ACM Word Template for SIG Site

Egezon Berisha
RIT
exb3825@rit.edu

Thad Billig
RIT
tabdar@rit.edu

Eitan Romanoff
RIT
ear7631@rit.edu

ABSTRACT

Expression of ideas and sentiment over social networks has been increasingly popular over the last few years, leading to both a time where such information is easy to broadcast, as well as easy to acquire, notably through services like Facebook and Twitter. Likewise, there is an obvious desire for people to utilize this information within a variety of different domains, usually for the purpose of trend and sentiment analysis for marketing purposes. However, just as the information is freely available, it is also abundant, and performing analysis on datasets of exceedingly large sizes is an increasingly common issue. In this paper, we will discuss the methodology utilized to analyze large samples of Twitter data on a chosen subject for sentiment and trend analysis.

General Terms
Algorithms, Measurement, Standardization, Languages.

Keywords
Amazon Web Services, Twitter, Mahout, Hadoop.

1. INTRODUCTION

Many businesses need to evaluate the outcome of social media efforts. To accomplish this there are big data acquisition and storage problems that need to be solved in addition to the actual natural language processing algorithms that need to be used in order to evaluate sentiment on a given topic. Distributed systems and distributed analytic tools are necessary to accomplish these tasks on acquired data.

This paper will demonstrate a solution built on top of several technologies that target the storage and handling of large datasets. Our project is an exploratory one that seeks to answer several key questions in this application of data mining – how can large sets of short fragmented natural language texts be stored and analyzed, and how can one meaningfully analyze these data sets in relevant ways to this domain of marketing? Our processes in mining the data must be scalable over large datasets, and will use current technologies marketed towards this use case. Thus, Amazon Web Services (AWS), and its various services will be leveraged, including the AWS Simple Storage Service (AWS S3) and the AWS Elastic Compute Cloud (AWS EC2) service. The AWS EC2 platform has out-of-the-box functionality built on top of the Hadoop map-reduce framework. Furthermore, the Mahout Data mining platform will be leveraged for all of our natural language processing and mining. Because Mahout is based on the Hadoop system, it can also run on the EC2 cloud.

For data analysis itself, with regards to Twitter data, one can propose two core questions with regards to any Twitter dataset. First, one may wish to know what the relevant topics contained within the tweets are, and second, what the overall sentiment is towards those topics. The Mahout system contains several data mining and machine learning algorithms that can tackle these problems, including Latent Dirichlet Allocation – an algorithm that serves to identify key topics in text corpora, as well as a variety of standard classification techniques which can be applied to the classification of positive or negative sentiment.

The overall workflow process is as follows:

1. A sufficiently large population of Twitter data will be captured for big data analysis.

2. Twitter data will acquired by taking advantage of the Twython Python library, using a Python program, and pulling data from Twitter VIA the streaming API. The streaming API that Twitter publicly provides allows for a limit of 1% of the entire Twitter stream, and also allows for filters over that stream.

3. Streamed data will be stored as JSON in textual files, and uploaded to an AWS S3 bucket in fixed size chunks.

4. Data stored in an AWS S3 bucket can then be used by an AWS EC2 instance seamlessly. Various classification, and natural language processes will be executed at this stage.

The rest of the paper will be organized as follows: We will discuss the chosen natural language processing algorithms applied to the data, as well as their various purposes in section 2. In section 3, we will discuss the architecture of our system, and the dynamics of communication between its parts as well as the problems solved by our method of storage. In section 4 we will present the results of our experimentation against identified Twitter data sets. In section 5, we will present recommendations for additional work and unresolved issues.

2. NATURAL LANGUAGE PROCESSING ALGORITHMS

Perhaps the most important aspect of handling text that is presented in a natural context, that is, the context of naturally flowing sentence structure, is in the extraction of features for analysis. Prior to the implementation of this project, we explored various algorithms and processes that are performed on natural text so that during implementation, these algorithms could be leveraged appropriately. We looked at two types of algorithms in particular: those that fit into the preprocessing stage, and those that deal with topic extraction on processed data.

2.1 Stemming

Naturally occurring text is difficult to process due to the naturally high dimensionality of language. Not only can a word have several meanings depending on the context to which it is used, but each word can also have several forms. For example, the words dimension, dimensional, and dimensionality are all forms of the same root word. Unfortunately, when performing processes that are based off word frequencies, the overall count of the same root word is split between the various forms of the word. It is often, then, a good idea to reduce this dimensionality by modifying the text in such a manner that the root words remain, but the various forms of the word are removed, thus strengthening recurring words as features. This process is known as stemming.

The stemming process has been around for some time, and has several different implementations. The Lovins stemmer was developed in 1968, and acted as the basis for the popularized (and de-facto standard) Martin Porter stemmer, published in 1980. Both methods of stemming can be reduced to a simple rule-based approach. The Porter stemming process is a fast process that is performed on a word-by-word basis, reducing the word to a "stem" by removing modifications in a linear manner that follows a decision-tree-like process. For example, a word may first remove all pluralization endings, followed by -ed or -ing if present. Depending on if the second step was relevant for a particular word, different actions may be taken.

In our system's current version, stemming has not been used in the preprocessing phase for two reasons. First, because our current implementation only runs on a single master node, the stemming process cannot be distributed to different nodes, and while the process only takes a few seconds on megabyte text chunks, that number increases as the scale of the data portion increases. While open source java implementation of this particular stemmer exist, the process would have to be converted to a MapReduce job to take advantage of a Hadoop cluster. The second reason is that running this algorithm means modifying the actual text in each tweet instance. If this was done, the final extracted topics for a particular portion of tweets in our dataset may be unintelligible, as stems are often not actual words. We leave the idea of incorporating stemming into the system as potential future work.

