



**VirginiaTech**  
*Invent the Future*

**Final Project Report — CS 5604  
Information Storage and Retrieval  
CLA Team, Fall 2016**

December 7, 2016

Blacksburg, VA 24061

**CLA Team:**

Saurabh Chakravarty

Eric Williamson

{saurabc,ericrw96}@vt.edu

**Project Advisor:**

Prof. Edward A. Fox

## Abstract

Content is generated on the web at an exponential rate. The type of content varies from text on a traditional webpage to text on social media portals (e.g., social network sites and microblogs). One such example of social media is the microblogging site Twitter. Twitter is known for its high level of activity during live events natural disasters, and events of global importance.

Improving text classification results on Twitter data would pave the way to categorize the tweets into human defined real world events. This would allow diverse stakeholder communities to interactively collect, organize, browse, visualize, analyze, summarize, and explore content and sources related to crises, disasters, human rights, inequality, population growth, resiliency, shootings, sustainability, violence, etc.

Challenges with the data in the Twitter universe include that the text length is limited to 160 characters. Because of this limitation, the vocabulary in the Twitter universe has taken its own form of short abbreviations of sentences, emojis, hashtags, and other non-standard usage of written language. Consequently, traditional text classification techniques are not effective on tweets.

Sophisticated text processing techniques like cleaning, lemmatizing, and removal of stop words and special characters will give us clean text which can be further processed to derive richer word semantic and syntactic relationships using state of the art feature selection techniques like Word2Vec. Machine learning techniques using word features that capture semantic and context relationships have been shown to give state of the art classification accuracy.

To check the efficacy of our classifier, we would compare our experimental results with an association rules (AR) classifier. This classifier composes its rules around the most discriminating words in the training data. The hierarchy of rules along with an ability to tune to support threshold makes it an effective classifier for scenarios where short text is involved.

We developed a system where we read the tweets from HBase and write the classification label back after the classification step. We use domain oriented pre-processing on the tweets and Word2Vec as the feature selection and transformation technique. We use a multi-class Logistic Regression algorithm for our classifier.

We are able to achieve an F1 score 0.96 for our classifier for classifying a test set of 320 tweets across 9 classes. The AR classifier achieved an F1 score of 0.90 on the same data. Our developed system can classify collections of any size by utilizing a 20 node Hadoop cluster in a parallel fashion, through Spark.

Our experiments suggest that the high accuracy score for our classifier can be primarily attributed to the pre-processing and feature selection techniques that we used. Understanding the Twitter universe vocabulary helped us frame the text cleaning and pre-processing rules used to eliminate noise from the text. The Word2Vec feature selection technique helps us capture the word contexts in a low dimensional feature space that results in high classification accuracy and low model training time. Utilizing the Spark framework to execute our classification pipeline in a distributed fashion allows us to classify large collections without running into out-of-memory exceptions.

# Table of Contents

List of Figures .....	6
List of Tables .....	7
1 Introduction.....	8
2 Literature Review.....	8
2.1 Textbook.....	8
2.2 Papers .....	9
3 Requirements .....	10
4 Design .....	12
5 Implementation .....	16
5.1 Environment.....	16
5.2 Training data .....	16
5.3 HBase access .....	17
5.4 Cleaning .....	18
5.5 Feature selection and transformation .....	19
5.6 Association Rules classifier .....	22
5.7 Classifier training and prediction.....	22
5.8 Emitting probability in a multi-class scenario.....	25
5.9 Spark partitioning and caching.....	27
5.10 System extensibility.....	28
5.11 Scheduled execution of the classification pipeline with a cron job.....	29
6 Experiments .....	30
6.1 Cleaning Experiment.....	30
6.1.1 Experimental setup.....	30
6.1.2 Experimental results.....	31
6.2 Association Rules support threshold experiment.....	32
6.2.1 Experimental setup.....	32
6.3 Word2Vec based Logistic Regression classifier .....	33
6.3.1 Experimental setup.....	33
6.3.2 Experimental results.....	34
6.3.3 Test of significance.....	35
6.3.4 Probability Experiment.....	35

6.3.5	Experimental setup.....	35
6.3.6	Experimental results.....	35
6.3.7	Inter-classifier mutual agreement .....	37
6.4	Runtime Comparison Experiment.....	38
6.4.1	Experimental setup.....	38
6.4.2	Experimental results.....	39
7	Timeline .....	40
8	User Manual.....	42
8.1	Environment Setup.....	42
8.2	Project Layout .....	42
8.2.1	Uploaded files .....	42
8.2.2	HDFS files .....	44
8.3	Generating training data for files .....	45
8.3.1	Github training data .....	45
8.3.2	HBase training data.....	45
8.4	Running the Association Rules classifier.....	46
8.5	Running the classifier.....	48
8.6	Configuring the cron job .....	50
9	Developer Manual.....	52
9.1	VTechWorks Inventory.....	52
9.2	Source Directory .....	52
9.3	Training data .....	53
9.4	Parameters for the Association Rules classifier .....	53
9.5	HBase Reading.....	54
9.5.1	Reading Prediction Data .....	54
9.6	Cleaning .....	56
9.7	Writing to HBase.....	57
9.8	Word2Vec Generation.....	59
9.9	Classification.....	60
9.10	Probability emission by the classifier.....	62
9.11	Spark partitioning and caching.....	63
10	Conclusion .....	64

11	Future Work .....	65
12	Acknowledgements.....	66
13	References.....	67

# List of Figures

Figure 1: Problem Statement .....	11
Figure 2: High level architecture .....	12
Figure 3: High level view .....	13
Figure 4: Training phase .....	13
Figure 5: Prediction phase .....	14
Figure 6: Data pre-processing before classification.....	16
Figure 7: HadoopRDD usage example .....	17
Figure 8: Scan usage example.....	18
Figure 9: Example raw tweet .....	18
Figure 10: Example cleaned tweet.....	19
Figure 11: A neural language model for Word2Vec [17].....	20
Figure 12: The hidden layer weight matrix [17].....	21
Figure 13: Word2Vec usage example.....	21
Figure 14: Sample output for the code example shown in Figure 13 .....	22
Figure 15: Logistic Regression training example .....	24
Figure 16: Logistic Regression prediction example .....	24
Figure 17: Spark example to generate probabilities for the tweets along with the predictions ....	26
Figure 18: Emission of probabilities along with the prediction for a sample tweet .....	26
Figure 19: Spark partitioning example .....	27
Figure 20: Spark caching example.....	28
Figure 21: Spark jobs being executed in a parallel fashion [19].....	28
Figure 22: Class distribution in the experiment sample.....	31
Figure 23: Accuracy experiment data generation .....	34
Figure 24: Average classifier accuracy results .....	34
Figure 25: Probability distribution of predicted tweets .....	36
Figure 26: Multi-class assignment distribution.....	37
Figure 27: Formula to compute kappa .....	38
Figure 28: Performance of the classifiers on different number of tweets.....	39
Figure 29: Performance of the classifiers including optimization .....	40
Figure 30: Repository directory tree .....	42
Figure 31: HDFS commands .....	44
Figure 32: Running the AR classifier .....	46
Figure 33: Results of running the Association Rules classifier .....	47
Figure 34: Output directory files.....	47
Figure 35: Association Rules evaluation output .....	48
Figure 36: Hue interface example.....	49
Figure 37: HUE timestamp viewing example.....	50
Figure 38: Parameter placeholder specifications for the cron job [20].....	51
Figure 39: Usage examples for the crontab entry [20] .....	51
Figure 40: Console output for the list of cron jobs .....	51
Figure 41: Parameterized Asscoaiton rule classifier command.....	53

Figure 42: Command to run the classifier.....	54
Figure 43: Configure scan for HBase reading .....	55
Figure 44: Batched processing code example.....	55
Figure 45: Conversion from HBase row to tweet example.....	56
Figure 46: Cleaning class entry point .....	56
Figure 47: '#' character removal.....	56
Figure 48: Stanford NLP example .....	57
Figure 49: Writing tweets to the database.....	58
Figure 50: Disposing of the HBaseInteraction object.....	58
Figure 51: Code for label mapping .....	59
Figure 52: Code snippet for Word2Vec training .....	59
Figure 53: Feature transformation for tweet text .....	60
Figure 54: Implementation code snippet for the classifier.....	60
Figure 55: Implementation code snippet for generating classifier metrics.....	61
Figure 56: Classifier metrics per class.....	61
Figure 57: Confusion Matrix and overall results for the classifier .....	62
Figure 58: Use of ClassificationUtility.scala.....	62
Figure 59: Probability normalization .....	62
Figure 60: Spark UI example.....	63
Figure 61: Port forwarding command to configure access to the Spark UI.....	63

## List of Tables

Table 1 Comparison of classification algorithms .....	23
Table 2 Training data schema .....	29
Table 3 Column mapping between training table label and real world event .....	29
Table 4 Real World Events.....	31
Table 5 Cleaning experiment results.....	32
Table 6 Number of tweets classified by support thresholds .....	33
Table 7 Comparative experiment results .....	35
Table 8 Kappa inter-classifier agreement .....	37
Table 9 Kappa agreement definitions .....	38
Table 10 Timetable of tasks.....	40
Table 11 Description of files in base directory.....	43
Table 12 Description of files in the data directory .....	44
Table 13 Mapping from label to event.....	45
Table 14 HBase training data schema.....	46
Table 15 ClassifyCollection.sh parameters.....	49
Table 16 src directory files .....	52
Table 17 data_scripts directory files .....	53
Table 18 Association rules classifier parameters.....	54

# 1 Introduction

The goal of the classification team is to take tweet collections and classify them as relevant or non-relevant to specific classes or topics. We will place these classification results into a database (HBase) table as a column family for use by the other teams. The classification results will be indexed by the SOLR team and allow other teams such as the Front End team to use the indexes. As we are classifying tweets we will be making use of the data from the Collection Management Tweets (CMT) team. This CMT pipeline takes the raw tweet data and pre-processes it to remove obvious spam, vulgarities, and unreadable text, and then uploads the processed tweet text to the database for us to use.

We begin in Section 2 by discussing the relevant information on classification that we have gained from the course textbook, the past team's report, and relevant papers. This helped us study the current state of the art classification techniques for our document collections while also pointing to a variety of different feature selection and classification methods that we will be able to use. We then outline the specific problem and requirements for this project team in Section 5.

In Section 4 we discuss our classification system design at a high level, and go into the details of how it was implemented, as well as the results from our experiments in Section 6. We lay out the weekly timeline of work done during the semester in Section 7.

We then provide manuals for both users and developers so others can use and expand on this project after it has been completed. The User Manual is in Section 8 and will guide setting up Spark and running the classifiers on the sample data given. The Developer Manual in Section 9 goes into detail about the codebase, with specifics on how it can be run on additional datasets, and how it can be extended by future groups. Sections 10, 11, and 12 describe the conclusion, future work, and acknowledgements, respectively.

## 2 Literature Review

### 2.1 Textbook

The textbook [1] introduces the classification problem we are trying to solve: Given a document and a set of classes, what is the subset of those classes that this document belongs to? It also discusses the different feature selection methodologies for text classification. These features are then used in the training of the classification methods discussed. The classification methods described in the book are Support Vector Machine, Naive Bayes, and Vector Space Classification. The textbook helps us get a head start into the problem from a breadth perspective and gives us a platform so we can start studying more recent techniques on feature selection, vector space representation of words, and classification, from the latest research literature in the area.



## 2.2 Papers

The process of text classification involves extracting features out of the text data and finding an appropriate vector space representation for it. This is the feature selection stage of the process. Once the feature representation of a piece of text is obtained, it can be fed into any classification algorithm like logistic regression or SVM, and be trained. This is the classification stage of the process. As part of our literature survey, we came across the following feature selection methods.

1. **Chi-squared statistic** - This technique [2] draws its discriminative ability through analysis of the independence between terms and the documents. The key feature of the technique is that it can reduce the dimensionality of the feature space to ensure a high performance of the classifier.
2. **Mutual Information** - This technique [1] is used to measure the global goodness of a term in feature selection. This technique measures the mutual information between a term and a class. The sequence of the words is ignored; a bag of words representation is employed.
3. **Tf-idf** - Term frequency–inverse document frequency is a statistic that measures how important a word is to a specific document [1]. This value increases as the term appears more frequently but is scaled by how often the term appears in the entire corpus. This makes terms that are common throughout the text such as “the” to not be weighted heavily even though it would appear many times in each specific document.
4. **Information Gain** - This is another technique to measure the goodness criterion. It measures the number of bits required for category prediction by knowing the presence or the absence of a term in the document [1].
5. **Word Class Popularity** - This technique explores the relative distribution of a feature among the different classes. The goal of this technique is to identify the features that discriminate the classes the most. A good discriminant term will have a skewed distribution across classes. This technique uses the gini coefficient of inequality to analyze the distribution of a feature across the classes [3].
6. **Word2Vec** - This technique [4, 5] generates the word vectors out of the training corpus based on the context in which they occur. The context of the word is defined as the word and its surrounding neighbors. For example, a word might have words preceding it and succeeding it. The word along with its surrounding neighbors form the context, though the length of the window in which the neighbors are defined is a parameter that can be tweaked. Two techniques that identify the context of a word are the CBOW (continuous bag of words) and the skip-gram method. The skip-gram based technique predicts the surrounding words given the current word. The CBOW technique predicts the current word given the surrounding words. A neural network is trained based on these techniques and the trained hidden layer weights are used to generate the word vectors. Once the word vectors are generated, the words that are closer in context to each other are close to each other in vector space based on their cosine distances. This attribute of the word vectors makes them very useful for text classification. The word vectors based on the skip-gram technique give the state of the art results [6] for text classification. From a feature selection perspective, since each word generates a vector, we average all the values in the vector and use that value as

the feature value for a given word. The bigger the corpus is, the more effective the word vectors become from the perspective of text classification.

The choice of classifier in our case is not that important since we can fairly easily generate a moderate amount of training data. We started off our experimentation with a simple implementation of multi-class logistic regression. As part of our literature survey, we came across the following classification algorithms.

1. **Logistic Regression** - Logistic regression [1] measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function, which is the cumulative logistic distribution.
2. **Support Vector Machine (SVM)** – This is a classifier that will perform linear or nonlinear classification through a kernel trick [1]. This classifier works by linearly separating the classes so that each class falls onto one side of the separator. This classifier maximizes the distance between the separator and the points on either side to identify the separator it will use.
3. **Multi-layer Perceptron (MLP)** - It is a feedforward artificial neural network [7] model that maps sets of input data onto a set of appropriate outputs. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. It uses the supervised learning technique named back-propagation to learn the network weights.
4. **Naïve Bayes** – This classification method is based on applying Bayes Rule under the assumption of independence. This means it treats each feature as independent of the others with respect to the class the document will fall into. Despite the assumptions, Naïve Bayes has been shown to perform well in real world situations [1].
5. **Association Rules** - This technique uses the training data to create association rules for each class by identifying rules that will lead to a specific class identification [8]. The rules can then be used by a rule engine to predict the class of new documents. These association rules can also more quickly predict the classes of documents. The challenge with this classifier is to see how well it performs on long texts since its efficacy has only been evaluated on short texts.

### 3 Requirements

The problem statement for the classification team is as follows:

*Given a tweet collection and a set of event classes in the real world, we are to build a classifier that can classify the tweets into the appropriate event class.*

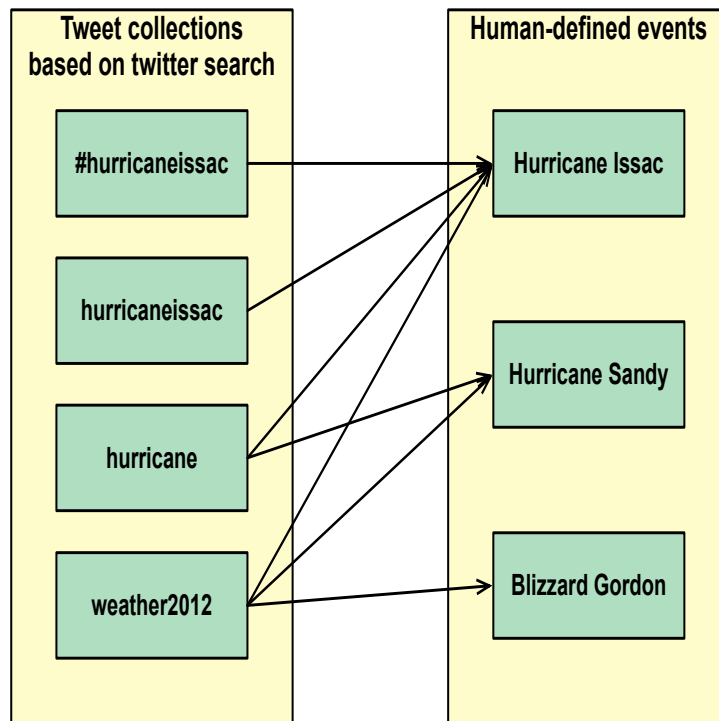


Figure 1: Problem Statement

Figure 1 explains the problem pictorially. Essentially, we have a set of collections of tweets that have been retrieved based on keyword/tag search performed using the Twitter API. These are shown in the box in the left section of Figure 1. The human defined events or the real-life events as stated in the goal are shown in the box in the right section of Figure 1. The relationship between the collection of tweets and the events is many-to-many.

For instance, the tweet collection weather2012 can have tweets related to the hurricanes “Sandy” and “Isaac” as they occurred in the same year and tweets from this collection can map to either the “Hurricane Sandy” or “Hurricane Isaac” event on the right. Likewise, for a given event, there can be many tweets that are associated with the event.

For the task of classification, we make the following assumptions about the collection of tweets:

- Tweets have been extracted and are available in the database (HBase) and some “basic” SPAM check has been done by the tweet collection management team or by the teams from previous offerings of this course.
- We provided the SOLR team with accompanying classification labels for tweets in HBase. Depending on the classification method used we were able to provide the respective probabilities of the tweet belonging to each of the real-world events we classified it as. The output of the classification step was written in the database which is used by the SOLR team. We added the classification label of a tweet to the “real-world-events” column in the “clean-tweet” column family.

- Due to the size of the classification team, this semester we only focused on the classification of tweets. Also, if any tweet contained an embedded hyperlink, we did not process the contents of the page that the hyperlink points to since our classification techniques were limited to short text data.

As part of the classification effort, we achieved the three major goals that are described as follows.

1. Develop a classifier that utilizes an effective feature selection and transformation technique along with a suitable classification algorithm to achieve high classification accuracy and run-time performance.
2. Develop the solution to be scalable and performant. It should be able to process large collections (tens of millions of tweets) and persist the classification label in HBase.
3. Develop the solution to be extendable in the future. This includes adding the flexibility to train the classifier on new classes and also be able to run the classification pipeline as a scheduled job that can process the new tweets as they are added in HBase.

