the root in decision tree, and greedily modify the single best feature at each iteration until the attack become successful to produce adversarial sample. This approach is also extended to black-box settings.

Chen et al. [36] studied the robustness verification problem for tree-based models, including decision trees, random forests (RFs) and gradient boosted decision trees (GBDTs). They applied two types of attack: the first one developed by Kantchelian et al. [35] and the second developed by Cheng et al. [37]. The first one is a white-box attack against binary classification tree ensembles. This method finds the exact smallest distortion (measured by some ℓ_p norm) necessary to mislead the model. The second one does not rely on the gradient nor the smoothness of output of a machine learning model. Cheng's attack method has been used to efficiently evaluate the robustness of complex models on large datasets, even under black-box settings. To deal with non-smoothness of model output, this method focuses on the distance between the benign example and the decision boundary, and reformulates the adversarial attack as a minimization problem of this distance. Some adversarial examples obtained by this method are shown in Figure 3.6, where we display results on both MNIST and Fashion-MNIST datasets. Cheng's attack is able to craft adversarial examples with very small distortions on natural models; for human eyes, the adversarial distortion added to the natural model's adversarial examples appear as imperceptible noise.

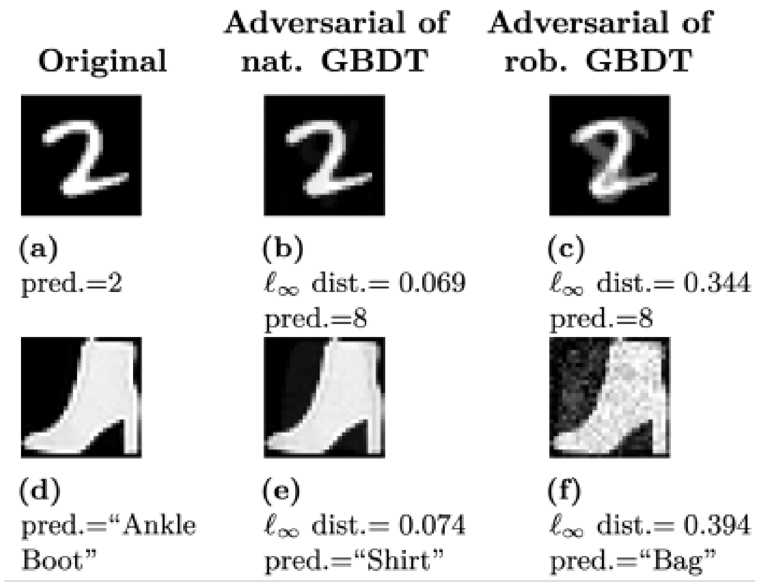

Figure 3.6: MNIST and Fashion-MNIST examples and their adversarial examples found using the untargeted attack proposed by [37] on 200-tree gradient boosted decision tree (GBDT) models trained using XGBoost with depth=8. Natural GBDT models are fooled by small ℓ_∞ perturbations(b,e); instead robust GBDT models require much larger perturbations (c, f) for successful attacks.

Chen et al. demonstrated that their robust model needed a greater ℓ_∞ perturbations to be fooled than the natural gradient boosted decision tree (GBDT) models. Since a typical decision tree can only split on a single feature at one time, it is natural to consider adversarial perturbations within an ℓ_∞ ball of radius ϵ around each instance $\mathbf{x}_i$. Such perturbations enable the adversary to minimize the score obtained by the split. So, instead of finding a split with highest score, an intuitive approach for robust training is to maximize the minimum score value obtained by all possible perturbations in an ℓ_∞ ball with radius ϵ. The framework defines $\Delta\mathcal{I}$ as the *ambiguity set* of instances that can fall to the right or on the left child of each node due to the attack application. For efficiency and scalability reasons, Chen's robust splitting procedure for boosted decision trees approximates the minimization by considering only four representative cases: no perturbations, perturb all points to the left, perturb all points to the right, swap the points. In this way, only a constant factor is added to the asymptotic complexity of the algorithm. The robust splitting is determined finding the minimum score of these four combinations. The split method proposed by Chen et al. has been implemented the code described in chapter 4.

Another novel algorithm, called TREANT, which uses optimization approach in optimal tree construction is presented in [1]. It optimizes an evasion aware loss function during the tree construction, by employing a formal threat model used to generate attacks. As the tree grows, this approach make sure that it is optimally constructed by means of splitting choices that are aware of the capabilities of the attacker. Considering the insufficiency of attack representation when an attacker is aware of the defense mechanism and the representation of the total possible attacks of the work in [36], the authors decided to base their algorithm on two main components. The first one is *robust splitting*, which identifies instances for which the outcome depends on the attacker's move, and evaluate the splitting quality. The second component is called *attack invariance*, a security property requiring that additional tree growth does not constitute a new attacking strategy for the attacker.

Chapter 4 is focused on the implementation of this algorithm.

3.5 Summary

In this chapter an overview of the possible attacks on NN and decision tree models has been presented. This presentation of adversarial machine learning can be summarized in the following points:

- adversarial machine learning is a novel field;
- some possible attack strategies (like white and black box attacks) has been described;
- there are few works in this area, but the interest is increasing due to the impact that ML algorithms can have in our lives in the near future;
- artificial neural network has been studied more than other ML models, probably because of their great success in the recent years and their low human readability;

- despite decision trees success in many applications, attack strategies and experiments on this kind of models are very few;
- the concept of *robust splitting* has a crucial importance in the defensive strategy of a decision tree model;
- some attack strategies on decision trees and some counter-measures has been presented;
- a defensive strategy for decision trees training called TREANT algorithm has been introduced.

Chapter 4

The Treant algorithm optimization

In this chapter it is discussed the core work of this thesis: the refactoring of a Python code implementing the TREANT algorithm [1].

After a brief description of the algorithm, the initial performance and design of the Python code are described as well as the strategies implemented to enhance its quality, readability and computation time in a new C++ implementation.

4.1 Description of the problem

The starting point of the work carried on in this thesis was a Python code used to solve the problem described by Calzavara et al. in [1]. The authors proposed a novel learning algorithm, called TREANT, designed to build decision trees which are resilient against evasion attacks at test time. Based on a formal threat model, TREANT optimizes an evasion-aware loss function at each step of the tree construction. This is particularly challenging to enforce correctly, considered the greedy nature of traditional decision tree learning. In particular, TREANT has to ensure that the local greedy choices performed upon tree construction are not short-sighted with respect to the capabilities of the attacker, who has the advantage of choosing the best attack strategy based on the fully built tree. TREANT is based on the combination of two key technical ingredients: a *robust* splitting strategy for decision tree nodes, which reliably takes into account at training time the attacker's capability of perturbing instances at test time, and an *attack invariance* property, which preserves the correctness of the greedy construction by generating and propagating constraints along the decision tree, so as to discard splitting choices which might be vulnerable to attacks.

The notation of symbols used in [1] is reported in Table 4.1.

The TREANT construction is summarized in algorithm 1. The core of the logic is in the call of the TSPLIT function (line 2), which takes as input a dataset $\mathcal{D}$, an attacker A and a set of constraints $\mathcal{C}$ initially empty, and implements the construction recursively. The construction terminates when it is not possible to further reduce $\mathcal{L}^A$ (line 3).

Symbol	Meaning
$\mathcal{L}^A$	Loss under attack
$\mathcal{D}$	Training dataset
$A(\mathbf{x})$	Set of all the attacks A can generate from $\mathbf{x}$
$\lambda(\hat{y})$	Leaf node with prediction $\hat{y}$
$\sigma(f, v, t_l, t_r)$	Internal node testing $x_f \leq v$, having sub-trees t_l, t_r
$\mathcal{D}_l(f, v, A)$	Left elements of ternary partitioning on (f, v)
$\mathcal{D}_r(f, v, A)$	Right elements of ternary partitioning on (f, v)
$\mathcal{D}_u(f, v, A)$	Unknown elements of ternary partitioning on (f, v)
$\mathcal{D}_L(f, v, A)$	Left elements of the robust splitting on $\hat{t}$ stump
$\mathcal{D}_R(f, v, A)$	Right elements of the robust splitting on $\hat{t}$ stump
$\mathcal{D}_L(f, v, A)$	Set of constraints for the left child of $\hat{t}$
$\mathcal{D}_R(f, v, A)$	Set of constraints for the right child of $\hat{t}$

Table 4.1: Notation summary

Algorithm 1 TREANT

```
Input: training data $\mathcal{D}$, attacker $A$, constraints $\mathcal{C}$
Output: Node of the tree structure (internal node or leaf)
1: $\hat{y} \leftarrow argmin_y \; \mathcal{L}^A(\lambda(y), \mathcal{D})$ subject to $\mathcal{C}$
2: $\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)), \mathcal{D}_l, \mathcal{D}_r, \mathcal{C}_l, \mathcal{C}_r \leftarrow$TSPLIT$(\mathcal{D}, A, \mathcal{C})$
3: if $\mathcal{L}^A(\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)), \mathcal{D}) < \mathcal{L}^A(\lambda(\hat{y}), \mathcal{D})$ then
4:     $t_l \leftarrow$TREANT$(\mathcal{D}_l, A, \mathcal{C}_l)$
5:     $t_r \leftarrow$TREANT$(\mathcal{D}_r, A, \mathcal{C}_r)$
6:     return $\sigma(f, v, t_l, t_r)$
7: else
8:     return $\lambda(\hat{y})$
9: end if
```

Function TSPLIT returns the sub-tree minimizing the loss under attack $\mathcal{L}^A$ on $\mathcal{D}$ subject to the constraints $\mathcal{C}$, based on the ternary partitioning. It then splits $\mathcal{D}$ by means of the robust splitting strategy and returns new sets of constraints, which are used to recursively build the left and right sub-trees. If the $\mathcal{D}_u$ set is not empty, then its elements will fall into $\mathcal{D}_L$ or $\mathcal{D}_R$ based on $(\hat{y}_l, \hat{y}_r)$ values resulting from the solution of the min-max problem. The constraints $\mathcal{C}_L$ and $\mathcal{C}_R$ are propagated downstream to the child nodes. During the tree construction, the implementation incrementally computes a sufficient subset of $A(\mathbf{x})$. A simple example of the algorithm execution can be found in [1].

