Spark Application on AWS EC2 Cluster:

ImageNet dataset classification

using Multinomial Naive Bayes' Classifier

Chao-Hsuan Shen, Patrick Loomis, John Geevarghese

Acknowledgements

Special thanks to professor Ming-Hwa Wang. And thanks to our parents for giving birth to us.

Table of contents

Abstract

Image processing is an expanding field with iterative applications identifying and classifying images. Many machine learning algorithms have been implemented to further the accuracy and increase performance of classification paradigms. One of the most used platforms, Hadoop MapReduce, serves as a great tool to identify and classify images, but with the increasing need for high-throughput applications, there is room for other paradigms to elevate the issues other platforms face. Those applications include, but are not limited to, Spark and Storm. Spark has touted a performance of being 100 times faster than Hadoop for certain applications. Spark improves upon the MapReduce paradigm by implementing in-memory processing. Other applications were built on top of the MapReduce paradigm, such as Mahout. Spark is attempting to implement and build a Machine Learning Library composed of functions used for various machine learning applications.

1. Introduction

1.1. What is the question ?

To determine what environment, Spark or Hadoop, best serves the developer's purposes. How is that environment impacted by Hadoop and Spark, and in what way can we measure effectiveness.

How does one set-up a cluster on an Amazon Web Services Elastic Cloud Computing environment?

What applications can be run on a cluster deployed on the cloud?

How to choose between different tiers of instances according to your application?

1.2. Why this is a project related the this class

Cloud Computing alleviates issues of memory and CPU resource restriction by using the latest distributed computation technology such as, Spark. Our goal is to deepen our understanding of Cloud Computing architecture by solving problems. We implement Image classification on Spark on EC2 cluster.

1.3. Scope of investigation

First, we will learn amazon web service. In this project we will use EC2 and S3 to deploy our virtual cluster on the cloud.

Second we will learn Spark? What is the difference between spark and hadoop? Advantages and disadvantages.

Third we will learn Naive Bayes based classification algorithm. And use it to classify ImageNet's dataset.

We evaluated *resilient distributed dataset* (RDD), a special data structure used by Shark to keep reusable data in memory to improve performance and also fault tolerance.

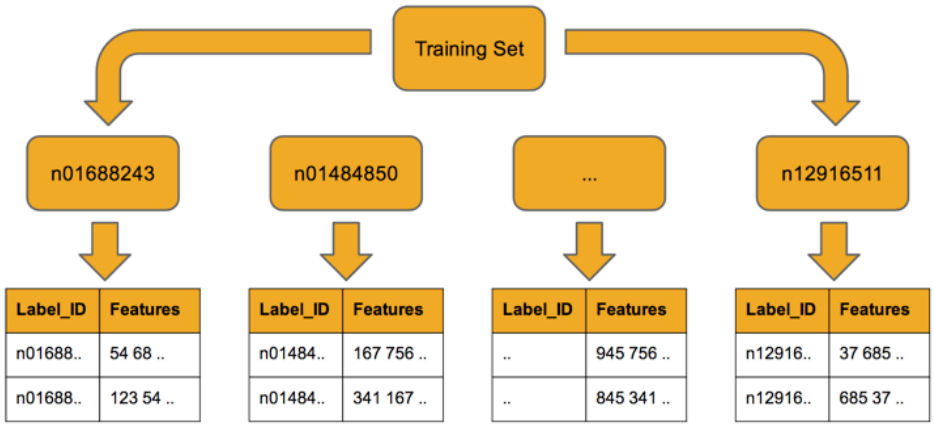
2. Theoretical bases and literature review

2.1. Theoretical background of the problem

Given the current computing paradigms, which would be the best to solve the problem of image classification

2.1.1. Naive Bayes' Classifier

The ImageNet dataset comes in a raw file hierarchy format. The top level folder contains files of every single class, each file has a number of images, where each image has a number of attribute objects describing the picture.



Our assumption for this dataset, is that features are independent of one another within each class. There are two steps to classify the data: training and predicting. Naive Bayes Classification is based on estimating $P(X \mid Y)$, the probability or probability density of features x given y. The ImageNet dataset requires Multinomial distribution, because all features represent counts of a set of words or tokens, also known as "bag-of-words" model.[1]

There are two things to know about the Naive Bayes algorithm; conditional probability and Bayes' rule. Conditional probability is the probability that something will happen, given that something has already happened. An example of this, is given all of the class labels within the training set, what is the probability that the next feature image will be a french fry and orange? To determine this we calculate the conditional probability
$P(french\ fry\ \&\ orange) = P(french\ fry) * P(orange/french\ fry)$.

Bayes' rule predicts an outcome given multiple evidence. This is done through 'uncoupling' multiple pieces of evidence and treating each piece of evidence as independent, which terms this idea as a naive Bayes extrapolation. This can be formulated as follows;
$P(Outcome/Multiple\ Evidence) = P(Evidencee_1/Outcome) * ... * P(E_n/O) * P(O)$.
There are multiple priors so that we gave a high probability to more common

[1] http://www.mathworks.com/help/stats/naive-bayes-classification.html

outcomes, and low probabilities to unlikely outcomes. These are also called base rates and they are a way to scale our predicted probabilities.[2]

Once the Naive Bayes' algorithm was understood, the next step is to apply Naive Bayes' to predict an outcome. To do this, we had to run the formula that was previously stated for each possible outcome. Since we are trying to classify, each outcome is called a class and it has a class label. We will be conducting an inspection on evidence to determine how likely it is to be a certain class. Deduction gives assignment of label to each entity. The class that has the highest probability is declared the "winner" and that class label gets assigned to that combination of evidence.

Running through the Naive Bayes' algorithm, using the ImageNet dataset as an example, the following formulas are applied;

Prior Probability:

$$P(ImageClass) = BaseRate = ImageClass/SumOfTotalClasses$$

Probability of Evidence:

$$P(Category) = SumOfProbCategory/SumOfTotalClasses$$

Probability of Likelihood:

$$P(Category/ImageClass) = CategoryValue/TotalCategoryValue$$

Determine "Best Fit" or Posterior Probability:

$$P(ImageClass/CategoriesOfClass) = (P(CategoryVal_1/TotalCategoryVal_1) * ...$$

$$* P(CategoryVal_n/TotalCategoryVal_b))/(P(CategoryValue_1) * ... * P(CategoryValue_n))$$

The "best fit" or posterior probability is calculated until the highest value is calculated. Once the highest value is found, the class label is then classified with the label that had the highest posterior probability.

2.2. Related research to solve the problem

From a previous project, running the ImageNet dataset on the Mahout platform gave a 7.3% accuracy. The performance of the text classification Multinomial Naive Bayes' algorithm on Mahout gave lackluster time (20 - 30 minutes, depending on size of dataset), mainly due to it's MapReduce tasks and a python script that converts the dataset into a sequence file with key/value pairs. The result was correct, and also had many modules used, such as TF-IDF and sparse matrix vectors, but the time it took to complete was rather high and thus gives us a chance to improve the efficiency by using another platform. The hope is that Spark, along with scala and akka, will contribute greatly to decreasing overhead and increasing algorithm efficiency.

[2] http://stackoverflow.com/questions/10059594/a-simple-explanation-of-naive-bayes-classification

2.3. Your solution to solve this problem

Implementing a text classification, Naive Bayes Classifier on Spark. Then we compare and contrast the difference. We will plan to collect various data attributes to potentially find patterns for use with that particular application. Once completed, comparisons can be made to a project previously completed, which used Mahout as a platform to classify the ImageNet dataset.

2.4. Where your solution different from others and why is better

2.4.1. Running comparisons on different frameworks to provide insight into the vast array of solutions that are currently provided. Instead of focus on one implementation and keep improving. This is like doing BFS instead of DFS.

2.4.2. Spark

Spark with Scala and Akka is an excellent way to build an application for parallelization. The actor based method has promising distributed properties to utilize resources and speed up efficient.

2.4.3. Resilient Distributed Dataset (RDD)

A special data structure used by Shark framework to keep reusable data in memory to improve performance and also fault tolerance. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

2.4.4. It will provide guidance to researchers and practitioners alike with references for others to understand and analyze their environment.

