Linguistic Analysis of Twitter

New Mexico

Supercomputing Challenge

Final Report

April 2, 2014

Team 63

Los Alamos High School

Team Members:

Sudeep Dasari

Colin Redman

Mentors:

Dr. Reid Priedhorsky, Los Alamos National Laboratory

Venkat Dasari, Los Alamos National Laboratory

Teacher:

Lee Goodwin

Contents

Executive Summary

Twitter is a social networking site that allows millions of users from all over the world to connect and share ideas. A tweet consists of, at most, 140 characters and as much available information as possible. Given its popularity, Twitter offers an immense wealth of information. First attempts to mine information from Twitter focused on developing machine learning algorithms to identify trending topics. As a result, many popular websites continuously update and publish Twitter trends. More recently, starting in 2011, there began a vigorous research effort to develop machine learning techniques that can not only identify trending topics, but also the location where the topic is popular. Information regarding location could help the Center for Disease Control (CDC) understand how a particular disease is spreading throughout the country, and then develop effective policy to combat it. Unfortunately, determining the location where a tweet came from is not as easy as it sounds, because the vast majority of tweets do not contain any specific geographic information. However, one can use trends in language established by the few tweets which do contain adequate location information to estimate where other tweets originated from. Returning to the prior CDC example, if one could expose where key words, such as “flu”, are most likely to occur, then a distribution of the most likely locations where the phrase “I have the flu” occurred can be determined. Initial investigations explored heuristic methods to estimate locations of the tweets based on the vocabulary and linguistics. Very recently, our mentor Dr. Reid Preidhorksy of Los Alamos National Laboratory (LANL) used machine learning techniques based on geographic Gaussian mixture models to expose location trends and estimate the location of tweets.

Our research utilized some of his mathematical framework and then expanded upon it. While we also used geographic Gaussian Mixture Models (GMM), we devised a different and unique methodology to adjoin trends of multiple words (n-grams), which we refer to as Boolean operations. Through the use of Boolean operations on GMMs one can improve both the computational efficiency and the accuracy of the results. Using this method we have identified several trends (e.g., Avian-flu in Southeast Asia) that would otherwise have gone undiscovered. Similarly, whereas Dr. Priedhorsky’s research was performed on vertically integrated computer clusters at LANL, we developed and deployed methods especially suitable for distributed on demand computing platforms such as Amazon clusters. We integrated a suite of tools – some developed by us and others from open libraries – that are capable of performing machine learning operations on Twitter information including; interfacing with Twitter to access data, pre-processing tweets to screen out extraneous characters, fitting GMM distributions to the data, and finally visualizing the results. To accomplish these tasks, the code was built atop the MapReduce framework which minimized compute time. Data regarding how execution time dropped as the code was scaled up and down was also collected as part of this project. The final product allows users without access to advanced hardware, such as a cluster, to still be able to analyze Twitter with our methodology, because the computationally intensive tasks can be executed easily on Amazon cloud, and the remaining tasks can be completed with hard ware present in most homes.

Once the code was developed and tested, we used it to analyze Twitter data collected over ten days, and then ran numerous test cases. Some of the results identified trends that were not anticipated, but were confirmed after the fact through news clippings. We recognize that any location estimates will have uncertainty (or error-bars), therefore, a large part of our effort is to quantify the uncertainty in our prediction. We discovered cases where our techniques can overestimate a trend.

Overall, we found that our methodology can be used to identify what people are tweeting about, and where they are when they tweet about it. At the same time our method was capable of predicting where a phrase would most likely be tweeted from, along with showing how trends on Twitter change over time. Finally, all results of our analysis were overlaid atop maps to help people easily understand their implications.

We recommend that our methods be further verified using a larger set of tweets collected over a longer duration. Given the cost and time implications of such an effort, we did not attempt such a large scale verification.

Introduction

Twitter and Tweets

Social media continues to gain prominence in the manner in which people communicate with each other and how people respond to local or global events ranging from earth-quakes to politics[1]. *Twitter* is one such social network in which users communicate through short (140 character) messages known as '*Tweets*'. Typically, Tweets discuss issues facing the user in the present, and thus trends on Twitter change rapidly. As stated by Mathiodakis and Koudas, "at the announcement of Michael Jackson's death on June 25, 2009 Twitter was immediately flooded with an enormous volume of related commentary"[8]. Numerous such examples exist, the latest being tweets from the 2014 Oscar Awards (we have analyzed this instance later in our report). Often trending topics not only change rapidly, but are at times completely unexpected, and thus can be issues of interest for a multitude of professions. The important role Twitter plays in our society can be gauged by the simple fact that numerous organizations pay nearly $400, 000 each per year to purchase 50% of the tweets. Marketing firms are exploiting Twitter data for product testing while politicians sound out their policy initiatives on Twitter. While many of these applications only need qualitative information (like or not-like a product or policy), other applications seek much more precise information along with error-bounds. For example, the Centers for Disease Control (CDC) and other public health organizations are attempting to mine tweets to locate and isolate a communicable disease breakout before it becomes a crisis; we discuss this further in our report. For such application, tying the changing trends of tweets to a location offers a wealth of information that could save lives and money, or at the very least help companies and governments better allocate time and resources. Our project developed and demonstrated a rigorous probabilistic approach to correlating Twitter trends to a location or a region and for the first time established that common probability Boolean set operators such as AND and OR could be used effectively to improve 'signal-to-noise ratio' of key trends.

In order to tie these trends to a location we needed to collect a wealth of data from Twitter. The base unit of information on Twitter is the 140-character tweet through which users communicate. The text of a tweet contains an arbitrary number of words (limited by 140 characters) which form a unique message mathematically represented as t_i = **("**$w_1 \oplus w_2 \oplus \ldots\ldots \oplus w_m$**"),** where, t_i is 'i^{th} tweet' *string* that is m-words long. The operator "⊕" is used to represent the fact that these words may be separated by empty spaces, punctuation marks or other characters. When a tweet is posted Twitter records the text of the message, along with other associated information such as; the user I.D. of the tweet's creator, the tweet's unique I.D., the language in which the tweet was made, the time when the tweet was sent, and of course (if available) a geo-tag which contains the longitude-latitude coordinates of the tweet's origin. Thus each 'raw' tweet is presented in Twitter's JSON format with as much data as is available. Interested analysts can download up to 1% of the raw tweets from Twitter without charge and use them in their research. Our analysis relied on this random sample of raw tweets collected over a period of few weeks.

Parsing and Classification of Tweets

Twitter does not subdivide the message any further before posting it on the web, but for our purposes – and those of numerous applications – it is necessary to parse the tweet message "string" into its constituent words (tokens) or phrases (N-grams). Parsing transforms a tweet from the string format into an array or a list, $t_i = \{w_1, w_2, \ldots..w_m\}$[1]. Together with location and time information, this list forms a superset $\Re \equiv \{(t_i, x_i, \text{time})\}$, where $x_i = (L_x, L_Y)$ is the latitude and longitude of its origin. Information contained in this super set $\Re$ can be mined *as is* to perform a multitude of analyses ranging from temporal trending to message classifications. Temporal trending refers to the process by which tweets on a particular topic are aggregated to gauge public interest in that topic or event (e.g., 2014 Oscars awards). Because this type of trending analyses aggregate geo-graphical data they can often lead to erroneous trends. One common example is how nation-wide trending of words "cold" and "flu" could be influenced by Bieber concert ('Bieber flu' or 'Bieb-flu' or 'B-flu') or a cold storm (such as the 2014 Atlanta Ice Storm). In such occasions knowledge of geographical distribution of each word or phrase (N-Gram) – if available – can be used to filter out the noise from the signal. If such information is available, then all tweets containing phase "Bieber flu" can be automatically subtracted out from the tally to improve the accuracy of true flu breakout. Not surprisingly developing machine learning techniques capable of aiding in such 'on-the-fly' analyses and implementing them on distributed computing architecture is being pursued vigorously by several researchers including Davis et al[3], Eisenstein et al[4], Chang et al[5], Cheng et al[6], Chandra et al[7]. Unlike these investigators, we seek a mathematically rigorous approach advocated by Priedhorsky et al[8]. Priedhorsky et al keep track of not only the number of times a particular phrase or topic was tweeted but also where each tweet containing that phrase originated. Using this data, they correlated the number density of key phrases, referred to by them as N-grams, using probabilistic geographic distributions.

To develop probabilistic distributions, we must accomplish two tasks. First we must subdivide the entire ensemble of tweets, $\Re$, into two subsets: a subset, $\Re^+$, that contains all the geo-tagged tweets leaving behind all the tweets whose location is unknown in subset $\{\Re - \Re^+\}$. The second task is to parse each of the tweets in $\Re^+$ into its constituent N-grams. For example, if the original tweet message is "The quick brown fox jumped over the lazy dog." the tweet can be broken into 9 1-grams ("the", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog") and eight 2-grams ("the quick", "quick brown", "brown fox", "fox jumped", "jumped over", "over the", "the lazy", "lazy dog") on and on until the 9-gram. Each of the above n-grams would inherit the time and location fields of the original message, that is, now we can assemble a unique set of key-value pairs for each n-gram. By breaking the message into n-grams we are able to apply machine learning to specific phrases and trends rather than being forced to remain at the

[1] Consistent with traditions of computer science, we chose '{' to represent a set. Therefore this should be read as a set of words w_1 through w_m. Key here is that parsing transforms a string into a set of words that are related to each other and to the other associated information.

much more rigid, and less useful, message level. Furthermore, our analysis depends on multiple points being collected, and for obvious reasons it is very unlikely that an exact message would occur more than once on Twitter, whereas words and phrases would have multiple occurrences each with its own lat-long coordinates and time stamp. The computational intensity of performing the classification can be illustrated by the fact that in the above example, a simple 44 character tweet, contains 45 N-grams ($1 \leq N \leq 9$). Some one million (on an average 124 character) tweets are posted per day generating possibly tens of millions of key value pairs for analysis. A MapReduce algorithm implemented on top of the Hadoop architecture was used in our study to perform this computationally intensive function. Whereas the mathematics and computer science behind MapReduce have been investigated by companies such as Cloudera and the programmers who made Hadoop[9-11], we focused on computational implementation and optimization of that algorithm on Amazon clusters. We further describe the task of applying Hadoop map-reduce algorithm to twitter analysis in the following chapter, "Problem Solution".

