The University of Melbourne

COMP90019 Distributed Computing Project (25 pt)

Conventional Research Project

Developing Incremental learning on a Distributed Streaming Platform

Author
Ahmed Fahmin
ID : 926184
Mohammad Nafis Ul Islam
ID : 926190

Supervisor
Professor Shanika Karunasekera

Semester 2, 2018

Abstract

Microblogging websites are a common platform where people all around the world can share their opinions and views on various issues and topics. Here we proposed to analyze geolocation of Twitter user at the time of uploading the post based on the content of the message. Correctly predicting geolocation of a tweet is useful for a wide range of applications. Since a considerable number of tweets are generated every second, we need a robust system that can process these tweets in real time. Here using batch processing has its limitations; therefore we are using Apache Storm, a real-time and fault-tolerant system. In the last few years, the rise of popularity of machine learning especially deep learning is incredible in both academic and industry. However, writing a deep learning algorithm from scratch is probably beyond the skill set of most people want to explore the field. It is much more efficient to utilize the tremendous resources available in different deep learning toolkit. Therefore we used Deeplearing4j with our streaming platform to build a distributed neural network. We carefully looked at the issues regarding high commutations of building a neural network and use Apache Spark for task distribution. As the data is available gradually over time, we are following an incremental machine learning approach to find whether a tweet is from Melbourne or not. Our model found an accuracy of 91.43% of correctly predicting the tweets from Melbourne while testing in real time.

Academic Declaration

We certify that:

- *this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.*
- *where necessary we have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School.*
- *the thesis is* ~ 10300 *words in length (excluding text in images, bibliographies and appendices).*

Acknowledgements

We would like to thank our supervisor – Professor Shanika Karunasekera – for her encouragement, ideas, support and eagerness that permeates this work and manifests itself in the successful completion of this scholarly endeavor throughout the whole semester. We also want to thank Dr. Aaron Harwood for providing us the initial dataset, which helped us to test the model.

Contents

List of Figures

List of Tables

1 Introduction

Geolocation of social media users is fundamental in applications varying from rapid disaster response [2–4], recommendation system [5–7] and opinion analysis [8, 9]. Social media platform like twitter allows users to manually input their location while posting or automatically with GPS-based geotagging. However, the user input based locations are noisy and only 1-3% twitter posts are associated with geolocation [10, 11], meaning the geolocation needs to extract from other information such text of the post, network relationships and previous behavior of the user.

As it is well established from previous research [10, 12, 13], it is acceptable to consider that user posts in social media follow their geospatial background, since lexical priors differ from region to region. For example, a user in Melbourne is much more likely to talk about *AFL* and *Australian* compare to a user in New York or Beijing. That is not to say that those terms are exclusively correlated with Melbourne, of course: *AFL* could be used by a user outside of the Melbourne to discuss something relating to the Australian Football League. However, the use of a range of such terms with high relative frequency is strongly indicative of the fact that a user is located in Melbourne.

A large portion of the previous research on user geolocation has focused on either on text-based classification approaches [12, 14–16] or, to a lesser extent, network-based regression approaches [17–19]. However, methods combining the two are rare.

Incremental learning is a method of machine learning, in which the input the data is continuously used to train the model to extend the existing model's knowledge. It represents a dynamic technique of both supervised and unsupervised learning which can be applied when training data becomes accessible gradually over time, or its size is out of system memory limits. Many traditional machine learning algorithms naturally support incremental learning, other needs some modification to adopt this technique.

Deeplearning4j (DL4J) is an open source distributed deep learning programming library written in Java with a Scala API, offering massive support for different types of neural networks (like CNN, RNN, RNTN, or LTSM) [20]. But there are parts of deep learning that are computationally heavy. Distributing these processes of deep learning algorithm may be the solution to this problem. Our interest in DL4J comes explicitly for the reason that, it has pre-build interface[1] to work with Apache Spark ensuring parallel computation with improved scalability. Apache Spark is a framework for distributing computations in a cluster in a secure and declarative way. In recent years, It has become a standard across industries. Currently, DL4J does not support Apache Storm; these shortcomings primarily attracted us to build an environment to work with Apache Storm and DL4J in the same architecture.

The fundamental *objective* of this research project is to incorporate some supervised learning method into the stream processing unit. We intended to use Apache Storm as the distributed stream processing unit. The primary focus of this project is to build an application that will continuously learn from twitter data and predict geo-location for a given tweet. We also provide a comparison among different system parameters while predicting the tweets from Melbourne.

[1] https://deeplearning4j.org/spark

Our work covers: (i) Using Apache storm to read real-time tweets and filter out Melbourne tweets, (ii) Using Deeplearning4j as machine learning library, to build the distributed incremental learning model with Apache Spark, (iii) Discuss and compare related literature regarding geo-location of twitter data, streaming platform, and machine learning frameworks , and (iv) evaluate the performance of the model, varying different parameters of it.

The remainder of the work is organized as follows. In Section 2, we briefly explain the related works in the field of geo-location prediction using twitter data, Apache Storm and Deeplearning4j. We also explain our motivation towards this project. In Section 3 and Section 4, we introduce two key technologies used in the project: Apache Strom and Deeplearning4j respectively. In Section 5 discuss the characteristics of the learning model. We present our proposed model in Section 6 and discuss the initial dataset, data processing and feature extraction in Section 7. In Section 8 and Section 9, we discuss the evaluation metrics and explain how we select the optimal parameters for our learner respectively. We present our findings in Section 10. In Section 11, we discuss the challenges and possible future works of this project. Finally, Section 12 contains the concluding remarks.

2 Related Work and Motivation

Apache Storm was mainly developed to support real time data processing (in contrast with the batch processing). Storm project was made open source by twitter[2]. Van der Veen et al., [21] explained that as a stream processing platform, resource allocation in Apache storm is a very challenging task since the volume and velocity of input data will vary over time. They proposed a tool that will continuously monitor different aspects of the storm platform and describe whether the more servers are needed to add or cut short from existing ones. For Apache storm, task migration happens when distributed data processing systems scale in real-time. Yang et al., [22] proposed three task migration methods, such as worker level, executor level, and executor level with reliable messaging. They showed that their methods significantly reduce the performance degradation and the number of task failures during each migration comparing to Storm's core migration execution.

Han et al., [16] applied a text-based geolocation prediction for Twitter users. They used the content of the tweets and profile information of the user to infer the primary city-level location. They also discussed how users' tweeting behavior affects geolocation prediction, and showed information gain ratio-based approach [23] outperforms other methods regarding accuracy. This work as a successor of their previous works [24], where they implemented a system that automatically identifies "location indicative words" (LIWs), that is words that implicitly or explicitly encode an association with a particular location. Jurgens et al., [25] conducted a comparison among nine geolocation inference techniques (all published in top-tier conferences) on the same dataset and set up close of the real-world environment. They also showed while predicting geo-location of future posts using old dataset, accuracy rates only decrease slightly, but the percentage of posts for which a geo-inference method can predict a location was estimated

[2]https://whatis.techtarget.com/definition/Apache-Storm

to drop by half after a four-month period.

Other than being a source for determining geolocation or sentiment, Twitter data has been used to forecast the political orientation of people efficiently [26]. Tumasjan et al., [27] considered thousands of Twitter posts containing mentions of a political party or a politician's name for German federal election in 2009. They showed that Twitter post could be used as a valuable measure of political deliberation. Twitter data have also been recognized to be a useful tool to predict the stock market. Bollen et al., [28] studied Twitter posts with mood indication to find correlations with the economy and stock market changes.

As for using geolocation features from microblog services like Twitter, most work to our knowledge has focused on helping with ongoing public safety or public health concerns. Achrekar et al., [29] presented the Social Network-Enabled Flu Trends (SNEFT) framework, which monitors messages posted on Twitter with a mention of flu indicators and predicts flu trends. Likewise, Singh et al., [30] use analysis of microblogging services to improve the performance of a Swine Flu monitoring application. Sakaki et al., [31] proposed an algorithm to monitor tweets and to detect the real-time occurrence of earthquakes. Therefore we can say with correct geolocation information is helpful to a wide range of applications.

Toshniwal et al., [32] first explained the use of Apache Storm at Twitter [3] in the year 2014. It was one of the first pieces of literature explaining the architecture of storm at a real-time system. From the then, Storm is being used to run various critical computations in Twitter at scale, and in real-time. They described how queries are executed in Storm and presented some crucial arguments on running Storm at Twitter. They also showed some results regarding the resilience of Storm while dealing with machine failures. Raina et al., [33] performed sentiment analysis on twitter data using Storm. They also addressed the limitations of batch-processing while running the same problem. Besides, there have been other works of Apache storm using twitter data [34–37] for real-time big-data analysis.

Kovalev et al., [38] presented a comparison among popular Deep Learning frameworks including Theano (with Keras wrapper), Torch, Caffe, Tensorflow, and Deeplearning4J based on speed and accuracy. They showed Deeplearning4J was the slowest in network training and prediction. Besides, it had unexpectedly dropped in the accuracy with increasing number of neurons. Bahrampour et al., [39] also presented a similar work of a Comparative Study of Deep Learning Software Frameworks. While comparing to different deep learning frameworks, Erickson et al., [40] emphasized the support of multiple GPUs for performance improvement for Deeplearning4J. They pointed out that Deeplearning4J is not heavily used in medical imaging, and use appears to be declining in the medical field.

On the other hand, Apache spark, which mainly supports micro-batch processing, has integration with deep learning methods, using DeepLearning4J. This has improved its scalability significantly. Many works [41–44] have been done on this platform as well.

In recent time, incremental and online learning gained more attention especially in the field of big data and learning from data streams, comparing with the traditional assumption of complete data availability. We can encounter ambiguity regarding the definition of incremental

[3]https://twitter.com/

and online learning in literature. Losing et al. [45] analyzed the critical properties of seven incremental methods such as Incremental Support Vector Machine (ISVM), On-line Random Forest (ORF), Gaussian Naive Bayes (GNB), etc. They analyzed and suggested in what scenario we should select which incremental method. Carpenter et al. [46] introduced a neural network architecture for incremental supervised learning which can recognize categories and work with multidimensional maps. Ross et al. [47] applied incremental learning on visual tracking and object detection. Joshi et al. [48] presented a survey on different incremental learning methods and their domains and highlighted their potentials in the aspect of decision making.

But currently, there is no support for incremental learning with Deeplearning4j during data stream processing with Apache Storm. This limitation motivates us to dig down to Apache storm and deeplearning4j technology while considering the Apache Spark and deeplearning4j collaboration as a baseline model.