The splitting method implemented in the TSPLIT function (line 2 in algorithm 1) solves an optimization problem that minimizes a cost function (also called *loss* *function*). In the scientific literature we can find many candidates for the loss function; among the others Gini's index, Entropy, SSE and Log-Loss. In particular, the loss function chosen in the Python code is the Sum of Squared Errors (SSE) and must be minimized with some constraints (see [1] for the mathematical formulation of the cost functions and the constraints imposed). The minimization problem is solved using the function `minimize` of the `optimize` package belonging to the `scipy` Python library. The authors identified in the `SLSQP` method the best suitable way to solve the problem at hand (for more details see par. 4.2).

In the end of the article (see section V of [1]), the algorithm has been tested on three real datasets:

1. CENSUS: Census Income (see [38])
2. WINE: Wine Quality (see [39])
3. CREDIT: Default of Credit Cards (see [40])

having the following main statistics:

Statistics type	CENSUS	WINE	CREDIT
n. of instances	45,222	6,497	30,000
n. of features	13	12	24
class distribution (pos.÷neg. %)	25÷75	63÷37	22÷78

Table 4.2: Main statistics of tested datasets

Even if the tested datasets are not very large, the original Python code is quite slow, especially if the target is not the construction of only one tree but an ensemble of decision trees like in the case of *bagging* classifiers.

For this reason, the code needed to be refactored with a translation to a compiled language since Python is an interpreted language and enhancing its performance using a multi-threading approach in the parts that could be parallelized. The chosen language was C++, a language able the go to a very low level and, in its modern version (the C++17 standard has been used), can easily generate parallel applications thanks to the powerful Standard Template Library (STL) where at least a couple of modules like `thread` and `future` can ease the life of the programmer avoiding the code complexity that was inevitable in former times.

The first step in this type of analysis is always a profiling of the existing code. On the Python side, it was used a powerful IDE distributed by JetBrains called *PyCharm*. In its Professional Edition available through the Student version of the University of Ca' Foscari license, provides a Python Profiler. The profiler is a very useful tool to identify the most expensive functions of the code. The refactoring should focus on these parts in order to get the maximum gain in terms of performance. The profiler typically provides some statistics, like: the space (memory) or time complexity, the usage of particular instructions or the frequency and duration of function calls. It is also able to show the results in a tree fashion where each function is plotted at its right stack level with a percentage related the ratio between the function CPU usage over the total application usage.

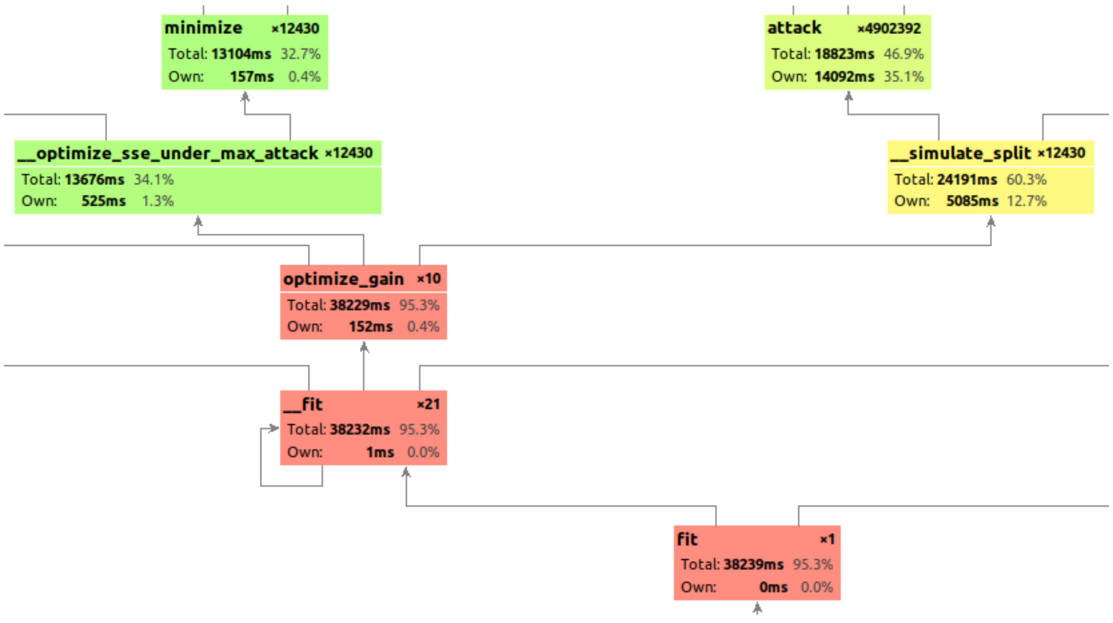

Figure 4.1: Example of *PyCharm* profiler results on the original code in graph form

The aim of the profiled code was training a single decision tree using a single thread. In the `sklean` Python package vocabulary, usually the training phase is performed by a function called `fit(X, y)` that takes as input a dataset in the form of a feature matrix `X` and a label vector `y`. The training phase is more time consuming than the test phase where the model built is used to predict a given set of instances. For this reason, all the analysis conducted is mainly focused on the traning phase. Matrices and vectors in the Python code are arrays built with the `numpy` library. The Figure 4.1 shows a portion of the profiler result given in the graph form. The red boxes are calls to functions that are heavily involved in the program execution. The colours degrade to yellow and green with the decrease of the CPU usage. We can see that in the training phase of a decision tree the `fit` function plays the main role. The `__fit` function is a private function that is called recursively (see self-pointing arrow in the box in the figure) because the decision tree holds a root node that is a recursive structure. So, the `fit` function must call the recursive function `__fit` and build the recursive structure of the tree formed by its nodes with a top-down procedure. The `optimize_gain` function takes almost all the computational burden of the `__fit` function (see percentages in the boxes). The `optimize_gain` function calls two fundamental function for the training problem at hand:

- `__simulate_split` (more or less two-thirds of load);
- `__optimize_sse_under_max_attack` (more or less one-third of load).

The first function simulates a split of the current portion of the dataset; just "a portion" because, during the training process, the instances contained in the initial dataset and falling inside the root node of the decision tree, are split recursively into two subsets and assigned downstream to two children until the requirements for a leaf creation are met (recall that the decision tree built is a binary tree, as described in [1]). Like in other algorithms developed in an hostile environment, the split function in the TREANT algorithm involves an attacker. For this reason, `__simulate_split` performs a loop on all the instances falling inside the node in order to generate a larger subset including the instances generated by the attack. This expanded set of instances is built calling the `attack` function of the `Attacker` class on each instance belonging to the node. The `__simulate_split`'s output are three subsets of the set of instances falling into the current node: the instances falling in the left child, the instances falling in the right child and the instances, if any, that can fall on the left or on the right child. The instances belonging to this last set are called *unknown* in [1] and their final destination (the left or the right child) is determined thanks to the output of the second function.

The `__optimize_sse_under_max_attack` function estimates two variables, $\hat{y}_l$ and $\hat{y}_r$, solving the min-max problem defined by equation (2) in [1] and reported hereafter:

$$(\hat{y}_l, \hat{y}_r) = \underset{y_l, y_r}{\operatorname{argmin}}\, \mathcal{L}^A(\sigma(f, v, \lambda(y_l), \lambda(y_r)), \mathcal{D}), \tag{4.1}$$

where (f, v) is a fixed pair representing a generic entry in the feature matrix (for the other symbols see Table 4.1).

Thanks to the $(\hat{y}_l, \hat{y}_r)$ solution of the min-max problem, the uncertainty due to the *unknown* instances is solved (see *Definition 3* regarding *Robust Splitting* in [1]) and the instances belonging to this set can be assigned to the pertinent child. This concludes the description of `fit(X, y)`, the most expensive function in the original Python code.

Other functions have been considered during the refactoring, for example the `predict` that is needed in the test phase and for the construction of a Python module created by the C++ code and usable as estimator in the `BaggingClassifier` class of the `sklearn` Python library.

Python classes like `Attacker`, `Node`, `Constraint`, `SplitOptimizer` among the other, have their C++ counterpart in the refactored code for the sake of consistency and readability. In this way, a user of the original Python code can find easily the references in the C++ code.

4.2 Solving the optimization problem

The construction of a robust decision tree involves the solution of the optimization problem described by equation 4.1.

Python library `scipy` provides a useful function in the `optimize` package called `minimize`. This function provides several methods taken from the scientific literature, among the other: `Nelder-Mead`, `Powell`, `CG`, `BFGS`, `Newton-CG`, `L-BFGS-B`, `TNC`, `COBYLA`, `SLSQP`. The `SLSQP` method uses Sequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The Python method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [41]. As a side note, the wrapper handles infinite values in bounds by converting them into large floating values.

In order to solve the same optimization problem in C++, the open-source `NLopt` library has been compiled separately and linked in the final phase of the compilation process. The target of this library is to solve non-linear optimization problems [42]. As the `scipy` Python library, it provides several methods to solve those problems. For the sake of consistency, the same solution method has been chosen: `SLSQP`. Some methods are derivative-free and the calculation of the jacobian matrix is optional, some other methods are gradient-based and they need the declaration of the gradient function that will be used internally in order to calculate the jacobian.

On one side, the C++ library `NLopt` implements the `SLSQP` method as gradient based, on the other side the `scipy.optimize.minimize` function implements the `SLSQP` method as derivative-free. In other words, only the function to be minimized should be provided and not the gradient function. In order to ensure consistency between the Python and C++ version, especially during the unit tests described in the chapter 4.5, the same procedure followed by `scipy.optimize.minimize` to calculate the jacobian matrix has been implemented in the function that calculates the gradient on the C++ side.

