UPPSALA UNIVERSITET

UPTEC X 15 040
Examensarbete 30 hp
Januari 2016

Distributed Ensemble Learning With Apache Spark

Simon Lind

UPPSALA UNIVERSITET

Degree Project in Bioinformatics

Masters Programme in Molecular Biotechnology Engineering,

Uppsala University School of Engineering

UPTEC X 15 040	**Date of issue 2016-01**

Author

Simon Lind

Title (English)

Distributed Ensemble Learning with Apache Spark

Abstract

Apache Spark is a new framework for distributed parallel computation for big data. Is has been shown that apache spark can execute applications up to 100 times faster than Hadoop. Spark can show these speedups in iterative applications, such as machine learning, when the data is cached in memory. Big data does however make this difficult due to the sheer volume. In this thesis I attempt to address this issue by training multiple models using Spark, and combining their output results. This is called Ensemble Learning and can reduce the required primary memory for big data machine learning and potentially speeding up training and classification times.
To assess Sparks usability in Big Data Machine Learning, a large, unbalanced dataset was used to train an ensemble of Support Vector Machines. The assessment was based on the performance of the ensemble as well as both the weak and strong scaling of the applications required to implement ensemble learning in Apache Spark.

Keywords
Big Data, Apache Spark, Machine Learning, Support Vector Machines, Ensemble Learning

Supervisors

Kenneth Wrife
Omicron Ceti AB

Scientific reviewer

Andreas Hellander
Uppsala Universitet

Project name	Sponsors
Language **English**	Security
ISSN 1401-2138	Classification
Supplementary bibliographical information	Pages **46**

Biology Education Centre	Biomedical Center	Husargatan 3, Uppsala
Box 592, S-751 24 Uppsala	Tel +46 (0)18 4710000	Fax +46 (0)18 471 4687

Populärvetenskaplig sammanfattning

Simon Lind

Big Data är en ny term inom både vetenskapen och den kommersiella marknaden som fått ökad spridning de senaste åren. Big data kan definieras som data som anländer i hög hastighet, har stor volym och hög variation. Denna typ av data för med sig stora möjligheter, men den kräver också speciella åtgärder för att man ska kunna hantera och analysera den.

Det distribuerade ramverket för datalagring och behandling Hadoop har den senaste tiden blivit mycket populära eftersom det tillåter enkel distribuering av både data och beräkningar i ett data-kluster. Hadoop är dock begränsat eftersom data ofta måste läsas och skrivas från disk i iterativa applikationer, vilket tar lång tid. Apache Spark löser detta genom att tillåta att data lagras i det snabbare, primära minnet i ett datakluster. Detta tillåter applikationer som behöver tillgång till data ofta att köras upp till 100 gånger snabbare.

Ett möjligt användningsområde inom big data är ”Machine Learning” vilket innebär att man tränar en modell genom att analyserar data av känd data karaktär. Modellen kan sedan användas för att avgöra okänd datas karaktär. När stora datamängder hanteras måste man välja vilken data som ska användas för att träna modellen. Mycket information kan förloras i detta steg eftersom en stor del inte används. Detta kan lösas genom att man tränar flera modeller på olika delar av tillgängliga data för att sedan kombinera deras bedömningar. Detta kallas ”Ensemble Learning”. Ensemble Learning kan leda till en mer träffsäker bedömning på ny data eftersom modellerna lärt sig olika saker och kan bidra med olika synvinklar på problemet.

I detta examensarbete har jag implementerat och utvärderat metoder för Ensemble Learning med Apache Spark på stora mängder, obalanserad data. Resultatet visar att det är möjligt att implementera Ensemble Learning metoder som skalar nära linjärt i Apache Spark och att dessa kan exekveras upp till 10 gånger snabbare än tidigare försök på samma data.

Examensarbete 30 hp Civilingenjörsprogrammet Molekylär bioteknik, inriktning Bioinformatik
Uppsala universitet, januari 2016

Table of Content

Abbreviations

AM	Application Master
auPRC	Area Under Precision Recall Curve
auROC	Area under Receiving Operating Curve
FN	False Negative
FP	False Positive
FPR	False Positive Rate
G	Gigabyte
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
I/O	input /output
JVM	Java Virtual Machine
LR	Logistic Regression
M	Megabyte
PRC	Precision Recall Curve
RDD	Resilient Distributed Dataset
ROC	Receiving Operating Curve
SGD	Stochastic Gradient Descent
SVM	Support Vector Machines
TN	True Negative
TNR	True Negative Rate
TP	True Positive
YARN	Yet another Resource Negotiator

1 Introduction

1.1 Background

The amount of data that is being collected by companies and scientists has increased greatly over the last decade. The Big Data trend shows no signs of slowing, and predictions say that Big Data is here to stay (Laney 2015; Kambatla et al. 2014; "2015 Predictions and Trends for Big Data and Analytics | The Big Data Hub" 2015). Big data can be used to find patterns, which can be used for many things. For example, Netflix uses data gathered from its user base to predict movie recommendations (Töscher, Jahrer, and Legenstein 2008). Another example is the 1000 genome project, where huge amounts of genetic data is being collected continuously, with the goal to map all the genetic variants of the human genome (The 1000 Genomes Project Consortium 2010).

Machine learning (ML) is a field in data science where one supplies an algorithm with data, from which the algorithm "learns" to identify patterns and correlations (Kohavi and Foster 1998). Big Data combined with ML can be a very powerful tool for actors on the commercial market and scientists alike when it comes to pattern detection and decision making. Big data makes it possible to find patterns and correlations that previously were undetectable using small data and analogue data gathering methods, such as interviews or surveys, simply because of the lack of information. However, when collecting big data, it is not uncommon for one type of data to be more common than another type of data (He and Garcia 2009). One example is email spam detection. In their latest report of email metrics of q2 2012, Messaging, Malware and Mobile Anti-Abuse Working Group (M3AAWG) states that approximately 80% of all emails sent were of an "abusive" nature, meaning it somehow seeks to exploit the recipient ("M3AAWG Email Metrics Report 16" 2014). The sum of all emails represent an imbalanced dataset, where 80% are of an abusive nature, and 20% are non-abusive. Another example of imbalanced data would be trying to identify criminals in a population. The vast majority are law abiding citizens whereas only a few are criminals. These types of datasets and problems are becoming the norm rather than the exception in modern ML. A search on "imbalanced data" on Uppsala University Library showed that the number of peer reviewed publications on the subject has increased by over 800% from 2000 to 2014

("Uppsala University Library" 2015). This increase shows a growing interest in these type of problems, and that knowledge about how to go about solving them is a valued resource both commercially and scientifically.

One way to handle the processing of big data is to parallelize the applications that use the data. The MapReduce programming model is a popular method for parallelizing applications in a computer cluster. The MapReduce model divides the computations into a number of Maps that reads the data and writes its results to disk, which is then read and aggregated by the reducers.

Apache Spark is a framework for parallelized computations in a computer cluster, which allows data to be stored in memory for fast access. This is especially suitable for iterative applications like machine learning. Spark also allows for parallelization of data and applications in a cluster, allowing them to be processed and executed simultaneously. It has been shown that some applications execute up to 100 times faster than MapReduce based methods (Zaharia et al. 2010). Apache Spark could therefore be a suitable next step for machine learning on big data in distributed file system due to its state of the art framework for iterative processing of data in memory.

1.2 Purpose and Aims

Big data requires big storing capabilities. One common solution is to compress the data, making it occupy less space on disk. This solution does, however, increase the time required for the data to be read from disk, which can be very detrimental for applications that read from disk repeatedly. Spark allows us to compressed data on disk without it having a strong negative effect on the overall execution time since Spark only requires one pass over the compressed data to store it in memory as a Resilient Distributed Dataset (RDD). Once the data is in memory, it will not have to be decompressed again (Zaharia et al. 2012).

If the dataset is too large, Spark will either re-compute the RDD using the RDDs lineage or spill it to disk for later access. This will severely deteriorate performance due to disk I/O. To address this, I suggest that the data can be divided into subsets that fit into memory. Training will be performed on data stored in memory, and each subset will produce one classifier. Each classifier will then be used to make predictions, which are later combined to make a final decision. This is called Ensemble learning(Witten, Frank, and Hall 2011, 351–373). Ensemble Learning have been used previously, it utilizes the diversity in the ensemble of classifiers in the ensemble to produce predictions with greater performance than a single classifier would (Yang et al. 2010; Xie, Fu, and Nie 2013; Polikar 2006; King, Abrahams, and Ragsdale 2015; Chawla and Sylvester 2007) . Ensemble learning using Spark would not only reduce the need of disk space, but also the RAM required, since data can be compressed on disk, and since the only subsets of the dataset has to be cached.

In this thesis, I will attempt to contribute to the state of research in Big Data ML by investigating the possibility of a scalable, big data solution for ensemble learning on imbalanced datasets using Apache Spark.

More specifically, the aims of this thesis are to evaluate Apache Spark usability in Big Data ML by implementing core ensemble learning applications in Apache Spark. These applications will be evaluated in regards to:

- Scalability
- Speed
- Classifier Performance

The work in this thesis was performed with a small 6 node cluster provided by Omicron Ceti. The cluster hardware configuration will be evaluated according to the following application areas in Big Data ML:

- Development
- Testing
- Production

2 Theory

2.1 Big Data

Big Data is difficult to define. It is usually described as a large amount of data in the terabyte to petabyte range, which is hard to store using conventional data storage methods (White 2012). Since the ability to store and process large data volumes are growing, the limit for what can be called “Big Data” is always changing.

One popular way to describe big data is the “three V’s” of big data which was first introduced in 2001 as *Volume, Velocity and Variety* (Laney 2001). These three V’s aim to explain the properties of big data other than just being “big” and what problems big data brings with it as well as what is required if one aims to utilize big data.

Volume refers to the “big” part of big data. It simply means that we have many bytes which we need to decide if we should store, and if so, how we should store it.

Velocity is the aspect of how fast data is generated, and processed. Other than the need for high bandwidth networking, this also puts pressure on the applications to process data as fast as it is received if real-time decisions are needed.

Variety of data means that the data is not always consistent on terms of quality, format or content. This can be one of the most problematic aspects of big data, since the data might have to be reviewed before it can be processed, adding a time consuming step in the processing pipeline.

These three V’s have been under some scrutiny from the community, and additional V’s have been purposed in addition to the three original (van Rijmenam 2015; Swoyer 2012; Shukla, Kukade, and Mujawar 2015), such as “value”, and “veracity”. However, many of these additions to the original Vs are not anything unique to Big data. Small data can also differ in value and veracity, but only big data has large volume, arrives in high velocity an in high variety. Therefore, only Volume, Velocity and Variety are used to define big data in this thesis. More specifically, in this thesis big data refers to data too large to store in the primary memory of a single computer.

2.3 Apache Hadoop

Apache Hadoop is a framework that allows for distribution of data storage and data processing in a computer cluster ("Welcome to Apache^{TM} Hadoop®!" 2015). Hadoop makes each node in a cluster take the role as both a storage and computational node, improving the data locality. The Apache Hadoop project was first started in 2002 as the Nutch project and was later acquired by Yahoo! and renamed Hadoop. Hadoop is designed to scale to several thousands of nodes and is widespread in the industry of big data. The Hadoop project contains and supports several frameworks which allows developers and users to leverage Hadoop distributed framework in several different application areas, ranging from data storage, to ML algorithms ("Welcome to Apache^{TM} Hadoop®!" 2015).

Data locality refers to where the data is stored in relation to the computations (Guo, Fox, and Zhou 2012). Limited data locality can be described as a situation in which the data must be transferred a lot between nodes. In High Performance Computing (HPC) the data locality is limited since all the computation nodes have to access the data via network connections, creating a bottleneck in applications where much data is transferred (Guo, Fox, and Zhou 2012). This is all and well when working with computationally heavy applications that do not require a lot of communication, or when using high performing networks. However, when faced with problems that require a lot of communication between nodes, and high performance networking is not available, the Hadoop framework might be the better choice.

Apache Hadoop improves upon the data locality of many applications by distributing the data, keeping replicas of the data across several nodes, and by instead of transferring data, moving the computations to the data instead, decreasing the total amount of data transferred between nodes, making communication intensive applications possible using commodity hardware, instead of expensive high end hardware (Guo, Fox, and Zhou 2012).

