Big Data Major Research Paper

Exploring the relationship between runtime, machine learning evaluation metrics, and the number of CPUs on AWS

Mirnes Salkic
November 30, 2018
Applied Modelling and Quantitative Methods: Big Data Analytics
Prepared for: Dr. Sabine McConnell

Table of Contents

1. Problem Statement

The purpose of the project is to explore the relationship between runtime, machine learning evaluation metrics, and the number of CPUs using three different datasets with pypark on Amazon Web Services (AWS). The project can be viewed as an experiment in which the following hypothesis would be tested:

1. As the number of nodes[1] increases the runtime decreases
2. As the number of CPUs increases the accuracy for a balanced dataset decreases
3. Following the logic from 1. and 2. there is a tradeoff between runtime and accuracy for balanced datasets
4. The hypothesis in 1. and 2. hold true for various algorithms

In simple terms, the goal is to assess the gains and loses as the data gets more and more distributed. Thus, the idea is to extract a model from data that is physically distributed i.e. where the data cannot travel from one processor to another.

2. Background research

In classical machine learning the data is processed using a single-thread procedure on a single machine. However, as larger datasets became more and more common and as the complexity of models grew it became extremely time-consuming to process the data on a single machine. The decrease in the cost of processing and storage and the ever increasing sizes of the available datasets have lead to an unprecedented demand for distributed computing solutions. Distributing computing makes it possible to partition and distribute large datasets, which cannot fit into a single computer's memory, across multiple machines. This way, distributed machine learning algorithms can extract models from data that are spread across many machines in a reasonable amount of time.

There are two fundamental tradeoffs in distributed computing, communication versus computation and communication versus accuracy (Zhang, 3-4). The first tradeoff can be understood with a simple example of running a parallel algorithm. When running a parallel stochastic gradient descent algorithm local (partial) gradients are computed at each of the nodes. The calculation of the global gradient (average gradient) requires aggregating the information from all of the nodes; it requires synchronization. Hence, at each iteration the partial gradients are aggregated at the master node and then the updated parameter is broadcasted from the master node to all the worker nodes. This illustrates that with more machines (i.e. nodes) more communication is required while at the same time less time will be spent on computation (Zhang, 4). For the second tradeoff, achieving a higher accuracy on a trained machine learning model generally means more frequent or more efficient communication between the worker nodes and the master node. If the communication occurs more frequently and in larger volumes, then communication becomes a bottleneck (Zhang, 5).

While the two tradeoffs are important they are not the primary concern of this paper. We are rather more interested in the tradeoff between the runtime of machine learning algorithms and the accuracy obtained from a dataset as we add more and more nodes to a cluster. The following section briefly reviews the literature on the relationship between

[1] A node is an EC2 instance.

runtime of an algorithm and distributed computing as well as the relationship between accuracy and distributed computing.

In their paper titled “Performance evaluation of big data frameworks for large-scale data analytics”, Veiga et al., compare MapReduce, Spark, and Flink as computation frameworks in terms of runtime for different tasks including one machine learning algorithm – K-means. For eight iterations of the k-means clustering algorithm of a 26GB dataset Spark achieves the lowest execution time followed by Flink and MapReduce. The experiment for Spark and other platforms was done using 13, 25, 37, and 49 nodes and it was shown that with the increase in cluster size the execution time for the K-means algorithm decreased (Veiga et al.). Identical conclusions were reached by Boden et al., who tested the k-means algorithm on a 200GB dataset with 100 dimensions and used. In addition, Boden et al., have run experiments using a Logistic Regression algorithm which yielded the exact same results. The maximum number of nodes used in the experiment was 25 and it could easily be seen that the runtime of the algorithm decreases by a very small margin as the number of nodes increases from 20 to 25 as opposed to the sharp decline when the number of nodes increases from 3 to 5 or 5 to 10. Unfortunately, we do not learn what happens to the runtime of the algorithms as they scale beyond 25 nodes (Boden et al., b)

In “Learning Rules from Distributed Data” Hall et al. explore the accuracy as a function of parallelism for a rule-based classifier. In distributed rule-based learning, the data is divided among N nodes where N sets of rules get generated. After the sets of rules are generated they are merged to make up a single rule set. Each rule that is created is assessed for its ‘goodness’ which is based on an accuracy measure as well as on the number and type of examples the rule covers. The paper through a simple example show that in an extreme situation, a merged rule set - trained in a distributed manner, may contain no rules in common with the rules obtained from training on the full training dataset. The authors claim that the final merged rule set depends on how the training data is partitioned. This simple proof of concept example indicates that the accuracy of a model extracted from distributed data may be different from the accuracy obtained from a model that is trained sequentially on the entire training set (Hall et al, 213). In the actual experiment, the authors used the IRIS dataset which only has 150 examples and 3 classes and the PIMA Indian diabetes data set that is comprised of 768 examples from 2 classes. The goal of the experiment was to partition these already small datasets into an increasing number of subsets and then generate rules from decision trees in parallel and finally merge them into a final set of rules. The experiments were done using 10 fold cross-validation. The authors also stressed that for a rule to be acceptable its accuracy needs to be grater than a pre-specified threshold. The threshold was chosen to be 51 which is slightly better than random guessing, and for the Iris dataset the threshold was chosen to be 75 arbitrarily. The performance on the Pima data set is more interesting and insightful as the experiments were run on N=2, N=4, …N=10 partitions while the Iris data set was only run on N=2 partitions. The accuracy of the overall rule set varied with the increase in the number of partitions – both increases and decreases in accuracy are observed. However, the accuracy showed a general trend of decrease in accuracy when the number of partitions increased. The authors believe that the trend of accuracy falling off can also occur with larger datasets (Hall et al., 216).

In contrast to Hall et al. findings, the paper by Provost and Hennessy titled “Scaling up: Distributed Machine Learning with Cooperation” uses a similar approach as the one previously described but makes use of a larger dataset (1,000,000 examples) and finds that there was no drop off in accuracy for partitions up to N=4. The explanation offered was as follows:

"Because of the distribution of examples across the subsets of the partition, some processors found rules that had fallen off the beam in the monolithic search. Thus, the distributed version actually learned *more* satisfactory rules than the monolithic version in addition to learning substantially faster" (Provost, 77).

In short, we note that the accuracy does not necessarily drop as the data gets partitioned over increasing number of nodes. There might be even a slight increase in accuracy as the data gets distributed, but overall the general trend should show a decrease in accuracy as the number of nodes increases at least for rule-based classifiers.