Gaussian Mixture Models for Geographic Density Estimate

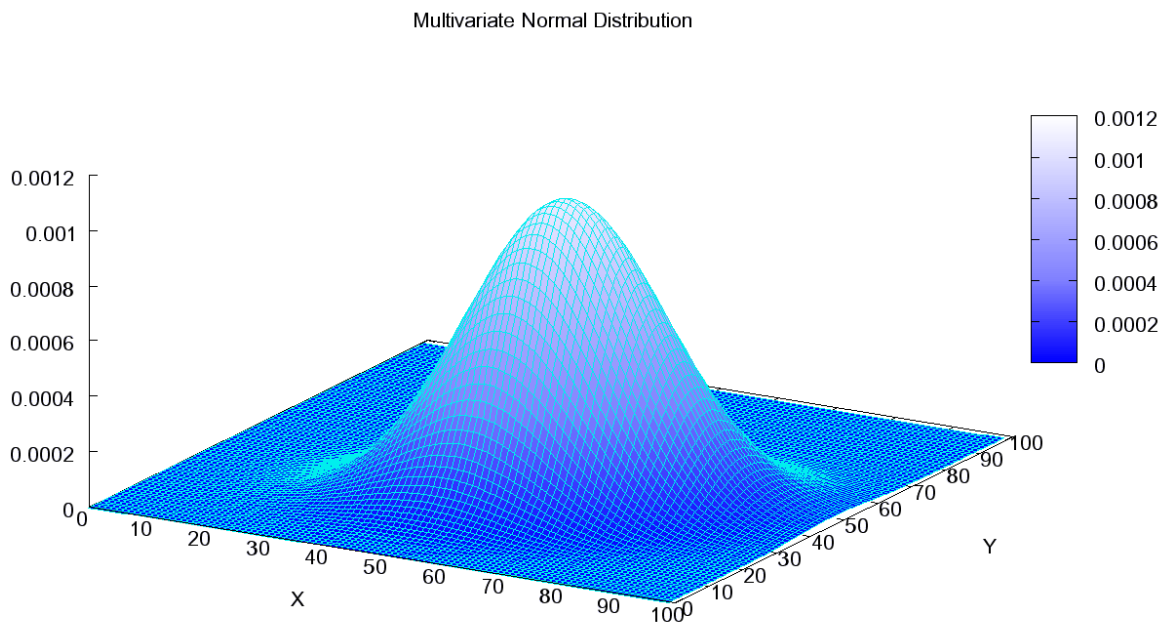


Figure 1: Visualization of arbitrary bivariate Gaussian mixture model [13]

Examining the geographic correlation of n-grams can take two forms: qualitative k-means clustering approach used by Eisenstein et al[4-6] or the rigorous statistical density distribution approach used here. In our approach, a geographic density distribution is expressed as weighted sum of multiple Gaussian distributions. Gaussian (or Normal) distributions, commonly referred to as the Bell curves, are exponential distribution functions used to correlate

stochastic variation of a random variable over its mean. The Multivariate Gaussian distribution function correlates density function of a random variable over multiple parameters; for example, in our case we use a bivariate Gaussian distribution to represent variation of the number density of tweets containing a unique n-gram with latitude and longitude as shown below:

$$G(\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)\Sigma}} e^{-\left\{\frac{1}{2}(x-\mu)^T \Sigma (x-\mu)\right\}}$$

Where, $\mu = \begin{bmatrix} L_x \\ L_y \end{bmatrix}$ is the vector containing mean values for latitude (L_x) and longitude (L_y), and Σ represents the covariance matrix, $\Sigma = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix}$. **Figure 1** illustrates an arbitrarily chosen bivariate Gaussian distribution. It is customary to represent a multivariate Gaussian distribution as $G(\mu,\Sigma)$, where μ is the mean and Σ is the co-variance – which is the nomenclature we have adapted in our report. In some cases, a single Gaussian distribution function may not be able to capture the entire trend. In such cases, a weighted sum of multivariate Gaussian distributions are used: that is, $\mathcal{N}(x) = \sum_{k=1}^{m} \pi_k G_k(\mu_k, \Sigma_k)$, where $\mathcal{N}$ is the mixture function, π_k is the weight assigned to the 'k^{th}' Gausssian distribution (with mean μ_k, and co-variance Σ_k), and m is the number of distributions (or elements) in the mixture. In Figure 2, we demonstrate the power of Gaussian Mixture Models by fitting entire company stock history of a company as a sum of fifty Gaussian distributions. In this figure, the blue dots are the true data and the red dots are the samples drawn from the mixture model. Efficacy of this approach is that in lieu of carrying around the entire stock price data, we only carry 30-40 floating-point numbers to truthfully create the price data distribution. Although not shown here explicitly, the accuracy of predictions (red versus black) improves with the number of elements (or the number of Gaussian distributions used in the mixture model). Whereas Gaussian Mixture Model algorithms have been used in machine learning for over a decade, we adapted a time efficient method for their use by directly incorporating them into MapReduce. One of the advantages of this approach is that a small company can rent on-demand super-computing infrastructure (maintained by companies such as Google and Amazon), process all the data remotely and export back just the model parameters instead of moving large GB size files across the network.

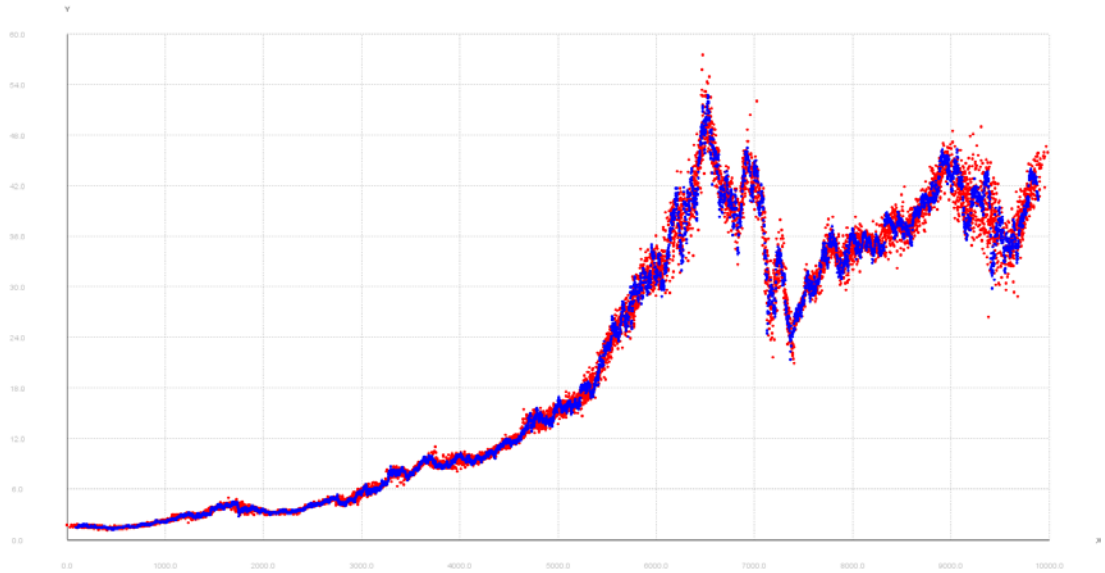


Figure 2: In this figure we demonstrate the use of Gaussian Mixture Model to mathematically capture complex trends. Here we correlated stock price of company using 50 elements.

Boolean Operations to enable on-the-fly trend extraction

Information extracted from the steps described above positions us to perform several additional functions that topical (qualitative classification) approach can't accomplish. First among them is to re-classify (allocate) non-geo-tagged tweets to a location or a region. Priedhorsky et al have demonstrated a statistically robust approach for accomplishing this function. We have implemented a similar approach in our study. In addition, we have explored what we refer to as Boolean operations to mix and match smaller length n-grams to generate probability distributions for a longer string. We explain the merits of the approach by reconsidering the tweet "The quick brown fox jumped over the lazy dog." A different person may have constructed the tweet as "The quick brown fox jumped over my dog that is lazy." The n-gram 'lazy dog' would obviously not be present within the second tweet. On the other hand, sorting based on all tweets containing lazy and dog would encompass both tweets. We extend this logic to probability distributions and postulate that true distribution of lazy dog can be obtained by a product of $\mathcal{N}(x|\text{lazy})$ and $\mathcal{N}(x|\text{dog})$. We verified this postulate by analyzing tweets bearing three tokens Malaysian, Airlines, MH370, and concluded that better estimate of the number density distribution for the Malaysian airline disaster is obtained by multiplication of $\mathcal{N}(x|\text{ Malaysian})$, $\mathcal{N}(x|\text{Airlines})$, and $\mathcal{N}(x|\text{MH370})$ than by any one of the n-grams. If this finding is further validated in the future using a larger tweet set, we believe a new method would be available for the future analysts to mix-and-match tokens to drill through the data. This will also minimize (or perhaps eliminate) the need for analysis using phrases larger than 1-grams. We call this approach "on-the-fly Boolean operations", and have applied it to extract trends not seen before, including swine/avain flu occurrence in Australia and Indonesia.

Real Earth for Displaying Relationships

In the previous section we described how geographic and temporal trends can be accurately correlated using Boolean operations on Gaussian Mixture Models. An important aspect of geographical decision support system is visualization – which is very effective in building trust of decision makers. To that end we have adopted Real Earth Software to seamlessly visualize our results. Once again, to the best of our knowledge, no body prior has integrated a disparate suite of free-ware tools to perform end-to-end analysis of tweets.

Summary and Report outline

Various professions, ranging from policy makers to news reporters, have a clear use for a technology, or a suite of technologies, that can unite the online world of Twitter with the real Earth. This project attempts to address this need by proposing a computational framework that can not only analyze how Twitter trends change over time (which is common), but can also provide a statistically rigorous approach to identify and locate crucial trends that are otherwise masked (or overwhelmed) by the aggregated data. To that end, our focus was to improve spatial, temporal and topical resolution such that each trend, however faint it is, can be exposed and expressed with a defensible confidence bound. To accomplish this objective we have devised a statistically rigorous approach for adjoining and subtracting component distributions that we referred to as the on-the-fly Boolean operations.