2.3.1 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is a framework in the Hadoop project for managing the distribution of data which is too large to store on a single node to several nodes in a cluster. When storing large amounts of data indexing of the data becomes necessary (White 2012, chap. 3). HDFS solves this by having two types of nodes in a cluster. The NameNode, which keeps the metadata in memory, and a DataNode, which stores the data. This way the NameNode can quickly point to what data block on which DataNode where one can find the data. The HDFS also supports replication of data in the cluster, providing high availability of data should one node fail. This allows for jobs to continue on other nodes where the data is replicated (White 2012, chap. 3).

2.3.2 YARN

YARN, Yet Another Resource Negotiator or MapReduce2 is the newest iteration of the resource negotiator in the Hadoop framework (White 2012, chap. 4). YARN is responsible for distributing and queuing applications across the cluster, while also allocating resources to the application. YARN aims to decrease the amount of data transfer occurring in the cluster by placing applications on the nodes where the data is located instead of moving the data itself. If a node is busy, a new application will be placed in a queue and wait until the needed resources are released (White 2012, chap. 4).

The ability to distribute data across a computer cluster as well as the computations efficiently, gives the Hadoop framework an advantage compared to conventional HPC methods in communicational intense applications. Traditional HPC makes each computation node transfer the data from a shared data storage to local storage via some kind of network connection, which gives it a disadvantage in communicational intense applications (Guo, Fox, and Zhou 2012). When applications use large amount of data or output large amounts of data, traditional HPC become network bound due to the amount of data that is being transferred. Using Hadoop, there is much less data transfer since both the data and computations are distributed across the cluster thanks to the HDFS and YARN.

When a user submits an application to the cluster, YARN first starts an application master (AM), which then requests the needed resources from the cluster resource pool. YARN then allocates a number of containers to the application (White 2012, chap. 4). A container is an abstraction in YARN which refers to a pool of resources which are allocated to one job. A container has a minimum size and a maximum size in regards of memory and virtual cores. When the containers have been allocated, the application starts. The containers communicate with the AM who in turn communicates with YARN. When the application has finished, the containers and the AM are decommissioned and their resources are returned to the cluster pool. YARN also allows multiple users to utilize a cluster by only requesting the resources needed for their applications. Additionally, resource pools can be allocated to different users, only allowing them to access resources within their designated pool. This makes YARN useful for when the cluster is used by several users (White 2012, chap. 4).

2.4 Spark

Spark is a framework for cluster computing that has been develop with iterative applications in mind (Zaharia et al. 2010). It further decreases the amount of data transfer needed compared to MapReduce applications in Hadoop by storing data in the primary memory, instead of writing it to disk after each job, and read at the beginning of each job, as is done in conventional MapReduce. Depending on the replication factor specified, this might have to be done several times at the start and end of each job. Additionally, in MapReduce a new JVM is started for each new job. This can be very time consuming, especially if there are many jobs to be done. Spark solves this by keeping two types of JVMS active until the application finishes, the driver and its executors. The executors are responsible for the calculations and data caching required by the application. Each executor and the driver occupy one YARN container each and are as such restricted by the maximum, and minimum allowed memory and virtual cores specified in the YARN settings (Zaharia et al. 2010).

A Spark job is divided between the executors and its driver (Zaharia et al. 2010). The driver is responsible for job scheduling and communication with YARN. The Spark driver is in fact running within a YARN Application Master, giving Spark the ability to communicate with yarn directly, instead of through a separate AM. In traditional MapReduce, after a job is finished, the JVM is decommissioned, and the application masters assigns a new JVM to the next job. In contrast to this, Spark starts each application by starting a JVM for each executor, to which the driver can assign a job to an executor directly. When a job is finished, the JVM is kept online, speeding up the overall execution time (Zaharia et al. 2010).

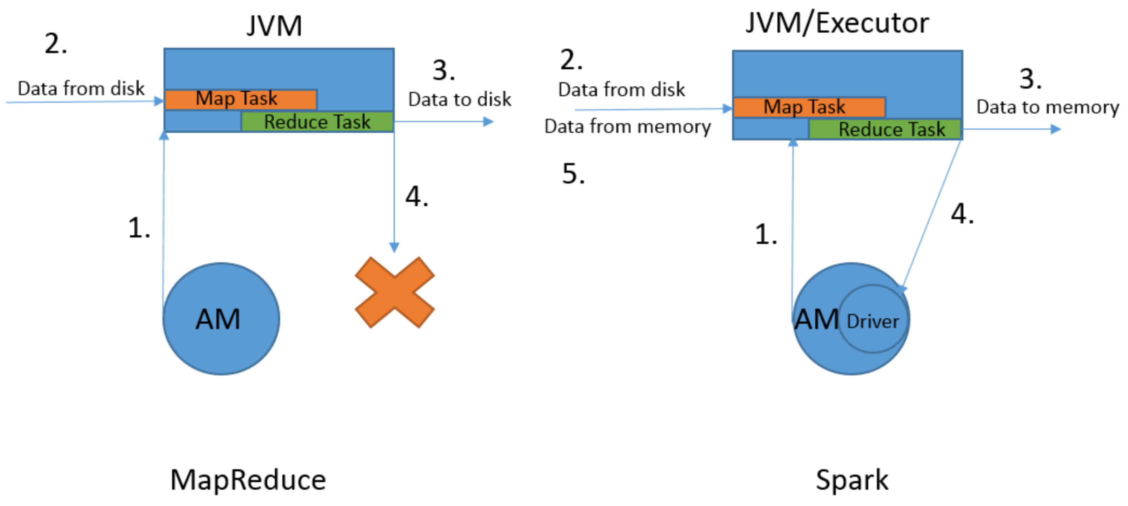


Figure 1 The Differences in how MapReduce and Spark manages job scheduling and its JVMS. In MapReduce, the AM first starts the JVMs and then assigns it a task with corresponding tasks (1). The JVM reads input data from disk (2), performs the computations and writes the output to disk (3) after which the JVM is decommissioned and the process restarts. In Spark, the driver within the AM requests the JVMs from the cluster (1) and assigns it with a task. The data is read from disk (2) and computations are performed. The output is written to memory (3) after which a signal is sent to the driver to inform it that the task is finished (4). A new task is assigned to the same JVM, who now can read the input data from memory (5).

Spark does however not work within memory entirely. When results from tasks must be aggregated from across the cluster, spark tries to keep these results in memory, however when they no longer fit, they will be spilled to disk in stage 3 of Figure 1. This aggregate of results to a single node is called the shuffle stage, and can have a serious impact on performance, since it requires both disk and network I/O. Reducing the amount of shuffles required in an application can therefore improve performance greatly (“Spark Programming Guide - Spark 1.5.1 Documentation” 2015).

Spark provides a native Machine Learning library (MLlib). MLlib is the native ML algorithm library available in Spark. It provides several ML algorithms and methods for evaluating the same (“MLlib | Apache Spark” 2015). This library will be the main source for ML algorithms in this thesis.

2.4.1 Resilient Distributed Datasets

Spark also introduces Resilient Distributed Datasets (RDD). An RDD is read-only collection of object that can be partitioned across the nodes in a cluster, allowing it to be accessed by several executors, and for the applications to work on the data in parallel

(Zaharia et al. 2012). An RDD is not computed as it is defined. RDDs supports so called "lazy" transformations. The transformations are instructions on how the data should be derived. Since a RDD can be derived from a series of lazy transformations, a roadmap over how the RDD should be computed is stored as the user defines the transformations. These roadmaps are called the RDDs "lineage". Using the lineage, the data can be replicated if an executor is assigned a job but does not have the entire RDD stored in its own primary memory. The lineage can also be used to restore the data, should a node or an iteration fail, providing fault tolerant data management (Zaharia et al. 2012). If the dataset is larger than the memory available, the RDD can be spilled to disk. This will require some disk I/O, and it will affect performance but it will keep the application running even if there is not enough memory available (Zaharia et al. 2012).

2.6 Evaluation of Scalability

One of the most important aspect of cluster computing is the ability to scale. Since both the data and problem complexity is always evolving, big data solutions need the ability to scale both in problem size, and cluster size. To evaluate performance in parallel scalability of the implementations in this thesis, two methods will be used to measure two different aspects of scalability, *speedup* and *scaleup*.

2.6.1 Strong Scaling - Speedup

When doing a speedup study, the number of computational nodes, *K*, are increased*,* while keeping the data size the same, *N,* as visualized in Figure 2. In ideal situations, the execution time should decrease linearly with the addition of computation nodes. The speedup tells you how the applications execution time will decrease as you add more resources to the problem.

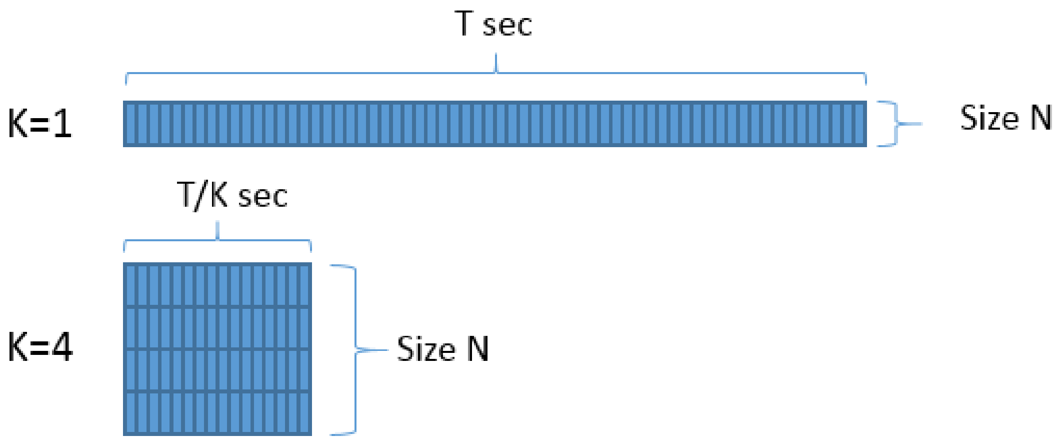


Figure 2 Visual representation of the data size N, number of nodes K and execution time T in a speedup study (Alan Kaminsky 2015).

The speedup gained from increasing the cluster size is calculated according to:

$$Speedup(N, K) = \frac{T_S}{T_p(N, K)} \qquad EQ.1$$

Where T_s is the execution time on a single node with data size N and T_p (N, K) is the execution time on K nodes with data size N.

The efficiency measure how close to “ideal scalability” the application is scaling. The efficiency is calculated according to:

$$Efficiency(N, K) = \frac{Speedup(N, K)}{K} \qquad EQ.2$$

In ideal situation the speedup is equal to K and the Efficiency is 1 (Alan Kaminsky 2015). However, an ideal speedup is not always possible. Amdahl’s law divides an application into two parts, one that can benefit from improving or adding the resources available, and one that cannot. The latter is called the non- parallelizable fraction. Depending on how large this non- parallelizable fraction is, one can expect it to affect the speedup negatively at various degrees (Amdahl 2007). This effect is visualized in Figure 3.

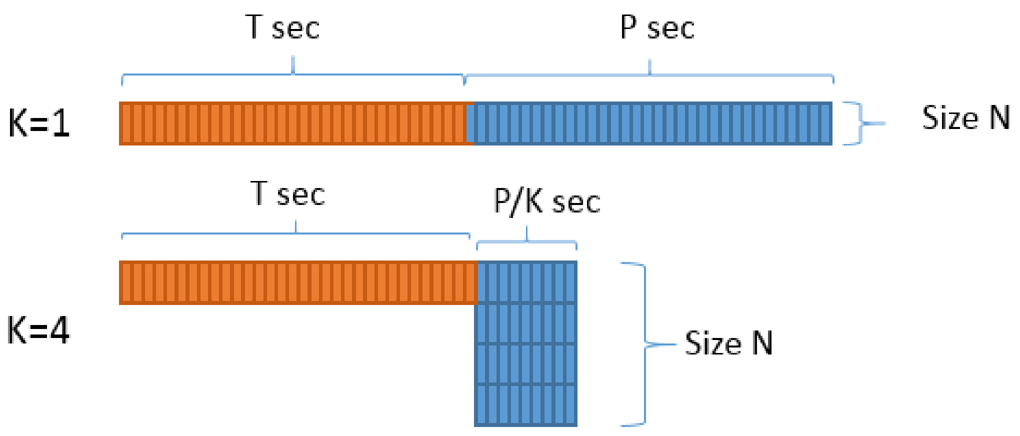


Figure 3 A visual representation of how the non-parallelizable fraction (orange) can affect the speedup of an application

2.5.2 Weak Scaling - Scaleup

In a scaleup study, the data size and the number of computational nodes are increased. E.g. if the data amount is doubled, the number of nodes in the cluster is also doubled. This is visualized In Figure 4. In an ideal situation the execution time is expected to stay constant as the problem, and the cluster grows in size. The scaleup can give one insight in how the well the application parallelizes as the problem and resources grow.

