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 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. Firstly, one may wish to know what the prevalent topics contained within the tweets are, and secondly, 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 our method of storage solves. 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

To be added later.

3. SYSTEM ARCHITECTURE

3.1 Twitter Data Acquisition

In order to acquire a large dataset from twitter, the public API to the service must be used to pull the data efficiently to make a local copy. The Twitter public API comes in two forms: 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. While these

RIT 13 Month 3, 2013, Rochester, NY, USA.

tweets are not selected from a distribution, and are instead pulled in terms of relevancy or temporal information, Twitter imposes limits to the number of requests that can be handled by any particular IP address. The new version of this API limits 180 queries per IP address, but should the developer have a developer account (publicly available), and authenticate using the Twitter O.Auth API, this 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 to 15 minutes.

Because of these rate limits, even with the authenticated 1800 queries per hour, each query only results in 100 resulting tweets "per page". Each page counts as a separate request. The 1% "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. It should be noted that all streaming endpoints have access to the same 1% sample.

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. Furthermore, there is a public python package Tweepy which can be used to interface with the Twitter stream API. This made the task much simpler to get started. Below is a snippet of the script we used for acquiring the tweets.

The scripts used were run in a mounted S3 bucket on an EC2 instance.

Below is a snipper from the data acquisition script.

```
l = StdOutListener("bda.dump")
auth = OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

stream = Stream(auth, l)
try:
    stream.sample()
except Exception:
    print "Done scraping, cleaning..."
    l.myfile.close()
```

3.2 Amazon Web Services Configuration

3.2.1 EC2

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 storage of streamed twitter data. The initial setup provided access to the root user, and allowed for the installation of python, thus also allowing for the data acquisition script to run without problem. 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 2gb of memory to be allocated to the JRE heapspace. Mahout's core unit tests failed on this environment.

We had to therefore switch and re-configure a larger instance (m1.large) 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 8gb of memory - enough to walk through the installation of Mahout and not fail on any of the core unit tests.

Once this instance was setup, the S3 bucket to be used for file storage 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 were unable to get proper per-user permissions to work with S3FS.

3.2.2 Hadoop, Maven and Mahout

The installation and running Mahout on Ubuntu 12.0.04 first required the installation of Maven, and Hadoop. 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 Hadoop version 0.20.2. In our attempts to set up Mahout for we ran into issues with the build process.

At first we installed the open JDK (Java Development Kit) 6, JRE Headless, and JRE lib. We updated the apt-get application, and following that we installed tools needed to run Hadoop and Mahout, the tools we installed were: python-setuptools, simplejson=2.0.9, boto=1.8d, ant, subversion, maven 2. 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 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 come in the core-site.xml and also mapred-site.xml file

```
<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 out .profile file which is located in out 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
export MAHOUT_HOME=/usr/local/mahout-0.4/
export MAHOUT_VERSION=0.4-SNAPSHOT
export MAVEN_OPTS=-Xmx1024m
```

Everything worked up to here, in this section we will explain the troubles we went through of installing Mahout.
We check out the Mahout repository using svn from the apache website http://svn.apache.org/repos/asf/mahout/trunk 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.

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

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, at first 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 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. Because Hadoop requires SSH access to manage the nodes, even though we are running locally (single node), the hdfs user need to have ssh access to the machine(localhost). We logged in as hdfs user, generated the keys, and added the keys to .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 seemingless, I 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.

3.3 Lucene Index Creation

3.3.1 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. 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. SOLR is solution that indexes the content of text from PDFs, HTML, Word, and Open Document formats. As we group 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 groups 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.

Shortly prior to concluding our work thus far, we discovered that there is an alternate process to making these vectors from plain files in a directory structure. We will try this next. Overall, much of our work resulting in a - difficult - learning process on the AWS services, and handling various integration issues that were not anticipated.

3.3.2 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. After the sequence files are created we used seq2sparse to generate the vectors to be used in Mahout's LDA. Mahout's implementation of LDA requires TF weighted vectors to complete. To do this we executed the following commands:

```
Seqdirectory –input <input location> -output <location> -c UTF-8 –chunk 64
seq2sparse – I <input> -o <output> -wt tf –minSupport -10 –minDF 5 –maxDFPercent 99
```

Mahout's implementation of LDA requires that sparse vectors are indexed with numerical values. This required the conversion of our TF vector file to rowids. To do this we utilized the Mahout rowid command shown below.

mahout rowid –I text_vec/tf_vectors –o sparse-vectors-cvb
From this output we were then will much trial and error able to execute Mahout's LDA algorithm against the data set. Our initial paremeters yielded results in a pseudo distributed mode after approxiamately 6 hours of execution. Our executed command is listed below.

mahout cvb -i /root/twitter_files/vectors/sparse_vectors_cvb/ -dict /root/twitter_files/vectors/dictionary.file-0 -o /root/twitter_files/topic_term_dist -dt /root/twitter_files/doc_topic_dist -k 30 -nt 400000 -mt /root/twitter_files/state -x 3 -mipd 10 -a 3 -e 3

Next steps for Phase 3 here are to evaluate the selection of the parameters for the LDA command and apply this to the remainder of the data set.

3.4 Phase 2 Next Steps

1. Establish a working Mahout build on an AWS image. This image will no longer be running Ubuntu, as another person in the class has recently gotten Mahout to compile and run on a CentOS distribution. He has not yet made the AMI public at the time of writing this paper.

Phase 2 update:
Since the phase 1 submission, this AMI has been made public, allowing us to use it as a working base model. The EC2 instance still needed various other packages installed for S3FS to work, and for our python-based processes to run. Once these issues were handled, the CentOS-based platform could be leveraged as expected.

2. Build and generate a Lucene Index against the collected twitter data set using the alternate method. We may be required to turn the twitter data into a structured format to do this.

Phase 2 update:
Since the phase 1 submission, we have decided to use the alternative instructions on building the tf-idf vectors necessary to running Mahout's implementation of LDA, by simply using a series of fragmented JSON files. While the vector generation over this data was successful, there are several problems with this approach. Firstly, fragmenting the data stream chunks is expensive - not in terms of processing, but in terms of data transfer to the S3 bucket. Whereas it is possible to fragment a 100 Megabyte chunk of data in near instant time on the local disk, it takes several hours to do so on the S3 bucket due to the huge amount of file synchronizations that occur (abstracted away by S3FS, but still present).

The reason that this is a large issue is because of the necessity for the vector generation. While generating the feature vectors on a single machine in pseudo-distributed mode is simple on small datasets, the processing time needed to perform this operation likely scales with the volume of fragments, as vectors are generated with respect to other files. Ideally, this problem can be avoiding by sending fragments "in batch" to an S3 bucket. Such approaches will need to be explored during the third phase.

Lastly, the tf-idf vectors are not, by default, usable as parameters in the LDA function, which was solved in the manner detailed in 3.3.2

3. Perform LDA and general MapReduce tasks against the twitter data set to determine the most prevalent topics and generate interesting and actionable information against the twitter data set.

4. REFERENCES

[1] Apache Maven Project. (April 2013). Retrieved April 3, 2012 from Maven.apache.org/what-is-maven.html.

[2] Apache Solr. . (April 2013). Retrieved April 3, 2012 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