2.4.5. Shows use/failure of our particular application

3. Hypothesis

Naive Bayes' Classifier on the ImageNet dataset using Spark architecture will be more efficient, because this application has small and iterative operations. The in-memory data processing that the Spark architecture utilizes will help to increase efficiency at the cost of memory. Compared with Hadoop MapReduce paradigm, which must reload the trained data classifications from disk to run an iterative computation and keep shuffling data back and forth, Spark should respond fast in our user case.

4. Methodology

4.1. How to generate/collect input data

4.1.1. Background on Competition:

ImageNet started in 2010 by Stanford, Princeton and Columbia University scientists.[3] The training and testing package can be download on Imagenet's website[4]. It is a contest that is run every year. The most current challenge, ImageNet 2014, is composed of 38 entrants from 13 countries. The contest was sponsored this year by Google, Stanford, Facebook and the University of North Carolina. Contestants run their recognition programs on high-performance computers based in many cases on specialized processors called G.P.U.s, for graphic processing units.[5] The accuracy results this year improved to 43.9 percent, from 22.5 percent, and the error rate fell to 6.6 percent, from 11.7 percent, according to Olga Russakovsky, a Stanford University graduate researcher who is the lead organizer for the contest. Since the Imagenet Challenge began in 2010, the classification error rate has decreased fourfold, she said.[6] The actual data that is used for this project is free and encouraged be download and to interact with by ImageNet organizers.

4.2. Algorithm implementation basis

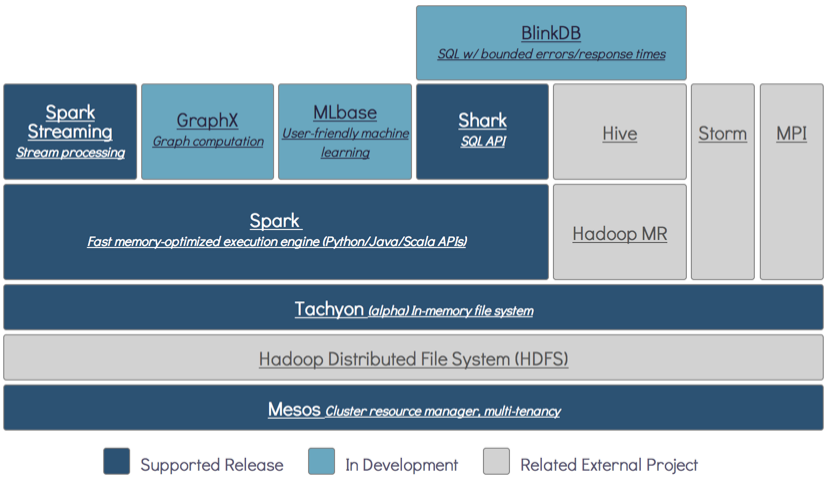
The following section explores which technologies are being used, and what their current uses are. To understand where these pieces fit within the process, it's imperative for our group to understand the ecosystem which houses the various applications that run "under-the-hood".

4.2.1. Spark Ecosystem

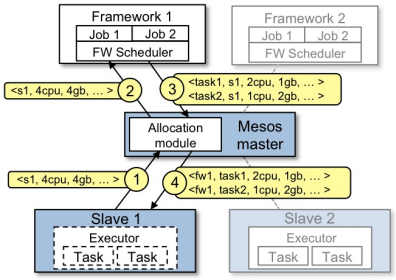
4.2.1.1. Spark

Spark is an open-source data analytics cluster computing framework originally developed in the AMPLab at UC Berkeley.[7] Spark's reason for creation was to rebuild Hadoop from scratch. Spark is built off of the Hadoop Distributed File System (HDFS). Spark is not tied to the two-staged MapReduce paradigm.[8] Spark's big advantage is utilizing in-memory cluster computing that allows user programs to load data into a cluster's memory and query it repeatedly, making it a great candidate for the machine learning task that we are attempting to solve.

[3] http://bits.blogs.nytimes.com/2014/08/18/computer-eyesight-gets-a-lot-more-accurate
[4] http://www.image-net.org/challenges/LSVRC/2010/download-public
[5] http://bits.blogs.nytimes.com/2014/08/18/computer-eyesight-gets-a-lot-more-accurate
[6] http://bits.blogs.nytimes.com/2014/08/18/computer-eyesight-gets-a-lot-more-accurate
[7] http://www.wired.com/2013/06/yahoo-amazon-amplab-spark/all/
[8] https://amplab.cs.berkeley.edu/wp-content/uploads/2013/02/shark_sigmod2013.pdf

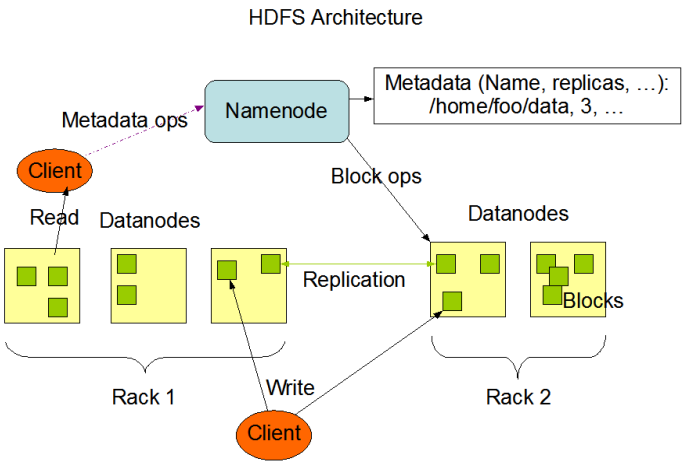


4.2.1.2. *Mesos cluster resource manager*



Mesos is a cluster manager that simplifies the complexity of running applications on a shared pool of servers[9]. Mesos is an important tool for researchers to configure. Spark can run on hardware clusters managed by Apache Mesos while also running alongside a Hadoop cluster.[10]

4.2.1.3. *Hadoop Distributed File System (HDFS)*



HDFS is a distributed, scalable, and portable file-system written in Java for the Hadoop framework. A Hadoop cluster is composed of worker nodes and a master node. The NameNode designates which datanodes hold 64 Mb size blocks across the cluster. To account for fault, HDFS stores two redundant copies of a block, one on the same rack, for rack awareness, and the other far away from the original.

4.2.1.4. *Tachyon (alpha) in-memory file system*

Tachyon is a memory-centric distributed file system enabling reliable file sharing at memory-speed across cluster frameworks, such as Spark and MapReduce. It achieves high performance by leveraging lineage information and using memory aggressively. Tachyon caches working set files in memory, thereby

[9] http://mesos.apache.org/
[10] https://spark.apache.org/docs/latest/running-on-mesos.html

avoiding going to disk to load datasets that are frequently read. This enables different jobs/queries and frameworks to access cached files at memory speed.[11]

4.2.2. Commits to MLlib:

Determining what functions and algorithms the library currently contains. As of now, the functions provided in Spark's machine learning library are very basic. We may be constrained by this aspect, and thus be unable to completed the required research

4.3. Language(s) used

4.3.1. Scala + Akka

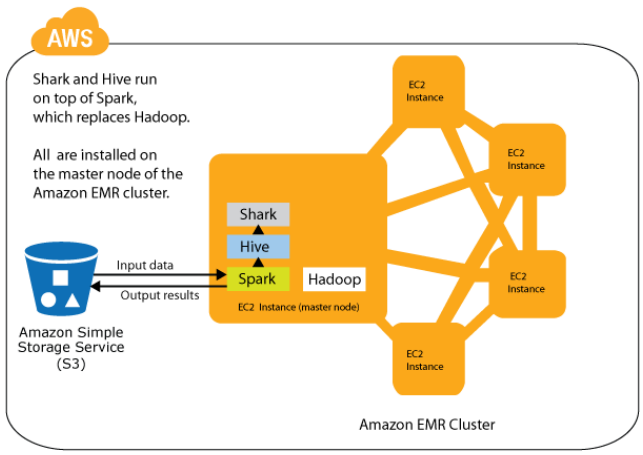
Scala is an object-functional programming and scripting language for general software applications.[12] Scala was built around fixing Java's shortcomings. The language is relatively new, as of 2003, and has gained much momentum in building its libraries. As a result of being built with Java in mind, scala runs off the JVM platform. Spark provides an interactive shell, which uses python and scala. The consciousness of scala makes it a nice choice to write MapReduce applications with.

