Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's thesis

Distributed Algorithms for Decision Forest Training in the Network Traffic Classification Task

Radek Starosta

Supervisor: Ing. Jan Brabec

May 25, 2018

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Starosta Radek** Personal ID number: **423311**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Distributed Algorithms for Decision Forest Training in the Network Traffic Classification Task

Bachelor's thesis title in Czech:

Distribuované algoritmy pro trénink rozhodovacích lesů v úloze klasifikace síťového provozu

Guidelines:

Decision forests are a popular algorithm for the classification task in machine learning. In the presence of huge datasets, it is often necessary to use distributed algorithms for decision forest induction. In this thesis, we are interested in algorithms available on the Apache Spark platform. Currently we have experience with the implementation of the PLANET algorithm available in Apache Spark MLlib v2.2.1. There are several issues that
have been identified with the current version of the algorithm. This thesis aims to achieve several main goals:
1. Describe the state-of-the-art of distributed decision forest training.
2. Explore the various suggested improvements over the current version of the algorithm that is available in Spark 2.2.1. Implement them inside of a forked Apache Spark MLlib and benchmark them.
3. Suggest, implement and evaluate custom algorithm improvements to achieve better performance of decision forests in the network classification for malware task.

Bibliography / sources:

[1] Criminisi, A., Shotton, J., & Konukoglu, E. (2011). Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning. Microsoft Research Cambridge, Tech. Rep. MSRTR-2011-114, 5(6), 12.
[2] Abuzaid, F., Bradley, J. K., Liang, F. T., Feng, A., Yang, L., Zaharia, M., & Talwalkar, A. S. (2016). Yggdrasil: An Optimized System for Training Deep Decision Trees at Scale. In Advances in Neural Information Processing Systems (pp. 3817-3825).
[3] Panda, B., Herbach, J. S., Basu, S., & Bayardo, R. J. (2009). Planet: massively parallel learning of tree ensembles with mapreduce. Proceedings of the VLDB Endowment, 2(2), 1426-1437.

Name and workplace of bachelor's thesis supervisor:

Ing. Jan Brabec, Cisco Systems, Inc., Prague

Name and workplace of second bachelor's thesis supervisor or consultant:

Ing. Jan Drchal, Ph.D., Artificial Intelligence Center, FEE

Date of bachelor's thesis assignment: **22.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Jan Brabec	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Acknowledgements

I would like to thank my supervisor Jan Brabec for his patience and valuable advice during the time of writing this thesis. I consider myself very lucky to have a supervisor who cared about this project and was always available to answer my questions and provide insights.

I also want to thank the rest of the CTA team at Cisco Systems, Inc. for providing me with the opportunity and resources to research this interesting project.

Finally, I would like to thank my family, friends, and colleagues for their support and tolerance during the months spent working on this thesis.

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague on May 25, 2018 ……

Czech Technical University in Prague
Faculty of Electrical Engineering

Citation of this thesis

Starosta, Radek. *Distributed Algorithms for Decision Forest Training in the Network Traffic Classification Task*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2018.

Abstrakt

V této práci se zaměřujeme na zlepšení výkonu distribuovaného trénování náhodných lesů v knihovně Spark MLlib. Trénovací proces optimalizujeme přidáním fáze lokálního trénování, ve které dotrénujeme podstromy pro dostatečně malé uzly lokálně v paměti jednotlivých strojů. Tyto uzly nejprve seskupíme do větších a vyváženějších lokálně trénovaných úloh pomocí bin-packingu, a následně tyto úlohy efektivně rozplánujeme s pomocí prediktoru, který přesněji odhaduje jejich dobu trvání. Lokální trénování nám také umožňuje trénovat hluboké rozhodovací stromy a eliminovat část paměťových problémů v současné implementaci. Naši implementaci testujeme na velkých datech ze síťového provozu, která se používají k detekci malwaru. Na této trénovací sadě je náš algoritmus více než 105× rychlejší než původní implementace. Toto zlepšení nám umožňuje trénovat náhodné lesy na větších trénovacích sadách, což může výrazně zlepšit výkon klasifikátorů. Klasifikátor pro detekci malwaru, který byl natrénovaný algoritmem popsaným v této práci, se již aktivně používá v systému Cisco Cognitive Threat Analytics, a naše implementaci jej umožnila natrénovat na více než desetinásobném množství dat.

Klíčová slova náhodné lesy, Apache Spark, MLlib, distribuované trénování

Abstract

In this thesis, we focus on improving the performance of distributed random forest training in Spark MLlib. To optimize the training process, we introduce a local training phase in which we complete the tree induction of sufficiently small nodes in-memory. Further, we group these nodes into larger and more balanced local training tasks using bin packing and effectively schedule the tasks using an offline-trained predictor to predict task duration more accurately. Our algorithm allows training of deeper decision trees and mitigates runtime memory issues. We benchmark our implementation on a huge, real network traffic dataset used for malware detection, for which it is up to 105× faster than the original MLlib implementation. This performance improvement allows us to train random forests on larger datasets, which can significantly improve classification predictive performance. A classifier for malware detection trained using the algorithm presented in this thesis is actively used in the Cisco Cognitive Threat Analytics system. Thanks to our implementation, we were able to train it using 10× more data than before.

Keywords random forest, Apache Spark, MLlib, distributed training

Contents

List of Figures

List of Tables

Introduction

In this day and age, when most people use computers in their daily lives, cybersecurity has become a major concern for both individuals and businesses. Malware attacks have caused serious damage over the the last decade and effective methods to detect malware have become highly demanded. In recent years, machine learning approaches are often used for malware detection.

Decision tree-based methods are widely used for classification and regression tasks, because they are easily interpretable, handle structured data well, are robust to outliers [1] and naturally allow multiclass classification. Random forests [2] are a commonly used tree ensemble method, which handles big data well and often achieves great prediction performance and generalization.

We use a random forest classifier to detect malicious network traffic. Because our dataset is huge, we need to train the random forest in a distributed fashion using a computer cluster. For that we use MLlib [3], a machine learning library built on Apache Spark [4]. However, the random forest implementation in MLlib is highly inefficient for training deep decision tree models, which are required to achieve good prediction performance on our data, and unstable due to memory management issues.

In this thesis, we mainly focus on improving the computational performance of distributed random forest training in MLlib, which will allow us to train more powerful models on larger datasets. We introduce a local training stage, in which we train suitable decision tree node subtrees in-memory by aggregating all their input data on a single worker. The subtrees are trained in parallel on multiple workers, but because the local training tasks are imbalanced, the workers can spend a significant portion of time waiting for the other tasks to finish when they are not optimally scheduled. We present our custom task scheduling method, which uses binpacking to balance the task sizes and a linear regression model to predict the task duration. Finally, we evaluate the performance of our implementation using a dataset of real network traffic.

In Chapter 1, we describe the PLANET [5] algorithm and give an overview of the available distributed random forest libraries. In Chapter 2, we show

how MLlib implements the PLANET algorithm on Spark and explain why the current implementation performs poorly. In Chapters 3 and 4, we give thorough description of our local training implementation and the task scheduling algorithm. Finally, in Chapter 5 we benchmark our implementation on a network traffic dataset.

Chapter 1

Distributed Random Forest Training

In this chapter, we first introduce a basic decision tree induction algorithm and show why it is not suitable for large-scale training.

We introduce the MapReduce model, thoroughly describe the PLANET [5] algorithm, which is the standard approach for distributed decision tree learning, and compare it to Yggdrasil [3], which attempts to improve on it using vertical partitioning.

Finally, we present an overview of several available distributed decision forest implementations and compare their properties.

1.1 Greedy Tree Induction

Because the task of finding the optimal decision tree for a given training dataset is NP-complete [6], a greedy top-down induction algorithm is usually used to construct the tree (Algorithm 1).

Let $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times Y$ be a set of training data, where $\mathbf{x}_i = (x_1, \ldots, x_d)$ are numeric feature vectors and y_i are classification labels from a set of classes $Y = \{c_1, \ldots, c_k\}$.

The tree is built recursively starting from the top node. The data corresponding to the current node are split into two subsets D_L and D_R according to a predicate $x_i < \theta$, which minimizes *impurity* – the diversity of labels of the resulting subsets. A leaf node is constructed if a stopping condition is reached (i.e., reaching a certain tree depth, having a small number of samples in D), or when the data in D are *pure* – all are labeled as a single class. The prediction of such leaf node is the class with maximum frequency among its data points.

Algorithm 1 Greedy tree induction algorithm

Require: $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
1: **function** BuildTree(D)
2: **if** StoppingCondition(D) or D is pure **then**
3: **return** leaf node
4: **else**
5: (S, D_L, D_R) = FindBestSplit(D)
6: T_L = BuildTree(D_L)
7: T_R = BuildTree(D_R)
8: T = new node storing split S and pointers to subtrees T_L and T_R
9: **return** T
10: **end if**
11: **end function**

1.1.1 Selecting Best Split

The most important stage of Algorithm 1 is *FindBestSplit*, which needs to determine the splitting dimension i and threshold θ of the optimal split by evaluating all possible splits. To measure the quality of splits in terms of impurity decrease, we use *information gain* (Equation (1.1)) which is defined as the change in *information entropy* H (Equation (1.2)) from the current state to a state after the split. The optimal split maximizes information gain.

$$IG(D) = H(D) - \frac{|D_L| \cdot H(D_L) + |D_R| \cdot H(D_R)}{|D|} \tag{1.1}$$

$$H(X) = -\sum_{c \in Y} p(c) \log p(c) \tag{1.2}$$

where $p(c)$ are the class frequencies in the given data subset X

Entropy is zero if all data points belong to the same class (node is pure), and high when all classes are evenly represented in the data subset. Other measures, such as Gini impurity or variance (usually in regression trees), can be used as well.

To find the best split efficiently, D is sorted along each dimension, and a split is considered between each adjacent pair of feature values x_i. If we first precompute class counts for each unique feature value, only one pass over the sorted feature values is then required to find the threshold of the best split on a given feature.

1.1.2 Handling Large Data

This simple algorithm, which is implemented in many machine learning libraries (e.g., scikit-learn [7], H_2O [8]), is designed to work in a single machine

setting and therefore is suitable for smaller datasets, where all data points fit into memory. To find the best split for a node, we need to iterate over all of its input data, which can be large, especially at higher levels of the tree. For larger amounts of data, this becomes inefficient, as the data has to be gradually loaded into memory from secondary storage and sorting the data along each dimension to find the exact optimal threshold can also become a bottleneck.

In scenarios with big data saved across multiple machines, this approach can be viable, as the feature vectors resulting from feature extraction are usually much more compact than the data itself. Cluster computing can be used first to process the data and persist the labeled data points, and the classifier training will then be done on a single machine with a large amount of memory. However, this method will not be able to deal with an arbitrarily sized dataset.