In the paper "Distributed Machine Learning – but at what COST?" Boden et al. argue that systems such as Spark while providing robust scalability take a substantial amount of additional resources to reach the performance of a single machine implementation. Their setup involved two different types of nodes: a small and a big node. The authors define a small node as a computer with 4 cores and 16GB of RAM and a big node as a machine with 24 cores and 256GB of RAM (Boden et al., a, 4). They have run two experiments in which they trained a logistic regression and gradient boosted trees model. I will only present their findings for the logistic regression model. The goal was to identify what cluster configurations and size are required for Spark to achieve identical performance [in terms of runtime and AuC] as a single small and large node trained using Vowpal Wabbit (VW). Vowpal Wabbit is an out-of-core machine learning library originally developed by Yahoo Research and currently maintained by Microsoft Research. When running a logistic regression on 6 small nodes Spark MLlib is slower than VW on single small node. However, running a logistic regression on 15 small nodes Spark gets close to VW's performance (both in runtime and AuC) on a single small node. For the runs on big nodes it takes 6 nodes to get identical performance of VW which used one node and two cores. Thus, the authors conclude that even modern tools such as Spark (v.2.2.0) require substantial hardware resources in order to obtain similar prediction quality and runtime as a powerful single machine (Boden et al., a, 5). This raises an interesting question as to when and under what circumstances is it worth using a distributed platform and tools. The paper warns us that opting for Spark or a similar big data processing engine may not always be the wisest choice in terms of monetary expenses.

3. **Data set -** American Community Survey

The Public Use Microdata Sample (PUMPS) is a sample data from the American Community Survey (ACS) which was gathered over a 5 – year period between 2012 and 2016. The entire data set contains approximately 5% of the population of the United States. The survey produced two distinct data sets, one containing population and the other household records. The household data set has been selected for this project and the data has been obtained from American FactFinder[2].

[2] https://factfinder.census.gov/faces/tableservices/jsf/pages/productview.xhtml?pid=ACS_pums_csv_2012_2016&prodType=document

The data set is composed of four distinct files which collectively take about 5.82GB of storage space on disk. There are 7,439,832 records and 208 variables.

The variables include different characteristics of a household such as: number of bedrooms, housing costs, property taxes, residence state, running water, vehicles available, lot size, number of rooms, year the structure was built among others. This data set could be used for regression analysis where one could predict the property value. However, the predictor variable will be transformed into a categorical binary variable which will have two levels indicating whether a particular housing record is above or below the median property value in the United States. Thus, the American Community Survey Household data set could be transformed into a balanced data set where accuracy as an evaluation metric could be used.

4. Platform and Tools

The cloud computing platform on which the project is implemented is Amazon Web Services (AWS). While AWS provides different instance types, given the project demands, the general purpose instance type will be sufficient to obtain the desired results. General purpose instances are suitable for many applications as they provide a balance of compute, memory, and network resources. In particular, the master node of each cluster will be a m4.large instance which has the following characteristics: 2 vCPUs, 8 GiB memory, moderate network performance. On the other hand, each worker node will be an a m4.xlarge instance that has 4 vCPUs, 16 GiB of memory, and a high network performance. Both the master and worker nodes run on a 2.4 GHz Intel Xeon E5-2676 v3 Processor.[3]

There will be a total of four Amazon Elastic MapReduce (EMR) clusters created. The first cluster will have two, the second four, the third six, and the fourth eight worker nodes. All clusters will run the emr – 5.19.0 version with Spark 2.3.2, Hadoop 2.8.5, Zeppelin 0.8.0, and Ganglia 3.7.2 installed.

Ganglia is an open source project and is a scalable and distributed system that allows us to monitor the cluster as a whole as well as on each individual node. For instance, we will be able to see load, memory usage, CPU utilization, and the network traffic (Amazon EMR Release Guide). The Zeppelin notebook is similar to Jupyter Notebook but it has integrated different big data back ends such as Spark, SQL, Hive, Scala, Hadoop, etc. It also has integrated Matplotlib with Python and PySpark interpreter. It is a multipurpose notebook for data discovery, data analytics, and it has visualization capabilities built into it. The notebook also supports many languages or data processing backend including Cassandra, R, Hive, Flink, Elasticsearch, Python, HDFS, among others. Perhaps, the key advantage over the Jupyter Notebook is that it is also has integrated Spark in it (Apache Zeppelin).

The data will be stored in Amazon S3 buckets while the Amazon EMR cluster will be used to process the data. The reason the data will be stored in S3 is that it can be accessed by multiple EMR clusters. Thus, we avoid the need to re-upload the data multiple times.

[3] https://aws.amazon.com/ec2/instance-types

When creating an Amazon EMR cluster different software components will get installed on each node type. The master node manages the cluster and distributes the tasks among slave nodes (e.g. how the data will be distributed and processing tasks). In addition, the master node will monitor the status of the slaves as well as the overall health of the cluster. The slave nodes have their own software components that run assigned tasks and that ensure that the data is stored in HDFS on the cluster. There third type of nodes is the task node, which is optional. It is also a slave node that has software components but only runs tasks. It is not mandatory to have those.

Spark is a popular big data solution. Typically, one uses Spark when the data no longer can fit onto one's local machine (i.e. if the machine does not have enough RAM). Apache Spark began as a research project at the UC Berkeley AMPLab in 2009 was built with the aim of addressing some of the limitations of MapReduce. [4] In the following section we describe both MapReduce and Spark and contrast the two.

5. MapReduce vs. Spark

Hadoop MapReduce

With the ever-increasing amount of available information, their processing has become a significant challenge. MapReduce is a programming model that was developed by Google for processing large data sets.

Prior to MapReduce there were many special purpose solutions whose aim was to process large amounts of data. While the computations have been relatively simple, the large amount of input data required the computations to be split across thousands of machines to enable the processing in a reasonable amount of time. Consequently, the relatively simple computations were shadowed with complex issues related to parallelizing the computation, distributing the data, and handling machine failures. MapReduce was developed to address these complex issues through an abstraction, so that a programmer's task is to specify the simple computation while hiding the complexities related to parallelization, fault-tolerance, data distribution, and load balancing in a library (Dean et al., 1).

MapReduce is a batch-oriented processing engine where the data is processed offline and is persisted to the disk. The data is generally split into numerous smaller chunks and operations are performed on each of these chunks independently and in parallel. In contrast to the traditional data processing paradigm where the data was moved from the storage nodes to the processing nodes, which for large data sets could mean large overheads due to the movement of data, in MapReduce the data processing algorithm is moved to the nodes that store the data. Thus, MapReduce is able to perform computations locally where the data is stored and hence saves network bandwidth (Erl et al., 125-126).

MapReduce jobs involve two tasks: map and reduce. Additionally, each task is composed of multiple stages. A maps task consists of a map, combine (optional), and partition stage while a reduce tasks is comprised of a shuffle and sort and reduce stage (Erl et al., 126). On a high level, a computation accepts a set of input key/value pairs and produces a set of output key/value pairs. The map function takes input key/value pairs and produces

[4] https://spark.apache.org/history.html

intermediate key/value pairs which are further passed to a reduce function. The reduce function, then, merges values together and creates an output value (Dean et al., 2). To better understand the underlying process on how the MapReduce framework works it is worth considering an example of a simple sales per product aggregation.