It is worth noting that, due to a strange mismatching between the two solution of the optimization problem, the two decision trees calculated by the Python code and by the C++ code presented a slight difference in one particular node. The case here discussed regards the CENSUS dataset with 5000 instances analysed and an attacker with a budget of 60.

Since this was a good test case for the solution of the optimization problem, it has been investigated step-by-step. Curiously, even if the input was exactly the same and with the same internal parameters (starting point, maximum number of iteration and solution tolerances), the two functions calculated two different minimum values. The minimum calculated by `NLopt` was less than the one calculated by `scipy.optimize.minimize`. For this reason, a last test was conducted: the initial value of the function, x_0, given as input to the Python function has been modified with the minimum found by the C++ code and, surprisingly, the code yielded to the same minimum calculated without modifying the value of x_0. So, even if starting from a lower point (the C++ minimum), the minimization algorithm yielded to a greater solution, that unfortunately is not the expected behaviour (see Table 4.3).

NIT	FC	OBJFUN	GNORM
1	4	1.003093E+00	8.672082E-01
2	9	1.059868E+00	1.602760E-00
3	13	3.279960E+00	8.579736E-00
4	17	5.383091E+00	1.192596E+01
5	23	5.386212E+00	1.187600E+01
6	28	5.933486E+00	1.259119E+01
7	32	6.556814E+00	1.335931E+01
8	36	6.650636E+00	1.347121E+01
9	40	6.652721E+00	1.347368E+01

Table 4.3: Iterations of the Python minimizer in a particular test case: even if the iterations begin with a good starting point x_0 with the lowest value of the GNORM, the error increases and the algorithm ends up with a solution that is worse than the initial one (the optimum found by the C++ code)

From this analysis, at least in this particular case, the C++ version of the `SLSQP` method was better than the Python version.

Giving a closer look to the `scipy.optimize.minimize`, it seems that an external Fortran function belonging to an external library is called. Most likely, this was the original function developed by Kraft in [41]. According with the `NLopt` online documentation, the original Kraft's code was modified for its inclusion by S. G. Johnson in 2010 with the following major changes: the inexact line search was modified, a bug on the LSEI subroutine was fixed and the LSQ subroutine was modified to handle infinite lower/upper bounds. These changes for sure improved the code quality and can be the reason of the inconsistencies encountered in the case discussed above.

4.3 Multi-threading

The first step of the refactoring involved the translation of the code to a compiled language: C++.

The second step involved the parallelization of the code in order to exploit the great resources that the modern computers can provide. The final target of any machine learning developer is, usually, bring his code to a level able to be launched on a cluster with the possibility to crunch a big amount of numbers per second generating the expected ML model as fast as possible. This would give to the researcher the possibility to run more tests of his algorithm, increase the robustness in a quicker way and open the code to the solution of bigger challenges.

The hardware used to test the code in its sequential and parallel versions had the following technical characteristics:

```
Architecture:         x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   2
Core(s) per socket:   4
Socket(s):            1
NUMA node(s):         1
Vendor ID:            GenuineIntel
CPU family:           6
Model:                142
Model name:           Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Stepping:             11
CPU MHz:              2944.316
CPU max MHz:          4600,0000
CPU min MHz:          400,0000
BogoMIPS:             3999.93
Virtualisation:       VT-x
L1d cache:            32K
L1i cache:            32K
L2 cache:             256K
L3 cache:             8192K
NUMA node0 CPU(s):    0-7
```

The OS used was an Ubuntu 18.04 Linux distribution. As described in the specifications above the hardware has the ability to run multiple threads concurrently, so it is enough to have a first understanding of the horizons opened by the new code. The code developed and described in this thesis is the first step of this process toward an High Performance Computing code design; for the time being, the code can be launched using a multi-threading approach in more than one way as described in the following. A future enhancement could be the multi-process support adding the possibility to run the code on a cluster of machines and increase even more its performance.

As described in chapter 4.1, the profiling analysis shown that the most expensive function in the training phase is `__simulate_split`. This function produces a split of the dataset based on a particular feature value. The function `__fit` loops through all the columns of the feature matrix (*outer loop*) and calls the `__simulate_split` function on each value assumed by the feature at hand using a nested loop (*inner loop*).

Python loops are, in general, very time consuming if compared with C++ ones. Translating the `__fit` function to C++ gave a considerable boost to the efficiency, but a further step can be done: parallelize the *outer loop* on columns. So, instead of processing each column sequentially we can split the column indexes in chunks and assign each chunk to a specific thread. We call this strategy *internal parallelization*, because it is performed at the single tree construction level.

Another kind of parallelization can be implemented during the creation of an ensemble model. For example, Random Forests and Bagging Classifiers are methods involving the creation of multiple decision trees. In this case the indexes of the trees that are going to be created can be split into chunks and assign each of this chunk to a thread responsible for the creation of all trees belonging to its chunk. This kind of strategy is called *external parallelization* in the following.

Finally, an *hydrid parallelization* can be generated mixing the two strategies described above: for example the training of an ensemble of four trees can involve one thread for the creation of the first two trees (first chunk) that internally run other two support threads for the creation of each tree belonging to its chunk, one thread for the creation of the last two trees (second chunk) that internally run other two support threads for the single decision tree construction.

C++ multi-threading support can be implemented in different ways. The first attempt involved *OpenMP* (Open Multi-Processing) library [43] that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The C++ code was linked properly but some issues arose during the Python binding implementation. For this reason, the final C++ code supports multi-threading through the use of Standard Template Library modules: `future` module has been used for *internal parallelization* and `thread` module has been used for *external parallelization*. Even if a little bit trickier than *OpenMP API* from the implementation point of view, the use of STL modules limits the number of external dependencies, that, in general, denotes a good code design.

Chapter 4.5 presents some test cases involving both *internal* and *external* parallelizations.

4.4 Memory management

The dataset is represented in Python by a matrix of features `X` and a vector of labels `y`, both defined as `numpy` array.

The co-existence of numerical and categorical variables in the `numpy` array representing the feature matrix has been solved using the Standard Template Library data structure `std::unordered_map<std::string, double>` that maps the categorical features expressed as unique strings to floating point numbers. These passages are not needed a Python code because a `numpy` array accepts multiple types of variables; e.g. integers, floating point numbers and strings. In order to create a proper dataset reading the data from a file, two support functions has been implemented:

-
  ```
  std::pair<unsigned, std::vector<std::string> >
  getDatasetInfoFromFile(const std::string &datasetFilePath)
  ```

-
  ```
  std::pair<std::vector<std::string>, std::vector<std::string> >
  fillXandYfromFile(double *X, const unsigned rows,
  const unsigned cols, double *y,
  const std::string &datasetFilePath)
  ```

The first one reads the number of rows and the column names from the file storing the dataset, the second one fills with the data the `X` and `y` arrays that are supposed to have been allocated before thanks to the information provided by `getDatasetInfoFromFile`. This process seems to be a little bit cumbersome, but it is needed in order to create a shared library and use the code not only in a pure C++ environment but also from Python where the memory of `X` and `y` is externally managed (see chapter 4.6). The feature matrix of the dataset is stored in the one-dimensional array `X` column-wise. This choice is very useful in order to speed up the *outer* loop of the `__fit` function described in chapter 4.3. Switching from one column to another can be easily done using pointer offsets.

The `attack` function is another function heavily used by the code when the construction of the decision tree involves the presence of an `Attacker` with budget.

This function is applied to the instances of the dataset, in this case a row-wise ordering of the features in the dataset could be more effective.

A test case has been developed to prove this behaviour using artificial datasets created with a random number generator. The maximum depth of the tree is set to 2, so the internal structure of the decision tree is only one internal node, the root, and two leaves.

n. of features	cache-misses [millions]	time elapsed [s]
128	3.2	3.14
256	19.8	10.79
512	353.9	38.21
1024	7678.2	267.48
2048	69265.7	2155.81

Table 4.4: Cache analysis of random generated dataset with 128, 256, 1024, 2048 features. The number of instances is always 1000 and no multi-threading support has been used. The cache-misses reported are concentrated in the `attack` function.

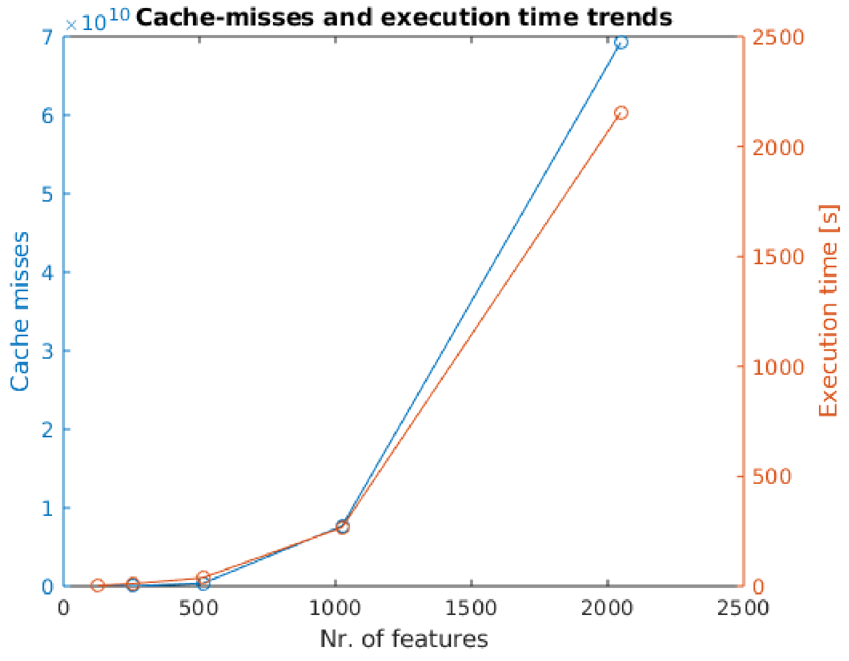

Figure 4.2: Cache analysis of random generated datasets with variable number of features

Table 4.4 shows how important is the cache management during the attack in the case of instances having a big number of features. Keeping constant the number of instances (1000) and the number of used threads (1), the cache-misses and the running times are increasing super-linearly with the number of features. The reason is the fetching of an instance during the attack (most of the cache-misses are related to `attack` function), unfortunately the values already present in the cache are hardly re-usable. This yields to a large number of cache misses. Figure 4.2 shows a strong correlation between cache misses and elapsed time during the execution of the program.