Many methods have been proposed to enable decision tree learning on huge datasets. Some of them still utilize centralized training but manipulate the data to speed up training from disk [9]. Other more recent attempts make use of GPUs to accelerate the training process [10]. We will focus on methods that parallelize the tree induction process to multiple machines, which is the only truly scalable solution.

1.2 PLANET

PLANET [5] is a distributed framework for training tree models over large datasets developed at Google. Instead of constructing trees on subsets of data, it breaks up the tree induction process into a series of distributed MapReduce [11] tasks, enabling learning in a distributed setting using commodity hardware. The authors also propose several optimizations to mitigate inefficiencies raised from utilizing the MapReduce model.

PLANET does not have a reference implementation, but most popular algorithms for approximate decision tree learning, such as MLlib [3] RandomForest or XGBoost [12], reuse the idea of proposing a set of candidate splits, distributing subsets of data to workers and aggregating sufficient statistics to find the best split among candidates.

1.2.1 MapReduce

MapReduce [11] is a programming model for processing big data in parallel using a computer cluster. The model is based on the idea of computing (key, value) pairs from each piece of input (*map*), grouping of all intermediate values by key (*shuffle*) and finally processing the grouped values to return the output (*reduce*), demonstrated in Figure 1.1.

MapReduce provides a high-level abstraction to programming distributed processing jobs and only requires users to specify the *map* and *reduce* functions.

The input data is distributed across multiple workers, and each worker applies the *map* function to its local subset. The workers then redistribute the data, so that all data belonging to one key are located on the same worker, and finally process these sets using the *reduce* function.

MapReduce is also fault-tolerant, as the data is stored on a file-system between all stages of the computation. The most common implementation of the MapReduce model is Hadoop MapReduce [13], which uses Hadoop Distributed File System for storage. Although recently the focus has shifted towards more flexible and less disk-oriented systems (e.g., Apache Spark [4]), MapReduce remains widely used.

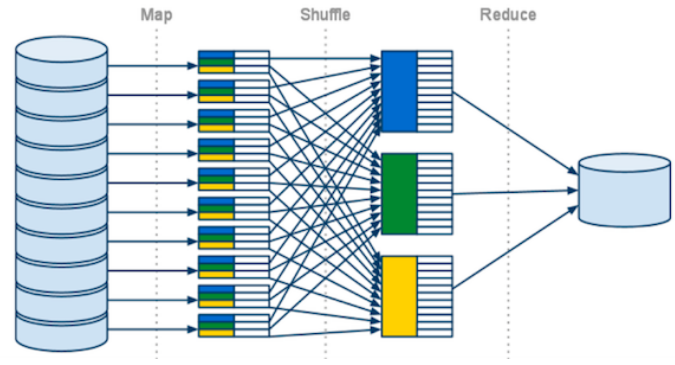


Figure 1.1: MapReduce model overview [14]

1.2.2 Algorithm Overview

The main component of PLANET is the *Controller*, which has access to the computation cluster, on which it schedules MapReduce jobs. It maintains a *ModelFile*, which stores the currently constructed tree model, and two queues:

- *MapReduceQueue*: Contains nodes too large to split in memory on a single machine.
- *InMemoryQueue*: Contains nodes that can be split on a single machine.

The *Controller* dequeues nodes off the two queues and schedules MapReduce jobs to find their best splits.

- *MR_InMemory*: Nodes from *InMemoryQueue* are processed using a local training algorithm similar to Algorithm 1, which completes the rest of the tree induction.
- *MR_ExpandNodes*: For nodes from *MapReduceQueue*, the mappers collect statistics required to compute the desired impurity measure for each of the candidate split predicates. In the reduce phase, these statistics

are grouped to form a set of candidate splits with complete statistics from the whole input dataset.

When a MapReduce job completes, *Controller* processes the results and updates *ModelFile* with the determined split predicates. In-memory jobs output the complete subtree, so new nodes are not added to the queues. Statistics resulting from the jobs with larger nodes are collected, and the best split is selected based on calculated impurity decrease. For every processed node, two new nodes are added to their appropriate queue based on the size of the dataset subset after splitting it using the determined predicate.

1.2.3 MapReduce Jobs

Now we will describe the two key MapReduce jobs in detail and explain an important tradeoff during candidate split selection, which makes the computation feasible.

Both MapReduce jobs take the same input parameters: a set of nodes N, training dataset D and current *ModelFile* M. Note that every MapReduce job scans the entire input dataset D, even though the subset of data belonging to nodes in N might be much smaller. However because the tree is built breadth-first, we can expand all nodes on one tree level in a single MapReduce job, meaning most data points in D will belong to some node that is being expanded, unless they already belong to a leaf node in a higher level of the tree. As we reach lower levels of the tree, scanning the entire dataset becomes more inefficient, as only a small portion of it will belong to non-leaf nodes.

1.2.3.1 MR_InMemory

The in-memory MapReduce job is straightforward, the mappers iterate over every data point $(\mathbf{x}, y)$ in D and traverse the tree in M to determine whether the data point belongs to some node in N. The mapping phase outputs $(node, (\mathbf{x}, y))$ pairs if such a node exists. These pairs are grouped by key in the reduce phase into sets of data points for each node in N, which are then used to complete the subtree induction using Algorithm 1.

1.2.3.2 MR_ExpandNodes

Recall that the *FindBestSplit* method from Algorithm 1 evaluates every single split threshold by sorting the data along each feature and then considering all adjacent feature values. This would be impractical in a distributed setting with large amounts of data – not only would the sorting take a substantial amount of time, but we would also need to save the results to secondary storage and deal with complicated partitioning across multiple mappers.

Instead, authors propose we precompute a set of thresholds for every feature, which will then be used as the potential split candidates throughout the

whole training process. Although we potentially sacrifice some model accuracy by not always splitting the node with the most optimal feature threshold, it allows us to effectively parallelize the process, as the mappers can immediately start computing class counts for the known split candidates.

To precompute the thresholds, we first sample a subset of the input dataset and compute an equi-depth histogram for a given number of bins [15]. In equi-depth histograms, the bin thresholds are not uniformly distributed, and we aim to have an equal number of samples in each bin instead. A single split candidate is then considered for every resulting bin.

Similarly to *MR_InMemory*, first we need to iterate over all data points in D and determine which node the points belong to by traversing the tree in M and filtering out unnecessary points. We need to collect statistics to compute the chosen impurity measure for every candidate split, in our case class frequencies to compute entropy and information gain (Equations (1.1), (1.2)). For every node $n \in N$, we will save a tuple agg_tup_n of total class counts, and then for every split candidate (i, θ) a tuple $T_{n,i,\theta}$ with class counts of the data points for which $x_i < \theta$. Each mapper computes these statistics over its data partition and outputs the class count tuples with (n, i, θ) as the key. At this point, the tuples need to be shuffled so that all values with the same key are grouped on a single worker – this requires a lot of communication between the cluster nodes and is the main bottleneck of this algorithm. In the reduce phase, tuples with the same key are aggregated into a single tuple with summed class counts for each candidate split. We can easily compute class counts for the right subset of the split data by subtracting it from agg_tup_n, which gives us all required information to compute information gain and determine the best split among the candidates. Reducers then output their proposed best splits, which are then finally collected by the *Controller* and the best one is selected and added to the *ModelFile* M.

1.2.4 Learning Random Forests

The described algorithm builds a single decision tree, but it can be easily extended to learn tree ensembles such as random forests, which tend to have more accurate predictions.

- To build multiple trees, *Controller* pushes multiple root nodes onto *MapReduceQueue* – either at the start of the learning process or after completing the previous tree.

- To bring randomness into the individual trees, we need to support learning on samples of the input dataset D. This feature is achieved using hash-based sampling – a combination of the training record's id and the tree id is hashed, and only the records for which this hash belongs into a specified range are used to train the tree.

- The *Controller* can also generate a random subset of feature indices for every node and pass it to the MapReduce jobs so that only some of the features are considered when finding the best split. This addition again brings more randomness to the tree ensemble.

1.3 Yggdrasil

Yggdrasil [16] is a distributed tree learning system, which aims to improve on PLANET regarding both learning time and accuracy. Instead of partitioning the data horizontally, authors propose vertical (by feature) partitioning, which results in less communication and the ability to determine exact split thresholds.

To demonstrate the advantages of this approach, recall that the output of the mapping phase of *MR_ExpandNodes* are tuples of class counts for every split threshold – the mapper does not have complete data available. To determine the best split for a node, all of the statistics for that node need to be aggregated to a single worker. Overall, this computation is very communication-heavy, especially for deep trees where the number of nodes can grow exponentially with tree level. Although the input subsets become smaller at lower levels of the tree, the size and amount of the transferred tuples stay the same for all nodes, regardless of input subset size. We further analyze this in Section 3.1.1.

If we instead partition the data by feature, each worker only stores a couple of features, but for the complete set of training points. This means we can immediately compute the best split on the set of features distributed to the worker, therefore communicating only the statistics of the best split is sufficient. However because every worker now stores a subset of every training point, we also need to deliver a bit vector indicating the split direction for each point to all workers. This requirement leads to a tradeoff between horizontal and vertical partitioning regarding communication cost, which depends on the number of training instances, number of features and desired tree depth. From the experiment results presented in [3], it is clear that for high dimensional data and deeper trees, Yggdrasil drastically reduces communication and is the better option.

Additionally, we no longer need to approximate the split thresholds beforehand and can use the method described in Section 1.1 to determine the optimal threshold precisely, which can result in better prediction accuracy of the model. To perform this precise split finding, we need to sort the feature values, but this is no longer a problem because all values for every feature are now available in the memory of a single worker. Additionally, this presents the opportunity to compress the data using run-length encoding – if multiple data points have the same feature value, it is sufficient to store the feature value and count. Because only the information about the class frequencies is

required to compute impurity, we can determine the optimal splits using the saved count and never need to decompress the data.

The main drawback of this method is that because most data are saved in a row-based format, it is more natural to distribute them using horizontal partitioning. Unless we already use column-based storage (e.g., Apache Parquet [17]) to store the training data, the algorithm does require transforming it into a column-based format, which requires additional memory and resources.

The available implementation [18] is built on top of an older version of Apache Spark and is not maintained. In its current state, it only supports learning single decision trees. While there were some efforts to adopt this method in Spark MLlib [19], not much progress has been made in that regard so far.

1.4 Available Implementations

1.4.1 Apache Mahout

Apache Mahout [20] is a Java library containing implementations of several distributed machine learning algorithms, primarily using the MapReduce model on Apache Hadoop framework. Its decision forest implementation partitions the input dataset across a set of mappers and then constructs individual trees on these smaller subsets of data, with each mapper using only the data in its designated partition.