Another reason in which scala has become more popular, is the addition of the open-source toolkit called, Akka. Akka employs the actor model, which helps distribute applications on the JVM. The actor model uses actors instead of objects. Actors do not share any attributes and have asynchronous-message passing.[13] When a message is received, an actor interprets the message and executes the instruction. The message that actors send are immutable.

4.4. Tools used

4.4.1. Amazon Elastic Compute Cloud

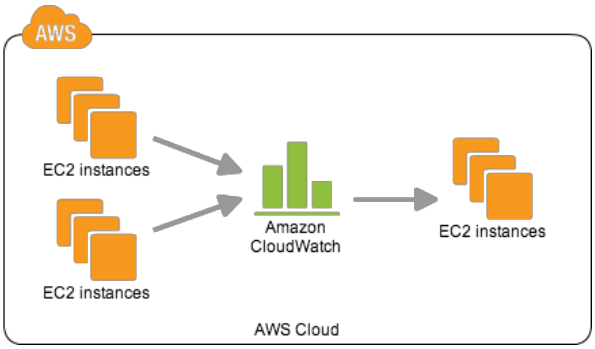
Also known as EC2, the Amazon Compute Cloud is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.[14] For our project, we setup a cluster of m1.large nodes consisting of 6 nodes, 5 worker nodes and a master node with the following specifications; Processor architecture is 64-bit, has 2 vCPU, 7.5 GiB memory,



[11] http://tachyon-project.org/
[12] http://en.wikipedia.org/wiki/Scala_(programming_language)
[13] http://c2.com/cgi/wiki?ActorsModel
[14] https://aws.amazon.com/ec2/?nc1=h_l2_c

instance storage consists of 2 x 420 GiB, is EBS optimized, and has a moderate network performance.

4.4.2. Amazon CloudWatch



Amazon CloudWatch is a monitoring service for AWS cloud resources and the applications that are run on AWS.[15] For our purposes, we used Amazon CloudWatch to to monitor Amazon Elastic Cloud Computing instances. This gave us system wide visibility into resource utilization, application performance, and operational health. Since we did not want to invest money into this project, we stuck with the basic monitoring service, which gave us seven pre-selected metrics at five-minute frequency and three status check metrics at one-minute frequency.[16]

4.4.3. AmpCamp:

AmpCamp[17], run by UC Berkeley, is a resource for understanding the basic components of using Spark on an AWS EC2 instance. The mini-course provides methods on how to interact with the spark-shell and how to interactively use the Machine Learning Library.

4.5. How to generate output

We run the training and testing on an EC2 cluster and then collect data points for analysis using Amazon CloudWatch. To generate the actual output (accuracy) we are running a scala script, both on local machine and on an EC2 cluster, on file sizes varying from 200Mb to 3.5Gb.

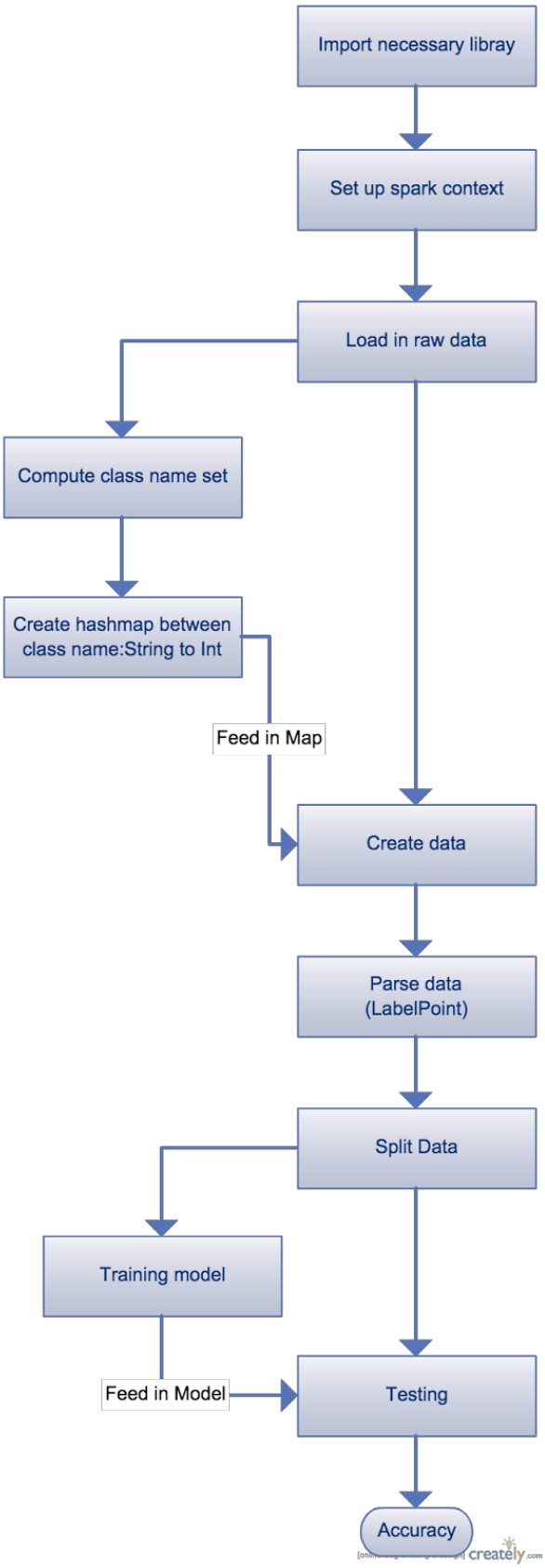
4.6. How to test your hypothesis

We have the same classification idea implemented in Hadoop and Spark. We can compare the running time and the accuracy between the two under the same cluster setup, to see which one performs better. With the help of AWS CloudWatch, we can have more metrics to base our conclusion off of. The point of this project is not to gain a higher accuracy, but to test the parallelization performance of cloud distributed technologies.

[15] http://aws.amazon.com/cloudwatch/
[16] https://aws.amazon.com/cloudwatch/details/#amazon-ec2-monitoring
[17] http://ampcamp.berkeley.edu/

5. Implementation

5.1. Code architecture



5.2. Design Flow

Our design flow is as follows:

1. Set up a goal and build concept inventory.
2. Divide our goal into small pieces.
3. Learn each individual idea and tool we need as we go.
4. Connect ideas and tools.
5. Test and tweak.
6. Deliver

To elaborate more on the design flow:

1. We have a clear goal: To implement Naive Bayes classification algorithm on Spark, and deploy the application on an EC2 cluster. Ideally, we want the implementation to be efficient, code elegant and easy to understand, all while being correct and concise. However, this project will be within a short period of time and will have many unknowns to learn, such as setting up a cluster that are all running the same configurations, learning to code in Scala+Akka, and what the ImageNet dataset is. We choose to focus on deepening our understand of Spark and AWS. With this particular focus, we can implement various aspects of the project without becoming too deeply ingrained in the details.

2. To fulfill our goal, we have three major questions to be answered:

    a. What is Naive Bayes classification?

    b. What is Spark?

    c. What is AWS?

    Naive Bayes classification is one of the easier parts to understand of this project. This is a mathematical application based on probability. However, Spark and AWS are both complex systems rather than mere simple tools. So we divide them into smaller pieces:

    - To learn and use Spark:
        - We must learn basis of what Scala is
            - Foundation of Spark and the use of Akka
        - Learn how to use Spark Shell
            - For interactive experiments to test ideas
        - Learn how to launch a Spark cluster
            - Distributed computation in both local machine
        - Find useful existing library for our purpose
            - MLlib for Spark, and Mahout for reference comparisons
        - Learn how to program in Scala API

        - Learn how to pre-process raw data
        - How to implement MLlib into Scala code
    - Learn how to submit application to a Spark cluster
- To learn and use AWS:
    - Get an overview of what AWS could provide. Identify we need to use EC2 and S3
    - Learn how to initiate an EC2 instance
    - Learn how to create S3 bucket and download | upload data to S3
    - Learn how to download and upload data on S3 from an EC2 instance
    - Learn how to initiate multiple EC2 instances and connect them into an Spark cluster
    - Learn how could Spark application load data from S3
    - Learn how to submit application on a Spark EC2 cluster

3. After dividing our project into specific goals, as I just depicted above, I proudly say we have accomplished “EVERY” task.
4. Basically, our code and the project are the results of connecting everything we have learned in constructing this project.
5. Of course, we test and tweak various components of the project as we go.
6. So now, we are able to deliver.