3 Basic Apache storm network architecture

Apache Storm is a distributed, open-source, real-time fault-tolerant and distributed Stream Processing Engine (SPE). It works for real-time data just as Hadoop works for batch processing of data (In Batch processing, data is divided into batches, and each batch is processed. It is opposite of real-time processing) [49]. Apache Storm is designed to be:

1. **Scalable:** The operating team must have the ease to add or remove nodes from the Storm cluster without interrupting the actual data flows. Besides, it is difficult to predict the size of a real-time system from the beginning.

2. **Resilient:** A Storm application is supposed to run a large cluster most of the time where hardware components can fail, so Fault-tolerance is crucial to Storm. The Storm cluster must continue processing existing topologies with a minimal performance impact.

3. **Efficient:** As the application is running real time, it is essential to act rapidly. Storm uses a number of techniques to achieve the goal, including including keeping all its storage and computational data structures in memory [32].

4. **Extensible:** Being a distributed Stream Processing Engine, it also needs to communicate with other services and protocol such as retrieving database query from MongoDB. Thus Storm must be a framework that allows extensibility.

5. **Easy to Administer:** Storm act like the heart of a real-time application. End-users can immediately notice poor performance (even system failur) if there are issues associating with Storm. The operating team should receive early warnings and they must be pointed to the source of errors as they arise.

A Storm application is described as a "topology" in the shape of a directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. Every node in the network have to process some logic and links between the nodes have to demonstrate how the data will pass.

We can be compare it with the Map and Reduce jobs of Hadoop. Together, the topology acts as a data transformation pipeline. Tuple, the primary data structure in a Storm cluster is a list of values. A stream is an unbounded sequence of tuples. We can have two types of operators in a topology -

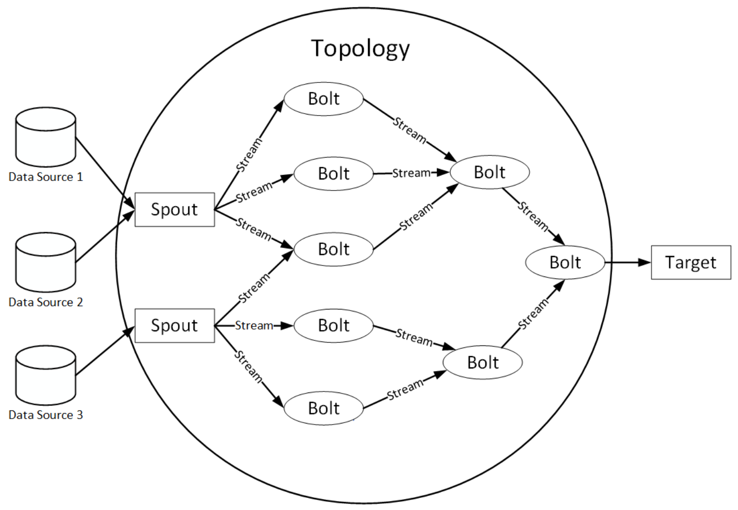

Figure 1: A sample application design for Apache Storm.

- **Spout :** It is responsible for getting in touch with the actual data source, receiving data continuously (external sources (e.g. Twitter API) or from disk), transforming those data into the actual stream of tuples and finally sending them to the bolts to be processed.
- **Bolt :** Bolts are in charge for performing all the processing of the topology. They work as the processing logic unit of a Storm application. Bolts can be used to perform many vital tasks such as - joins, aggregations, filtering, applying custom function, connecting to databases and so on [50].

3.1 Apache Storm - Cluster Architecture

Apache Storm has two types of nodes, Nimbus (master node) and Supervisor (worker node). The Nimbus works as the central component of the storm cluster. It uploads the topology in the cluster. It also launches worker nodes and in case of failure re-assigns worker nodes. A supervisor will have one or more worker process. The supervisor will assign the tasks to worker processes. Worker process will generate as many executors as needed and complete the task. An executor is a single thread created by a worker process, and it can run one or more tasks, but they must be specific for a particular spout or bolt. A task performs actual data processing. Apache Storm follows an internal distributed messaging system while communicating among different nodes in the cluster. Fig. 2 shows the cluster architecture and how different components are connected.

Apache Storm does not have any state-managing capabilities. Although being stateless has its disadvantages, but it helps Storm to process real-time data in the most effective and fast way. For managing its cluster state, it depends heavily on Apache ZooKeeper which is a centralized

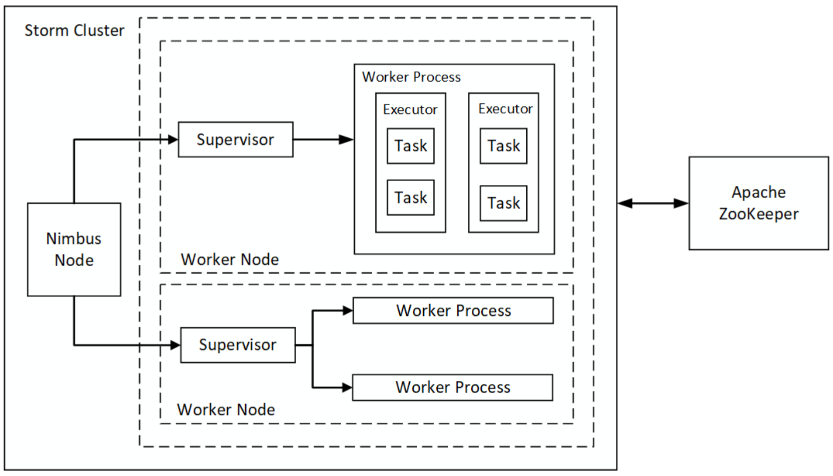

Figure 2: Cluster architecture of Apache Storm.

service for managing the configurations in Big Data applications [50]. As the state is available in Apache ZooKeeper, a failed nimbus can be reassigned and made to work from where it died. For the most parts, service monitoring tools like monit will monitor the activities of Nimbus and restart in case of a failure.

3.2 Apache Storm - Workflow

A functioning Storm cluster should have one nimbus and one or more supervisors. Here we have discussed the work flow of of Apache Storm

- At first, the nimbus will wait for the Strom topology to be submitted to it.
- The Nimbus node will process the topology and gather all the tasks need to be executed; It will also make an order by which tasks need to be carried out.
- The nimbus node will evenly distribute the tasks among the available supervisor nodes.
- After a particular time interval, all supervisors will send a response message to the nimbus node to inform that they are still alive. If a supervisor node dies and does not send a response to the nimbus, then the nimbus allocates the tasks to another supervisor.
- In case the nimbus itself dies, supervisors will continue working on the already assigned task without any issue. After the tasks are completed, the supervisor nodes will wait for a new task to come in. By this time the dead nimbus will be restarted automatically by the service monitoring tools.
- The newly restarted nimbus node will continue from where it stopped. Similarly, a dead supervisor node can also be restarted automatically. As both the nimbus and the supervisor can be restarted spontaneously and will resume as before, Storm is guaranteed to process all the task at least once.
- After all the topologies are processed, the nimbus node waits for a new topology to arrive and similarly the supervisor nodes wait for new tasks.

Apache Storm is highly scalable, easy to use, and offers low latency with guaranteed data processing [51]. It enables the developers to develop their logic virtually in any programming language. These make Apache Storm as a first choice tool for processing real-time unbounded data.

4 DeepLearning4J (DL4J)

DeepLearning4J (DL4J) is the implementation framework for our machine learning model. It is an open source distributed project, to be used for deep learning programming [20]. Although this framework mostly relies on java programming language and java virtual machine (JVM), it has support for Clojure and Scala programmers too. It consists of a wide range of deep learning algorithm implementations like, multilayer artificial neural network (MLP), convolutional neural networks (CNN), recurrent neural networks (RNN), restricted Boltzmann machine, generative adversarial network (GAN), deep autoencoder, word2vec, doc2vec, GloVe etc. It has multiple sub-projects, including these implementations with examples, which can be incorporated for other applications.

Our interest in DL4J specifically comes for the reason that, it has distributed deep learning training implementations. DL4J has two implementations of distributed training:

- Gradient Sharing, which is an asynchronous Stochastic Gradient Descent (SGD) [52, 53] implementation that uses Spark[4] and Aeron.
- Parameter Averaging, which is a synchronous SGD implementation, that uses only Spark platform.

We have chosen the parameter averaging implementation, which uses only Spark platform. The motivation to choose this approach is that, we are already using Storm for our input stream processing. Storm (true stream processing model) and Spark (micro-batch processing) are closely related, as both are used for data stream processing, with specific distinctions[5] as well. Using both Storm and Spark in the same topology has encouraged us to choose the Spark implementation of DL4J. But, we had to modify the Spark based distributed implementation on DL4J, to suit our needs, as our application is quite different from their examples.

5 Characteristics of the Learning Model

Our machine learning model has three characteristics:

- **Multilayer Neural Network:** This will be the core learning part, that will do the predictions. It is an artificial neural network with multiple adjustable hidden layers associated with it. In other words, it is a Multilayer Perceptron (MLP) [54]. In our project, this will be a supervised learning model.

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

[5]https://data-flair.training/blogs/apache-storm-vs-spark-streaming/

- **Distributed Learning:** The learning model will function in a distributed manner. We have used Spark platform to implement the distributed learning.
- **Incremental Learning:** It also needs to be an incremental learning model. It will improve its learning, when new input data is made available to the model over time. This is much suitable for a real time environment based project like ours.

In the following sub-sections, we will discuss all the details of these characteristics.

5.1 Multilayer Neural Network

Neural network learning algorithms provide a robust, general and practical approach for learning real-valued, discrete-valued and vector-valued functions from examples [55].

Now, a Neural Network in computer science, is basically a computing program that acts similarly to the neural network system of a human brain. That is how the name is inspired. A basic neural network consists of a collection of nodes and connections between them. The nodes are called neurons and the connections are called synapses. Each node transmits information to its connected node via the synapse. The receiving node does the same towards its next connected node. This process keeps going on until an output is calculated.

Here, we will briefly go through some basic concepts [56] of different components of a multilayer neural network or multilayer perceptron; along with specifications of our implementation, for those components.

5.1.1 Perceptron

The very basic unit of a neural network is a perceptron. It explains the basic functionality of a neural network very simply. Fig. 3 shows how a perceptron works.