While this approach is simple to implement and relatively fast, because we mostly utilize a straightforward in-memory training algorithm, the resulting forests often have lower accuracy compared to other methods. On datasets with very rare classes, this method can also perform poorly, because the training samples of these classes may not be present in many of the input subsets for individual trees.

At the time of writing, the decision forest libraries are deprecated, as Mahout shifts towards building a backend-independent environment and support for MapReduce algorithms is gradually phased out [21].

1.4.2 H_2O

H_2O [8] is a distributed machine learning platform written in Java. It uses the MapReduce paradigm to distribute work, with a custom implementation capable of processing the data mainly in-memory, similarly to Apache Spark. It directly supports Java, Python and R languages, but the server also exposes a REST interface.

Its Distributed Random Forest (DRF) library implements an algorithm similar to PLANET [22], although inner details are not evident from the documentation. According to a benchmark on subsets of the airline dataset [23], both the in-memory and distributed versions of H_2O's random forest libraries

perform well and can handle large amounts of data smoothly, although no direct comparison is made between DRF and MLlib implementations in the distributed setting.

1.4.3 MLlib

MLlib [3] is a machine learning library developed as a part of the Apache Spark project. The current RandomForest implementation is heavily based on PLANET but utilizes the Spark model, which is optimized towards in-memory processing as opposed to the mainly disk-based MapReduce. It also includes several optimizations that take advantage of this more complex model but omits the local training stage, which significantly affects its performance.

The MLlib implementation is optimized for training shallow trees, which do not yield optimal results for data with a large number of features. At the moment it only supports training trees up to a maximum depth of 30, for which it already performs poorly. From our experience, in addition to being very slow when dealing with large datasets, it has severe memory management issues which lead to executors or entire jobs failing. Tuning job settings and cluster parameters can mitigate these issues, but they remain a problem that complicates training forests on larger data.

In this thesis, we will focus on extending this implementation and overcoming the stated problems. We choose this implementation because it is well documented, both Spark and MLlib are widely used and actively developed, and because the codebase used to run the experiments is already closely tied to it.

1.4.4 Sequoia Forest

Sequoia Forest [24] is a Random Forest implementation on Apache Spark. It was developed during early stages of MLlib development and aimed to improve the speed and ability to handle large data of its Random Forest libraries.

Sequoia Forest implementation added the option of training on random subsets of features, and a node id cache. The cache is used to keep track of which node currently has the given data point in its input subset, which eliminates to need to transfer and traverse the current tree model. Both of these features are implemented in the current version of MLlib, and the Sequoia Forest implementation is no longer maintained.

The most significant improvement that still has not been implemented in MLlib is the addition of local subtree training – once the input dataset of a node becomes small enough to fit into memory, all of its data is shuffled to one executor and rest of the subtree induction is finished there. This idea was already present in the original PLANET algorithm (Section 1.2.2), and authors even include a performance comparison of their implementation with and without local training in the paper, showing a significant difference in

training time. In MLlib, the entire forest is trained using the distributed algorithm similar to the one described in Section 1.2.3.2, which greatly affects its performance, mainly when training deep trees.

Once the training reaches lower levels of trees, we encounter a large number of nodes with small input subsets. Every executor needs to accumulate a tuple of sufficient statistics for every split candidate for every node, which has data partitioned in its designated partitions, and these tuples then have to be aggregated to compute the impurity measure. Therefore most resources are used on communication between executors, which could easily be prevented if the input subsets are already small enough to be trained locally.

1.4.5 Woody

Woody is an implementation of a recently published algorithm [25], which presents an interesting compromise between the subset method used in Apache Mahout (Section 1.4.1) and fully distributed algorithms that aggregate data from workers (e.g., PLANET). It enables training of random forest models on large datasets using a single machine due to its multi-level construction scheme.

It separates the training into two stages – it first trains a top tree using a small random subset of the data. This tree is then used to distribute all input data into smaller partitions based on the corresponding leaf nodes of the tree. Separate standard random forests are then trained on the resulting partitions. To get a prediction, the top tree is traversed first to retrieve a pointer to the random forest for the designated partition of the given data point.

To optimize the performance of this algorithm, authors propose several modifications to the basic decision tree induction, such as modifying the impurity measure criterion to create well balanced top trees consistently, and continuing splitting pure data if the subsets are not small enough.

Because this algorithm is brand new, it has not been extensively tested. It showed promising results in the experiments presented in the paper, where the resulting models do not show any significant loss in accuracy compared to standard random forest implementations in scikit-learn and H_2O, while still being very fast to train and able to handle large data. More benchmarks need to be done on large data to show how the size proportion of the sampled dataset for top trees impact accuracy, but this algorithm could potentially be ported to distributed implementations and significantly speed up their training.

Chapter 2

Random Forest Training in MLlib

In this chapter, we first describe the fundamental concepts and terminology of the Apache Spark [4] framework.

We then provide an overview of how MLlib uses it to implement the PLANET algorithm, summarize the stages of the current implementation and go over several important implementation details.

2.1 Apache Spark

Apache Spark [4] is an open-source cluster computing system written mainly in Scala. It uses a custom processing engine that can handle general execution graphs, as opposed to the two-stage MapReduce engine used in the original PLANET implementation. The engine is optimized for in-memory processing and brings a significant performance boost over the disk-based MapReduce in specific applications, such as iterative algorithms, because the reused data is kept in memory and does not need to be saved and loaded from disk between every task. However, Spark outperforms MapReduce even in purely disk-based tasks [26].

2.1.1 Spark Components

To achieve parallelism, Spark operates on a cluster of worker nodes (Figure 2.1). In this section, we introduce the key concepts of the Spark computation model and its components.

Application The highest-level unit of computation is a Spark application, which is a user program written on Spark.

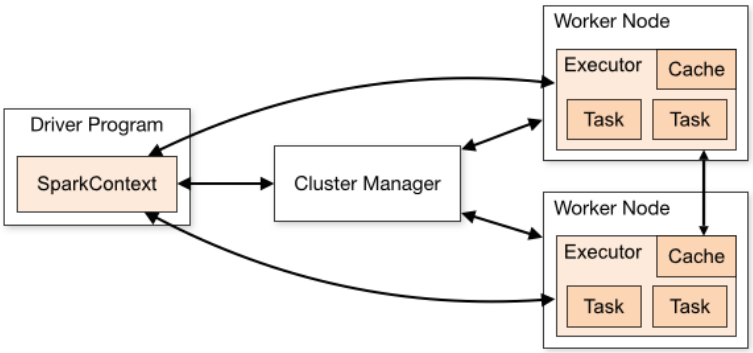


Figure 2.1: Spark Cluster Overview [27]

Driver An application corresponds to a single instance of the *SparkContext* class in a driver program, which coordinates the computation of one or multiple jobs. It acquires executors, transfers the application code to them and sends information about the tasks the executor should process.

Executor An executor is a JVM instance on a worker node, which runs tasks and keeps data for a single Spark application. It usually stays active throughout the whole lifetime of the application and processes multiple tasks. Executors can also process tasks in parallel if they get multiple cores assigned to them, and multiple executors can run on one worker node.

Cluster Manager The SparkContext of a driver program connects to a cluster manager (e.g., YARN [28]), which allocates resources from worker nodes to individual applications.

Job A job represents a parallel computation triggered by an action on a distributed dataset. It consists of stages that depend on each other and are processed sequentially.

Stage A stage represents one step of a job on a distributed dataset. It is a set of parallel tasks, where each task processes a portion of the data.

Task A task represents a unit of work on one partition of a distributed dataset.

2.1.2 Resilient Distributed Dataset

Resilient Distributed Dataset (RDD) is one of the available data abstractions in Spark applications, which was first described in [29]. RDDs are fault-tolerant collections of records distributed over multiple partitions on the cluster nodes. The RDD API abstracts the partitioning and distribution process

away and allows users to manipulate the collections using operations similar to those available for standard Scala collections.

RDDs support two types of operations:

- transformations - lazy operations that return another RDD (e.g. map, reduce, filter)
- actions - operations that start the computation and return values (e.g. collect, count)

As of Spark 2.0, RDDs have been replaced by DataFrame and Dataset APIs as the primary data abstraction methods. They are conceptually very similar to RDDs, but organize the data into named columns, are more optimized and allow higher-level abstraction (e.g., using SQL style queries). However, RDDs are not deprecated and are still used in many scenarios requiring low-level control of the dataset, e.g., in the MLlib RandomForest libraries.

2.2 MLlib RandomForest

MLlib implements the PLANET algorithm for random forest training on Apache Spark. It follows the same ideas described in Section 1.2, with several implementation differences resulting from using Apache Spark instead of MapReduce and the absence of local subtree training previously discussed in Section 1.4.4.

2.2.1 Overview

Algorithm 2 presents an overview of the main procedure running in the driver program.

Input data D are stored in an RDD and are partitioned by rows as they get distributed across the memory of allocated executors. If the data are too large to fit in memory, the disk (or other secondary storage) is used to store the remaining data. On each iteration, the algorithm splits a set of nodes from *DistributedQueue*. To find the best split, executors compute sufficient statistics on partitions of the distributed data. All statistics for each node are then collected to an executor, which selects the best split. The driver receives information about the best splits, updates the model and enqueues daughter nodes unless stopping conditions are met.

The algorithm pseudocode uses the following methods:

FindSplitCandidates This method performs discretization and binning of features as described in Section 1.2.3.2. It samples a subset of data, computes equi-depth histograms and returns a set of split candidates for every feature.

Algorithm 2 Distributed Training

Require: D = $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
Require: DistributedQueue = {} ▷ holds pairs (TreeId, Node)
1: Splits = FindSplitCandidates(D)
2: push all top nodes for all trees onto DistributedQueue
3: **while** DistributedQueue is not empty **do**
4: Nodes = SelectNodesToSplit(DistributedQueue)
5: BestSplits = FindBestSplits(Splits, Nodes)
6:
7: **for** (TreeId, N, S, D_L, D_R) in BestSplits **do**
8: N → Split = S
9: N → Left = HandleSplit(D_L, TreeId)
10: N → Right = HandleSplit(D_R, TreeId)
11: UpdateNodeIdCache(TreeId, N, S)
12: **end for**
13: **end while**

Algorithm 3 HandleSplit

1: **if** StoppingCondition(D) or D is pure **then**
2: N = new LearningLeafNode
3: **else**
4: N = new LearningNode
5: DistributedQueue → Push ((TreeId, N))
6: **end if**
7: **return** N

SelectNodesToSplit This method dequeues several nodes off the *DistributedQueue* (based on the required memory for their sufficient statistics) and generates a random subset of features to be considered for each node. Note that the *DistributedQueue* is implemented using a stack so that the children of the nodes split in the last iteration and trained next. This way, the algorithm focuses on completing trees rather than training all of them at once, which means fewer trees have to be transferred to executors when a node id cache (further described in Section 2.2.3) is not used.