The same problem can arise whenever we pass a matrix to the C++ code with the wrong optimal order. For example, the `predict(X)` function where a set of instances is passed as input. In this case the input matrix should be stored in row-wise order because each instance, i.e. each row, is analysed sequentially and the data already present in the cache can be re-used. The `predict(X)` function works also when the input matrix is stored with a column-wise order, even if this is not the optimal order.

A possible enhancement could be hidden inside the attack logic: are all the features needed to perform an attack on a single instance? If the needed value is only the one of the attacked feature, than the memory management can be improved.

There is always a trade-off between memory storage and data race conditions: sometimes storing more data than the strictly needed can lead to better CPU performance, even if at the first glance the extra memory needed can be seen as a waste of space. For example, duplicating the memory accessed in read-only mode by two different threads avoids data-race conditions, but at the cost of doubling up the space needed to run the program.

The cache analysis performed in this chapter was possible thanks to the `perf` software available on Linux distributions.

4.5 Code validation

The C++ version of the code has been tested on the CENSUS dataset with different configurations: changing the attacker rules, changing the number of instances considered, changing the number of threads used to build the robust decision tree. In order to check the correctness of the code, many models were generated with both codes and then compared; `save` and `load` functions has been provided in order to save/load the trained decision trees to/from a given file path. These test cases were very useful to find possible bugs in the C++ code.

The following snippet shows on example of a decision tree file representation:

```
Prediction:0,Score:0.25684930465943501,Num_instances:292,Loss:55.736301369862943
Prediction:1,Score:0.99999987106504784,Num_instances:10,Loss:1.6624221888340663e-13
[0,1]Feature_ID:10,Threshold:1740,Num_instances:302,Loss:55.736301369863085,Gain:5.3398575705348819,Num_constraints:0
Prediction:1,Score:0.99999996800328261,Num_instances:18,Loss:1.8428218623484035e-14
[2,3]Feature_ID:9,Threshold:5013,Num_instances:320,Loss:61.076158940397967,Gain:8.770716059602556,Num_constraints:0
Prediction:0,Score:-2.7808630443829319e-08,Num_instances:2,Loss:1.5466398543229416e-15
Prediction:0,Score:1.2866744451594745e-08,Num_instances:2,Loss:3.3110622556528832e-16
Prediction:1,Score:0.8196721393986538,Num_instances:122,Loss:18.03278688524588
[6,7]Feature_ID:5,Threshold:9,Description:Other-service,Num_instances:124,Loss:18.03278688524588,Gain:1.3220518244315045,Num_constraints:0
[5,8]Feature_ID:5,Threshold:23,Description:Craft-repair,Num_instances:126,Loss:19.354838709677388,Gain:1.280081925243266,Num_constraints:0
[4,9]Feature_ID:3,Threshold:12,Num_instances:446,Loss:90.4817956349209,Gain:20.121343378533723,Num_constraints:0
Prediction:0,Score:0.025882355661001211,Num_instances:425,Loss:10.71529411764701
Prediction:1,Score:1.0000000192178116,Num_instances:1,Loss:3.6932428132215565e-16
[11,12]Feature_ID:10,Threshold:2205,Num_instances:426,Loss:10.71529411764701,Gain:0.94667771333884332,Num_constraints:0
Prediction:0,Score:0.30952379946335418,Num_instances:42,Loss:8.9761904761904905
Prediction:0,Score:0.088235270427997087,Num_instances:68,Loss:5.4705882352941577
[14,15]Feature_ID:8,Threshold:5,Description:Male,Num_instances:110,Loss:14.446778711484596,Gain:1.2714031066972264,Num_constraints:0
[13,16]Feature_ID:3,Threshold:11,Num_instances:536,Loss:27.38015364916755,Gain:1.8269359030714938,Num_constraints:0
Prediction:1,Score:0.94444439509149503,Num_instances:18,Loss:0.94444444444448894
[17,18]Feature_ID:9,Threshold:4650,Num_instances:554,Loss:30.151533996683511,Gain:13.689621238000115,Num_constraints:0
[10,19]Feature_ID:4,Threshold:11,Description:Married-civ-spouse,Num_instances:1000,Loss:154.4442942481401,Gain:33.554705751863878,Num_constraints:0
```

Implicitly each line of the block is 0-indexed and each line represents a node having its line number as implicit index. The floating point numbers are represented with an accuracy of ten decimals in order to assure a good precision when the tree is reconstructed using the `load` function. The internal nodes starts with a pair of integers inside squared brackets. This pair refers to the children, left and the right respectively, of the internal node. Leaves do not start with such a pair and contain different attributes, like prediction and prediction score.

The input dataset, CENSUS, was considered a good test for the code because of its characteristics: a good number of features, a good number of instances and both feature types (numerical and categorical).

The basic attacker is described in the following json structure file:

```
{
    "attacks": [{
            "workclass": [{
                "pre": "('Never-worked')",
                "post": "Without-pay",
                "cost": 1,
                "is_numerical": false
            }]
        },
        {
            "marital_status": [{
                "pre": "('Divorced', 'Separated')",
                "post": "Never-married",
                "cost": 1,
                "is_numerical": false
            }]
        },
        {
            "occupation": [{
                "pre": "('Craft-repair', 'Prof-specialty', 'Exec-managerial', ...
                     'Adm-clerical', 'Sales', 'Machine-op-inspct', 'Transport-moving', ...
                     'Handlers-cleaners', 'Farming-fishing', 'Tech-support', ...
                     'Protective-serv', 'Priv-house-serv', 'Armed-Forces')",
                "post": "Other-service",
                "cost": 1,
                "is_numerical": false
            }]
        },
        {
            "education_num": [{
                "pre": "(13, 16)",
                "post": -1,
                "cost": 20,
                "is_numerical": true
            }]
        },
        {
            "capital_gain": [{
                "pre": "(0, np.inf)",
                "post": 2000,
                "cost": 50,
                "is_numerical": true
            }]
        },
        {
            "hours_per_week": [{
                "pre": "(0, np.inf)",
                "post": 4,
                "cost": 100,
                "is_numerical": true
            }]
        }
    ]
}
```

Listing 4.1: Basic attacker used in the validation tests

As shown in Listing 4.1, the attacker is constructed from a list of rules, each rule is related to the name of a feature, the `pre` field indicates the values that can assume the feature before the attack and the `post` field is the value of the feature after the attack. The `cost` field indicates the cost that an attacker should pay in order to apply that rule and the `is_numerical` flag determines is the feature is numerical or categorical.

In the case of the basic attacker defined in the Listing 4.1, we define six rewriting rules: (i) if a citizen never worked, he can pretend that he actually works without pay; (*ii*) if a citizen is divorced or separated, he can pretend that he never got married; (*iii*) a citizen can present his occupation as a generic "other service"; (*iv*) a citizen can cheat on his education level by lowering it by 1; (*v*) a citizen can add up to \$2,000 to his capital gain; (*vi*) a citizen can add up to 4 hrs per week to his working hours. We let (*i*), (*ii*), and (*iii*) cost 1, (*iv*) cost 20, (*v*) cost 50, and finally (vi) cost 100 budget units. The applied attacker encompasses both feature types, numerical and categorical, as well as different attack rule costs.

The header-only `nlohmann` library has been used for the json structure parsing.

4.5.1 Single thread

The test cases presented in this section were generated using only one thread because the purpose was proving the consistency of the models generated by the new C++ code compared with the ones generated by the original Python code.

i	n. of instances	budget	time C++ [s]	time Python [s]	Speed-up
0	1000	5	0.867	24.99	28.82
1	1000	20	0.924	24.79	26.83
2	1000	40	0.838	25.95	30.97
3	1000	60	0.907	25.98	28.64
4	5000	0	11.62	345.46	29.73
5	5000	60	15.79	440.68	27.91
6	10000	60	56.61	1336.65	23.61

Table 4.5: Validation tests results.

The case number 5 in Table 4.5 presented a slight difference in the resulting model of the C++ code caused by the different optimizer used in the two codes. This case has been discussed in details in chapter 4.2. All the other unit tests were passed successfully generating models completely consistent with the ones generated by the original Python version.

Figure 4.3 shows an average speed-up greater than `x28`, in some cases greater than `x30`, obtained from the translation of the Python code to C++. The speed-up can be even increased using the multi-threading support of the C++ code as demonstrated in sections 4.5.2 and 4.5.3.

Following the cases presented in [1], more tests were performed on the three datasets: CENSUS, WINE, CREDIT. The attack configurations are consistent with the ones reported in the article. The maximum depth imposed is 4 and the maximum number of instances per node is 20. The number of features considered are a portion of the ones reported in Table 4.2: 27,000 for CENSUS; 4,000 for WINE and 18,000 for CREDIT.