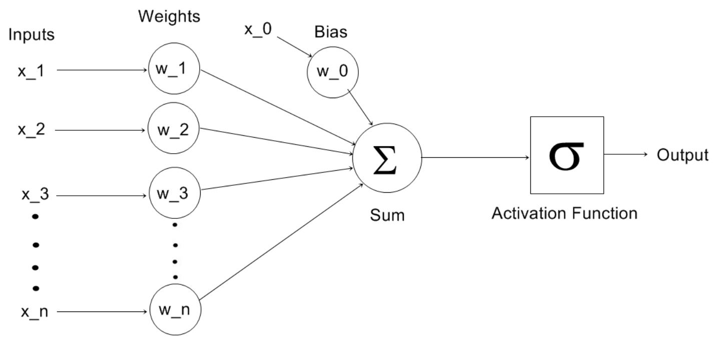

Figure 3: Perceptron

Here, the perceptron takes some real valued inputs like x_1, x_2, x_3 etc. and calculates a linear combination of these inputs. In Fig. 3 the linear combination is just the weighted sum of the inputs with respect to weights w_1, w_2, w_3 etc. So, the weighted sum will be, as shown on eq. (1).

$$weighted\ sum = w_1x_1 + w_2x_2 + ... + w_nx_n \tag{1}$$

Now, the output could be 1 if the result is greater than some threshold value and -1 otherwise. It can be shown like this:

$$output = \begin{cases} -1, & \text{if } \sum_i w_i x_i \leq threshold \\ 1, & \text{if } \sum_i w_i x_i > threshold \end{cases} \tag{2}$$

This is the simplest form of a perceptron. Things like the initial weights, the threshold value, the activation function, concepts of multiple layers of perceptron will be explained in the coming sub-sections, keeping sync with our implementation. Eq. (1) and eq. (2) will change accordingly.

5.1.2 Weight Initialization

From eq. (1), we can see that, we need to define some initial values to the weights. This is kind of a random process, as we cannot be sure about the nature of the data in the beginning.

In our implementation, the initial weights are set with *Xavier* Initialization method [57]. According to this initialization, it assigns weights from a gaussian distribution, having 0 mean and the variance as given in eq. (3)

$$Var(W) = \frac{2}{fan_{in} + fan_{out}} \tag{3}$$

Considering a multilayer perceptron like our case, in eq. (3), $\mathrm{fan_{in}}$ is the number of neurons feeding into the corresponding neuron and $\mathrm{fan_{out}}$ is the number of neurons, the result is injected to, from the corresponding neuron.

The Xavier initialization helps the model to converge pretty fast. According to this scheme, the weight are initialized in such a way that, the variance of **X**(inputs) and the weighted sum remain same through each passing layer. It helps preventing the signal to go to a very high value or to a very low value.

5.1.3 Bias Term (Threshold Value)

If we want to control the activation of a neuron according to some threshold value, then we need a bias term. The neuron will be activated only when the weighted sum is above that threshold value. After including a bias term in eq. (1), it will look like this:

$$weighted\ sum = w_0 x_0 (biasterm) + w_1 x_1 + w_2 x_2 + ... + w_n x_n \tag{4}$$

And finally eq. (2) will be like this:

$$output = \begin{cases} -1, & \text{if } \sum_{i=0}^{i=n} w_i x_i \leq threshold \\ 1, & \text{if } \sum_{i=0}^{i=n} w_i x_i > threshold \\ where, & w_0 x_0 = bias\ term \end{cases} \tag{5}$$

In our implementation, the bias term is 0.

5.1.4 Activation Function

The weighted sum in eq. (4) can be any value. If we want this value to be limited by a range, then we need an activation function. It can be any activation function as per the nature of the data and the application. For example, if we need the output to be between -1 and 1, we can use *tanh* function (Fig. 4) as the activation function.

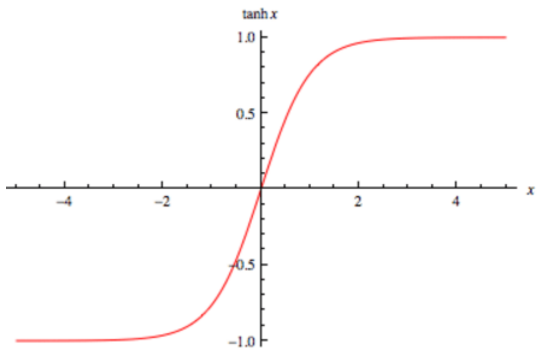

Figure 4: TanH Function(Hyperbolic Tangent)

As shown on Fig. 4, very negative inputs get mapped to -1 and very positive inputs get mapped to 1, while 0 inputs remain close to the 0 value. That is how an activation function behaves in a perceptron. So, the output will be the value, that we get after getting passed through the *tanh* function.

$$output = \tanh(\sum_i w_i x_i) \tag{6}$$

We have used this tanh activation function for the initial layers of our multilayer neural network.

For output layer, we have used Softmax activation (Fig. 5). Softmax activation calculates the probability of a class over all possible classes. That is how the probability range stays between 0 and 1, and the sum of probabilities for all classes will be 1. These probabilities are used to predict a target class, which will have higher probability than other classes.

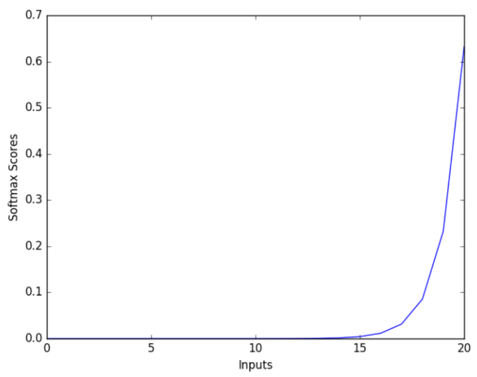

Figure 5: Softmax Function

For now, our model will be basically a binary classifier, determining the tweets from Melbourne location and other locations will be considered as the rest of the world. But, for future implementation, we could use this model to classify multiple locations as well. For that, softmax function in the output layer will be most appropriate, as it is used for multi-class classification.

5.1.5 Hidden Layer

A single perceptron shows us the fundamental mechanisms of a neural network, but it can only express linear decision surfaces. In real time complex applications like location prediction, we cannot be sure of the nature of the data points and determine if they can be linearly separable or not. That is why, we need a multilayer network that can be used to represent highly nonlinear decision surfaces.

Now, in multilayer neural network or multilayer perceptron (MLP), the basic addition is the hidden layer. We need one or more hidden layers, consisting of a number of neurons to build the multilayer network. So, there will be at least three layers in an MLP; one input layer, one output layer and one or more hidden layers (Fig. 6). The neurons in each layer of an MLP are fully connected with the neurons of its adjacent layers.

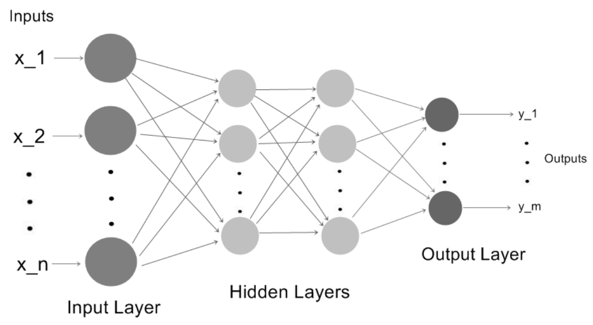

Figure 6: A General Multilayer Neural Network (Multilayer Perceptron)

The training in MLP occurs by recomputing and adjusting the weights associated with each connection; after each processing. This adjustment is done by calculating the error, which can be expressed by different types of loss functions. This weight adjustment is called backpropagation. It has two parts. The forward pass evaluates the expected output, based on given inputs. The backward pass takes derivatives of the loss functions based on the errors, with respect to different parameters and propagates back to the network. This process continues to adjust the weight vectors, until the error is at its lowest value. The final adjusted values of the weight vectors actually determine the importance of different input features, while making the final predictions in the output layer.

In our implementation, the weight updater function uses Stochastic Gradient Descent (SGD). Backpropagation is set using backprop(true) in the code. The loss function in our model uses negative log likelihood. As we need to minimize the loss function, minimizing negative log likelihood is actually equivalent to maximizing the log likelihood or the likelihood itself.

Now, for the final MLP architecture (Fig. 7), we have 100 neurons in the input layer, as our training data set has 100 features. We have chosen one hidden layer and the number of neurons in the hidden layer is set to be the same as the number of neurons of output layer. The reasons for these choices are explained later in section: 9.

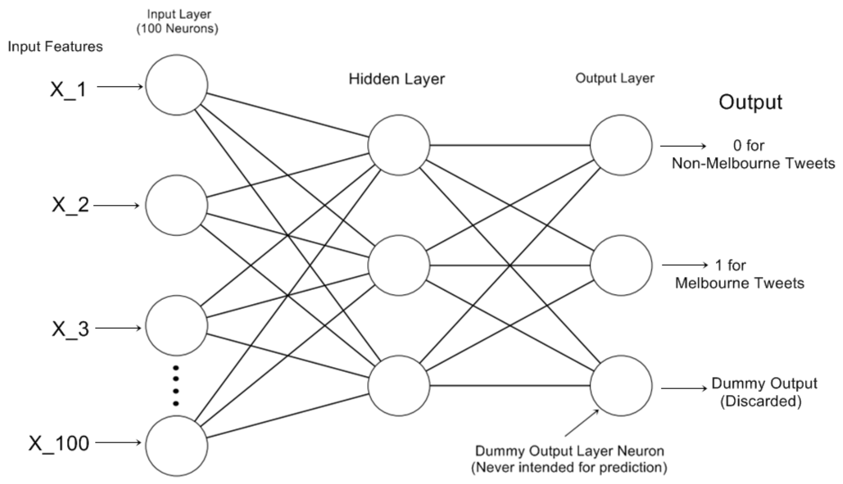

Figure 7: Our Multilayer Neural Network Architecture

One important thing to notice on Fig. 7 is that, the number neurons in output layer is 3, even though we have only two classes to classify. The third neuron is just a dummy neuron. This is given for DL4J implementation specification, otherwise the matrix dimensions do not match inside the inner calculations. This third neuron just serves as a dummy label for a test data, whose label we need to predict through the model. For Melbourne tweets, we put the class label as 1, for Non-Melbourne tweets, we put the class label as 0 and for test data, the unknown label of test data is initially given as 2.

The code snippet of this implementation can be found in the appendix section: B.5.

5.2 Distributed Learning

The learning process of our Multilayer Neural Network occurs in a distributed manner. As mentioned earlier in section: 4, we have chosen to follow the *Parameter Averaging* concept of DL4J's distributed learning implementation. This method only uses Spark platform for the distribution of the learning process.

5.2.1 Spark Architecture

The Apache Spark architecture [58] is basically a master-slave architecture. It has two main components: Master Process (Driver) and Worker Processes along with a cluster manager in between them (Fig. 8).

A spark cluster consists of a single master and a varying number of workers. The master and slave processes run their own processes and user can run them on single machine or separate machines as well. Other than these, Spark platform uses two special abstractions: **RDD** and **DAG**. They are described below.