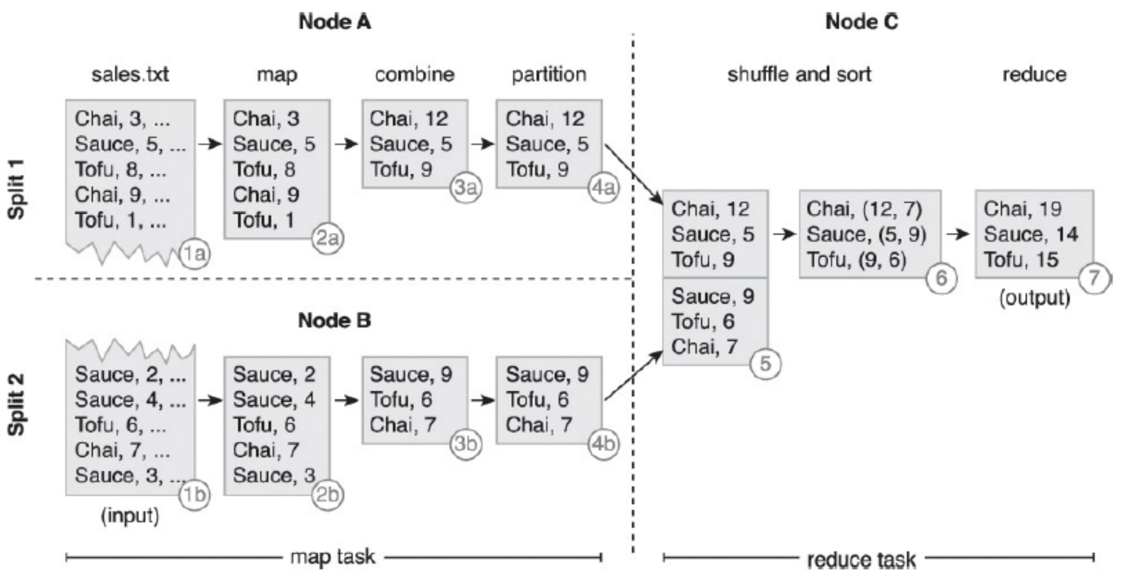


Figure 6.18 An example of MapReduce in action.

MapReduce in action: Adopted from Erl et al., page 133.

First, the input data is divided into chunks called input-splits. Each chunk is passed to a single map function. In our example, the input is split into two chunks. Second, the map task processes a single split and extracts the key/value pairs. In our example, we have data located on two nodes and the key/value pairs are extracted in parallel. Each product in a chunk corresponds to a key and its quantity represents the value. Next, the combiner locally (on each node) sums up the product quantities. It is important to note that the combiner function is optional. In principle, the reduce function can directly work with the output of the mapper function, however as the map and reduce tasks are typically run on different nodes the movement of data can contribute to processing latency. Hence, the optional combiner stage could be thought of as an optimization stage (Erl et al., 128). In the partition stage, the output obtained from the mapper or combiner is divided into partitions equal to the number of reducers. Records that have the same key belong to the same partition. In our example, there is only one reducer task and thus the data was not partitioned. The partition stage concluded the map task and what follows is the reduce task.

The first stage in the reduce task is known as the shuffle stage which copies the output from all partitions to the nodes that will run the reduce task. Next, the input key/value pairs are grouped and sorted so that there is a single key/value pair per group where the key represents a unique group key and the value is a list of all group values. In our example, the shuffle and sort stage are represented with numbers 5 and 6. Finally, the reducer stage summarizes all the relevant records together. In the example displayed above, the reduce function sums up the quantities of each unique product and creates an output.

It is important to note that Hadoop's MapReduce requires files to be stored in the Hadoop Distributed File System (HDFS). This ensures that very large files are distributed across multiple nodes. Thus, it gives us an advantage of working with very large data sets in a distributed system. HDFS splits the large file into blocks (128MB by default) and each block gets replicated multiple times for fault tolerance (typically three times). Thus, HDFS prevents loss of data if under any circumstance a node fails. When an HDFS client wants to read a file, the NameNode is asked for the location of the blocks that make up the file requested. Then it reads block contents from the DataNode closest to the client. When writing data, the client requests from the NameNode three DataNodes that will be able to store the block replicas (Shvachko *et al.* 2010).

MapReduce has two types of trackers a JobTracker and a TaskTracker. In a typical scenario, a client submits a job to the JobTracker, which is located on the NameNode, to process some data. The JobTracker identifies the DataNodes that store the particular blocks of data needed to conduct the job and then assigns tasks to different TaskTrackers. The JobTracker also monitors the status of each of the tasks. TaskTrackers, located on different DataNodes, allocate CPU and memory for the tasks assigned. They also constantly communicate with the JobTracker (e.g. send back results) (Dean *et al.).*

Spark

Spark is a flexible alternative to MapReduce. It is flexible as it does not require the data to be stored in HDFS although it supports it. Besides allowing data to be stored on HDFS, Spark supports other formats including Apache Cassandra[5] and AWS S3[6]. Spark was mainly written in Scala, but it provides developer API for Java, Python, and R. Besides supporting batch processing, Spark also supports streaming, business intelligence, graphs, and machine learning workloads (Apati, 148).

Spark can perform operations up to hundred times faster than MapReduce[7]. The main reason for this superiority is because MapReduce writes data to disk after each map and reduce operation while Spark keeps most of the data in memory (i.e. RAM) after each transformation. Spark can spill over to disk if the memory is filled. Hence, for applications that require iterative algorithms and interactive operations Spark performs significantly better than MapReduce as the time required to execute jobs is not spent on reading and writing to disk, creating replications (due to HDFS), and for serialization. It is then no surprise that Spark is the computation engine of choice when it comes to machine learning as most machine learning algorithms are iterative in nature. Hence, we can state that Spark supports more applications than MapReduce while still retaining the scalability and fault tolerance of MapReduce. In fact, Spark was explicitly designed to overcome the inefficiency of the MapReduce model when it comes to performing iterative and interactive computations. (Alapati, 148). Spark is able to achieve this through an abstraction called resilient distributed datasets (RDDs).

---

[5] Apache Cassandra is distributed NoSQL database management system that can store large amounts of data across many commodity servers.
[6] AWS S3 is a storage mechanism provided by Amazon for storing and retrieving large amounts of data.
[7] https://spark.apache.org/

RDDs represent read-only collection of objects (i.e. immutable) which are partitioned across multiple machines. The ability to be partitioned allows us to perform parallel operations on them. RDD's can be cached in memory and reused so that many MapReduce-like parallel operations can be performed. The fault tolerant property is achieved through a concept known as lineage. For example, if a partition is lost an RDD will be able to rebuild it as it possesses the information of how it was built in the first place (Zaharia et al, 1). Besides being immutable and cacheable RDDs are also lazily evaluated. RDD's can be created in the following ways: when loading a file from a shared file system such as Hadoop Distributed File System (HDFS), by parallelizing a Scala collection in the driver program, by transforming an existing RDD (e.g. using a filter or map operation), and by calling an action (e.g. cache or save) (Zaharia et al., 2). There are two types of operations that can be performed on an RDD transformations and actions. Actions such as count(), take(), top(), first(), and others return values. On the other hand, transformations such as map(), sample(), filter(), distinct(), sortBy() create a new RDD based on the one we are operating on (Alapati, 177-178).

Besides being better than MapReduce when running interactive and iterative computations, Spark is also easier to use and more accessible. According to Alapati, "Compared to MapReduce job, a Spark job means a 10:1 reduction in lines of code" (Alapati, 151). It is more accessible as it has more APIs that enable processing – Java, R, Python, and SQL as opposed to only Java that is supported by MapReduce. Hence, developers can be more productive when using Spark compared to MapReduce (Alapati, 151).

