Apathetic ML

Architecture Independent System for Distributed Machine Learning

Interns

Vishwas Rajashekar	PES1201700704
Aditi Ahuja	PES1201800165
Raunak Sengupta	PES1201700072
Sparsh Temani	PES1201800284

Mentor

Animesh N D	01FB16ECS058

Microsoft Innovation Lab
PES University
Summer Internship
June - July 2019

Abstract

This project aims to parallelize the computations involved in machine learning algorithms, thereby improving their efficiency. It also aims to make the framework architecture-independent by using Kubernetes for both implementation across platforms such as Google Cloud Platform, Azure, AWS etc. or a cluster of machines connected locally. This work explores the effect of parallelism on some algorithms such as Linear and Logistic Regression, K-Means Clustering, Random Forest and Artificial Neural Networks and discusses the effects of external factors such as network delays and the mode of communication used.

Contents

Chapter 1

Introduction

Machine Learning is one of the most prominent buzzwords in the current technical sphere. Given the increasing computational needs of recent models, common computers might not always possess the specialised hardware to train such models. In such cases, distributed computing, can increase the efficiency of such computations using multiple computational units, each dealing with a part of calculations for the model.

1.1 Problem Statement

To build an architecture-independent system for distributed machine learning to improve the efficiency of the algorithms and distribute the workload.

Chapter 2

Literature Survey

The literature Survey included researching existing distributed learning frameworks like Apache Hadoop, Spark and Storm. We identified the issues with each one of them and tried to come up with a solution to solve the issues in the existing frameworks.

2.1 Existing Solutions

2.1.1 Apache Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than relying on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Data locality i.e.map code is executed on the same data node where the data resides, is one of the key advantages of Hadoop Map/Reduce.

All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines, or racks of machines) are common and thus should be automatically handled in software by the framework. Issues with Hadoop:

- Localisation decreases with increasing data nodes and data: Larger

clusters tend not to be complete homogeneous, some nodes are newer and faster than others, bringing the data to compute ratio out of balance. Speculative execution will attempt to use computation power even with non-local data.

- Non-local data processing: The nodes that contain the data in question might be computing something else, leading to another node doing non-local processing. This results in a strain on the network, which poses a problem to scalability. The network becomes the bottleneck. Additionally, the problem is hard to diagnose because it is not easy to see the data locality.

To improve data locality, you need to first detect which of your jobs have a data locality problem or degrade over time. 2

2.1.2 Apache Spark

Apache Spark[5] is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application. Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apache Spark has the following features : Speed Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.

Supports multiple languages Spark provides built-in APIs in Java, Scala, or Python. Spark comes up with 80 high-level operators for interactive querying.

Advanced Analytics Spark supports Map, reduce, SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

Issues with Spark:

- Spark is notoriously difficult to tune and maintain. That means ensuring top performance so that it doesnt buckle under heavy data science workloads is challenging. Jobs failing with out-of-memory errors are

very common and having many concurrent users makes resource management even more challenging.

- Memory errors and errors occurring within user-defined functions can be difficult to track down.

- Distributed systems like Spark are inherently complex. Error messages can be misleading or suppressed, logging from a PySpark User Defined Function (UDF) is difficult and introspection into current processes is not feasible. Creating tests for your UDFs that run locally helps, but sometimes a function that passes local tests fails when running on the cluster. Figuring out the cause in those cases is challenging.

- One of Sparks key value propositions is distributed computation, yet it can be difficult to ensure Spark parallelism computations as much as possible. Spark tries to elastically scale how many executors a job uses based on the jobs needs, but it often fails to scale up on its own.

- Spark divides RDDs (Resilient Distributed Dataset)/DataFrames into partitions, which is the smallest unit of work that an executor takes on. If you set too few partitions, then there may not be enough chunks of work for all the executors to work on. Also, fewer partitions means larger partitions, which can cause executors to run out of memory.

2.1.3 Apache Storm

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for real time processing what Hadoop did for batch processing. Storm is simple and can be used with any programming language. Storm has varied use cases. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate. Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed.

2.2 Orchestration Using Docker Swarm / Kubernetes

2.2.1 Docker Swarm

Docker Swarm is a clustering and scheduling tool for Docker containers. With Swarm, IT administrators and developers can establish and manage a cluster of Docker nodes as a single virtual system. Clustering is an important feature for container technology, because it creates a cooperative group of systems that can provide redundancy, enabling Docker Swarm failover if one or more nodes experience an outage. A Docker Swarm cluster also provides administrators and developers with the ability to add or subtract container iterations as computing demands change.

Swarm uses the Docker CLI to run its programs. Hence, knowledge about only a single set of tools is needed to be learned in order to build environments and configurations. Since the Swarm program runs on your current Docker, you can begin by opting into Swarm.