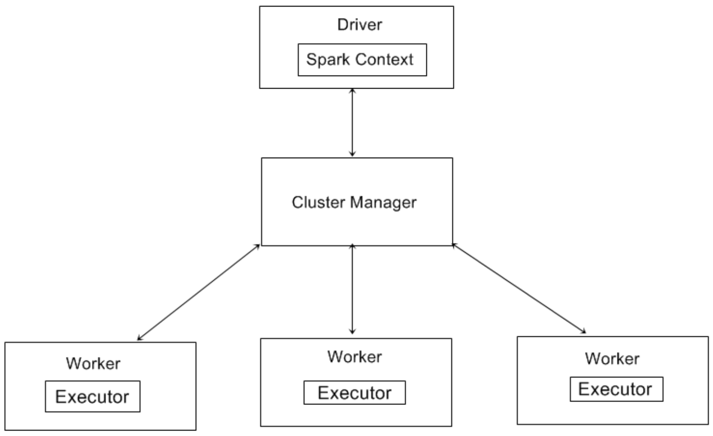

Figure 8: Spark Architecture

- **RDD:** Resilient Distributed Data sets are special type of data items, which can be split into a number of partitions for distribution and stored in memory of the worker nodes.
- **DAG:** Directed Acyclic Graph in spark determines the sequence of operations to be done on each rdd partitions.

Now, the roles of the Spark components are described briefly below:

- **Driver/Spark Context:** The driver program runs the main() function of an application. The spark context is created here too. Only one spark context is allowed to be created per JVM, in a driver node. The driver program of the master node also schedules jobs of the executors and communicates with the cluster manager to do so. It also creates execution graphs using RDDs and splits them into different stages.
- **Worker/Executor:** Executors perform specific tasks allocated to them, by the master node. They process the data; do read and write operations on external sources and also save the computation results.
- **Cluster Manager:** The cluster manger provides a special service that has access to resources of spark clusters. The cluster manager allocates those resources to spark jobs, depending of different scheduling properties.

5.2.2 Distributed Learning using Spark

DL4J uses this master-slave architecture of Spark to implement the distributed learning method.

Now, according to DL4J documentation [59], Spark should be used for training a distributed learner in the following cases:

- There is cluster of machines, not just a single machine for training.
- More than a single machine is needed for training.

- Network is large enough to justify a distributed training.

Now, in our case, as we are building a prototype system, we have used single machines only. But, for future implementation flexibility, into large networks, with more machines; Spark is chosen. The other motivation to choose Spark for the compatibility with Storm is mentioned in section: 4 earlier.

DL4J's *Parameter Averaging* implementation, used for Spark's distributed learning; is a synchronous Stochastic Gradient Descent approach [1]. There is a single parameter server that is maintained by the master node of Spark. The users need to specify the frequency of averaging that controls how the worker nodes can synchronize with other worker nodes and also the master node. The detail training process in parameter averaging is described below:

1. The master node (driver process) initializes spark configuration. As mentioned earlier, the spark context gets created here; but only once per JVM.

2. The training data then gets split into multiple subsets according to the configuration of the training master.

3. Next, each split data set gets iterated over. For each of those data splits, the configuration and their respective parameters (mainly weights, which need to be learned through the neural network) are distributed from master to the workers.

4. Then, each worker runs its own process (the multilayer neural network) and fits the model on its own portion of the data split, which was distributed to it from the master node.

5. Then, the parameter server does the averaging of different values of parameters(weights) from each worker and returns the averaged results to the master. The overall state of the network also gets updated to make the model more fault tolerant, in case of any worker node failure.

6. Finally, the training is complete after the parameter server averages the results from the distributed portions and the master gets a copy of the final trained network.

Fig. 9 shows the above mentioned steps 3 to 5.

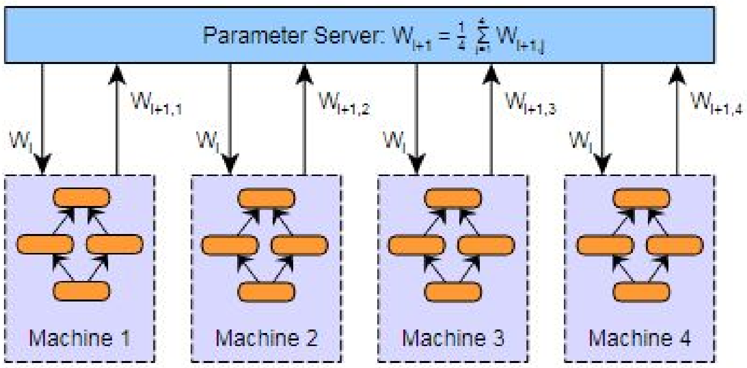

Figure 9: Parameter Averaging [1]

The **W**s(parameters) are the weights here. The subscripts give us the idea about different version of the weights for different machines. The example here is given using 4 machines. When run locally, they are just replaced by worker nodes under the same single machine.

There are many huge libraries and big code snippets for the whole learning part. To keep it simple, we just provide some brief code snippets in appendix section: B; explaining briefly about the key work flow of the implementations using DL4J framework, on a high level abstraction.

5.3 Incremental Learning

Incremental learning has been practiced in various ways as discussed in the literature [60]. One of the simplest ways is storing all the previous data which will be used for retraining with new data. Another way is the training of the data, instance by instance, in an online learning fashion. Methods using an online approach for incremental learning have been implemented in many, but this method does not consider all the issues of learning, specifically the learning of new classes [61]. Fig. 10 shows the workfow of an incremental model -

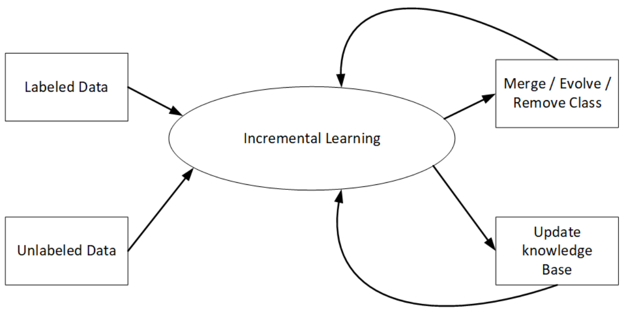

Figure 10: Working of an Incremental Learning

In Incremental learning, we can have newly gained knowledge as well as evolving new class or a cluster. It can also reform or combine the existing classes. The classifier will be dynamic in nature with the changing environment [48]. The objective of incremental learning is to endorse new data without forgetting its current knowledge. But Learning new information without forgetting previously acquired knowledge, however, introduces one of the fundamental problems in knowledge management (KM) : *stability-plasticity dilemma* [62]. The dilemma points out the fact that some information may have to be lost to learn new information, as learning new patterns will tend to overwrite previously acquired knowledge. For a classifier to be incremental, it should meet the following criteria [61]:

1. It should be able to learn additional data from new data.
2. It should not have the access of original data to build the classifier.
3. It should preserve previously acquired knowledge and it should not suffer from catastrophic forgetting.
4. It should be able to add new classes that may be brought in with new data.

Some incremental learning algorithms have built-in parameters to adjust the relevancy of old data. Others learn representations of the training data that are not even partially forgotten over time. These are known as stable incremental machine learning algorithms. Fuzzy ART [46] and TopoART [63] are two examples for this second approach.

Incremental algorithms are repeatedly applied to big data or data streams, addressing issues in resource shortage and data accessibility respectively. This setting is present whenever systems act autonomously such as in autonomous robotics or driving. Further, online learning becomes essential in interactive scenarios where training examples are provided based on human feedback over time [64]. User profiling and stock trend prediction are some examples of data streams where new data frequently become available. Implementing incremental learning to big data intents to generate faster classification or forecasting times.

Here we will train and predict with labeled data over time and classify unlabeled data. For predicting a labeled data, we first classify the sample instance then added the sample in training instances to avoid over-fitting.

6 Our proposed Architecture

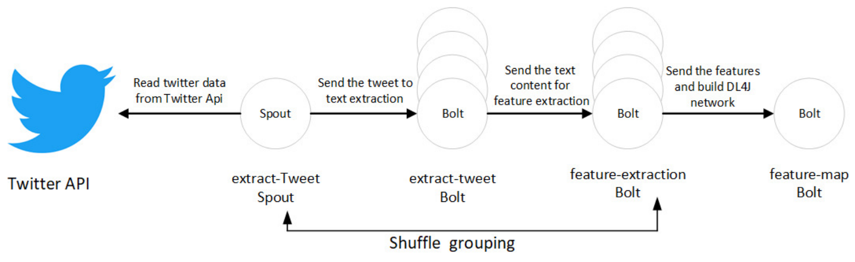
Twitter API
Read twitter data from Twitter Api
Spout
extract-Tweet Spout
Send the tweet to text extraction
Bolt
extract-tweet Bolt
Send the text content for feature extraction
Bolt
feature-extraction Bolt
Send the features and build DL4J network
Bolt
feature-map Bolt
Shuffle grouping

Figure 11: Our proposed architecture.

Fig. 11 shows our proposed network topology. Here, we have a twitter spout that will continuously read twitter data from Twitter API. At the time of filtering these posts, we are only considering posts from the English language. We are also using a boundary box to identify the Melbourne region, and the posts from this boundary box will be filtered out and saved in the query. The code is shown in Appendix B.1.

These raw twitter posts will be sent to ”extract-tweet” bolt. The bolt will extract the text from the post and forward to ”feature-extraction” bolt. This bolt will analyze the text with the help of the Natural Language Toolkit of Python [65]. The codes for Apache storm’s spout and bolts are generally written in Java, but using ”Multi-Language” protocol [6] of Apache storm, we can write codes for some bolts or spouts in a different programming language. As the libraries of python’s natural language processing are vibrant and effective, we decide to use this. The detail description and steps of data prepossessing and feature extraction is discussed in Sec.7.

The extracted features will be sent to the ”feature-map” bolt. To train the model with deeplearing4j, we need to pass a CSV file in the model constructor. As the model is incremental,

[6]http://storm.apache.org/releases/1.1.2/Multilang-protocol.html

we will convert the features into a row of the CSV file and append the row after existing data. As time goes on the size of the training CSV file will increase and as well as the accuracy of the model. There are some limitations of the model. We can't dynamically change the feature dimension, as it needs to be fixed from the beginning. Therefore we are using an initial dataset for this purpose, and the extracted features from that dataset are our baseline features. Only the tweets that have features from the initial feature set are classifiable in our model. We discussed the improvement of the model in Sec.7. After the CSV file is created, the bolt will call the deeplearning4j's prebuild Apache Spark library and pass the CSV file to train the model. The detail description of neural network of deeplearning4j and distributed workers of Apache Spark is presented is Sec.5.1 and Sec.5.2 respectively. Here in model we are having multiple instances of the extract-tweet, feature-extraction bolt and using shuffle grouping among the bolts.

7 Data Description

Twitter is a social networking and microblogging service that allows users to post real-time messages known as *tweets*. Tweets are short messages and restricted to 140 characters, but it can contain media files like pictures or videos. Because of the short nature of the message, people use acronyms, emoticons and other characters that express special meanings. Besides original posts, a user may also retweet others' tweets. While composing tweets contents, a user may include hashtags to mark topics, which are words or unspaced phrases starting with '#'. Users primarily add it to increase visibility. Besides, one can mention other users by a preceding '@' before their name in tweet content. Referring to others in this manner will automatically notify that user, and may start a conversation using subsequent mentions.