Section 2 describes details of the computer implementation of our methodology. This includes discussions of the programming details, challenges encountered and some of our innovations. Computer code itself is attached as Appendix-B.

Section 3 describes the results which provided five different test cases. It should be pointed out that, many of the trends were automatically discovered by our software – with the exception of Malaysian Airlines disaster and Oscars 2014. Validation generally validated our methods, but also identified limitations of our approach –perhaps any approach would have similar pros and cons.

The final two sections present conclusions and a list of original achievements.

Problem Solution

Overview of Methodology

As shown in Table 1, the tasks necessary to create a framework that would fulfill our expectations were broken into four main categories: obtaining tweets from Twitter, Parsing tweets into kev-value pairs using MapReduce algorithm, fitting geographic bivariate Gaussian Mixture Models (GMM) to the geo-location data contained within the tweets, performing Boolean operations on the GMMs to improve the noise to signal ratio of the trends, and visualization of the results. In order to compute GMMs for each token in a timely fashion the MapReduce framework was utilized. MapReduce adds a layer of complexity to the problem which will be elaborated on later in the report. We utilized a combination of Java and Python for most of the implementation of the solution. We describe the solution scheme in the following sections.

Tweet Collection

Tweets were collected from Twitter by interfacing with Twitter Streaming API, a library that allows developers registered with Twitter to collect a random sample of all the tweets made during a time period. Essentially, the streaming API randomly chooses and then funnels 1% of all tweets to developers, as they are being posted on Twitter. While there are greater access levels than just 1%, they require large financial commitment to subscribe to and in some cases require a corporate partnership with Twitter. For our purpose 1% tweets was sufficient to demonstrate the method – which is scalable to analyze the full 100% tweets if available. Rather than working directly with the Twitter Streaming API, we used a python library named *Tweepy* which handles much of the connection and parsing work that we would otherwise have to implement. For example, the laborious task of developing code for connecting to Twitter's servers, authorizing the various connections, and parsing Twitter's JSON format is handled by Tweepy. We developed a script that initializes Tweepy's streaming class and then stores the tweets as they are gathered. In our case, the script stored the tweets in a tab separated value (tsv) format. As shown below in figure three, the tsv format consisted of the tweet's text, latitude, longitude, tweet id, and creation time separated by tabs, with each tweet placed on a new line. To collect the necessary tweets the collection script was run non-stop on a Unix Virtual Private Server, and was also run during events with high Twitter activity, such as the 2014 Oscar Awards. Overall we collected nearly 400, 000+ tweets (50 MB) and used them in our analyses. If we purchased 100% of the tweets this would have amounted to 40 million tweets, and we are confident that our program is scalable to that volume of collection.

Tweet text	Latitude	Longitude	TweetID	Creation Date
Free Text	Decimal	Decimal	64 bit int	Date in ISO 8601 format

Figure 3: The tsv format used to store the processed Tweets collected by Tweepy

Table 1. An overview of our methodology and important innovations

Task	Challenge	Solution
Tweet collection	Software Integration	We used Tweepy, a Python module, to interface with the Twitter Streaming API, and ran it non-stop on a Unix Virtual Private Server. The script stores geo-tagged tweets in the above tsv format as they are collected.
Tweet Parsing	MapReduce	We have used Hadoop implementation of MapReduce on the Amazon network. Whereas other options, such as begging time on Los Alamos cluster, are available our choice is influenced by our vision of developing tools in the open and making them available to other investigators. The Map task handles the parsing work and n-gram creation.
GMM Construction	MapReduce	Use of geo-graphic GMM is very new and we are the second team to have explored use of GMM for Twitter analysis. Many GMM packages are available, both from Python SciPy library and other universities. We used jMEF software package developed by French Polytechnique. In lieu of performing standalone GMM computations on one n-gram at a time, we performed computation on the entire set simultaneously using MapReduce. As such, compute time and the amount of data transfer required was reduced. A second improvement is that we have developed an approach to automatically limit the maximum number of elements to be used – thus eliminating the need for man-in-the-loop.
Trend analysis	Boolean Analysis of Probability distributions	We have devised and demonstrated a unique approach to combining probability distributions of separate N-Grams to estimate joint probability of the larger phrase. This enabled us to mix and match key words on the fly and isolate trends that otherwise could not be identified.
Visualization	Mixture Density maps on GIS	We have adapted QGIS visualization program. QGIS is a free-ware application.
Results		We have developed a software suite that performs lingusitic analysis of Twitter feeds on a cluster. Making use of our 'on-the-fly' Boolean operations we were able to discern several important trends hidden in the Twitter data, trends we believe could have been missed by other software implementations. We have demonstrated an approach to harness the power of social network, by making use of supercomputing on Amazon cluster.

Parsing, Clustering and GMM by MapReduce

Introduction

MapReduce is a computational framework that allows large sets of data to be broken into chunks, which are then processed in parallel across a cluster. By splitting the work across all the nodes in a cluster, MapReduce allows for jobs to complete significantly faster than if they were just run using normal techniques. It is important to note that MapReduce uses a master-slave setup, in which one node in the cluster acts as a master and handles all the necessary coordination work (scheduling jobs etc.) and the remaining nodes do all the processing work assigned to them. As the name may suggest every MapReduce job is split into two separate tasks, Map and Reduce. During execution, input data is "split" across the nodes in a cluster. After each node receives the data it is to process, the Map task is run on the input data and the resulting output is in the form of a key-value (KV) pair. After all the nodes finish their assigned Map tasks the job enters a phase referred to as "shuffle and sort" in which all the KV pairs outputted by the Map tasks with the same key are bundled together and sent to the same node. Finally the nodes conduct a Reduce operation on all the KV pairs they receive with the same key and then output another KV pair(s) which contain the result of the Reduce operation. This final output constitutes the output of the MapReduce job. Once all the nodes finish their Reduce operation the output is saved, any final operations are executed, and the job is terminated.

To gain a better understanding of MapReduce it is useful to examine how a common process, such as a word count, would execute in MapReduce. Figure 4 displays the program flow for a MapReduce word count operation which is counting the instances of words in the Shakespearian phrase "To be or not to be" using a four node cluster. The input phrase is split across the 3 slave nodes which then run the Map task on their segment of input data. In this case, the Map task consists of splitting the input phrase into its component words and then outputting a KV pair for each word in the form of (word, 1). During Shuffle and Sort all the KV pairs with the same key, in this case the same word, are aggregated and sent to the same reducer. The reducers then count the amount of (word, 1) pairs associated with each word and then output the KV pair (word, # times cited). The final output is a list of all the output of the reducers, which in this case is each word followed by the number of times it appeared in the input text. Note that there are only three slave nodes in this cluster, and there are four reduce tasks required, so one cluster will be required to perform the Reduce task twice, once on each of the two separate data sets.

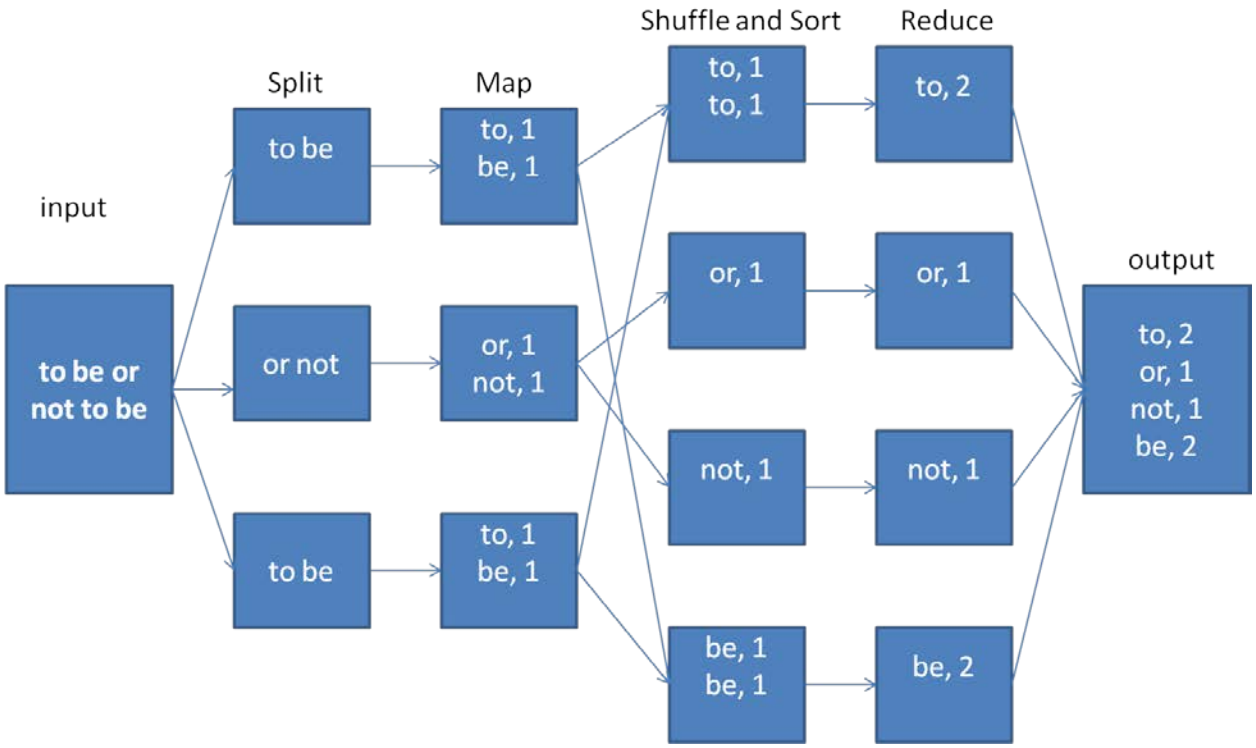


Figure 4: Illustration of the inner workings of MapReduce Algorithm. We used this algorithm to parse the geo-tagged tweets. In our case, the required output is not only the total number tweets with each 'word' (N-Gram), but also the geographic bivariate Gaussian mixture model of the origin of the tweets containing that word.