6. Data analysis and discussion

6.1. Output Generation

The output generation we are accumulating is from Amazon CloudWatch. Our focus is primarily on how AWS EC2 cluster runs, not the actual accuracy of the algorithm. We are interested in the network utilization, memory usage, and CPU utilization. These metrics will give us a better understanding of what type of resources a Spark Machine Learning application needs.

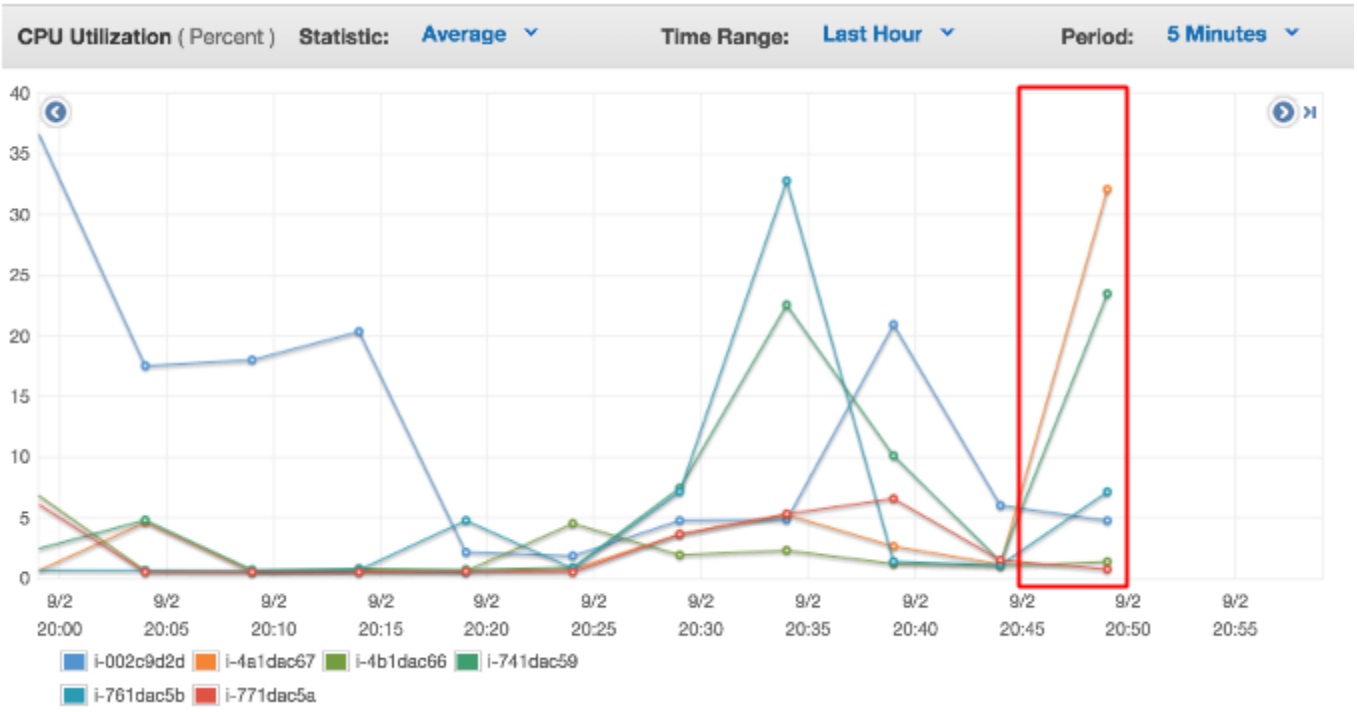
6.2. Compare output against hypothesis

6.2.1. Metrics

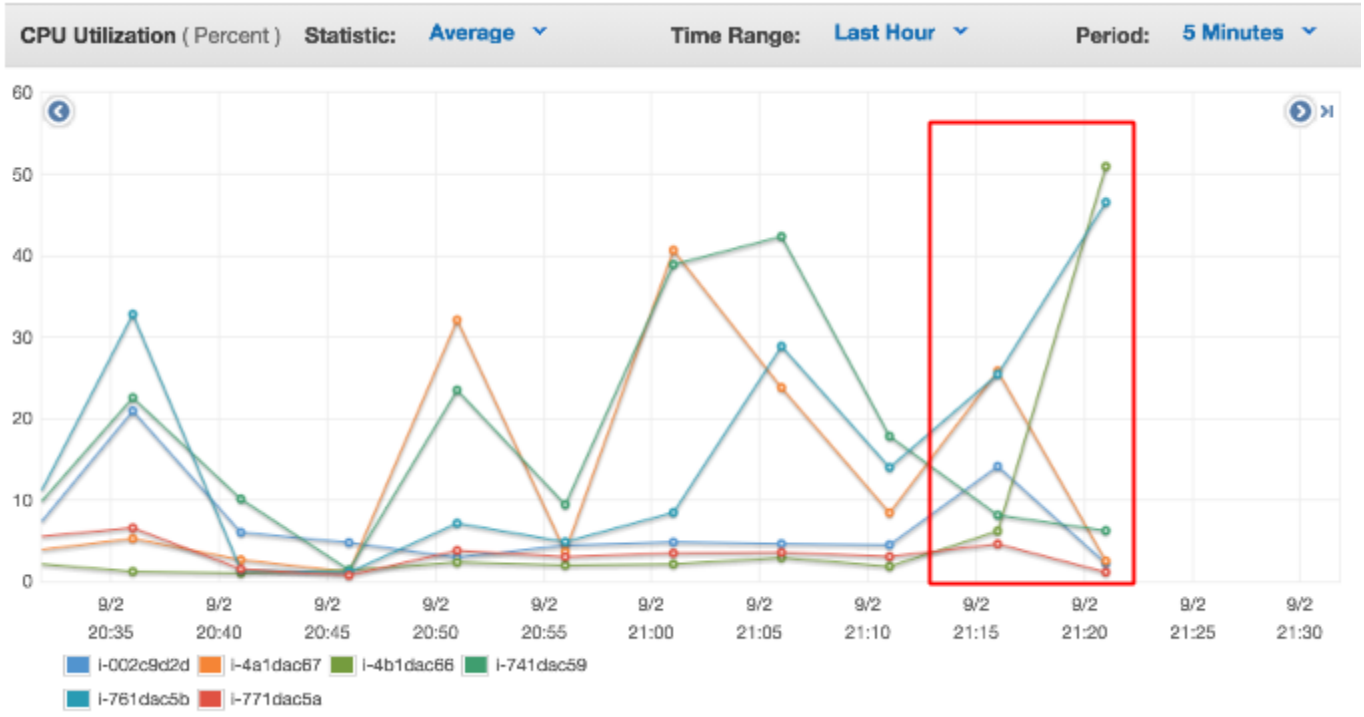
Following performance evaluation metrics will be used to compare Hadoop vs Spark performance of our implementation.