7.1 Geolocation in Twitter

Twitter provides different data APIs, and tweets come as JSON objects that include the tweet text along with various metadata, such as the time, user profile information, the location (if provided by the user), etc. These metadata can be used as a source of information to geolocate the tweets. In particular, there are four primary [7] ways in which geolocation is commonly predicted from Twitter users and messages [69, 70] :

1. **Place object from tweets :** Some tweets extracted by the Twitter API include a JSON object named "place" which indicates the location associated with the tweet. It contains general fields like city and country of the location. Sometimes it provides more precise information such as business names and street address. While posting users have the option to tag their tweets with a place, this tagging can automatically be done based on the matches to the user's current GPS position. For tweets containing place objects, it is straightforward to track the geo-location. However, only 1% of the tweets come with place object in their JSON representation.

[7]Others have analyzed using the social network structure [66–68]

2. **Coordinates from tweets :** A few tweets are geotagged with the coordinates (latitude and longitude) of the user, based on the user's GPS position at the time of uploading the post. For these tweets, we have the exact location of the tweet. But As They have not been resolved to places, we don't have any high-level information like the city and country. We can use reverse geocoding using APIs from Google Maps to obtain detail information. There is a broader set of tweets containing geocoordinates than those that have been posted with places, but compare to all tweets this percentage is tiny.

3. **Location from user profile :** Many users publicly provide a location in their profile. Such strings can be used to resolve to structured locations by map APIs. But these strings are mostly static, referring to the user's primary location at the time of creating the account rather than the location of uploading the post which may be different if the user is traveling. Many more users have profile locations than geocoordinates. However, users may lie or provide nonsensical locations such as "Candy Land" [71, 72].

4. **Content-based geolocation :** Geolocation can also be extracted based on the content of the message [10,12,14,15]. A user's primary location be predicted based on their dialect or mention of regional specific terminologies like public holidays and political or sports team as well as local landmarks. Such a method can be used be where the user doesn't provide any location information explicitly. To do this accurately, we may be need samples from many users to establish a model.

7.2 Primary Twitter dataset

To build an initial feature vector, we used the Big_GeoBox [8] dataset. It has tweets collected over a few months from around the world including a geo bounding box. There are total 7916382 tweets, and among them, 28218 tweets were from Melbourne[9] (Here we used boundary coordinates, all tweets inside the boundary box were labeled as tweets from Melbourne). If we consider only tweets that are tagged with a location "Melbourne", then we can have only 13132 tweets, but while considering the boundary box, we are getting double amount of samples. It also shows people use GPS coordinates are more frequently then explicitly defining the place.

7.3 Prepossessing Tweets

We will process the raw texts of the tweets by following these steps -

- Convert the tweets to lower case. Replace the non-English letter and emoticons with white spaces.
- Here in this project we didn't follow the short urls mentioned in a tweet and ignore the content of that site. So we can remove all of these URLs via regular expression matching and replace them with a generic token "URL".

[8]$https://sunrise.cis.unimelb.edu.au:82/aaron/Big_GeoBox.json.gz$

[9]The bounding boxes that we used to define Melbourne(140.34, -39.71, 149.66, -32.087)

- we can eliminate tagged user in the tweets(@username) via regular expression matching and replace it with generic token "MENTIONED_USER".
- We replace all the hash-tags with the exact same word without the hash. E.g. #unimelb will be replaced with 'unimelb'.
- While processing the tweets, we remove punctuation and white spaces at the beginning and ending of the tweets. E.g. ' the weather is sunny! ' will be replaced with 'the weather is sunny'. We also replace multiple white spaces with a single white space.

7.4 Filtering tweet words

After preprocessing, we will extract features from the tweets by following these steps -

- At first, we remove all the common stop words[10] that do not contain any location information. We also append "URL" and "MENTIONED_USER" in our stop words list. This step will remove all the added URLs and tagged users from the tweets because at the time of prepossessing; we replace them with "URL" and "MENTIONED_USER" respectively.
- We will look for two or more repetitions of a character and replace with the character itself, such as, 'wooooowwwwwww!' and 'whatttttt?' will be replaced by 'wow!' and 'what?' respectively.
- We will strip all kind of punctuation (e.g., comma, question-marks, quotes, etc) from the beginning and start of each word of the tweet. E.g., wow!!!!!!! will be replaced with wow.
- Here in this project; we are only considering those words that start with an alphabet. We will remove all the words other than this (e.g., 31st, 7:14 pm)
- Finally we will perform stemming to get the root words, such as, 'makes' and 'disappointing' will be replaced be 'make' and 'disappoint' respectively.

7.5 Creating dataset for our model

Here We have two class labels (Melbourne and non-Melbourne). We extract features from the selected tweets for both classes. For each category, we ranked top 50 words based on their frequencies. These 100 words (50 from each class) were used as features to create the base dataset. We named it "top_50_dataset". For each of the tweets, we used the frequencies of the featured word in the feature vector. To make a DL4j friendly dataset, we need to work with integer only (DL4J only supports Integer).

We also made another approach by combining two classes and ranked top 100 words based on their frequencies. These 100 words were used as features to create the dataset. We named it "most_100_dataset"

[10]https://github.com/AhmedFahmin05/research/blob/master/stopwords.txt

8 Evaluation Metrics

For evaluating the performance of our model, we primarily have considered two things: **Accuracy** and **Runtime** measure for the training. Among these two metrics, we give more priority to the runtime measure. Accuracy is obviously important as we want to correctly predict the tweets from Melbourne as much possible. But, we need to keep in mind that our system is intended to be deployed in a real time environment. In that case, a lower running time of the program is very much desired; even more than a too good accuracy. So, we evaluated the model based on both accuracy and runtime while giving a little more priority to runtime.

Now, this primary evaluation criterion is mainly intended for choosing the final optimal model for the system, which will be discussed more elaborately in the following section: 9.

After deciding on the final model, we show its performance evaluation using accuracy, recall and precision. We also use these metrics for the evaluation on a real time data set too, while doing the learning incrementally.

9 Model Analysis

There are a number of external (hyper)parameters, that control the behaviour of the Distributed Multilayer Neural Network model. The ones that we are considering, are as follows:

- **Hidden Layer Number:** Determines how many hidden layer should we have in our neural network.
- **Number of Neurons in Hidden Layer:** They can be varied for each hidden layer.
- **Number of Epochs:** The number of times the model will fit the training data.
- **Batch Size Per Worker:** It determines how distributed our learner will be. Low batch size per worker means there will be many workers, and more distribution of the learning process. On the other hand, high batch size means less number of worker nodes.

Now, taking all of them into account as variables, at once, makes the decision making very complex. So, we first decide on the epochs and batch size per worker, fixing the hidden layer number to be one, and number of neurons in that layer to be same as the output layer. After deciding these two, we will decide on the hidden layer number and number of neurons in them.

9.1 Choosing Number of Epochs and Batch Size Per Worker

As mentioned in section: 8, we need to choose a balanced value of accuracy and runtime, giving runtime a higher priority. In case of epochs and batch size per worker, we can see their relation with accuracy and runtime of the program, from Tab. 1.

Increasing epochs will obviously train the model better, so accuracy will get higher, but it will take much time. Decreasing epochs will do the opposite for both accuracy and time.

On the other hand, if we increase batch size per worker, each worker will be training on a lot more data, so the fitting of the model will not be much good. Hence, accuracy will drop. But

Evaluation Metric	**Number of Epochs**		**Batch Size Per Worker**	
	High	**Low**	**High**	**Low**
Accuracy	High	Low	Low	High
Runtime	High	Low	Low	High

Table 1: Relation of Accuracy and Runtime with Epoch and Batch Size Per Worker

at the same time, it will take less runtime. It is because, it will take less amount of computing for lower number of workers, while combining their results. So, we get both low accuracy and low runtime. Decreasing batch size per worker will improve training and hence the accuracy. But, it will take much time to combine the results of more worker nodes.

So, from Tab. 1, we can see that we cannot get a high accuracy and a low runtime at the same time. That is why, we need to find an optimal value for epochs and batch size per worker so that, we can get a run time as low as possible, with an acceptable good accuracy.

To determine on these values, we ran the model on our base training data set, for different values of epochs and batch sizes. We have selected an epoch number, in range from 1 to 9. After 9 epochs, the initial runtime rises very sharply (Fig. 13). So, we do not go further above that. For batch size per worker, we have selected a range from 100 to 3000. The blue line is for a model with 1 epoch, the orange line is for 5 epochs, and the grey line is for 9 epochs in Figs. 12,13,14.

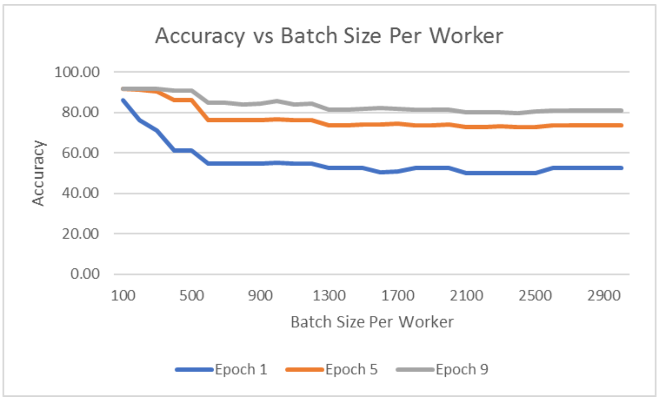

Figure 12: Accuracy vs Batch Size Per Worker

From Fig. 12, we can see that, from around 600 to 700 batch size, the accuracy drops for all epoch lines and remains somewhat stable afterwards. For the model with 1 epoch, the accuracy is not that good. The models with 5 epochs and 9 epochs are relatively close after that drop point. But, as we give much priority to run time than accuracy, we need to observe the runtime behaviour for these models, for the exact batch size numbers, to reach a decision.

From Fig. 13, we can see a similar behaviour like Fig. 12. As the accuracy drops, the run time also drops around the same region. Although it does not remain as much stable as the accuracy, but the run time still remains pretty low after that drop point.