While there are many implementations of MapReduce, each with their own pros and cons, we chose to use the Hadoop implementation because of its integration in Amazon cloud. In order to take full advantage of the MapReduce architecture, we needed access to a cluster with appreciable hardware, rather than the limited hardware we could procure. Furthermore, it would be an added bonus if the cluster's hardware could be easily changed as certain MapReduce jobs are less intensive than others. Amazon cloud was the logical option because it provides and sets up the necessary hardware for a MapReduce job on demand. Essentially, Amazon cloud took care of the hardware aspect of MapReduce, allowing this project to focus mainly on algorithm development and software integration. The only appreciable difference between Amazon's implementation of Hadoop and the standard version is specific to data storage. In the Amazon implementation data is streamed from Amazon's cloud storage rather than being broken apart and stored on each node's hard drive. The streaming process, or perhaps aspects of how Hadoop utilizes memory, may be responsible for the abnormal affect increasing the amount of cluster nodes had on execution time. More on this topic will be discussed during results analysis.

Our Implementation

For our implementation of MapReduce the input data was the tweets outputted by the tweet collection script described earlier. The output was a list of (n-gram, GMM) values stored in a custom output format. When our MapReduce job was run the first step, as always, was splitting the input data across the "slave" nodes in our cluster. The Map task run on each portion of the

input data converted the tweet messages into n-grams using the process described earlier. The Map task then outputted each n-gram along with the latitude and longitude coordinates associated with the main tweet in the form of (n-gram, <longitude, latitude>). The Reduce task then fit a GMM for each n-gram using the jMEF library, and then outputted the data in the format (n-gram, GMM). The Hadoop library does not have a default method to handle the storing of Mixture Models so we extended Hadoop's 'FileOutputFormat' and 'RecordWriter' classes to handle the export of GMM objects. Each (n-gram, GMM) pair was outputted as a string in the format:

Key:Size:OCC:x1,y1,a1,b1,c1,d1:…><

Where:

Key is the n-gram

Size is the number of elements in the GMM

OCC is the # of occurrences of the n-gram in the input data

x1,y1 are the points which define element 1's mean

a1,b1,c1,d1 are the 2x2 covariance matrix values for element 1

Each element contains the above six values

Each elements is separated by "**:**" hence "…"

Mixture models are separated by "><"

Once all the mixture models were outputted, the result was concatenated into one large string and then stored in a gzip'ed file using Java's ObjectOutputStream and GZipOutputStream. Our custom output format had many advantages over Hadoop's default text output format, as the default format took up much more space than the custom format and took substantially longer to load into other programs for post-processing and visualization.

Post-Processing

After GMM models were generated by the MapReduce code, a post-processing script conducted result analysis and outputted the findings in a coma separated value (CSV) format for visualization. The program (TweetCombiner.java) allows the user to input a "target phrase" which it tokenizes and breaks apart into n-grams using the same technique already discussed. Then it searches the results of the MapReduce program and pulls all the available mixture models corresponding to n-grams in the target phrase (if an n-gram in the target phrase doesn't have a GMM associated with it is ignored). The program then generates a density map for each n-gram, by finding the probability the n-gram occurred at every whole number longitude and latitude combination. For example, for the n-gram "the" a density map would consist of the

probability of the n-gram occurring at each point on a map ranging from a longitude of -180 to 180 and a latitude of -90 to 90 (only whole numbers so no 179.9 etc.). The probability of a non-geotagged phrase occurring at each point on the globe, is calculated by multiplying the density values at each point for each n-gram making up the phrase together. For example, for the phrase "Malaysia airlines mh370" the density map can be calculated by multiplying the density map of the n-gram "malaysia airlines"[2] with the map of the n-gram "mh370". If there exists a situation where an n-gram, say "malaysia airlines", contains two or more n-grams, in this case "malaysia" and "airlines", the singular n-grams will not be included in the final phrase in favor of the larger n-gram if the larger n-gram has a low enough error. Density maps for the combined phrase and individual n-grams are then outputted in individual CSV files for visualization. Along with generating density maps, the post-processing script can also pull random points from the GMM and then export these points in CSV format so they may be used in the creation of a heatmap. Because density maps give a better visual understanding of the data, they are preferred over the use of heatmaps. The post-processing program can also output other data, such as the amount of times an n-gram occurred, which allows us to, among other things, analyze how the popularity of trends change over time. Soft ware modules for performing the post-processing were developed by us.

Visualization

As our project is centered on synthesizing the computerized world of Twitter with the real Earth, it is only logical that we have a way of overlaying our results on top of a map rather than leaving them as numbers in a spreadsheet. In order to achieve this goal we utilized the geographic visualization program QGIS to handle the bulk of our visualization work. QGIS was able to overlay our density maps atop a bare bone map of Earth and its countries obtained from Natural Earth. Furthermore, when visualizing heatmap data we used the QGIS heatmap plugin to create a heatmap of the coordinates exported by the post-processing program, and then overlaid that heatmap atop the same bare bone map previously described. To make most of the other visuals seen in this report, such as graphs and flow charts, we stuck to Microsoft Excel and PowerPoint.

Results

Preface

For all the following results, except for the data regarding MapReduce's scaling, QGIS was used to overlay points on the map, which were colored using the pretty break graduated coloring scheme. Pretty break was used in favor of other schemes, such as standard deviation, because it did the best job of emphasizing the high probability areas while eliminating the "background noise" associated with the GMMs. For further reference, all GMM's that we used in result analysis are included in Appendix A of this report. The only exception will be during

[2] Malaysia is not capitalized because during n-gram creation all characters are switched from upper to lower case.

the “earthquake test case” in which the GMMs will be overlaid on the visualizations and included in the appendix. Shown below in figure five, is an example visualization of a univariate GMM. One can imagine the results in the following sections as bivariate versions of the following visualization projected on a map (figure 1 was bivariate). The tweets that were used to create this data set were collected on the day of February 23rd, the afternoon and night of February 24th (before, during, and slightly after the Oscar ceremony), and during the week of March 17th through the 22nd.

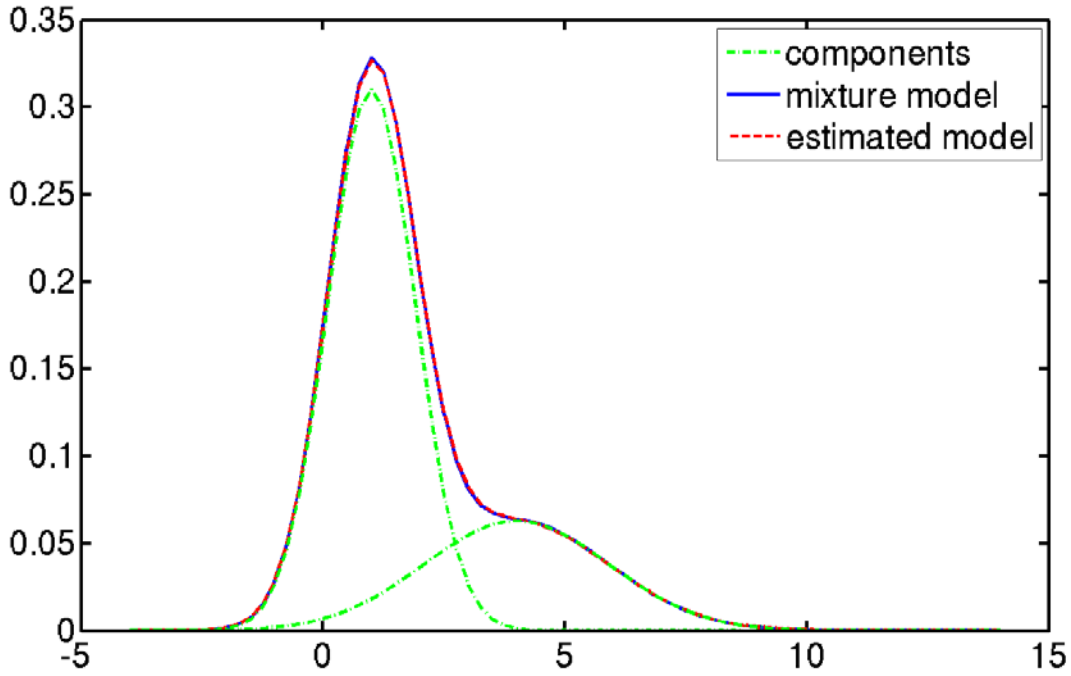


Figure 5: Graphical representation of a Gaussian mixture mode, and the elements that combine to form it [14]

MapReduce Scaling

The first area of analysis focused on how execution time decreases as the number of nodes in our MapReduce cluster increases. In other words, we were investigating how MapReduce scales. One would expect that the execution time for a job would decrease linearly with the number of nodes in a cluster. To quote a report by the cloud company Cloudera, “We expect embarrassingly parallel Hadoop MapReduce applications will scale linearly and our Node Scalability results align very well with this expectation"[3]. As such we also expected that increasing the nodes in a cluster would linearly decrease execution time. However, as shown below in figure six, increasing the number of nodes in a cluster did not drop execution time linearly. Rather at first the amount of clusters decreased execution time rapidly, and after a certain point (in our case five nodes) the graph displayed diminishing returns to scale. It is possible that MapReduce scales linearly past the 5 clusters point, but not enough data has been collected to draw such a conclusion. Such abnormality could possibly be explained by how

Amazon streams data rather than storing on the node's hard drive, how Hadoop handles memory, or perhaps could be related to some aspect of our code, such as the data compression. It is important to note that all nodes used to create the above graph were of Amazon's m1.medium type, and that the m1.large node type also displayed such non-linear behavior.