2.2.2 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Installing Kubernetes is easy when it comes to installing it in a test bed. Though if you plan to run it at scale requires a bit more planning and effort. Specific commands need to be run for every step including:

- Bringing up the cluster
- Defining the environment
- Defining a pod network which enables the containers to interact
- Setting up the dashboard
- Hosting the cluster

Chapter 3

Main Body

3.1 Distributed Computing

3.1.1 Containerization using Docker

A container[1] enables the efficient packaging of an application with its dependencies and requirements for smooth running across platforms. A container is lightweight since it runs natively on Linux and shares the kernel of the host machine with other containers. Docker[2] is a platform used for the smooth development, deployment and running of applications using containers. In case of Docker, a Docker image is an independently executable software package including system settings, libraries and dependencies.

3.1.2 Orchestration using Kubernetes

Container orchestration is used for the arrangement, coordination and management of software clusters. Kubernetes(or K8s)[3] is an open source container orchestration platform to automate deployments, scaling and operations of container clusters. It is also useful for addressing challenges such as load balancing, storage management, cluster health checks and auto-restarting and scaling of nodes.

3.1.3 Stream processing using Apache Kafka

A streaming platform, like a message queue, publishes and subscribes to streams of records with fault tolerant provisions for storage and real-time

processing. Kafka[4] can be used to build real-time data streaming pipelines between systems and applications, which also react to streams of data. Kafka can run of clusters spanning multiple servers across multiple data-centers. Each record(message) in kafka consists of a key, a value and a timestamp. Streams of such records are stored in a topic for a specified retention period. Each topic has a partitioned log with an ordered, immutable sequence of records, each identified by a unique offset.

3.1.4 Share Storage using NFS

NFS(Network File Storage) is a distributed file system standard mainly used to let a user view, store and modify files from multiple disks and directories on a remote server similar to doing the above on the users computer. NFS mounts the file on a remote server, to be accessed with either read-only or read-write privileges. NFS uses RPCs(Remote Procedure Calls) to route communication between clients and servers. NFS is especially useful in computing environments where centralized management of data and resources is necessary. In this project, NFS is being used to share the dataset among the various worker nodes.

3.2 Parallelisation of Tasks

Each implementation has certain common features including:

- NFS for shared data storage of all the datasets.
- Docker images built for each node(i.e. Controller, master, workers and brokers). Certain images were used as a base image for building other images. A local Docker registry was set up for the same.
- Kafka as a message queue was implemented using Kubernetes pod architecture.
- Flask servers running within containers as pods.
- Utilising Kubernetes pod crashing and restarting provisions.

  Some sample code snippets for the above are provided:

- Sample YAML file for NFS

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nfs-server-core
spec:
  # clusterIP: 10.3.240.20
  ports:
    - name: nfs
      port: 2049
    - name: mountd
      port: 20048
    - name: rpcbind
      port: 111
  selector:
    role: nfs-server-core

---
```

Figure 3.1:

- A sample Dockerfile for controller:

```
FROM 192.168.0.10:8080/controller

WORKDIR /app

COPY . /app

EXPOSE 4000 5000 22 873

ENV NAME ApatheticML
```

Figure 3.2:

- Sample kubectl commands for deployment(in a shell script):
  Kubectl apply creates resources for the persistent volumes and their claims from a YAML file.The script further creates a deployment based on number of workers and subworkers. A service is also created based on the number of workers and subworkers(input by the user).

```
kubectl apply -f pvol.yaml && \
kubectl apply -f pvc.yaml && \
kubectl apply -f deploy_creator1.yaml
kubectl apply -f svc_create1.yaml
echo "Setup complete"
~
```

Figure 3.3:

- Sample Kafka python code:

```
    producers=[KafkaProducer(value_serializer=lambda v: dumps(v).encode('utf-8'),bootstrap_servers = ['kafka-s
ervice:9092']) for i in range(workers)]
    KConsumer = KafkaConsumer('w2m',bootstrap_servers=['kafka-service:9092'],group_id="master",auto_offset_res
et='earliest',value_deserializer=lambda x: loads(x.decode('utf-8')))
    topics=[]
```

Figure 3.4:

- Sample YAML file for Kafka broker:

```
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: kafka-broker0
spec:
  template:
    metadata:
      labels:
        app: kafka
        id: "0"
    spec:
      containers:
      - name: kafka
        image: 192.168.0.10:8080/kafka      ##LOCAL
        #image: altariax0x01/kafka          ##CLOUD
        ports:
        - containerPort: 9092
        - containerPort: 4000
        env:
        - name: KAFKA_ADVERTISED_PORT
          value: "9092"
        - name: KAFKA_ADVERTISED_HOST_NAME
          value: kafka-service
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: zoo1:2181
        - name: KAFKA_BROKER_ID
          value: "0"
        command: ["sh","start.sh"]
```

Figure 3.5:

3.2.1 Regression

A regression problem is when the output variable is a real or continuous value, such as salary or weight. Regression consists of repeated matrix operations and as the matrix size increases, so does the time for the operation. Many different models can be used, the simplest is the linear regression. It tries to fit data with the best hyper-plane which goes through the points. This project has implemented both Linear and Logistic Regression models, while incorporating features of distributed computing.

Optimized Distributed Architecture:

A single batch of inputs from the dataset is divided into multiple smaller batches, with matrix operations performed on each. The master sends weights and biases iteratively to each worker, with the messages being stored in particular topics.The producers(in the master) send messages to the topic asynchronously with respect to each other on separate threads. After computation, the workers send gradients iteratively to the master using another kafka topic.The consumer processes run on a single main thread.
Refer to Fig. 3.1 for the distributed architecture.

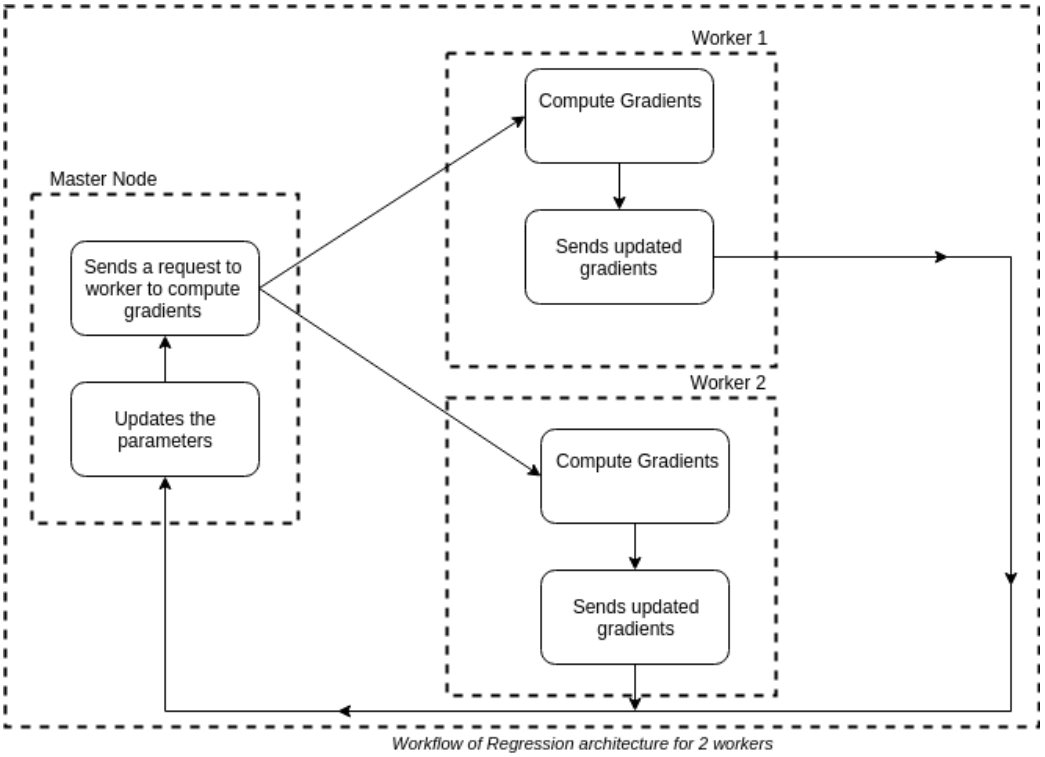

Worker 1
Compute Gradients
Sends updated gradients
Master Node
Sends a request to worker to compute gradients
Updates the parameters
Worker 2
Compute Gradients
Sends updated gradients
Workflow of Regression architecture for 2 workers

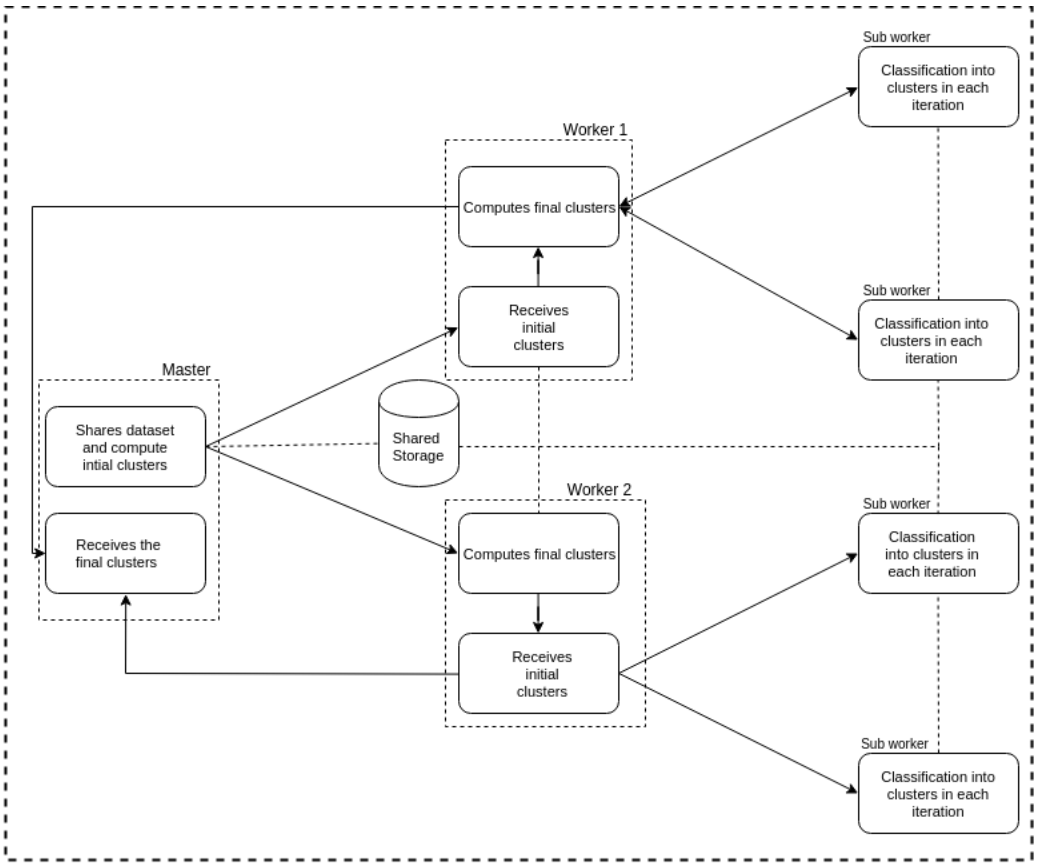
Figure 3.6:

3.2.2 Clustering

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them. This project has implemented K-Means Clustering,in which the majority of operations involve computing distances between data points. This algorithm can be implemented using this methodology since the computation of the distances is independent of each other hence the dataset can be split and distributed among the workers.