23.07 22.27 21.73 23.24 22.13 22.22 19.60 20.19 20.84

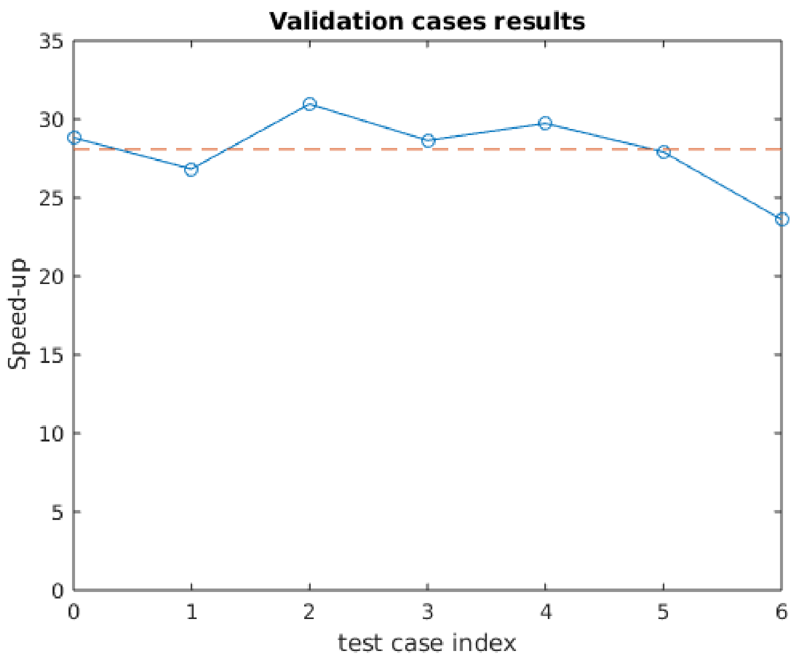

Figure 4.3: Validation test cases speed-up with a single thread. The test case index is the one reported in table 4.5.The average value is represented with a dashed line.

Also in these cases involving greater training sets, the performance improvements gained with the C++ code are quite remarkable (see Figure 4.4). The average of the running times per dataset has been chosen because the values were quite close to each other. Increasing the number of instances can also change the load balancing among the feature columns of the training matrix as well as the number of constraints to be imposed in the optimization problem when the attacker budget enlarges the set of *unknown* instances.

4.5.2 Internal parallelization

The same decision tree can be generated also using a parallel strategy in the `__simulate_split` function treating chunks of columns in different threads instead of having only one big chunk as in the single thread approach.

This parallel strategy can be strongly affected by a different load balancing between columns. The column computational load depends mainly on two factors: the number of unique values and the type of attack that can be performed on that column.

For example, in the CENSUS dataset there are 13 features. Considering only the first 10,000 instances, one feature is quite unbalanced because has 8471 unique values, one or two orders of magnitude greater than the unique values counted in the other features.

The time spent on a single column by the unlucky thread holding the most

Dataset name	budget	time (Python) [s]	time (C++) [s]	Speed-up
census	30	7517.45	456.70	16.46
	60	7296.28	456.46	15.98
	90	7324.80	466.38	15.71
	120	7367.98	477.06	15.44
wine	20	50.07	2.17	23.07
	40	56.57	2.54	22.27
	60	61.70	2.84	21.73
	80	67.64	2.91	23.24
	100	69.70	3.15	22.13
	120	69.98	3.15	22.22
credit	30	384.28	19.596	19.60
	40	396.95	20.185	20.19
	60	407.31	20.840	20.84

Table 4.6: Training of robust decision trees using a single thread and real datasets.

expensive column on his chunk can be much bigger than the time spent by the other threads on the other columns. In this case the power of the multi-threading approach is lost and the computation time of any multi-threading approach is more or less the same of the single thread approach. Sometimes the overhead brought by the creation of more than one thread can lead to elapsed times even greater than the single thread case (see Table 4.7).

max depth	n. of threads	time elapsed [ms]
1	1	15339
1	2	14684
1	4	14613
2	1	32528
2	2	33183
2	4	29557
3	1	48250
3	2	51753
3	4	45718
4	1	66666
4	2	65338
4	4	62990

Table 4.7: CENSUS dataset with 10000 instances and the basic attacker with a budget of 100 and multi-threading approach

Table 4.8 shows that during the formation of the first node (the root node) where all the instances are taken into account, the time spent by the thread 0, that has to analyze the column number 2, is clearly outlier. The thread 0 has to analyse the most expensive column of the dataset and represent the bottleneck in the decision tree formation.

Another experiment has been performed with a random dataset of 300,000

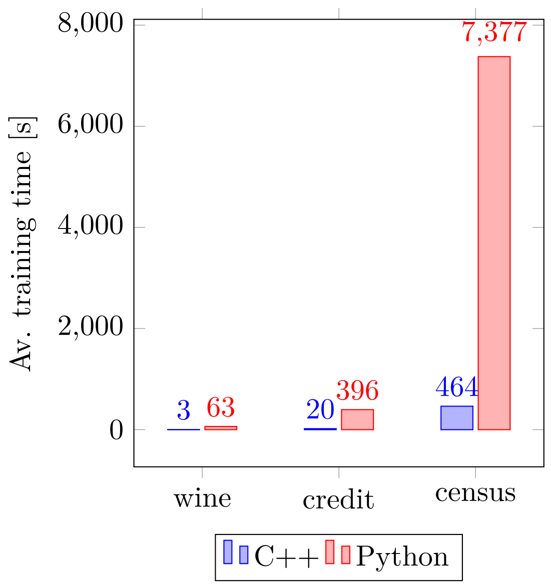

Figure 4.4: Average computation times for decision tree training on the three real datasets (WINE, CREDIT and CENSUS)

instances and 16 features. Since it is a random dataset the load between the columns is balanced. The attacker budget was set to zero in order to not affect the balance between the different features.

Different decision trees with different depths has been trained, but the speed-up recorded follows almost the same trend for all the 4 different type of internal tree structures. The results reported in Figure 4.5 and in Table 4.9 confirm that work on the columns is balanced the algorithms exploit the multi-threading approach scaling quite well with the number of employed threads.

column ID	unique values	thread ID	analysis time [ms]
0	71	0	14865
1	7		
2	8471		
3	16	1	168
4	7		
5	14		
6	6	2	45
7	5		
8	2		
9	96	3	1205
10	66		
11	83		
12	40		

Table 4.8: First split analysis on CENSUS dataset: 4 threads

max depth	n. of threads	time elapsed [s]	Speed-up
1	1	101.845	1.00
1	2	61.606	1.65
1	4	38.752	2.63
2	1	215.919	1.00
2	2	132.433	1.63
2	4	85.913	2.51
3	1	337.276	1.00
3	2	196.173	1.72
3	4	128.153	2.63
4	1	441.504	1.00
4	2	277.455	1.59
4	4	180.447	2.45

Table 4.9: Random generated dataset of 300,000 rows and 16 features used to generate a decision tree with 1, 2 and 4 threads

4.5.3 External parallelization

The Python class `RobustDecisionTree` is a subclass of the `BaseEstimator` class, part of the `sklearn` package. Thanks to its parent, `RobustDecisionTree` can be an estimator of the `BaggingClassifier` class that is used to build an ensemble model. The `BaggingClassifier` class can build its set of decision trees in parallel specifying the number of `jobs` used during the training phase. The same number of jobs can be used in the predict phase where, from a set of instances, the model is able to classify each instance on a voting system that makes use of all the decision trees inside the ensemble model.

Imitating this behaviour, a `BaggingClassifier` class has been designed also in the C++ code; this class stores a set of decision trees. The training of these

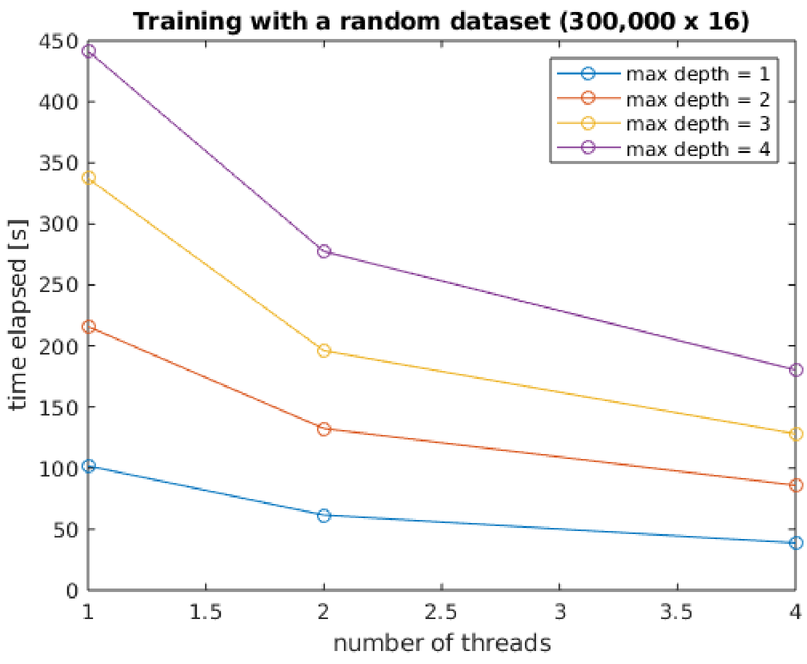

Figure 4.5: Training of a decision tree from a random dataset of 300,000 instances and 16 features using 1, 2, 4 threads

trees can be carried on concurrently using more than one thread. In this way the trees are not trained sequentially but in parallel. The instances used for the training of the single decision tree are randomly sampled from the whole dataset with or without replacement. A percentage determining the number of elements in each sample can be set. This kind of parallelization where more than one tree is trained concurrently is denoted as *external parallelization* (see chapter 4.3).

The instances are sampled randomly, so, quite likely, the load for the training of a single decision tree should be approximately the same.

Here we revisit the example discussed in chapter 4.5.2 where the CENSUS dataset with 10,000 instances has been trained in an adversarial environment with an attacker having a budget of 100. In that case, the training of a single tree with multi-threading support on the features gave a poor results, because of an unbalanced column. The `BaggingClassifier` divides the decision trees to train in chunks and each chunk is assigned to one thread.