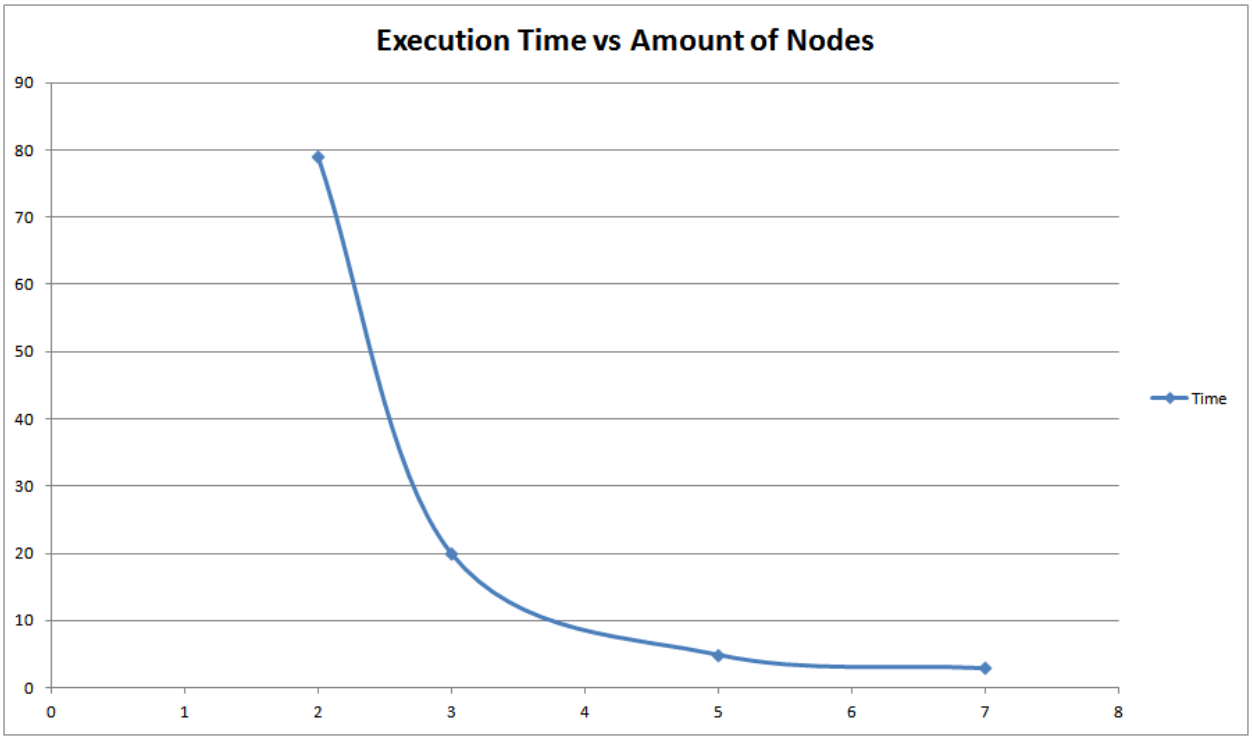


Figure 6: We plot here execution time as a function of the number of nodes in a cluster. For our data set, execution time decreased from 80 minutes to just over two minutes as the number of nodes increased from 2 to 7.

Oscars Test Case

Analysis of tweets collected during the 2014 Oscar Awards ceremony reveals that our method of combining n-grams using Boolean operations can yield accurate predictions. Shown below in figure 7, are the probability distributions for the n-gram "frozen won" (top) and the probability distribution obtained by multiplying the probability distributions of the constituent n-grams "frozen" and "won" (bottom)[3]. The results show that the phrase "frozen won" had a less accurate prediction than the combined phrase. While both the combined phrase and real n-gram were centered in the New York area, combined phase only had only 13 instances, whereas the instances of adjoined n-grams "frozen" and "won" had more than 40 occurrences. As a result, the co-variance values for the distribution of "frozen won" ($\Sigma_{xx} = 46.2$, $\Sigma_{yy} = 735.3$) were larger than the alternative. Further details are provided in Appendix-A, based on which one can reasonably

[3] In statistics this is referred to as intersection of sets {frozen} and {won} or p(frozen ∩ won)=p(frozen)*p(won)

conclude that our method of combining n-grams creates valid estimations of the location of a combined phrase. This conclusion is further validated by additional cases shown below.

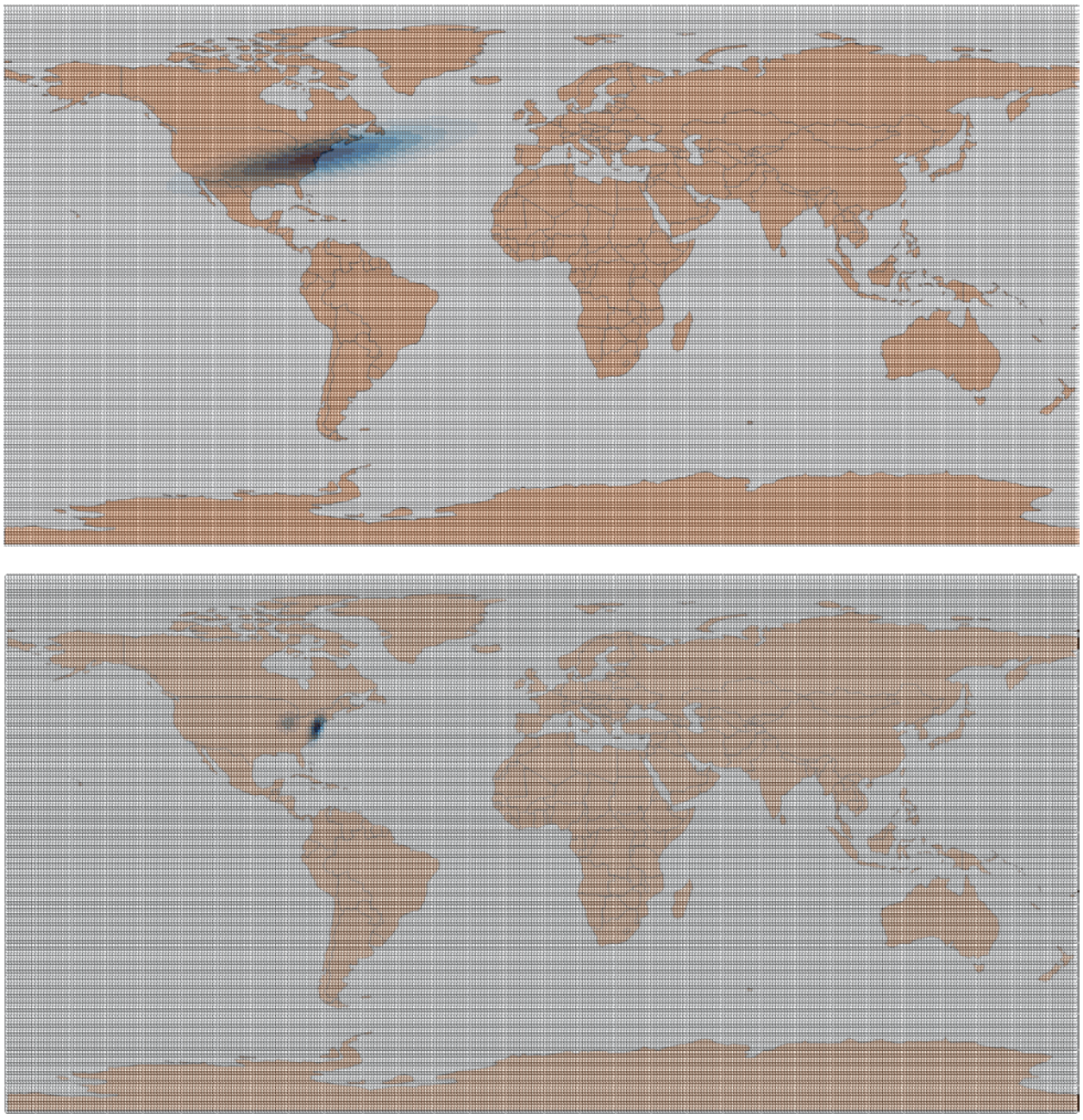
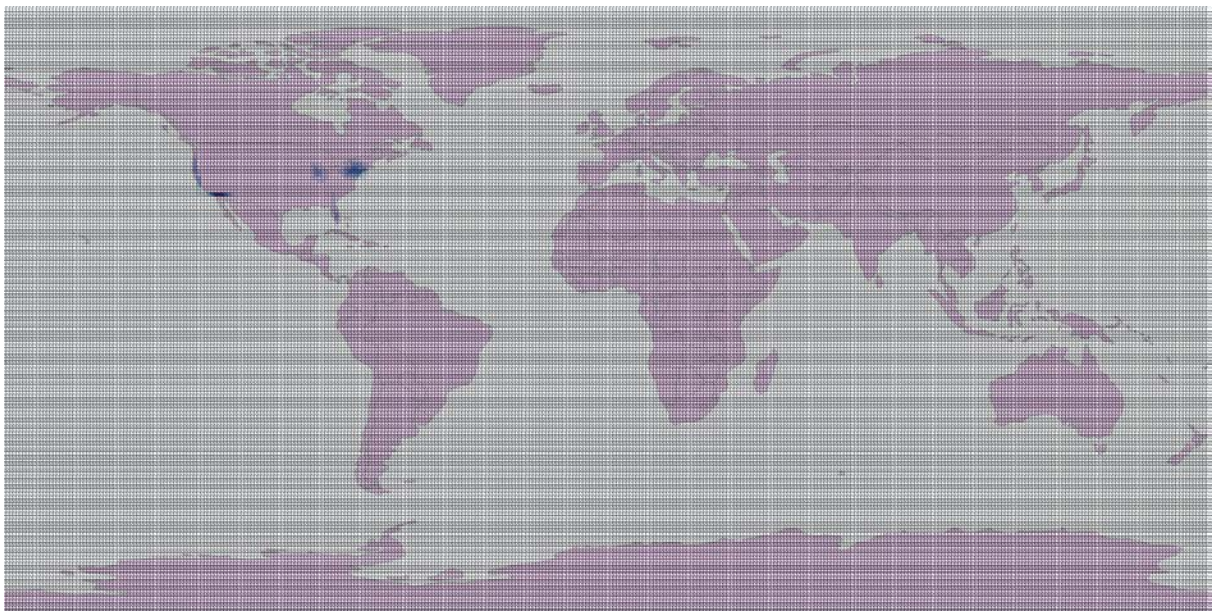


Figure 7: Geographical overlay of tweets related to Frozen winning 2014 Oscar. Top figure illustrates Geographic Probability Distribution of the N-Gram "Frozen Won" which had 13 occurrences. The bottom figure plots the geographic probability distribution constructed by multiplying probability distributions of 1-Grams "Frozen" and "Won".

Another rather surprising result that stemmed from analysis of the Oscar data set was large differences in geography between two similar n-grams. Shown below in figure eight, is the probability distribution for the n-gram “#oscars2014” (top) and the n-gram “#oscars” (bottom). The density distribution for the “#oscars2014” was concentrated in various areas of the country, ranging from New York to Los Angeles, and Florida and the mid-Eastern regions. On the other hand the distribution for the n-gram “#oscars” was concentrated to the New York region and appeared in few, if any, other places. While both n-grams refer to the same event, the 2014 Oscars ceremony in Los Angeles, it is notable that they have very different geographies. This is contrary to what most would expect as similar n-grams referring to the exact same event would theoretically have very similar distributions.