Optimized Distributed Architecture:

The master sends a split dataset and centroids to each worker using separate kafka topics for each worker.The producers(in the master) send messages to the topic asynchronously with respect to each other on separate threads. Each worker then sends a part of the split dataset to the subworkers using multiple topics. The subworkers return the cluster of the part of the dataset they receive in a topic. The workers then return clusters and errors to the master using another topic.The consumer processes run on a single main thread.
Refer to Fig. 3.2 for the distributed architecture.



Workflow of Clustering architecture for 2 workers each having 2 sub workers

Figure 3.7:

3.2.3 Tree Based Classifier

Random forests or random decision forests are an ensemble learning methods for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

Random Forest classifier consists of several number of trees specified by the user. This algorithm can be implemented in an optimal, distributed way since these trees are independent of each other and each takes a separate part of the dataset.

Optimized Distributed Architecture:

A bootstrapped dataset is generated at the master and sent to the worker nodes routed through separate Kafka topics.The producers(in the master) send messages to the topic asynchronously with respect to each other on separate threads. The worker nodes compute a decision tree for each of the bootstrapped datasets.These trees are then sent to a common Kafka topic, which is consumed by the master. These trees are then combined to give the resulting random forest model.The consumer processes(from worker to master) run on a single main thread.
Refer to Fig. 3.3 for the distributed architecture.

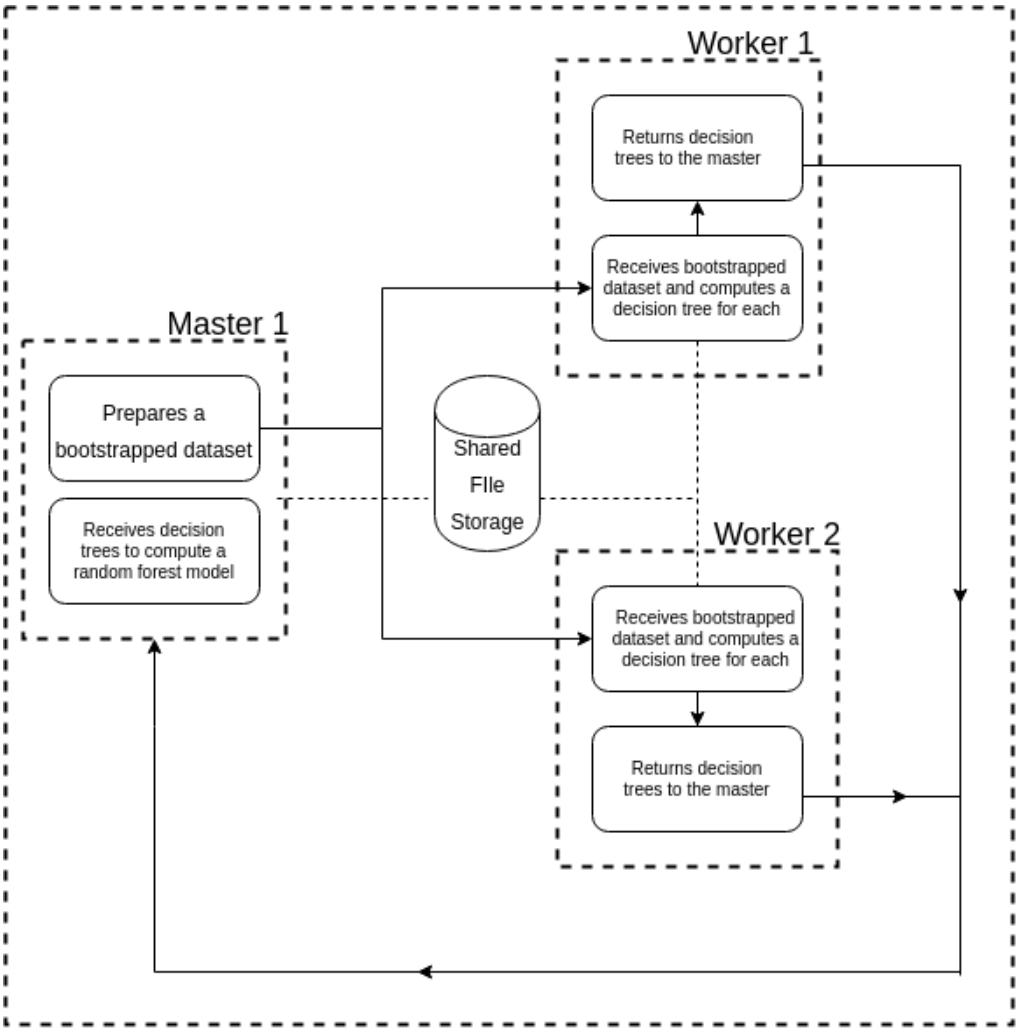

Worker 1
Returns decision trees to the master
Receives bootstrapped dataset and computes a decision tree for each
Master 1
Prepares a bootstrapped dataset
Receives decision trees to compute a random forest model
Shared FIle Storage
Worker 2
Receives bootstrapped dataset and computes a decision tree for each
Returns decision trees to the master
Workflow of Random Forest architecture for 2 workers

Figure 3.8:

3.2.4 Artificial Neural Networks

Artificial neural networks (ANN) or connectionist systems are computing systems that are inspired by, but not necessarily identical to, the biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge about cats, for example, that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the learning material that they process.