FindBestSplits This method performs the distributed logic of the algorithm and selects the best split for each node. Executors pass over its partitions of data, and for each node selected in *SelectNodesToSplit*, they collect splitting statistics about every split candidate for the selected subset of features. The statistics for each node are aggregated using *reduceByKey*. From the aggregated statistics, the executors then compute the chosen impurity measure and use it to determine the best split

among the candidates for each node.

HandleSplit This a helper method that checks stopping conditions and creates and enqueues new nodes. Note that the D_L and D_R passed to this method are not the full data subsets, but rather only the information required to check the stopping conditions – instances of the *ImpurityCalculator* class containing subset size and impurity value.

UpdateNodeIdCache This method updates the node id cache with node ids of the new nodes created after splitting. It only updates the indices for data points that belong to the split node in the appropriate tree.

2.2.2 Learning Nodes

During the training, MLlib represents the nodes as *LearningNode* objects. Nodes that have already been split contain information about the split and pointers to left and right children. The node objects can also represent nodes which have not been split yet - these are kept in the *DistributedQueue* and gradually dequeued and processed. Once the training process completes, the model is converted into *InternalNode* and *LeafNode* objects, which removes fields that are only used during training and not required for evaluation.

MLlib uses binary encoding to index nodes. If a node has index i, then its left and right children have indices $2i + 1$ and $2i + 2$. An example of this indexing is shown in Figure 2.2. Note that the indices 10 and 11 are not used to index children of node 6, because they are reserved to encode daughter nodes of node 5.

Binary encoding indices allows for the model to be saved and reconstructed easily without the need to store the tree structure itself, as the indices directly encode the position of the nodes. The downside of this approach is that because the indices are positive integers stored in *Int* primitives, only $2^{32} - 1$ nodes can be indexed this way. It introduces a limit to the maximum tree depth that the algorithm can handle – MLlib itself sets this limit at 30. In the newer RandomForest API in Spark 2.x [30], the finished model no longer relies on indices, but because it uses the same underlying class for training, the maximum depth limit remains.

2.2.3 Node Id Cache

In PLANET, we described how the *ModelFile* is passed to all MapReduce jobs. The trees in the *ModelFile* need to be traversed for the mappers to determine to which node each data point belongs.

MLlib offers the option to use a node id cache, which stores this information. It is an RDD of integer arrays, which store indices of nodes that the data point would evaluate to in the current model for every tree. This means the model no longer needs to be transferred to the executors, which reduces

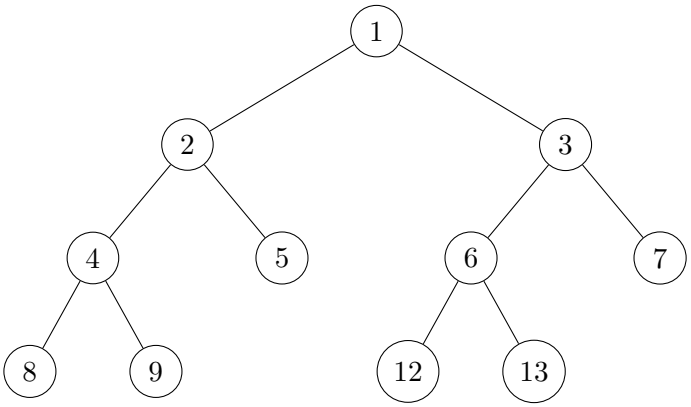


Figure 2.2: Node Indexing Example

communication costs significantly. It does, however, require additional storage memory – for every data point, we keep *NumTrees* extra integers, which can be a substantial increase in memory usage for low-dimension data or a large number of trees.

We choose to enable the node id cache because, for our dataset and testing parameters, the memory increase is acceptable. In addition to reducing communication costs, the cache also allows us to implement local subtree training more efficiently.

Chapter 3

Local Training

In the chapter, we discuss the advantages of adding local training to the MLlib RandomForest implementation. We present the iterations of our custom implementation, starting from adding a local training stage which consecutively trains single decision trees of the random forest, and finishing with an implementation capable of parallel training of multiple nodes across different trees of the forest.

3.1 Advantages of Local Training

Implementing local training in MLlib brings us two main benefits – we drastically reduce the communication cost of the algorithm, which speeds up the training process, and enable training of deeper trees, which can improve model accuracy.

3.1.1 Reducing Communication Cost

We mainly aim to improve the speed of the random forest training, which will enable learning on larger datasets. As mentioned previously, the current implementation is communication-heavy. For each node split using the distributed approach, the executors compute the class counts on their set of partitions and communicate a tuple of sufficient statistics of ($features \cdot bins \cdot classes$) integers. The total communication cost of splitting one node is therefore equal to ($workers \cdot features \cdot bins \cdot classes$). Note that this does not depend on the input dataset size, size of the node subsets or tree level. As the algorithm progresses to lower levels of the tree, the input subsets of the nodes become significantly smaller, but the communication cost stays the same for all nodes.

The number of nodes can grow exponentially with tree depth. A binary tree of depth d contains at most $2^{d+1} - 1$ nodes. Recall that MLlib limits the maximum tree depth to $M \leq 30$, therefore the subtree of a node on level k will have at most $N = 2^{M-k} - 1$ nodes. In the worst case, the communication

cost of fully training the subtree of a node on level k will be ($N \cdot \mathit{workers} \cdot \mathit{features} \cdot \mathit{bins} \cdot \mathit{classes}$). In a setup with 50 workers, training the subtree of a node on level 15 with training data of 300 features discretized into 32 bins in a classification task with 10 classes will in the worst case require communicating about 585 Gb of data.

In contrast, the communication cost of fully splitting a node using local training is only the memory consumed by all data points in its input subset. Assume our input dataset has a total size of 100 Gb. A node on level 15 was already split 14 times – assume it was always split in a 4 to 1 ratio. In this case, the largest nodes have input subsets of size $100 \cdot 0.8^{14} = 4.40$ Gb, which we will be able to use to train locally in most cluster setups. Note that the distributed communication cost further scales with the number of workers, the number of discretization bins and the number of classes, whereas for local training, this does not matter at all.

3.1.2 Training Deeper Trees

Furthermore, implementing local training effectively removes the limit on the maximum tree depth, because the local training algorithms do not rely on node indices. Unless the input data are so massive that we never reach the local training stage, we will be able to train forests of any chosen depth, which could significantly improve the accuracy of the resulting models.

3.2 Local Subtree Training Implementation

The training process is divided into two stages - distributed and local training. We keep the distributed stage described in the previous chapter (Algorithm 2) and modify it slightly to store appropriate nodes in a separate local training queue. Nodes from the local training queue are then processed using our custom local training pipeline.

The distributed splitting continues until all remaining nodes can be split locally, at which point we start the local training. Although simultaneously running both stages is possible, it would only complicate the implementation without bringing any significant performance improvements.

Initially, we choose to locally train one tree at a time. Because we are training a random forest, each data point will belong to different nodes in the individual trees. If we concurrently train only nodes from a single tree, their input subsets will never overlap. This allows us to use simple partitioning logic, and also guarantees that the amount of shuffled data will be at most the size of the dataset because each data point only needs to be present in a single partition.

Algorithm 4 presents an overview of the local training procedure running in the driver program. The algorithm iterates over all trees in the forest and processes smaller batches of local training tasks. First, it filters only the data

belonging to nodes in the processed batch and partitions them by node. Each executor then processes one partition and completes the training of its node by running a local training algorithm. Finally, the model is updated with the completed nodes after collecting them in the driver.

Algorithm 4 Local Training

```
Require: Dataset D                                  ▷ RDD with bagged data points
Require: LocalQueue = {(TreeId, Node), ...}
 1: for CurrentTree in (1 ... NumTrees) do
 2:     TreeNodes = LocalQueue → filter (TreeId == CurrentTree)
 3:     while TreeNodes is not empty do
 4:         Batch = TreeNodes → take (BatchSize)
 5:
 6:         Partitions = FilterAndPartitionData (D, Batch)
 7:         CompletedNodes = RunLocalTraining (Partitions)
 8:         UpdateParents (CompletedNodes)
 9:     end while
10: end for
```

3.2.1 Selecting Nodes for Local Training

First, we need to select the nodes for local training during the distributed stage. Compared to the distributed Algorithm 2, the only addition is a local training queue *LocalQueue*, where we store nodes with input datasets small enough to be split in memory. Therefore we only need to modify the *HandleSplit* method of the original distributed algorithm, as shown in Algorithm 5.

Algorithm 5 HandleSplit for local training

```
 1: if StoppingCondition(D) or D is pure then
 2:     return new LearningLeafNode
 3: else
 4:     N = new LearningNode
 5:     if D → NumRows < LocalTrainingThreshold then
 6:         LocalQueue → Push ((TreeId, N))
 7:     else
 8:         DistributedQueue → Push ((TreeId, N))
 9:     end if
10:     return N
11: end if
```

To decide whether the node is ready for local training, we compare the number of records in its input dataset, which we will call *NumRows*, to a

precomputed threshold *LocalTrainingThreshold*.

To find *LocalTrainingThreshold*, we need take into account the memory allocated to one executor *MaxMemory*, the total number of features *NumFeatures*, number of classes *NumClasses* and the number of bins that each feature is discretized into, *NumBins*.

First we need to make sure that the data itself is small enough. Each data point is stored as a TreePoint object, which contains an integer array with a bin index for every feature value and a floating point label, which takes up approximately $PointSize = 4 \cdot NumFeatures + 8$ bytes. The total approximate size of the data *DataSize* is $NumRows \cdot PointSize$ bytes.

The other factor is the memory used to store the statistics aggregates required to compute the impurity of all splits. Every feature gets discretized into a number of bins, and for each bin, we need to compute label counts for every class. Therefore the approximate size of the statistics for every node *StatisticsSize* is $4 \cdot NumClasses \cdot NumFeatures \cdot MaxBins$ bytes.

The data are stored in JVM objects and further manipulated in the local training implementation, so we need to account for additional memory consumption. We also want to have more control over the size of the tasks. We add an additional parameter *MemMultiplier* and assume that the total memory usage for a given task *TotalSize* is equal to $MemMultiplier \cdot (DataSize + StatisticsSize)$.