At a high level, a Spark application has a driver program, one that contains the main method or the starting point of the program and executes parallel operations on a cluster (Spark)[8]. In simple terms, the driver program has the processing code that will be run on each of the worker nodes in the cluster. The SparkContext, RDDs and jobs are defined within the driver program. The location of the driver is not dependent upon the master or slaves. The driver program can also launch more than one job on the cluster (Alapati, 180). The SparkContext helps us establish a connection to a Spark cluster and is used to create RDDs[9]. A Spark application is an instance of a SparkContext class.

As we submit a job (e.g. perform an action on an RDD) the driver process works with the resource manager (e.g. YARN) to get resources assigned to various nodes to perform the given tasks. Specifically, for YARN, the application master (which is the first container started for that application) requests resources from the resource manager and after it gets them it launches YARN containers on the slave nodes with the help of Node Managers that run on each of the nodes. Each YARN container has a number of virtual cores and memory as well as a Java Virtual Machine (JVM) that runs the code. When running Spark on YARN it is important to understand that Spark supports two modes: cluster and client mode. In yarn-cluster mode the driver program runs inside the application master. As stated by Sandy Ryza,

"This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container. The client that starts the app doesn't need to stick around for its entire lifetime."

[8] http://spark.apache.org/docs/latest/rdd-programming-guide.html#overview
[9] http://spark.apache.org/docs/latest/running-on-yarn.html

In yarn client mode the driver is run outside the cluster and the only duty of the application master is to negotiate resources from the resource manager. According to Ryza, "The client communicates with those containers to schedule work after they start." The cluster mode is better suited from production jobs while the client mode is more appropriate for interactive use when one needs to see the output immediately (Ryza). Most importantly, understanding the difference is crucial as it has consequences when it comes to tuning Spark. For example, in cluster mode the driver program is run in a YARN container in a worker node while in client mode the driver is run inside the master node of EMR. All of my clusters were run in client mode.

Once the driver process gets the resources requested it formulates an execution plan in terms of a Direct Acyclic Graph (DAG). Next, the DAG scheduler, which is part of the driver process, divides the DAG into tasks. Then the task scheduler, which is also part of the driver process and knows about available resources on the cluster, schedules the tasks on worker nodes in the cluster. The DAG is sent to the worker nodes that contain executors which are responsible for carrying out the DAG operations (Alapati, 663-664).

Each node performs parts of the total calculations requested by the user on a subset of data stored locally, so that the data processing is done in parallel.

To summarize briefly, there would be multiple servers hosted on AWS and the user would link to them. Tasks will be sent out by the master and the slaves will process them and send the results back. This system ensures that there is no communication between the slave nodes and that the models can be run in a distributed manner.

6. Preprocessing

An important component of the project was to gain familiarity with the pyspark syntax. Hence, the following section describes the basic preprocessing techniques performed using pyspark. It is important to note that arriving at a model with the highest accuracy was not the main priority of this project thus the preprocessing techniques used were merely done to make the data ready for all the algorithms.

The original dataset was composed of four files was uploaded to S3 storage which was then loaded into a Zeppelin notebook. While the dataset had 208 variables, many of them did not appear relevant and a number of them had many fields with missing values. After inspecting the documentation, I have selected 26 variables that seem to be useful in predicting property value (i.e. the target variable). This reduced the overall size of the data significantly and ensured that the preprocessing could be done faster. The table below displays the names of the all the variables initially selected.

Set of initial variables			
State	Unit type	Number of people	Lot size
Has bath	Number of bedrooms	Number of units	Heating fuel type
Has fridge	Number of rooms	Has hot/cold water	Has running water
Has sink	Has stove	Has phone service	Has toilet
Vacancy status	Family type employment status	Household language	Limited English speaking household
Multigenerational household	Has plumbing facilities	Number of workers in family	Year built
Adjustment factor house price	Property value		

- The original variable names were encoded using a few letters and for clarity they were renamed to names that could easily be understood.
- Many of the variables had the wrong data type, for example the variable "state" was encoded as an integer. All variables were casted to the correct data type. This was easily done using the df.withColumn() method.
- Next, the variables with missing values were identified. All the records that had a missing value for the variable "property value" were removed. For the rest of the variables NA's had a meaning. For instance, for the vacancy status variable the NA's represent an occupied household. Thus, they were filled according to the documentation.
- Many of the categorical variables had their levels encoded with numeric strings, and hence the levels of those variables were renamed for easier understanding. This was done by creating a user defined function (udf) which was imported from pyspark.sql.functions.
- The preprocessing revealed that the variable "unit type" only contained one level, hence signaling that it will not be useful in modelling. It was removed.
- Similarly, after reading the documentation, the variable adjusted factor housing price was removed as it did not apply to any of the variables.
- As I was interested in a classification task, I have found the median value of the property value ($175000), and labeled instances below that price as "below" and instances equal or above this price as "above." This concluded the first part of preprocessing and the dataset was saved to S3. Since machine learning algorithms in Spark cannot work with categorical input variables, the "above" category was encoded as 1, and the "below" as 0.
- The "year built" variable was somewhat problematic given the way the data was encoded. Here is a snapshot:

  1 – 1939 or earlier
  2 – 1940- 1949
  3 – 1950 – 1959
  …
  8 – 2000 – 2004
  9 – 2005

We can see that the years are not evenly spaced. For example, number 8 captures the period of 5 years, while number 9 only one year. Thus, the difference between 2 and 3 is not the same as the distance between 8 and 9. I have recoded this variable so that the number representing the year captures the median year if it represents a range, otherwise it will be equal to that year.

Code - Range
1945 – 1940-1949
…
2002 – 2000 – 2004
2005 – 2005
…

- All the numeric variables were scaled using the MinMaxScaler. Since the MinMaxScaler accepts a vector as input, I used a VectorAssembler to transform the numeric features into a vector and then transformed them.
- The categorical features needed to be one hot encoded. First, I used a String Indexer which transforms each categorical variable into a numeric counterpart. Second, the One Hot Encoder would create a separate column for each unique level within a categorical variable.
- Finally, the scaled numeric variables and the newly created variables would have to be transformed and then joined into a vector. The vector assembler helps us achieve that.
- The work of creating a vector indexer, one hot encoder, and vector assembler for each variable was greatly simplified with the Pipeline object. This is because we can run a fit and transform method once on the entire collection of variables instead of once per variable.
- Last, but not least, the data was split into train and test with 70% of the data reserved for test and 30% of the data used for test.
- Prior to exporting the data to S3, the data had to be transformed into the correct format. First, the data was transformed from a Spark data frame into an RDD. Next, the data was mapped from a RDD of tuples to a RDD of labelpoint. Then, the rdd was repartitioned into a single partition and stored to S3 in a libsvm format.