2.2 Topic Extraction

Our system, at its core, attempts to record the change in trends over time of a twitter stream. While term frequencies can be analyzed, such an approach would entirely rely on the preprocessing techniques employed in removing all stop words. Furthermore, such a simplistic technique does not create word associations, especially in the case when a topic or subject is represented with multiple synonyms or related words. Mahout contains an implementation of the Latent Dirichlet Allocation algorithm, which creates a statistical model of the topic distributions in natural text instances.

LDA is a machine learning algorithm introduced by Blei et al in 2003 that refines a guess on the most common topics present in a series of natural text instances. Furthermore, LDA compiles a list of the terms linked to that particular topic. LDA supposes that a document is a series of words that reflect a set of topics which has a particular prior distribution, which in this case follows the Dirichlet prior. A document can therefore be generated by selecting topics from a topic distribution, and words within a topic following this distribution as well. LDA is a generative model, so it attempts to "make up" these topics using various sampling techniques, and refines these guesses with multiple passes based on the premise that correctly chosen topics will maximize the presence of words within that topic.

LDA has been shown to work in several different use-cases, effectively identifying topics within larger documents with high accuracy. [7] Furthermore, the algorithm has been shown to be parallelizable, and can run on a distributed framework.[10] For this project, we seek to use an implementation of this algorithm that included in a system built to scale, and on a large dataset of small instances. We have decided to use the Apache Mahout in our software stack because of its implementation of LDA, which takes advantage of the Hadoop MapReduce framework.

3. SYSTEM ARCHITECTURE

3.1 Amazon Web Services Configuration

3.1.1 AWS Setup

Initially in phase 1, an initial t1.micro instance instance of Ubuntu 12.0.04 on Amazon Web Services EC2 was created to support an installation of Mahout, and direct the storage of streamed Twitter data. The initial setup provided access to the root user, and allowed for the installation of python, allowing for the data acquisition script to run without problems. The micro instance, however, only provided a single core for processing, and only approximately 512mb of RAM. As a result, there was not enough free memory to support the installation of Mahout, as the installation of both Hadoop and Mahout require at least 2 GB of memory to be allocated to the JRE heap space. Mahout's core unit tests failed in this environment, citing problems with the JVM heap memory allocation. Because of this, we abandoned the usage of free instances on AWS.

We had to therefore switch and re-configure a larger instance, thus we configured an extra-large instance (m3.xlarge) so that installation would not be halted due to hardware requirements. It should be noted that when the process is eventually set up to work as a distributed system, each node will have to meet the minimum specification. The m1.large instance comes with 4 cpus, 2 cores, and 15 GB of memory - enough to walk through the installation of Mahout and not fail on any of the core unit tests.

The AWS instance that we decided to use is a public instance (ami-5055c260), and the operating system is CentOS 5.4 (Final). At this point we did not worry about the security, because we wanted to test different connections such as SSH, FTP, HTTP, etc. Therefore in the security group we opened all TCP ports of the AWS instance. It should also be noted that each time you power down and power up the instance, the new public dns name will be generated, which you need to connect to to the instance. It should be noted that CentOS 5.4 comes with an antiquated version of python 2.4. For this project, our python modules have dependencies on packages that require python 2.6 or higher. We then recompiled python's 2.7 release, and used it as the default python version for this EC2 image

3.2 S3

Once this AWS instance was set up, we configured the S3 bucket to be used for file storage, and it was mounted as a drive by using the S3FS program. Setup was generally simple, though all programs must be run with root permissions to have access to the bucket. We also needed to install prerequisites before installing s3fs on CentOS. Required packages were installed through yum and included: gcc; libstdc++-devel; gcc-c++; curl; curl*; curl-devel; libxml2; libxml2*; libxml2-devel; and openssl-devel. In addition to the required packages, we also needed to compile and install fuse. Finally we needed to create a credential file to connect to our S3 bucket, from EC2 instance. The name of the file which holds the credentials should be passwd-s3fs located in the /etc directory, the file should contain the following line:

```
bucketName:accessKeyId:secretAccessKey
```

where the bucket name is the actual name of the S3 bucket, and the access key id and secret access key are located in AWS my account, under the Access Credentials in Security Credentials. One more thing that is worth noting is that the passwd-s3fs should not have worldwide editing permission. We used the command chmod 600 passwd-s3fs, which means only the owner has the write and read permissions and nobody else.

At this point we have installed and configured s3fs, and it is ready to be used. We used the following command to mount our bucket: `s3fs bucketName mountpoint /mnt/mountFolder`. s3fs is the name of the program that does the mounting, bucketName is the name of our bucket in S3, and the mount point /mnt/mountFolder is the folder which contains the S3 bucket.