CPU Utilization: Total amount of CPU processing for each iteration of running

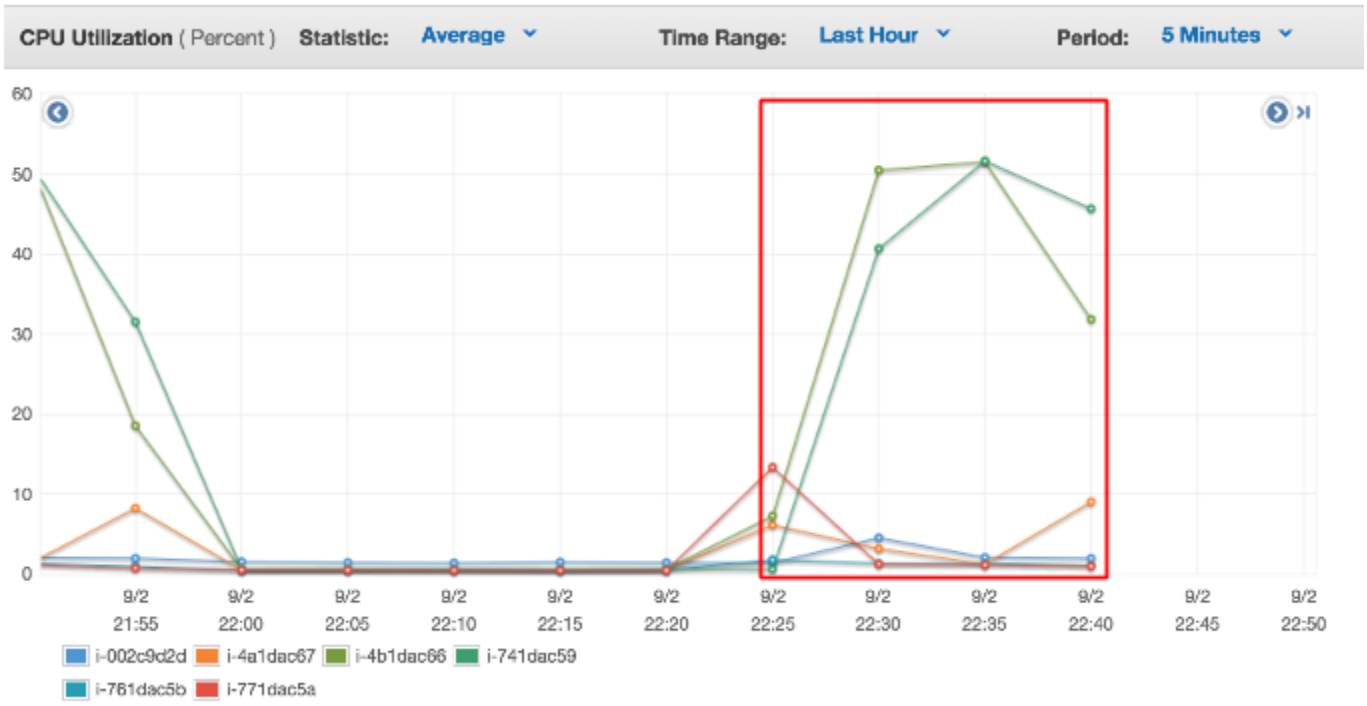
Tachyon - 200Mb training set, Naive Bayes Classifier



Tachyon - 550Mb ImageNet Naive Bayes' Classifier:



Tachyon - 1Gb ImageNet Dataset:



Memory Usage: Amount of memory usage in percentage

Unfortunately, Amazon CloudWatch does not provide this key metric. They do not have plans in the near future to embed this metric within Amazon CloudWatch, as well.[18] As a substitute, we grabbed information from the http. The information doesn't change much when increasing dataset size, however.

[18] https://forums.aws.amazon.com/message.jspa?messageID=265299

Spark Master at spark://ec2-54-85-149-164.compute-1.amazonaws.com:7077

URL: spark://ec2-54-85-149-164.compute-1.amazonaws.com:7077
Workers: 5
Cores: 10 Total, 10 Used
Memory: 31.4 GB Total, 30.0 GB Used
Applications: 1 Running, 8 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

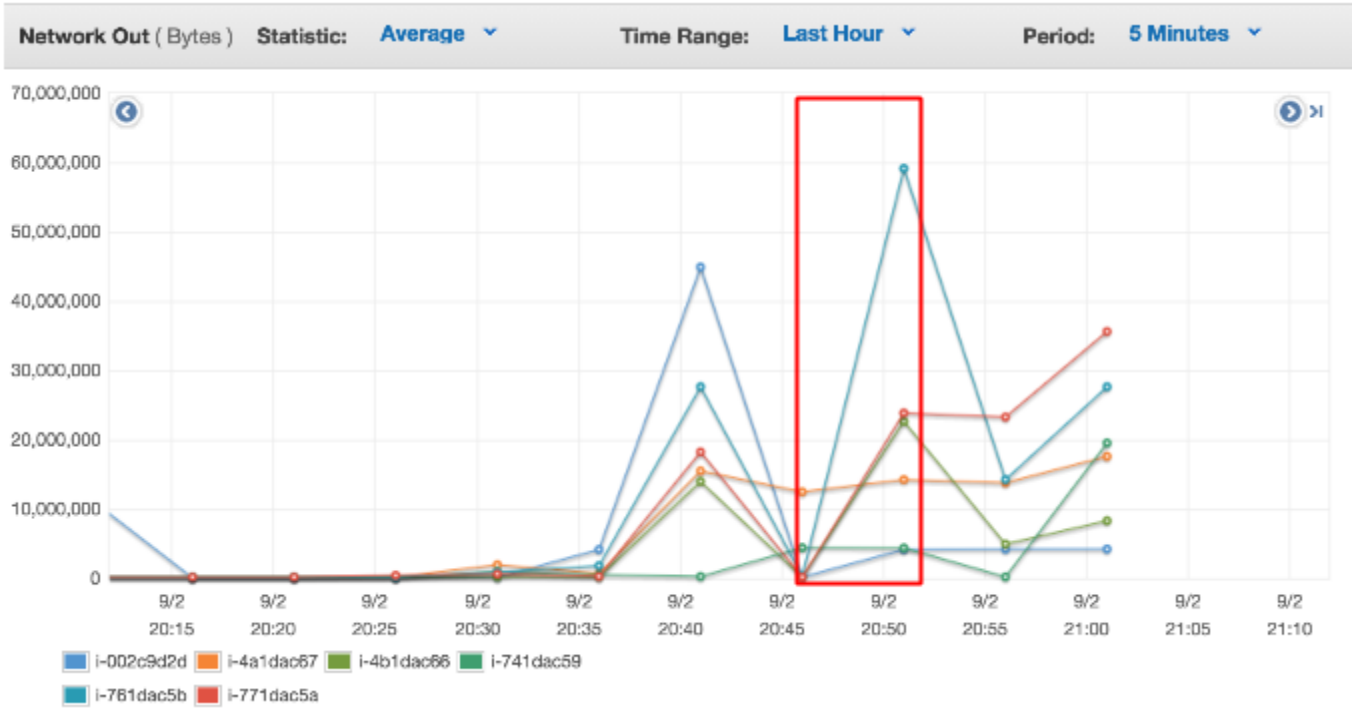
Workers

Id	Address	State	Cores	Memory
worker-20140902200238-ip-172-31-15-122.ec2.internal-52597	ip-172-31-15-122.ec2.internal:52597	ALIVE	2 (2 Used)	6.3 GB (6.0 GB Used)
worker-20140902200238-ip-172-31-15-123.ec2.internal-36830	ip-172-31-15-123.ec2.internal:36830	ALIVE	2 (2 Used)	6.3 GB (6.0 GB Used)
worker-20140902200238-ip-172-31-15-124.ec2.internal-58044	ip-172-31-15-124.ec2.internal:58044	ALIVE	2 (2 Used)	6.3 GB (6.0 GB Used)
worker-20140902200238-ip-172-31-15-125.ec2.internal-60410	ip-172-31-15-125.ec2.internal:60410	ALIVE	2 (2 Used)	6.3 GB (6.0 GB Used)
worker-20140902200238-ip-172-31-15-126.ec2.internal-54495	ip-172-31-15-126.ec2.internal:54495	ALIVE	2 (2 Used)	6.3 GB (6.0 GB Used)