7. Algorithms Used

Decision Tree

In predictive modelling, the main goal is to use feature variables to estimate the value of the target variable. One way to think about is to segment the population into subgroups that have different values for the target variable. For instance, in a binary classification problem, ideally there would be two subgroups each corresponding to a different label of the target variable. More importantly, we would want the instances within the subgroup to have identical values for the target variable. More importantly, if the segmentation is done using the variables that will be known when the target variable is not then we could use these segments to predict the value of the target variable. In order to for our segments to be meaningful and predictive (i.e. homogeneous in respect to the target variable) we need to use features that contain important information about the target variable. Now, assuming that our dataset contains many candidate features that may be used to split an attribute into segments with respect to the target variable, how do we choose the one that is most informative? (Provost et al, 43-79).

In decision trees, the most common splitting criterion is information gain which in based on a measure called entropy. Entropy is measure of disorder or purity that can be applied to a segment. Intuitively, if a segment has many instances of of both target variables there would be a high level of disorder, the segment would be impure and would have high entropy. Conversely, if a segment has all instances belonging to a single class the level of disorder would be at its minimum, hence the segment would be considered pure and the entropy would be 0 (Provost et al, 43-79).

An attribute segments a set of instances into multiple subsets and entropy informs us how pure each of those subsets is. However, we are interested in how much the entropy changes over entire segmentation an attribute creates. For that, a measure called information gain is used. A high information gain means that a particular variable provides a lot of information about the target variable. On the other hand, a low information gain corresponds to a less informative variable (Provost et al, 43-79).

While entropy is a common impurity measure used for selecting the best split another measure called Gini is also frequently used. It is important to note that in general the results they achieve are fairly consistent (Tan et al, 160).

Thus, a decision tree recursively finds informative attributes that segments the instance space into similar regions. This is repeated until each region in the partition (each leaf) only has records that belong to a single class or until a stopping criterion is reached.

The advantage of decision trees is that they can be easily visualized and understood even by nonexperts. Because each variable is processed separately the algorithm does not require scaling of the numeric features. Redundant features do not affect the accuracy of model as one of the two redundant attributes will not be used for splitting. Furthermore, decision trees are robust to noise, especially when methods that prevent overfitting are used. (Tan et al, 169). The main disadvantage of decision trees is that even with pre-pruning they tend to overfit which leads to poor generalization performance (Muller et al, 85).

Ensembles

Ensembles represent a set of techniques that combine multiple machine learning models to create more powerful models. We have seen that a decision tree is prone to overfitting, which means that it suffers from high variance. In other words, if we change the training data on which the decision tree was trained the resulting models could be quite different. Ensemble techniques are designed to tackle this problem. The idea, which can be mathematically proven, is that the variance could be reduced if we train multiple independent base models and then average their results if we are facing a regression problem or take a majority vote on the predictions if we are dealing with a classification problem (James et al, 316). The pre-condition is that each base classifier has to perform better than random guessing and the errors of the base models should not be correlated (Tan et al, 275). Examples of ensemble methods include Random Forests and Gradient Boosted Trees.

Random Forest

The idea behind random forest is that each individual tree trained will do relatively well in predicting a part of the data but will will also likely overfit on that part of the data. As previously mentioned, taking a majority vote will reduce the amount of overall overfitting. Random forests get their name by injecting randomness in the tree building process so that each tree that is built will be different from the other. There are two ways in which the randomness gets introduces in random forests: by selecting random samples and by selecting random features. The random number of samples is typically created using a bootstrap sample, which means that from the total of n samples, the samples are drawn with replacement n times. The data will thus be of the same size as the original set, but the samples that make it up will be different. Another important parameter is max_features which represents the total number of features considered when building a tree. A high max_features parameter will guarantee that the individual trees will be relatively similar while a low n_features will mean that the trees will be very different and it would require the trees to be very deep in order to fit the data well. In general, a rule of thumb is to set max_features = sqrt(n_features) (Muller et al, 90).

Random forests are widely used in modelling; they do not require heavy tuning of the parameters and they do not require the data to be scaled. They generally take all the advantages from a decision tree except that for interpretability. Random forests do not generally perform well on highly dimensional sparse data. On the other hand, they tend to do well with large amounts of data but are also slower to train than linear models. In terms of selecting parameters n_estimators, max_depth, and max_features are the most important. In general, the more trees we have the better, however training a large number of trees requires more computing resources and more time (Muller et al, 90).

Gradient Boosted Trees

Gradient boosted trees are another method for combining multiple decision trees into a more powerful learner. Gradient boosting works in a serial manner where each new decision tree tries to correct the mistakes (i.e. wrong predictions) of the previous one. Inducing randomness could be done using the subsamplingRate parameter which is the fraction of the training data used to train a decision tree. In this case, a subsample of the training data is drawn without replacement (Friedman, 1). The main disadvantage of gradient boosted trees is that to achieve a high accuracy careful fine tuning of the parameters is required which in turn leads to long training times. Gradient boosted trees work well with both categorical and continuous data and it does not require any scaling (Muller, 93-94)

Multilayer Perceptron Classifier

An artificial neural network is a model inspired the way human brains work. The basic unit of a neural network is a neuron or a node which receives inputs from either the data set (the input variables from the training set) or from other nodes and then computes an output. Each input is associated with a weight, which is a manifestation of the strength of the connection between the neurons. The idea is to find the weights that best match the input-output relationship of the training data. The multilayer perceptron classifier is a feed-forward neural network which means that the nodes in one layer are only connected to the nodes of the next layer (Tan, 249). The computation that is done at each node is done through an activation function. In Spark's implementation of this algorithm the activation function is fixed to a sigmoid function while the output layer of the network is the softmax. The advantage of neural networks is that they are able to build incredibly complex models and capture hidden patterns in large amount of data. Neural networks work especially well with large amounts of data. The downside is the computation time.

Logistic Regression

Logistic regression uses a linear model (a weighted average of the input features) to classify a set of records.

$$f(x) = w_0 + w_1x_1 + w_2x_2 + \cdots$$

In a standard linear model the output of a linear function, f(x), gives us the distance of an instance from the decision boundary. It ranges from $-\infty$ to $+\infty$. However, the goal of logistic regression is to produce a model that gives good estimates of class probability which should range from 0 to 1. Hence, the resulting range of $-\infty$ to $+\infty$ needs to be casted to some sort of representation of likelihood measure. The log of odds, has the same range, and it can be converted into a probability of class membership. Thus, we can think of logistic regression as of a model for the probability of class membership. What actually distinguishes logistic regression from other linear models is the choice of optimization function. Logistic regression uses the maximum likelihood model which on average give the highest probabilities to positive examples and lowest probabilities to the negative examples (Provost et al, 101-102). In Spark, logistic regression is implemented to minimize the negative log-likelihood with elastic-net penalty that controls for overfitting.

8. Cluster Tuning

This section describes the cluster configurations used in this project. For all the clusters, the ‘spark.dynamicAllocation.enabled’ property and ‘maximizeResourceAllocation’ were set to false. The dynamic resource allocation property in Spark is used when a user has multiple applications running concurrently. This way, Spark adjusts the resources used by an application based on the workload. For instance, the number of executors will increase or decrease based on the workload. Spark would create new executors when a new job starts. (Alapati, 676). Since I was working on one particular application and wanted to have the number of executors fixed to ensure fixed scaling I disabled this property. The ‘maximizeResourceAllocation’ property would calculate the maximum amount of compute and memory resources available for an executor on an instance and override the spark defaults setting[10]. Hence, different jobs might have different properties set for the executors which would not make them easy to track.