Each epoch consists of repeated matrix operations and as the matrix size increases, so does the time for the operation.

Optimized Distributed Architecture:

The master node sends part of the split dataset to the workers using different Kafka topics. The producers(in the master) send messages to the topic asynchronously with respect to each other on separate threads. Matrix operations are performed on a smaller section of these inputs. The workers compute losses and gradients and route them to a common topic to the master. The results of the aforementioned matrix operations are combined at the master.The consumer processes(from worker to master) run on a single main thread.
Refer to Fig. 3.4 for the distributed architecture.

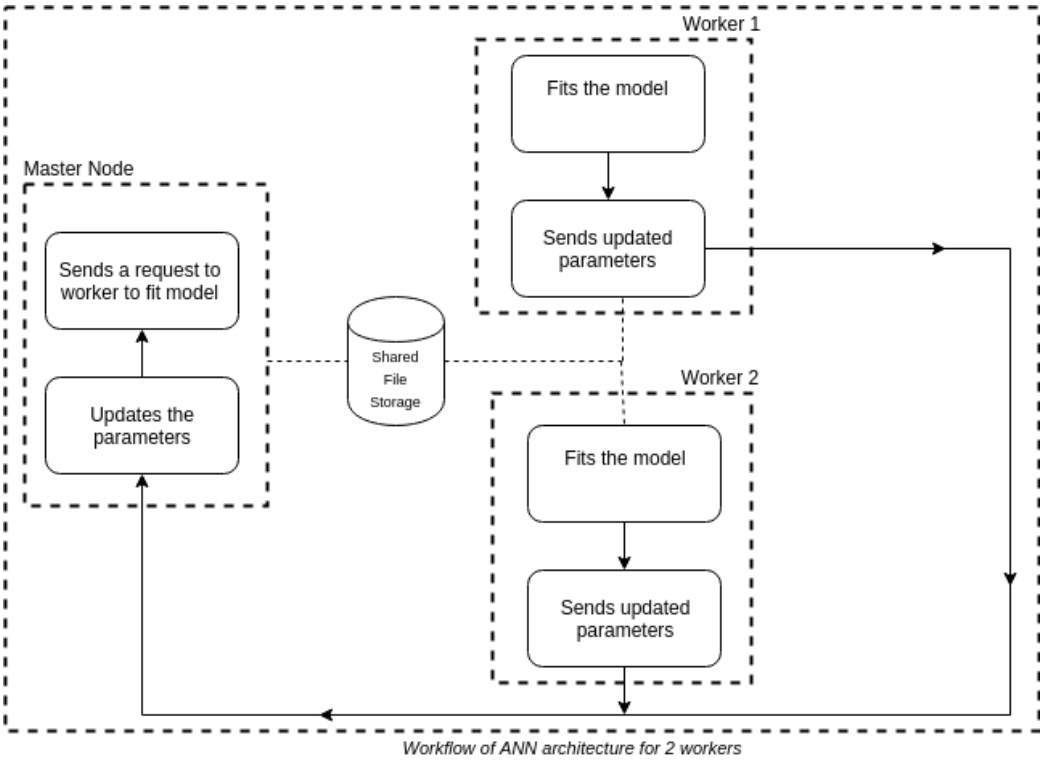

Worker 1
Fits the model
Sends updated parameters
Master Node
Sends a request to worker to fit model
Updates the parameters
Shared File Storage
Worker 2
Fits the model
Sends updated parameters
Workflow of ANN architecture for 2 workers

Figure 3.9:

3.3 Progress Report

3.3.1 Week 1

Week 1 dealt with gathering information about existing Parallel Computing systems. In particular, the popular Apache Hadoop and its MapReduce, Apache Storm for its Real Time processing and Apache Spark for its generalized approach were explored and the pros and cons of each were compared in order to identify the requirements for the project as a whole.

The team familiarized itself with the basics of Docker usage and containerization including:

- Pulling from DockerHub and running hello world and alpine images, using DockerFiles.
- Incorporating persistent storage.
- Running Flask servers and API calls in containers.
- Implementing a local swarm on laptops to understand the workings of Load Balanced, Round-Robin networks using Docker Swarm for Distributed Computing.

The week ended with an introduction to Kubernetes and an assessment of whether the team requirements for orchestration would be fulfilled by Kubernetes.

3.3.2 Week 2

The project was henceforth divided into two sub-teams dealing with Distributed computing and Machine Learning. The Distributed Computing team

- Delved briefly into the key features of Kubernetes and learning about their applications through their documentation.
- Experimented with GCP (Google Cloud Platform) and ran the servers on cloud VMs in a Kubernetes managed cluster.