## 4 Design

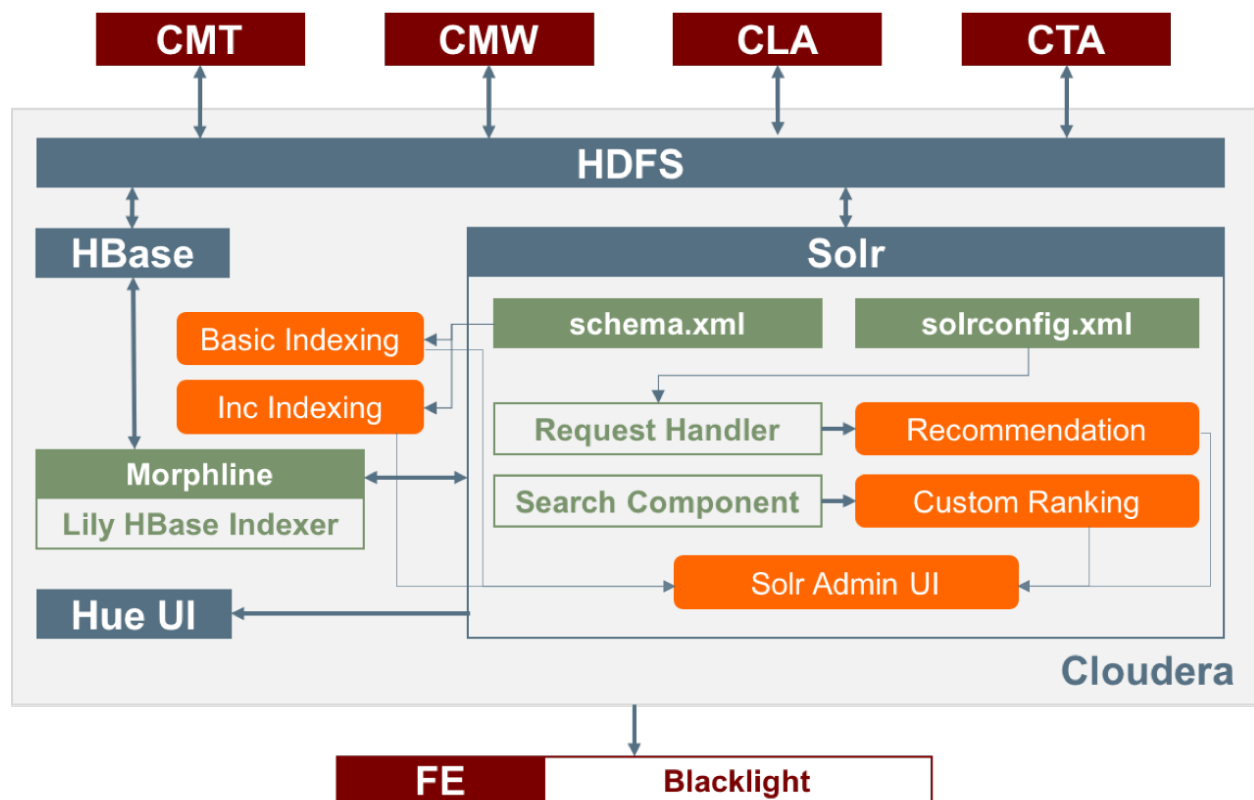


Figure 2: High level architecture

Figure 2 shows the high-level architecture for all the components for the class. The classification component is shown along with other components that are part of the whole system that the class built during the semester.

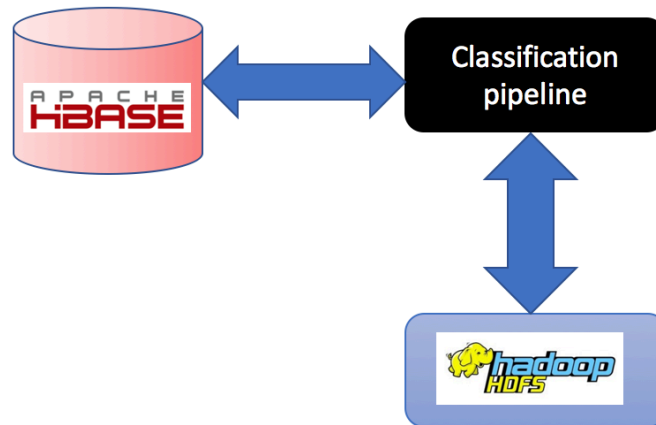


Figure 3: High level view

In Figure 3, we have the high level black box view of the classification pipeline and the data sources that it interacts with. The classification pipeline will access the HBase database to read the raw tweets from the table *ideal-cs5604f16* and write back the classification label to the **real-world-events** column in the same table. This column is part of the column family **clean-tweet** that was created for storing the classification related columns.

The process of classification has two major phases.

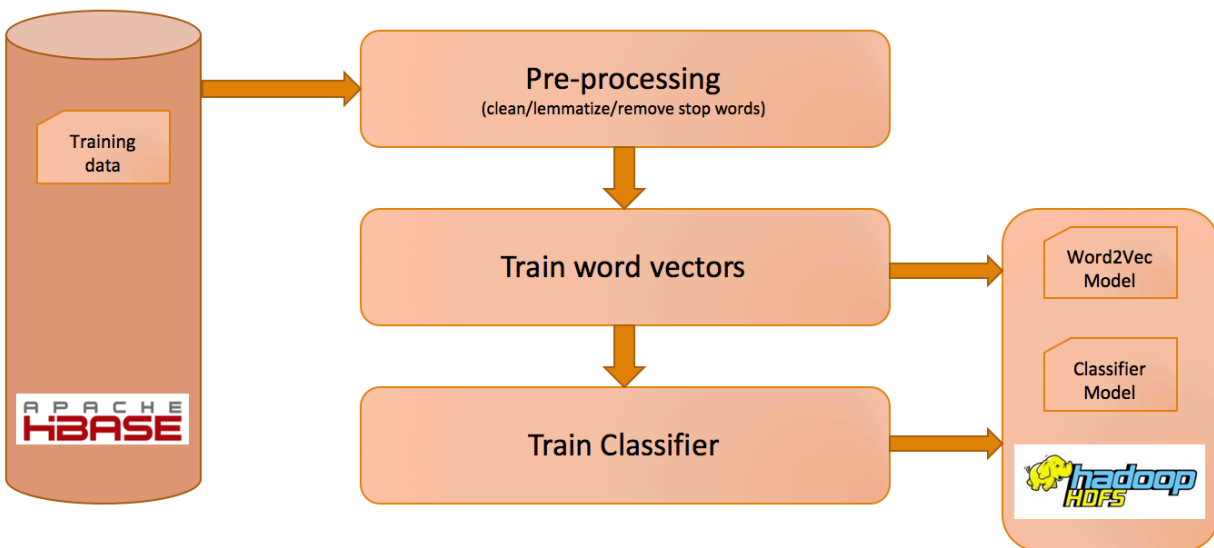


Figure 4: Training phase

1. **Training** – In this step we read<sup>1</sup> the training data from HBase that we generated<sup>2</sup> to train a classifier. As shown in Figure 4, we read the raw tweet from HBase and then performed

<sup>1</sup> The reading of the data from HBase can be found in Section 5.3

<sup>2</sup> The generation of training data can be found in Section 5.2.

some pre-processing on it. We used Word2Vec as the feature selection method for our classification pipeline. Once we cleaned the tweets, we generated a Word2Vec [5] model to get the word vectors for each tweet in the training data. After the generation of the word vector model off the training data, we persisted the Word2Vec model file in HDFS.

In the classifier training phase, we transformed a tweet to a feature array based on the word vectors that we calculated for each word. We trained the classifier using these features. Once we trained our classifier, we persisted the model file in HDFS. For our project, we used the Logistic Regression classifier.

The training pipeline is executed in an offline manner. This means that we generated the word vector and classifier models beforehand to ensure that this step is not repeated during run-time. All the software artifacts(models) generated out of this phase that are persisted in HDFS were used later in the prediction phase which runs online.

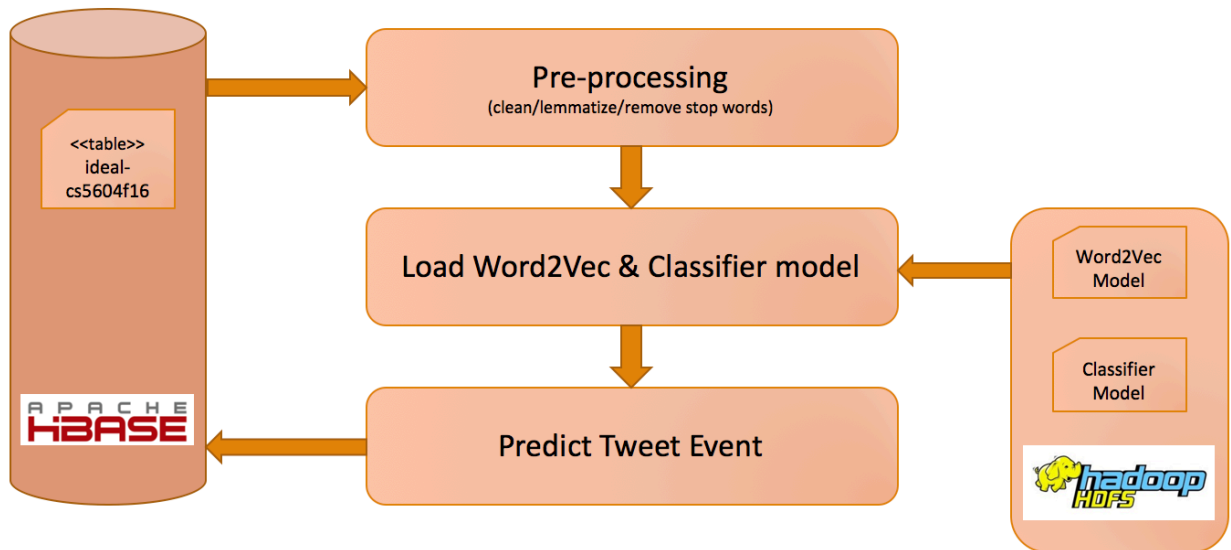


Figure 5: Prediction phase

2. **Prediction** – This phase of the pipeline runs online periodically as a timer-based job<sup>3</sup> in Linux. We used the cron utility in Linux to configure the job to be run once every 4 hours. As part of the prediction phase shown in Figure 5, a block of tweets is read from the table *ideal-cs5604f16* in HBase and labelled by the classifier. This label is written back to the same table in HBase. The Word2Vec and Logistic Regression models are loaded from HDFS in the beginning of the phase and are persisted in memory till the end of the prediction phase.

<sup>3</sup> The cron job details can be found in Section 5.11

Based on our research, we used the high-level approach as shown in Figure 4 to identify the best choice of the feature selection and classification techniques. We performed our experiments in the following way.

- **Training data creation** - We generated training data that by selecting 3 broad categories and then selecting tweets from the 3 sub-categories each from the collection. We manually annotated the data and divided it into a ratio of 70:30 for the train/test mix for our experiments.
- **Classification via Association Rules** - We used the association rules based classifier [8] and generated a baseline result. We compared the results of this baseline with our Word2Vec with Logistic Regression classifier.
- **Feature selection via Word2Vec method** - We used this technique to generate the word vectors [4, 5] on the same training data and generate features for the tweet texts using the word vector model that we have generated from the training corpus.
- **Classification via Logistic Regression** – We used the multi-class logistic regression classifier to train a model based on the Word2Vec feature selection technique. We also performed a 10-fold cross-validation to select the best model and save it into HDFS. We performed this step so that we can load the best model out of the file system instead of training the classifier again. This helps in reducing runtime.
- **Evaluation of results** – Since we are implementing a multi-class classifier, we computed the micro-F1 scores [9] across all classes to evaluate the overall classification efficacy of the classifier.
- **Writing to database** - The classification results for tweets were recorded in the database by writing, to a column family for each tweet, the real-world events we have determined it belongs to.
- **System Extensibility** - To keep our system extensible, we loaded our annotated training data in a table in the database. This will allow anyone to add more training data in the future and use it to retrain the classifier. We also implemented the ability to generate a new set of word vectors using the same approach.
- **Pre-processing the training data** – To aid faster development of the classification system, we cleaned the tweets for our training data. This flow is shown in Figure 6. We remove all the non-English words, short URLs, and emojis as part of this process. We also interpreted the hashtag and mentions into terms by breaking them into multiple words based on the casing. An example would be the phrase “#HurricaneSandy” would be broken into the tokens “hurricane” and “sandy”. We also removed the stop words and lemmatized all the words that remain. We assume though that the cleaning of tweets in the final system will be done by the CMT team as part of their system implementation.

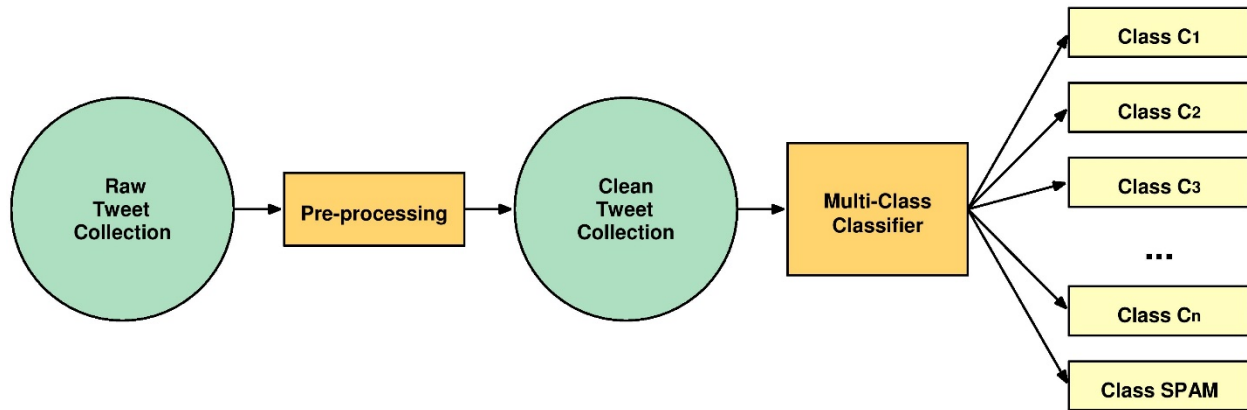


Figure 6: Data pre-processing before classification

We used the following technologies and frameworks for our project.

1. **Apache Hadoop** – This is the base layer of the distributed computing framework that we used. [10]
2. **Apache Spark** – This is an optimized RDD based framework built on top of Hadoop. [11]
3. **HBase** – This is the distributed database from the Apache Hadoop stack that is widely used as a NOSQL database in many implementations. [12]
4. **HDFS** – This is the distributed file system of the Apache Hadoop stack. [13]
5. **Spark MLlib** – This is a machine learning library that is based on the Apache Spark framework. For all the classification work related to our project, we used this library. [14]

## 5 Implementation

In this Section, we describe the implementation details of our system for our project this semester.

### 5.1 Environment

To rapidly develop the code needed for the project we opted for a hybrid environment between the DLRL cluster and our local machines. Fast iterative development was done on our local machines where we could take advantage of tools such as the IntelliJ IDEA [15] to quickly develop our Scala code. We could verify that the code is working on small datasets on our local machines, and then move the code up to the cluster when we ran classifications on the large datasets. This allowed us to take advantage of the processing power of the cluster when we needed to run our experiments. From the cluster, we were able to communicate with the database to store and retrieve data from the IDEAL and GETAR project collections.

### 5.2 Training data

To be able to build classifiers we had to create training and test sets to train and evaluate our classifiers. To begin we took data from the database present in the column family “cleantext” and assume that that data has been cleaned of profanity and unreadable text.

For our initial studies, we took a size 200 random sample of the documents (tweets) from each of the collections to form the basis for different classes. We then hand labeled the documents with the class that they belong to.



For our training sets we removed all stop words and performed lemmatization after seeing a better performance of all classifiers with lemmatization in place. We show the comparative performance results of the classifiers against clean and raw data in Section 5.3.

For our comparative tests, we split this labeled data -- 70% training, 30% testing -- to keep our comparisons consistent so we could evaluate the best classification result for our data.

We have provided documentation in Section 8.3 to allow for generation of more training data if desired.

### 5.3 HBase access

HBase is the database that is being used to store all of the tweets that our classifiers will run on. For our classification to operate satisfactorily we must read the tweets from HBase efficiently and correctly. Spark provides many ways to read records from HBase such as Scan that allows iteration over an HBase table, and HadoopRDD that reads HBase data into an RDD for further processing.

To use the HadoopRDD API one first specify the table name that you want to generate an RDD of. The API only supports making an RDD out of a single table, to get multiple tables you will have to create multiple RDDs. The next parameters that must be specified are the type of the data that each row-key is stored as in the HBase table. Finally, you can specify the columns that you want to retrieve from HBase.

The HadoopRDD API works by streaming the data to the driver node, then partitioning it across the cluster so each node can get a different piece of the data. This means that then you can execute operations on that data across the cluster. Example code for how a HadoopRDD will be created can be seen in Figure 7.

```
// This will store any configuration for what we read from HBase
val conf = HBaseConfiguration.create()

// configure the name of the HBase table to read
conf.set(TableInputFormat.INPUT_TABLE, "Table Name")

// the RDD to be retrieved from HBase
val hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],
  classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable], // key type
  classOf[org.apache.hadoop.hbase.client.Result]) //result type
```

*Figure 7: HadoopRDD usage example*

The Scan API works similarly to the HadoopRDD API. It requires the table name to be specified along with the columns to read and any filtering of records that you want to do. The Scan API returns a Result object. This object allows you to iterate over these records without drawing many into memory. This operation is not parallelized and would be executed on the driver node, so to achieve parallelism for the later operations you have to take some records and parallelize them into an RDD. An example framework for showing how this is set up can be seen in Figure 8.

```

// a scan object that will specify the columns to scan
val scan = new Scan()

// a reference to the specific HBase table we will be reading from
val table = new HTable(HBaseConfiguration.create(),"table name")

// add the specific column to read from the table
scan.addColumn(Bytes.toBytes("column family"), Bytes.toBytes("column"))

// get an iterator over the records in the table matching the scanner
val resultScanner = table.getScanner(scan)

val batchSize = 5000;
while (!complete){
    // read one batch into memory to process
    val results = resultScanner.next(batchSize)
    // process results
}

```

Figure 8: Scan usage example

One challenge that we faced is that there are collections of tweets that have millions of records. Because of this our read code must be able to still work efficiently when operating on the large record sizes. We found empirically that the HadoopRDD API did not finish reading after multiple hours when run on a collection of size greater than 1 million records. This led us to use the Scan API and read in small sections of the large collection at a time. By reading in a small block, parallelizing it across the cluster, then only reading the next one once we had finished processing the previous block, we were able to keep our code running in parallel while eliminating memory errors that large collections generate.