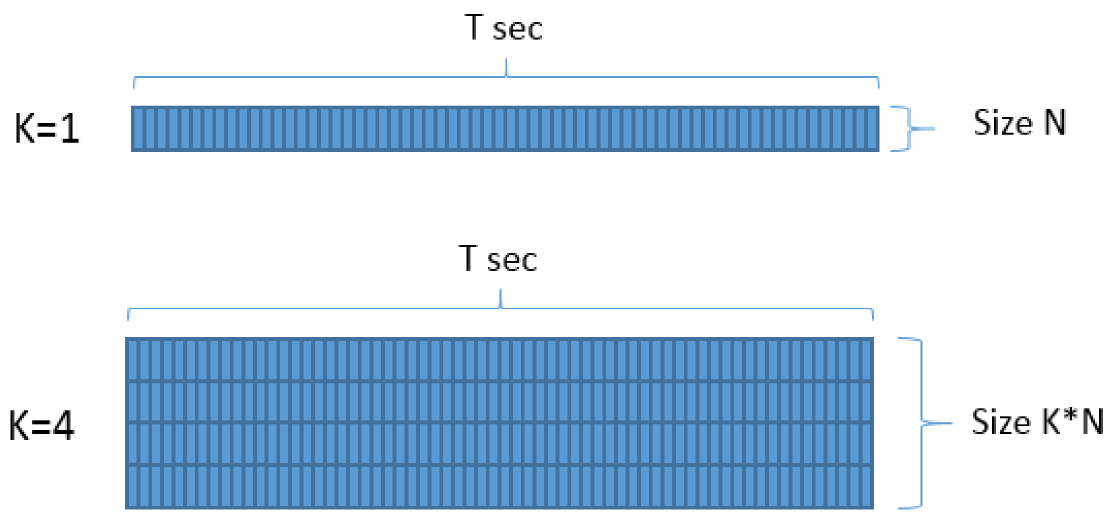


Figure 4 Visual representation of the data size N, number of nodes K and execution time T in a scaleup study (Alan Kaminsky 2015).

The Scaleup is measured according:

$$Scaleup(N,K) = \frac{N(K)}{N(1)} \frac{T_S(N(1),1)}{T_p(N(K),K)} \qquad EQ.3$$

Where *N(K)* is the data size on *K* nodes.

Similar to the efficiency of the speedup, the efficiency of the scaleup is calculated according to:

$$Efficiency(N,K) = \frac{Scaleup(N,K)}{K} \qquad EQ.4$$

Again, in ideal situation the scaleup is equal to K and the Efficiency is equal to 1 (Alan Kaminsky 2015).

2.6 Machine Learning

Machine Learning is the process of finding patterns and correlations in data by using different methods for analyzing this data. The goal is to be able to make predictions on future data based on what was learned on previous data (Witten, Frank, and Hall 2011; Kohavi and Foster 1998).

In this thesis the ML methods will fall into the supervised learning category. That is the algorithms will be presented with the solution and try to find the patterns according to the solution. These methods are often called classifiers as each instance in the dataset has a set of attributes and decision class, which the ML algorithm tries to determine (Witten, Frank, and Hall 2011; Kohavi and Foster 1998). For example, the dataset of emails. Each email would be an instance in the dataset, and its content would be its attributes. The decision class of the email could be if it is of malicious nature or not (as previously explained in 1.1). The goal of the supervised learning algorithm would be to find patterns in the emails content, and correlate these to the decision class of the email. The patterns found could later be used to classify emails of unknown nature.

2.6.2 Linear Support Vector Machines with Stochastic Gradient Decent

Linear Support Vector Machines, (SVM) is a popular method for big data classification. The SVM algorithm attempts to separate instances in the training set by representing each instance as a support vector composed by its attributes (Witten, Frank, and Hall 2011, chap. 6). A hyperplane that separates the support vectors of the different classes is then computed iteratively. The hyperplane aims to minimize the errors made, i.e. support vectors on the wrong side of the hyper plane, while maximizes the distance between the hyper plane and the vectors of the different classes. The vectors closest to the hyper plane are its support vectors and are used to define the hyper plane (Witten, Frank, and Hall 2011, chap. 6). Figure 5 shows a simple 2D maximum margin plane with the support vectors outlined.

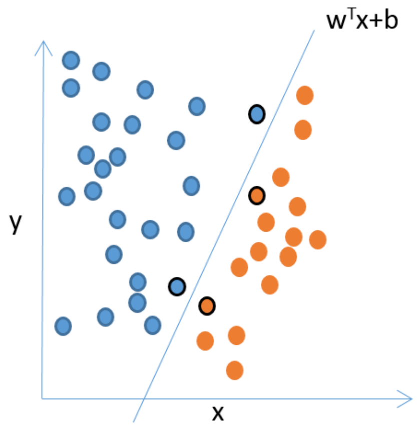


Figure 5 A 2-dimensional feature space with a maximum margin hyperplane, w^{T}x+b, separating two classes, orange and blue. The outlined dots are the support vectors. b is a bias which can be defined to move the hyperplane closer to one class cluster, preferring the other class.

As of November 2015, The SVM implementation in MLlib only supports Stochastic Gradient Decent (SGD) optimization and will as such be used as the optimization algorithm.

SGD in MLlib aims find the maximum margin hyperplane from n data points by solving the following optimization problem:

$$min_{w \in R^d} F(w), \qquad F(w) := \lambda \frac{1}{2}||w||_2^2 + \frac{1}{n}\sum_{i=1}^{n} \max\{0,1 - yw^T x\}$$

Where $x_i \in R^d$ are the feature vectors, and y is each feature vectors corresponding label. w is a vector of weights which represent the hyperplane.

λ is the regularization parameter. λ defines the cost of a training error and regulates the tradeoff between minimizing the training error, and minimizing the complexity of the model which is being trained (“Linear Methods - MLlib - Spark 1.5.1 Documentation” 2015).

SGD seeks the optimal solution by “walking” in the direction of steepest decent of the sub-gradient. The gradient can be computed using a subset of the dataset in memory. The size of this subset can be specified using the *mini-batch fraction* (“Optimization - MLlib - Spark 1.5.1 Documentation” 2015). A smaller mini batch would require less computations being made, resulting in faster execution times. Additionally, a mini-batch fraction would allow for some randomness in the training. If the full dataset was used, the direction of the steepest decent would always point in the same direction, which could lead to some local optima, resulting in one missing the global optima. In MLlib SGD implementation, the step-size decreases for each iteration, allowing for more fine adjustments as the training progresses. The step-size γ is defined as:

$$\gamma = \frac{s}{\sqrt{t}}$$

where s is the initial step size, and t is the iteration number (“Optimization - MLlib - Spark 1.5.1 Documentation” 2015).

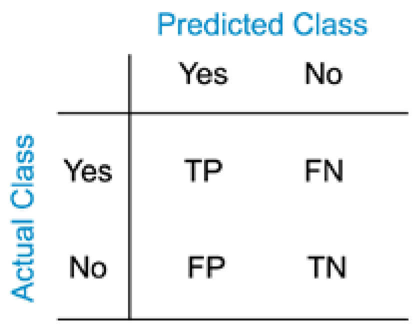
The resulting SVM model makes predictions based on $w^T x$, where the decision class is predicted to be either positive or negative if $w^T x$ is greater or lesser than a defined threshold. This threshold should be selected to maximize the classifier accuracy or some other quality measure (“Linear Methods - MLlib - Spark 1.5.1 Documentation” 2015).

SVM are however biased towards the majority class, since minimizing the errors made on that class will reduce the total error, which the SGD aims to minimize. One must therefore keep the class distribution in mind when training SMVs on imbalanced datasets.

2.6.4 Evaluation of Classifier

An essential practice in ML is to divide the data into a training and test set. The algorithm only sees the training set, where the classes are known. The classifiers performance is then tested on the test set, where the classes are hidden from the model.

When testing the classifier, you get a number of correct predictions, true positives (TP) and true negatives (TN), and a number of incorrect predictions, false positives (FP) and false negatives (FN). Using these, a confusion matrix (Figure 6) can be constructed which can be used for further analysis (Witten, Frank, and Hall 2011, chap. 5).



		Predicted Class	
		Yes	No
Actual Class	Yes	TP	FN
	No	FP	TN

Figure 6 Binary confusion matrix

From the confusion matrix one can compute the true positive rate and the false positive rate according:

$$True\ Positive\ Rate = \frac{TP}{(TP + FN)} \qquad EQ.5$$

$$False\ Positive\ Rate = \frac{FP}{(FP + TN)} \qquad EQ.6$$

The Receiver Operating Characteristic Curve, or ROC can then be constructed by measuring the TRP and FPR of a classifier at different thresholds. The area under the curve, auROC gives an approximation of how accurate a classifier is in general. Ranging from 1 to 0, where 1 is a perfect classifier, and 0.5 is no better than random chance (Figure 7) (Witten, Frank, and Hall 2011, 629).

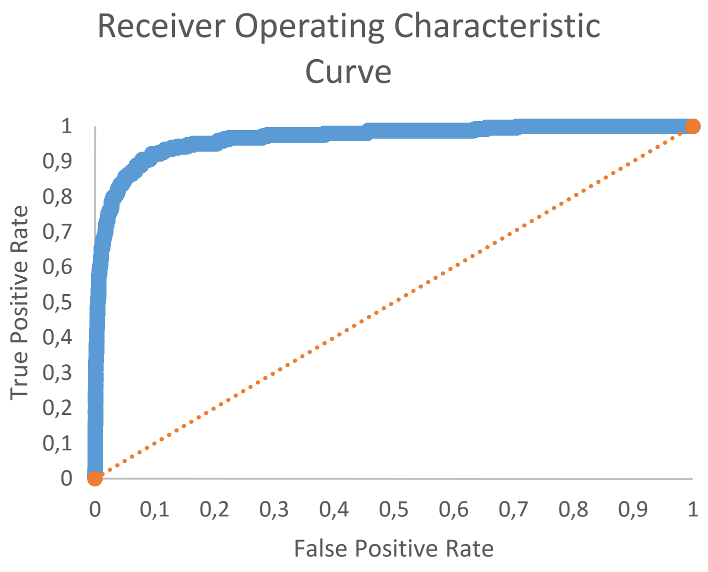


Figure 7 Receiver operating characteristic curve of one classifier. Each blue dot represents the classifiers TPR and FPR different thresholds. Dotted orange line represents a random classifier, which is no better than random guessing.

When building classifiers on imbalanced datasets however, the auROC is not a suitable metric for classifier performance. In a highly imbalanced dataset, one can reach a high auROC, while still being unable to accurately predict the minority class. For instance, if a classifier is trained on a 1:100 class imbalance, the classifier would reach an auROC of 0.99 by simply classifying each instance as the majority class. A classifier such as this is by no means accurate. In these cases the Precision Recall Curve (PRC) is a much more suitable metric (Davis and Goadrich 2006).

Precision refers to the fraction of correctly identified positives over the total amount of instances classified as positive.

$$Precision = \frac{TP}{TP + FP} \qquad EQ.7$$

Recall is defined as the fraction of correctly identified positives over the total amount of positive instances (Witten, Frank, and Hall 2011).

$$Recall = \frac{TP}{TP + FN} \qquad EQ.8$$

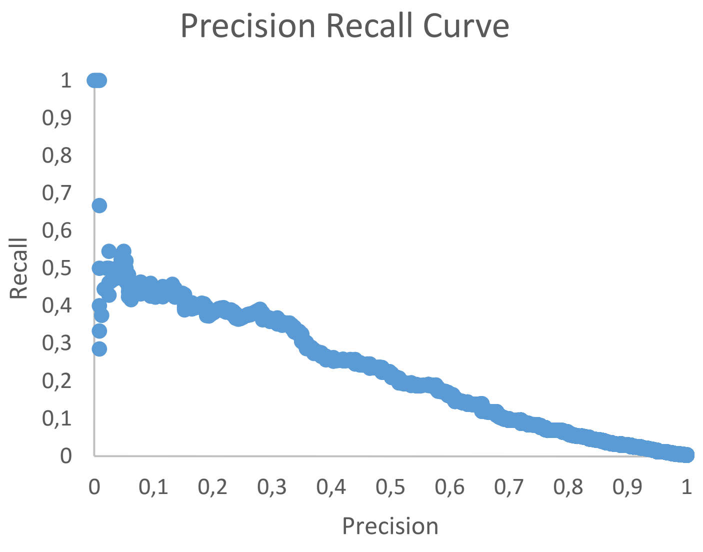


Figure 8 A Precision Recall Curve. Each dot represents a classifiers Recall and Precision at different thresholds.