The ‘spark.locality.wait’ property is an extremely important parameter for this project as it represents the time for which Spark waits to launch a job in a more local location before it decides to assign the task to a more remote location. There are three different modes and Spark searches in the following order (process_local, node_local, rack_local). The default time Spark waits to switch between these locality modes is three seconds (Alapati, 715). I have adjusted the wait time to be 600 seconds, which means that Spark will try to assign a job

[10] https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html

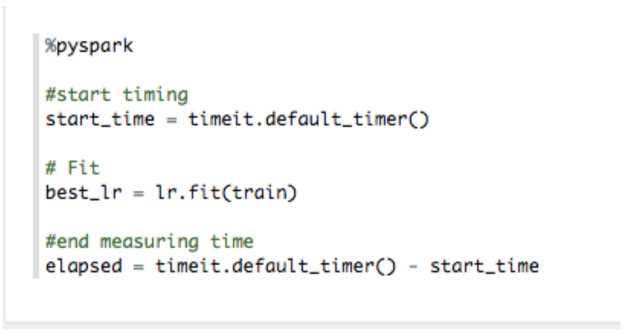
to process_local (where the data is) and if the executor is not available for 600 seconds it will schedule the tasks to another executor within the node. If for another 600 seconds an executor is not available, it will then assign the task to another node within the rack. I believe setting the time to 600 seconds will ensure that the data is processed locally (i.e. the data want travel between the nodes).

For each cluster, the amount of executor memory was fixed to 1GB, the driver memory to 6GB, number of executor cores to 1, and number of driver cores to 3. The executor memory overhead and driver memory overhead were set to 1GB. In practice, for a two node cluster (i.e. two worker nodes) YARN would show that the available memory is 24GB and 16 virtual cores. This translates to 12 YARN containers of which one is reserved for the application master. In other words, only 11 executors will be allocated for the execution of the tasks. The 12 containers are restricted because of the way the memory is allocated (1GB per executor + 1GB executor overhead) – thus, we can have maximum of 6 containers per node. On the downside, given that each YARN container uses one core, the total number of used cores for the cluster is 12. Hence, there will be 4 unused cores in the cluster. Utilizing the full capacity of the cluster could have been achieved by lowering the available memory per executor or adjusting the overhead memory per executor. Nevertheless, each of the clusters had the same number of unutilized cores per node.

The 'spark.default.parallelism' determines how many parts a resilient distributed dataset will be split into. Each part will be fed into an executor on which a task (map/reduce) will be performed. In principle, the larger the default.parallelism is the more tasks need to be launched and the smaller the parts that are fed into an executor. Hence, all else being held equal, having the input split into more chunks will ensure faster processing time, however it might lead to a situation in which the time required for processing such a chunk of data is lower than the amount of time required to schedule such a task. I have set the default parallelism to be 28, 56, 84, and 112 for the 2, 4, 6, and 8 node cluster respectively. Reflecting back on it I think I should have kept the value fixed and should have made it a multiplier of the number of available executors. This way I would have avoided a situation where one executor has more tasks than another which leads to some executors sitting idle.

9. Algorithm Timing

The time used for training an algorithm was done using the module "timeit." The method used is default_timer() which according to the python documentation is always time.perf_counter(). According to the python documentation, this method returns a float value "… (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid" (Python timeit). Below is a code snipped on how the logistic regression algorithm was timed. Other algorithms were timed in a similar manner.



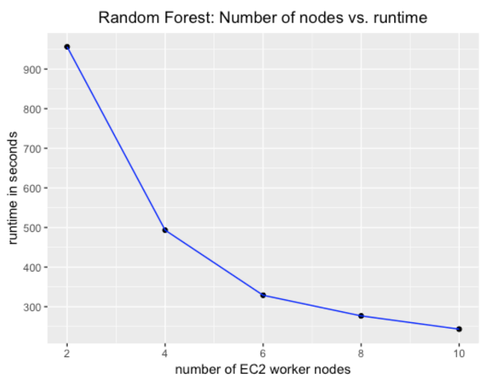
```
%pyspark

#start timing
start_time = timeit.default_timer()

# Fit
best_lr = lr.fit(train)

#end measuring time
elapsed = timeit.default_timer() - start_time
```

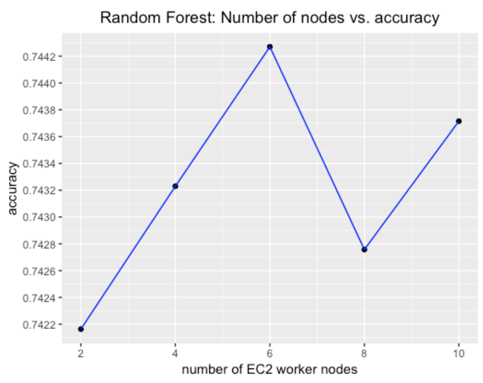
Measuring execution time of an algorithm

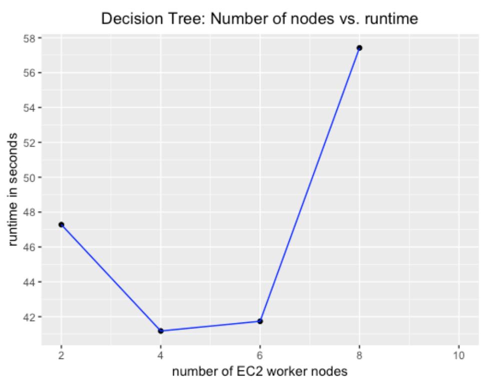
It is important to point out that rerunning the same algorithm on the same cluster often yielded to slightly different results the variation of which was not taken into account. The graphs in the results section present one time measurements. Nevertheless, it was observed that the timing was relatively consistent when the experiment was repeated across different cluster configurations. In other words, if the timing for an algorithm on a two-node cluster was greater than on a four-node cluster, then even after repeating the experiment for the same algorithm, despite showing slightly different time measurements, the two-node cluster would still consistently show that it took longer to train than the four-node cluster.