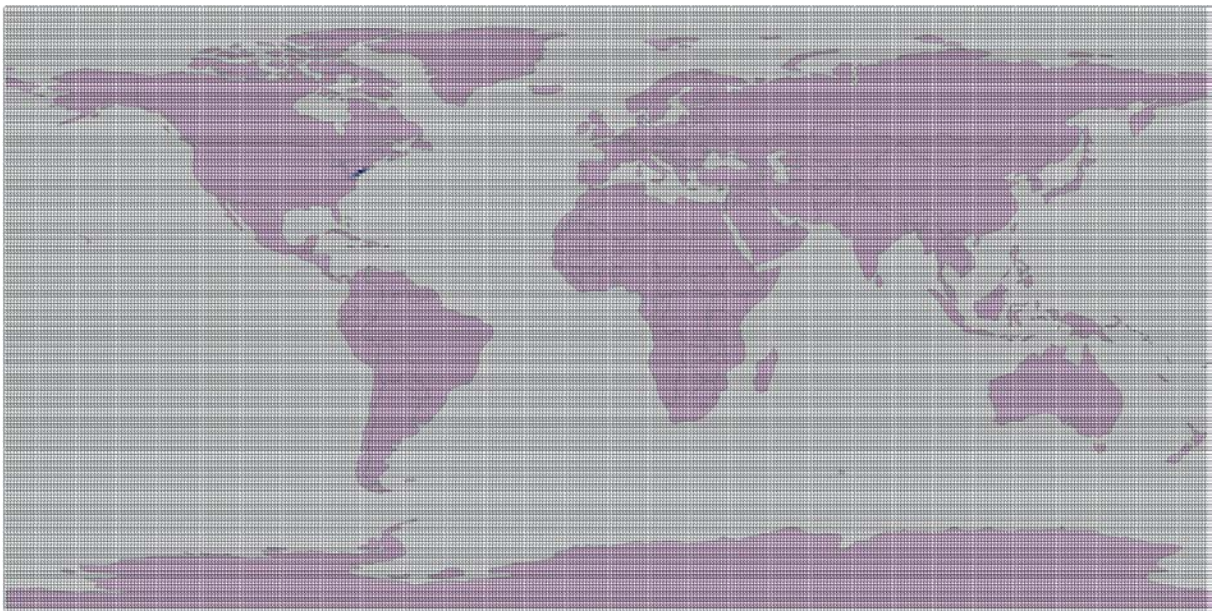


Figure 8: Geographical overlay of tweets containing '#oscars' and '#oscars2014'. Top figure illustrates geographic probability distribution of the N-Gram '#oscars' which is distributed fairly across the country. The bottom figure plots the geographic probability distribution of 'oscar2014'. This illustrates influence of local vocabulary on the trends. Once again using Boolean operation we could add these distributions to obtain the overall distribution of 2014-Oscars.

Malaysia Test Case

We started collecting tweets a week or so after Malaysia Airlines flight 370 disappeared over the Indian Ocean. While collection started too late for analysis of the initial reaction to the planes disappearance to be viable, some traffic presumably regarding the search operation or late reactions were captured. This event, while solemn, allowed an opportunity to test a differnt form of n-gram combination. Rather than multiplying together the density maps of the separate n-grams "malaysia airlines" and "mh370" we added. The change is justifiable because a user could be talking about the incident if the tweet contained either "malaysia airlines" or "mh370" and did not necessarily need to contain both. This relationship is in contrast to the average phrase such as "I have a cold", because the n-gram "cold" needs to be present with a token such as "have" to indicate that a person has a cold rather than a different meaning such as the weather is cold. The probability distribution for the combination of the n-grams "malaysia airlines" and "mh370" is shown below in figure nine. For most part the traffic regarding this incident is concentrated in the Malaysia region, which makes sense considering much of the international traffic regarding the incident would die down a week after the incident occurred. As a note, one would expect more traffic from China as most of the citizens aboard the missing plane were Chinese, but presumably chatter in China would utilize one of the various Chinese languages and thus would not be included in our analysis because of our focus on the English language (We did try to isolate tweets containing Chinese characters for Malaysia but without success). Note that multiplication corresponds to the Boolean AND operator, addition to OR.

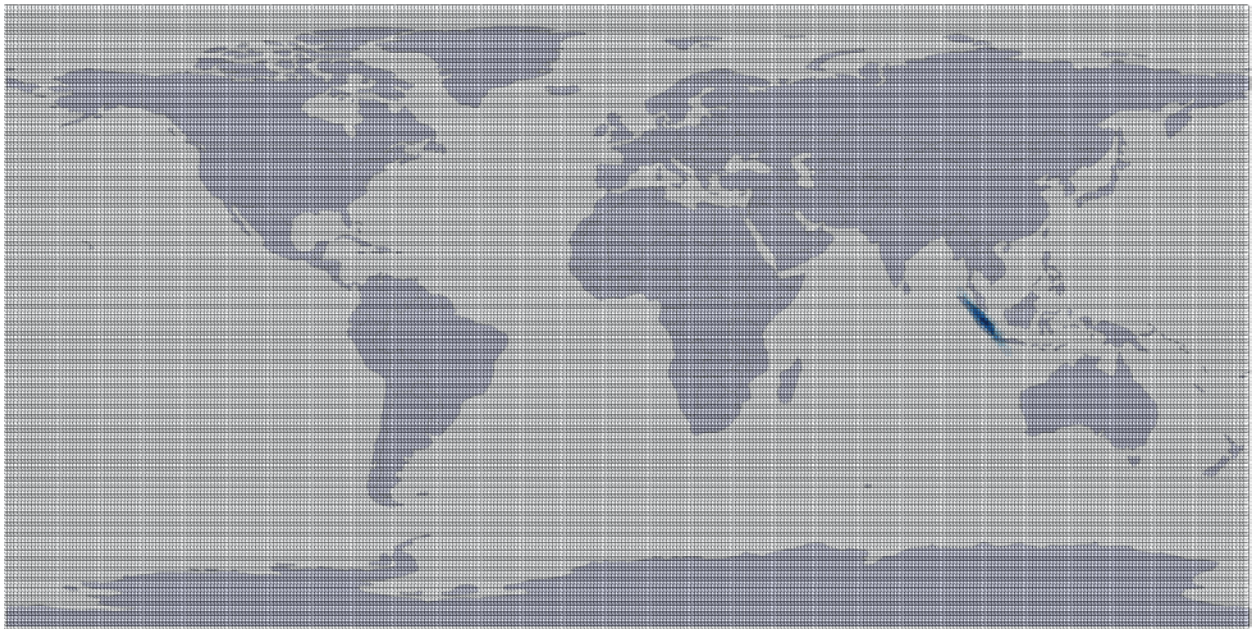


Figure 9: Geographic probability distribution of tweets related to the solemn topic of Malaysian airlines MH370. Approximately a week after the crash, these tweets were mainly centered on Malaysia.

Earthquake Test Case

On March 17th, the day our week long tweet collection began, a 4.4 magnitude earthquake struck the Los Angeles area. Shortly after the earthquake Twitter exploded with tweets containing the n-gram "earthquake", most of which were tagged to the Los Angeles area. Occurrence "earthquake" quickly died down by the next day and remained low until ultimately going to zero by the last day of collection, the 22nd. Figure ten displays a bar graph plot of the number of occurrences of the n-gram "earthquake" on every day from when we started collection (March 17th) to when we ended (March 22nd). Note how on March 17th the number of occurrences of "earthquake" peaks at 138, and then quickly dies down to about five per day, until finally dying out completely by the 22nd. Below in figure 11, are density maps created for the n-gram "earthquake" for each day of collection. The maps are arranged in order of increasing date (March 17th top March 18th is next etc.), and the region for March 17th surrounded with a circle for ease of viewing.

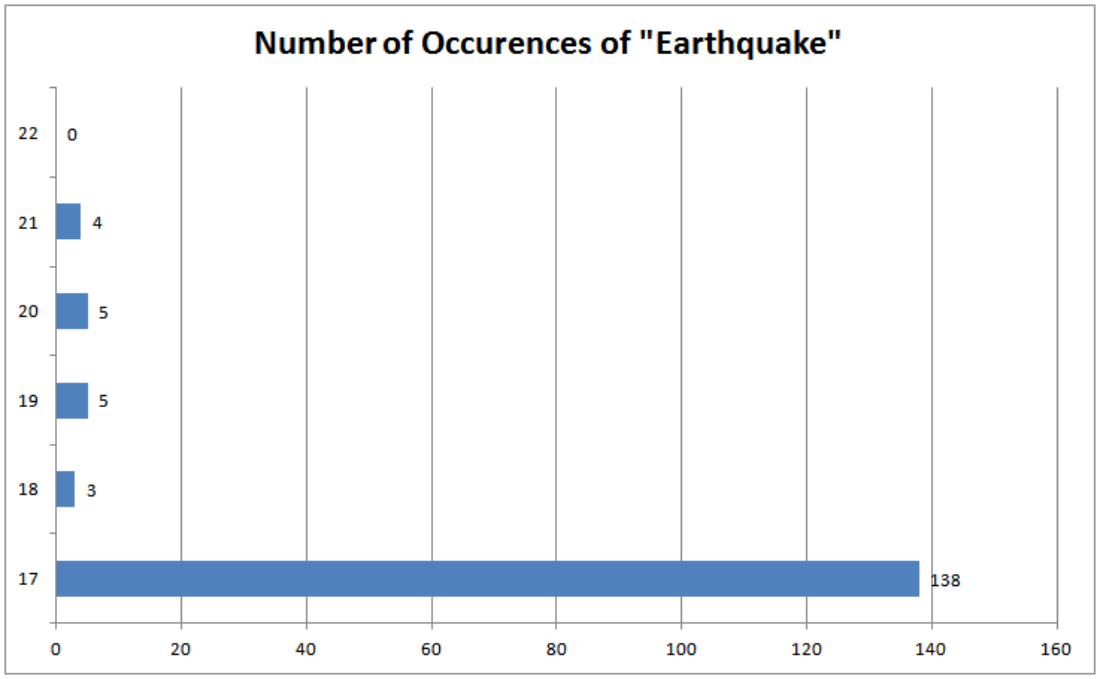


Figure 10: A relatively small earth quake triggered considerable Twitter activity first centered in CA but then diffusing across the globe. This trend was automatically detected by our software (we were not specifically looking for this topic).