3.2.1 S3 Issues

The s3fs itself has some issues and limitations. Some of the s3fs limitations are: 1) Files/Objects cannot contain more than 5GB. 2) You cannot update a file/object partially, for example if you want to update 2KB in a 500MB file, then you would have to re-upload the entire 500MB. Initially we wanted upload the all the Twitter data that we gathered, however we experienced difficulties doing that. When we gather the Twitter data, we are splitting each tweet and saving into a text file, therefore generating thousands of text files with a size 3 to 4 KB per file. We tried different methods for transferring the files to S3 bucket. The first method was just to move the files from EC2 to S3, and that was very slow. The second method was to create a script to gzip each text file and to transfer it over to S3 bucket, but this was slow as well because it would take a lot of time to unzip the file when the file was in S3 bucket. The third method was to tar the whole directory that contains the text files, which was about 150 MB, and then transfer it over to S3 and untar it. This method seemed feasible, and the tar file transferred in about 5 minutes, but we quickly found out that while un-taring the directory in the S3 bucket it took a long time because it was un-taring each file individually, and there were about 32,500 files. After doing extensive research and based on our experience we identified the problem. S3 bucket seemed fast, but comes with overhead. It takes 200 ms for writes and 350 ms for reads per each S3 transaction, meaning we can only write about 5 files a second sequentially. For example if we had 2 KB files, we would be able to transfer about 5 files per second, and we had thousands of files in about 150 MB. Therefore that was the reason why it would take so long to transfer the files to S2 bucket. We decided to keep everything locally as a viable solution for the moment.

3.3 Data Acquisition

In order to acquire a large dataset from Twitter, it is necessary to use the public API to pull in a local copy of tweets. Twitter provides two forms of interfacing with their service: the standard polling API that works based on search queries for tweets on certain criteria, as well as the streaming interface, that allows for access to a certain percentage of the entire Twitter stream.

The polling API is a RESTful interface for making queries on messages in the form of simple `GET` web requests. These tweets are not selected from a random sample of the overall Twitter stream, allowing for accurate searching of particular tweets pertaining to given subject filters. This API comes with the cost of an API call limit in that Twitter imposes limits to the number of requests that can be handled by any particular IP address. As of May 2013, this API limits the number of API calls to 180 queries per IP address, but should the developer have a developer account (publicly available, and free to create), and authenticate using the Twitter OAuth API, this limit is increased to 450 queries per application for the rate limiting period. These limits are reset once per rate-limiting period, which is currently defined as 15 minutes.

Because of these rate limits, even with the authenticated 1,800 queries per hour, each query only results in 100 resulting tweets “per page”. Each page counts as a separate request. The “sample” from the full Twitter stream, on the other hand, may result in larger quantities of tweets, and is the suggested format for doing large amounts of data acquisition, as the results can be easily appended to a file in the same format to which it is received. Furthermore, the application does not have to implement logic for handling rate limits, or redundancy of duplicate tweets across multiple queries. It should be noted that all streaming endpoints have access to the same 1% sample. In terms of actual volume, the rate of tweets varies throughout the day, and the amount of data that can be collected from the standard 1% sample stream does not exceed the number of tweets that can be acquired through the polling API, but the rate of processing is much faster, as the data can be streamed to a single file destination where complete buffered portions of incoming text are simply appended to the file.

To actually acquire data from the streaming API, we created a series of python scripts to perform the acquisition tasks, data cleaning, file dividing, and so forth. We used the python programming language for these tasks because of its wide support on all systems, ease of use, and because of its interactive interpreter, which proved to be quite useful for trying out certain tasks and exploring how Twitter formats a tweet prior to sending it through the stream. Furthermore, there is a public python package called Tweepy[9] which creates a python module layer on top of the stream API calls, making the resulting code cleaner and simpler to read. The resulting utility script that we created can be found in the source code packaged with this project under the name **acquire.py**.

The **acquire.py** script is kept simple for agile use. The script contains the definition for a single class, `StdOutListener,` which is a class that overrides the callback functionality as defined by its superclass, Tweepy’s `StreamListener` class. Essentially, all it does is create a handle to a file for appending its information, appending the data on execution of the `on_data()` callback. For convenience, the program can take in an additional file of “filter topics” to filter out tweets that are not relevant to the topics listed in the file. Finally, the program can halt by sending it a keyboard interrupt, though only a portion of the most recently read tweet will have actually been written to the designated output Twitter dump file. This partial tweet gets ignored in the preprocessing stage of our workflow. For more information on how to use **acquire.py**, check the user manual in the appendix section.

3.3.1 Contents of the Twitter Dump File

The content pulled from acquire.py get stored as "Twitter dump" files which are tweets represented as JSON, concatenated together in a single file. Portions of JSON are delimited by carriage returns ("\r\n"). It is worth going over the anatomy of a tweet, and to identify which portions of the data were leveraged.

A typical tweet contains a lot of information, but for this project, most of it was left unused. Because the core of this project is to look at the popularity of *topics* over time, many of the metadata fields included in the tweet data structure are not needed. The id of a particular tweet is used for keeping the tweets "fragmented" in separate files, a necessary step for the vector creation (described in section 3.7). It is important to note that tweets contain two forms of ID: a numerical representation, which is unusable due to precision problems for many programming languages, and a string-based representation, **id_str**, which is what we used. The body of the tweet is stored in the **text** field, and contains unicode characters due to the variety of different languages supported. The **created_at** field is a timestamp for a particular tweet, represented as a string format, which is parsed and used for the binning process explained in section 3.4.1. Finally, each tweet contains an internal dictionary called **entities** that holds onto other metadata portions contained within the text itself, including an entry for **hashtags** (other included entities are urls and user mentions).

3.4 Preprocessing

The data collected was not stored as-is, nor was it stored in a particular database system. For the final version of our system, the data was stored in fragments on the host node's file system that could be analyzed by both the Hadoop and Mahout layers of the system (future work is to better leverage the underlying HDFS, as detailed in section 5). Due to time constraints, the overall system is still contained to pseudo-distributed mode, and has not been expanded to an actual cluster. Thus, for the sake of simplicity, the preprocessing stages were kept to quick instructions that could be executed at the same time as file fragmentation without largely impacting the performance. For our implementation, we bundled both the fragmentation process and the data cleaning process in the same utility script, which can be found in the source code packaged with this project under the name **fragment.py**.