Table 4.10 and Figure 4.6 show how the elapsed time during the training of an ensemble model scales well when the the number of used threads is increased. The model had internally 8 estimators trained with samples having the 50% of the total instances present in the dataset. Only the *external* parallelization strategy has been applied. The good speed-up obtained with the training of the ensemble model demonstrate that this kind of parallelization is able to distribute more

max depth	n. of threads	time elapsed [s]	Speed-up
1	1	69.295	1.00
1	2	40.223	1.72
1	4	25.601	2.71
2	1	138.535	1.00
2	2	82.364	1.68
2	4	52.423	2.64
3	1	213.442	1.00
3	2	122.088	1.75
3	4	81.212	2.63
4	1	323.428	1.00
4	2	170.771	1.89
4	4	108.506	2.98

Table 4.10: BaggingClassifier trained with 8 estimators, sample ratio of 0.5, budget 100

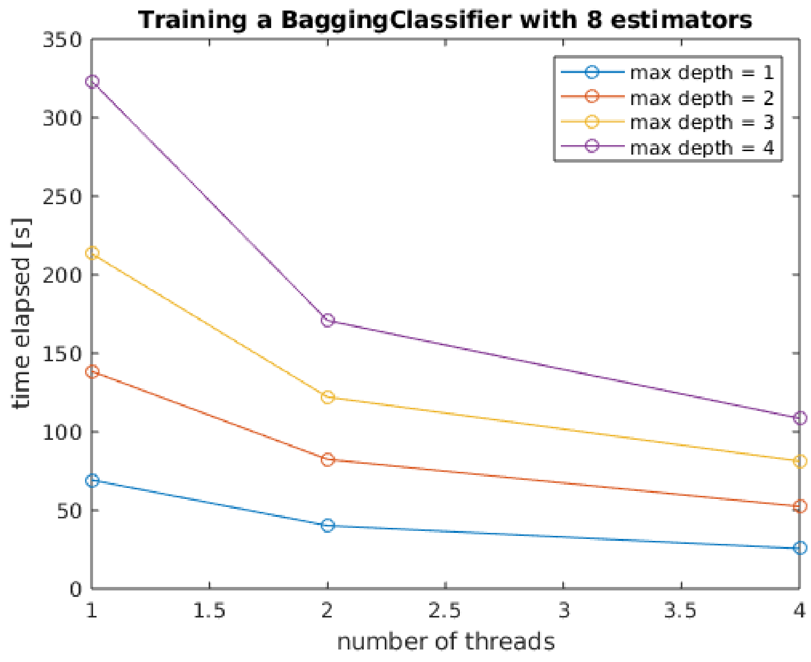

Figure 4.6: Training of an ensemble model with 8 estimators using 1, 2, 4 threads

uniformly the load on the threads even with unbalances on the features inside the training set. This kind of parallelization is a valid alternative especially when the *internal parallelization* does not scale well.

4.6 Python Binding

A C++ library can be called and used from Python, but there must exist a middle layer in order to bind the two languages. Many tools has been developed to solve this problem, like `Cython`, `SWIG` and `ctypes`. C++ compilation pipeline is managed with `CMake`, that is a cross-platform tool able to manage the deploying of C++ applications. As suggested in [44], `CFFI` (C Foreign Function Interface) is a good tool to bind an already existing C/C++ library. The `CFFI` goal is to call C code from Python without learning a third language. Thanks to its design, `CFFI` requires users to know only C and Python, minimizing the extra bits of API that need to be learned. In `CFFI` there is no C++ support. Sometimes, like in the Treant case, it is reasonable to write a C wrapper around the C++ code and then call this C API with `CFFI`.

The class between Python and C++ is called `PyDecisionTree` and it encapsulates the `DecisionTree` class defined in the C++ library. The Python class must implement two fundamental methods:

- `fit(X,y)`
- `predict(X)`

in order to be a subclass of the `BaseEstimator` class and be used in combination with the `sklearn` package for the construction of ensemble model.

The `predict_proba(X)` method is the same of the original Python code but uses the `predict(X)` bound to the new developed Python module. To complete the methods exposed, there are also:

- `load(file_path)`
- `save(file_path)`

that load/save a model from/to a specified file. The strings are passed as streams of bytes.

When a vector of strings is needed in the C++ code, it is passed instead a long string where all the entries are concatenated by a comma. This long string is then parsed on the C++ side to derive all single entries of the vector.

The main application must be responsible for the management of the arrays passed to the C++ library. Matrices and vectors are passed as raw C pointers. In the case of matrices the ordering must be determined by the main application. Also `numpy` arrays can be passed as row pointers. The following snippet shows how to get the raw C pointer referring to the data of a `numpy` matrix:

```python
import numpy as np
import FFI

rows = 1000
cols = 1000
X = np.zeros((rows, cols), dtype='float64')
if not X.data.f_contiguous:
    np.asfortranarray(X, dtype='float64')
pointer_x = FFI().cast('double*', X.ctypes.data)
```

The column-wise ordering is also enforced in this case. There must be consistency between the Python type, `float64`, and the C type, `double` of the array data. Once the raw C pointers are available, the functions on the middle layer can be called from Python (see appendix B). It is worth to remind that in some applications it is good to have a row-ordering (for example using the `predict(X)` function) whether in the other cases it is mandatory a column-ordering of the input matrix (for example using the `fit(X,y)` function). For this reason it is better to wrap all this methods in a class that internally manages the storage of the arrays and their ordering. Detailed examples of applications using the generated Python module are reported in appendix B.

Chapter 5

Conclusion and future work

The original Python code implementing the Treant algorithm has been successfully translated to C++ and validated using real training sets. The construction of the robust decision tree has been studied also with different attack scenarios that involve the solution of an optimization problem with the imposition of multiple constraints. The resulting trained model, in most of the unit tests, matched perfectly, only few differences were recorded in the construction of some nodes due to the different optimizer implementation: Python code and C++ use different libraries to solve the min-max problem. In these cases, some limitations in the Python library that can lead to unexpected behaviours has been found. These inconsistencies proved that the new C++ code uses a better optimizer.

The code has been analysed from the memory point of view: the crucial dataset storing strategy has been analyzed and optimized. The code has been tested also with random generated dataset for a better understanding of the algorithm behaviour when the training set has a big number of features.

The C++ code provided a significant boost to the algorithm performance, the execution times using only one thread are 28 times smaller than the ones recorded with the Python version. The multi-threading approach on the training set features can even further increase the performance: when the computational load is balanced among the feature columns, the execution running times scale quite well increasing the number of involved threads. Usually, feature unbalancing can be caused by the feature type (typically categorical features have less unique values than numerical features) or by the attack applied on that feature (cost and *pre* condition of an attack rule are parameters that control the aggressiveness of the attacker).

An example of ensemble model class has been provided and tested: the tests carried on proved that, when the load on feature columns is unbalanced, parallelizing over the training of the ensemble estimators can yield to a very good speed-up.

In some cases, the logic behind the attack on a single instance can be improved from the memory point of view: if the attacker needs to know only the value of a single feature and its cost, then retrieving all the instance features is a waste of memory and can yield to a lot of cache misses. Storing of all the possible attacks on the dataset before running the application is another strategy that can improve a bit the performance of the training phase, nevertheless it can have a great memory cost.

Considering the case of multi-threading on features, when an unbalance is present then the task organization among the threads can be improved: an alternative strategy can distributed uniformly among the threads the unique features values. Future development of the C++ code can include multi-processing support with `MPI` library; in this way the code could also be launched on a cluster of machines.

If the original Python code will still be maintained, a possible enhancement can involve the integration of `NLopt` library, the one used in the C++ code. `NLopt` authors provide a Python wrapper for that library.

Appendix A

Code distribution

A.1 Random datasets generation

In chapter 4.4 has been tested the C++ code with random generated datasets, these dataset has been generated using the following simple Python code.

```
import numpy as np

cols = 16
rows = 300000
header_list = ['feature_%d' % i for i in range(0, cols + 1)]
header_list[-1] = "LABEL"
header_str = ' '.join(header_list)
X = np.random.randint(100, size=(rows * (cols + 1))).reshape(rows, ...
    cols + 1)
X[:, -1] = X[:, -1] % 2
output_file = "data/random_%d-%d.txt" % (rows, cols + 1)
np.savetxt(output_file, X, fmt='%d', header=header_str, delimiter=' ', ...
    comments='')
```

A.2 Compilation process workflow

The code has been compiled using CMake [44]. The `nlohmann` header-only library has been used for json parsing (see `include/` folder). The `NLopt` library, used to solve the optimization problem, has been compiled and installed in the `3rdparty/` folder. The C++ Standard Template Library, distributed usually with the C++ compiler, is widely used through the project.

The folder `py_decision_tree` contains the code for the Python binding. The `data/` folder contains typically the training sets, the attacker file and, in general, all the input files needed to run the program. In the following the directory tree of the project.

```
treant/
 |__3rdparty/
 |   |__nlopt-2.6.1/
 |       |__include/
 |       |__lib/
 |__data/
 |__include/
 |   |__Attacker.h
 |   |__BaggingClassifier.h
 |   |__Constraint.h
 |   |__Dataset.h
 |   |__DecisionTree.h
 |   |__FeatureColumn.h
 |   |__Logger.h
 |   |__nlohmann
 |   |__json.hpp
 |   |__Node.h
 |   |__SplitOptimizer.h
 |   |__types.h
 |   |__utils.h
 |__main.cpp
 |__py_decision_tree/
 |   |__implementation/
 |   |   |__c_cpp_interface.cpp
 |   |   |__cpp_implementation.cpp
 |   |   |__cpp_implementation.hpp
 |   |__interface_file_names.cfg
 |   |__CMakeLists.txt
 |   |__py_decision_tree.h
 |__README.md
 |__src/
 |   |__Attacker.cpp
 |   |__BaggingClassifier.cpp
 |   |__Constraint.cpp
 |   |__Dataset.cpp
 |   |__DecisionTree.cpp
 |   |__FeatureColumn.cpp
 |   |__Logger.cpp
 |   |__Node.cpp
 |   |__SplitOptimizer.cpp
 |   |__utils.cpp
 |__test_py_decision_tree.py
 |__CMakeLists.txt
 |__tests/
```

Appendix B

Examples of usage

The following section is dedicated to present some use cases of the C++ library and the generated Python module. The include part in the C++ code is avoided.