One can record the precision and recall of different thresholds and construct the PRC (Figure 8). From the PRC the area under the precision recall curve (auPRC). This can be used as a measure of how well a classifier can catch positive instances and at its precision. A high auPRC tells us that the classifier has a high recall while maintaining a high precision.

Using both the auPRC and auROC, one can choose what threshold the final classifiers should use to make their predictions. The chosen threshold should yield a recall and precision suitable for the application at hand. For example, a model for diagnosing a disease should have a high recall, even at the cost of precision, since a false negative is much more expensive than a false positive.

2.6.4.4 F-Measure

The F-measure is a harmonic average of both the precision and the recall of a model at a given threshold (Witten, Frank, and Hall 2011, 175). The F-measure effectively sums up the models ability to recall positive instances, and at what precision, in a single metric. The F-measure is calculated according:

$$F_\beta = (1 + \beta^2) * \frac{precision * recall}{(\beta^2 * precision) + recall} \qquad EQ.9$$

The F-measure allows us to consider both a models recall and precision at the operating threshold when deciding on the importance of a models prediction.

The F-measure also allows us to change the Beta value, to put more or less emphasis on the recall of the model. A beta value of 2 would value a high recall over precision, whereas a beta value of 0.5 would value precision over recall (Chinchor 1992).

2.6.2 Ensemble Learning

Ensemble learning has in several cases proven to provide ensemble classifiers whose accuracy were comparable to classifiers trained on the entire dataset (Polikar 2006; Yang et al. 2010; Chawla et al. 2003). Using Ensemble Learning, new data can be added to the classifiers in batches as it arrives. As old classifiers grow obsolete, their performance will deteriorate, and when a classifiers performance falls below a certain threshold, it can be discarded.

The process would begin with splitting the dataset into subsets, using some suitable sampling technique. Each subset would be sued to train one classifier each in sequence. These classifiers make up the ensemble of classifiers. Each classifier in the ensemble is then allowed to make predictions on the same data point. Their resulting predictions are then combined to make one final ensemble prediction. This way, you can utilize the strengths of one classifier, while not allowing the weaknesses of the individual classifiers to affect the end result. Figure 9 shows a visual representation of the workflow of an ensemble.

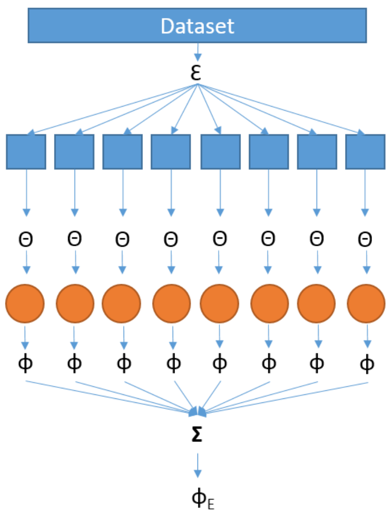


Figure 9 A visual representation of the workflow of ensemble learning and ensemble classification. Initially the dataset is divided into subsets using some subsampling technique ε. Each subset is used to train a classifier using some training scheme Θ. Each classifier then makes individual predictions Φ on input data. The predictions are combined using some voting scheme Σ to produce a final prediction $Φ_E$.

One drawback of doing ensemble learning is of course that making predictions will require more time since more predictions are made. However, it has previously been shown that the extra time for classification does not exceed that of the reduce time required for training since ensemble learning allows of classifiers to be trained entirely in memory (Chawla et al. 2003).

When predicting the class of a new instance using ensemble methods, the most intuitive technique is a simple majority vote. The different models simply vote on what class they predict an instance belongs to and the majority decides. Since each classifier will be trained on different subsets of the data, their performance will differ. One classifier might be much more accurate than another. A simple majority vote might not be the best way to combine individual predictions. In these cases, one can give the more accurate classifier more weight in the voting scheme. Another, more prudent way to weigh a majority vote, is to do so by some sort of quality measure. Previously a F-measure weighted majority vote has been used, it proved to be more accurate than the standard majority vote (Chawla and Sylvester 2007). Each classifier will have its F-measure calculated on an unseen validation set for different thresholds. To performed a F-measure weighted majority vote, the threshold that yields the highest F-measure bust first be computed. The corresponding F-measure will be used as the classifiers individual voting weight.

This method does add an additional step when classifying new instances since the F-measure and corresponding thresholds must be computed. Luckily spark allows us to cache a small validation set which will be used to find F-measure. The validations set will be a randomly sampled subset of the training data with equal class distribution as the test set. This way, I avoid fitting my models to the test set, which would yield misleading results.

In MLlib, the model outputs a raw score. This score can be used to classify the instance based on a threshold. Average Majority Vote takes the average value of the raw scores from the different classifiers before comparing the score with the threshold. Using an average majority vote, both a PRC and ROC can be constructed, representing the average performance of the ensemble.

3. Method

To evaluate Sparks usability in big imbalanced data ML applications, three vital parts of big data ensemble ML will be implemented in Spark. Hyper parameter tuning, Ensemble Training, and ensemble classification. To bring the problem into the realm of big data, the Splice-Site dataset was selected. The dataset was selected because of its large volume (3T), and imbalanced class distribution, 1 positive instance for each 300 negative instances. Additionally, previous work has been conducted on the same dataset, which makes comparisons possible (Sonnenburg and Franc 2010; Agarwal et al. 2014).

3.1 Material and Configuration

3.1.1 The cluster

The hardware for this study is a 6 node cluster, with 1 master node and 5 worker nodes.

Each worker node is equipped with 16G ram, of which 4G are reserved for OS, and other running applications. This leaves us with 12G per worker node. Each worker node has an intel i5 quad core processor of which 3 will be reserved for yarn containers.

Initial testing showed that the minimum memory require for the driver was 2G + 1G overhead for the SVM training application. This leaves us with only 8G + 1G per worker node. This yields a total of 40 G. It was decided that 4 executors with 11G +1G was the preferable configuration, since it would allow for more work to be done by each executor, while still allowing for more data to be cached than by the 5 executor configuration.

As for the classification application disk I/O and data locality will most likely be the limiting factor. As such a 5 executor configuration is more suitable for this application since the data is distributed across all 5 worker nodes.

The cluster uses Hortonworks HDP 2.3 distribution and Spark runs at version 1.4.1

A more extensive summary of the Spark settings and cluster hardware can be found in Appendix B.

3.1.2.1 Data prepossessing

Data preprocessing is a vital part of ML. Data must sometimes be sanitized of ill formatted data points. The data can also be discretized, yielding a more generalized data set. Since data preprocessing only requires one pass of the data being processed, this part of the ML process will not benefit from being implemented in Spark. Performance measurements of this step is therefore out of the scopes of this thesis.

It is not always possible to find linear solutions to the problem we want to solve. In these cases, we might end up with a $O(n^2)$, or even worse $O(n^n)$ time complexity. When considering big data, where n->∞, it is clear that these kind of non-linear solutions quickly becomes unfeasible. One way to deal with these non-linear correlations is to expand the feature space. In an expanded feature space one might find a linear solution to a non-linear problem. A simple way one might expand the feature space is with a polynomial kernel. A dataset with features x and y, whose instances cannot be linearly separated, can have an expanded feature space of; x, y, xy, xx, yy where a linear solution can be found (Witten, Frank, and Hall 2011). Figure 10 shows a simple visual representation of how an expanded feature space might allow for linear separation. There are many other kernel methods, some of which are more suitable for some applications than others. Some care must therefore be taken when deciding what kernel to use.

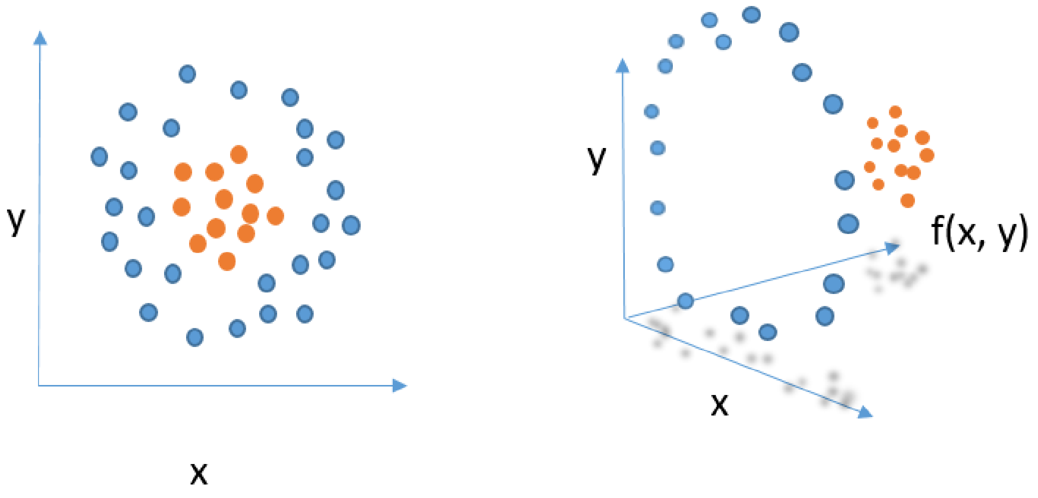


Figure 10 A simple schematic of how a feature expansion by applying some arbitrary function "f" on the variables x and y can make linear solutions possible on a nonlinear problem.

The Splice Site training set is composed of 50 000 000 sequences of human DNA. These are sites that are either splice sites, or non-splice sites, represented as a label of 1 and 0 respectively. Additionally, 4 600 000 instances available for classifier evaluation. From the raw sequences, the feature space was expanded to 12 725 480 features were originally derived by (Sonnenburg, Rätsch, and Rieck 2007),using a weighted degree kernel of d=20 and gamma =12. The same kernel was also used by (Agarwal et al. 2014) and (Sonnenburg and Franc 2010), which makes it ideal for comparative purposes. The same kernel was applied to the sequence data using the Shogun toolbox (Sonnenburg et al. 2010) and a slightly modified version of the script used by (Agarwal et al. 2014) to parallelize the computation in the Hadoop framework. The modifications rendered the dataset readable by MLlib native LIBSVM functions. The data is stored on HDFS with 3 replications across 5 executors.

3.1.2.1 Compression

Since the data will be loaded into memory, the applications only requires one pass of the data stored on disk. This allows us to store the data in a compressed format without the extra time needed for decompression severely increasing the execution time. Due to storage limitations compression of the data is necessary. BZip2 and Snappy compression codecs were compared against raw data in a simple test, where 120.000 rows of the data were read from HDFS into memory and the total number of attributes were counted, to be followed by 10 SGD iterations.

3.2 Ensemble Application Settings

Due to time constraints, only one ensemble will be trained. To ensure that this ensemble is the best possible, a series of tests were performed to identify the optimal settings for the ensemble.

3.2.1 Data Sampling

Since the dataset is highly unbalanced, with only 143 668 positives instances out of a total of 50 000 000, each individual classifier in an ensemble will only be allowed to train on a small fraction of these. The imbalance can be addressed by oversampling the minority class by replication. This does however not add any new information, but only strengthens the bias towards the minority class, which can cause overfitting. Overfitting means that the models will not be able to accurately classify new data points that differ from the ones used during training. An over fitted model cannot not guarantee generalization when making future predictions. To attempt to improve the models ability to generalize over positive instances, two different sampling methods of positive instances will be evaluated. First the original dataset with the original class ratio will be used. The second alternative is to extract all positive instances from the full dataset, and always include them in the ensemble training. The latter method will induce under sampling of the majority class. Under sampling the majority class to provide a less imbalanced dataset does reduce the bias towards the majority class, but it does however generate information loss, often leading to less accurate classifiers (He and Garcia 2009). To investigate to what extent under sampling of the majority class will cause information loss, the majority class will not only be under sampled to the point of a 1:1 class ratio, but also to approximately a 1:3 class ratio. The latter would result in a larger total data volume for each model since a 1:1 class ratio between the classes would not utilize the entire cacheable memory fraction without some type of oversampling of the minority class. A larger dataset with a 1:3 class ratio could reduce the information loss since it utilized more data.