Like acquire.py, **fragment.py** is kept simple so that it could be run in an automated process without much difficulty. The fragment.py script is kept to be linear, and parses through each JSON chunk stored in the Twitter dump file outlined in 3.3.1. Fragment.py loads the data dump and splits the JSON chunks by using the carriage return as a delimeter. For each chunk, the built-in **json** module was used to load in the string representation into a python dictionary. Using this dictionary, the desired fields could be easily extracted and analyzed. The id_str field is first extracted, to be used in the fragment file name. In the case that a tweet has no id_str field, the tweet is a "deleted" tweet, which is ignored. The timestamp is extracted and handed off to the **dateutil.parser** module - a third party date-time parser that is robust, and can handle many input formats. From this, a date object is returned with all fields (year, month, etc) as numbered ints. The text is also extracted and handed off to a few other quick and standard processes. Finally, the hashtags are pulled out of the JSON chunk.

The text portion of each tweet goes through a fast linear cleaning process. First, all punctuation in the tweet is stripped. Then, the text is stripped of most "stop words", which for the sake of speed is identified by the length of the word through a standard python regular expression. Finally, the text is normalized in terms of case sensitivity, reducing the entire string to lowercase letters. As an optional argument, **fragment.py** can be supplied with a CSV file containing a list of all words to be stripped. The text is then examined word-by-word and each word matches against these redacted entries, which are stored in a hashed set to keep the operation fast.

3.4.1 Data Fragmentation

Each JSON fragment of the Twitter data dump is stored within its own file, replicated in three locations on the file system - one for each fragment family. For posterity, a copy of the JSON is stored within the json_fragments directory structure. The cleaned text portion is stored within the content_fragments directory structure. Finally, the hashtags are stored within the hashtag_fragments directory structure. The three files all share the same name, which is the ID of the tweet in string form. For each of the structures, the parent directory is the year of the parsed date, which leads into a month, day, and hour substructure. In organizing the files this way, necessary vectors could be parsed by recursively going through a particular directory. Storing the files in this way, however, comes at the cost of tripling the amount of data that is to be stored on disk.

3.5 Hadoop

Hadoop requires Java 1.5 (Java 5) and above, however it is recommended to have Java 1.6 to run Hadoop. At first we installed the open JDK (Java Development Kit) 6, JRE Headless, and JRE lib. The other tools we installed were: python-setuptools, simplejson=2.0.9, boto=1.8d, ant. To the Hadoop source we used the wget using the apache website http://apache.cyberuse.com//hadoop/core/hadoop-0.20.2/hadoop-0.20.2.tar.gz.

We installed Hadoop, and initially we wanted to test if Hadoop runs locally (single node). We decided to have a Hadoop dedicated user, even though it is not required, but nevertheless recommended, because we need to format the file system to HDFS and we wanted to keep Hadoop separately from other applications, software, and user accounts, because everything was running locally (single node). The Hadoop dedicated username was hdfs.

After we installed Hadoop, we added the following line in the hadoop.sh shell script:

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk/
# The maximum amount of heap to use, in MB. Default is
1000
export HADOOP_HEAPSIZE=2000
```

We also added the following lines of code in the core-site.xml and mapred-site.xml files:

```
<pre class="xml" name="code"><configuration>
<property>
<name>fs.default.name</name>
```

```
<value>hdfs://localhost:9000</value>
</property>   <property>
<name>mapred.job.tracker</name>
<value>localhost:9001</value>
</property>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
<property>
<name>hadoop.tmp.dir</name>
<value>/mnt/tmp/</value>
</property>
</configuration></pre>
```

In addition to that, we also added the following lines to our '.profile' file which is located in our home directory:

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
export HADOOP_HOME=/usr/local/hadoop-0.20.2
export HADOOP_CONF_DIR=/usr/local/hadoop-0.20.2/conf
```

We tested the Hadoop build in wordcount example, and everything worked as expected; the job ran in a mapreduce fashion and we were able to see the results.

3.5.1 Hadoop Automation

Running an AWS instance costs money per hour, therefore we needed to shutdown the instance every time we were not using it. Every time we turn the instance on, we need to manually start hadoop, and then work/troubleshoot mahout. In order to make this process more efficient, so we do not have to touch Hadoop at all, we decided to make Hadoop start on boot, thus we can go straight and run Algorithms on Mahout. There are 5 daemons/processes that needs to be running for Hadoop to function properly; the demons are datanode, jobtracker, namenode, secondarynamenode, tacktracker. We started off by executing the following commands line in the terminal:

```
chkconfig --level 345 hadoop-0.20-datanode on
chkconfig --level 345 hadoop-0.20-jobtracker on
chkconfig --level 345 hadoop-0.20-namenode on
chkconfig --level 345 hadoop-0.20-secondarynamenode on
chkconfig --level 345 hadoop-0.20-tasktracker on
```

The chkconfig command is used to activate/deactivate services at startup , --level specified the level, then is the service name, and the command on or off to activate/deactivate.