To add a node to the local training queue, we need to check that its $TotalSize < MaxMemory$. Therefore, we can compute the *LocalTrainingThreshold* as the value of *NumRows* for which this condition is satisfied.

$$TotalSize < MaxMemory$$
$$MemMultiplier \cdot (NumRows \cdot PointSize + StatisticsSize) < MaxMemory$$
$$\vdots$$
$$LocalTrainingThreshold = \frac{MaxMemory}{MemMultiplier \cdot PointSize} - \frac{StatisticsSize}{PointSize}$$
$$LocalTrainingThreshold \approx \frac{1}{MemMultiplier} \cdot \frac{MaxMemory}{PointSize}$$

Because the value of $StatisticsSize/PointSize$ is negligible, we can now interpret this value as the number of times a data point fits in total memory, reduced by a fraction controlled by *MemMultiplier*.

3.2.2 Partitioning

To locally train a node, first we need to move all of its training data onto a single machine, and in the context of Spark, into the memory of an executor. We can only guarantee that all of the data are together and no data from other

tasks are included when the data are in a single partition. The partitioning process is shown in Algorithm 6.

Algorithm 6 FilterAndPartitionData

Require: Dataset D
Require: Batch = {(TreeId, Node), ...}
1: Partitions = create a partition for every (TreeId, Node) in Batch
2: DataWithNodeIds = GetDataWithNodeIds (D, Batch)
3: **for** (TreeId, NodeId, Point) in DataWithNodeIds **do**
4: Partitions (TreeId, Node) += Point
5: **end for**
6: **return** Partitions

The local training queue contains only nodes that we can split on a single executor. However, if we tried to run multiple splitting tasks on one executor concurrently, we would soon reach the memory capacity and encounter errors. This is the reason we process the nodes from the local training queue in smaller batches of size *BatchSize*. At the beginning of the local training stage, we query the *SparkContext* and set *BatchSize* to the number of available executors, so that each executor always handles one training task.

Additionally, instead of relying on the default grouping and partitioning, we use a custom key partitioner, which distributes data from every node of the current batch into its partition. This gives us absolute control over the number and content of the partitions so that we can distribute the data evenly and ensure each executor only handles one partition.

Recall from Section 2.2.3 that we store the information about the nodes that each data point belongs to in the node id cache. The cache allows us to easily filter the dataset to only include data points from nodes in the current batch, as shown in Algorithm 7. Because we are training nodes from a single tree, we only need to check the appropriate column of the node id cache.

Algorithm 7 GetDataWithNodeIds

Require: Dataset D
Require: Batch = {(TreeId, Node), ...}
1: BatchTreeId = Batch → Head → TreeId
2: BatchNodeIds = Batch → Map (N → Id)
3:
4: DataWithNodeIds = NodeIdCache → Zip (D)
5: **return** DataWithNodeIds
6: → Map ((NodeIds, Point) ⇒
7: (BatchTreeId, NodeIds (BatchTreeId), Point))
8: → Filter ((TreeId, NodeId, Point) ⇒
9: BatchIds → contains (NodeId))

3.2.3 Local Tree Induction

Once the data is partitioned accordingly, we are finally able to run the local training algorithm. Each executor converts the data representation of its partition into a standard Scala *Array* and uses it as training data for the tree induction of the given node (Algorithm 8).

Algorithm 8 RunLocalTraining

Require: Partitions = $\{(\text{TreeId}, \text{Node}, \text{Data}), \ldots\}$

```
1: return Partitions
2:     → MapPartitions ((TreeId, Node, Data) ⇒ {
3:         PointArray = Data → ToArray ()
4:         LocalTreeInduction (Node, PointArray)
5:         return (TreeId, Node)
6:     })
```

We utilize a local training implementation by Siddharth Murching specifically tailored to work with MLlib random forest classes, that is currently in the process of being merged into Spark [31]. It is an implementation of the Yggdrasil algorithm discussed in Section 1.3, which we extended to allow selection of random feature subsets.

Using a Yggdrasil implementation is not ideal in this case, because the data is stored in a row-based RDD and the distributed stage uses horizontal partitioning. This means that the entire partition has to be first transformed into a column-based format, for which the data needs to fit into the executor memory twice. This significantly limits the size of tasks we can train locally. Additionally, the implementation reuses the split candidates proposed in the distributed stage, so we do not gain any accuracy from finding the exact best splits. Therefore, choosing a more suitable local training implementation and making it compatible with MLlib data structures might significantly improve results and is one of the aims for future experimenting.

3.2.4 Updating the Model

In the distributed stage, the splitting is done in the driver program, so we can directly update the model stored in the driver memory. During local training, we perform the whole splitting process on the executors, and the driver then receives a different node object, which we need to plug into the appropriate position in the model.

After we finish the induction of all nodes from the processed batch, we collect the completed nodes in the driver program and update our current model with pointers to them (Algorithm 9). Recall from Section 2.2.2 that we use breadth-first indexing of the learning nodes. This allows us to use only the

node id to easily find the parent of a completed node and determine whether it is the left or right child of that node.

Algorithm 9 UpdateParents

```
Require: CompletedNodes = {(TreeId, Node), ..., }
1: for (TreeId, Node) in CompletedNodes do
2:     Parent = FindParent (TreeId, Node → Id)
3:     if IsLeft (Node → Id) then
4:         Parent → Left = Node
5:     else
6:         Parent → Right = Node
7:     end if
8: end for
```

3.3 Simultaneous Training of Multiple Trees

In this section, we demonstrate how we modified the described algorithm to allow training nodes from multiple trees in a single batch. This may seem counter-intuitive at first, as it increases the complexity of the data partitioning process without any immediate benefits, but it will eventually allow us to optimize the process further, as we will show in Chapter 4.

The original distributed stage already allows splitting nodes from multiple trees together, although the algorithm attempts to minimize the amount of these operations and prefers splitting nodes from a single tree. In the distributed stage, this does not require manipulating the input dataset, as each executor only iterates over its designated partitions and calculates statistics for the given set of nodes, regardless of which trees they belong to.

Recall that because we are building a random forest, each tree will be split differently and the nodes will have different input datasets. If we allow processing nodes from multiple trees at once, a single data point may be present in multiple input subsets of the nodes in the currently processed batch.

In the case of local training, we need the whole input subset shuffled to an executor in one partition. Therefore it is no longer sufficient to filter and partition the input dataset, and we also need to clone the data points which are used in multiple nodes, so that they can be distributed to multiple partitions. Additionally, instead of simply checking one column of the node id cache, we now need to check every (*TreeId*, *NodeId*) pair to determine whether a data point is used during the processing of the batch. This requires extending the *GetDataWithNodeIds* method, as shown in Algorithm 10.

To be able to clone the appropriate data points efficiently, we first precompute a map *TreeNodeSets*, which stores indices of all nodes for every tree in the current batch, and a set of all tree indices in the batch *BatchTreeIds*. For

every datapoint in the input dataset, we filter and map over *BatchTreeIds*, so that we get tuples (*TreeId*, *NodeId*, *Point*) for every node that requires *Point* in its input subset. We concatenate these tuples into one RDD collection using *flatMap* and then use our standard partitioning logic (Algorithm 6).

Algorithm 10 GetDataWithNodeIds for multiple trees

```
Require: Dataset D
Require: Batch = {(TreeId, Node), ...}
TreeNodeSets = Batch
    → Map ((TreeId, Node) ⇒ (TreeId, Node → Id))
    → GroupByKey ()
    → Map ((TreeId, (TreeId, Nodes)) ⇒ (TreeId, Nodes))

BatchTreeIds = TreeNodeSets → Map ((TreeId, Nodes) ⇒ TreeId)

DataWithNodeIds = NodeIdCache → Zip (D)
return DataWithNodeIds
    → FlatMap ((NodeIds, Point) ⇒
        BatchTreeIds
        → Map (T ⇒ (T, NodeIds(T)))
        → Filter ((T, N) ⇒ TreeNodeSets (T) → Contains (N))
        → Map ((T, N) ⇒ (T, N, Point))
```

This process theoretically has the same complexity as partitioning nodes from a single tree, as we still iterate over the same amount of nodes for each data point. However, it requires checking multiple columns of the node id cache, and the amount of shuffled data also increases when the same data points need to be included several times. These factors will increase the communication cost of the partitioning, but the ability to process nodes from multiple trees together gives us more options to select optimized batches and simplifies the local training procedure in the driver program, as demonstrated in Algorithm 11.

Algorithm 11 Local Training of multiple trees

```
Require: Dataset D
Require: LocalQueue = {(TreeId, Node), ...}
while LocalQueue is not empty do
    Batch = LocalQueue → take (BatchSize)

    Partitions = FilterAndPartitionData (D, Batch)
    CompletedNodes = RunLocalTraining (Partitions)
    UpdateParents (CompletedNodes)
end while
```

3.4 Reducing the Number of Tasks Using Bin Packing

The local training queue contains nodes small enough to be split locally, but the sizes of these tasks may greatly differ. Some of these tasks may be close in size to the *LocalTrainingThreshold*, meaning they indeed require the whole memory of an executor, but many tasks will be significantly smaller. Tiny tasks take up only a small portion of the available memory and complete the tree induction very fast, meaning most of the total time required to complete these tasks is spent on partitioning and communication. Additionally, because the entire batch needs to be processed before the driver can collect the completed nodes, the executor will be idle until the longest task in the batch completes – this problem will be further discussed in Chapter 4.

We choose to group these smaller tasks and train them consecutively on one executor, which allows us to minimize the number of batches necessary to finish local training. We create a lower number of larger partitions, which are more balanced in terms of size. It also lowers the number of data shuffles, which removes some overhead of this expensive operation.

3.4.1 Bin Packing Problem

We want to create a minimal amount of task groups, where the total sum of *NumRows* is lower than *LocalTrainingThreshold*. This is an instance of the bin packing problem [32], where our tasks are the items we want to pack into the desired groups, or bins. It is an NP-hard problem, and although several optimized methods for finding the exact solution exist (e.g., [33]), the computation is not feasible for a large number of tasks we may need to pack.

Luckily, decent solutions can be found using greedy approximation approaches, such as the *first-fit decreasing algorithm*. The algorithm sorts the items in decreasing order and then attempts to place each item in the first bin that can accommodate the item. If no such bin exists, it creates a new one.

It has been proved that if we solve the problem using the first-fit decreasing algorithm, the number of the resulting bins will be at most $(11/9\ Opt + 1)$ bins, where *Opt* is the number of bins in the exact optimal solution [34]. Because our main goal is to balance the size of the partitions, this is more than sufficient.