Easy Ensemble is a subsampling technique that under samples the majority class using random sampling with replacement to the point of no imbalance. An ensemble of classifiers trained on all positive instances and different subsets of randomly sampled negative instances is thereby trained (Liu, Wu, and Zhou 2006). True random sampling is however not very efficient. It would require data to be loaded into memory and for singe data points to be selected at random. To increase efficiency and reduce data transfer, the data will be randomly sampled from disjoint partitions. Randomly sampled disjoint partitions has been shown to yield comparable results to classifiers trained on subsets produced by true random sampling (Chawla et al. 2003).

3.2.2 Hyper-parameter Tuning using Random Grid Search

To determine the optimal settings for the Ensemble SVM application one must tune the parameters of the individual classifiers. These are; the initial step size, regularization parameter, the number of iterations, the mini batch size and whether or not to under sample the majority class. To do this a random grid search will be used. The random grid search trains several models in sequence, where the parameters are randomly sampled from a uniform distribution in a given range. This method is not only reliable but has also shown to find the global optima in faster than a standard exhaustive grid search (James Bergstra and Yoshua Bengio 2012).

The random grid search will choose a small subset of original data or a dataset with all positive instances and randomly under sampled negative instances, on which several models will be trained with different parameter settings. The **models** will then be evaluated using a small validation subset from the training data. The model that performs best on the validation subset will then be evaluated on the test set.

To assess the performance of the classifiers in the random grid search the auPRC will be the dominant metric. However, since MLlib native auPRC functions interpolate between data points, a high auPRC can be very misleading (Figure 11) (Davis and Goadrich 2006). Therefore, the auROC will also be used as a supplementary metric to assess the classifiers performance.

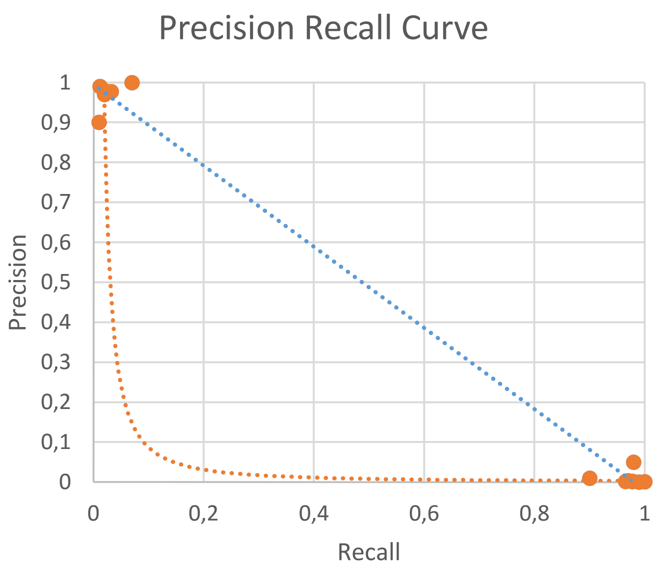


Figure 11 Misleading Precision Recall curve. Blue line show interpolated curve, orange shows true curve. Adapted from Figure 6 in reference (30).

The random grid search will search the parameter space of the following parameters with values randomly selected from the following pools:

- **Number of SGD iterations:**
  - [20, 50, 100, 200, 300, 400, 500, 600]

- **Initial step size:**
    - [50, 100, 150, 200, 250, 300, 350, 400, 450 ,500, 550, 600,650,700,750,800]
- **Regularization Parameter:**
    - [10^{-4}, 10^{-5}, 10^{-6}, 10^{-7},10^{-8}, 10^{-9}]
- **Mini Batch fraction:**
    - [0.1, 0.2, 0.3, 0.4, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

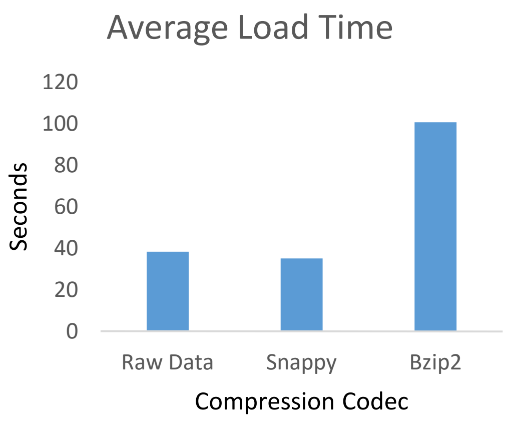
The random grid search will train models on either; Randomly selected original data with replacement, or all positive Instances and randomly under sampled negative instances with replacement. Regardless of the sampling technique, the models will be trained on approximately 250 000 rows of data.

4. Results

4.1 Compression Codec

To determine what compression codec should be used, a simple test was performed where 120 000 rows were compressed using Snappy and Bzip2 compression codes. The final size of the data and the time required to load the data was compared to that of raw data. The results are shown in Figure 12-13.

As expected, the average iteration time for the SVM was not affected by the compression compared to raw data since there is no difference when it has been read into memory as an RDD. The average time required for the data to be read into memory does however vary. The BZip2 codec does show a high compression ratio, it does however take twice as long to read data into memory compared to raw and snappy compressed data (Figure 12-13). Due to it being read into memory as fast as raw data, and its good compression ratio, the Snappy codec will be used to decrease the need for disk space, at small to no cost of execution time.



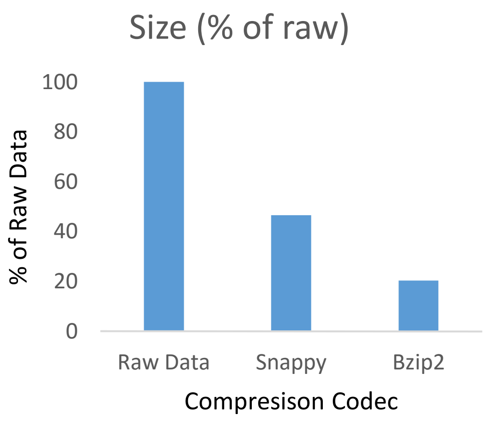


Figure 12 Time to load 120 000 rows of data.

Figure 13 Data Volume. Percentage of Raw data

4.2 Scalability

This section will cover the results of the scaling studies. The speedup, speedup efficiency, scaleup and scaleup efficiency are calculated according to EQ.1-4 respectively.

4.2.1 Hyper parameter tuning and Training

The hyper parameter tuning and training applications in this thesis both revolve around two core aspect when scalability is concerned. These are the ability to load data into memory, and to performed SGD iterations. This part of the thesis covers the scalability of these two core aspects of the applications in this thesis.

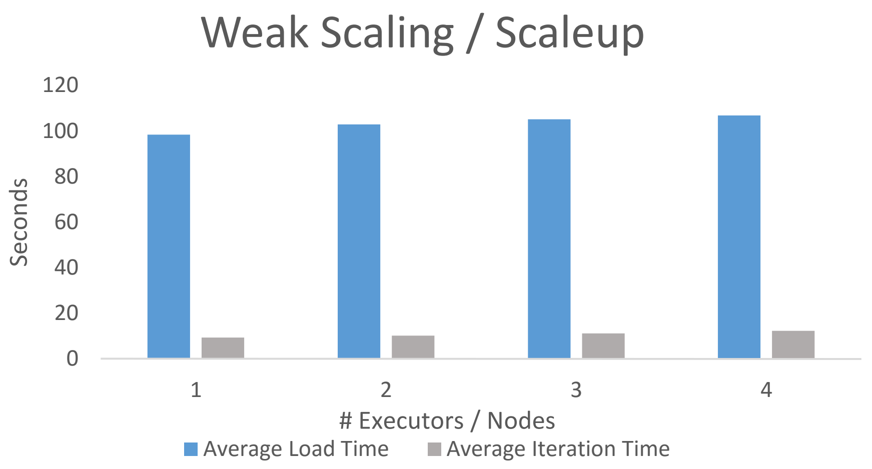


Figure 14 Scaleup study across 4 nodes. Data was read in multiples of 3.7 G snappy compressed data (156 778 rows, 5,75 G RDD in memory). Average of 3 Load times and 10 SGD iterations

Table 1 Scaleup and efficiency of the Data Loading and Training scaleup study

#Executors	Load Time Scaleup	Load Time Efficiency	Iteration Scaleup	Iteration Efficiency
1	1	1	1	1
2	1,91	0,96	1,83	0,91
3	2,81	0,94	2,52	0,84
4	3,69	0,92	3,03	0,76

Figure 14 and Table 1 show the results from the weak scaling study of the load time and iteration execution time. In this study the input data volume was increased by 3.7 G snappy compressed data as nodes were added. The load time and iteration scaleup and efficiency were derived from the average of 3 and 10 runs respectively.

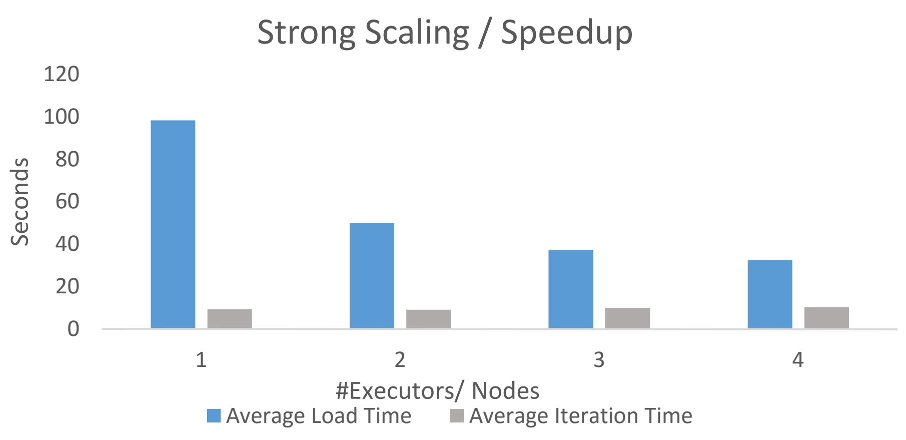


Figure 15 Speedup study across 4 nodes. 3.7 G snappy compressed data (156 778 rows, 5,75 G RDD in memory). Average of 3 Load times (blue) and 10 SGD iterations (grey).

Table 2 Speedup and efficiency of the Data Loading and Training Speedup study

#Executors	Load Time Speedup	Load Time Efficiency	Iteration Speedup	Iteration Efficiency
1	1	1	1	1
2	1,97	0,98	1,03	0,51
3	2,64	0,88	0,94	0,31
4	3,03	0,76	0,90	0,22

Similarly, Figure 15 and Table 2 shows the results from the strong scaling study of the load time and iteration execution time. In this study the input data was kept the same, at 3.7 g snappy compressed data, as nodes were added to the cluster. Again, the load time and iteration scaleup and efficiency were derived from the average of 3 and 10 runs respectively.

4.2.2 Classification

When measuring the scalability of the Ensemble classification there are three steps of measurement, time to load the models, time to calculate the voting weights, and finally the time for ensemble classification. Further in the scaleup study, both the input data for classification and the number of models can grow. Therefore, two scaleup studies was performed, one where the input data is increased, and one where the number of models is increased. The validation set is cached in memory to allow for fast weight calculation.