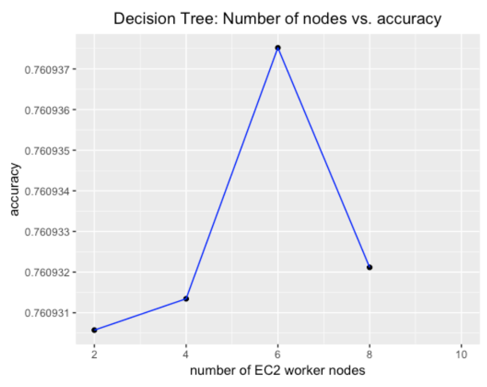
10. Results:

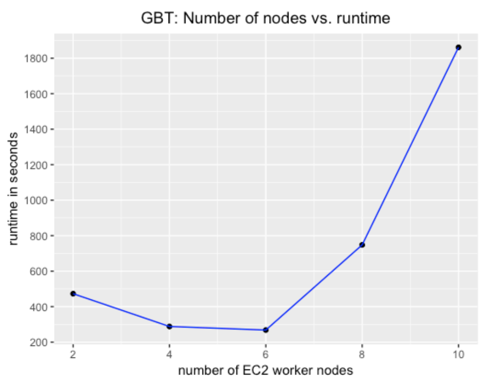
The goal of the project was to measure the training time of an algorithm and the accuracy achieved using different algorithms on different cluster sizes. The algorithms were not fine tuned due to time constraints and computation time considerations (e.g. cost incurred on AWS). In order to observe how the accuracy changes as the data gets distributed over an increasing number of nodes the same training parameters were used and a fixed seed was set. Using the same seed should have ensured that rerunning the same algorithm on the same cluster should yield to the same results. This was only observed to be true for the decision tree and random forest algorithm. For the gradient boosted trees and multilayer perceptron setting the seed did not ensure reproducibility of the results. The logistic regression algorithm implemented in Spark did not have a seed option.

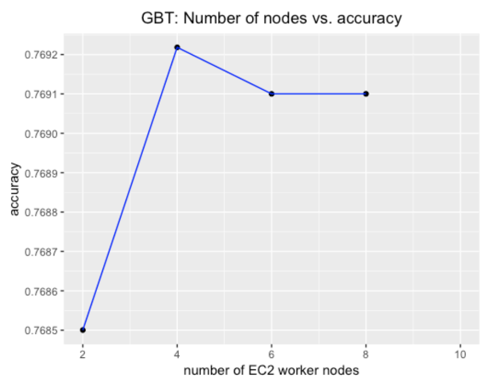


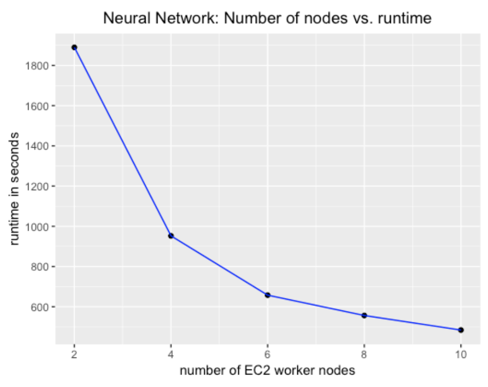


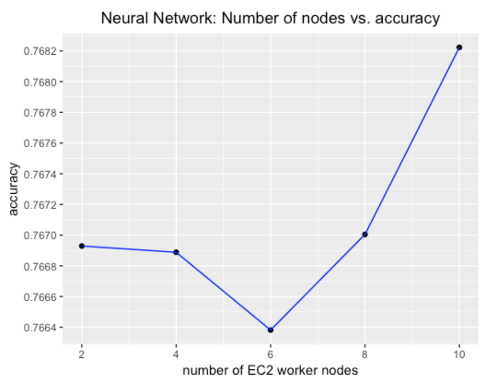


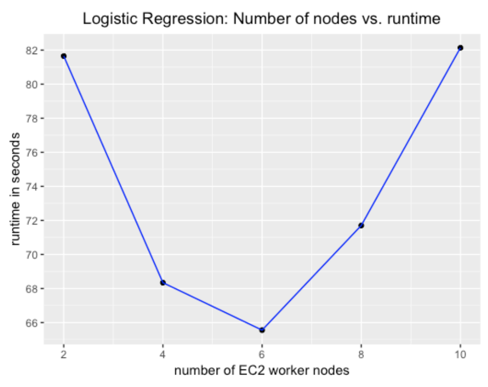


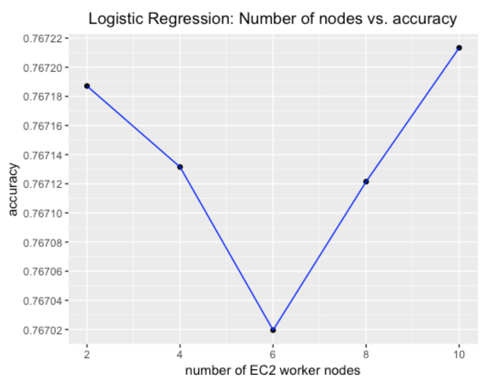








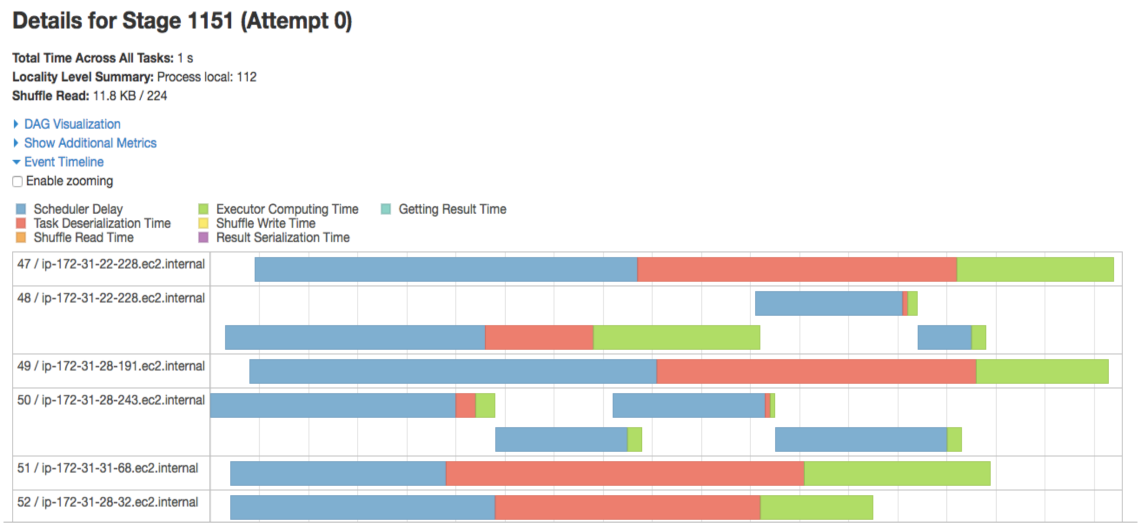




Results: **graphs do not start from a 0 scale***

11. Discussion and Future Work

We can see that as we increase the number of nodes the runtime behaves differently for different algorithms. Two trends seem to emerge. For the logistic regression, gradient boosted trees, and decision trees the runtime decreases as we scale out the cluster from a two to four to six-node cluster. After the six-node cluster, the runtime starts increasing drastically. What could explain the runtime increase for these algorithms on an eight-node cluster?



Details for Stage 1151 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Process local: 112
Shuffle Read: 11.8 KB / 224

DAG Visualization
Show Additional Metrics
Event Timeline
Enable zooming
Scheduler Delay
Task Deserialization Time
Shuffle Read Time
Executor Computing Time
Shuffle Write Time
Result Serialization Time
Getting Result Time
47 / ip-172-31-22-228.ec2.internal
48 / ip-172-31-22-228.ec2.internal
49 / ip-172-31-28-191.ec2.internal
50 / ip-172-31-28-243.ec2.internal
51 / ip-172-31-31-68.ec2.internal
52 / ip-172-31-28-32.ec2.internal

A stage in decision tree algorithm 8-node cluster [only showing 6 executors]

A possible explanation comes from observing a stage (i.e. a part of a job) of a decision tree algorithm from an eight-node cluster. We can see that that scheduler delay (blue area) is significantly larger than the actual time it takes to execute the tasks (green area).