3.4.2 Implementation

Because we now have additional information that we need to store in *LocalQueue*, we create a *Task* class to hold the required data for the local training tasks. It stores the node and tree indices and *NumRows* of the input subset. We then modify the *HandleSplit* method to store these objects in the *LocalQueue*. We also create a *Bin* class, which holds the set of *Tasks* and a

number of *TotalRows* of the currently packed tasks. The bin packing procedure, which we run between the distributed and local training stages, is shown in Algorithm 12.

Algorithm 12 Bin Packing

Require: LocalQueue = {(Task), ...}
Bins = {}
SortedByRows = LocalQueue → SortBy (−Task → NumRows)
for Task in SortedByRows **do**
    PackTask (Task, Bins)
end for

Algorithm 13 PackTask

for Bin in Bins **do**
    if Task → NumRows + Bin → TotalRows < Threshold **then**
        Bin → Tasks += Task
        Bin → TotalRows += Task → NumRows
        return
    end if
end for
NewBin = new Bin
NewBin → Tasks += Task
NewBin → TotalRows += Task → NumRows
Bins += NewBin

Several small changes need to be made to the current local training process. Instead of *LocalQueue*, all methods will now process the *Bins* queue. In the *FilterAndPartitionData* method, we will create a partition for every bin instead of creating it for every node. In the *GetDataWithNodeIds* method, we need to iterate over all nodes for every bin in the batch to determine *TreeNodeSets* and *BatchNodeIds*. In *RunLocalTraining*, each partition now contains data points for multiple nodes, so we need to group them by the node indices before we run the local training algorithm.

Chapter 4

Handling Task Imbalance

In this chapter, we introduce the problem of task imbalance present in our implementation and show how it is currently impairing its performance.

We describe how we collected statistics about the local training process and used them to analyze the problem.

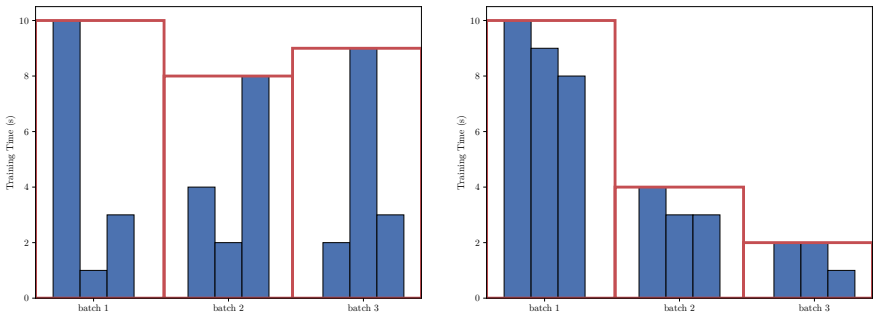
Finally, we present two approaches to task scheduling which mitigate the effects of task imbalance – a universal method based on the data size, and a data specific method using a linear regression model to predict the training time of nodes.

4.1 Task Imbalance

After running experiments with the early versions of our local training implementation, we quickly noticed that the local training process was not optimal. Spark uses a synchronous computing model, so every time we distribute a batch of tasks to executors, all of the tasks in the batch need to complete before we can collect the completed nodes, update the model, and distribute another batch. This means that the total time required to complete the training of every batch is equal to the duration of the longest task in it.

As discussed in Section 3.4, the nodes in the local training queue have different input subsets and take different amounts of time to complete. This is problematic because if we have a batch of 50 tasks, where one task takes 10 minutes to complete and the rest all take 1 minute, the driver has to wait for the longest task to complete, and 49 executors will stay idle for 9 minutes. Although bin packing helps us balance the tasks in terms of row count, the training time differences between the bins can still be significant.

Ideally, we would sort the tasks (or bins) in descending order by training time. That way, the tasks with similar training times would be grouped into batches, and we would minimize the idle time of the executors, as demonstrated in Figure 4.1. However, we do not know precisely how much time



(a) Before sorting (b) After sorting

Figure 4.1: Improvements in total training time from optimal task scheduling

will be spent training the nodes. Therefore we need to make a reasonable prediction of the training time from the information available before running the tasks.

4.2 Collecting Statistics

To tackle this problem, we first need to collect statistics from the local training stage. We modify the *RunLocalTraining* method so that in addition to running the tree induction algorithm, it measures the time it takes for each node to complete the local training. We also save the row count and label entropy of each node, both of which were already computed to determine the best split of their parent nodes during the distributed stage. A structure containing the training time, row count and label entropy is then returned together with the completed node. In the driver program, we collect all of these statistics and log them in a file.

An example of the statistics measured on our dataset, which will be described in Section 5.1, can be seen in Table 4.1.

4.3 Prioritizing Larger Tasks

The intuitive approach to solving the task imbalance problem is to sort the tasks by row count. Using bin packing, this translates to sorting the bins by the total rows count of all their nodes. This makes sense because, for nodes with larger input subsets, it will take longer to complete their full subtree. The nodes in the subtree will take longer to split, as more data needs to be iterated to compute the impurity measure used to determine the best split. The subtree will usually be deeper and therefore contain more nodes because we are less likely to get a pure subset after splitting a node with a larger

	row count	**entropy**	**training time**
1	165023	0.3775	184.05
2	416	0.9883	0.64
3	817	0.1966	1.49
4	104561	0.9955	198.48
6	52450	1.0593	74.49
7	108337	0.3919	130.96
8	117986	0.9460	190.08
9	7004	0.1079	2.12
10	49710	0.0089	6.57

Table 4.1: Statistics collected from *RunLocalTraining*

input dataset. Additionally, these nodes will require shuffling more data to the executor.

In Table 4.1, we can see the that it took the longest to complete the training of nodes 1, 4, 5 and 7, which all had large datasets with over 100,000 rows. This indicates that row count could be used to effectively predict the training time of our tasks. To further analyze this theory, we computed the Pearson correlation coefficient on the complete set of statistics from one training job of a random forest with 40 trees, which contains information about 8735 locally trained nodes. On this set of nodes, the coefficient is $r = 0.4934$.

The coefficient of determination R^2 can be used to measure the goodness of fit of a linear regression model obtained by applying the least squares method – a model minimizing the sum of squares of residuals. Although a more thorough analysis could be done to determine the statistical significance of the predictors, it is not the main focus of this thesis. We will only use the coefficient of determination to measure the potential quality of models and test their actual performance using experiments in Chapter 5.

In a situation with only one observed variable, R^2 is equal to the square of Pearson correlation coefficient r between the observed and target variables, in our case $R^2 = 0.2434$. Although this value would generally indicate a fairly weak predictor, in our case, it can help substantially. We do not need to predict the exact duration of the task, only to preserve the ordering. Because this is a linear model, we can omit the coefficients and intercept, simply sort the tasks by the row count and achieve the same result ordering.

After we implemented prioritizing larger tasks based on row count, the performance of the local training improved drastically and started to outperform the current MLlib implementation by order of magnitude.

4.4 Predicting Task Duration Using Logistic Regression

If we analyze the statistics in Table 4.1 further, we can notice that the training time does not depend only on the row count of the node, but also on the data structure – high label entropy appears to be a good indicator of more complex tasks. We can see that tasks 6 and 10 have very similar data sizes, but the training of task 6, which has much higher entropy, took significantly longer. This is a reasonable assumption because data with high entropy probably contains data points with many different labels, and therefore it will take longer to obtain pure subsets using binary splitting, meaning the full subtree of the node will be deeper. In contrast, nodes with very low entropy might only require a few splits to obtain pure subsets and the final depth of their subtrees will be much lower.

After measuring the Pearson correlation coefficient between the entropy and training time variables, we get a value of $r = 0.4499$, which is comparable to the correlation with row count. Therefore, we would like to use a predictor which combines both row count and entropy. We use the method of least squares to obtain a multiple linear regression model, as described in [35].

In this case with multiple observed variables, the coefficient of determination R^2 will be equal to the square of the coefficient of multiple correlation. On our dataset, we get a value of $R^2 = 0.471$. This indicates a potentially more accurate model than a simple predictor using only row count.

We also suspect that the relation between the variables is not linear. Because the data is very inconsistent and it will never be possible to predict the training time precisely, we did not focus on finding the most optimal model. However, we decided to add additional non-linear variables based on row count and entropy and select the significant ones using stepwise multiple regression. The final predictor has $R^2 = 0.606$ and uses the following combination of row count r and entropy e:

$$c_1 \cdot r + c_2 \cdot e + c_3 \cdot (r \cdot e) + c_4 \cdot \log r + c_5 \cdot (r \cdot \log r) + c_6 \cdot \log e + c_7 \cdot (e \cdot \log e) + c_8$$

Because the predictor now uses a specific combination of the observed valuables measured on our dataset in a specific cluster setup, it is much more biased than the predictor based on row count. While it also improved the performance of local training significantly, as we will show in Section 5.4, we choose to use sorting by row count as the default scheduling method of our implementation and allow users to specify a custom prediction model tailored to their dataset, such as the one described in this section.

Chapter 5

Experiments

In this chapter, we present the results of our experiments with network data, which we use to train random forest classifiers. We describe the cluster setup used for the experiments and compare our implementation with the current MLlib implementation in terms of training time. We also examine the significance of the individual scheduling improvements in our implementation and show how it behaves for different random forest parameters.

5.1 Dataset Characteristics

Our training dataset originates from network traffic data in the form of proxy logs. These logs contain HTTP flows, which represent a single communication between a user and a server. Flows include both directions of the communication and store interesting data about the communication, such as URLs, source, and destination IP addresses, number of transferred bytes, etc. A total of 357 features were extracted from each flow, most of them are based on the URL. A more thorough description of the data and features can be found in [36].

All flows were initially unlabeled. Some positive labels were added on the domain level, either by using available blacklists or manually by human analysts. However, most of these flows remain unlabeled, are considered as innocent network traffic and therefore assigned a negative label. This includes some undetected malicious traffic, which is therefore incorrectly labeled.

For the experiments, we used data from proxy logs recorded in January 2018. Because the whole dataset is huge, we sampled only 1% of the negatively labeled data points. The resulting dataset contains 30,004,828 objects, and 2,149,472 of them are labeled as one of the 152 positive malware classes.

5.2 Cluster Setup

All of our experiments were performed on Amazon Web Services using the Elastic MapReduce service [37]. The experiments were run on a cluster of 11 r4.2xlarge memory-optimized EC2 instances, set up as one master node and ten worker nodes using YARN [28] as the cluster manager. These instances offer 8 vCPUs on Intel Xeon E5-2686 v4 processors and 61 GiB of memory each, leaving us with 80 vCPUs and 530GB available after cluster setup.

5.3 Benchmarking

First, we compare the performance of our final version of the local training algorithm using bin packing and linear regression time predictions (*bin-reg*) against the current MLlib implementation (*old*). As we previously mentioned, MLlib has memory management issues, which leads to executors and eventually entire applications crashing. Therefore, we had to tune the application parameters to be able to complete the random forest training on our large dataset.