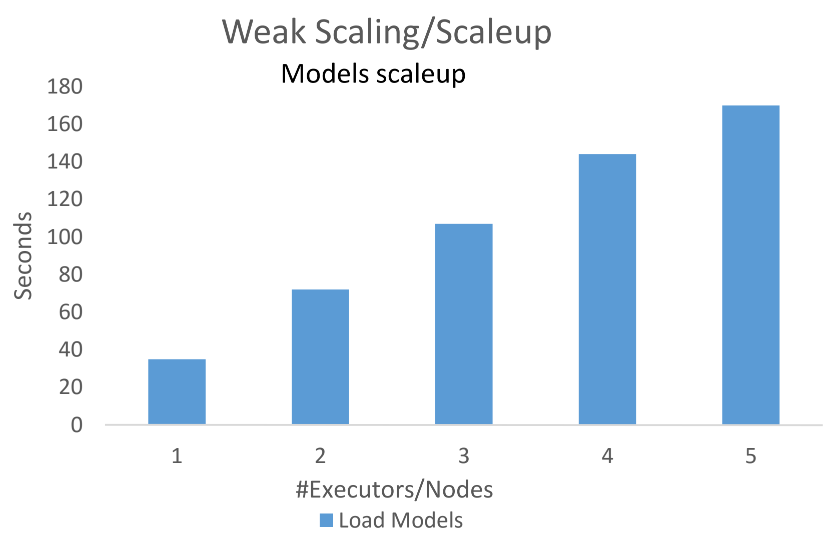


Figure 16 Scaleup Study of the Model Scaleup. Models were loaded in multiples of 5.

Table 3 Scaleup and Efficiency of the Model Scaleup study

#Executors	Load Time Scaleup	Load Time Efficiency
1	1	1
2	0,97	0,49
3	0,98	0,33
4	0,97	0,24
5	1,03	0,21

Figure 16 and Table 3 shows the results from the weak scaling study of the model load time. In this study the number of models were increased in multiples of 5 as nodes were added and the load time scaleup and efficiency of the ensemble classification application was measured. Since the data is not increased in volume in this study, the data load time scaleup is not measured in this step.

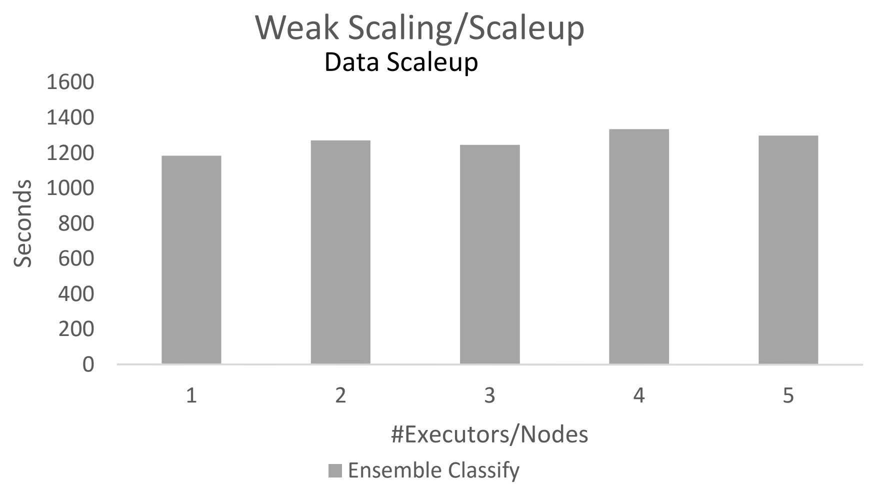


Figure 17 Scaleup Study of Data Scaleup. Data to be classified was loaded in multiples of 925 567 rows ((50.1 G snappy compressed) and classified by 20 models.

Table 4 Scaleup and Efficiency of the Data Scaleup study

#Executors	Ensemble Classify Scaleup	Ensemble Classify Efficiency
1	1	1
2	1,86	0,93
3	2,85	0,95
4	3,55	0,89
5	4,90	0,98

Figure 17 and Table 4 shows the results from the weak scaling study of the data load time. In this study the data volume was increased in multiples of 50.1 G as nodes were added, and the scaleup and scaleup efficiency was measured of the ensemble classification application.

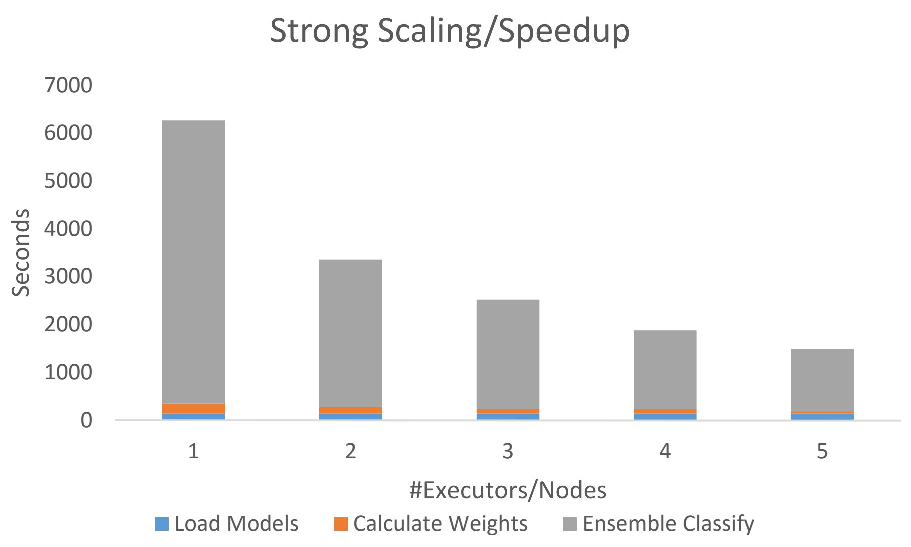


Figure 18 Speedup Study of the data and model speedup study. 20 models were loaded after which 230 000 rows (50.78 G snappy compressed) were used to calculate the voting weights of each classifier. Finally, an ensemble classification of the full test set was performed.

#Executors	Speedup Load time	Load Time Efficiency	Calculate Weights Speedup	Efficiency Calculate Weights	Speedup Ensemble Classify	Efficiency Ensemble Classify
1	1	1	1	1	1	1
2	0,99	0,50	1,55	0,77	1,92	0,96
3	1,01	0,34	2,16	0,72	2,59	0,86
4	0,98	0,245	2,33	0,58	3,59	0,90
5	1	0,20	4,14	0,83	4,56	0,91

Table 5 Speedup and Efficiency of the Data and Model speedup study

Finally, the speedup and speedup efficiency of the ensemble classification application was measured. 20 models were used to classify 50.78 G snappy compressed data (230 000 rows). The results of this speedup study is shown in Figure 18 and Table 5.

4.3 Random Grid Search

The random grid search resulted in 36 different models trained with varying parameters Appendix A. The search found that an under sampled SVM with all positive training instances yielded the highest auPRC. The hyper parameters of the model is shown in Table 6. The best model was evaluated on the test set and it performed as well as on the validation set, proving that no overfitting has occurred.

Table 6 Settings and resulting auPRC and auROC of the best model produced by the Random Search application.

Sample Technique	#iterations	Step size	RegParam	Mini-Batch	auPRC	auROC
Positives +Under sampled	600	650	1,00E-05	0,1	0,32	0,97

After the random grid search, two test runs were performed to determine the minimum number of iterations needed for the model to converge to the target auPRC (Figure 19).

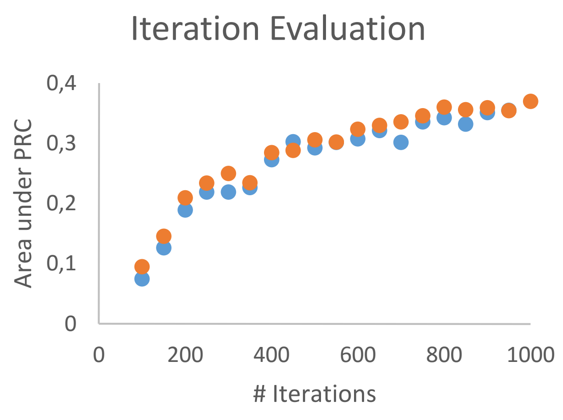


Figure 19 Resulting auPRC from increasing the number of iterations.

The test showed that the SGD converges slowly and that the increase in auPRC decreases significantly after 600 iterations.

Additionally, one model was trained with the same settings, but utilizing more of the clusters memory, allowing for more data to be cached. This allowed for more negative samples to be used, yielding a final class ratio of 1:3. This model yielded a auPRC of 0,34 on the test set, and confirms the information loss caused by the small total sample size of the 1:1 under sampled datasets. The final ensemble will as such be trained on a larger, 1:3 under sampled dataset, with the settings found by the random grid search.

4.4 Ensemble Training and Performance

The final ensemble was composed of 20 models, trained on approximately 445 000 instances each, including the 143 663 positive instances. The total amount of data used sums up to 8 904 778, or approximately 1/6 of the total training set size. Note however, that approximately 2 900 000 of these are the 143 663 positive instances, repeated in the training set of each model. Each of the 20 models took an average of 145 minutes[1] to train, resulting in a total of approximately 48 hours. Using the same cluster and same methodology, the entire training set would yield about 120 models, requiring 290 hours of training. Note however that the total volume of the training sets would be closer to 60 million instances, due to the minority oversampling.

When investigating the ensembles performance, an average majority vote was used to get average raw scores from the ensemble. These were then used to compute the average auPRC and auROC. The average was compared to the classifier that yielded the highest auPRC, hence forth called "the best" classifier (Table 7). The performance measures of all classifiers in the ensemble can be found in Appendix C

Table 7 Average auPRC and auROC of the ensemble compared to auPRC and auROC of the best classifier in the ensemble.

Average auPRC	Best auPRC	Average auROC	Best auROC
0,373	0,383	0,967	0,970

Finally, three ensemble Confusion matrix were constructed at the threshold which yielded the highest F-measure. The following voting schemes were applied; majority vote, and F-measure weighted majority vote. the confusion matrix of the best classifier, yielded at its best threshold, was constructed (Table 8). The confusion matrix of the best classifier was then used to evaluate the ensemble performance. The confusion matrices resulting from the different voting schemes are shown in Table 9-10.

[1] This average excludes models 10 and 18 runtime since the application failed during their training and had to restart.

Table 8 Confusion matrix of the Best Classifier

Actual Predicted	Positive	Negative
Positive	6935	12023
Negative	7614	4601268

Table 9 Confusion matrix of the Ensemble using F-measure weighted voting

Actual Predicted	Positive	Negative
Positive	7154	14476
Negative	7395	4598815

Table 10 Confusion matrix of the Ensemble using Majority voting

Actual Predicted	Positive	Negative
Positive	7176	14846
Negative	7373	4598445

From these confusion matrices, the precision and recall for the different voting schemes was calculated according to EQ. 7-8. These are shown in Table 11.

Table 11 precision and Recall of The best classifier and the Ensemble for different voting schemes.

Best Classifier Precision	Best Classifier Recall	Ensemble FM Precision	Ensemble FM Recall	Ensemble MV Precision	Ensemble MV Recall
0,37	0,48	0,33	0,49	0,33	0,49

4.5 Throughput

To measure the throughput of the training application the same methodology used by Agarwal et al. (2014) will be used for comparative purposes. The feature throughput of a training application is calculated according to:

$$\frac{\#NonZeroFeatures * \#iterations * \#rows}{\#Computational\ Nodes} * \frac{1}{Seconds} = \frac{\frac{Features}{Node}}{Second} \quad EQ.10$$

Using (Agarwal et al. 2014) estimate of 3300 non zero features per instance in the Splice Site dataset, and using a mini-batch fraction of 1, ensuring that full passes of the data are made, this yields a feature throughput when training a model of:

$$\frac{3300 * 600 * 443886}{4} * \frac{1}{9120} = \frac{24 * 10^6\ features}{Node} / Second$$

5. Discussion

5.1 Scalability

This section of the discussion will cover the scalability of the applications developed during this thesis.

5.1.1 Training and Data Loading

When examining both the strong and scaleup study the deteriorating iteration efficiency of both the scaleup and speedup stand out (Table 2). This could have been caused by the increase in network I/O required when adding nodes. When investigating the iteration execution time further, it was discovered that the time needed for network shuffling was not affected during the speedup study, while the computation step of the iteration decreased close to linearly. This would have been all and well until the computations are reduced to their non-parallelizable fraction, at which point adding more nodes would not be beneficial for the execution time. This is most likely also the case in the speedup study of load time, and iteration time (Figure 15, Table2). One possible solution would be to increase the available RAM of each worker node. Increasing the available RAM would allow each node to perform more computations, while still only shuffling the same amount of data to the driver. This would shift the time consumption towards the computational stage, and would make the time for shuffling less noticeable as a whole, and thereby improving the scaleup and speedup efficiency until the application becomes CPU bound. The efficiency of the scaleup study confirms that the non-parallelizable fraction is less noticeable as the dataset grow. It is however still noticeable.