The second trend is produced by the random forest and neural network algorithm. It shows that the runtime decreases as the number of nodes in a cluster increases. While I have not preserved a screenshot of a stage in an eight- node cluster of one of these algorithms, I would suspect that the computation time of tasks for these algorithms offset the scheduler delay. Additionally, I would suspect that the runtime would not decrease indefinitely had the cluster sized increased even further. I would expect that at some point the communication overhead and scheduler delay would yield to an increase in training time for these algorithms.

The resulting graphs of accuracy can only be fairly interpreted for the decision tree and random forest algorithm as rerunning them on the same cluster (with a fixed seed) always produced identical results. We can observe that the accuracy of the model keeps increasing as we move from a two-node to a four-node cluster. It further increases on a six-node cluster and then drops on an eight-node cluster. How is it possible that the accuracy on a two-node cluster is worse than on a six-node cluster? One would expect that training a model on fewer larger chunks of data and then averaging the results would produce a better model than when a model is trained on more chunks of data.

There are two ways to think about these findings. First, the model extracted from on a two-node cluster is clearly not the best possible model that could have been created because the model was not fine tuned and cross-validated. If the model had been tuned and cross-validated I suspect the results would look different. Hence, extracting a model from a more and more distributed chunks of data seemed to have created a more generalizable model purely **by chance** than the one trained on fewer chunks. This result is similar to Hall et al., findings who used rule-based classifiers. I would suspect that training the two algorithms using different seeds would produce different patterns for accuracy. Additionally, I would suspect if the data was partitioned differently even with the same seed the results might have been different. Furthermore, to adequately test our initial hypothesis I believe the models should have been fine tuned and cross-validated before their generalizability could have been assessed. In that case, I would expect a general trend in which the accuracy drops as the data gets distributed over increasing number of nodes.

Second, the accuracy for the decision tree algorithm has changed only for the fifth decimal place while for the random forest algorithm for the third decimal place. This raises a question of whether the accuracy results obtained from different cluster sizes are significantly different from each other. My suspicion is that there are not. If we follow this line of logic and conclude that the accuracies obtained for all the algorithms over increasing number of nodes (2-8) are not statistically different from each other then we may ask what is the optimal number of nodes for this data set to be run on? Then, for the decision tree, gradient boosted tree, and the logistic regression algorithm it would be 6 as it has the lowest training time. On the other hand, for the neural network and random forest it seems to be 10[11]. Hence, we would conclude that there is a tradeoff between runtime and number of nodes in the cluster (for the range 2-8). In addition, there does not seem to be a tradeoff between accuracy and number of nodes (for the range 2-8).

These findings naturally raise a question: until how many nodes will the accuracy remain acceptable? What will happen to accuracy if we trained the data on 20, 50, or 100 nodes? I would suspect the accuracy to start falling after a certain point as we would be training our model on less and less data. At the same time, until how many nodes will the runtime keep decreasing for the random forest and neural network algorithm? That is, at what point will communication costs and task scheduling become a bottleneck? There must be a point at which either the accuracy drops significantly to a level which is unacceptable or when the time required to train these algorithms starts increasing. This raises another question: what constitutes a significant accuracy falloff and how do we recognize it? If we use the accuracy obtained from training a model on s single processer as a benchmark for determining what constitutes a significant accuracy falloff, then, what happens when the dataset is too large to learn from a single processor? In that case, it might not be possible to know what the maximum accuracy is.

12. Skills acquired

During the project I have acquired numerous skills. After this project I am able to:

- Create EMR clusters on AWS
- Use pyspark for preprocessing and modelling data
- Fine tune Spark clusters using YARN
- Ensure that processing on Spark is done locally (within a node)
- Use Ganglia, Zeppelin, and AWS S3

[11] Please note that some algorithms were run on ten nodes while other have failed.

13. Bibliography

Alapati, S.R. (2017). *Expert Hadoop administration: managing, tuning, and securing Spark, Yarn, and HDFS*. Addison-Wesley.

Amazon EMR Release Guide. (2018). Retrieved from: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-ganglia.html

Apache Zeppelin: Web-based notebook that enables data-driven, interactive data analytics and collaborative documents with SQL, Scala and more. (2018). Retrieved from: https://zeppelin.apache.org/

Boden [a] C., Rabl, T., & Markl, V. Distributed machine learning – but at what cost? (2017). *Machine Learning Systems Workshop at the 2017 Conference on Neural Information Processing Systems*.

Boden [b], C., Spina, A., Rabl, T. & Markl, V. (2017). Benchmarking data flow systems for scalable machine learning. *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, 1-10, Chicago, IL, USA.*

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *Google, Inc.*

Erl, T., Khattak, W., Buhler, P. (2015). *Big data fundametals: concepts, drivers & techniques*. Arcitura Education Inc.

Friedman, J.H. (March 26, 1999). Stochastic Gradient Boosting. Retrieved from: https://statweb.stanford.edu/~jhf/ftp/stobst.pdf

Hall, O. H., Chawla, N., Bowyer K. W., & Kegelmeyer, W.P. Learning rules from distributed data. Retrieved from: https://www3.nd.edu/~nchawla/papers/LSPDM00.pdf

James, G., Witten, D., Hastie, T., Tibshirani, R. (2013). *An introduction to statistical learning with applications in R.* Springer.

Muller, A., Guido S. (2016). *Introduction to machine learning with python: A guide for data scientists.* First edition. O'Reilly Media, Inc.

Provost, F., Fawcett. T. (2013). *Data science for business: What you need to know about data mining and data-analytic thinking*. O'Reilly Media, Inc.

Provost, J.F., Hennessy, D.N. (1996). Scaling up: Distributed machine learning with cooperation. Proceedings of AAAI'96, pp.74-79

Python timeit – Measure execution time of small code snippets. Python version 3.7.1. Retrieved from: https://docs.python.org/3/library/timeit.html

Ryza, S. (May 30, 2014). Apache Spark resource management and YARN App Models. [Blog post]. Retrieved from: https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

Shvachko, Konstantin. Kuang, Hairong. Radia, Sanjay. Robert Chansler. (May 2010). "The Hadoop Distributed File System". *MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies* (MSST), pages 1-10.

Tan, Pang-Ning. Steinbach, Michael. Kumar, Vipin. (2005). *Introduction to Data Mining*

Veiga, J., Expósito, R., Pardo, X., Taboada, G., & Tourifio, J. (2016, December). Performance evaluation of big data frameworks for large-scale data analytics. *2016 IEEE International Conference on Big Data*, 424-431.

Zaharia, M., & Chowdhury, M., & Franklin, M. J., & Shenker, S., & Stoica I. Spark: Cluster computing with working sets. University of California, Berkley.

Zhang, Y. Distributed machine learning with communication constraints. (2016). PhD thesis. Retrieved from: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-47.pdf