Our implementation scales well with the number of executors and performs well even with a large number of small executors with only one core. To fully utilize the cluster resources, we set the number of executors to 80 and executor memory to 6 GB. However, we were not able to complete the training with the current MLlib implementation using this setup. Therefore, our implementation also mitigates some memory issues, as a portion of nodes is saved to the local training queue, and fewer nodes are split concurrently during distributed training.

From our experiments, we noticed that crashes are not tied to the level of parallelism, but rather the memory size of the executors. Using a setup of 30 executors with 15 GB of executor memory with two cores each, we were able to consistently train on our dataset without crashing. Although we only utilize 75% of the available vCPUs on the cluster, the training times are better compared to the setup with 80 executors, because the two threads on the executor share the data and less total communication is required.

We also created another dataset by sampling one-third of the negative data points from the original dataset to show how both algorithms behave on smaller data. This dataset contains 10,498,783 objects, again with 2,149,472 positive samples. Note that the size of this smaller dataset is about 35 GB – in practice, the random forest would probably not be trained on a cluster of this size.

Figure 5.1 and Table 5.1 show the comparison of training times of a random forest classifier with 5 trees using both datasets. The trees are trained to a maximum depth of 30 and with $\sqrt{357} \approx 19$ random features considered for each split. Unless stated otherwise, these parameters are also used in all fol-

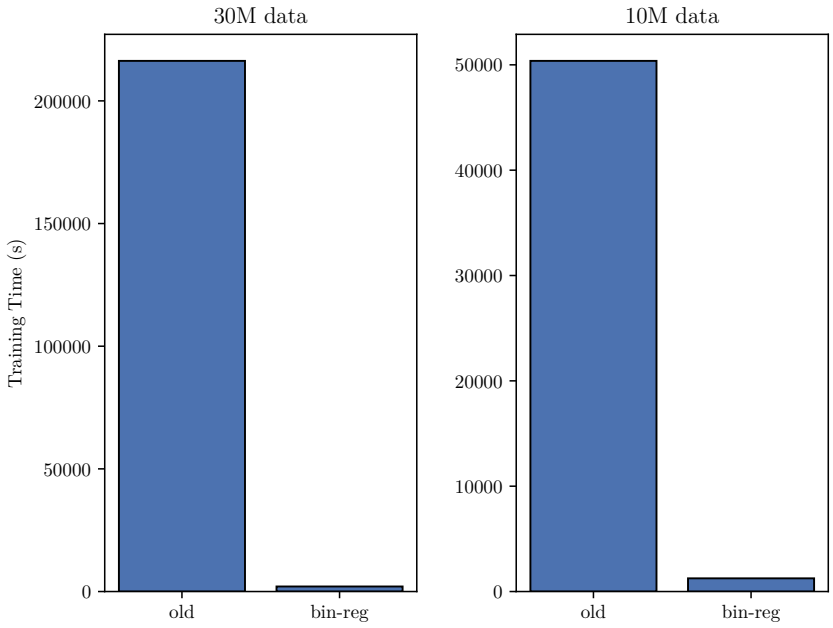


Figure 5.1: Comparison between the training times of MLlib and our implementation for random forests with 5 trees on two datasets

executors	30M data		10M data	
	old	bin-reg	old	bin-reg
30	216304	2056	50374	1252
80	crash	2231	crash	1087

Table 5.1: Training times in seconds for MLlib comparison (Figure 5.1)

lowing experiments. For this experiment, we also tuned the *MemMultiplier*, which was set to 1.2. In other experiments, *MemMultiplier* was set to 4.0, which proved to be unnecessarily pessimistic. We can see that implementing local training and optimizing its task scheduling drastically improves the performance, with our final implementation being over 105× faster on the larger 30M dataset, and over 40× faster on the smaller 10M dataset.

5.4 Comparison of Local Training Methods

In this section, we want to compare the local training methods described in Chapters 3 and 4 and evaluate the effects of the individual improvements.

First, we compare our original local training implementation described in Section 3.2, which trains the forest tree by tree and sorts the nodes by row

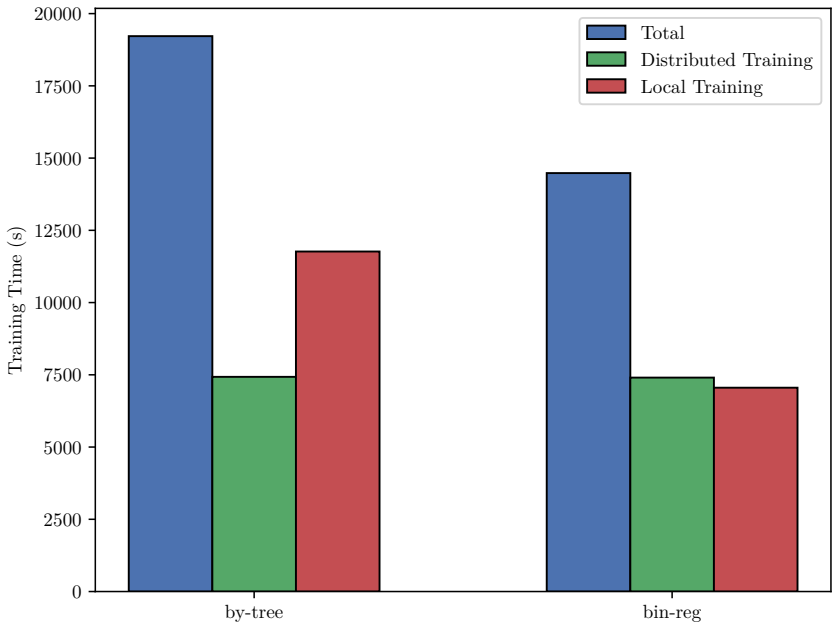


Figure 5.2: Comparison of by-tree and multiple-tree local training scheduling methods for 30 trees with 30M data

count, to our final implementation using bin packing and linear regression time predictions to optimally schedule the training of nodes from multiple trees. To demonstrate the significance of being able to train nodes from multiple trees together, we trained a random forest with 30 trees for this experiment.

In Figure 5.2 and Table 5.2, we can see that the multiple-tree training helps greatly, with the *bin-reg* implementation decreasing the local training time by over 40%. The reason for this is that large training tasks are scattered across all trees, and if we train the forest tree by tree, we are not able to group them during scheduling. Therefore a large number of executors will be idle throughout most of the local training of every tree, waiting for the large tasks to complete. In the multiple-tree implementation, we mitigate the effects of this problem by first bin packing the nodes to balance the number of rows between the tasks and then grouping the bins which we predict will take longer, which is now possible even for nodes from different trees.

Next, we evaluate the significance of our task scheduling optimizations for multiple-tree training. As our baseline method, we use an implementation which schedules the nodes by sorting them by row count and processes them in descending order (Section 4.3). Note that this method already gives better results than training the forest tree by tree, but we focus on evaluating the performance improvements brought by bin packing and time prediction using

method	total	distributed	local
by-tree	19220	7429	11766
bin-reg	14481	7403	7053

Table 5.2: Training times in seconds for the multiple-tree scheduling method comparison (Figure 5.2)

method	total	distributed	local
base	3429	1242	2162
bin	3225	1254	1946
reg	2973	1246	1702
bin-reg	2768	1259	1484

Table 5.3: Training times in seconds for the multiple-tree scheduling method comparison (Figure 5.3)

linear regression.

Figure 5.3 and Table 5.3 show the results of these experiments on a random forest with 5 trees. *Base* uses sorting by row count, *bin* performs only bin packing, *reg* sorts tasks by time predictions from a linear regression model, and *bin-reg* combines bin packing and time prediction scheduling. Note that we only modify the local training scheduling, the distributed splitting remains unchanged and always takes roughly the same amount of time, which is a significant fraction of the total time. If we compare only the time spent on local training, we can see that the final implementation combining bin packing and time prediction yields about 30% improvements over the basic task sorting approach. The performance increase will probably be slightly higher for larger forests, as having more locally trained nodes allows us to schedule them more effectively.

5.5 Effects of Random Forest Parameters

Finally, we show how the performance of our implementation depends on the parameters of a random forest. We focus on the impact of two parameters – random forest size and maximum tree depth.

Increasing the random forest size does not slow down the local training, because the local training batch size remains constant and we only increase the number of local training tasks. We can see that increasing the number of trees in some cases improves the performance of the local training because it results in more local training tasks and that allows optimizing their scheduling. The performance of the distributed training slightly deteriorates for a larger

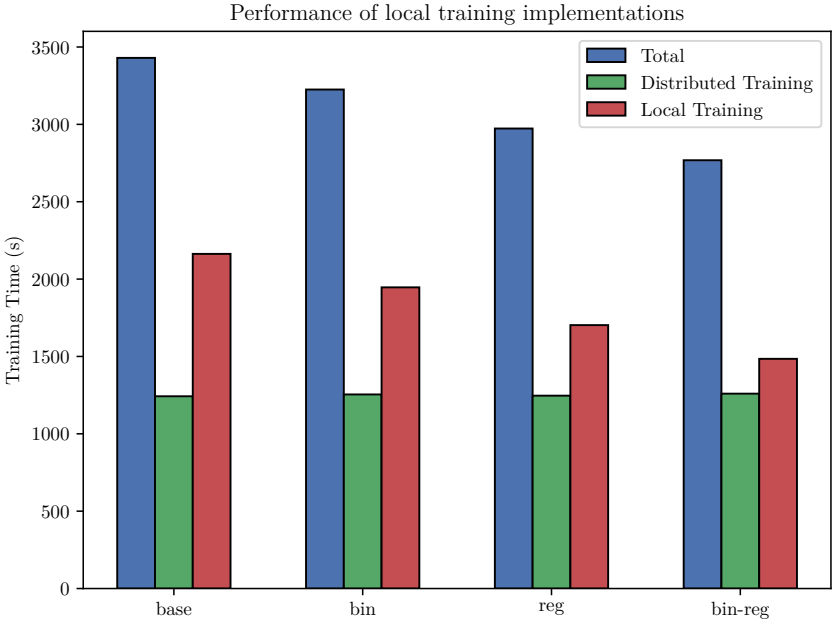
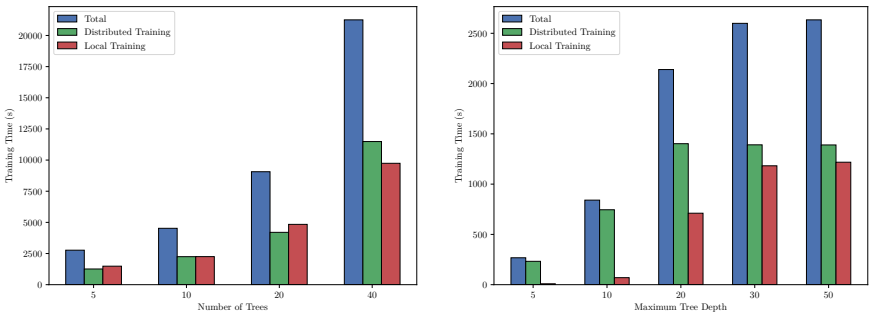


Figure 5.3: Comparison of multiple-tree local training scheduling methods for 5 trees with 30M data

number of trees, but this can be mitigated by training smaller sets of trees and combining the resulting models.