## 5.4 Cleaning

Twitter limits the number of characters in each message. This means that each document we want to classify has a limited amount of information. To be able to correctly classify these tweets we clean the data of non-discriminative stop words, perform lemmatization, and remove non-English characters such as hashtags ‘#’ and URLs.

RT: @AssociationsNow A Year After Texas Explosion Federal Report Outlines Progress on Fertilize... <http://t.co/8fDbMu9asU> #meetingprofs

Figure 9: Example raw tweet

An example uncleaned raw tweet can be seen in Figure 9. This tweet has a short URL that is irrelevant to the class it is part of. The words in this tweet are also capitalized, and we would like to have them still match with words of other tweets that are lowercase. It is also important to lemmatize words so that the features that the classifier will train on will consist of the word lemmas. [16] An example of the raw tweet from Figure 9 after being cleaned is shown in Figure 10.

year texas explosion federal report outline progress fertilize meetingprof

*Figure 10: Example cleaned tweet*

We utilized these cleaning methods on our training and test data for our experiments, as well as on the new tweets that we predict when reading from HBase. The specific details on how we accomplished each cleaning method can be found in Section 9.6. The accuracy gain that the classifiers experienced when they ran on cleaned data can be found in Section 6.1.

## **5.5 Feature selection and transformation**

As part of the pre-processing phase for our project, we clean the tweets, lemmatize the words contained, and remove the stop words. In spite of this, the number of words in a bag of words representation is still large. Feature selection methods assist in further reducing the dimensionality of the feature set by removing the irrelevant words. The goal of reducing the curse of dimensionality is to improve classification accuracy and reduce over fitting.

Methods for feature subset selection for text document classification use an evaluation function that is applied to a single word. The goal is to identify a subset of words that assist in discriminating between the classes the most. Techniques like Document frequency (DF), Term frequency (TF), Mutual information (MI), Information gain (IG), and Chi-square statistic (CHI) use feature-scoring methods to rank the features by their independently determined scores, and then select the top scoring features.

Another technique to reduce the size of the feature space is referred to as feature transformation. This approach does not eliminate features because of their low scores, but compacts the feature dimension based on feature concurrencies.

Words are central to text classification. The challenges with traditional feature selection techniques are that they are based on a bag-of-words representation. This representation fails to capture the neighboring context of a word in a sentence. The absence of this context results in the loss of the semantic relationship of the word with its neighboring words.

Word embeddings [4] offer distributional features about words. They capture the context of the word in its neighborhood. This results in an extension to the bag-of-words representation along with context and word sense information. Word embeddings are low-dimensional, dense vector representation of words. The compact representation along with capturing of the context make this a strong choice for the feature representation for words in text classification scenarios.

Work in [5] defines specific objective functions for efficient training of word embeddings, by simplifying the original training objective of a neural objective model. The two variants of the objective functions are as follows:

- a. **Continuous bag-of-words (CBOW)** – Given a word, predict the context.
- b. **Skip-gram** – Given a context, predict the word.

Figure 11 shows the Word2Vec CBOW neural language model. It is a one layer, 300-neuron neural network with  $[1 \times V]$  Boolean vector as input and a  $[1 \times V]$  float vector as output, where  $V$  is the vocabulary size, in this case 10000. The input to the neural network is a one-hot representation of a word in the form of a  $[1 \times V]$  Boolean vector. The word “ant” is the given word in the example, and the neural network objective is to maximize the probability of the words that could be its neighbors. The words are fed into the neural network from a training corpus and it generates the relative probabilities for all the words in the corpus.

The goal of the Word2Vec implementation is to just store the word weights that are in the hidden layer representation of neural network. As for the example given in Figure 11, the output of importance is the  $[300 \times 10000]$  matrix that gets generated for the word corpus, which in this case consists of 10000 words. This is shown in Figure 12.

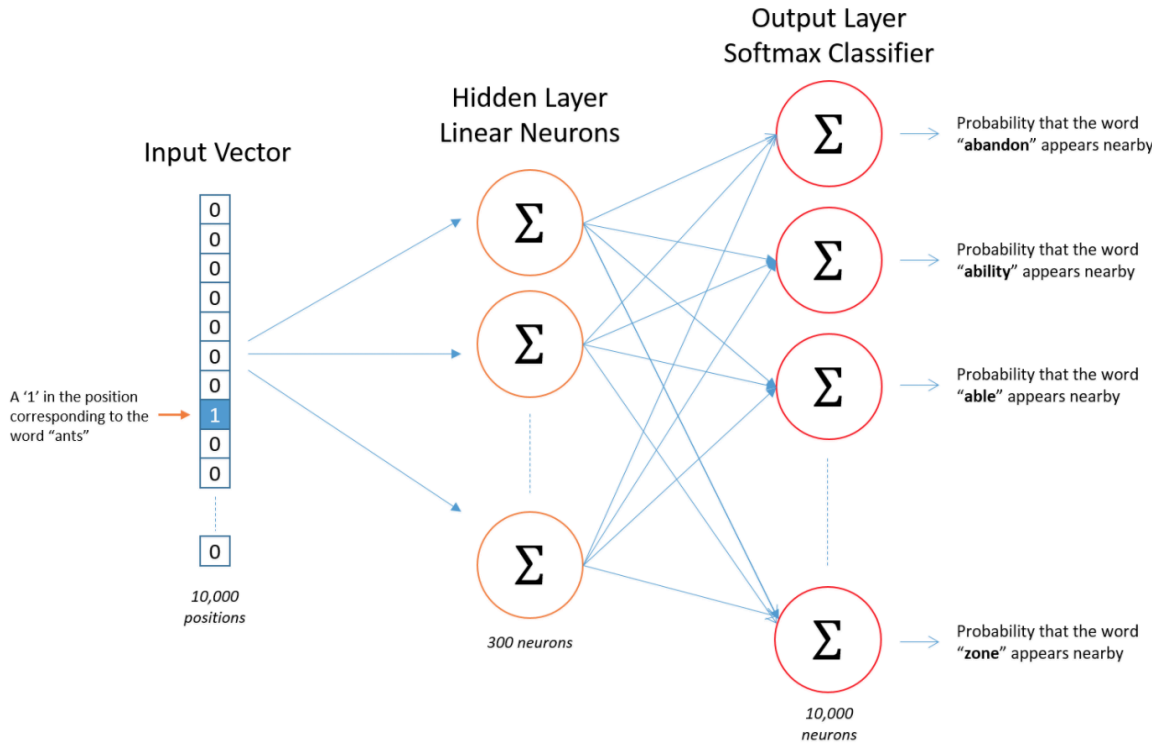


Figure 11: A neural language model for Word2Vec [17]

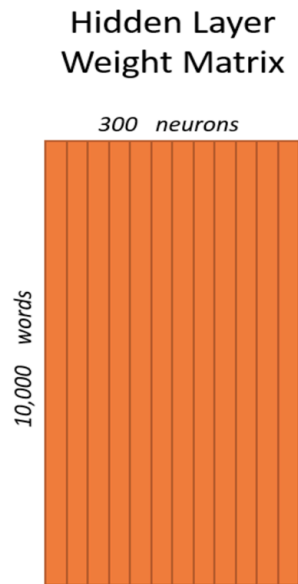


Figure 12: The hidden layer weight matrix [17]

Figure 13 shows an example on how to load a corpus from text file and train a Word2Vec model. Once the model is generated, it can be used to transform a word to its corresponding feature representation. Also, it can be used to find synonyms for any word that it has been trained on.

```

1 |
2 | //Load the tweets from a text file
3 | val trainTweets = sc.textFile("data/trainTweets.txt")
4 |
5 | //Train a word2vecModel from the tweet RDD
6 | val word2vecModel = new Word2Vec().fit(trainTweets.values)
7 |
8 | //Display the word vectors for the word "hurricane"
9 | println(word2vecModel.transform("hurricane"))
10 |
11 | //Find the synonyms for the word "shooting"
12 | println(word2vecModel.findSynonyms("shooting", 4))
13 |
14 | //Trigger the action to execute the code.
15 | trainTweets.collect

```

Figure 13: Word2Vec usage example

Figure 14 shows the part of output for the word vector for the word “hurricane” and the 4 synonyms for the word “shooting”.

```
[-0.18926458060741425,-0.04260385408997536,0.2629537880420685,0.1021256297826767,
0.1934381127357483,0.13710986077785492,-0.06425432860851288,0.10318592935800552,
0.023081598803400993,0.057771969586610794,0.025135912001132965,0.13008072972297668,
-0.01222849078476429,-0.06492101401090622,-0.06662728637456894,-0.06422432512044907,
-0.11410374194383621,-0.017767108976840973,0.018755711615085602,0.006567936856299639,
-0.08067511767148972,0.1711040735244751,-0.0658523291349411,0.1397089660167694,
-0.1619211584329605,-0.02747095562517643,0.10349776595830917,0.020320793613791466]
```

```
fire, gunfire, hit, wound
```

Figure 14: Sample output for the code example shown in Figure 13

We utilized the Word2Vec feature selection technique to generate the features for our words. We used the Word2Vec class in the Spark MLLib framework for our implementation. The development details are in Section 9.8 and the experimental results for classification using the Word2Vec model are in Section 6.3.2.

## 5.6 Association Rules classifier

Association rules is a technique for data mining that provides relationships between elements through the form of rules. The rules take the form of implications with a certain confidence value. For use in classification, if the text contains a collection of terms, it will belong in the rules respective class. If there is no rule that will classify the record, the record will be classified using similarity measures such as cosine similarity to determine the class it is closest to in vector space. Association rules has been shown to be effective at matching publication venue title variations to their actual titles. [8]

An example rule would be:

```
{<KY>, <Bluegrass State>} --> Kentucky
```

where the presence of the token KY, or the phrase “Bluegrass State” implies that this document is talking about the state Kentucky.

Association rules are useful when classifying large collections because the prediction operation can be performed efficiently in constant time when a hash table is used to provide a lookup for the rules. One specific parameter of interest for the association rules classifier will be the support threshold that is required to use a rule in prediction. A lower support threshold will allow more predictions to be made by the rules, speeding up the overall prediction time. However, rules with a lower support will not yield as confident implications as ones with high support. The effect of different support threshold on the number of tweets classified with rules and similarity is explored in Section 6.2.

The classifier implementing association rules was provided to us by Dr. Pereira. Instructions on how to use it and all the parameters that can be tweaked can be found in Section 9.4.

## 5.7 Classifier training and prediction

The definition of classification is to determine which class(es) a given object belongs to, given a set of pre-defined classes. As for the problem defined for the project, we have to classify a given tweet to a real-world event as explained in Section 3.

The process of classification involves the following three steps.

1. Feature selection
2. Feature representation
3. Choosing a classification algorithm

Feature selection and feature representation have been mentioned in Section 5.5. In this Section, we focus on choosing a classification algorithm. The most common classifiers that are predominantly used in text classification are:

1. Logistic Regression
2. Support Vector Machines (SVM)
3. Multi-Layer Perceptron (MLP or Neural Networks)
4. Naïve Bayes
5. Association Rules (AR)

The following table compares some of the intricacies associated with the classification algorithms.

*Table 1 Comparison of classification algorithms*

Classifier	Training Time	Prediction Time	Easy to interpret results	Performs well with number of small observations	Model complexity
Naïve Bayes	Fast	Fast	Somewhat	Yes	Linear
Logistic Regression	Fast	Fast	Somewhat	Yes	Linear
SVM	Slow	Fast	No	Yes	Polynomial
MLP	Slow	Fast	No	No	Quadratic
AR	Fast	Fast	Yes	Yes	Linear

We started our first implementation with Logistic Regression and considering that we have a lot of training data at our disposal, the choice of classifier [1] in our case is not significant. After performing some initial experiments with a small set of test data, the accuracy results were quite promising. More details related to this experiment can be found in Section 6.3.2. Dr. Fox also suggested the we focus on the creation of more training data and tuning the classifier further. Thus, we continued with Logistic Regression as our classification algorithm of choice. We use a multi-class classifier for our project.

Figure 15 shows an example on how to use a Word2Vec model to transform a tweet into a feature vector and train a Logistic Regression classifier and save it into HDFS.

```

1 //Initialize the variables
2 val numberOfClasses = 9
3 val trainingDataFile = "data/trainingData.txt"
4 val word2VecModelFilename = "data/word2VecModel.model"
5 val lrClassifierModelFilename = "data/lrClassifierModel.model"
6
7 //Load the training data into an RDD
8 val trainTweetsRDD = sc.textfile(trainingDataFile)
9
10 //Load the word2vecModel object from HDFS
11 val word2vecModel = Word2VecModel.load(sc,word2VecModelFilename)
12
13 //Define a delegate function to perform the transform using the word2vecModel
14 def transformTweetToWordVector(words:Iterable[String]):Iterable[Vector] =
15 words.map(w=>Try(word2vecModel.transform(w)))
16 .filter(_.isSuccess)
17 .map(x=>x.get.sum)
18
19 //Apply the transform to all the tweets in the training data
20 val trainTweetsFeaturesRDD = trainTweetsRDD mapValues transformTweetToWordVector
21
22 //Train a model using the transformed RDD of feature vector
23 val logisticRegressionTrainedModel = new LogisticRegressionWithLBFGS()
24 .setNumClasses(numberOfClasses)
25 .run(trainTweetsRDD )
26
27 //Save the model to HDFS
28 logisticRegressionTrainedModel .save(sc,lrClassifierModelFilename)

```

Figure 15: Logistic Regression training example

Figure 16 shows an example on how to load a logistic regression model from HDFS and generate predictions on data.

```

1 //Initialize the variables
2 val numberOfClasses = 9
3 val testDataFile = "data/testData.txt"
4 val word2VecModelFilename = "data/word2VecModel.model"
5 val lrClassifierModelFilename = "data/lrClassifierModel.model"
6
7 //Load the test data into an RDD
8 val testTweetsRDD = sc.textfile(testDataFile)
9
10 //Load the word2vecModel object from HDFS
11 val word2vecModel = Word2VecModel.load(sc,word2VecModelFilename)
12
13 //Define a delegate function to perform the transform using the word2vecModel
14 def transformTweetToWordVector(words:Iterable[String]):Iterable[Vector] =
15 words.map(w=>Try(word2vecModel.transform(w)))
16 .filter(_.isSuccess)
17 .map(x=>x.get.sum)
18
19 //Apply the transform to all the tweets in the data to turn into feature vectors
20 val testTweetsFeaturesRDD = testTweetsRDD mapValues transformTweetToWordVector
21
22 //Load a model using the transformed RDD of feature vector
23 val logisticRegressionTrainedModel = LogisticRegressionModel.load(sc, lrClassifierModelFilename)
24
25 //Generate predictions using the logisticRegressionModel
26 val logisticRegressionPredictions = logisticRegressionModel.predict(testTweetsFeaturesRDD)
27

```

Figure 16: Logistic Regression prediction example

The exact source code references can be found in the developer's manual in Section 9.9. Also, the experimental results are shown in Section 6.3.2.



## 5.8 Emitting probability in a multi-class scenario

As described in the previous Section, we implemented our solution using a multi-class Logistic Regression classifier. The multi-class implementation for Logistic Regression in the Spark MLlib framework is a one-vs-all wrapper over the binary Logistic Regression classifier. Also, this implementation only predicts the class of a given tweet. It does not emit the raw probability for each of the classes.

For our classification process to be more effective in classification efficacy and also from the perspective of error analysis, it is imperative that we have a mechanism to generate the probabilities of each of the classes. Having the probability values for each of the class would help us analyze the classification results in further detail and also to understand what is the relative distribution of probabilities across the classes for a given tweet since there is a high possibility that a tweet can belong to multiple classes.

We want to have a system that lets us control how stringent we are about assigning one class to a tweet. If we have the probability of the all the classes, we can set a probability threshold based on the precision we want to have with our classes and balance it with the coverage we want to have for our tweet collections. Having this threshold based mechanism would also allow us to classify a tweet in an “unknown” class, since we are not sure what class it belongs to.

Setting a threshold to a high value would result in high precision but low recall, since a lot of classes with low probability will be rather classified as “unknown”. Setting a threshold to a low value will get a very good recall or coverage on our collection, but will suffer in precision since a class can belong to multiple categories. If there are multiple classes with the same probability, the choice of class in that case will be the first one with the highest probability.

Since the Spark MLlib does not come with an implementation for generating the probability for a multi-class Logistic Regression, we chose to have a custom implementation by building over a sample given in the example [18] on the web. We have abstracted the implementation in the class `ClassificationUtility` for easy usage in our main codebase.

Figure 17 shows the example to extend the Logistic Regression method call to start emitting probabilities for a tweet. Especially important to note are the line numbers 26 and 28, where the former is the way we were using it in the past and the latter is the new call that we make to the newly implemented `ClassificationUtility` class that emits the probabilities along with the predictions.

```

1 //Initialize the variables
2 val numberOfClasses = 9
3 val testDataFile = "data/testData.txt"
4 val word2VecModelFilename = "data/word2VecModel.model"
5 val lrClassifierModelFilename = "data/lrClassifierModel.model"
6
7 //Load the test data into an RDD
8 val testTweetsRDD = sc.textfile(testDataFile)
9
10 //Load the word2vecModel object from HDFS
11 val word2VecModel = Word2VecModel.load(sc,word2VecModelFilename)
12
13 //Define a delegate function to perform the transform using the word2vecModel
14 def transformTweetToWordVector(words:Iterable[String]):Iterable[Vector] =
15 words.map(w=>Try(word2VecModel.transform(w)))
16 .filter(_.isSuccess)
17 .map(x=>x.get.sum)
18
19 //Apply the transform to all the tweets in the data to turn into feature vectors
20 val testTweetsFeaturesRDD = testTweetsRDD mapValues transformTweetToWordVector
21
22 //Load a model using the transformed RDD of feature vector
23 val logisticRegressionTrainedModel = LogisticRegressionModel.load(sc, lrClassifierModelFilename)
24
25 //Generate predictions using the logisticRegressionModel - OLD WAY
26 val logisticRegressionPredictions = logisticRegressionModel.predict(testTweetsFeaturesRDD)
27 //Generate the predictions along with the probabilities - NEW WAY
28 val (prediction,probabilities) = ClassificationUtility
29 .predictPoint(testTweetsFeaturesRDD, logisticRegressionModel)

```

Figure 17: Spark example to generate probabilities for the tweets along with the predictions

Figure 18 shows the probabilities in the expanded object view for the 9 classes for a given tweet that was classified during our experiments. Note that 5 classes among the nine have the same probability. The empirical analysis and our interpretation for the scenarios is discussed in Section 6.3.4. The first two values in the screenshot are the true and predicted labels for the tweet, and the last entry is the redacted tweet text.

```

▶ _1 = {Double@10524} "8.0"
▶ _2 = {Double@10525} "8.0"
▼ _3 = {double[10]@10526}
  0 = 0.0
  1 = 0.2
  2 = 0.2
  3 = 0.2
  4 = 0.2
  5 = 8.465449716413542E-42
  6 = 4.1838155536391876E-39
  7 = 6.43411559229462E-83
  8 = 0.2
  9 = 1.5974681929602393E-121
▶ _4 = {String@10527} "obama issues order prevent next west fer

```

Figure 18: Emission of probabilities along with the prediction for a sample tweet

## 5.9 Spark partitioning and caching

Spark is a general-purpose cluster computing system that empowers other higher-level components to leverage its core engine. While it allows building other higher-level applications on top of it, it has a few components that are tightly integrated with its core engine to take advantage of the future enhancements at the core. Spark is built on top of the Hadoop MapReduce framework to provide an extension to it based on its basic primitive, the Resilient Distributed Dataset (RDD).