Running the commands above started the daemons at boot, however, because we formatted the file system as hdfs user, we needed to run the hadoop/bin/start-all.sh as hdfs user. Hadoop requires SSH access to manage the nodes, even though we are running locally (single node), the hdfs user needs to have ssh access to the machine(localhost). We logged in as hdfs user, and by executing the following command /.ssh/id_rsa.pub >> /.ssh/authorized_keys. The ssh keys were generated, and added to the keys file .ssh/authorized_keys. At this point, the hdfs user can execute the hadoop/bin/start-all.sh script. We login at AWS instance as root, and to make the process even more seamless,we added the following command "runuser -l hdfs -c 'bin/start-all.sh'" in the .bash_profile file of the root, this file gets executed every time the root logs in. Therefore, when we power on the instance and we log in, the command will be executed, and we do not have to login as hdfs user to run the script. The command runuser runs the command in quotes as hdfs user.

3.6 Mahout

Initially in phase 1, the installation and running Mahout on Ubuntu 12.0.04 first required the installation of Maven and Hadoop. We updated the apt-get application, and following that we installed tools needed to run Mahout, the tools we installed were: python-setuptools, simplejson=2.0.9, boto=1.8d, ant, subversion, maven 2. we also added the following lines to our .profile file which is located in our home directory.

```
export MAHOUT_HOME=/usr/local/mahout-0.4/
export MAHOUT_VERSION=0.4-SNAPSHOT
```

Between phase 1 and 2 we experienced problems on installing mahout, and in this section we will explain the troubles we went through in installing Mahout.

We checked out the Mahout repository using svn from the apache website http://svn.apache.org/repos/asf/mahout/trunk, and after that we tried to install Mahout by using the following maven command:

```
mvn clean install
```

The build process successfully continued and all the tests that ran succeeded, including the "core" mahout tests that test against Mahout's algorithm collection. However, after the tests seemed to be finished, the process failed in attempting to open a zip file. The build error was the following:
[INFO] Failed to create assembly: Error creating archive job: error in opening zip file

This error message did not explicitly identify the zip file and resulted in lost hours attempting to troubleshoot with no success.

After we changed the operating system to CentOS 5.4, we did not experience those problems anymore. Finally we were able to install and run all the mahout default tests successfully.

3.6.1 Maven

Maven is an open source Apache project that supports the works to standardize, simplify, and test the build of packages such as Mahout[1]. We installed maven 2, and we also added the following lines to our .profile file which is located in our home directory:

```
export MAVEN_OPTS=-Xmx1024m
```

3.7 Lucene Index Creation

3.7.1 SOLR Vectors

Most (if not all) of Mahout's internal algorithms are based on the Map Reduce for Machine Learning[3]. Many of these algorithms require the input text to be put in a specific weighted vector format, which is then consumed by the algorithm to perform classification or clustering. This process also applies to running LDA on natural language instances, and thus it is necessary to generate this vector.

The Mahout documentation lists two ways in particular to create this vector file.[4] The first suggested way is to create a Lucene index. A lucene index is an index that maps a particular word to the location of a file that contains the word. Because Mahout is an Apache project, the suggested method of creating a lucene index is by using another framework - in particular, a project called Apache SOLR[2]. SOLR is a solution that indexes the content of text from PDFs, HTML, Word, and Open Document formats. As a group we researched the methodologies for the generation of the index.

SOLR utilizes the Lucene Java search library and exposes a RESTful HTTP/XML web service for the update and retrieval of index information. Through multiple iterations of testing with the group's Twitter data feed, we were unable to successfully generate a Lucene index through the application on our data, regardless of the format (XML, JSON, even as a CSV - all of which were verified as correct in structure by other programs). Furthermore, attempting to remove a generated index through the web interface resulted in the package being unusable, and the service no longer launched.

This work was completed as part of phase 1 in our project work. At the conclusion of this work we discovered that there is an alternate process to making these vectors from plain files in a directory structure as documented in the section on Mahout Vector Generation below.

3.7.2 Mahout Vectors

Mahout also supports the creation of vector files directly. The first step for this is the creation of a sequence file in a chunked manner. Sequence files are the combination of large numbers of small documents into large sequences of binary pair keys. These are generals required to eliminate the need to read many small documents and rather parse through one larger sequence file generated. [5] After the sequence files are created we used seq2sparse to generate the weighted topic vectors to be used in Mahout's implementation of LDA. Mahout's implementation of LDA requires TF(term frequency) weighted vectors to complete and not TD-IDF(term frequency, inverse document frequency)[8] vectors. The actual implementation of LDA utilizes only term frequency; the algorithm itself handles the topic generation and the algorithm has an option to filter topics based on the maximum document frequency as detailed below.

3.7.3 Sequence File Generation

3.7.3.1 Seqdirectory

seqdirectory takes the location of a folder of documents and an output location for your resulting chunked sequence file. Our files were stored in UTF-8 format and were stored in the default chunk size of 64 mb.

```
seqdirectory \
--input <PARENT DIR WHERE DOCS ARE LOCATED> --output
<OUTPUT DIRECTORY> \
<-c <CHARSET NAME OF THE INPUT DOCUMENTS> {UTF-
8|cp1252|ascii...}> \
<-chunk <MAX SIZE OF EACH CHUNK in Megabytes> 64> \
<-prefix <PREFIX TO ADD TO THE DOCUMENT ID>>
```

```
seqdirectory –input <input location> -output
<location> -c UTF-8 –chunk 64
```