When considering the load time speedup and scaleup it is clear that it is not quite ideal. This is most likely due to the executors were not being able to access the data locally but had to load it from another node via network. One explanation for this is the limited data locality caused by the 4 executor configuration (Figure 14-15, Table 1-2). A 5 executor configuration could mend the low load time scaleup and speedup efficiency, it would however have a negative impact on the Iteration scaleup and speedup, due to the extra networking required. Since the majority of the time of the application is spent in the training stage, detrimental effects to the load time is preferred over detrimental effects to the training time. Further, the decrease in cacheable memory of a 5 executor configuration would, as previously stated, yield a smaller dataset for training since both the driver and an executor would have to fit on the same node. When investigating the results of the speedup study of the iteration time (Figure 14, Table 2), it Is clear that distributing data that could have been processed in fewer nodes does not improve the execution time. This indicates that the improved data locality of a 5 executor configuration would not have reduced execution times.

5.1.2 Classification

Both the speedup and scaleup studies of the ensemble classification application showed that classification scales remarkably well, both when increasing the number of models, and when increasing the volume of the training data (Figure 17-18, Table 4-5). The studies also show that increasing the number of models in the ensemble does not increase the classification time linearly as one would have expected. This means that the extra time required for ensemble classification, compared to single model classification is almost negligible. Note however the speedup and scaleup of the model loading stage. This is a perfect example of a non-parallelizable step, which is caused by the fact that the models are loaded by the driver alone, and not parallelizable using Sparks native functions. This could become an issue if the number of models increase greatly, causing the model load time to be the majority part of the ensemble classification application. If the application could be extended to load the models in parallel, ensemble classification would truly be a viable option to single model classification. If parallelization of the model loading is not possible, an increase in RAM would allow for more data to be cached, which in turn would mean less models would be trained. This is however not a long term solution. Parallelization of the model loading would be the preferable option.

The speedup study further shows that the applications speedup is close to ideal in all regards of ensemble classification. These results prove that ensemble classification on big data using Spark is as viable as single model classification when considering execution time.

5.2 Algorithm and Cluster Tuning

The hyper parameter tuning found parameters that, when used, yielded individual models with comparable auPRC of those previously trained on equally sized subsets of the data (Sonnenburg and Franc 2010, Table 2). However, it is important to remember that the global optima is not guaranteed to be within the parameter ranges used by the random grid search. Additionally, the pools were not composed of a continuous value ranges, which also makes it possible for the random search to miss the global optima. In real applications with more available resources, the random search should have trained more models, over larger, continuous value ranges, resulting in a greater coverage of the hyper parameter space. However, despite the limitations in this thesis, the random grid search application does work as a proof of concept of scalable big data hyper parameter tuning using Spark.

In addition to this, the clusters YARN and Spark configurations used might not have been the most optimal for the applications at hand. Additional YARN and Spark tuning can provide a larger storage fraction, a higher disk throughput, and reduce network I/O, which in turn could result in faster, more scalable applications as well as more accurate classifiers.

5.3 Ensemble performance

The auPRC of the best classifier in the ensemble was just short of what was previously achieved by (Sonnenburg and Franc 2010)) where 10^5 rows generated an auPRC of approximately 0,31 using the same kernel, and 10^6 achieved an auPRC of approximately 0,46. The ensemble did however fail to achieve similar performance in regards to precision and recall considering the amount of data used regardless of the voting scheme being used in this thesis (Sonnenburg and Franc 2010, Table 2). This is most likely due to the lack of variety in the ensemble since each classifier in the ensemble was trained on the same positive instances. This indicates that an addition in classifiers to the ensemble would not have increased the overall performance, and that training a large ensemble on the entire dataset would not have yielded comparable results to (Agarwal et al. 2014) and (Sonnenburg and Franc 2010) who achieved 0.5857 and 0.5778 auPRC respectively using the same kernel used in this thesis. Another possible reason which could explain the poor ensemble performance could of course be ill tuned hyper parameter in the training application.

The ensemble of this thesis did not outperform the best classifier in the ensemble (Table 5- 9). One explanation for this, could be that since each classifier was trained on the same positive instances, there is not enough variety in the training sets to generate a diverse ensemble of classifiers. One possible option to mend this could be to try to synthetically generate positive instances, similar to the ones in the dataset. One example would be Synthetic Minority Over-Sampling Technique (SMOTE). SMOTE is a method for generating synthetic instances from the original data. SMOTE has shown to help mend data imbalance and generate more accurate classifiers in previous studies (Chawla et al. 2002; Blagus and Lusa 2013; García et al. 2012).

Another approach would be to train classifiers on the original distribution for an Ensemble. It has been proven that accurate classifiers can be trained on the Splice Site dataset without under- or oversampling (34,35). This would generate classifiers of greater variety considering the minority class, which could in turn produce more accurate ensemble. It did however prove difficult during this thesis to find suitable parameters for the original data distribution, which was why an under sampling of the majority class was preferred.

5.4 Throughput

The most important result in this thesis is the high feature throughput in Spark. With an average feature throughput of 24 million features per node per second during training, Spark provides a speedup just short of 10 compared to Logistic Regression (LR) solutions, implemented within AllReduce, which reached a throughput of 2.6 million features per node per second[2] (Agarwal et al. 2014). This is a long way from the reported 100 speedup factor, but it does however prove that Spark can provide with significant speedups in iterative applications (Zaharia et al. 2010). None of the applications in this thesis showed any indications of being bottlenecked by the CPU, it therefore stands to

[2] Estimates were made using (Agarwal et al. 2014) reported 1920 second execution time on 500 nodes, 3300 non-zero features, 50 000 000 rows and 14 LR iterations on the Splice Site dataset.

reason, that given more RAM, more data could have been cached, and processed as fast, potentially yielding a higher feature throughput. As previously stated, the majority of the time taken for the training application was used for network shuffling between the driver and executors. This fraction was reduced by 66% by repartitioning the RDD to as many partitions as there were executor cores in the cluster[3].

Note that the SVM application in this thesis needed 600 iterations to converge (Figure 19). Compare this to 14 LR iterations needed to converge in a previous study (Agarwal et al. 2014). This does, by no means, indicate that all SVM with SGD applications will converge this slowly. One could surely improve the algorithm by further tweaking the settings, which could yield faster convergence. Further, the number of iterations does not affect the feature throughput, since fewer iterations would also yield a shorter execution time. Additionally, the fact that the training time of only one model was used to estimate feature throughput does not affect the throughput, since an increase in the number of models would increase the number of rows passed as well as the time required linearly with the increase in models.

These results prove that Spark provides a high feature throughput on cached data, and it stands to reason, that models trained in Spark should be able to reach as good accuracy as algorithms trained in any other framework in as few iterations, provided that the hyper parameters are tuned equally well.

6. Conclusion

In this thesis I have implemented three core applications of Ensemble Machine Learning, Hyper parameter tuning, ensemble training and ensemble classification in Apache Spark on YARN. Although the implementations could all be developed and tuned further to provide more accurate classifiers and more scalable applications, these results do however provide some insight into the possibility and feasibility of scalable big data ensemble ML using Apache Spark.

[3] The Repartitioning was done using Coalesce with shuffle =false to reduce network I/O.

During the work on this thesis, I was not able to implement an Ensemble that could justify the increased training time generated by adding to the performance of the ensemble. This result should not be viewed as a general result regarding ensembles, and should not be interpreted as general limitations in ensemble techniques, only that, in this particular case, an increase in performance was not achieved by training an ensemble. The most important thing to remember, is that the model parameters were only estimates of the optimal values. The resulting accuracy of the best classifier does by no means imply any limitations in Sparks ability to train accurate models.

Despite the poor performance of the ensemble, the primary goal of assessing Apache Sparks usability in big data ML was achieved. Apache Spark proved to be a great tool for big data ML, providing a high feature throughput up to (but not necessarily limited to) 10 times larger than previous methods based on AllReduce on Hadoop (Agarwal et al. 2014). Apache Spark MLlib is not limited to SVMs, but also support Linear Regression, Naïve Bayes, Random Forest and more algorithms are bound to be implemented in future releases. The high feature throughput provided by Sparks is bound to favor any iterative ML application.

Omicron Cetis cluster is well suited for development, however, when testing scalability, the number of nodes does not provide enough data points in a scaling study for the results to be very reliable. As for production purposes, the cluster is not well suited for Spark applications that require data to be cached, due to the nodes limited amount of RAM. Most of the 12 G available RAM on each node was used by other fractions than the storage fraction, such as shuffle, unroll, safety, and heap fractions, leaving only about 4.5G for caching on each node. A larger storage fraction resulted in the applications running out of memory and failing. Additionally, during the speedup studies of the training application it was discovered that the application worked at or near the non-parallelizable fraction, providing no speedup when increasing the number of computational nodes.

For these reasons I do not recommend further development of the cluster by adding nodes with similar configuration for any other reason other than to improve the development environment. A more suitable approach for Spark applications would be increasing each nodes available RAM and adding more disks. The extra RAM would allow each node to cache more data and to performed more computations, while not increasing the network I/O and allowing for more accurate models to be trained. More disks would allow for more parallelizable disk I/O, speeding all applications that read from HDFS. Another viable alternative is to investigate the possibility and viability of cloud computing service, such as Amazon Work Spaces or similar services.

I believe that the high feature throughput provided by Apache Spark makes it a very powerful tool for big data ML and that more knowledge in the area of Big data ML using Apache spark will be a valuable resource in the future.

9. References

IBM Big Data & Analytics Hub. 2015. “2015 Predictions and Trends for Big Data and Analytics | The Big Data Hub.” *IBM Big Data & Analytics Hub*. January 26. Accessed September 30 2015. http://www.ibmbigdatahub.com/video/2015-predictions-and-trends-big-data-and-analytics.

Agarwal, Alekh, Olivier Chapelle, Miroslav Dudík, and John Langford. 2014. “A Reliable Effective Terascale Linear Learning System.” *Journal of Machine Learning Reaserch* 15: 1111–33.

Alan Kaminsky. 2015. *BIG CPU, BIG DATA: Solving the World’s Toughest Computational Problems with Parallel Computing*. Accessed June 30 2015. http://www.cs.rit.edu/~ark/bcbd/.

Amdahl, Gene M. 2007. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, ” *IEEE Solid-State Circuits Newsletter* 12: 19–20.

Blagus, Rok, and Lara Lusa. 2013. “SMOTE for High-Dimensional Class-Imbalanced Data.” *BMC Bioinformatics* 14: 106.

Chawla, Nitesh V., Kevin W. Bowyer, Lawrence O Hall, and W. Philip Kegelmeyer. 2002. “SMOTE: Synthetic Minority Over-Sampling Technique.” *Journal of Artificial Intelligence Research* 16: 321–57.

Chawla, Nitesh V, Thomas E Moore, Lawrence O Hall, Kevin W Bowyer, W. Philip Kegelmeyer, and Clayton Springer. 2003. “Distributed Learning with Bagging-like Performance.” *Pattern Recognition Letters* 24: 455–71.

Chawla, Nitesh V., and Jared Sylvester. 2007. “Exploiting Diversity in Ensembles: Improving the Performance on Unbalanced Datasets.” In *Multiple Classifier Systems*, edited by Michal Haindl, Josef Kittler, and Fabio Roli, 397–406. Lecture Notes in Computer Science 4472. Springer Berlin Heidelberg.

Chinchor, Nancy. 1992. “MUC-4 Evaluation Metrics.” In *Proceedings of the 4th Conference on Message Understanding*, 22–29. MUC4 ’92. Stroudsburg, PA, USA: Association for Computational Linguistics.

Davis, Jesse, and Mark Goadrich. 2006. “The Relationship Between Precision-Recall and ROC Curves.” In *Proceedings of the 23rd International Conference on Machine Learning*, 233–40. ICML ’06. New York, NY, USA: ACM.

García, Vincente, José S. Sánchez, Raúl Martín-Félez, and Ramón A. Mollineda. 2012. “Surrounding Neighborhood-Based SMOTE for Learning from Imbalanced Data Sets.” *Progress in Artificial Intelligence* 1: 347–62.

Guo, Zhenhua, Geoffrey Fox, and Mo Zhou. 2012. “Investigation of Data Locality in MapReduce.” In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 419–26.