The main idea behind RDDs is that they are immutable collections of statically typed objects spread across a Hadoop cluster. The partitioning of the RDDs and storage is designed to be user controlled. The Spark SDKs have extended the programming language to support RDD operations (map, filter etc.) that have the capability to be executed lazily depending on user implementation.

The RDDs are designed to be automatically rebuilt from failure and a lineage of a failed RDD operations can be detected automatically and be assigned for computation to another node implicitly.

The power of Spark comes from the ability to partition data across the cluster and perform computation on a piece of data in a parallel fashion. The Spark SDK exposes operations on the RDD primitive to enable the partitioning to be user controlled. In addition to the partitioning, it also allows an ability to checkpoint an RDD using the cache operation so that repeated processing does not occur by the code using that RDD later.

For the tweet collections that we have to classify, we have to read and classify tweets ranging in number from a few thousand to tens of millions. It is important for us to ensure fast execution for the classification pipeline and have our implementation optimized to run in a distributed fashion across the cluster.

We read data from HBase and partition it accordingly based on our cluster topology. Once the data is distributed across the cluster via partitioning, we perform the classification related processing on the partitioned data.

An example of the partitioning and caching is shown in Figure 19 and Figure 20 respectively.

```
1
2 //Read logs from a log file
3 val logs = sc.textFile("path/to/log-files")
4
5 //Filter the errors and warnings
6 val errorsAndWarnings = logs filter {l => l.contains("ERROR") || l.contains("WARN")}
7
8 //Repartition the errorsAndWarnings across the cluster
9 errorsAndWarnings.repartition(12)
10
11 //Caching ensures that the following two lines don't invoke the read from the log file twice
12 val errorLogs = errorsAndWarnings filter {l => l.contains("ERROR")}
13
14 //This code is executed in parallel across the cluster on partitioned data
15 //Do something with the error logs
16 val errorCount = errorLogs.count
17
18
```

Figure 19: Spark partitioning example

```

1
2 //Read logs from a log file
3 val logs = sc.textFile("path/to/log-files")
4
5 //Filter the errors and warnings
6 val errorsAndWarnings = logs filter {l => l.contains("ERROR") || l.contains("WARN")}
7
8 // Cache the data so that it is not read from again.
9 errorsAndWarnings.cache()
10
11 //Caching ensures that the following two lines don't invoke the read from the log file twice
12 val errorLogs = errorsAndWarnings filter {l => l.contains("ERROR")}
13 val warningLogs = errorsAndWarnings filter {l => l.contains("WARN")}
14
15 //These two lines are the ones that invoke the collect on the RDD.
16 //Till here the code is just lazily evaluated
17 val errorCount = errorLogs.count
18 val warningCount = warningLogs.count
19

```

Figure 20: Spark caching example

Figure 21 shows an example of Spark code running across a cluster using partitioning in Spark UI. The blue bars in jobs panel show parallel execution.

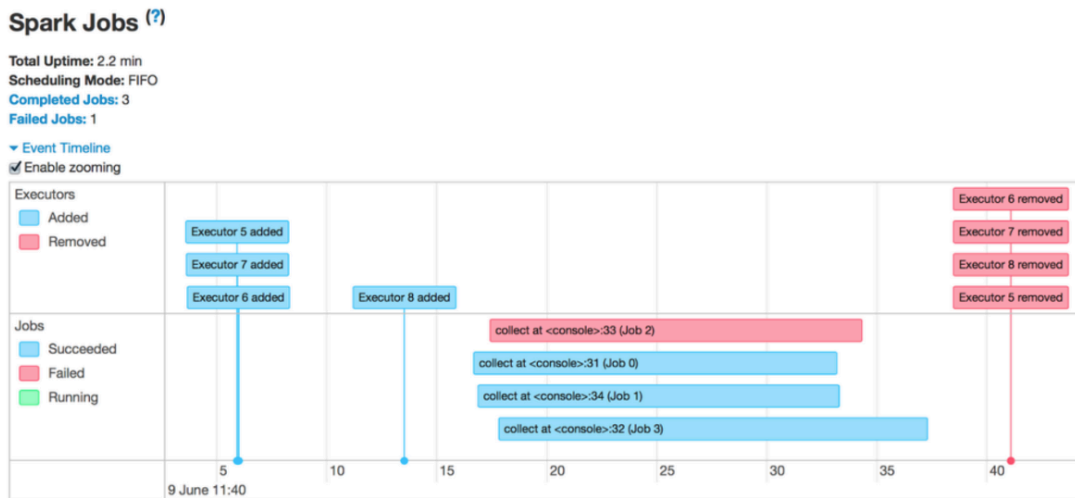


Figure 21: Spark jobs being executed in a parallel fashion [19]

We utilized both repartitioning and caching operations for processing various datasets through our codebase. The development manual Section 9.11 has the exact details for our project and the experiments Section 6.4.2 shows the speed gains from the repartitioning and caching based optimization.

### 5.10 System extensibility

An extensible system allows additional features to be added without changing the entire system. To have extensibility a system should read from configuration files, get passed in parameters and allow data to be changed.

For our project, we must allow the classifier to be extended to predict on more classes than are currently trained. This is because there are many real-world events that are present in the

collections such as the Egyptian Revolution that the classifier is not currently trained on. For the classifier to be effective, it will need to be able to classify tweets into all relevant real world events. This includes future events that are not currently in the collection.

To support additional classes being added we created a table that will store the training data. This table is called “cs5604-f16-cla-training”. The schema for the table is shown in Table 2.

*Table 2 Training data schema*

Column Name	Column Description	Column Example
training-tweet:label	The numerical label of the tweet	1.0
training-tweet:text	Clean text of the tweet to be used for training	report die china factory explosion

The classification label corresponds to a real-world event that training data was generated for. The mapping from labels to real world events can be found in Table 3. The labels are doubles to be compatible with the Logistic Regression API. If any additional data is added to the training table, the models can be retrained by passing the ‘-retrain’ flag. This will instruct the classifier to first reconstruct the Word2Vec and Logistic Regression models from the training data table and save them onto HDFS for use in prediction.

*Table 3 Column mapping between training table label and real world event*

training-tweet:label	real-world-event
1.0	ChinaFactoryExplosion
2.0	KentuckyAccidentalChildShooting
3.0	ManhattanBuildingExplosion
4.0	NewtownSchoolShooting
5.0	HurricaneSandy
6.0	HurricaneArthur
7.0	HurricaneIsaac
8.0	TexasFertilizerExplosion
9.0	NewYorkFirefighterShooting
10.0	QuebecTrainDerailment
11.0	FairdaleTornado
12.0	OklahomaTornado
13.0	MississippiTornado
14.0	AlabamaTornado

### 5.11 Scheduled execution of the classification pipeline with a cron job

A cron job is a scheduled task that is executed by the system at a specified date/time. It is a Linux utility that allows tasks to be automatically run in the background at regular intervals by the cron daemon. These tasks are often termed as cron jobs in Linux. Crontab is the file which contains the schedule of cron entries to be run at specified times.

For our project, it is a given that at every moment new tweets will be added to the HBase table and we should have a mechanism to perform classification on the new tweets that are added. Adding a cron job to perform classification periodically on newly detected tweets would ensure that the HBase table is updated with the classification labels for the new tweets. The cron job can also notify via email once the run is complete.

The cron daemon schedules all the jobs to be run as defined in in the crontab file. The following is an example of an entry in the crontab file. It is supposed to delete all the files in the temp folder of someuser at 18:30 every day.

```
30 18 * * * rm /home/someuser/tmp/*
```

More details about the configuration and scheduling of the cron job for the classification can be found in Section 8.6.

## 6 Experiments

### 6.1 Cleaning Experiment

The purpose of this experiment was to determine if cleaning the training and test data would result in better accuracy for the classifiers.

#### 6.1.1 Experimental setup

To conduct this experiment, we divided the hand labeled data into a training and test dataset. The split used was 70% for the training and 30% for the test data. The same split data was run on both classifiers with the raw text and cleaned text.

The dataset that these experiments were run on can be found in the `experiment_data_(un)lem.txt` files in our Github repository. This data is drawn from 3 broad categories, with 3 sub-categories for each broad category:

1. Shootings
  - a. Kentucky Shooting
  - b. Newton Shooting
  - c. Firefighter Shooting
2. Hurricanes
  - a. Hurricane Arthur
  - b. Hurricane Sandy
  - c. Hurricane Isaac
3. Explosions
  - a. China Factory Explosion
  - b. Texas Plant Explosion
  - c. Manhattan Explosion

The distribution of the specific classes for the sample data is shown in Figure 22. This data shows the class imbalance in the sample set. Such an imbalance was intentionally created to reduce the

bias of the classifier. The matching of the real world events to class labels can be found in Table 4.

Table 4 Real World Events

Class Label	Real World Event
0	Firefighter Shooting
1	China Factory Explosion
2	Kentucky Shooting
3	Manhattan Explosion
4	Newton Shooting
5	Hurricane Sandy
6	Hurricane Arthur
7	Hurricane Isaac
8	Texas Plant Explosion

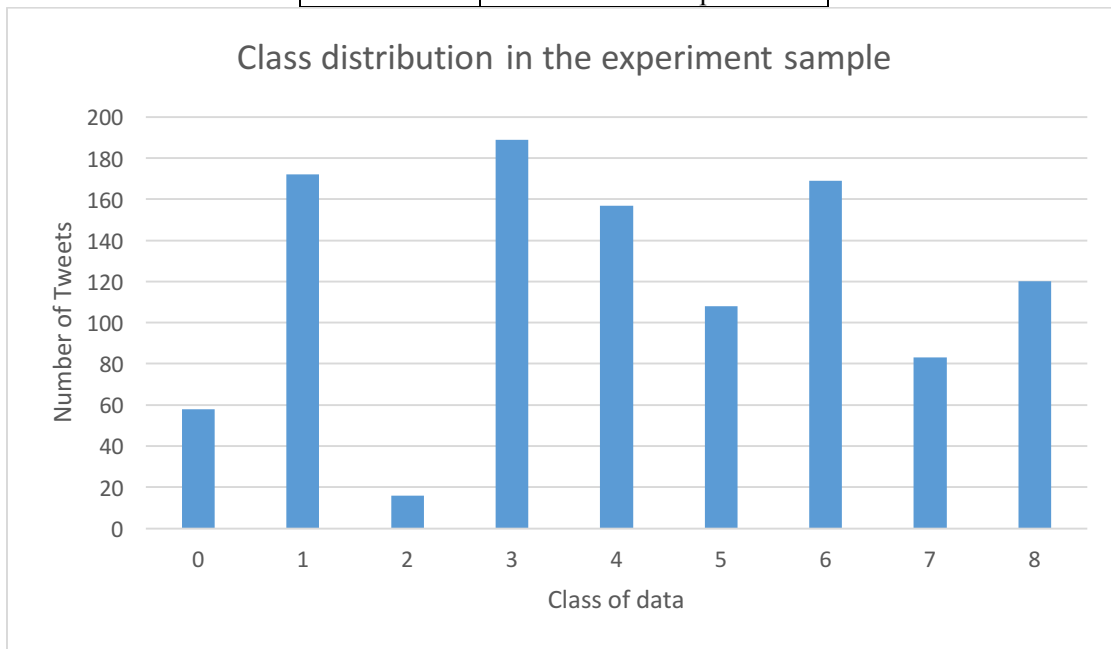


Figure 22: Class distribution in the experiment sample

The cleaning that was done for this experiment was:

- Lemmatization
- Stop word removal
- Hashtag removal
- Non-English character removal

The motivation behind cleaning in this way is discussed in Section 5.4.

### 6.1.2 Experimental results

We can see the summary of the results in Table 5. Cleaning of the data yielded a large reduction in the misclassifications for both the Association Rules classifier as well as the Logistic Regression

classifier. To be able to best classify the tweets in the collections to specific real world events we will employ pre-processing both on the training tweets as well as the tweets whose class we are predicting.<sup>4</sup>

Table 5 Cleaning experiment results

Classifier	% reduction in misclassifications
Word2Vec with Logistic Regression	28.63
Association Rules	50.84

## 6.2 Association Rules support threshold experiment

This experiment will look at the support threshold and see the number of tweets that are predicted by rules and distance for a given support threshold. By looking at this we will be able to see the support that the rules in the Twitter domain need, to classify most by the association rules. The reason for wanting to classify by the rules instead of cosine similarity is that the rules can be applied much faster.

### 6.2.1 Experimental setup

This experiment was performed with a limited number of hand-labeled training data from 3 different classes of shooting events. The data was pulled from the ‘ideal-tweet’ table and represents the following real world events:

- New York Firefighter shooting
- Connecticut School shooting
- Kentucky Accidental Child Shooting

The data can be found in the folder “shooting\_data” on the project’s Github repository.

For this experiment the Association Rules classifier was run on the same dataset with different values for the support threshold. The support threshold parameter dictated the association rules that will be used in prediction. Only rules that have support greater than the threshold will be used to predict tweets.

### Experimental results

Table 6 shows the percentage of the test set that was classified by association rules and distance respectively. We can see that when there is no threshold on the rule values 83.33% of the tweets are predicted by the association rules. As expected when the support threshold is increased to 0.05 the percentage classified by rules decreases to 33.33% because many of the rules that had very small support were excluded. We found that the rules created had low support because the tweets themselves did not have a significant number of common terms even after cleaning. This means that to be able to predict by association rules we will have to use a support threshold close to 0.

---

<sup>4</sup> A comparison of the accuracy of these classifiers can be found in Section 6.3



Table 6 Number of tweets classified by support thresholds

Support threshold	Percent predicted by rules	Percent predicted by distance
0.0	83.33	16.67
0.05	33.33	66.67
0.1	15.56	84.44
0.15	14.44	85.56
0.2	14.44	85.56

### 6.3 Word2Vec based Logistic Regression classifier

The purpose of this experiment was to determine which classifier performs better from an accuracy perspective out of the Word2Vec based Logistic Regression classifier and the Association Rules classifier on our hand-labeled set.

#### 6.3.1 Experimental setup

This experiment used the same hand-labeled data procured for the Cleaning experiment<sup>5</sup>. This data consisted of 9 classes falling into 3 broad categories.

To fairly judge the performance of these classifiers we split up the hand-labeled data into 10 different 70% train - 30% test sets. This was accomplished by first splitting the data into 10 equal sets, then placing 7 of those into training and 3 into test. The remaining splits were generated by rotating the sets between training and test so that each set was in both train and test for at least one of the splits. This procedure is explained pictorially in Figure 23.

Each of the splits was then run on each classifier to compare their results on the exact same training and test data.

---

<sup>5</sup> The specific breakdown and name of classes can be found in Section 6.1.1.

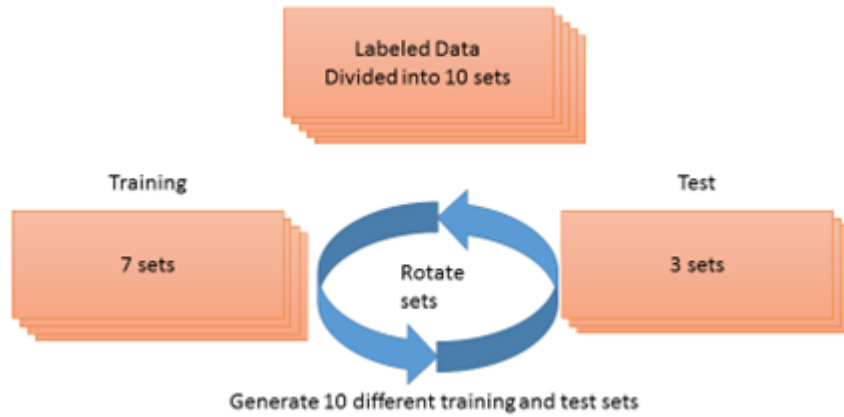


Figure 23: Accuracy experiment data generation

### 6.3.2 Experimental results

Each classifier was run on all 10 experimental splits and the metrics Weighted F-Measure, Weighted Precision, and Weighted Recall were recorded. The results from each classifier on a specific set as well as the averages are shown in Table 7. The averages calculated are shown in Figure 24. From these results, we can see that for all the metrics tested, the averages were higher for the Word2Vec with Logistic Regression classifier on the hand-labeled dataset used.