3.7.3.2 Seq2sparse

seq2sparse generates vectors from the output of the sequence file generation. For our implementation of the algorithm we evaluated the parameters in conjunction with our dataset to choose appropriate analysis values for a Twitter data set. In particular we needed to choose a weighting method of TF to support Mahout's CVB requirements. We set a value for minimum support for key word appearance of 1 for our data set. This is typically set at a value of two or higher for LDA analysis, but Twitter keywords are typically not duplicated within the body of a message. We also set the minimum document frequency at 50, this parameter sets the threshold for appearances across the various tweets to be included as an identified topic. We also set the Max DF Percentage of Documents(x) threshold at 99. This parameter is utilized to eliminate high frequency words from the content of an individual document. As an example a tweet containing "Hello Hello World" the word Hello would have a MaxDF value of 66% while World would have a MaxDF Value of 33%. Again because terms are not typically repeated in tweets we do not want to exclude content within a tweet unless it is at or above 99% of the content of the tweet. We implemented the default chunk size (chunk) and standard lucene analyzer, and created named vectors (nv). The last value we set as a parameter for the output of the seq2spare command was the normalization value. For this we chose infinite (INF) such that the process could determine the appropriate size of the vectors. Below is a list of the available parameters and a sample of the command we utilized to run our process.

```
seq2sparse \
-i <PATH TO THE SEQUENCEFILES> -o <OUTPUT DIRECTORY
WHERE VECTORS AND DICTIONARY IS GENERATED> \
<-wt <WEIGHTING METHOD USED> {tf|tfidf}> \
<-chunk <MAX SIZE OF DICTIONARY CHUNK IN MB TO KEEP IN
MEMORY> 100> \
<-a <NAME OF THE LUCENE ANALYZER TO TOKENIZE THE
DOCUMENT>
org.apache.lucene.analysis.standard.StandardAnalyzer>
\
<--minSupport <MINIMUM SUPPORT> 2> \
<--minDF <MINIMUM DOCUMENT FREQUENCY> 1> \
<--maxDFPercent <MAX PERCENTAGE OF DOCS FOR DF. VALUE
BETWEEN 0-100> 99> \
<--norm <REFER TO L_2 NORM ABOVE>{INF|integer >= 0}>"
<-seq <Create SequentialAccessVectors>{false|true
required for running some algorithms(LDA,Lanczos)}>"
```

```
/usr/lib/mahout/trunk/bin/mahout seq2sparse -i
/root/Twitter_files/seqfiles/0/ -o
/root/Twitter_files/vectors/0/ -wt TF -s 1 -md 50 -ow
-x 99 -chunk 100 -n INF -nv
```

3.7.3.3 Rowid

Mahout's implementation of LDA requires that sparse vectors are indexed with numerical values, the default vectors created using seq2sparse creates vectors with textual keys. This requirement was discovered through an initial implementation in Phase 1 of the algorithm with string based indexes and resulted in the following cvb algorithm erroring. This required the conversion of our TF vector file to rowids. To do this we utilized the Mahout rowid command shown below. The command requires an input directory and an output location to append rowids. Below is a sample execution of our command:

```
mahout rowid –I text_vec/tf_vectors –o sparse-vectors-
cvb
```

3.8 CVB

Mahout's implementation of LDA implements CVB(Collapsed Variational Bayes) instead of a native Latent Dirichlet Allocation algorithm because the CVB implementation is more efficient to paralyze. [7] In implementing this against our Twitter data sets, we researched the parameters to pass to the algorithm and applied them as shown below. Throughout this process there was much trial and error in determining not only the correct parameters but also in interpreting the errors the algorithm returned. For the purposes of this project we deployed this in a pseudo distributed mode due to constraints described earlier in the paper. Likewise the documentation on the Mahout pages did not accurately describe all of the available options of the algorithm. [7] In our implementation we passed the obvious values to the algorithm including input path(i), output path(o), and dictionary(dict). For the number of iterations through the model(x) we choose 20 to allow for appropriate convergence of the models. We chose to allow the algorithm to determine the number of unique features (nt), and utilized the default value of 1 for the number of learning iterations for each document(mipd) as the Twitter data does not have the depth of information that a word or pdf document would. We also allowed the algorithm to manage the the state of the model through the iterations. We experimented with various values for the number of latent topics (k) and settled on a value of 30 for the purposes of our experimentation. In running the process we noticed that there was an exponential increase in the amount of time required to process the data set as we increased the number of topics to be processed. Below is a breakout of the parameters and a sample of a command we executed.

```
cvb \
   -i <input path for document vectors> \
   -dict <path to term-dictionary file(s) , glob
expression supported> \
   -o <output path for topic-term distributions>
   -dt <output path for doc-topic distributions> \
   -k <number of latent topics> \
   -nt <number of unique features defined by input
document vectors>
   -mt <path to store model state after each
iteration> \
   -maxIter <max number of iterations> \
   -mipd <max number of iterations per doc for
learning> \
   -a <smoothing for doc topic distributions> \
   -e <smoothing for term topic distributions> \
   -seed <random seed> \
   -tf <fraction of data to hold for testing> \
   -block <number of iterations per perplexity check,
ignored unless test_set_percentage>0> \
```

```
/usr/lib/mahout/trunk/bin/mahout cvb -i
/root/Twitter_files/sparse_vectors_cvb/23/matrix -dict
/root/Twitter_files/vectors/23/dictionary.file-0 -o
/root/Twitter_files/output/23k30/ -k 30 -x 20
```

3.9 Vectordump

To generate the resulting topics from the LDA analysis Mahout implements vectordump. Vectordump takes as an input the results of the modeling process and outputs it in a vector containing topics and their weights. To generate the results we provided values for the dictionary utilized to generate the vectors(d), defined the dictionary type as a sequence file(dt), sorted the results(sort) and set the vector sizes (vs) for the result set returned. Below is a sample of a command we executed:

```
/usr/lib/mahout/trunk/bin/mahout vectordump -i
topic_term_dist4/ -d content_vectors/dictionary.file-0
-dt sequencefile -vs 5 -sort true
```

4. RESULTS

In generating the results of analysis we wanted to compare two pieces of the twitter results. First we wanted to review the results of LDA and look for similarities and attempt to verify/validate the topics that were generated through the execution of the algorithm. Our second goal was to evaluate the results of a simple map reduce job returning hash tag frequencies for separate time slices for interesting correlation between the time slices and the topics generated through LDA.