B.1 Python

In order to use properly all the methods exposed by the Python module, an intermediate Python class called `DecisionTree` should be designed. This class is responsible for the memory management (for example row-wise or column-wise ordering of matrices) and the management of the arguments passed to the middle layer exposed methods. This class is a subclass of `BaseEstimator` and `ClassifierMixin` and can be used as estimator by the `BaggingClassifier` class of the `sklearn` package.

```python
class DecisionTree(BaseEstimator, ClassifierMixin):

    # Constructor
    def __init__(self, column_names_str, attacker_file, budget, ...
        threads, useICML2019):
        # Init all private members
        self.py_dt = py_decision_tree.new()
        # used by fit(X, y)
        self.column_names_str = column_names_str
        self.attacker_file = attacker_file
        self.budget = budget
        self.threads = threads
        self.useICML2019 = useICML2019
        self.max_depth = 4
        self.min_per_node = 20
        self.is_affine = False
        # variables initialized by fit()
        self.categorical_entries = None
        self.column_names = None  # needed by the attacker
        self.is_numeric = None

    # This function also sets self.is_numeric and ...
        self.categorical_entries if they are None
    def transform_to_all_floats(self, X):
        if self.is_numeric is None:
            numerics = ['integer', 'floating']
            self.is_numeric = np.isin(np.apply_along_axis(lambda x: ...
                pd.api.types.infer_dtype(x, skipna=True), 0, X),
                                      numerics)
        if False in self.is_numeric:
            mask_col_not_numerics = (self.is_numeric == False)
            # update self.categorical_entries_str
            if self.categorical_entries is None:
                self.categorical_entries = [i for i in set(X[:, ...
                    mask_col_not_numerics].flatten())]
            dict = {k: self.categorical_entries.index(k) for k in ...
                self.categorical_entries}
            X_all_float = np.array(X, copy=True)
            X_all_float[:, mask_col_not_numerics] = ...
                np.vectorize(dict.get)(X_all_float[:, ...
                mask_col_not_numerics])
            return X_all_float
        else:
            return X

    def fit(self, X, y):

        # set the number of classes (needed for predict_proba in the ...
            BaggingClassifier)
        self.classes_ = np.unique(y)

        X_all_float = self.transform_to_all_floats(X)
        assert (self.is_numeric is not None)
        assert (self.categorical_entries is not None)
        if not X_all_float.data.f_contiguous:
            np.asfortranarray(X_all_float, dtype='float64')
        assert (X_all_float.data.contiguous)
        assert (X_all_float.data.f_contiguous)

        rows, cols = X_all_float.shape
        y = y.astype('float64')
        pointer_y = FFI().cast('double*', y.ctypes.data)
        X_all_float = X_all_float.astype('float64')
        pointer_x = FFI().cast('double*', X_all_float.ctypes.data)
        # Launch fit
        is_numeric_joint = ...
            (','.join(self.is_numeric.astype('str'))).encode('ascii')
        cat_joint = (','.join(self.categorical_entries)).encode('ascii')
        py_decision_tree.fit(self.py_dt,
                                pointer_x, rows, cols,
                                pointer_y,
                                is_numeric_joint,
                                cat_joint,
                                self.column_names_str,
                                self.attacker_file,
                                self.budget,
                                self.threads,
                                self.useICML2019,
                                self.max_depth,
                                self.min_per_node,
                                self.is_affine)

    def predict(self, X):
        if self.is_trained():
            predictions = np.empty(X.shape[0])
            predictions = predictions.astype('float64')
            X_all_float = self.transform_to_all_floats(X)
            rows, cols = X_all_float.shape
            if not X_all_float.data.c_contiguous:
                X_all_float = np.ascontiguousarray(X_all_float)
            assert(X_all_float.data.contiguous)
            assert(X_all_float.data.c_contiguous)
            X_all_float = X_all_float.astype('float64')
            pointer_x = FFI().cast('double*', X_all_float.ctypes.data)
            pointer_predictions = FFI().cast('double*', ...
                predictions.ctypes.data)
            return_scores = True
            is_rows_wise = True
            py_decision_tree.predict(self.py_dt, pointer_x, rows, ...
                cols, pointer_predictions, return_scores, is_rows_wise)
            return predictions
        else:
            raise "decisionTree is not trained, predict() cannot be ...
                called"

    # Needed to use BaggingClassifier (see original python code)
    def predict_proba(self, X, y=None):
        probs_0 = np.empty(X.shape[0])
        probs_1 = np.empty(X.shape[0])

        # Check if the current tree is trained
        if self.is_trained:
            # Get the prediction scores for class 1
            probs_1 = self.predict(X)
            # Get the prediction scores for class 0
            probs_0 = (1 - probs_1)

        return np.column_stack((probs_0, probs_1))

```

```python

    def save(self, filename):
        py_decision_tree.save(self.py_dt, filename.encode('ascii'))

    def load(self, filename):
        py_decision_tree.load(self.py_dt, filename.encode('ascii'))

    def is_trained(self):
        return py_decision_tree.is_trained(self.py_dt)

    def pretty_print(self):
        py_decision_tree.pretty_print(self.py_dt)

    # Destructor
    def __del__(self):
        py_decision_tree.free(self.py_dt)
```

The following example shows how to use the previously defined class to train a single decision tree and to train a bagging classifier. The ensemble model is trained and a set of predictions is calculated.

```python
import os
import sys
import numpy as np
import pandas as pd
from nilib import load_atk_train_valid_test
import ctypes
from cffi import FFI
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.ensemble import BaggingClassifier
import time

import py_decision_tree

if __name__ == '__main__':
    cwd = os.path.dirname(os.path.abspath(__file__))
    attacker_file = (cwd + "/data/attacks.json").encode('ascii')
    budget = 0
    threads = 1
    max_depth = 4
    use_icml_2019 = False
    min_per_node = 20
    is_affine = False
-
    options = {}
    options['training_set'] = cwd + '/data/census/train.csv.bz2'
    options['valid_set'] = cwd + '/data/census/valid.csv.bz2'
    options['test_set'] = cwd + '/data/census/test.csv.bz2'
    options['jobs'] = 1
    options['n_instances'] = 1000
    options['n_estimators'] = 1
    # Building the decision tree
    # Get dataset X like done in the old python code
    train = load_atk_train_valid_test(
        options['training_set'], options['valid_set'], ...
            options['test_set'])[0]
    # Get X and y
    y = train.iloc[:, -1].replace(-1, 0).values[:options['n_instances']]
    X = train.iloc[:, :-1].values[:options['n_instances']]
    column_names_str = ...
        ','.join(train.columns.tolist()[:-1]).encode('ascii')

    decision_tree = DecisionTree(column_names_str, attacker_file, ...
        budget, threads, use_icml_2019)
    decision_tree.fit(X, y)
    assert decision_tree.is_trained() == True
    decision_tree.pretty_print()
    print("Saving the trained decision tree:")
    file_path = "decision_tree_example.txt"
    decision_tree.save(file_path)
    # Calculates prediction scores
    dt_predictions = decision_tree.predict(X)

    # Create a copy of the previously defined decision_tree from file
    decision_tree_copy = DecisionTree(column_names_str, attacker_file, ...
        budget, threads, use_icml_2019)
    decision_tree_copy.load(file_path)
    # Get predictions on the decision tree copy
    dt_predictions_copy = decision_tree_copy.predict(X)

    decision_tree_for_bagging = DecisionTree(column_names_str, ...
        attacker_file, budget, threads, use_icml_2019)
    bagging = BaggingClassifier(base_estimator=decision_tree_for_bagging,
                                n_estimators=options['n_estimators'],
                                max_features=1.0, max_samples=1.0,
                                bootstrap=False, bootstrap_features=False,
                                n_jobs=options['jobs'])
    bagging.fit(X, y)
    bagging_predictions = bagging.predict(X)
```

B.2 C++

The following code reads the input data from a file and from command line, builds a `Dataset` and an `Attacker`, finally trains a `DecisionTree` and calls `predict` on a test set. The code also uses `load` and `save` methods to build a copy of the decision tree and make predictions on the same test set.

```
int main(int argc, char **argv) {

  if (argc < 2) {
    std::cout << "Usage: possible flags are:\n"
              << "-a <name of the attacker json file>, "
              << "-b <budget>, "
              << "-d <max depth>, "
              << "-f <dataset file path>, "
              << "-j <number of threads>\n"
              << "Example:\n./" << argv[0]
              << " -a ../data/attacks.json -b 60 -d 4 -f "
                 "../data/test_training_set_n-1000.txt";
  }
  // Parse the input arguments
  const auto [attackerFile, datasetFile, maxDepth, budget,
              threads] = utils::parseArguments(argc, argv);

  // Allocate dataset matrix X and label vector y
  const auto [rows, columnNames] = ...
      Dataset::getDatasetInfoFromFile(datasetFile);
  const unsigned cols = columnNames.size();
  // The principal program must manage the memory of X and Y
  feature_t *X = (feature_t *)malloc(sizeof(feature_t) * rows * cols);
  label_t *y = (label_t *)malloc(sizeof(label_t) * rows);
  const auto [isNumerical, notNumericalEntries] =
  Dataset::fillXandYfromFile(X, rows, cols, y, datasetFile);
  Dataset dataset(X, rows, cols, y, utils::join(isNumerical, ','),
                  utils::join(notNumericalEntries, ','),
                  utils::join(columnNames, ','));
  //
  DecisionTree dt;
  const bool useICML2019 = false;
  const unsigned minPerNode = 20;
  const bool isAffine = false;
  dt.fit(dataset, attackerFile, budget, threads, useICML2019, maxDepth,
         minPerNode, isAffine, Impurity::SSE);

  std::cout << "Is the decision tree trained? " << dt.isTrained() << ...
      std::endl;
  std::cout << "Decision tree height: " << dt.getHeight() << std::endl;
  std::cout << "Decision tree node count: " << dt.getNumberNodes() << ...
```

```
        std::endl;

  dt.save("example.txt");
  DecisionTree dt_copy;
  dt_copy.load("example.txt");
  // Get X as C-order
  feature_t *X_test = (feature_t *)malloc(sizeof(feature_t) * rows * ...
      cols);
  std::size_t index = 0;
  for (std::size_t i = 0; i < rows; i++) {
    for (std::size_t j = 0; j < cols; j++) {
      X_test[index] = dataset(i, j);
      index++;
    }
  }
  double *predictions = (double *)malloc(sizeof(double) * rows);
  dt.predict(X_test, rows, cols, predictions, true, false);
  std::cout << "Predictions (rows-wise X):" << std::endl;
  std::cout << std::setprecision(1) << predictions[0];
  for (index_t i = 1; i < rows; i++) {
    std::cout << "," << std::setprecision(1) << predictions[i];
  }
  std::cout << std::endl;
  // Predict with the decision tree loaded from file
  double *predictions_copy = (double *)malloc(sizeof(double) * rows);
  dt_copy.predict(X_test, rows, cols, predictions_copy, true, false);
  std::cout << "Predictions of dataset loaded from file:" << std::endl;
  std::cout << std::setprecision(1) << predictions_copy[0];
  for (index_t i = 1; i < rows; i++) {
    std::cout << "," << std::setprecision(1) << predictions_copy[i];
  }
  std::cout << std::endl;

  // Free memory
  free((void *)X);
  free((void *)y);
  free((void *)X_test);
  free((void *)predictions);
  free((void *)predictions_copy);

  return 0;
```