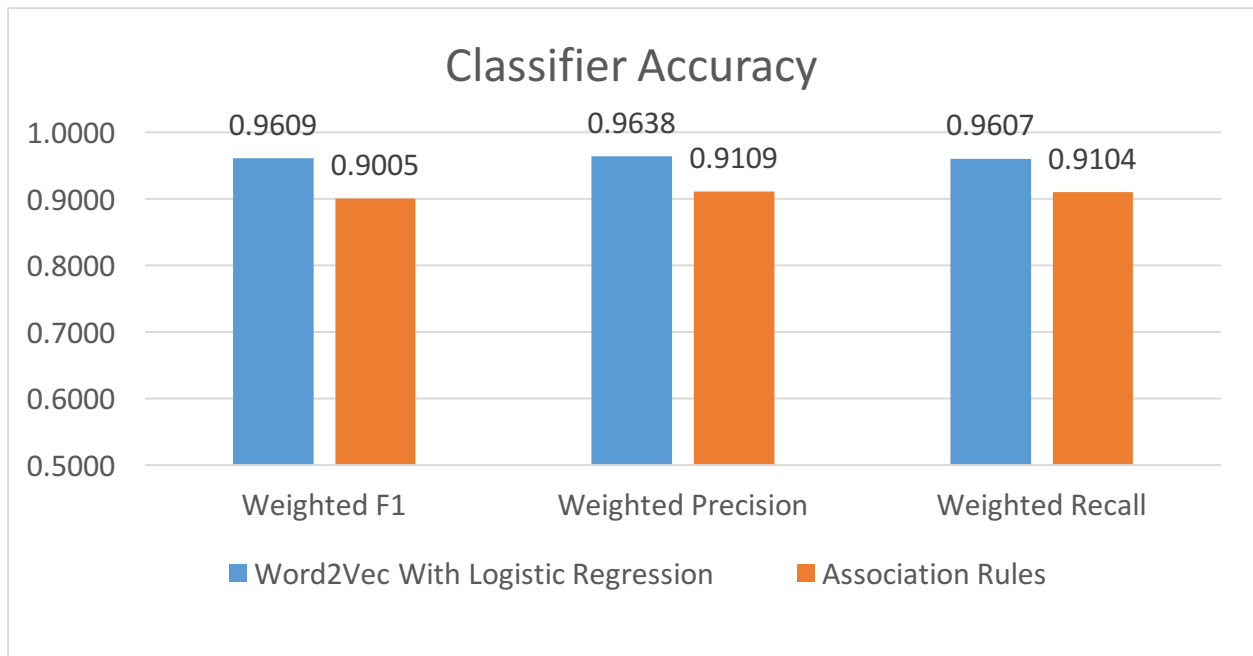


Figure 24: Average classifier accuracy results

Table 7 Comparative experiment results

Data Set	Word2Vec with Logistic Regression			Association Rules		
	Weighted F1	Weighted Precision	Weighted Recall	Weighted F1	Weighted Precision	Weighted Recall
10	0.9720	0.9730	0.9720	0.9555	0.9584	0.9562
9	0.9636	0.9658	0.9626	0.9119	0.9167	0.9221
8	0.9421	0.9521	0.9439	0.8932	0.9039	0.9031
7	0.9688	0.9696	0.9688	0.9658	0.9673	0.9657
6	0.9635	0.9673	0.9626	0.8796	0.8967	0.8938
5	0.9593	0.9600	0.9595	0.9021	0.9085	0.9128
4	0.9755	0.9776	0.9751	0.8467	0.8775	0.8656
3	0.9506	0.9554	0.9502	0.8834	0.8819	0.8972
2	0.9549	0.9574	0.9533	0.9082	0.9143	0.9159
1	0.9590	0.9595	0.9595	0.8583	0.8837	0.8719
Average	<b>0.9609</b>	<b>0.9638</b>	<b>0.9607</b>	<b>0.9005</b>	<b>0.9109</b>	<b>0.9104</b>

### 6.3.3 Test of significance

We performed a t-test to determine if the results from the accuracy experiment in Table 7 were statistically significant. The t-test is used to determine if two sets of data are different. The null hypothesis for this test is that the average weighted F1 scores are equal. The result of a t-test is a p-value is the probability of obtaining a difference at least as large as what was observed assuming the null hypothesis. A higher p-value indicates that the null hypothesis is likely to be true, while a lower p-value indicates that the null hypothesis is likely to be rejected. A normal p-value that will determine if the null hypothesis should be rejected is less than 0.01.

We performed a 2-tailed t-test and found that the resulting p-value was 0.000600527. Given the results of this test we will reject the null hypothesis and conclude that the difference between the average weighted F1 scores are statistically significant.

### 6.3.4 Probability Experiment

The purpose of this experiment is to see the distribution of probabilities for the Word2Vec based Logistic Regression classifier predictions on the test set. This will let us see how confident the classifier is on the predictions it makes.

### 6.3.5 Experimental setup

For this experiment, we used one of the sets from the same training test data split from the cleaning experiment in Section 6.1.1. The data used was the cleaned data as it had produced more accurate predictions for the Logistic Regression classifier.

### 6.3.6 Experimental results

The distribution of the probabilities of the test set can be found in Figure 25. One use of these probabilities is to set a threshold where tweets with probabilities below that threshold will not be classified into their predicted class. The first 3 bars in the figure represent the number of samples that were classified as belonging to 4, 3 and 2 classes respectively. This means that the classifier is not absolutely certain which one class the tweets belong to. It makes a random choice between

the probable classes that have the same probabilities. If we set a threshold to be 0.3, we will be able to classify about 87% of the tweets in the test set, but 64% of them are of the nature where the classifier believes that they can belong to more classes than what has been labelled. The choice of this threshold represents the precision-recall tradeoff. If we set a high threshold such as 0.9 we would get 74 results that the classifier was very confident about, while the other tweets would not get classified. This would lead to high precision but low recall as the tweets we are very confident about would get a classification label, but other tweets are about a specific classification label that the classifier is less confident about would not be classified lowering the recall.

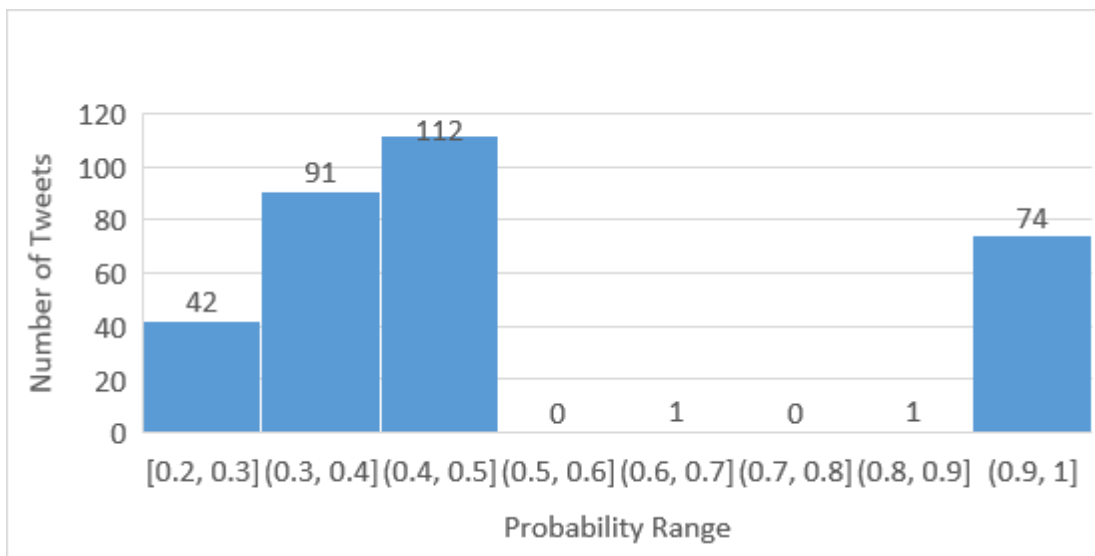


Figure 25: Probability distribution of predicted tweets

We believe that the distribution across the different probability ranges is happening because of two reasons.

- **Limited number of word vectors** – We trained our word vectors using about 750 training tweets. During classification of test data, we omit all the words that the Word2Vec model was not trained on. This results in the culling of words that would have otherwise contributed to the probability for some class. Having a larger set of word vectors would result in more scattered but equal distribution of probabilities across more classes. Or it might also contribute strongly towards the probability of one class. This will result in fewer cases where a tweet has a probability of 0.5 for 2 classes (35% of samples) as shown above. These kinds of classification labels will move into either of the low or high probability ranges.
- **Training on broad categories** – In our experiments, we observed that the probabilities indicated a tweet was more likely to belong to multiple classes than belong to a specific class. For example, a tweet that just had the presence of the word “hurricane” was classified with a probability of 0.33 across all the three specific hurricane classes that we had. This happened because the classifier thinks that each of the 3 classes is equally probable. Training a class specific to a broad category like “hurricane” would ensure that a tweet that is not specific to a particular hurricane gets a probability score close to 1. This would help

classify a tweet that could be of a broad nature into the respective broad category, instead of the probabilities being scattered equally across the specific classes under that broad category.

If a lower threshold is used, the classifier is not sure which label to assign to a given tweet. The label in this case will be any of the equally probable classes at random, but the tweet will be classified. If we select a high threshold, we are very confident about our prediction, but we are not getting a lot of coverage in our classification. Figure 26 shows the distribution of the number of classes our classifier labels the tweets for, based on their probabilities. If we choose a probability threshold of 0.9, then we only get a coverage of 23%. These predictions are very precise, but this precision comes at the cost of recall. If we select a threshold of 0.3, we get a coverage of 87%. The increase in recall comes at the cost of precision. User and product requirements should be the best factor in defining what the ideal threshold should be.

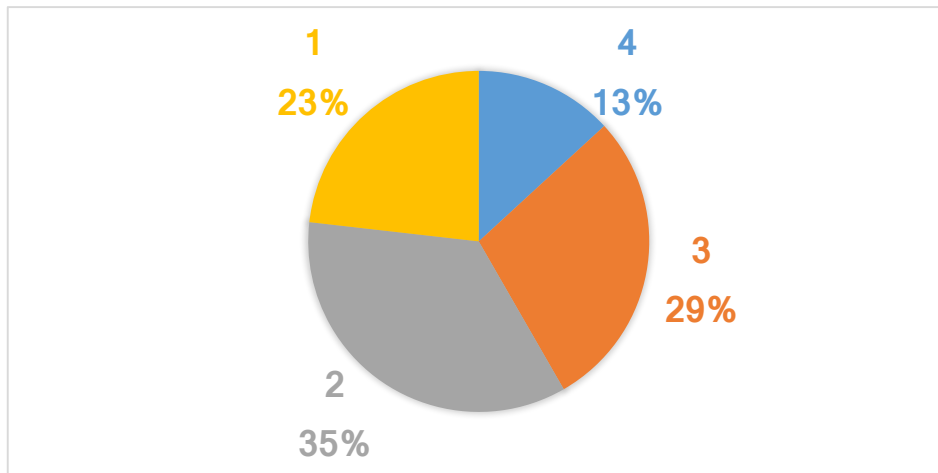


Figure 26: Multi-class assignment distribution

### 6.3.7 Inter-classifier mutual agreement

In this experiment, we measure the level of agreement between the classifiers that we are evaluating. We used the results of the accuracy experiments in Section 6.3.2 to compare the classification results between our logistic regression classifier (LR) and the association rules based classifier (AR). We used the kappa statistic [1] to calculate the inter-classifier agreement for our experiment.

Table 8 Kappa inter-classifier agreement

AR \ LR	Correct	Wrong	total
Correct	299	7	306
Wrong	13	1	14
total	312	8	320
P(A)	0.94		

P(Correct)	0.97		
P(Wrong)	0.03		
P(E)	0.93		
kappa	0.06		

$$kappa = \frac{P(A) - P(E)}{1 - P(E)}$$

Figure 27: Formula to compute kappa

Table 8 shows the distribution of the classification results. The LR and AR classifier agreed on 299 correct classifications and 1 misclassification. They disagreed on 7 and 13 misclassifications for LR and AR respectively. We compute the marginal values P(A) and P(E) and use the formula in Figure 27 to compute the kappa value. Table 9 shows the different ranges of the kappa values and the definition of agreement for the different range of kappa values. For our computed value of 0.06, it signifies that there is only a slight agreement between the two classifiers.

Table 9 Kappa agreement definitions

kappa	Definition of agreement
> 0.8	Good
0.67 – 0.8	Fair
< 0.67	Slight

We believe the reason for slight agreement between the classifiers is because of the fact that the classifiers agreed only once during misclassification compared to 299 correct classifications. This imbalance skews the kappa statistic. A kappa computation on a larger set would definitely give us more data for the kappa statistic to be of significance. We have outlined an experiment in the future work Section that would address this very scenario and help compute a relevant kappa statistic that would make the results more interpretable.

## 6.4 Runtime Comparison Experiment

In addition to the accuracy of the classifiers another property to consider is the time that the classifier takes to predict the classes for large sets of data. This is an important consideration because we have a very large number of tweets to be classified, and it is imperative for us to efficiently classify these into their respective classes within reasonable time.

Being able to classify efficiently means that we can reclassify large parts of our collection as we add additional training data. The classifiers must also be able to classify at a rate faster than the production of tweets so that they can run on future tweets.

### 6.4.1 Experimental setup

For this experiment, we reused the cleaned hand-labeled data described in Section 6.1.1. Instead of splitting the set into training and test we used all of the labeled data as training for the classifiers. We then selected from unlabeled tweets in the IDEAL collection a large set of 640,000. This set

was then shuffled and different sizes of data were used to see the time each classifier spent in the prediction stage. Each classifier was instrumented so that only the prediction time would be tested and the training of the classifiers was done before. Both classifiers classified the same sets of data for each different tweet number tested to ensure one classifier did not get tweets that were more expensive to classify.

### 6.4.2 Experimental results

The initial results can be seen in Figure 28 which shows that for smaller collections of tweets, the Logistic Regression classifier was faster at prediction, while on the larger input sets the Association Rules classifier began to predict much faster. As the collections for this project will have millions of tweets, the performance at large datasets are of top concern.

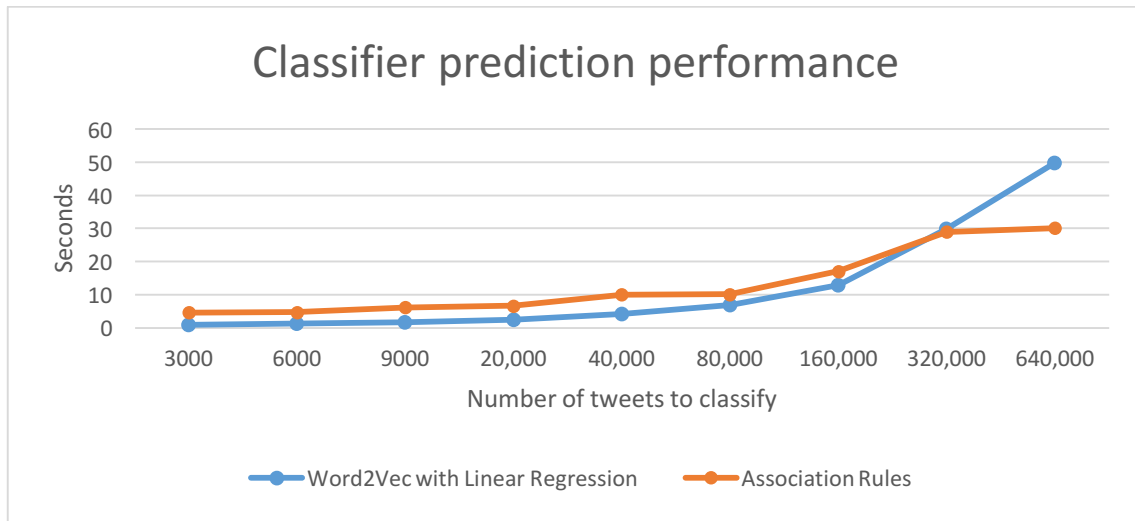


Figure 28: Performance of the classifiers on different number of tweets

To be able to use the Logistic Regression classifier effectively we must reduce the runtime it experiences over large datasets. We optimized the classifier using the partitioning and caching methods mentioned in Section 5.9. The details of the optimization performed can be found in Section 9.11. The results of the experiment when run with the optimized classifier can be seen in Figure 29. Applying the optimization showed an improvement of 57% less time spent in prediction than the original classifier as well as a 14% faster prediction time than the Association Rules classifier at the tweet number of 640,000.

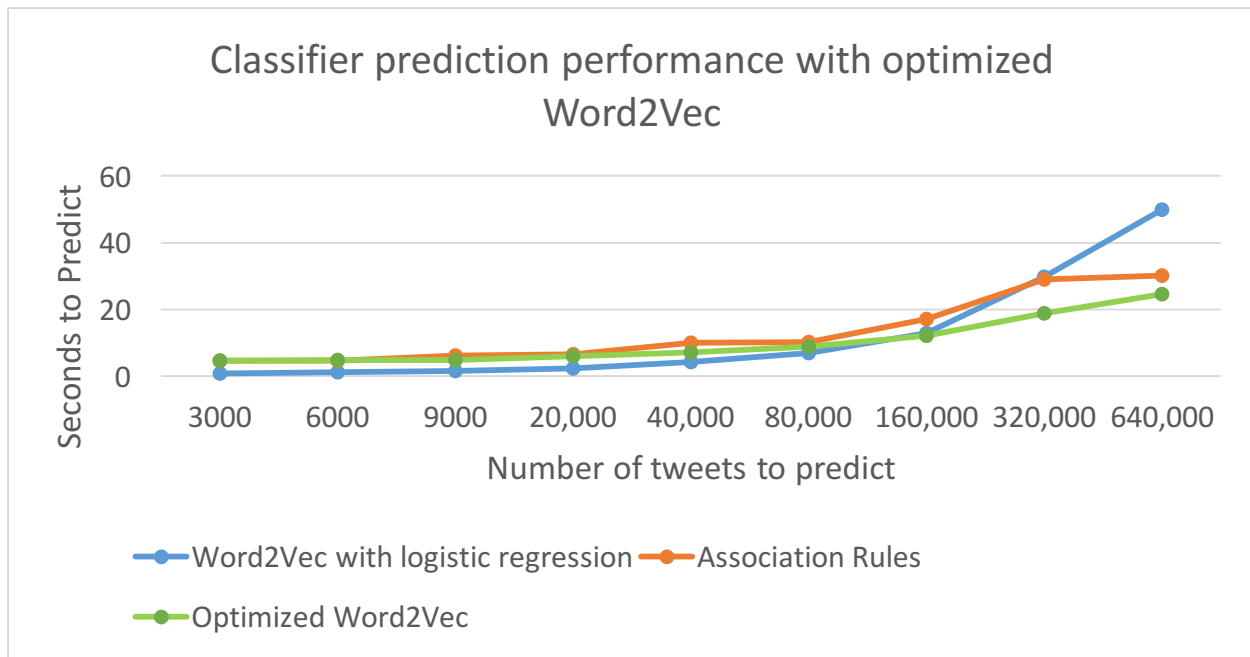


Figure 29: Performance of the classifiers including optimization

## 7 Timeline

This Section contains the timetable that we followed showing the weekly breakdown of the project tasks worked on and completed over the course of this semester.