- Attempted to utilize features such as Persistent Storage and NFS storage for sharing data between pods.

- Developing and deploying simple Flask servers that could communicate with each other, exploring the working of a simple DNS system.
- Produced sample bash scripts and sample YAML files that are used to create and deploy the cluster to the cloud.
- Started building an interactive terminal user interface for users.

The Machine Learning team

- Started off with reading a few articles on how parallelism is done on existing platforms and decided to start with data parallelism on linear regression to check for improvement in accuracy of the algorithm with the number of workers.
- The linear regression algorithm implemented included repetitively splitting a single batch into multiple smaller batches depending on the number of workers chosen by the user and finally combining the gradients to update the actual parameters.
- Began experiments to run Linear Regression in parallel mode on the cluster by the end of the week.
- Implemented the same technique for logistic regression at the end of the week.

3.3.3 Week 3

The Distributed Computing team:

- Deployed a parallelised version of Linear Regression with the master-worker architecture.
- Learnt and experimented with HDFS.
- Wrote simple scripts to interface with HDFS. However, some problems lead the team to preferring to use NFS.
- Created YAML files that would be used to allocate the PersistentVolume and PersistentVolumeClaims that pods need to be able to read/write into the storage.

- Devised a way for the pod to connect to a controller pod to get any data it needs in case of pod failure.
- The above will also ultimately lead to implementation of the checkpointing feature.
- Implemented a client side bash script that uses rsync and the kubectl commands to push a dataset to the cluster for use.
- Extended the terminal user interface to run basic scripts and execute a makefile(with pulumi functionalities) to create a cluster on GCP (Azure and AWS as well).

The Machine Learning team:

- Implemented parallelism for K means by splitting the independent steps among the worker nodes and thus decreasing the overall time taken for the classification.
- The tasks were parallelised by splitting the task of using different set of centroids among the workers and then using the one with the best accuracy / least loss as our final model.
- Came up with an idea for data parallelism, which was later implemented in week 6.