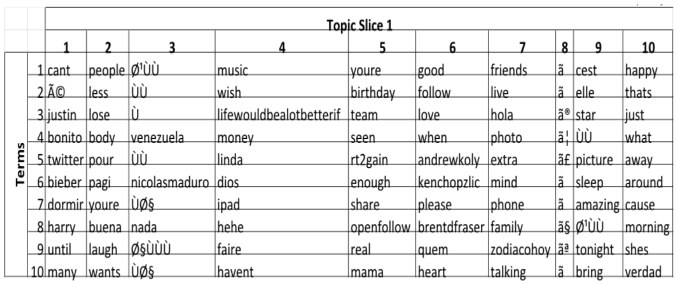
The datasets this process was run against were sized at approximately 200 MB each. The start to stop analysis of this process executed in just under an hour in a pseudo distributed mode. With the performance of the algorithm and analysis against the dataset for a 1% Twitter stream, this approach can be successfully be utilized to provide a near real-time analysis of topics and hashtags.

4.1 LDA Topic Results

In reviewing the results generated through LDA, we choose a size of ten for returned topics terms. This was an appropriate size for usage in identification of trends in topics and the relevant weights we saw in reviewing our result vectors. In our initial run of LDA we saw a large number of short (three characters or less) terms for topics and many filler words. This prompted us to perform additional preprocessing against the text of the tweets and eliminate terms for consideration in topics. However, in our final analysis of topics there were still a large number of terms that could be filtered from the general topic identification. Examples of these terms included cant, less, share, and live.

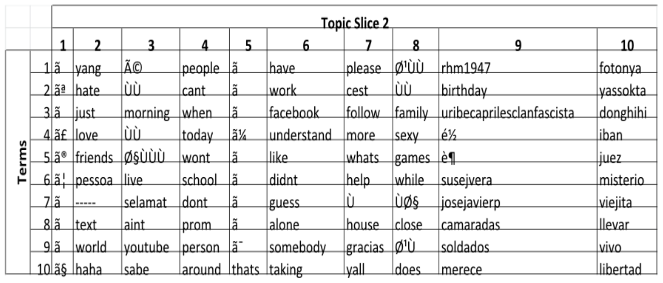
Another output of the process that was revealing was the discovery of many foreign language and non-standard US characters returned within the topics. This presents an opportunity to do more location based pre-processing on the dataset itself. Between the two time slices there didn't appear to be overlap in the topics, but typical across general terms that refined preprocessing would remove.

Unfortunately, in our analysis of the two time slices for topics and hashtag data we did not see any strong correlations between the topics and the hashtags. Additional pre-processing of the data could potentially lead to a stronger correlation between the resulting datasets.



		Topic Slice 1									
		1	2	3	4	5	6	7	8	9	10
Terms	1	cant	people	Ø¹ÙÙ	music	youre	good	friends	ã	cest	happy
	2	Ã©	less	ÙÙ	wish	birthday	follow	live	ã	elle	thats
	3	justin	lose	Ù	lifewouldbealotbetterif	team	love	hola	ã®	star	just
	4	bonito	body	venezuela	money	seen	when	photo	ã¦	ÙÙ	what
	5	twitter	pour	ÙÙ	linda	rt2gain	andrewkoly	extra	ã£	picture	away
	6	bieber	pagi	nicolasmaduro	dios	enough	kenchopzlic	mind	ã	sleep	around
	7	dormir	youre	ÙØ§	ipad	share	please	phone	ã	amazing	cause
	8	harry	buena	nada	hehe	openfollow	brentdfraser	family	ã§	Ø¹ÙÙ	morning
	9	until	laugh	Ø§ÙÙÙ	faire	real	quem	zodiacohoy	ãª	tonight	shes
	10	many	wants	ÙØ§	havent	mama	heart	talking	ã	bring	verdad

Figure 1. Topics generated for tweets on 5/4/13, 7:00 pm to 7:15pm



		Topic Slice 2									
		1	2	3	4	5	6	7	8	9	10
Terms	1	ã	yang	Ã©	people	ã	have	please	Ø¹ÙÙ	rhm1947	fotonya
	2	ãª	hate	ÙÙ	cant	ã	work	cest	ÙÙ	birthday	yassokta
	3	ã	just	morning	when	ã	facebook	follow	family	uribecaprilesclanfascista	donghihi
	4	ã£	love	ÙÙ	today	ã¼	understand	more	sexy	é½	iban
	5	ã®	friends	Ø§ÙÙÙ	wont	ã	like	whats	games	è¶	juez
	6	ã¦	pessoa	live	school	ã	didnt	help	while	susejvera	misterio
	7	ã	-----	selamat	dont	ã	guess	Ù	ÙØ§	josejavierp	viejita
	8	ã	text	aint	prom	ã	alone	house	close	camaradas	llevar
	9	ã	world	youtube	person	ã¯	somebody	gracias	Ø¹Ù	soldados	vivo
	10	ã§	haha	sabe	around	thats	taking	yall	does	merece	libertad