Table 10 Timetable of tasks

Week start date	Tasks	Performed by
Sep 5 <sup>th</sup>	Reading of previous year's reports, finalizing goals for the project	Saurabh
Sep 12 <sup>th</sup>	Setup the development environment and VM. Getting familiar with the development environment and the cluster. Literature survey on feature selection techniques started.	Saurabh
Sep 19 <sup>th</sup>	Literature survey on feature selection and classification techniques. Start looking at the Association Rules (AR) classifier.	Saurabh
Sep 26 <sup>th</sup>	Perform a run of the AR classifier based on sample data. Literature survey on feature reduction and transformation techniques.	Saurabh
Oct 2 <sup>nd</sup>	Eric joins the team. Knowledge transition process started	Saurabh, Eric
Oct 9 <sup>th</sup>	Gather hand curated training data to start off.	Eric
	Literature survey completed. Start implementation of a Word2Vec based logistic regression classifier.	Saurabh



<b>Oct 16<sup>th</sup></b>	Integrate with the Association Rules based classifier	Eric
	Conclude implementation of the classifier and emit classifier metrics.	Saurabh
<b>Oct 23<sup>rd</sup></b>	Compare Association Rules and Word2Vec with logistic regression classifier based on decided metrics (F1, precision, recall).	Eric
<b>Oct 30<sup>th</sup></b>	Compare the classifiers based on the runtime performance of large collections of tweets.	Eric
	Research ways to improve the run-time performance of the classifier.	Saurabh
<b>Nov 6<sup>th</sup></b>	Optimize Word2Vec with logistic regression classifier to run on the cluster and repeat the runtime performance experiment.	Saurabh, Eric
	Implement the saving and loading of the models for Word2Vec and Logistic Regression.	Saurabh
<b>Nov 13<sup>th</sup></b>	Perform end-to-end integration of the system with the HBase tables we will be reading from and writing to.	Eric
<b>Nov 20<sup>th</sup></b>	Gather additional training data on a variety of classes. Work with other teams on the integration plan.	Eric
	Find the root cause of the HBase read performance issue and research ways to resolve it.	Eric, Saurabh
	Implement the fix and refactor the codebase to work with the new fix	Eric, Saurabh
<b>Nov 27<sup>th</sup></b>	Begin/conclude the integration of the system with other components from the different teams.	Eric, Saurabh
	Research how to emit probabilities for the Logistic Regression classifier and implement it.	Saurabh
	Run the classification on the selected 25 small, medium and large collections.	Eric
	Final project presentation	Eric, Saurabh
<b>Dec 4<sup>th</sup></b>	Final project report	Eric, Saurabh
	Upload project artifacts to VTechWorks	Eric

# 8 User Manual

## 8.1 Environment Setup

The code developed will run on the class' provided cluster. As of right now the cluster is running the following software versions:

- Java version 1.7.0\_101 (<http://www.scala-lang.org/download/2.10.4.html>)
- Scala version 2.10.4 (<http://www.scala-lang.org/download/2.10.4.html>)
- Apache Spark version 1.5.0 (<https://spark.apache.org/releases/spark-release-1-5-0.html>)
- Python version 2.6.6 (<https://www.python.org/download/releases/2.6.6/>)

It is recommended to install these software versions to ensure that the code given will run correctly on your setup.

The code for this project can be found on Github, and should be cloned onto your machine using the “git clone <repoUrl>” command. The Github URL for the team is <https://github.com/saurabhc123/ISRProject>.

## 8.2 Project Layout

This Section will detail the layout of the code supplied with the report submission and the locations on the HDFS that project artifacts have been stored.

### 8.2.1 Uploaded files

This Section will talk about the user files uploaded alongside the report to the VTechWorks site as well as their usage in the overall classification system. Figure 30 shows the file tree of the uploaded files.

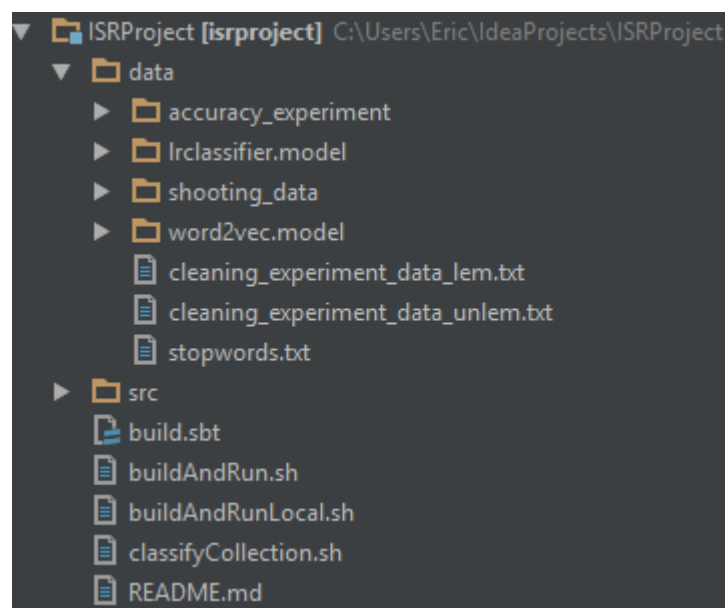


Figure 30: Repository directory tree

The project folder ISRProject contains a data folder which stores experimental data to rerun experiments, a src folder that has all of the code file, and scripts to build the project as well as run the classification code on a certain collection. The details of the src folder will be discussed in Section 9.2. Table 11 gives a brief description of the files in the base directory. The experimental data is found in the data directory. Table 12 gives a brief description of those files in the data directory.

*Table 11 Description of files in base directory*

<b>File name</b>	<b>Description</b>
<b>README.md</b>	This file holds instructions for how to use the provided code as well as how to build the code into a jar file. This serves a function similar to the user manual but at a higher level.
<b>classifyCollection.sh</b>	This is a shell script that will run the classification code on the specific collection specified in the ideal-cs5604f16 table. This script takes 2 parameters: the collection ID to classify, and the batch size for tuning the performance.
<b>buildAndRun.sh</b>	This will build the project from the latest source code and run classifyCollection.sh on a successful build. This will submit the job across the cluster for parallel execution.
<b>buildAndRunLocal.sh</b>	This files serves a similar function to buildAndRun.sh but will not submit the job across the cluster. Instead the code will be run locally. This aids in fast debugging.
<b>build.sbt</b>	This file is used in the build process to specify what packages need to be downloaded automatically to use in the project. This should only be changed when updating the Spark version or adding new external libraries.

Table 12 Description of files in the data directory

File name	Description
<b>stopwords.txt</b>	This is the stop words file that is used for cleaning the data. Any additional stop words can be added to this file.
<b>cleaning_experiment_data_unlem.txt</b>	The raw tweets used in the cleaning experiment.
<b>cleaning_experiment_data_lem.txt</b>	The cleaned tweets that were used in the cleaning experiment.
<b>word2vec.model</b>	The Word2Vec trained model used for the collection classification.
<b>shooting_data</b>	The training and test files specific for shooting data to more quickly see everything is working as intended before it is run on the larger sets.
<b>lrclassifier.model</b>	The Logistic Regression trained model used for the collection classification.
<b>accuracy_experiment</b>	This folder holds all of the training test splits used in the accuracy experiment so that the results can be reproduced.

When the scripts are run in local mode, they will be read from the local file system, but when they are run on the cluster they are read from HDFS. To ensure the code operates the same, the modified files should be copied to HDFS. This can be done by first removing the old file, then copying the new file as seen in Figure 31.

```
#remove the file from HDFS
hadoop fs -rm <FILE TO REMOVE>

#add the file HDFS
hadoop fs -put <FILE TO ADD>
```

Figure 31: HDFS commands

### 8.2.2 HDFS files

This Section will document the various HDFS artifacts. Unless otherwise noted these artifacts are stored in the base HDFS directory of /usr/cs5604f16\_cla.

Similar to the local directory the models are stored on HDFS in that data folder. The folders the models are stored in are:

- word2vec.model – the previously trained Word2Vec model
- lrclassifier.model – the previously training Logistic Regression model

Because the files are too big to store on Github the data used for the performance experiment is stored in the data/mega\_sets folder with a suffix describing the number of tweets in each file.

### 8.3 Generating training data for files

#### 8.3.1 Github training data

Training data is provided in the Github repository in the folder data/shooting\_data for small proof of concept tests. Training data to perform accuracy comparisons is provided in the data/accuracy\_experiment folder discussed earlier. The data provided is divided into 2 files, the training data and the test data. These are divided using a 70-30 split where 70% is in the training data and 30% is in the test data. The provided files have a list of classified tweets with the following structure.

“class the tweet belongs to”; “unique ID of the tweet”; text that makes up the tweet.

The class numbers correspond to a specific real world event label. This mapping is provided in Table 13. This mapping can be changed in the code in the DataWriter.scala class file. The details for how to do this will be discussed in Section 9.7.

Table 13 Mapping from label to event

training-tweet:label	real-world-event
1.0	ChinaFactoryExplosion
2.0	KentuckyAccidentalChildShooting
3.0	ManhattanBuildingExplosion
4.0	NewtownSchoolShooting
5.0	HurricaneSandy
6.0	HurricaneArthur
7.0	HurricaneIsaac
8.0	TexasFertilizerExplosion
9.0	NewYorkFirefighterShooting
10.0	QuebecTrainDerailment
11.0	FairdaleTornado
12.0	OklahomaTornado
13.0	MississippiTornado
14.0	AlabamaTornado

A user can add additional classes by adding more tweets in the same format and placing a unique class number for each of their additional classes. This number should be 1 greater than the maximum class label. For example, the first label number to be added would be 15.0. The labels are real numbers to be compatible with the Logistic Regression Classifier API.

#### 8.3.2 HBase training data

In addition to the data provided on the Github repository, we created a HBase table to store the training data called “cs5604-f16-cla-training”. Table 14 stores the training data schema used.

Table 14 HBase training data schema

Column Name	Column Description	Column Example
training-tweet:label	The numerical label of the tweet	1.0
training-tweet:text	Clean text of the tweet to be used for training	report die china factory explosion

The labels used in the table correspond to the mappings in Table 13. Additional training data can be added by following the same schema. This data can then be used to retrain the models by passing the ‘—retrain’ which will reread the training data from the table and save the models back into their respective places on HDFS.

### 8.4 Running the Association Rules classifier

To run the association rules classifier on the training data you will have to run the jar file with Spark submit. An example command for running the classifier is shown in Figure 32. This will output the association rules results into the base directory on HDFS into a folder called tokensClassRules. The predictions from the classifier will be in a directory called outputPredictedDir.

An example command to run the AR classifier is shown in Figure 32. The jar file for the classifier is found on HDFS and called assocer-spark-1.2.jar. This file should be pulled down from HDFS to run the classifier.

```
spark-submit --master yarn-client --deploy-mode client --class main.java.com.mestrado.main.MainSpark \
  assocer-spark-1.2.jar 0.3 8 printers nameservice1 node1.dlrl /user/cs5604f16_cla/ \
  input/train_data input/test_data stopwords.ser true output
```

Figure 32: Running the AR classifier

The main parameters to look for are “input/train\_data” and “input/test\_data” which specify the training and test data files, respectively. These can be changed to run the classifier on different datasets. Note that these datasets are being read from HDFS so any files should be pushed to HDFS before being run.

```

|*****|
DATE=07/12/2016 18:29:39
-----
RULES QTY:
(k=1): 6
(k=2): 1
(k=3): 0
-----
PREDICTED QTY:
(k=1): 10
(k=2): 4
(k=3): 0
-----
PRED SIMI QTY:
106

TRAIN=25.966
TEST=3.198
SIMI=1.714
Evaluation time: 0.043

Total time: 30878ms 30.878s 0.51463333333333334m
TIME=30.878
|*****|

```

Figure 33: Results of running the Association Rules classifier

Once the command has been executed, statistics will be sent both to the screen and to local directories. The data shown on the screen will include how the test data was classified either through the rules or by similarity. An example of that is found in Figure 33.

This shows the number of rules generated for each specific class, the quantity of data that was predicted by rules, and the time that the total evaluation took.

In addition, the metrics such as precision and recall were placed into the output folder in the directory the code is run in. The content of this output file is seen in Figure 34. In this example the output folder was called “output”. The output folder will contain 2 subdirectories as shown in Figure 34.

```

[cs5604f16_c1a@node1 AssocER-Spark]$ ls -l output/
total 8
drwxrwxr-x 2 cs5604f16_c1a cs5604f16_c1a 4096 Nov  1 15:16 evaluation
drwxrwxr-x 2 cs5604f16_c1a cs5604f16_c1a 4096 Nov  1 15:16 times

```

Figure 34: Output directory files

The evaluation directory will contain specific metrics related to the test set such as precision and recall. Note that there are different metrics than are computer by the other classifier. Any comparisons should use the same metrics. An example evaluation file can be seen in Figure 35.

```

Equals:          92
Different:       28
Precision:       0.7062984344931607
Recall:         0.8
Macro:          0.7429601279792641
Micro:          0.7666666666666667
Precision      Recall  MacroF1  MicroF1
70.63%  80.00%  74.30%  76.67%

```

Figure 35: Association Rules evaluation output

The times directory will store the values printed out at the end of running the script so they can be used later to see the number of tweets that were predicted by rules as well as the time required to predict the tweets. The file contents are shown in Figure 33.

### 8.5 Running the classifier

To run the classifier first run the buildAndRun.sh script. This will build the project from source and download any required packages. Then copy the Stanford-nlp files downloaded into “~/ivy2” into the directory the code is in. This will allow the jar files to be distributed to each node in the cluster to aid in cleaning.

Once the script has been run there should be a target directory created containing a jar file.

If this is the first time the code is being run, the training models must be created. By running the command “classifyCollection.sh –retrain” it will generate the feature selection and classifier models and save them onto HDFS. These models were trained using the training data table on HBase mentioned in Section 8.3.2.

Now that the models have been trained the classifier can be run on a specific collection. The collections can be seen at <http://hadoop.dlib.vt.edu:82/twitter/>. The Archive ID corresponds to the collection number. To run the code on a specific collection use the command “classifyCollection.sh <collection number> <batch size>”. The specific parameters for the classifyCollection script are listed in Table 15.



Table 15 ClassifyCollection.sh parameters

Parameter	Use Description
<b>collection number</b>	This is the number of the specific collection to run the classifier on.
<b>batch size</b>	The size of the batch that is pulled from HBase before being parallelized across the cluster. A larger value may improve performance but risk memory related errors. The default and recommended size is 5000.
<b>--retrain</b>	This flag will cause the program to retrain the models and save them onto HDFS for use in future runs. The training data comes from the “cs5604-f16-cla-training” table on HBase.
<b>--metric &lt;training file&gt; &lt;test file&gt;</b>	This flag will tell the program to generate metrics for the predictions by training models from the training file and testing the predictions of the test file. This can be used for the accuracy experiments.

The results of running this script are that the “real-world-events” column in HBase has been populated with the classification results for the tweets in the collection specified. This can be seen by using the HUE interface found at <http://hadoop.dlib.vt.edu:8888/hbase/#HBase/ideal-cs5604f16>. This interface can be queried for a prefix of the row keys. The row keys in HBase are based on the collection number. An example of HUE being queried to show tweets in collection 28 is shown in Figure 36.

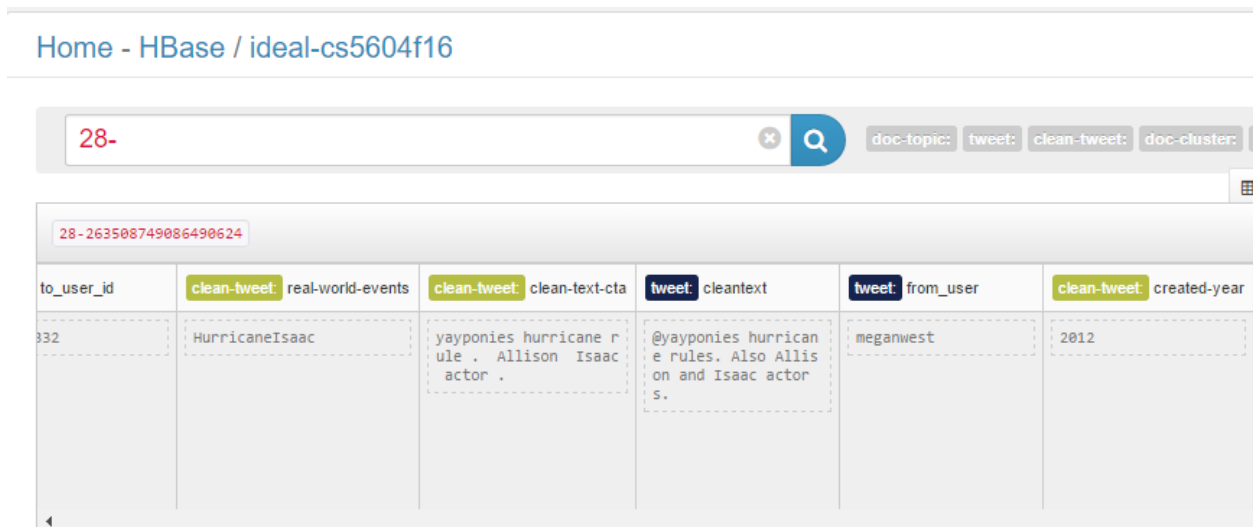


Figure 36: Hue interface example

The timestamp of the “real-world-events” field can be used to verify that the tweet predictions have been written. This can be seen by hovering over the “real-world-events” column. This is demonstrated graphically in Figure 37.

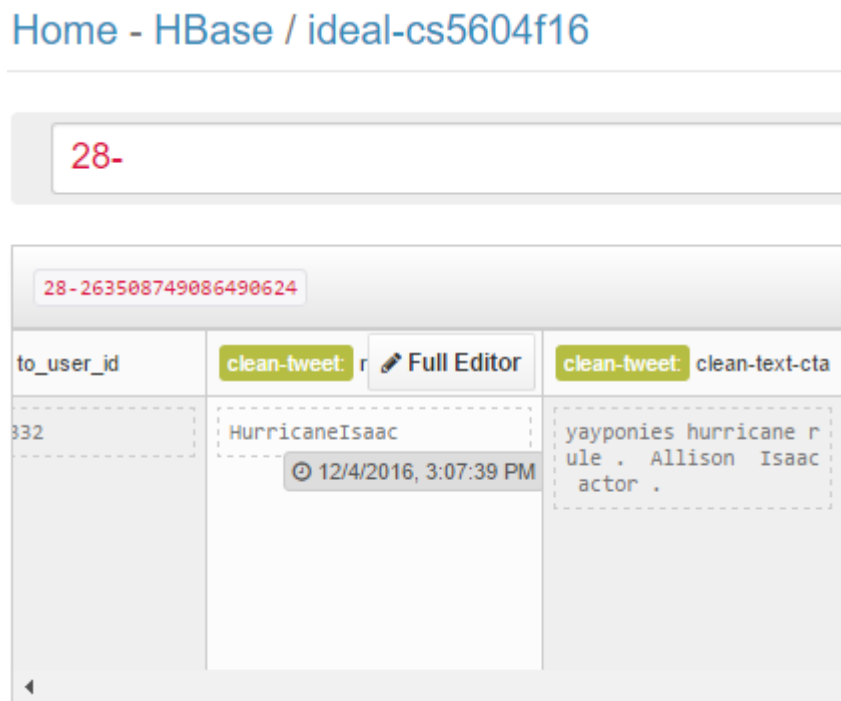


Figure 37: HUE timestamp viewing example

## 8.6 Configuring the cron job

We need to execute the following command to be executed at each run of the cron job.

```
“classifyCollection.sh <collection number>”
```

We need to either specify the collection number or execute the command without the parameter. Doing so would run classification on all the collections which could take a long time.