From analyzing Fig. 12 and Fig. 13, we come up with a batch size number for each epoch line, that primarily gives the lowest run time for each of them, with a considerably good stable

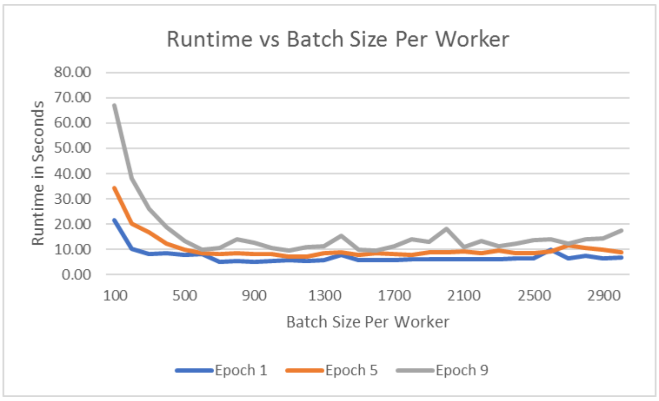

Figure 13: Runtime vs Batch Size Per Worker

accuracy, while run on the base training data set (non-incrementally). The values are shown on Tab. 2

Number of Epochs	**Batch Size Per Worker**	**Run Time (Seconds)**	**Accuracy (%)**
1	700	5.10	54.62
5	1100	7.11	76.20
9	600	9.77	84.76

Table 2: Runtime and Accuracy for Selected Batch Size Values

Finally, using these values, we ran three different distributed learning model incrementally, on our base training data set to see the accuracy behaviour of them (Fig. 14). For testing purpose, we have used 5000 training data at each iteration, from the whole training data set.

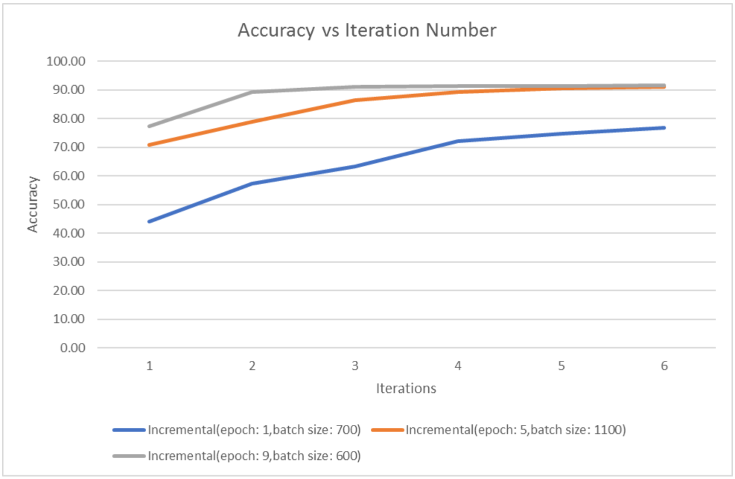

Figure 14: Accuracy vs Iterations of Incremental Learning

As can be seen from Fig. 14, the model with 5 epochs eventually catches up to the model with 9 epochs. The model with epoch 1 does not show much improvement on accuracy over time. So, among 5 epochs and 9 epochs, we finally choose the epoch number to be 5, because it gives a similar accurate performance like 9 epochs, but the run time is relatively lower (Tab.

2) than the model with 9 epochs.

9.2 Choosing Number of Hidden Layers and Neurons in Hidden Layers

We now analyze the behaviour of the model for different number of hidden layers and neurons for those hidden layers, by using the selected values of epochs and batch size.

We first select only one hidden layer for the model and change the number of neurons in them. Based on these variations, we ran the model separately for evaluation set (portion of base training data set) and real time test set (Melbourne tweets collected in present time) to see how well the model fits (Fig. 15).

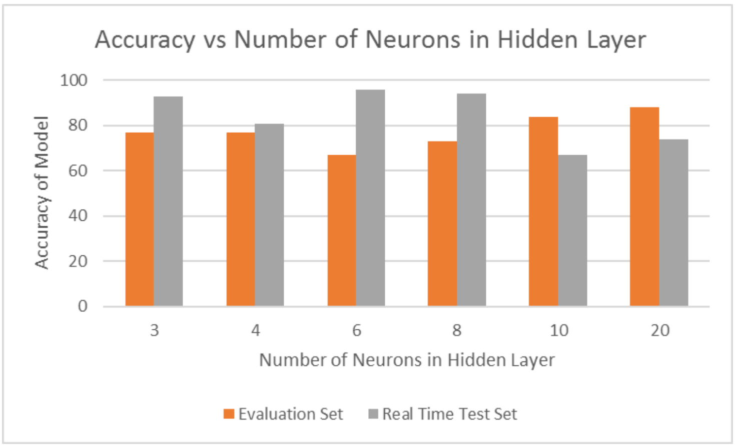

Figure 15: Accuracy vs Number of Neurons in a Single Hidden Layers

As shown on Fig. 15, if we increase neurons in the hidden layer, the model shows some irregular behaviour. Sometimes the evaluation set gives better result, sometimes the real time test set gives better result. We want a model that behaves relatively similar, in both types of data sets. To choose the best-fit, and to keep it simple, we go for the same number of neurons (which is 3) as output layer, to be in the hidden layer. We could have chosen 4 neurons as they show the most similar behaviour, but 3 neurons will take lower computation time, which is a major concern for us.

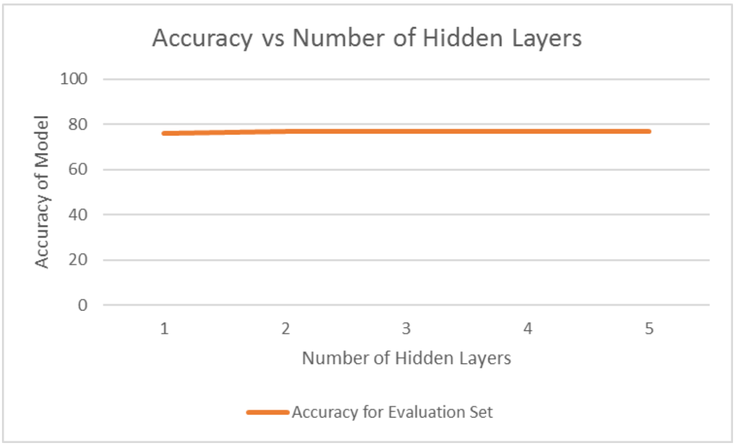

Figure 16: Accuracy vs Number of Hidden Layers

After deciding on the neuron number, changing the number of hidden layers does not change the result very much (Fig. 16). Again, more hidden layers means more computing and thus, more run time. Moreover, we are just doing binary classification for now, so, we do not need much composition by increasing hidden layers. That is why, we keep it to only one hidden layer as it reduces extra computing time and keeps the model simple, according to the nature of the classification.

10 Results

After analyzing the model's behaviour in section: 9, we have come up with the following values of the Distributed Multilayer Neural Network's external parameters:

- **Hidden Layer Number:** 1
- **Number of Neurons in Hidden Layer:** 3 (same as output layer neuron number)
- **Number of Epochs:** 5
- **Batch Size Per Worker:** 1100

We then evaluate the final model with incremental approach, using these values. The overall performance of the model on the evaluation set (a portion of our base training data set), is summarized on Tab. 3.

Evaluation Metrics	**Value**
Average Accuracy	84.59%
Average Recall	81.15%
Average Precision	82.34%
Average Runtime	7.11 sec

Table 3: Overall Performance of our Learning Model on Evaluation Set

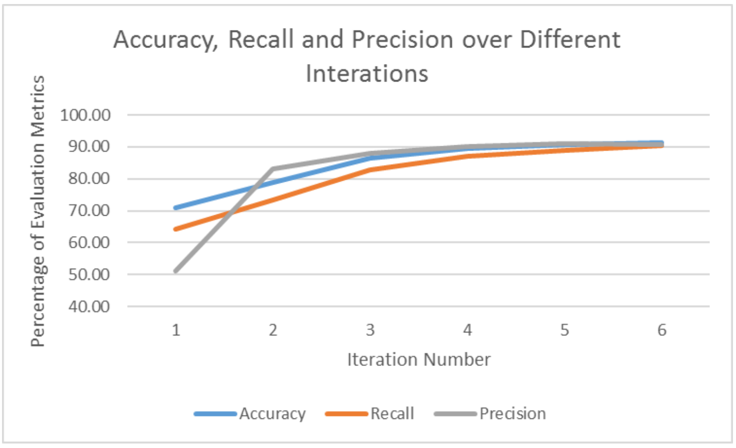

Figure 17: Accuracy, Recall, Precision vs Iterations

The behaviour of accuracy, recall and precision, in different iterations are shown on Fig. 17. We have used 5000 data at each iteration like before.

From Fig. 17, we can see that, over time, the learning gets better in higher iterations. Accuracy, recall and precision also show similar results in higher iterations. So, the model gives a stable and better performance, as the incremental approach improves the learning over time.

Using this final model, we then test it on tweets from present time. We filter out tweets from Melbourne only and see how our model predicts them. As mentioned earlier, we can classify certain tweets, which match our feature set. We call them classifiable tweets. So, among the filtered tweets, we get about 25.36% tweets as classifiable. Among those 25.36% classifiable tweets, we can predict almost 91.43% correctly as Melbourne tweets. The results on the real time test set is summarized on Tab. 4.

Evaluation Metrics	**Value**
Classifiable Tweets	25.36%
Average Accuracy on Classifiable Tweets	91.43%
Overall Accuracy	23.19%

Table 4: Overall Performance of our Learning Model on Real Time Test Set

11 Challenges and Future Scope

There are still many challenging aspects in this project, and they can be implemented in the future. Some of those challenges and future scopes are as follows:

- **Using More Metadata from Tweets:** In our project, we use the texts from the tweets to create our feature vector for the training data. The significance of textual features varies from time to time, especially in the case of social media platforms, as the trends change rapidly over time [73, 74]. So, for future, more metadata from tweets could be extracted and included in the feature set of the training data. From our analysis of the BigGeoBox dataset, which was the source of our base training data, we have found various other metadata keys that can be useful in future implementations. Some of them are followers, followings, time zone, timestamp, hashtags, language, etc. Besides we can also use social network relationships of the users to predict geolocation. We are even ignoring the content of the URLs present in a tweet. These URLs can provide useful information about the geolocation of the tweet.

- **Implementing Dynamic Feature Addition Mechanism:** This is another challenging aspect of this project which is mainly for the restrictions of DL4J. DL4J training models require the feature dimensions to be fixed. When dimension changes, certain parameters need to be adjusted to run the training model properly. Moreover, all of the training data then need to incorporate those additional features into their feature set, because the training matrix need to be of uniform dimensions. For now, we have used a fixed sized feature vector for the training. But, to fully exploit the capabilities of incremental learning, the feature vector needs to change dynamically. Then, we would need to add those features to all the training data and assign values to them as well. After all those modifications, we can change the parameters inside DL4J program to run the model. With each addition of features, this same process needs to be done repeatedly. For our limited time constraint, we implemented the model with fixed feature set for now. In future, it can be enhanced by following the above mentioned mechanisms.