Figure 2. Topics generated for tweets on 5/4/13, 8:00 pm to 8:15pm

4.2 Hashtag Frequencies

In reviewing the results for hashtag frequencies we observed interesting similarities and trends over the two time slices displayed below. In both instances we see that rt, short for retweet, is considered a hashtag by Twitter. This presents an opportunity for future work to eliminate typical “filler” related hashtags from result sets and to generate a dictionary of exclusions for use in future analysis. Also of interest was the correlation and prevalence of a few hashtags in the top ten for frequency. In both timeslices generic hashtags related to teamfollowback, gameinsigh, lifewouldbealotbetterif, android appeared in both datasets. It was also interesting to see the inclusion very large hashtags in the top 10 for tweets, in contrast with the small amount of text available in typical tweets.

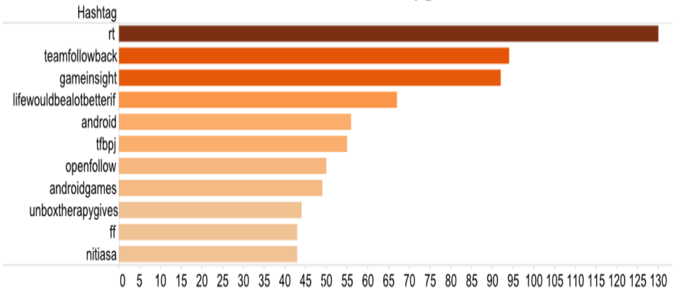


Figure 3. Hashtag frequencies generated for tweets on 5/4/13, 7:00 pm to 7:15pm

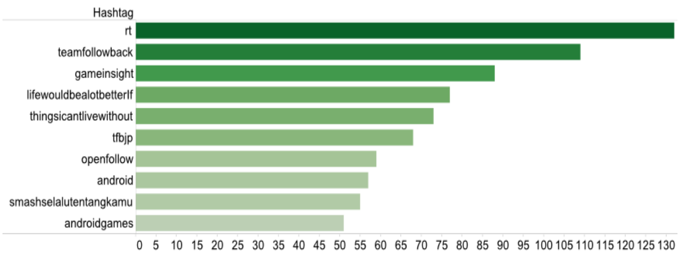


Figure 4. Hashtag frequencies generated for tweets on 5/4/13, 7:00 pm to 7:15pm

5. FUTURE WORK

As it stands right now, the system as a whole runs only in pseudo-distributed mode, and does not delegate its work to various worker nodes (these worker portions are services that run locally instead on another machine). While the AWS EC2 service makes it relatively easy to create virtual machines to serve as worker nodes, as was discussed in previous sections, the installation of Mahout was a time consuming process. Regardless, some initial steps were taken to convert some of our workflow processes to distributed MapReduce jobs, in particular the preprocessing portion of the system.

Because the Twitter files must necessarily be fragmented for the creation of the feature vectors, the map-reduce configuration for the pre-processing stage has a few restraints. The most important requirement is that the version of Hadoop must be new enough to have included in the API the MultipleOutputs module. The MapReduce preprocessor would take in a data dump file and partition it into the various JSON segments between worker mapping nodes (splitting phase). The mapping phase would perform the preprocessing operations, mapping both the tweet id string and the cleaned text together to an intermediate key as defined by the date. If the sheer volume is too large, the binning can be made to be more granular, binning by the minute instead of by the hour. Thus, the reduce process would take these mappings and concatenate each entry to the same file depending on the date. The reason for this is that HDFS is known to not perform well with large volumes of small files, and does much better with large files split between various nodes. Fragmenting for the vectors would be a separate process.

6. REFERENCES

[1] Apache Maven Project. (April 2013). Retrieved April 3, 2013 from Maven.apache.org/what-is-maven.html.
[2] Apache Solr. . (April 2013). Retrieved April 3, 2013 from http://wiki.apache.org/solr/.
[3] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (HPCA '07). IEEE Computer Society, Washington, DC, USA, 13-24.DOI=10.1109/HPCA.2007.346181 http://dx.doi.org/10.1109/HPCA.2007.346181
[4] Creating Vectors from Text. (February 2011) Retrieved May 4, 2013 from https://cwiki.apache.org/MAHOUT/creating-vectors-from-text.html
[5] Sequence Files in Mahout. (November 2012) Retrieved May 4, 2013 from http://codeochronicles.blogspot.com/2012/11/sequence-files-in-hadoop.html
[6] Latent Dirichlet Allocation. (February 2011) Retrieved May 4, 2013 from https://cwiki.apache.org/MAHOUT/latent-dirichlet-allocation.html
[7] LDA: Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *The Journal of machine Learning research*, 3 (2003): 993-1022.
[8] tf-idf. (May 2013) Retrieved May 4, 2013 from http://en.wikipedia.org/wiki/Tf%E2%80%93idf
[9] Tweepy. Retrieved April 3, 2013 from http://tweepy.github.io/

[10] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y. Chang, PLDA: *Parallel Latent Dirichlet Allocation for Large-Scale Applications. in Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management (AAIM '09)*, Andrew V. Goldberg and Yunhong Zhou (Eds.), Springer-Verlag, Berlin, Heidelberg, 301-314. DOI=10.1007/978-3-642-02158-9_26 http://dx.doi.org/10.1007/978-3-642-02158-9_26