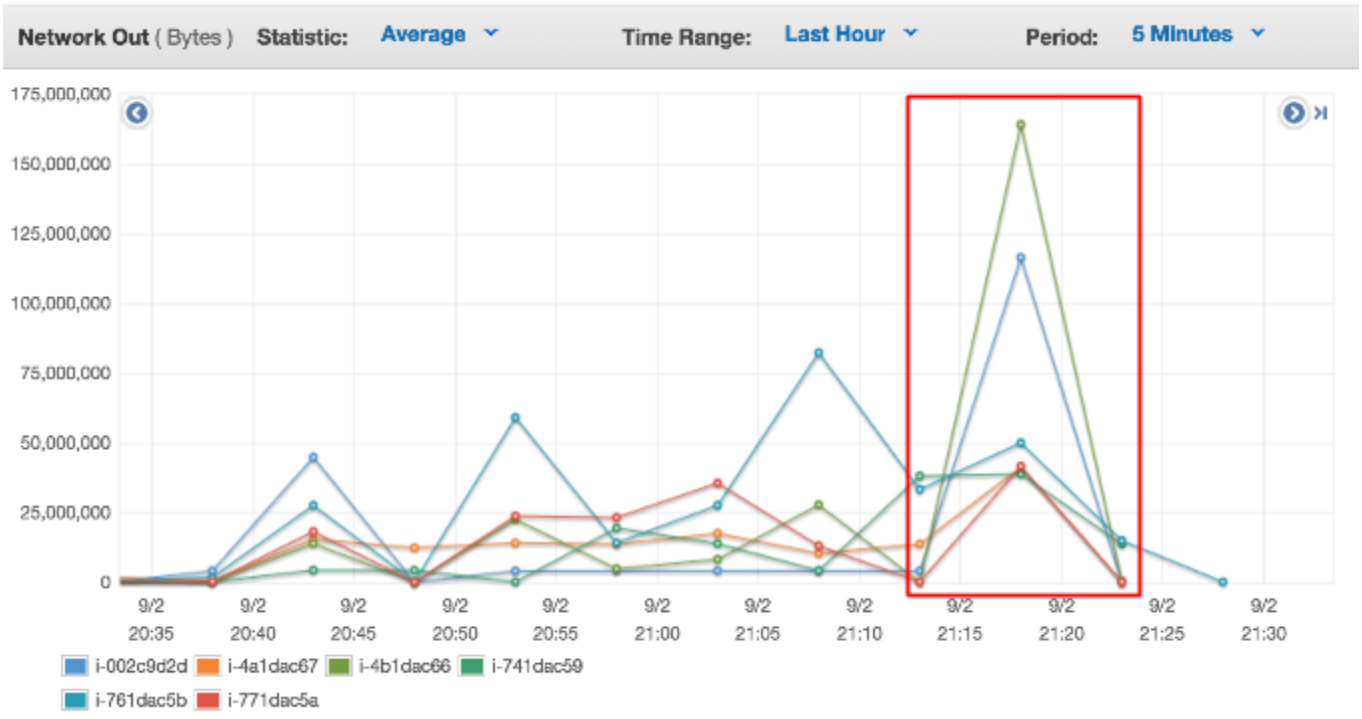
Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duratio
app-20140902210625-0006	NaiveBayes Application	10	6.0 GB	2014/09/02 21:06:25	root	RUNNING	37 s

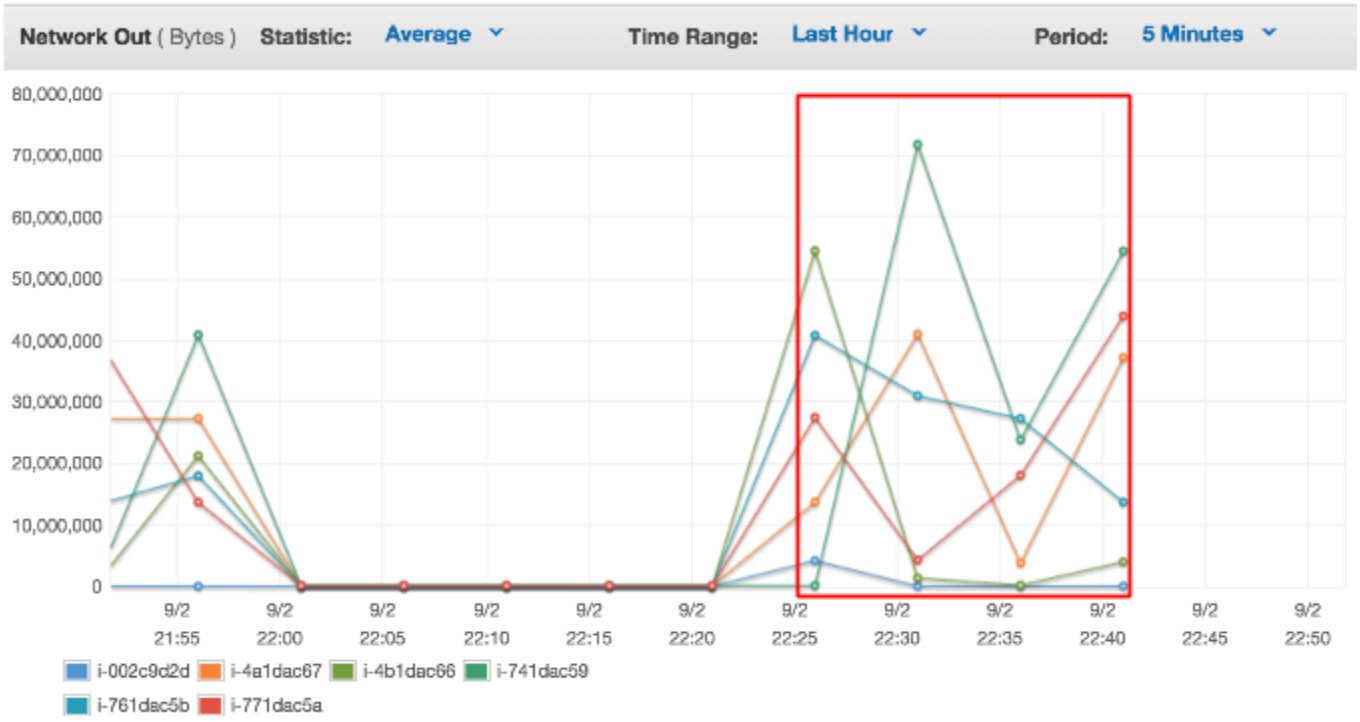
Network Utilization: How much network was congested a given point of time for each systems



Tachyon 550Mb ImageNet Dataset:

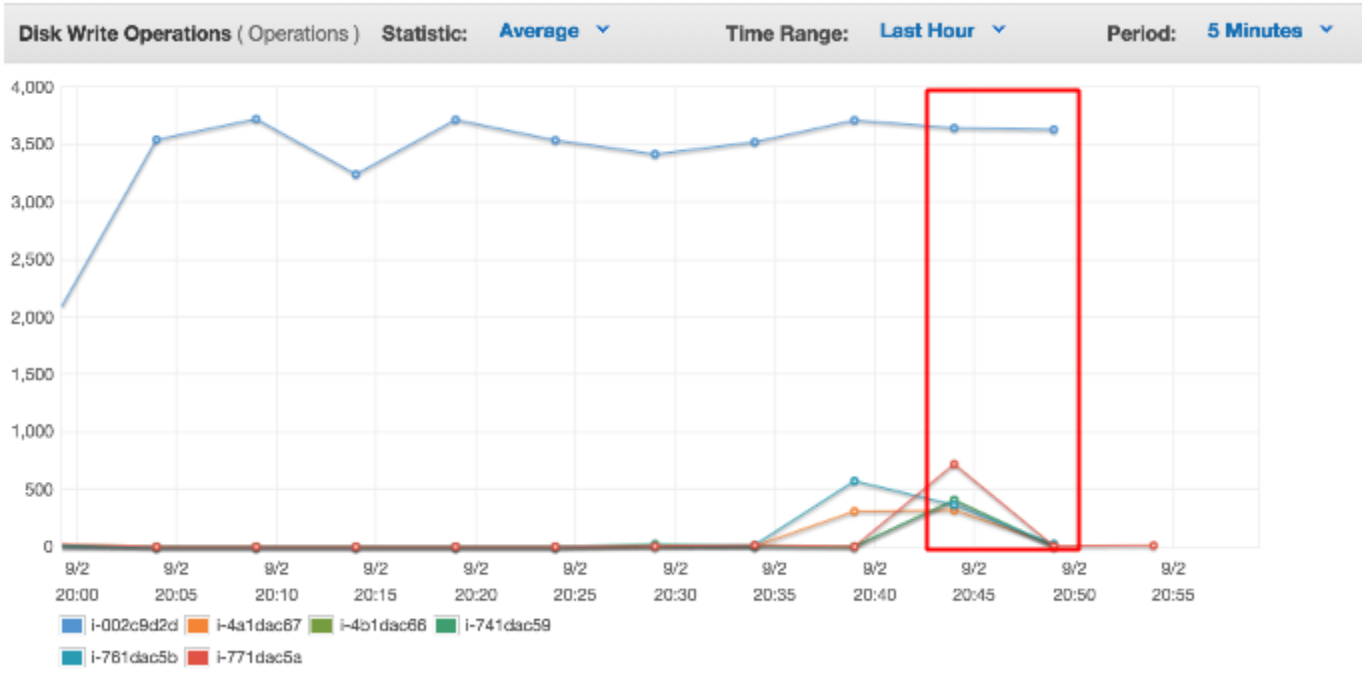


Tachyon 1Gb ImageNet dataset:

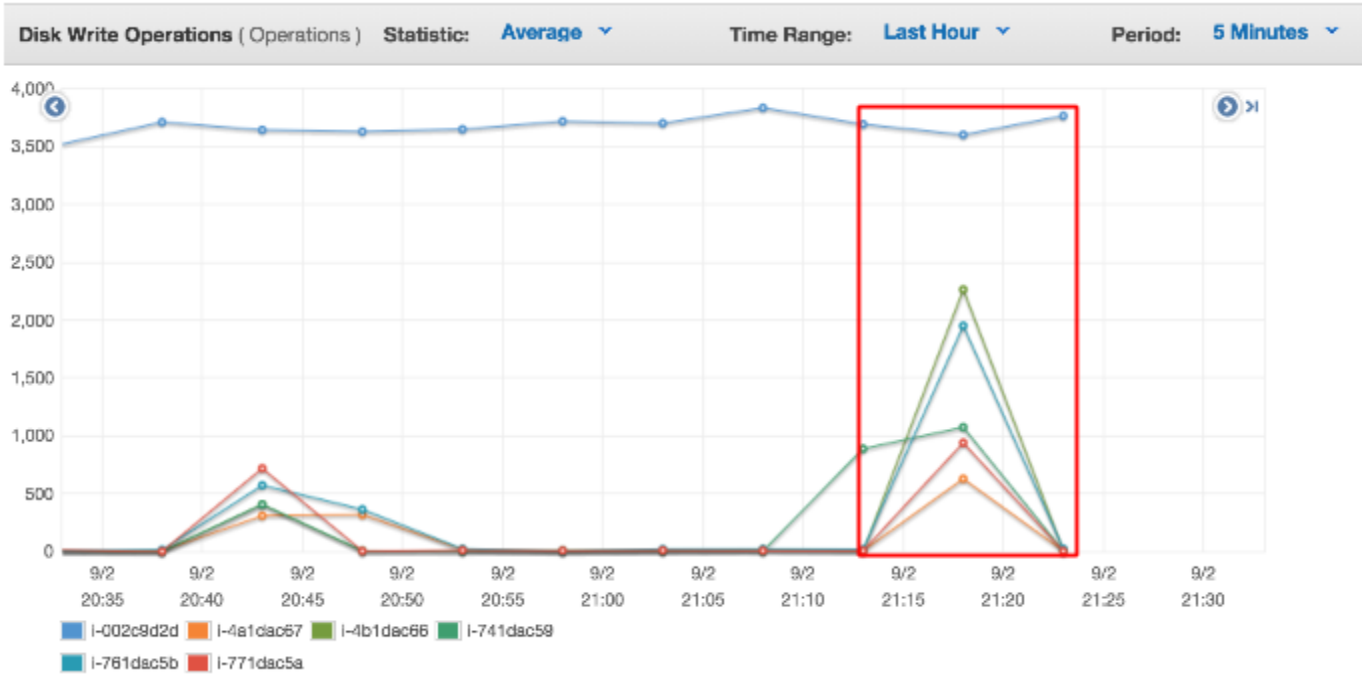


Disk Write Operations:

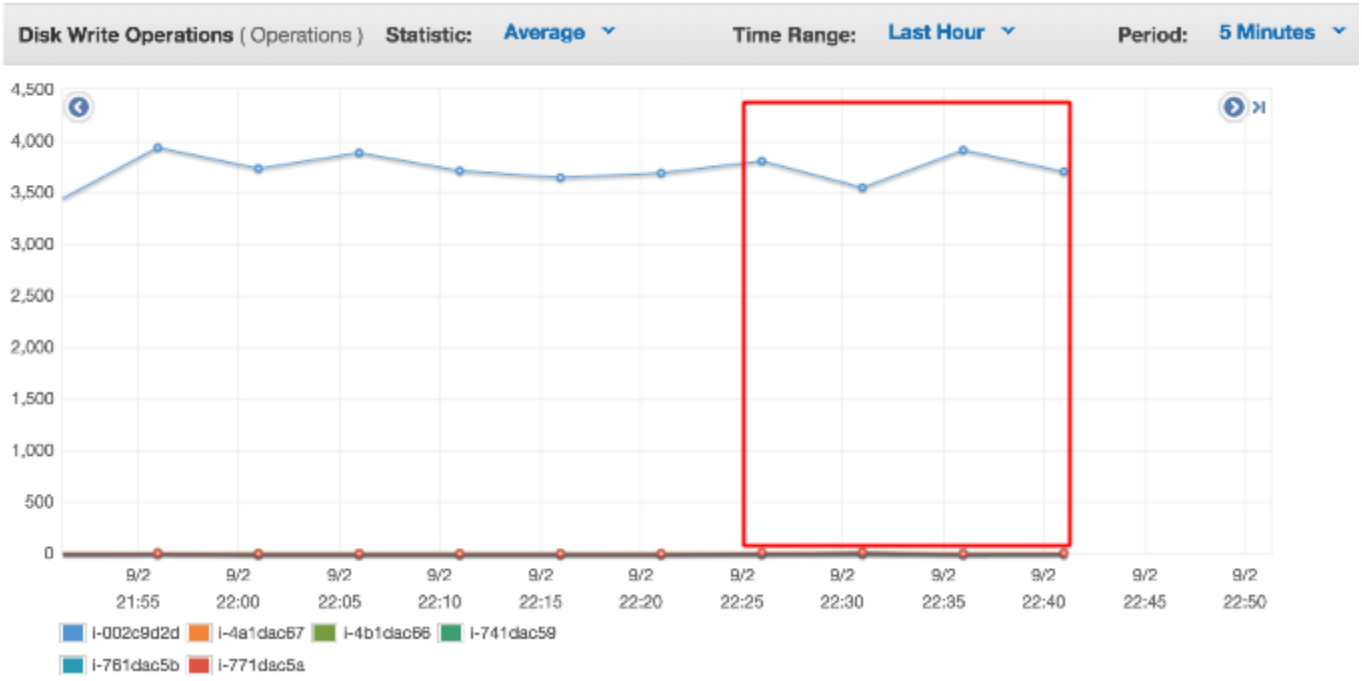
Tachyon 200Mb ImageNet dataset:



Tachyon - 550Mb ImageNet Dataset:



Tachyon - 1Gb ImageNet Dataset:



Running Results:

We ran the algorithm twice to collect run times. Our accuracy is so low, because we were not able to successfully vectorize the text files, per what the algorithm needs (as evidence from a previous project impelementation on Mahout and Hadoop). However, it is better than random blind guess! :D

Dataset Size	Time1 (seconds)	Accuracy1	Time2 (seconds)	Accuracy2
200Mb	211	0.28%	211	0.28%
550Mb	473	0.36%	474	0.36%
1Gb	902	0.42%	894	0.42%

6.3. Abnormal case explanation (the most important task)

An abnormality that is spotted through the analysis of our graphs, is once the master node (blue line) has completed its processing, the worker nodes then start, but not all of the worker nodes are equal in processing and resource utilization. To understand this occurrence, we would need to dive deeper under the hood, unfortunately, we are poor graduate students, so we don't have the funds to pay for more detailed monitoring. Another thought would be if Spark is implemented on the SCU Design Center cluster, we could use Apache monitoring tools to gain deeper insights into how Spark distributes the work between nodes.

6.4. Discussion

It appears for the most part, the variance in resources were not as high as one would imagine. This is good news, as it indicates sizes of datasets can be quite high. The next step for testing would be to implement an intensive iterative algorithm and push the EC2 cluster to the limit. CPU utilization only hit roughly 60% at max. Unfortunately, we were not able to gain any insightful metrics on memory, which is the biggest point of emphasis in a Spark cluster. This really determines the efficiency of the Spark cluster.