- **Predicting More Locations:** Our primary goal for the project was to predict tweets from Melbourne location only. So, we implemented a binary classification model, that classifies tweets as Melbourne Tweets or Non-Melbourne (Rest of the World). Given more data and time, we can enhance the model to classify more locations specifically. We have kept some flexibility in our training model too (mentioned in section: 5.1.4), keeping this in mind, for future implementations.

- **Developing a fully Apache Storm based Platform:** This was one of the main motivation of this project and at the same time, most challenging part of it. We have implemented the system partially in Storm, where the data stream pre-processing is being done on Storm, and the distributed learning is being done on Spark, under the same topology. We primarily intended to observe the Spark implementation of DL4J and then implement a Storm version of it. But, considering the initial challenge of learning these new technologies, and limited time, we were only able to partially implement the system using both Storm and Spark. So, developing a fully Storm based platform remains as a future scope of this project.

- **Advanced Incremental learning:** As we are running our model with DL4J library, we are passing the complete training file each time we want to retrain the model by following the way DL4j works. But training with a large training set each time is computationally expensive, and the size of the training set increases exponentially while working in real time. In the future, we will try to learn from new instances and tune the parameters of the previous model without training the full system again and again.

- **Running the application in cluster:** Here, we are using Apache storm only in the local mode. We have not submitted the topology in a remote cluster and observed the performances. There are some challenging aspects of running the model in a cluster. Although we will receive higher computation power, we need some more research to include Apache spark inside Apache storm in a remote mode, as we did not find any existing work regarding this. We have worked with Storm and DL4j(standalone, without Apache spark) within the remote mode. But we are building a distributed neural network; the standalone DL4j does not meet the goals.

12 Conclusion

In this project, we have proposed a geo-location predicting system using tweets, from the Twitter platform. We have mainly tried to predict tweets from Melbourne, based on the texts of the tweets. Since it needs to be functioning in real time, we have used Apache Storm for real-time stream processing of the tweets. After processing tweets for training, we inject them into our machine learning model and do the learning incrementally and distributively using Apache Spark Platform. Finally, we have used that model for predicting whether an unknown tweet is from Melbourne or not. Among the filtered tweets, we get about 25.36% tweets as classifiable. Among those 25.36% classifiable tweets, we can predict almost 91.43% correctly as Melbourne tweets.

But, this project still has a lot of challenging aspects that can be implemented in the future, with time. Especially, predicting more geo-locations, based on dynamic features will improve this system's capabilities and extend its usability to a more general platform. Again, developing a distributed incremental learning system; entirely on a stream processing platform like Apache Storm, remains a challenge and has lots to work on.

References

[1] "Dl4j distributed training: Technical explanation," https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-technicalref, 2018.

[2] P. Earle, M. Guy, R. Buckmaster, C. Ostrum, S. Horvath, and A. Vaughan, "Omg earthquake! can twitter improve earthquake response?" *Seismological Research Letters*, vol. 81, no. 2, pp. 246–251, 2010.

[3] Z. Ashktorab, C. Brown, M. Nandi, and A. Culotta, "Tweedr: Mining twitter to inform disaster response." in *ISCRAM*, 2014.

[4] F. Morstatter, S. Kumar, H. Liu, and R. Maciejewski, "Understanding twitter data with tweetxplorer," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2013, pp. 1482–1485.

[5] A. Noulas, S. Scellato, N. Lathia, and C. Mascolo, "A random walk around the city: New venue recommendation in location-based social networks," in *Privacy, security, risk and trust (PASSAT), 2012 international conference on and 2012 international confernece on social computing (socialcom).* Ieee, 2012, pp. 144–153.

[6] M. Schedl and D. Schnitzer, "Location-aware music artist recommendation," in *International Conference on Multimedia Modeling.* Springer, 2014, pp. 205–213.

[7] O. Phelan, K. McCarthy, and B. Smyth, "Using twitter to recommend real-time topical news," in *Proceedings of the third ACM conference on Recommender systems.* ACM, 2009, pp. 385–388.

[8] M. M. Mostafa, "More than words: Social networks' text mining for consumer brand sentiments," *Expert Systems with Applications*, vol. 40, no. 10, pp. 4241–4251, 2013.

[9] A. P. Kirilenko and S. O. Stepchenkova, "Public microblogging on climate change: One year of twitter worldwide," *Global environmental change*, vol. 26, pp. 171–182, 2014.

[10] Z. Cheng, J. Caverlee, and K. Lee, "You are where you tweet: a content-based approach to geo-locating twitter users," in *Proceedings of the 19th ACM international conference on Information and knowledge management.* ACM, 2010, pp. 759–768.

[11] F. Morstatter, J. Pfeffer, H. Liu, and K. M. Carley, "Is the sample good enough? comparing data from twitter's streaming api with twitter's firehose." in *ICWSM*, 2013.

[12] B. P. Wing and J. Baldridge, "Simple supervised document geolocation with geodesic grids," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1.* Association for Computational Linguistics, 2011, pp. 955–964.

[13] S. Kinsella, V. Murdock, and N. O'Hare, "I'm eating a sandwich in glasgow: modeling locations with tweets," in *Proceedings of the 3rd international workshop on Search and mining user-generated contents.* ACM, 2011, pp. 61–68.

[14] J. Eisenstein, B. O'Connor, N. A. Smith, and E. P. Xing, "A latent variable model for geographic lexical variation," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2010, pp. 1277–1287.

[15] S. Roller, M. Speriosu, S. Rallapalli, B. Wing, and J. Baldridge, "Supervised text-based geolocation using language models on an adaptive grid," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning.* Association for Computational Linguistics, 2012, pp. 1500–1510.

[16] B. Han, P. Cook, and T. Baldwin, "Text-based twitter user geolocation prediction," *Journal of Artificial Intelligence Research*, vol. 49, pp. 451–500, 2014.

[17] D. Jurgens, "That's what friends are for: Inferring location in online social media platforms based on social relationships." *Icwsm*, vol. 13, no. 13, pp. 273–282, 2013.

[18] R. Compton, D. Jurgens, and D. Allen, "Geotagging one hundred million twitter accounts with total variation minimization," in *Big Data (Big Data), 2014 IEEE International Conference on.* IEEE, 2014, pp. 393–401.

[19] A. Rahimi, D. Vu, T. Cohn, and T. Baldwin, "Exploiting text and network context for geolocation of social media users," *arXiv preprint arXiv:1506.04803*, 2015.

[20] D. Team *et al.*, "Deeplearning4j: Open-source distributed deep learning for the jvm," *Apache Software Foundation License*, vol. 2, 2016.

[21] J. S. van der Veen, B. van der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on.* IEEE, 2015, pp. 154–161.

[22] M. Yang and R. T. Ma, "Smooth task migration in apache storm," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 2067–2068. [Online]. Available: http://doi.acm.org/10.1145/2723372.2764941

[23] Y. Yang and J. O. Pedersen, "A comparative study on feature selection in text categorization," in *Icml*, vol. 97, 1997, pp. 412–420.

[24] B. Han, P. Cook, and T. Baldwin, "Geolocation prediction in social media data by finding location indicative words," *Proceedings of COLING 2012*, pp. 1045–1062, 2012.

[25] D. Jurgens, T. Finethy, J. McCorriston, Y. T. Xu, and D. Ruths, "Geolocation prediction in twitter using social networks: A critical analysis and review of current practice." *ICWSM*, vol. 15, pp. 188–197, 2015.

[26] M. D. Conover, B. Gonçalves, J. Ratkiewicz, A. Flammini, and F. Menczer, "Predicting the political alignment of twitter users," in *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on.* IEEE, 2011, pp. 192–199.

[27] A. Tumasjan, T. O. Sprenger, P. G. Sandner, and I. M. Welpe, "Predicting elections with twitter: What 140 characters reveal about political sentiment." *Icwsm*, vol. 10, no. 1, pp. 178–185, 2010.

[28] J. Bollen, H. Mao, and X. Zeng, "Twitter mood predicts the stock market," *Journal of computational science*, vol. 2, no. 1, pp. 1–8, 2011.

[29] H. Achrekar, A. Gandhe, R. Lazarus, S.-H. Yu, and B. Liu, "Predicting flu trends using twitter data," in *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on.* IEEE, 2011, pp. 702–707.

[30] V. K. Singh, M. Gao, and R. Jain, "Situation detection and control using spatio-temporal analysis of microblogs," in *Proceedings of the 19th international conference on World wide web.* ACM, 2010, pp. 1181–1182.

[31] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: real-time event detection by social sensors," in *Proceedings of the 19th international conference on World wide web.* ACM, 2010, pp. 851–860.

[32] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* ACM, 2014, pp. 147–156.

[33] I. Raina, S. Gujar, P. Shah, A. Desai, and B. Bodkhe, "Twitter sentiment analysis using apache storm," *Int J Recent Technol Eng*, vol. 3, no. 5, pp. 23–26, 2014.

[34] M. T. Jones, "Process real-time big data with twitter storm," *IBM Technical Library*, 2013.

[35] R. Ranjan, "Streaming big data processing in datacenter clouds," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78–83, 2014.

[36] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[37] M. Illecker and E. Zangerle, "Real-time twitter sentiment classification based on apache storm," 2015.

[38] V. Kovalev, A. Kalinovsky, and S. Kovalev, "Deep learning with theano, torch, caffe, tensorflow, and deeplearning4j: Which one is the best in speed and accuracy?" 2016.

[39] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv preprint arXiv:1511.06435*, 2015.

[40] B. J. Erickson, P. Korfiatis, Z. Akkus, T. Kline, and K. Philbrick, "Toolkits and libraries for deep learning," *Journal of digital imaging*, vol. 30, no. 4, pp. 400–405, 2017.

[41] C. Chen, Y. Yan, L. Huang, and L. Qian, "Implementing a distributed volumetric data analytics toolkit on apache spark," in *Scientific Data Summit (NYSDS), 2017 New York*. IEEE, 2017, pp. 1–8.

[42] J. Landthaler, B. Waltl, D. Huth, D. Braun, F. Matthes, C. Stocker, and T. Geiger, "Extending thesauri using word embeddings and the intersection method," 2017.

[43] K. Sid and M. Batouche, "Ensemble learning for large scale virtual screening on apache spark," in *Computational Intelligence and Its Applications: 6th IFIP TC 5 International Conference, CIIA 2018, Oran, Algeria, May 8-10, 2018, Proceedings 6*. Springer, 2018, pp. 244–256.

[44] N. Johnsirani Venkatesan, C. Nam, and D. R. Shin, "Deep learning frameworks on apache spark: A review," *IETE Technical Review*, pp. 1–14, 2018.

[45] V. Losing, B. Hammer, and H. Wersing, "Choosing the best algorithm for an incremental on-line learning task," 2016.

[46] G. A. Carpenter, S. Grossberg, and D. B. Rosen, "Fuzzy art: Fast stable learning and categorization of analog patterns by an adaptive resonance system," *Neural networks*, vol. 4, no. 6, pp. 759–771, 1991.