3.3.4 Week 4

The Distributed Computing team:

- Implemented a script that is capable of splitting a CSV file into sections based on the number of lines in the file and the number of workers.
- The system was given the ability to allocate the split files to workers and re-allocate them if necessary (such as when a pod crashes and reboots).
- Re-configured the scripts to allow for the user to choose the number of workers on execution (we had fixed the number of workers to 2 for testing purposes).

- Began collecting data regarding execution time and accuracy of the algorithms, observing some unexpected results due to network delays and serialization.

The Machine Learning team:

- Focussed on algorithm implementation this week.
- Implemented Linear Regression, Random Forest and K-Means Clustering while allowing worker pods to pull data sets for themselves in order to reduce network transfer delays.
- Implemented model parallelism for Random forest using model parallelism by distributing the independent tress among the worker nodes.
- These workers would first create a bootstrapped dataset which made each one of them unique since the creation of this dataset was randomized and each of these from the dataset was fed to a decision tree classifier.Finally, these trees were saved and combined at the master node, which then used all of them at once to predict the class of a given input.

The teams also worked on providing the functionality of allowing users to provide code for execution, provided it adhered to certain constraints in terms of program layout. In specific, this was implemented with K-Means Clustering code for versions implemented from scratch as well as involving standard libraries such as sklearn.

3.3.5 Week 5-6

The Machine Learning team:

- Implemented Neural Networks, which is the highlight of our project.
- It basically splits the dataset among the different nodes of a network and then at the end of each iteration(iteration period decided by the user), the network combines all the models and sends the updated model to the remaining nodes.
- Implemented data parallelism for K Means, where each datapoint in the dataset was distributed further to several sub worker nodes and each one of them would classify them into their respective clusters.

- In the end, the worker nodes would combine this result and find the new means and updated clusters means, repeat the process till the means remain constant and send the best model generated to the master node to use it for prediction.