Create a file name **classifierCronTask.txt** and add the following line to the file with the appropriate collection number. The below format would run the task at 6:30 PM every day.

```
30 18 * * * classifyCollection.sh <collection number>
```

To change the run time and scheduled interval, please refer to Figure 38 and Figure 39 for a detailed list of position placeholders and parameter settings, respectively.

```

* * * * *      command to be executed
- - - - -
| | | | |
| | | | | +----- day of week (0 - 6) (Sunday=0)
| | | | | +----- month (1 - 12)
| | | | | +----- day of month (1 - 31)
| | | | | +----- hour (0 - 23)
+----- min (0 - 59)

```

Figure 38: Parameter placeholder specifications for the cron job [20]

min	hour	day/month	month	day/week	Execution time
30	0	1	1,6,12	*	— 00:30 Hrs on 1st of Jan, June & Dec.
0	20	*	10	1-5	—8.00 PM every weekday (Mon-Fri) only in Oct.
0	0	1,10,15	*	*	— midnight on 1st ,10th & 15th of month
5,10	0	10	*	1	— At 12.05,12.10 every Monday & on 10th of every month

Figure 39: Usage examples for the crontab entry [20]

Create a cron job from the file by executing the following command.

```
crontab /<path to file>/ classifierCronTask.txt
```

Execute the following command to see the list of cron jobs that are registered to run.

```
crontab -i
```

Figure 40 shows the list of jobs that have been configured to run by the cron daemon. The job that we configured to run by the cron daemon should be shown as part of the list.

```

dlr11:~# crontab -l
30 18 * * * classifyCollection.sh >

```

Figure 40: Console output for the list of cron jobs

To remove the cron job, remove the line containing the `< classifyCollection.sh >` script from the `classifierCronTask.txt` file and execute the following command.

`crontab /<path to file>/ classifierCronTask.txt`

## 9 Developer Manual

### 9.1 VTechWorks Inventory

We have uploaded the following to the VTechWorks repository:

- ClassificationFinalReport - This document.
- ClassificationFinalPresentation – The final presentation our team gave. This document highlights what we completed and the experimental result observed.
- ClassificationGithubRepository – The zip file containing a snapshot of the team Github repository. This contains the project source code as well as scripts for building and running the project. The specific files are discussed in the user and developer manuals. The repository is also publically available at <https://github.com/saurabhc123/ISRProject>.

### 9.2 Source Directory

The developer manual discusses portions of our code that can be extended. All of these code files can be found in the src directory. Note that the code can also be easily browsed on Github. Our project URL is <https://github.com/saurabhc123/ISRProject>. A short description of each file is found in Table 16.

Table 16 src directory files

Filename	Description
CleanTweet.scala	Contains functions that will clean the tweet text to provide better classification results.
DataRetriever.scala	Contains functions to retrieve prediction as well as training data from HBase.
DataWriter.scala	Contains functions to write the predicted tweet real-world-events into HBase so other teams can make use of them.
FeatureGenerator.scala	Generates the feature vectors with Word2Vec given tweet text.
SparkGrep.scala	The entry point for the project that calls the other functions.
Word2VecClassifier.scala	Contains functions to process clean and unclean tweets as well as train the models.
HBaseInteraction.scala	Contains functions for interacting with HBase at a lower level.
ClassificationUtilities.scala	Contains functions for generating the probabilities as well as the predictions of tweets.

### 9.3 Training data

We have provided in the `data_scripts` directory on HDFS numerous scripts that will retrieve data from HBase, randomly sample the retrieved datasets, and partition those sets into training and test sets.

The files in the `data_scripts` directory are shown in Table 17.

Table 17 `data_scripts` directory files

File name	Description
<b>Command.sh</b> <prefix>	Run an HBase shell command to get all records that begin with that prefix.
<b>Rand_select.py</b> <input file>	Select and random sample the lines from the input file that can be used to select training and test data to be hand-labeled.
<b>Partition.py</b>	Partition the labeled set into a 70% training 30% test split. This will be used on the hand labeled data to generate separate sets.

These files can be run and modified to generate the training and test sets from a certain collection within HBase.

### 9.4 Parameters for the Association Rules classifier

The jar file with the classifier can be found in the code directory under the name `assocer-spark-1.2.jar`. Unfortunately, the source is unavailable, but there are numerous parameters that can be tweaked when running it from the command line. The contact for the Association Rules classifier is Dr. Pereira ([denilson@vt.edu](mailto:denilson@vt.edu)).

The parameters to run the Association Rule classifier are shown in Figure 41.

```
spark-submit -master yarn-client -deploy-mode client -class main.java.com.mestrado.main.MainSpark\  
  <path_to_jar_file> <support threshold> <block amount> printers nameservice1 <cluster node>\  
  <base_dir_on-HDFS> <path_to_training_file> <path_to_test_file>\  
  <stopwords_file> true <output_directory>
```

Figure 41: Parameterized Association rule classifier command

The parameters to this command are shown in Table 18.

Table 18 Association rules classifier parameters

Parameter name	Description
<b>path_to_jar_file</b>	The directory where the file assocer-spark-1.2.jar is located on the local file system.
<b>support threshold</b>	The support required to use an association rule for classification.
<b>block size</b>	The number of blocks to split the data into to distribute across the cluster.
<b>cluster node</b>	The node that the code is running on (node1.dlrl).
<b>base_dir_on-HDFS</b>	The directory on HDFS that will be used as the base for the training and test file paths. (/user/cs5604f16_cla/)
<b>path_to_training_file</b>	The file that will be used for training the model
<b>path_to_test_file</b>	The file that will be used to test the predictions and output metrics.
<b>stopwords_file</b>	The file on the local file system for stop word removal.
<b>output_directory</b>	The path on the local file system where the program will write additional metrics to on completion.

The example command we ran to generate the classifier on our cluster is shown in Figure 42.

```
spark-submit --master yarn-client --deploy-mode client --class main.java.com.mestrado.main.MainSpark \
    assocer-spark-1.2.jar 0.3 8 printers nameservice1 node1.dlrl /user/cs5604f16_cla/ \
    input/train_data input/test_data stopwords.ser true output
```

Figure 42: Command to run the classifier

Running the command in Figure 42 resulted in the file input/training\_data being read as the training file and input/test\_data read as the test file. The results were written to the output folder.

## 9.5 HBase Reading

The code for reading from HBase can be found in the DataRetriever.scala file. This file provided methods for retrieving all tweets from a specific collection into a Spark RDD as well as retrieving the labeled training data from the table to allow retraining of the models.

### 9.5.1 Reading Prediction Data

The data is read in batches based on the parameter passed in from the result scanner object. This results in a list of result objects returned from HBase. The code to set up the Scan is seen in Figure 43. To only scan over one collection we specified the start and stop prefixes that will only give us row keys that fall within.

Each batch is then processed by taking up to the batch size number of records. These batches then go through the pipeline as seen in Figure 44. This allows our code to work on any arbitrarily large tweet set.

```
// scan over only the collection specified
val scan = new Scan(Bytes.toBytes(collectionID), Bytes.toBytes(collectionID + '0'))

val hbaseConf = HBaseConfiguration.create()
val table = new HTable(hbaseConf, _tableName)

// add the specific column to scan
scan.addColumn(Bytes.toBytes(_columnFamily), Bytes.toBytes(_column))
// add caching to increase speed
scan.setCaching(_cachedRecordCount)

// query HBase using the scan
val resultScanner = table.getScanner(scan)
println(s"Caching Info:${scan.getCaching}")
println("Scanning results now.")
```

Figure 43: Configure scan for HBase reading

```
// read batchSize from the HBase table
val results = resultScanner.next(_batchSize)

// map the result objects to tweet objects for processing
val resultTweets = results.map(r => rowToTweetConverter(r))

//parallelize the tweets across the cluster
val rddT = sc.parallelize(resultTweets)

// cache to allow every cluster node to easily access the data
rddT.cache()

println("***** Cleaning the tweets now. *****")
val cleanTweets = CleanTweet.clean(rddT, sc)

println("***** Predicting the tweets now. *****")
val predictedTweets = Word2VecClassifier.predict(cleanTweets, sc, word2vecModel, logisticRegressionModel)

println("***** Persisting the predictions now. *****")
val repartitionedPredictions = predictedTweets.repartition(12)
DataWriter.writeTweets(repartitionedPredictions)
```

Figure 44: Batched processing code example

The raw HBase result is mapped into a tweet object to work with all of our other code by the `rowToTweetConverter` method that will take a result and extract the tweet text, and row key to allow the prediction to be written back to the correct tweet. The method used for the conversion is shown in Figure 45.

```

// This method will take the HBase result and return the corresponding tweet
def rowToTweetConverter(result : Result): Tweet = {
  // this will retrieve the latest data from the tweet's text column
  val cell = result.getColumnLatestCell(Bytes.toBytes(_columnFamily), Bytes.toBytes(_Column))
  // retrieve the row key to uniquely identify this tweet
  val key = Bytes.toString(cell.getRowArray, cell.getRowOffset, cell.getRowLength)

  // retrieve the tweet text
  val words = Bytes.toString(cell.getValueArray, cell.getValueOffset, cell.getValueLength)

  // construct and return a tweet object with id=key and text=words
  Tweet(key,words)
}

```

Figure 45: Conversion from HBase row to tweet example

## 9.6 Cleaning

As shown in our experiments, cleaning was effective to increase the accuracy of both classifiers. In this Section, we will detail how we implemented cleaning of the tweet text.

The code can be found in the file called CleanTweet.scala.

Our cleaning method takes in an RDD of tweets, this can be tweets read from a file, or from HBase. The result of running the cleaning will be an RDD of tweets that has the text changed to be the cleaned text. Cleaning will not modify the identifier not the label on a tweet. The entry point to the cleaning code is the method clean shown in Figure 46.

```

def clean(tweets: RDD[Tweet], sc : SparkContext): RDD[Tweet] = {
  // get the ids of the tweets to preserve them
  val keys = tweets.map(tweet => tweet.id)
  // get the tweet text to clean
  val values = tweets.map(tweet => tweet.tweetText)

  // clean the tweet text and attach it to the id
  keys.zip(getCleanedTweets(values,sc)).map(va => Tweet(va._1.toString,va._2.toString))
}

```

Figure 46: Cleaning class entry point

The cleaning step performed is # removal seen in Figure 47.

```

// remove all occurrences of the '#' character in the tweet text
val t = text.replaceAll("#", "")

```

Figure 47: '#' character removal

In addition, lemmatization is performed using the Stanford NLP library. This library will tokenize the input, and perform lemmatization. One important note is that the library is large and so we use a foreachPartition to only load the model once per every partition of the data. This provides a large speedup. The code for using the Stanford NLP is found in Figure 48.



```

val doc = new Annotation(tweetText)

// annotate the tweet text with the NLP pipeline
pipeline.annotate(doc)

// where the lemmas will be stored
val lemmas = new ArrayBuffer[String]()

// break the sentence up into tokens and lemmatize each word
for (sentence <- doc.get(classOf[SentencesAnnotation]).iterator()){
  val t = new ListBuffer[CoreLabel]() ;
  for (tw <- scalaIterator(sentence.get(classOf[TokensAnnotation]).iterator())){
    t += tw
  }
  for (token <- t){
    val lemma = token.get(classOf[LemmaAnnotation])
    if (lemma.length > 2 && !stopWords.contains(lemma) && isOnlyLetters(lemma)) {
      lemmas += lemma.toLowerCase // put the lemmas to lower case
    }
  }
}
lemmas

```

Figure 48: Stanford NLP example

Stop word removal is performed by taking each lemma and discarding any that are in the stop word set. This is done efficiently by using a set for the collection of stop words and broadcasting it across the cluster.

## 9.7 Writing to HBase

The predicted tweets must be written back to HBase so that SOLR can index it and provide faceted searches on the classes the tweets are predicted to be. The data has already been partitioned, so each partition can be written to the database in one chunk. By keeping the writing distributed across the cluster it provides a speedup.

The file that contains the writing code is `DataWriter.scala`.

The table the tweets are written to currently is “ideal-cs5604f16”. This value as well as the column family and column can be changed in the code. Note that the table and column family must be created before data can be written to the table. The API we are using does not allow for automatic table or column family creation if it is non-existent.

The main method of interest in this file is the `writeTweets` method. This takes in an RDD of tweet objects, and writes, to rows with those IDs, the values of their mapped real-world-event. The logic is shown in Figure 49.

```

def writeTweets(tweetRDD: RDD[Tweet]): Unit = {
  // definitions of the table, column family, and column
  val _tableName: String = "ideal-cs5604f16"
  val _colFam : String = "clean-tweet"
  val _col : String = "real-world-events"

  // creation of an HBaseInteraction object that allows writing to HBase
  val interactor = new HBaseInteraction(_tableName)

  // place every tweet from the RDD into HBase
  tweetRDD.collect().foreach(tweet =>
    interactor.putValueAt(_colFam, _col, tweet.id, labelMapper(tweet.label.getOrElse(999999.0)))
  )

  // close the interactor to not leak resources
  interactor.close()
}

```

*Figure 49: Writing tweets to the database*

The tweets are written so that the `tweet.id` field is used as the row key for the record. This was done so that if the tweets predicted are read from the same table, then the column will be added to the rows that were read, which immediately pairs the predicted tweets back to the original. If this is not desired the tweet ID can be changed to whatever row key is desired.

As seen in Figure 49, the `DataWriter.scala` class takes advantage of `HBaseInteraction.scala`, which was written by Matthew Bock for writing data to HBase. These files are included in our source directory and can also be found in the Canvas file directory in the `F2016/Code` folder.

We made one modification to the code to allow the interaction object to be closed. This frees up the resources such as the HBase connection after it is no longer needed. We needed this closing functionality because we create a new `HBaseInteraction` object every time we run a new batch. The closing code is given in Figure 50.

```

def close(): Unit= {

  // close the HBase table connection and flush any buffered data
  table.close()

  // close the connection pool that maintains multiple connections to HBase
  connection.close()
}

```

*Figure 50: Disposing of the HBaseInteraction object*

The other component of the writer is the mapping from labels to the corresponding real-world-events. Table 13 shows this mapping. Figure 51 shows how the mapping was implemented in code. New classes should have their mappings defined in this Map structure.

```

def labelMapper(label:Double) : String= {
  // define the mapping from labels to real-world-events
  val map = Map(
    1.0->"ChinaFactoryExplosion",
    2.0->"KentuckyAccidentalChildShooting",
    3.0->"ManhattanBuildingExplosion",
    4.0->"NewtownSchoolShooting",
    5.0->"HurricaneSandy",
    6.0->"HurricaneArthur",
    7.0->"HurricaneIsaac",
    8.0->"TexasFertilizerExplosion",
    9.0 -> "NewYorkFirefighterShooting",
    10.0->"QuebecTrainDerailment",
    11.0->"FairdaleTornado",
    12.0->"OklahomaTornado",
    13.0->"MississippiTornado",
    14.0->"AlabamaTornado"
  )

  // return the mapping of the label
  map.getOrElse(label,"CouldNotClassify")
}

```

Figure 51: Code for label mapping

## 9.8 Word2Vec Generation

As described in the Section 5.5 for implementation of Word2Vec based feature selection and transformation technique, Figure 52 shows the exact way we have implemented it for our project.

```

1 //Initialize
2 val bcWord2VecModelFilename = sc.broadcast("data/word2VecModel.model")
3
4 //Tweets have been loaded in a RDD of tweetId and the array of words in the tweets
5 val trainingTweetsRDD: RDD[(String, Iterable[String])] = samplePairs.mapValues(_.tweetText.split(" ").toIterable)
6
7 //Declaring a word2vecModel
8 var word2vecModel:Word2VecModel = null
9
10 //Partition the training data across partitions
11 trainingTweetsRDD.repartition(partitionCount)
12
13 //Cache the data so that the training data is not read twice from the file upstream
14 trainingTweetsRDD.cache()
15
16 //We try to load any existing model that we migh have saved.
17 try {
18 //Load the model from HDFS if it exists
19 word2vecModel = Word2VecModel.load(sc, bcWord2VecModelFilename.value)
20 println(s"Model file found:${bcWord2VecModelFilename.value}. Loading model.")
21 }
22 catch{
23 case ioe: IOException =>
24 println(s"Model not found at ${bcWord2VecModelFilename.value}. Creating model.")
25 //If the model does not exist in HDFS, create a new model.
26 word2vecModel = new Word2Vec().fit(trainingTweetsRDD.values)
27
28 //Save the model in HDFS so that the training doesn't have to be performed for the next run.
29 word2vecModel.save(sc, bcWord2VecModelFilename.value);
30 println(s"Saved model as ${bcWord2VecModelFilename.value} .")
31 }

```

Figure 52: Code snippet for Word2Vec training

Once the word vectors are trained, we have to transform a given tweet into a feature vector. For that we have to transform a word into its vector representation, and average the sum of the vector that we get back from the transformation operation. We used the default vector size of 100 values for the word vectors.

```

1
2 //Define a delegate function to perform the transform using the word2vecModel
3 def transformTweetToWordVector(words:Iterable[String]):Iterable[Vector] =
4 words.map(w=>Try(word2vecModel.transform(w)))
5 .filter(_.isSuccess)
6 .map(x=>x.get.mean)
7
8 //Apply the transform to all the tweets in the training data
9 val trainTweetsFeaturesRDD = trainTweetsRDD mapValues transformTweetToWordVector
10