[47] D. A. Ross, J. Lim, R.-S. Lin, and M.-H. Yang, "Incremental learning for robust visual tracking," *International journal of computer vision*, vol. 77, no. 1-3, pp. 125–141, 2008.

[48] P. Joshi and P. Kulkarni, “Incremental learning: Areas and methods-a survey,” *International Journal of Data Mining & Knowledge Management Process*, vol. 2, no. 5, p. 43, 2012.

[49] A. Jain and A. Nalya, *Learning storm.* Packt Publishing, 2014.

[50] M. Soni, “Everything you need to know about apache storm,” February 2018. [Online]. Available: https://www.upgrad.com/blog/everything-you-need-to-know-about-apache-storm

[51] M. H. Iqbal and T. R. Soomro, “Big data analysis: Apache storm perspective.”

[52] L. Bottou, “Stochastic gradient descent tricks,” in *Neural networks: Tricks of the trade.* Springer, 2012, pp. 421–436.

[53] ——, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010.* Springer, 2010, pp. 177–186.

[54] H. Taud and J. Mas, “Multilayer perceptron (mlp),” in *Geomatic Approaches for Modeling Land Change Scenarios.* Springer, 2018, pp. 451–455.

[55] T. M. Mitchell *et al.*, “Machine learning. 1997,” *Burr Ridge, IL: McGraw Hill*, 1997.

[56] “Basics of neural networks in ai artificial intelligence,” http://kindsonthegenius.blogspot.com/2017/12/basics-of-neural-networks.html, 2017.

[57] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[58] “Apache spark architecture explained in detail,” https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338, 2018.

[59] “Distributed deep learning with dl4j and spark,” https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-intro, 2018.

[60] P. Jantke, “Types of incremental learning,” in *AAAI Symposium on Training Issues in Incremental Learning*, 1993, pp. 23–25.

[61] R. Polikar, L. Upda, S. S. Upda, and V. Honavar, “Learn++: An incremental learning algorithm for supervised neural networks,” *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, vol. 31, no. 4, pp. 497–508, 2001.

[62] S. Grossberg, “Nonlinear neural networks: Principles, mechanisms, and architectures,” *Neural networks*, vol. 1, no. 1, pp. 17–61, 1988.

[63] M. Tscherepanow, M. Kortkamp, and M. Kammer, “A hierarchical art network for the stable incremental learning of topological structures and associations from noisy data,” *Neural Networks*, vol. 24, no. 8, pp. 906–916, 2011.

[64] A. Gepperth and B. Hammer, “Incremental learning algorithms and applications,” in *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

[65] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit.* ” O’Reilly Media, Inc.”, 2009.

[66] L. Backstrom, E. Sun, and C. Marlow, “Find me if you can: improving geographical prediction with social and spatial proximity,” in *Proceedings of the 19th international conference on World wide web.* ACM, 2010, pp. 61–70.

[67] A. Sadilek, H. Kautz, and J. P. Bigham, “Finding your friends and following them to where you are,” in *Proceedings of the fifth ACM international conference on Web search and data mining.* ACM, 2012, pp. 723–732.

[68] C. A. Davis Jr, G. L. Pappa, D. R. R. de Oliveira, and F. de L. Arcanjo, “Inferring the location of twitter messages based on user relationships,” *Transactions in GIS*, vol. 15, no. 6, pp. 735–751, 2011.

[69] R. Gonzalez, G. Figueroa, and Y.-S. Chen, “Tweolocator: a non-intrusive geographical locator system for twitter,” in *Proceedings of the 5th ACM SIGSPATIAL international workshop on location-based social networks.* ACM, 2012, pp. 24–31.

[70] M. Oussalah, F. Bhat, K. Challis, and T. Schnier, “A software architecture for twitter collection, search and geolocation services,” *Knowledge-Based Systems*, vol. 37, pp. 105–120, 2013.

[71] B. Hecht, L. Hong, B. Suh, and E. H. Chi, “Tweets from justin bieber’s heart: the dynamics of the location field in user profiles,” in *Proceedings of the SIGCHI conference on human factors in computing systems.* ACM, 2011, pp. 237–246.

[72] M. Graham, S. A. Hale, and D. Gaffney, “Where in the world are you? geolocation and language identification in twitter,” *The Professional Geographer*, vol. 66, no. 4, pp. 568–578, 2014.

[73] S. Asur and B. A. Huberman, “Predicting the future with social media,” in *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology-Volume 01.* IEEE Computer Society, 2010, pp. 492–499.

[74] S. Asur, B. A. Huberman, G. Szabo, and C. Wang, “Trends in social media: persistence and decay.” in *ICWSM*, 2011.

Appendix A Workload Distribution

A.1 Ahmed Fahmin

Task Contribution: This member mainly focused on Apache Storm, literature review and incremental learning. He used Apache Storm to read tweets from Twitter API and build the proposed model. He also did prepossessing and applied natural language processing to extract featured from the content of a tweet. He also made the initial feature list from the initial dataset. Besides, he implemented the incremental learning part while working with DeepLearning4J.

Writing Contribution in the Report: Abstract, section: 1, 2, 3, sub-section: 5.3 (Incremental learning), 6, 7, 11 (Partially).

A.2 Mohammad Nafis Ul Islam

Task Contribution: This member mainly focused on DeepLearning4J framework and used it to implement the Multilayer Neural Network and the Distributed Learning Method, using Spark Platform on it. Besides merging the learner code with the Storm code, he also analyzed different characteristics of the initial raw dataset and extracted specific tweets, which were later used for feature extraction. He also did the analysis of the learning model, by running different experiments, using the pre-processed data; and finally came up with the results after choosing an optimal learner, based on the analysis.

Writing Contribution in the Report: Section: 4, 5 (except sub-section: 5.3), 8, 9, 10, 11, 12.

Appendix B Distributed Learning Brief Code Snippets

B.1 Tweets Filtering

```
    this.twitterStream = new TwitterStreamFactory(cb.build()).getInstance();
    this.queue = new LinkedBlockingQueue<Status>();
    final FilterQuery query = new FilterQuery();
    double[][] GEOBOX_MELBOURNE = {{140.3482679711,-39.7125077525},
                                   {149.6598486874,-32.0871185235}};
    query.locations(GEOBOX_MELBOURNE);
    query.language(new String[]{"en"});
    twitterStream.filter(query);
```

B.2 Configure Spark

```
public static JavaSparkContext configureSpark(){
    //Set up Spark configuration and context
    SparkConf sparkConf = new SparkConf();
    if (useSparkLocal) {
        sparkConf.setMaster("local[*]");
    }
    sparkConf.setAppName("Twitter Location Example");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);
    return sc;
}
```

This is done in the master node to create the spark context.

B.3 Configure Training Master

```
public static TrainingMaster
    configureTrainingMaster(int batchSizePerWorker) {
    TrainingMaster tm = new ParameterAveragingTrainingMaster.
    Builder(batchSizePerWorker)
            .averagingFrequency(5)
            .workerPrefetchNumBatches(2)
            .batchSizePerWorker(batchSizePerWorker)
            .build();

    return tm;
}
```

This training master defines how the distributed training will occur, by defining the worker node numbers, averaging frequency etc.

B.4 Transforming Dataset for Training

```
DataSet trainDataSet;
DataSet testDataSet;

if (testFile == null) {
    String trainingFileName = inputFile;

    int batchSizeTraining = getRowCount(trainingFileName);

    DataSet wholeDataSet = readCSVDataset(trainingFileName,
            batchSizeTraining, labelIndex, numClasses);

    SplitTestAndTrain testAndTrain =
        wholeDataSet.splitTestAndTrain(0.65);

    trainDataSet = testAndTrain.getTrain();
    testDataSet = testAndTrain.getTest();
}

else {
    String trainingFileName = inputFile;
    String testingFileName = testFile;

    int batchSizeTraining = getRowCount(trainingFileName);

    int batchSizeTest = getRowCount(testingFileName);

    trainDataSet = readCSVDataset(trainingFileName,
    batchSizeTraining, labelIndex, numClasses);
    testDataSet = readCSVDataset(testingFileName,
    batchSizeTest, labelIndex, numClasses);
}


DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainDataSet);
normalizer.transform(trainDataSet);
normalizer.transform(testDataSet);

List<DataSet> trainDataList = trainDataSet.asList();
List<DataSet> testDataList = testDataSet.asList();

JavaRDD<DataSet> trainData = sc.parallelize(trainDataList);
JavaRDD<DataSet> testData = sc.parallelize(testDataList);
```

This partial snippet gives us the idea about how the dataset is read and then converted to RDD format, which is required to do the training in Spark Architecture.

We first read the data from CSV files into normal DataSet objects. Next, we normalize the dataset. Then we convert them into lists. These data lists are finally used to build the JavaRDD datasets after parallelizing the lists using the spark context. This too, is done in the master node as described in section:5.2.2.

B.5 Configuring the final Distributed Learning Model

```
public static SparkDl4jMultiLayer
    configureLearningNetwork(JavaSparkContext sc,
    TrainingMaster tm) {
    long seed = 6;
    log.info("Build model....");

    MultiLayerConfiguration conf =
    new NeuralNetConfiguration.Builder()
    .seed(seed)
    .activation(Activation.TANH)
    .weightInit(WeightInit.XAVIER)
    .updater(new Sgd(0.1))
    .l2(1e-4)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(numInputs)
    .nOut(hiddenLayerNeuronNumber)
    .build())
    .layer(1, new DenseLayer.Builder().nIn(hiddenLayerNeuronNumber)
    .nOut(hiddenLayerNeuronNumber).build())
    .layer(2, new OutputLayer
    .Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
            .activation(Activation.SOFTMAX)
            .nIn(hiddenLayerNeuronNumber)
            .nOut(outputNum).build())
    .backprop(true).pretrain(false)
    .build();



    SparkDl4jMultiLayer sparkNetwork =
        new SparkDl4jMultiLayer(sc, conf, tm);

    return sparkNetwork;
}
```

Here, we first build the basic Multilayer Neural network of ours. Then, we supply that model to the SparkDL4jMultiLayer network, which is basically a wrapper, that enable the distributed functionality for training. We also need to give the spark context and training master to it.

B.6 Distributed Training

```
for (int i = 0; i < numEpochs; i++) {
    sparkNetwork.fit(trainData);
    log.info("Completed Epoch {}", i);
}
```

```
TrainingModel trainingModel = new
TrainingModel(sparkNetwork, testDataSet, testMetaData);

log.info("\n\nExample complete");

return trainingModel;
```

This partial code snippet comes after the snippet given in section: B.4. The sparkNetwrok is the SparkDL4jMultiLayer network, which fits the model distributively(by distributing to the worker nodes from the master) using the RDD trainData.