Note how on the 17th, the day which had the most occurrences of "earthquake", the mixture model's probability distribution is so precise that it is visualized as a point at the location of Los Angeles, California. On the following days the model's accuracy quickly decreases and center changes as fewer people across the world discuss the earthquake. The accuracy decrease is visualized by the region increasing in size from a tiny point to a massive region. Furthermore, one can see the accuracy decrease quantitatively as the error values in the model's covariance matrix increase day by day. Finally, the 22nd does not have a distribution for "earthquake" as no mentions of the incident were captured. This particular example demonstrates how quickly trends on Twitter rise and fall, and how their geography changes rapidly on a day by day basis. On the 17th if one was to predict where a tweet containing the n-gram "earthquake" came from, one would find that the tweet almost certainly came from the Los Angeles area. However, if the tweet came from later that week no strong conclusion could be drawn. Furthermore, this analysis shows how quickly people react to a natural incident on Twitter, and how the incident leaves their mind and thus stops being tweeted about by literally the next day.

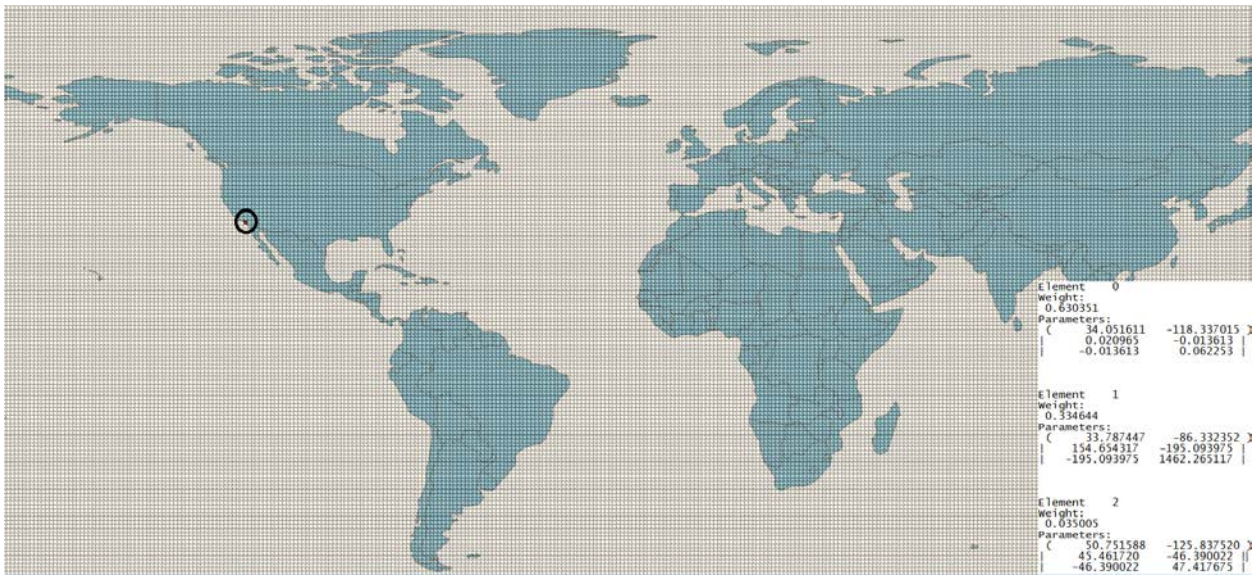


Figure 11 (a): A relatively small earth quake triggered considerable Twitter activity as shown in Figure 10. The geographic distribution is very tight centered on Los Angeles, CA on the first day. A three element GMM captured the trend very accurately.

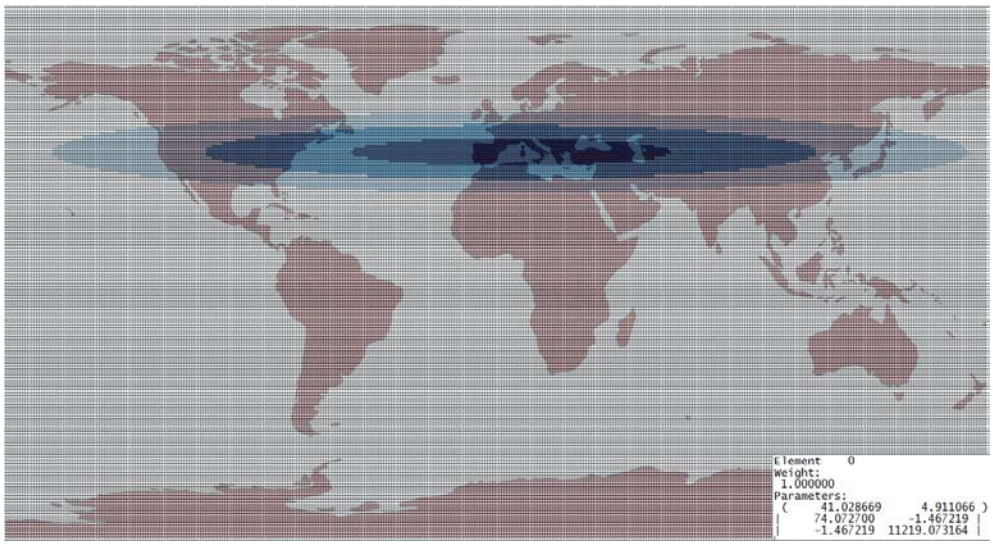


Figure 12 (b): On the second day the twitter traffic on this incident spread widely across the globe. The standard deviation (or variances) have increased to 74 and 11219 illustrating very poor correlation.

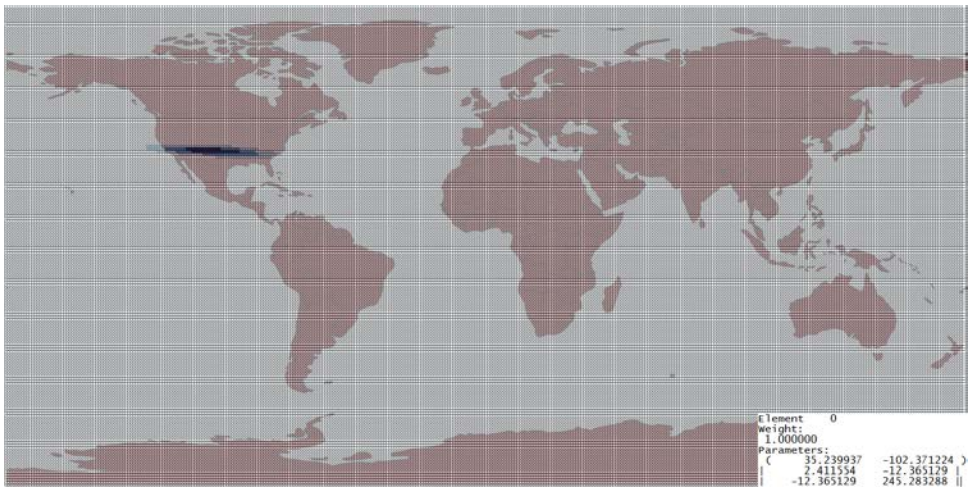


Figure 13 (c): On the third day the geographic mean of the twitter traffic on this returned to US as illustrated by a fairly tight Gaussian distribution.

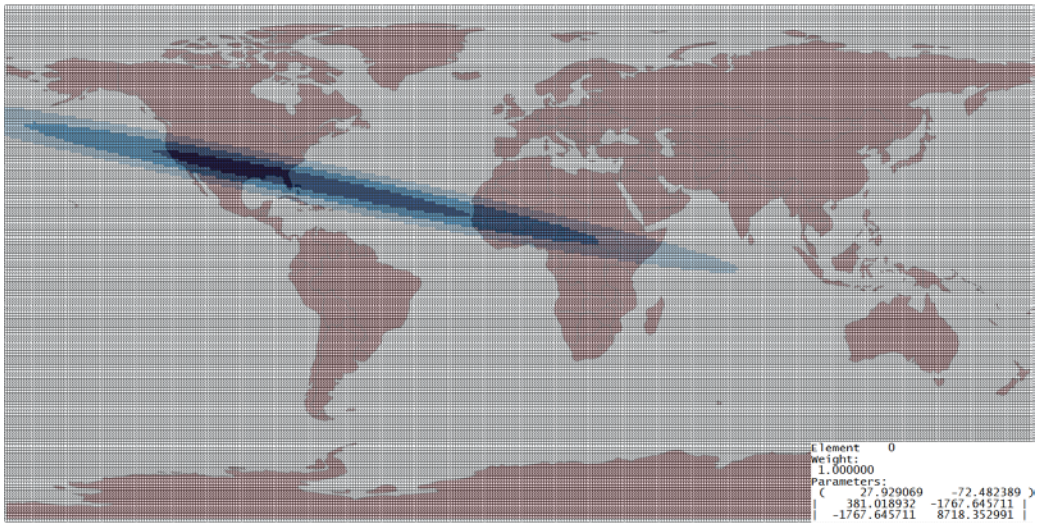


Figure 14 (d): On the fourth day the geographic mean of the twitter traffic still remained within US but the spatial correlation is very poor.