Examining the effects of the maximum tree depth parameter is also important, as it greatly impacts the number of nodes we can train locally. Increasing the maximum tree depth results in the dataset getting split more times, meaning the subsets eventually become small enough to be trained locally. For smaller values of this parameter, almost no nodes are split locally using our setup, and we see that training shallow trees using the only the distributed splitting is feasible. However, using a large number of shallow trees would yield poor classification results for our dataset with a large number of classes.

After reaching level 10, the number of locally trained nodes quickly increases. As more nodes are processed in the local training phase, the overall process speeds up as well. Although the number of nodes in the forest can grow exponentially with tree depth, the overall time increase is acceptable, because we can train the nodes locally and avoid the expensive communication required by the distributed splitting. Note that the time increase between depths 30 and 50 is very small because distributed splitting stops after reaching level 30. Roughly the same amount of nodes will be trained locally and increasing the maximum depth for local training has a negligible performance cost.



(a) Random forest size (b) Maximum tree depth

Figure 5.4: Effects of random forest parameters on training time with 30M data

trees	**total**	**distributed**	**local**
5	2768	1259	1484
10	4524	2247	2251
20	9065	4200	4840
40	21251	11486	9741
max depth	**total**	**distributed**	**local**
5	267	232	8
10	840	745	69
20	2139	1402	710
30	2599	1390	1182
50	2633	1389	1217

Table 5.4: Training times in seconds for random forest parameter comparison (Figure 5.4)

The results of both of these experiments can be found in Figure 5.4 and Table 5.4.

Conclusion

In this thesis, we focused on the training of decision forests on large datasets in a distributed setting.

To give an introduction to distributed random forest training, we thoroughly described the PLANET algorithm and its use of horizontal data partitioning, which is a widely adopted approach to this task. We analyzed its shortcomings, demonstrated an alternative vertical partitioning method used in Yggdrasil, and finally gave an overview of available distributed forest training implementations.

To explain why the current random forest implementation in MLlib performs poorly, we showed how MLlib implements PLANET on Apache Spark, analyzed the communication cost of this implementation, and demonstrated how the absence of local training negatively impacts its performance.

We implemented local training in a fork of the MLlib random forest library using a simple tree by tree approach and then extended the algorithm to support parallel training of nodes from multiple trees and training multiple nodes on a single executor. Implementing local training also allowed us to remove the maximum tree depth limitation and therefore enabled the training of deep decision trees.

Then, we described how task imbalance impairs the performance of the local training process and presented several methods for optimized task scheduling that minimize executor inactivity. Our best performing method uses a linear regression model to predict task duration from the input size and label entropy. We use bin packing to reduce the number of training tasks and balance their sizes.

Finally, we presented the results of our experiments on a network dataset created from proxy logs. We compared the performance of our algorithm and the original MLlib implementation, showed the incremental performance improvements of bin packing and task duration predictors, and demonstrated how the performance of our implementation scales with random forest size and maximum tree depth parameters.

Overall, the proposed improvements resulted in a drastic performance increase and also made the training process more stable. On our dataset, the presented algorithm is up to 105× faster than the current MLlib implementation. We expect an even larger improvement on bigger datasets.

This performance increase allows us to improve predictive performance by training the model on larger datasets, or by increasing number of trees in the random forest. Additionally, local training gives us the ability to train deeper trees, which can also have a positive effect on the model performance.

A classifier for malware detection trained using the algorithm presented in this thesis is actively used in the Cisco Cognitive Threat Analytics system. Thanks to our implementation, we were able to train it using 10× more data than before.

Bibliography

[1] Hastie, T.; Tibshirani, R.; et al. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics, Springer, 2009, ISBN 9780387848846, 305–317 pp. Available from: https://books.google.cz/books?id=eBSgoAEACAAJ

[2] Breiman, L. Random Forests. *Machine Learning*, volume 45, no. 1, Oct 2001: pp. 5–32, ISSN 1573-0565, doi:10.1023/A:1010933404324. Available from: https://doi.org/10.1023/A:1010933404324

[3] Meng, X.; Bradley, J. K.; et al. MLlib: Machine Learning in Apache Spark. *CoRR*, volume abs/1505.06807, 2015, 1505.06807. Available from: http://arxiv.org/abs/1505.06807

[4] Zaharia, M.; Xin, R. S.; et al. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, volume 59, no. 11, Oct. 2016: pp. 56–65, ISSN 0001-0782, doi:10.1145/2934664. Available from: http://doi.acm.org/10.1145/2934664

[5] Panda, B.; Herbach, J.; et al. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *PVLDB*, volume 2, no. 2, 2009: pp. 1426–1437. Available from: http://www.vldb.org/pvldb/2/vldb09-537.pdf

[6] Hyafil, L.; Rivest, R. L. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, volume 5, no. 1, 1976: pp. 15 – 17, ISSN 0020-0190, doi:https://doi.org/10.1016/0020-0190(76)90095-8. Available from: http://www.sciencedirect.com/science/article/pii/0020019076900958

[7] Pedregosa, F.; Varoquaux, G.; et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, volume 12, 2011: pp. 2825–2830.

[8] H_2O. https://www.h2o.ai/h2o.

[9] Alsabti, K.; Ranka, S.; et al. CLOUDS: A Decision Tree Classifier for Large Datasets. In *KDD*, AAAI Press, 1998, pp. 2–8.

[10] Jansson, K.; Sundell, H.; et al. gpuRF and gpuERT: Efficient and Scalable GPU Algorithms for Decision Tree Ensembles. In *IPDPS Workshops*, IEEE Computer Society, 2014, pp. 1612–1621.

[11] Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, volume 51, no. 1, Jan. 2008: pp. 107–113, ISSN 0001-0782, doi:10.1145/1327452.1327492. Available from: http://doi.acm.org/10.1145/1327452.1327492

[12] Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. *CoRR*, volume abs/1603.02754, 2016, 1603.02754. Available from: http://arxiv.org/abs/1603.02754

[13] Hadoop MapReduce. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[14] Google App Engine: MapReduce Overview. https://web.archive.org/web/20130120053524/https://developers.google.com/appengine/docs/python/dataprocessing/overview.

[15] Yildiz, B.; Büyüktanir, T.; et al. Equi-depth Histogram Construction for Big Data with Quality Guarantees. *CoRR*, volume abs/1606.05633, 2016, 1606.05633. Available from: http://arxiv.org/abs/1606.05633

[16] Abuzaid, F.; Bradley, J. K.; et al. Yggdrasil: An Optimized System for Training Deep Decision Trees at Scale. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 2016, pp. 3810–3818. Available from: http://papers.nips.cc/paper/6366-yggdrasil-an-optimized-system-for-training-deep-decision-trees-at-scale

[17] Apache Parquet. https://parquet.apache.org.

[18] Yggdrasil implementation. https://github.com/fabuzaid21/yggdrasil.

[19] [SPARK-3162]: Train Decision Trees Locally When Possible. https://docs.google.com/document/d/1baU5KeorrmLpC4EZoqLuG-E8sUJqmdELLbr8o6wdbVM.

[20] Apache Mahout. https://mahout.apache.org.

[21] [MAHOUT-1510]: Goodbye MapReduce. https://issues.apache.org/jira/browse/MAHOUT-1510.

[22] H2O: Building Random Forest At Scale. https://www.slideshare.net/0xdata/rf-brighttalk.

[23] Random Forest Benchmark. https://github.com/szilard/benchm-ml.

[24] Chung, S. H. Sequoia Forest: A Scalable Random Forest Implementation on Spark. https://pdfs.semanticscholar.org/presentation/a1c1/0b3767d73105aa1184985e38c3ea9b4a54a2.pdf, 2016.

[25] Gieseke, F.; Igel, C. Training Big Random Forests with Little Resources. *CoRR*, volume abs/1802.06394, 2018, 1802.06394. Available from: http://arxiv.org/abs/1802.06394

[26] Spark wins Daytona Gray Sort 100TB Benchmark. https://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html.

[27] Apache Spark: Cluster Overview. https://spark.apache.org/docs/2.3.0/cluster-overview.html.

[28] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[29] Zaharia, M.; Chowdhury, M.; et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. Available from: http://dl.acm.org/citation.cfm?id=2228298.2228301

[30] Apache Spark MLlib: Random Forest Classifier. https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier.

[31] [SPARK-3162] [MLlib] Add local tree training for decision tree regressors. https://github.com/apache/spark/pull/19433.

[32] *Bin-Packing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-29297-5, pp. 426–441, doi:10.1007/3-540-29297-7_18. Available from: https://doi.org/10.1007/3-540-29297-7_18

[33] Schreiber, E. L.; Korf, R. E. Improved Bin Completion for Optimal Bin Packing and Number Partitioning. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, AAAI Press, 2013, ISBN 978-1-57735-633-2, pp. 651–658. Available from: http://dl.acm.org/citation.cfm?id=2540128.2540223

[34] Yue, M. A simple proof of the inequality FFD (L) 11/9 OPT (L) + 1, L for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, volume 7, no. 4, Oct 1991: pp. 321–331, ISSN 1618-3932, doi:10.1007/BF02009683. Available from: https://doi.org/10.1007/BF02009683

[35] Kutner, M. *Applied Linear Statistical Models*. McGrwa-Hill international edition, McGraw-Hill Irwin, 2005, ISBN 9780071122214, 214–255 pp. Available from: https://books.google.cz/books?id=0xqCAAAACAAJ

[36] Bartos, K.; Sofka, M. Robust Representation for Domain Adaptation in Network Security. In *Proceedings, Part III, of the European Conference on Machine Learning and Knowledge Discovery in Databases - Volume 9286*, ECML PKDD 2015, Berlin, Heidelberg: Springer-Verlag, 2015, ISBN 978-3-319-23460-1, pp. 116–132, doi:10.1007/978-3-319-23461-8_8. Available from: https://doi.org/10.1007/978-3-319-23461-8_8

[37] Amazon EMR. https://aws.amazon.com/emr/.