```

Figure 53: Feature transformation for tweet text

## 9.9 Classification

As described in the Section 5.7 for implementation of the classifier, Figure 54 shows the exact way we have implemented it for our project.

```

1
2 ▾ /** This is the method that generates the trains the classifier and
3 * also generates predictions.
4 */
5
6 def GeneratePredictions(trainingData: RDD[LabeledPoint], //Training data RDD
7                         testData: RDD[(LabeledPoint, String)], //Test Data RDD
8                         sc:SparkContext, //SparkContext passed from upstream call
9                         bcNumberOfClasses: Int, //Number of classes to classify
10                        bcLRClassifierModelFilename:String): //Classifier model file name
11 (RDD[(Double, Double)], Long) =
12 {
13     //Initialize the model type
14     var logisticRegressionModel: LogisticRegressionModel = null
15
16     try {
17         //Load the model from HDFS if it exists
18         logisticRegressionModel = LogisticRegressionModel.load(sc, bcLRClassifierModelFilename)
19         println(s"Classifier Model file found:${bcLRClassifierModelFilename}. Loading model.")
20     }
21     catch{
22         case ioe: IOException =>
23             println(s"Classifier Model not found at ${bcLRClassifierModelFilename}. Creating model.")
24
25             //Retrain the classifier
26             logisticRegressionModel = GenerateOptimizedModel(trainingData, bcNumberOfClasses)
27
28             //Save the model in HDFS for future use
29             logisticRegressionModel.save(sc, bcLRClassifierModelFilename);
30             println(s"Saved classifier model as ${bcLRClassifierModelFilename} .")
31     }
32
33     //Generate the predictions and probabilities for each tweet.
34     val logisticRegressionPredictions = testData
35     .map { case (LabeledPoint(label, features), tweetText) =>
36         val (prediction, probabilities) = ClassificationUtility
37         .predictPoint(features, logisticRegressionModel)
38         (prediction, label, probabilities, tweetText)
39     }
40 }
41

```

Figure 54: Implementation code snippet for the classifier

To measure the accuracy of the classifier, we implemented a helper function that provides micro and macro averaged F1 scores, along with the precision and false positive counts. Figure 55 shows our implementation to generate the classifier metrics for the project.

```

1
2 ▾ /** This is the method that generates the metrics for the predicted results
3   */
4   def GenerateClassifierMetrics(predictionAndLabels: RDD[(Double, Double)]//predicted and actual labels
5     ,classifierType : String, //Classifier name
6     bcNumberOfClasses: Int) //Number of classes
7 ▾   : Unit = {
8     // Get evaluation metrics.
9     val metrics = new MulticlassMetrics(predictionAndLabels)
10
11    //Iterate through the class results and print the F1-Measure, truePositiveRate and falsePositiveRate
12    //for each class
13 ▾   for (i <- 0 to bcNumberOfClasses - 1) {
14 ▾     //for (i <- uniqueLabels) {
15       val classLabel = i
16       println(s"\n***** Class:$classLabel *****")
17       println(s"F1 Score:${metrics.fMeasure(classLabel)}")
18       println(s"True Positive:${metrics.truePositiveRate(classLabel)}")
19       println(s"False Positive:${metrics.falsePositiveRate(classLabel)}")
20     }
21
22    //Print the confusionMatrix for the classification results
23    println(s"\nConfusion Matrix \n${metrics.confusionMatrix}")
24
25    val f1Measure = metrics.weightedFMeasure
26    val precision = metrics.weightedPrecision
27    val recall = metrics.weightedRecall
28
29    //Print the micro-averaged F1-Measure along with weighted precision and recall
30    println(s"\n***** Classifier Results for $classifierType *****")
31    println(s"F1-Measure = $f1Measure")
32    println(s"Weighted Precision = $precision")
33    println(s"Weighted Recall = $recall")
34  }
35

```

Figure 55: Implementation code snippet for generating classifier metrics

Figure 56 shows the per class metrics for a few classes.

```

37 ***** Class:3 *****
38 F1 Score:0.9484536082474226
39 True Positive:0.9787234042553191
40 False Positive:0.014598540145985401
41
42 ***** Class:4 *****
43 F1 Score:0.9357798165137614
44 True Positive:0.8947368421052632
45 False Positive:0.003787878787878788
46
47 ***** Class:5 *****
48 F1 Score:0.9885057471264368
49 True Positive:1.0
50 False Positive:0.0035971223021582736

```

Figure 56: Classifier metrics per class

Figure 57 shows the confusion matrix and overall results of the classification. The counts along the diagonal shows the true positives, and all the other numbers in the matrix are the misclassification counts. Also shown are the micro-averaged F1-score along with weighted precision and recall. These numbers are sample representation of the confusion matrix and the overall results and in no way, indicative of the actual experimental results for our project.

```

55 Confusion Matrix
56 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
57 0.0 56.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
58 0.0 0.0 4.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
59 0.0 1.0 0.0 46.0 0.0 0.0 0.0 0.0 0.0 0.0
60 0.0 1.0 0.0 2.0 51.0 0.0 0.0 1.0 0.0 2.0
61 0.0 0.0 0.0 0.0 0.0 43.0 0.0 0.0 0.0 0.0
62 8.0 0.0 0.0 0.0 0.0 0.0 0.0 38.0 0.0 0.0
63 10.0 0.0 0.0 0.0 0.0 1.0 0.0 8.0 0.0 0.0
64 19.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 11.0 0.0
65 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 15.0
66
67
68 ***** Classifier Results for Logistic Regression *****
69 F1-Measure = 0.8818812204822138
70 Weighted Precision = 0.9546278471769932
71 Weighted Recall = 0.8473520249221184
72

```

Figure 57: Confusion Matrix and overall results for the classifier

## 9.10 Probability emission by the classifier

The probability code can be found in the `ClassificationUtility.scala` file. This code uses the logistic classifier to generate the probabilities that a tweet belongs to each class. This function can be called in place of the classifier prediction, and instead of just giving a prediction, it will add an array of doubles representing the probabilities that the tweet is of that class.

An example of using the `ClassificationUtility.scala` file can be found in Figure 58.

```

//Generate the predictions and probabilities for each tweet.
// These can then be thresholded to not classify tweets with low probabilities
val logisticRegressionPredictions = testData
    .map {
      case (LabeledPoint(label, features), tweetText) =>
        val (prediction, probabilities) = ClassificationUtility
            .predictPoint(features, logisticRegressionModel)
            (prediction, label, probabilities, tweetText)
    }

```

Figure 58: Use of `ClassificationUtility.scala`

Note that the probability generation code normalizes the probabilities produced. This can be removed if desired. The normalization code is shown in Figure 59.

```

val sumProbabilities = classProbabilities.sum
//Normalize probabilities across the classes
(0 until model.numClasses - 1).foreach { i =>
  // normalize the specific probability at i
  classProbabilities(i) = classProbabilities(i) / sumProbabilities
}

```

Figure 59: Probability normalization

## 9.11 Spark partitioning and caching

To take advantage of the entire cluster the data must be parallelized across the cluster. This is accomplished by doing as much work in RDDs as possible. When the work is done in RDDs it can be executed on multiple nodes simultaneously. We found in particular the Spark UI interface provided by the cluster to be valuable in seeing how parallelized the data is. A snapshot of the interface is shown in Figure 60. Each horizontal section represents the work performed by an individual executor. The horizontal axis is the time that the spark job has been running. This snapshot shows that tasks are being executed in parallel as they appear vertical on the time axis. When multiple tasks are being executed at the same time, the tasks are being well distributed across the cluster. If we see the tasks executing in sequence, then that will pinpoint the place where optimizations can occur.

Note that the UI is only available when a task is running.

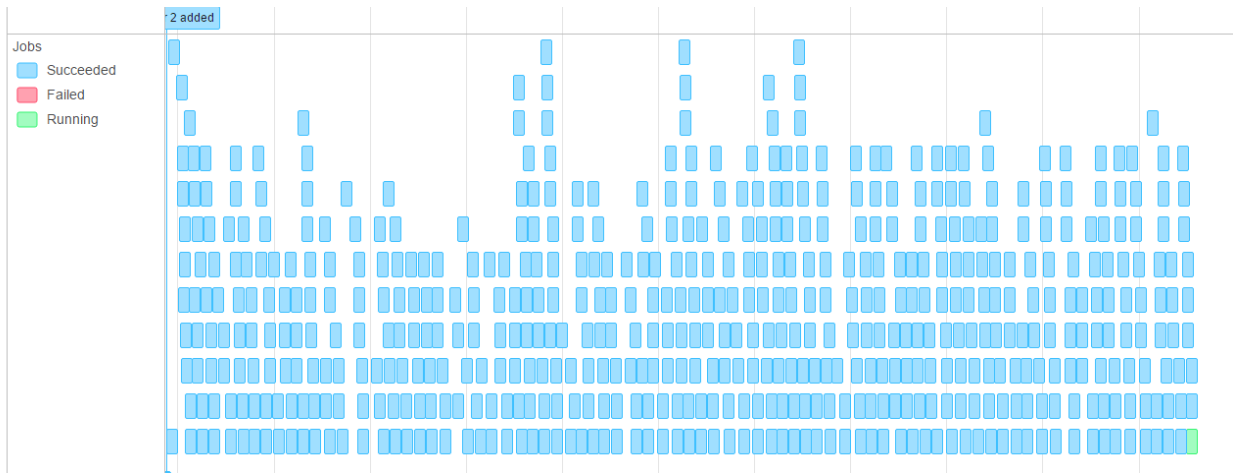


Figure 60: Spark UI example

The UI can be viewed by first setting up port forwarding. Port forwarding will let your local browser connect to the UI on the cluster.

The command we used to do port forwarding is found in Figure 61. The user then directs their browser to localhost:4040 to see the running job.

```
# port forwarding command that will also spawn a shell for the user cs5604f16_cla
ssh -L 4040:localhost:4040 cs5604f16_cla@hadoop.dlib.vt.edu
```

Figure 61: Port forwarding command to configure access to the Spark UI

## 10 Conclusion

The classifier that we have developed based on Word2Vec with Logistic Regression gives us very good accuracy (0.96 F1 score) when applied to a sample of 9 classes. This result also performed better than the AR classifier which had an F1 score of 0.90.

We focused on cleaning the tweets that we have in the collections and this pre-processing step gave us improved results with both the classifiers. This proved that cleaning the text after understanding the domain (Twitter) does result in better F1 scores for both the classifiers, as shown through experimentation.

In addition to classifying a tweet, we also generate the probabilities for a tweet for all the classes. There were a large number of instances where a tweet was classified into multiple classes. This suggests that it is also important to analyze the hierarchical structure in our classes. We need to have one broad category for each combination of sub-categories. This would ensure that if a tweet has equal probabilities across multiple sub-categories, it will be classified into the broad category instead.

Using the partitioning and caching operations on the data provided by the Spark library functions gave us a 57% faster run-time performance over our basic implementation. The Spark/Hadoop development platform lets us monitor the performance metrics in detail. The platform documentation on task optimization enabled us to identify the various performance bottlenecks visually and design the corresponding optimization strategy accordingly.

The tweet collections that we have in HBase to perform classification on, vary in size. We faced out-of-memory exceptions while loading some huge (4 million tweets) collections in memory. We resolved this issue by refactoring our implementation and performing the read from HBase in blocks. Performing a blocked read and then distributing this data across the cluster for further execution of classification tasks gave us a tremendous boost in run-time performance. It also gives us reliability in terms of being able to complete classification of huge collections.

Overall, we conclude that there is no single technique in the text classification domain that would contribute to the accuracy dramatically. Each component in our classification pipeline contributes to improving the accuracy and must be adjusted to fit the problem domain. While the accuracy of the classifier is vital, it should not come at the cost of large performance degradation. The use of platform tools to keep an eye on runtime performance and optimize it from time to time is equally important.



# 11 Future Work

To fully explore the problem of classifying tweets as well as the results of our experiment we have identified the following future work:

1. To get a better feature selection for the classifier we propose using a larger corpus such as Google News, or our entire tweets and webpage collection, to train the Word2Vec model. This will give more documents for the model to build the syntactic and semantic relationships for the words. Currently by training on the training set itself, the model cannot generate any word vectors for terms not present in the training set. This is a big limitation of the system. It will be interesting to see which of these bigger corpuses provide us the best classification accuracy empirically.
2. As additional classes are added we expect the accuracy of the classifier to decrease as it has to distinguish across more classes. We need to investigate whether the accuracy actually decreases in such cases, and then experiment with approaches to classify large numbers of classes without the accuracy penalty, such as breaking up the multiclass classifier into one classifier for each broad category or using ensemble methods for voting based prediction across multiple classifiers.
3. We would like to classify large collections using ours and the AR classifier. We would then want to perform random sampling of the labels that are generated for the tweets by both the classifiers and calculate the kappa value for the inter-classifier agreement between the two classifiers. This would help us understand the agreements and disagreements between the classifiers and assist in evaluating whether an ensemble of AR and our classifier would be effective for classification of tweets.
4. We would like to evaluate the classification results by using the output of the clustering team and comparing the clusters formed with the classes predicted. We propose taking all of the classified tweets and performing clustering on the entire set. Set the number of clusters equal to the number of real world events. Then, see if the clusters formed have tweets that are of the same class, and analyze any differences that occur.
5. The original goal of the classification team was to perform classification on the web pages too. This was curtailed from the scope due to the limited capacity available in the classification team. It would be interesting to modify the techniques used to classify web pages and tune it to achieve a satisfactory classification accuracy for web pages.
6. Based on recent research [6] it was found that text classification achieves state of the art results if the word vectors are trained on multiple broad classes separately and tweets from a broad class are classified using the specific trained model. It would be interesting to train multiple classifiers on broad categories like hurricanes, floods, and shootings, and then measure the efficacy of the classifiers empirically.
7. A further analysis of the probabilities should be done to distinguish between tweets that are part of multiple classes and those that are ambiguous. This will allow a tweet to be classified into multiple real world events. This could be useful for finding events that are similar as well as providing a better classification of the data.
8. To make the system be able to be used on multiple HBase tables and write to different columns than the project's schema we should place hardcoded values in a configuration

file. This file could also be used to store the names of the model files that we use for loading the Word2Vec and the Logistic Regression model at run-time. This file would be read in at the startup of every script to target the reading and writing of tweets to the proper tables. The fields that would be configured are the table names, column family names, column names, and the names of the model files.

## **12 Acknowledgements**

We would like to acknowledge and thank the following for assisting and supporting us throughout this project.

- Dr. Edward Fox, Dr. Denilson Alves Pereira
- NSF grant IIS - 1619028, III: Small: Collaborative Research: Global Event and Trend Archive Research (GETAR)
- NSF grant IIS - 1319578, III: Small: Integrated Digital Event Archiving and Library (IDEAL)
- Digital Library Research Laboratory
- Graduate Research Assistant – Sunshin Lee
- All teams in the Fall 2016 class for CS 5604

## 13 References

- [1] C. D. Manning, P. Raghavan and H. Schütze, *An Introduction to Information Retrieval*, vol. 1, Cambridge: Cambridge University Press, 2008.
- [2] P. Meesad, P. Boonrawd and V. Nui pian, "A chi-square-test for word importance differentiation in text classification," *Proceedings of International Conference on Information and Electronics Engineering*, pp. 110-114, 2011.
- [3] S. R. Singh, H. A. Murthy and T. A. Gonsalves, "Feature Selection for Text Classification Based on Gini Coefficient of Inequality," *FSDM*, vol. 10, pp. 76-85, 2010.
- [4] Y. Bengio, R. Ducharme, P. Vincent and C. Janvin, "A neural probabilistic language model," *The Journal of Machine Learning Research*, vol. 3, pp. 1137-1155, 2003.
- [5] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *eprint arXiv:1301.3781*, 2013.
- [6] P. Jin, Y. Zhang, X. Chen and Y. Xia, "Bag-of-Embeddings for Text Classification," *International Joint Conference on Artificial Intelligence*, no. 25, pp. 2824-2830, 2016.
- [7] S. K. Pal and S. Mitra, "Multilayer perceptron, fuzzy sets, and classification," *IEEE Transactions on Neural Networks*, vol. 3, pp. 683-697, 1992.
- [8] D. A. Pereira, E. E. Silva and A. A. Esmin, "Disambiguating publication venue titles using association rules.," *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries*, pp. 77-85, 2014.
- [9] Z. Chase Lipton, C. Elkan and B. Narayanaswamy, "Thresholding Classifiers to Maximize F1 Score," *eprint arXiv:1402.1892*, 2014.
- [10] "What is Apache Hadoop?," 2016. [Online]. Available: <http://hadoop.apache.org/>.
- [11] "Apache Spark is a fast and general engine for large-scale data processing," October 2016. [Online]. Available: <http://spark.apache.org/>.
- [12] "Welcome to Apache HBase," 09 October 2016. [Online]. Available: <http://hbase.apache.org/>.
- [13] "APACHE HADOOP HDFS," 2016. [Online]. Available: <http://hortonworks.com/apache/hdfs/>.
- [14] "MLlib is Apache Spark's scalable machine learning library," 2016. [Online]. Available: <http://spark.apache.org/mllib/>.

- [15] JetBrains, "IntelliJ IDEA," [Online]. Available: <https://www.jetbrains.com/idea/>. [Accessed 11 October 2016].
- [16] R. Jindal, R. Malhotra and A. Jain, "Techniques for text classification: Literature review and current trends," *webology*, vol. 12, no. 2, pp. 1-28, 2015.
- [17] C. McCormick, "Word2Vec Tutorial - The Skip-Gram Model," 19 April 2016. [Online]. Available: <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model>. [Accessed 2016].
- [18] Emit Classifier Probability in Spark for Logistic Regression, <http://stackoverflow.com/questions/30391399/predicting-probabilities-in-logistic-regression-model-in-apache-spark-mllib/36238801#36238801>.
- [19] A. Or, "Understanding your Apache Spark Application Through Visualization," 22 June 2015. [Online]. Available: <https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html>. [Accessed 2016].
- [20] "Crontab – Quick Reference," Admin's Choice, [Online]. Available: <http://www.adminschoice.com/crontab-quick-reference>. [Accessed 2016].
- [21] American Press Institute, "Twitter and the News: How people use the social network to learn about the world," 1 September 2015. [Online]. [Accessed 2016].
- [22] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD international conference on Management of data*, pp. 1-12, 2000.
- [23] D. D. Lewis, "Reuters-21578 text categorization test collection, distribution 1.0," 1997. [Online]. Available: <http://www.research.att.com/~lewis/reuters21578.html>.
- [24] C. Sherman, "Humans Do It Better: Inside the Open Directory Project," July 2000. [Online]. Available: <http://www.infotoday.com/online/OL2000/sherman7.html>. [Accessed 2016].
- [25] M. Oakes, R. Gaaizauskas, F. H. A. Jonsson, V. Wan and M. Beaulieu, "A method based on the chi-square test for document classification," *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 440-441, September 2001.