7. Conclusion and recommendations

7.1. Summary and conclusion

Spark is still in its infancy and it is shown by the limited amount of algorithms implemented in Spark's machine learning library, MLlib. Other factors that need to change before Spark becomes a preferred method of computation is the mental overhead of MapReduce and the use of Tachyon. Spark is limited by the in-memory processing architecture it is built around. Memory has not taken as many leaps and bounds in-terms of disruptive technologies, like CPU and on-disk storage, so utilization through virtual machines and memory utilization software, such as Tachyon, will be crucial for Spark's continuing growth in both the academic and profiled realms.

7.2. Recommendation for future studies

This project did not go all according to play. The infancy of Spark constricted our ability to build a Naive Bayes' text classification model in a timely manner. If we were given more time, we would have built out the components needed to properly utilize the ImageNet dataset. The following are components of Spark that we feel are important for other researchers to understand when considering implementing applications on Spark, and how they can expect their results to appear.

7.2.1. Tachyon

Utilizing Tachyon would be a great way to configure the in-memory filesystem of HDFS with Spark. This can be handy when memory resources are restrained by the typed of Spark applications and dataset that is being used.

7.2.2. RDD Improvement

In-memory computation is the key to improve performance of Shark or any cluster computing framework. Data available in memory helps to avoid Disk latency and network latency. RDD features that allows data to be persisted in memory for iterative computation helps give better performance numbers. Following are the proposal we have proposed as RDD enhancements to make data available in memory for better performance.

Proposal 1: Pre-fetch next accessible data from disk to memory. RDD Cache size is limited and when data is more than cache space, prefetching can make more data in-memory available for access at the time of data needed.

Algorithm: when data offset X is accessed in RDD, fetch data X+N bytes from disk

files and keep it ready for future use. Take RDD errors example and apply the algorithms.

Proposal 2: RDD meta data in disk: while training a large graph database, classify data area based on attributes and use those attributes to decide whether to keep the data in-memory or not

For example, 1 TB size picture of a galaxy may have different regions, regions with stars, without stars. If meta data of RDD can tag regions with stars, iterative application that use only such regions, need to be kept in-memory, other regions need not be kept in memory.

Proposal 3: A new set of transformation rules such as rdd1.map() will create a new rdd and if rdd1 was persistent in memory the new one also gets the properties of the parent to be in cache memory

7.2.3. Building up MLlib

MLI supported text classification, but once MLlib (stable release) was implemented, the text classification functions were not provided. This causes a huge issue for us, as we do not have the time to develop such an algorithm. For Spark 1.1.x, there will be TF-IDF, TFHashing, Tokenization and text classifications. This is evidence to Spark being a new paradigm. The issue of in-memory resource limitation and certain functions not being built into Spark's Machine Learning Library is evidence of Spark still residing in its infancy. As time goes on, and more contributors develop functions that are then implemented within MLlib, Spark will become a more prominent tool for use in the academic and professional realm.

If we had more time, our project could have focused on building out modules of a text classification algorithm. The algorithm would consist of a textDirectoryLoader, which would be able to take in an input, composed of files, each of which are a class label. Within each class label there would be individual records. A tokenizer would extract the class label and unique id to form a key, while the value would compose of a string of feature vectors. The vectorizer would then compute the term frequency-inverse frequency and create a sparse matrix. Once the sparse matrix has been created, the next step would be to train the Naive Bayes' classifier.

8. Bibliography

8.1. Classifying documents using Naive Bayes on Apache Spark / MLlib

8.2. MLlib: Scalable Machine Learning on Spark

8.3. Spark on EC2

8.4. ImageNet 2010 Competition

8.5. Spark AmpCamp

8.6. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud

8.7. Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark

8.8. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics

8.9. Kendall et all - A Note on Distributed Computing

8.10. On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS

8.11. Ethane: Taking Control of the Enterprise

8.12. **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing** - *Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica University of California, Berkeley*

8.13. **Shark: SQL and Rich Analytics at Scale** - Reynold Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica, AMPLab, EECS, UC Berkeley

8.14. http://spark.apache.org/docs/latest/**programming-guide.html**

9. Appendices

- **sbt file**

```
name := "Naive Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.0.2"

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

- **README**

1. set up EC2 cluster, login to master node
2. chown -R ec2-user /root #this enable sftp
3. ./sbt package #compile .jar file local, upload to master
4. submit script:
    ./spark-submit --class <main object name> --master <local or sc.master> <.jar path>

Script repository:
Launch ec2 cluster

submit application:
./spark/bin/spark-submit --class Naive --master
spark://ec2-54-85-149-164.compute-1.amazonaws.com:7077 /root/naive-project_2.10-1.0.jar

use tachyon:
./tachyon/bin/tachyon tfs copyFromLocal /root/550_MB_Train_and_Test_data.tsv
/data/550_MB_Train_and_Test_data.tsv

local: /Users/lucasshen/Desktop/imagenet/data N/200_MB_Test_dat_tab.tsv

ec2 master: /root/data/200_MB_Test_dat_tab.tsv

S3:
s3n://AKIAIDOQLVVLPIIK5GDA:wbAT9nuLG3FtHXa7dTW7TDag5UdnHXGGoVCmQQPi@lucas-coen241/200_MB_Test_dat_tab.tsv

- scala codes

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.mllib.classification.NaiveBayes
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint

object Naive {
  def main(args: Array[String]) {
    val time_start = System.currentTimeMillis

    // val dataFile =
"s3n://AKIAIDOQLVVLPIIK5GDA:wbAT9nuLG3FtHXa7dTW7TDag5UdnHXGGoVCmQQPi@lucas-coen241/200_MB_Test_dat_tab.tsv"
    val dataFile =
"tachyon://ec2-54-85-149-164.compute-1.amazonaws.com:19998/data/1_GB_Train_and_Test_data.tsv"
    val conf = new SparkConf().setAppName("NaiveBayes Application")
    val sc = new SparkContext(conf)

    val rawdata = sc.textFile(dataFile)

    val classname = rawdata.map(line =>{ val x=line.split("\t"); x(0)}).distinct

    val name_mapping = classname.zipWithIndex().collect.toMap

    // val reverse_mapping = name_mapping.map(_.swap)

    val data= rawdata.map( line=>{
      val triple = line.split("\t"); name_mapping(triple(0)).toString+","+triple(2)})

    val features_per_image:Array[Int] = data.map{ line=> val part = line.split(',');
part(1).split(' ').size}.collect
    val average_feature_quant = (features_per_image.sum /
features_per_image.size).toInt
    val max_feature_quant = features_per_image.max

  //   val parsedData = data.map{ line => //max version
  //     val parts = line.split(',')
  //     val feature_array = parts(1).split(' ').map(_.toDouble)
  //     val feature_quant = feature_array.size

  //     if (feature_quant == max_feature_quant){
  //        LabeledPoint(parts(0).toDouble, Vectors.dense(feature_array))
  //     }
  //     else{
```

```
  //        val diff= max_feature_quant - feature_quant
            // LabeledPoint(parts(0).toDouble, Vectors.dense(feature_array.union(new
Array[Double](diff))))
  //     }
  //   }

   val parsedData = data.map{ line => //average version
      val parts = line.split(',')
      val feature_array = parts(1).split(' ').map(_.toDouble)
      val feature_quant = feature_array.size

      if (feature_quant > average_feature_quant){
            LabeledPoint(parts(0).toDouble,
Vectors.dense(feature_array.slice(0,average_feature_quant)))
      }
      else if (feature_quant < average_feature_quant){
            val diff= average_feature_quant - feature_quant
                LabeledPoint(parts(0).toDouble, Vectors.dense(feature_array.union(new
Array[Double](diff))))
      }
      else{
            LabeledPoint(parts(0).toDouble, Vectors.dense(feature_array))
      }
    }

    /*val parsedData = data.map{ line => //original version
      val parts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split('
').map(_.toDouble).take(features_to_make)))
    }*/

    val splits = parsedData.randomSplit(Array(0.8, 0.2), seed = 11L)
    val training = splits(0)
    val test = splits(1)

    val model = NaiveBayes.train(training, lambda = 1.0)

    val prediction = model.predict(test.map(_.features))

    val predictionAndLabel = prediction.zip(test.map(_.label))
    val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() /
test.count()
    println(accuracy)
    println("Total execution time: "+
((System.currentTimeMillis-time_start)/1000).toString+ " seconds.")
  }
}
```