3.3.6 Week 7

- Completed literature survey for kafka and wrote kafka backend code for use in algorithms.
- Transitioned to using Kafka from flask as a means of communicating between master and workers.
- Deployed regression algorithms and Random Forestusing the latest code and the results were observed.

3.3.7 Week 8

- Integrated the kafka broker server into existing code in place of Flask.
- The Distributed Computing team:
- Started final deployment of algorithms on local cluster, fine-tuning it with the Machine Learning teams inputs.

Chapter 4

Results and Discussion

Linear Regression demonstrates a smooth decrease in Test Loss with a rise in the number of workers. This is due to the efficacy of Mini-Batch Stochastic Gradient in working with a Data Parallel Approach.

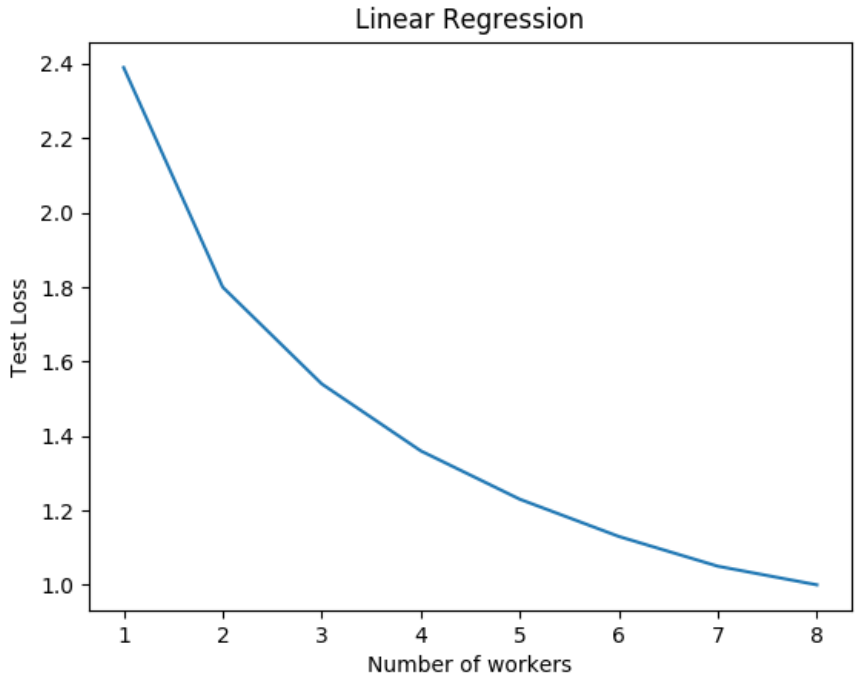


Figure 4.1:

Random Forest Classification shows drastic speedup with the increase in the number of workers. Model Parallelism ensures that parts of the computation are effectively distributed across the workers.

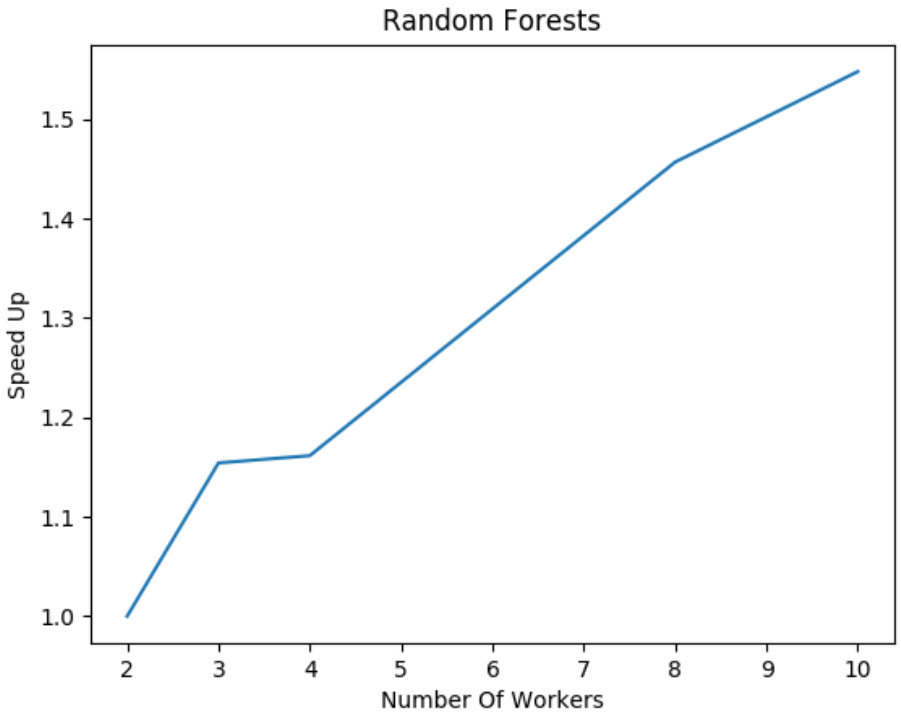


Figure 4.2:

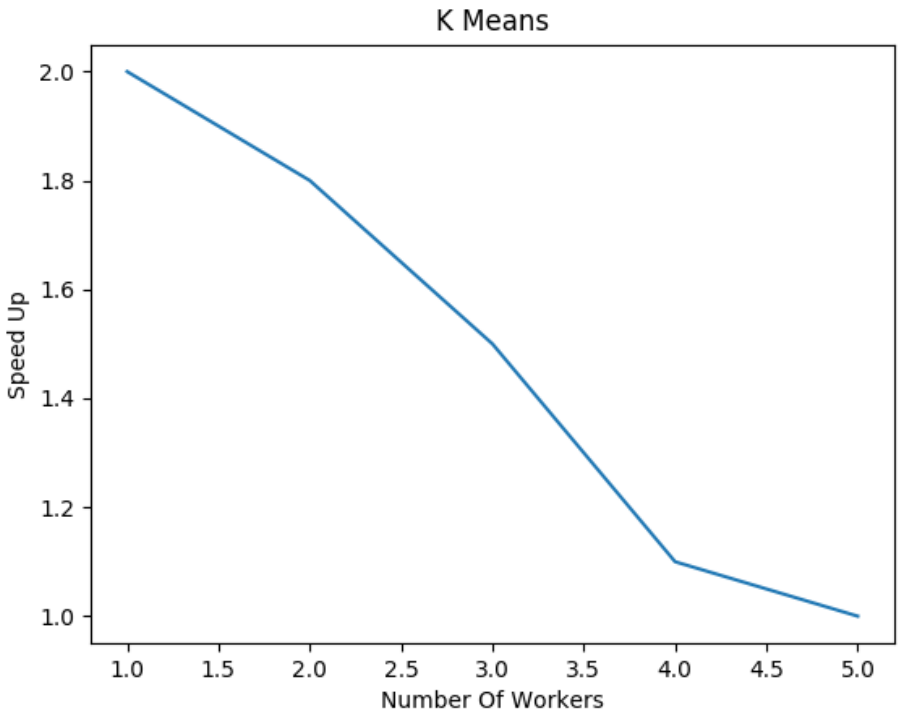

K Means
Speed Up
2.0
1.8
1.6
1.4
1.2
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Number Of Workers

Figure 4.3:

Chapter 5

Conclusion and Future work

Future work on this project beyond the scope of the internship includes

- Increased optimization in the model:aiming for consistent behaviour across various configurations and datasets, with minimal to no changes to achieve the same results.
- Increased flexibility in our models: aiming for a greater degree of universality in terms of code run on the framework. This is specifically for custom code given by the user, which should be run with minimal constraints, if at all.
- Implementation of Checkpointing: replication and ultimately shared checkpointing are being considered in case of node failure and/or data loss.
- Improving architecture of parallelised implementations: aiming to reduce Network Communication as far as possible, and increase independence among systems on the cluster.
- Adding further models: creating a platform for every Machine Learning Application possible.

Appendix A

References

1. https://www.docker.com/resources/what-container
2. https://docs.docker.com/get-started/
3. https://kubernetes.io/docs/tutorials/
4. https://kafka.apache.org/intro.html
5. https://spark.apache.org/mllib/