The following code reads the data from the command line or from file and then trains a `BaggingClassifier` object.

```
int main(int argc, char **argv) {

  assert(argc ≥ 2);
  // Parse the input arguments
  const auto [attackerFile, datasetFile, maxDepth, budget,
              threads] = utils::parseArguments(argc, argv);
  // Allocate dataset matrix X and label vector y
  const auto [rows, columnNames] = ...
      Dataset::getDatasetInfoFromFile(datasetFile);
  const unsigned cols = columnNames.size();
  // The principal program must manage the memory of X and Y
  double *X = (double *)malloc(sizeof(double) * rows * cols);
  double *y = (double *)malloc(sizeof(double) * rows);
  const auto [isNumerical, notNumericalEntries] =
      Dataset::fillXandYfromFile(X, rows, cols, y, datasetFile);
  //
  Dataset dataset(X, rows, cols, y, utils::join(isNumerical, ','),
                  utils::join(notNumericalEntries, ','),
                  utils::join(columnNames, ','));

  assert (budget ≥ 0);
  Attacker attacker(dataset, attackerFile, budget);
  std::cout << "The dataset size is: " << dataset.size() << std::endl;
  //
  const bool useICML2019 = false;
  const unsigned minPerNode = 20;
  const bool isAffine = false;
  BaggingClassifier baggingClassifier;
  std::cout << "Fitting the BaggingClassifier\n";
  baggingClassifier.setMaxFeatures(0.5);
  baggingClassifier.setEstimators(8);
  std::cout << "Jobs for training the forest = " << threads << std::endl;
  baggingClassifier.setJobs(threads);
  baggingClassifier.setWithReplacement(false);
  baggingClassifier.fit(dataset, attacker, useICML2019, maxDepth, ...
      minPerNode, isAffine);
  // Free memory
  free((void *)X);
  free((void *)y);

  return 0;
```

Bibliography

[1] S. Calzavara, C. Lucchese, G. Tolomei, S. Assefa, and S. Orlando, "Treant: Training evasion-aware decision trees," *Data Mining and Knowledge Discovery*, 06 2020.

[2] X. Wu, V. Kumar, R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. Mclachlan, S. K. A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, 12 2007.

[3] A. Maas, Q. Le, T. neil, O. Vinyals, P. Nguyen, and A. Ng, "Recurrent neural networks for noise reduction in robust asr," *13th Annual Conference of the International Speech Communication Association 2012, INTERSPEECH 2012*, vol. 1, 01 2012.

[4] B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., *Advances in Kernel Methods: Support Vector Learning.* Cambridge, MA, USA: MIT Press, 1999.

[5] C. Neocleous, C.and Schizas, "Artificial neural network learning: A comparative review," *Neural Networks*, pp. 300–313, 2002.

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014. [Online]. Available: http://arxiv.org/abs/1404.7828

[7] M. N. H. Siddique and M. O. Tokhi, "Training neural networks: backpropagation vs. genetic algorithms," in *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, vol. 4, 2001, pp. 2673–2678.

[8] M. A. Kon and L. Plaskota, "Information complexity of neural networks," *Neural Networks*, vol. 13, no. 3, pp. 365–375, 2000.

[9] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies.* NLD: IOS Press, 2007, p. 3–24.

[10] A. R. Barron, “Universal approximation bounds for superpositions of a sigmoidal function,” *IEEE Transactions on Information Theory*, vol. 39, no. 3, pp. 930–945, 1993.

[11] S. Haykin, *Neural Networks: A Comprehensive Foundation*, ser. International edition. Prentice Hall, 1999. [Online]. Available: https://books.google.it/books?id=bX4pAQAAMAAJ

[12] V. N. Vapnik, *The Nature of Statistical Learning Theory.* Berlin, Heidelberg: Springer-Verlag, 1995.

[13] J. C. Platt, “Using analytic qp and sparseness to speed training of support vector machines,” in *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II.* Cambridge, MA, USA: MIT Press, 1999, p. 557–563.

[14] D. Wettschereck, D. W. Aha, and T. Mohri, *A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms.* USA: Kluwer Academic Publishers, 1997, p. 273–314.

[15] D. Lee, J. Park, J. Shim, and S.-g. Lee, “An efficient similarity join algorithm with cosine similarity predicate,” in *Database and Expert Systems Applications*, P. G. Bringas, A. Hameurlain, and G. Quirchmayr, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 422–436.

[16] Y. Park, S. Park, S.-g. Lee, and W. Jung, “Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction,” in *Database Systems for Advanced Applications*, S. S. Bhowmick, C. E. Dyreson, C. S. Jensen, M. L. Lee, A. Muliantara, and B. Thalheim, Eds. Cham: Springer International Publishing, 2014, pp. 327–341.

[17] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih, “A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms,” *Machine Learning*, vol. 40, pp. 203–228, 2000.

[18] S. Ruggieri, “Efficient c4.5 [classification algorithm],” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, pp. 438 – 444, 04 2002.

[19] T. Y. Olcay and D. Onur, “Parallel univariate decision trees,” *Pattern Recognition Letters*, vol. 28, no. 7, pp. 825 – 832, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865506002923

[20] K. Jansson, H. Sundell, and H. Boström, “gpuRF and gpuERT: Efficient and Scalable GPU Algorithms for Decision Tree Ensembles,” *IEEE International*

Parallel and Distributed Processing Symposium Workshops, pp. 1612–1621, 2014.

[21] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "PLANET: Massively parallel learning of tree ensembles with MapReduce," *Proceeding of VLDB Endowment*, vol. 2, pp. 1426–1437, 8 2009.

[22] D. Miyamoto, H. Hazeyama, and Y. Kadobayashi, "An evaluation of machine learning-based methods for detection of phishing sites," *Australian Journal of Intelligent Information Processing Systems*, vol. 10, pp. 539–546, 11 2008.

[23] R. N. Rojas-Bello, L. F. Lago-Fernández, G. Martínez-Muñoz, and M. A. Sánchez-Montañés, "A comparison of techniques for robust gender recognition," in *2011 18th IEEE International Conference on Image Processing*, 2011, pp. 561–564.

[24] S. Aruna, D. Rajagopalan, and L. Nandakishore, "An empirical comparison of supervised learning algorithms in disease detection," *International Journal of Information Technology Convergence and Services*, vol. 1, 08 2011.

[25] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, ser. AISec '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 43–58. [Online]. Available: https://doi.org/10.1145/2046684.2046692

[26] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognition*, vol. 84, pp. 317–331, 2018.

[27] J. Angwin, J. Larson, S. Mattu, and L. Kirchner, "Machine bias," *ProPublica*, 2016.

[28] A. Lambrecht and C. E. Tucker, "Algorithmic bias? an empirical study into apparent gender-based discrimination in the display of stem career ads," Available at SSRN: https://ssrn.com/abstract=2852260 or http://dx.doi.org/10.2139/ssrn.2852260, 2018.

[29] "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)," p. 1–88, May 2016, official J. European Union.

[30] A. Kerckhoffs, "La cryptographie militaire," *Journal des Sciences Militaires*, vol. 9, p. 5–83, 1883.

[31] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 427–436, 2015.

[32] D. Hendrycks, K. Zhao, S. Basart, J. Steinhardt, and D. Song, "Natural adversarial examples," *CoRR*, vol. abs/1907.07174, 2019. [Online]. Available: http://arxiv.org/abs/1907.07174

[33] N. Papernot, P. McDaniel, and I. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *arXiv preprint arXiv:1605.07277*, 2016.

[34] F. Zhang, Y. Wang, S. Liu, and H. Wang, "Decision-based evasion attacks on tree ensemble classifiers," *World Wide Web*, 2020.

[35] A. Kantchelian, J. Tygar, and A. Joseph, "Evasion and hardening of tree ensemble classifiers," *arXiv:1509.07892v2*, 05 2017.

[36] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh, "Robust decision trees against adversarial examples," *ICML 2019*, pp. 1122–1131, 02 2019.

[37] M. Cheng, T. Lê, P.-Y. Chen, J. Yi, H. Zhang, and C.-J. Hsieh, "Query-efficient hard-label black-box attack: An optimization-based approach," *ArXiv*, vol. abs/1807.04457, 2019.

[38] G. Tolomei, F. Silvestri, A. Haines, and M. Lalmas, "Interpretable predictions of tree-based ensembles via actionable feature tweaking," *SIGKDD*, p. 465–474, 2017.

[39] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *ICLR*, 2018.

[40] E. B. Hunt, P. J. Stone, and J. Marin, *Experiments in induction.* Academic Press New York, 1966.

[41] D. Kraft, *A Software Package for Sequential Quadratic Programming*, ser. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988.

[42] S. G. Johnson, "The NLopt nonlinear-optimization package," http://github.com/stevengj/nlopt.

[43] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[44] R. Bast and R. Di Remigio, *CMake Cookbook.* Packt Publishing Ltd, 2018.