He, Haibo, and Edwardo A. Garcia. 2009. “Learning from Imbalanced Data.” *IEEE Transactions on Knowledge and Data Engineering* 21: 1263–84.

James Bergstra, and Yoshua Bengio. 2012. “Random Search for Hyper Parameter Optimization.” *Journal of Machine Learning Reaserch* 13: 281–305.

Kambatla, Karthik, Giorgos Kollias, Vipin Kumar, and Ananth Grama. 2014. “Trends in Big Data Analytics.” *Journal of Parallel and Distributed Computing*, Special Issue on Perspectives on Parallel and Distributed Processing, 74: 2561–73.

King, Michael A., Alan S. Abrahams, and Cliff T. Ragsdale. 2015. “Ensemble Learning Methods for Pay-per-Click Campaign Management.” *Expert Systems with Applications* 42: 4818–29.

Kohavi, Ron, and Provost Foster. 1998. “Glossary of Terms.” *Journal of Machine Learning Reaserch* 30: 271–74.

Laney, Douglas. 2015. “Gartner Predicts Three Big Data Trends for Business Intelligence.” *Forbes*. February 12. Accessed September 9 2015. http://www.forbes.com/sites/gartnergroup/2015/02/12/gartner-predicts-three-big-data-trends-for-business-intelligence/.

———. 2001. “3D Data Management: Controlling Data Volume, Velocity, and Variety.” *Application Delivery Strategies by META Group Inc.*, February 6. Accessed November 23 2015. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

“Linear Methods - MLlib - Spark 1.5.1 Documentation.” 2015. Accessed October 20 2015. http://spark.apache.org/docs/latest/mllib-linear-methods.html.

Liu, Xu-Ying, Jianxin Wu, and Zhi-Hua Zhou. 2006. “Exploratory Under-Sampling for Class-Imbalance Learning.” In *Sixth International Conference on Data Mining, 2006. ICDM ’06*, 965–69.

“M3AAWG Email Metrics Report 16.” 2014. November 14. Accessed December 27 2015. https://www.m3aawg.org/for-the-industry/email-metrics-report.

“MLlib | Apache Spark.” 2015. Accessed November 23 2015. http://spark.apache.org/mllib/.

“Optimization - MLlib - Spark 1.5.1 Documentation.” 2015. Accessed October 20 2015. http://spark.apache.org/docs/latest/mllib-optimization.html.

Polikar, Robi. 2006. “Ensemble Based Systems in Decision Making.” *IEEE Circuits and Systems Magazine* 6: 21–45.

Shukla, Soumya, Vaishnavi Kukade, and Sofiya Mujawar. 2015. “Big Data: Concept, Handling and Challenges: An Overview.” *International Journal of Computer Applications* 114: 6–9.

Sonneburg, Sören, and Vojtech Franc. 2010. “COFFIN : A Computational Framework for Linear SVMs.” In *International Conference on Machine Learning*.

Sonneburg, Sören, Gunnar Rätsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtěch Franc. 2010. “The SHOGUN Machine Learning Toolbox.” *Journal of Machine Learning Reaserch* 11: 1799–1802.

Sonnenburg, Sören, Gunnar Rätsch, and Konrad Rieck. 2007. “Large-Scale Learning with String Kernels.” In *Large-Scale Kernel Machines*, 73–103. MIT Press.

“Spark Programming Guide - Spark 1.5.1 Documentation.” 2015. Accessed September 30 2015. http://spark.apache.org/docs/latest/programming-guide.html.

Swoyer, Stephen. 2012. “Big Data -- Why the 3Vs Just Don’t Make Sense.” *Tdwi*. July 24. Accessed November 27 2015. https://tdwi.org/articles/2012/07/24/big-data-4th-v.aspx.

The 1000 Genomes Project Consortium. 2010. “A Map of Human Genome Variation from Population-Scale Sequencing.” *Nature* 467: 1061–73.

Töscher, Andreas, Michael Jahrer, and Robert Legenstein. 2008. “Improved Neighborhood-Based Algorithms for Large-Scale Recommender Systems.” In *Proceedings of the 2Nd KDD Workshop on Large-Scale Recommender Systems and the Netflix Prize Competition*, 1–6. NETFLIX ’08. New York, NY, USA: ACM.

“Uppsala University Library.” 2015. Accessed December 27 2015. http://ub.uu.se/?languageId=1.

van Rijmenam, Mark. 2015. “Datafloq - Why The 3V’s Are Not Sufficient To Describe Big Data.” *Datafloq*. August 7. Accessed November 23 2015. https://datafloq.com/read/3vs-sufficient-describe-big-data/166.

“Welcome to Apache^{TM} Hadoop®!” 2015. Accessed June 25 2015. https://hadoop.apache.org/.

White, Tom. 2012. *Hadoop: The Definitive Guide*. 3rd ed. Beijing: O’Reilly.

Witten, Ian H., Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. 3. ed. Morgan Kaufmann Series in Data Management Systems. Burlington, MA: Morgan Kaufmann.

Xie, Hua-Lin, Liang Fu, and Xi-Du Nie. 2013. “Using Ensemble SVM to Identify Human GPCRs N-Linked Glycosylation Sites Based on the General Form of Chou’s PseAAC.” *Protein Engineering Design and Selection* 26: 735–42.

Yang, Pengyi, Yee Hwa Yang, Bing B. Zhou, and Albert Y. Zomaya. 2010. “A Review of Ensemble Methods in Bioinformatics.” *Current Bioinformatics* 5: 296–308.

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing.” In *Networked Systems Design and Implementations*.

Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. “Spark: Cluster Computing with Working Sets.” In *Hot Topics in Cloud Computing*.

Appendices

Appendix A – Random Grid Search Results

Sample Technique	#iterations	Stepsize	RegParam	Mini-Batch	auPRC	auROC
Positives +Under sampled	600	650	1,00E-05	0,1	0,321056538	0,973356928
Positives +Under sampled	500	450	1,00E-09	0,1	0,26365499	0,97181113
Positives +Under sampled	300	400	1,00E-07	1	0,231289464	0,966628833
Positives +Under sampled	200	400	1,00E-08	0,9	0,169685159	0,957664738
Positives +Under sampled	300	100	1,00E-04	1	0,160607955	0,957965781
Positives +Under sampled	200	200	1,00E-05	0,4	0,154464121	0,954943403
Positives +Under sampled	500	700	1,00E-04	0,4	0,114417866	0,949408405
Positives +Under sampled	200	400	1,00E-04	0,8	0,097897652	0,944058368
Positives +Under sampled	100	400	1,00E-05	0,3	0,094515714	0,939041335
Positives +Under sampled	100	400	1,00E-07	0,3	0,09225479	0,939012977
Positives +Under sampled	100	800	1,00E-05	0,7	0,084750434	0,938393925
Positives +Under sampled	100	500	1,00E-09	0,7	0,083212081	0,936284916
Positives +Under sampled	100	500	1,00E-09	0,7	0,083212081	0,936284916
Positives +Under sampled	100	350	1,00E-08	0,8	0,082614683	0,936509396
Positives +Under sampled	100	100	1,00E-07	0,7	0,06430015	0,93265768
Positives +Under sampled	50	500	1,00E-04	0,6	0,036503812	0,910225507
Positives +Under sampled	50	550	1,00E-06	0,4	0,035961005	0,911306594
Positives +Under sampled	50	400	1,00E-07	0,2	0,035461203	0,910776222
Positives +Under sampled	50	250	1,00E-09	0,1	0,03456653	0,910411966
Positives +Under sampled	20	200	1,00E-04	0,4	0,025203026	0,894528438
Positives +Under sampled	20	200	1,00E-04	0,4	0,025203026	0,894528438
Positives +Under sampled	20	100	1,00E-06	0,9	0,024887277	0,894353622
Original	500	450	1,00E-07	1	0,002304333	0,474062057
Original	500	350	1,00E-09	0,9	0,002303352	0,473888161
Original	400	450	1,00E-05	0,1	0,002290243	0,471441394
Original	200	250	1,00E-04	0,8	0,00226997	0,467686471
Original	300	150	1,00E-09	0,9	0,002245392	0,462980143
Original	300	450	1,00E-06	0,1	0,002242017	0,462296691
Original	200	450	1,00E-09	0,6	0,002214093	0,456838389
Original	200	200	1,00E-08	0,9	0,002212619	0,456524991
Original	200	450	1,00E-07	0,1	0,002208304	0,455635985
Original	100	400	1,00E-06	0,4	0,00217578	0,449014664
Original	100	250	1,00E-09	0,6	0,002174899	0,448844765
Original	50	50	1,00E-04	0,6	0,002151247	0,443861267
Original	50	150	1,00E-06	0,3	0,002149625	0,443534403
Original	50	100	1,00E-08	0,1	0,00214473	0,442433308
Original	20	550	1,00E-04	0,4	0,00213494	0,440341266
Original	20	500	1,00E-04	0,4	0,002134337	0,440210064

Appendix B- Spark configuration and Cluster Hardware

Setting	Value
spark.history.kerberos.keytab	none
spark.history.kerberos.principal	none
spark.history.provider	org.apache.spark.deploy.yarn.history. YarnHistoryProvider
spark.history.ui.port	18080
spark.driver.extraJavaOptions	-Dhdp.version=-2557 -Dspark.akka.frameSize=128
spark.yarn.am.extraJavaOptions	-Dhdp.version=2.3.0.0-2557
spark.yarn.applicationMaster.waitTimes	10
spark.yarn.containerLauncherMaxThreads	25
spark.yarn.driver.memoryOverhead	1024
spark.yarn.executor.memoryOverhead	1024
spark.yarn.historyServer.address	agena.omicron.se:18080
spark.yarn.max.executor.failures	10
spark.yarn.preserve.staging.files	false
spark.yarn.queue	default
spark.yarn.scheduler.heartbeat.interval-ms	5000
spark.yarn.services	org.apache.spark.deploy.yarn.history. YarnHistoryService
spark.yarn.submit.file.replication	3
spark.scheduler.maxRegisteredResourcesWaitingTime	1200
spark.scheduler.minRegisteredResourcesRatio	1
spark.shuffle.manager	hash
spark.shuffle.compress	true
spark.shuffle.consolidateFiles	true
spark.shuffle.spill	true
spark.shuffle.blockTransferService	nio
spark.driver.maxResultSize	2g
spark.shuffle.safetyFraction	0.9
spark.shuffle.memoryFraction	0.2
spark.storage.memoryFraction	0.6
spark.storage.safetyFraction	0.9
spark.rdd.compress	false
spark.serializer	org.apache.spark.serializer.KryoSerializer
spark.kryoserializer.buffer.max	512m

Application Specific settings:

Training

--num-executors 4 –executor-memory 11g –driver-memory 4g –executor-cores 5 –driver-cores 5 –master yarn-cluster

Classification

--num-executors 5 –executor-memory 6g –driver-memory 4g –executor-cores 5 –driver-cores 5 –master yarn-cluster

Random Search

--num-executors 2 –executor-memory 11g –driver-memory 4g –executor-cores 5 –driver-cores 5 –master yarn-cluster

Slave Hardware:

2x 8GB 1600MHz DDR3

Intel i5-4460 3.20 GHz

1GBit Ethernet port

1TB 7200 RPM HDD

Appendix C – Individual model Performance

Model	auPRC	auROC	#Rows used	Time Required for training in minutes
Model0	0.3473365043	0.9700500389	445758	138
Model1	0.3542048162	0.9703641169	443944	143
Model2	0.3731360307	0.9700342076	447852	144
Model3	0.3771984429	0.9702648524	444250	142
Model4	0.3769994855	0.9704187144	443254	140
Model5	0.3779121113	0.9702318718	446512	137
Model6	0.3497192412	0.9700675228	445231	144
Model7	0.3360254382	0.9697897239	444025	145
Model8	0.3823664078	0.9705788686	446158	149
Model9	0.3532737861	0.9703011754	443958	139
Model10	0.3292577368	0.9698577529	445523	234
Model11	0.3461104473	0.970192659	444522	157
Model12	0.3184986596	0.9695488757	445548	151
Model13	0.3645723928	0.97022999	444698	146
Model14	0.3541102443	0.9703264079	447662	146
Model15	0.3645911411	0.9699976746	444927	146
Model16	0.3833391822	0.9704324432	445722	146
Model17	0.352032043	0.9698979545	444476	161
Model18	0.3798485499	0.9702591117	445215	267
Model19	0.3738273688	0.9703964428	445543	142