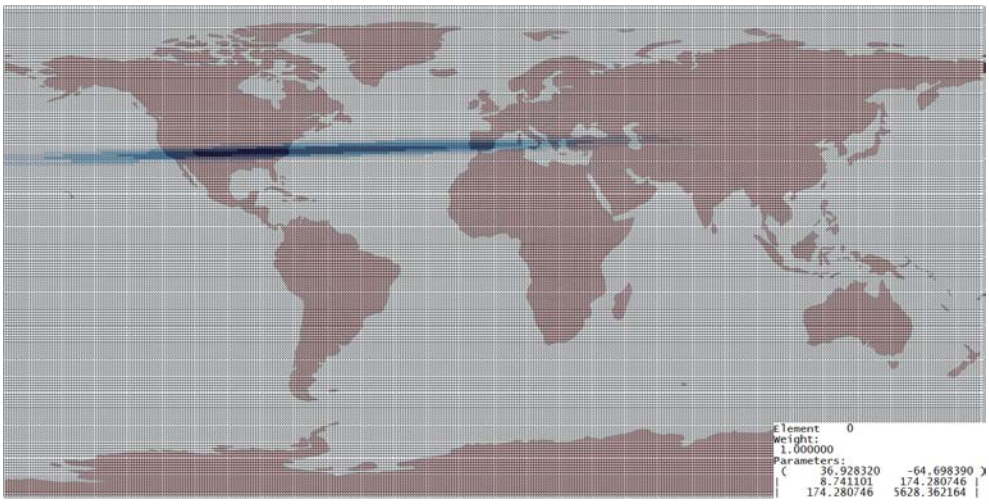


Figure 15 (e): On the fifth day the geographic mean of the twitter traffic still remained within US but the spatial correlation is still very poor.

Flu Test Case

As we have mentioned earlier on, a tool that unites Twitter and geography could be useful for policy makers at institutions like the CDC. As such, we analyzed tweets collected on February 23rd, and through the week March 17th for mentions of the n-grams "flu", "sick", and "cold". Below in figure 12, is the probability distribution for the n-gram "flu" from March 17th through March 22nd.

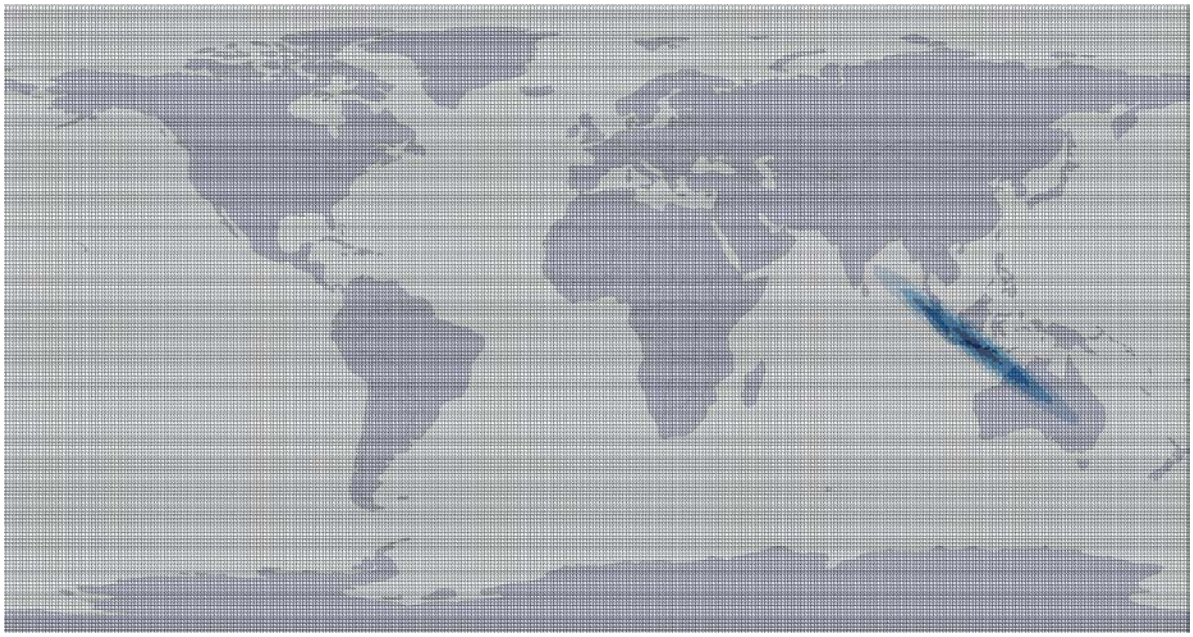


Figure 12: In March of this year several tweets centered on Southeast Asia and Australia were on the topic of flu – more specifically avian and swine flu. The machine learning algorithm, as implemented by us using Boolean logic, discovered this trend which we would have missed using traditional methods. Later search identified several news paper clippings[10-11].

As you can see from the figure the distribution for the n-gram "flu" is concentrated around the countries of Indonesia and Australia. The models prediction of a region near Australia and Indonesia correspond with recent reports indicating an outbreak of swine flu in Australia and New Zealand, and an outbreak of H5N1 avian flu in Cambodia. As stated by *Radio New Zealand News*[10] on March 26th, "Outbreaks of H1N1 swine flu have occurred this summer in Wellington, Geraldine, Canterbury and most recently Hawke's Bay … 'I personally haven't seen this type of activity since 1974 in New Zealand. Australia is reporting a similar phenomena, in Queensland they've had double the number of hospital admissions due to influenza'". Another report published on *The Poultry Site* on March 24th states, "A new outbreak of avian flu started on 18 March in a backyard flock of 526 birds at Kandal in the region of Kampot, which borders Viet Nam. Around 300 of the birds died and the rest have been destroyed" [11]. Together these reports indicate that during the time period we were collecting tweets there were two outbreaks of flu, one in the Cambodia area another in the Australia and New Zealand area. Furthermore, the GMM's generated by the collected tweets show a high probability region for the n-gram "flu" around Australia and Cambodia. As the information provided by the GMN's prediction and news reports complements one another, it is reasonable to conclude that in certain circumstances tying n-grams such as "flu" to location can reveal where disease outbreaks are occurring.

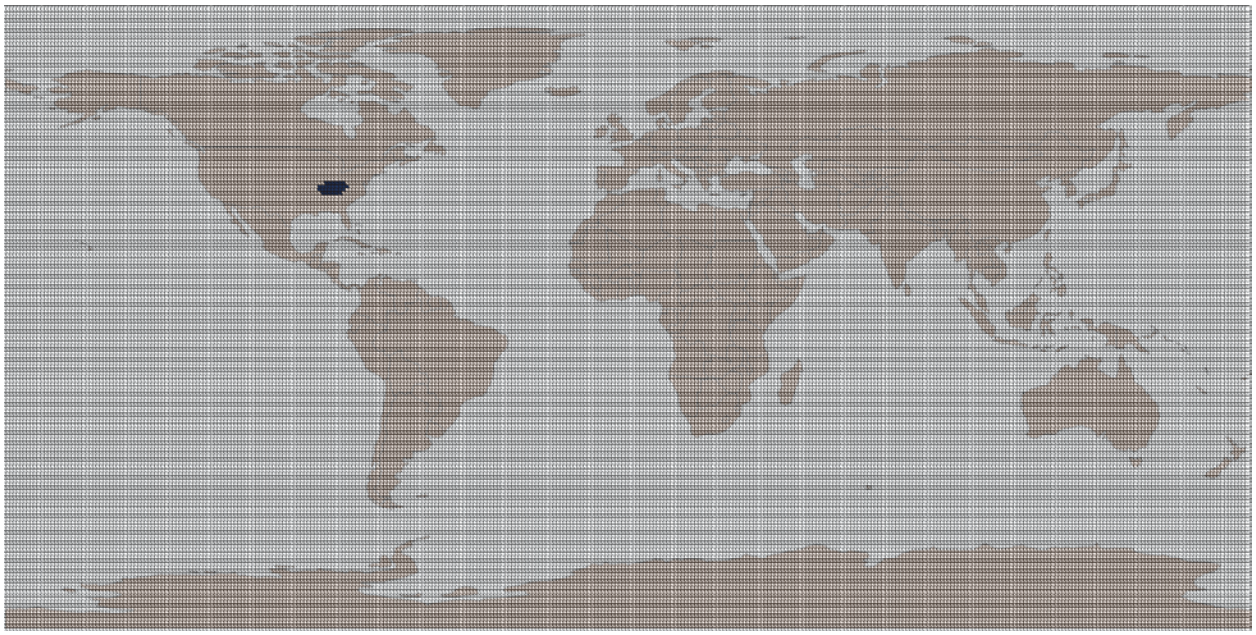


Figure 13: Several tweets centered on the Atlanta region were discovered when searched using Boolean logic cold and sick. No additional data could be used to confirm this trend – either because the trend is erroneous or because CDC database was not accurate. This example illustrates the uncertainty associated with extrapolation of probabilistic methods.

However, just because n-grams such as "sick", "cold", and "flu" are used more frequently in a certain location does not necessarily mean that there is a disease outbreak in that location. Context matters as well. As shown above in figure 13, on February 23rd the combined phrase of

the separate n-grams “sick” and “cold” showed a high probability of occurring in the Southern US, particularly Georgia. But when one looks to data presented on the CDC’s “Weekly US Map: Influenza Summary Update” [12] the region highlighted by our model showed sporadic flu activity. It is more likely that the high incidence region highlighted by the model had less to do with flu activity and more to do with the unusually cold weather that area was receiving during the late February time period. As such, sometimes n-grams that can indicate disease outbreaks, such as “cold” and “sick”, actually indicate something completely different. Overall, there was an instance in which tracking the n-gram “flu” revealed information that correlated with actual reports of flu, while in another case context (specifically weather) messed with the GMM such that faulty results were produced. Thus we have demonstrated that our methodology can generate useful results for policy makers, but further research needs to be conducted to eliminate faulty results.

Conclusions

It is our conclusion that by tying chatter on Twitter to the Earth's geography, using a Gaussian Mixture Model, one can extract and visualize trends that can be useful for various professions. Furthermore, we have demonstrated that it is possible to visualize how topics gain and loose prominence over time, and how they spread over the planet. Also, we have shown that Boolean combination can be used to adjoin phrases with unknown geographic properties on the map and enhance the results of trend analysis. Finally, we have created a framework that allows interested individuals to perform the same analysis we did with ease. Rather than having to implement parts of this process from scratch, or have access to expensive hardware, one can download the various scripts we wrote, deploy them easily on average hardware and Amazon cloud, and then visualize the results with the free QGIS software.

Three areas of further research should be pursued. First the validity of the Boolean logic we developed for result analysis should be verified. Second, more analysis of n-grams such as "flu" should be conducted on larger data sets that span more time in order to conclude for sure whether our method can be useful in applications such as disease prevention policy. Finally, more optimization work on the MapReduce script should be pursued so that the program becomes as resource efficient as possible.

Original Accomplishments

Our yearlong effort has yielded a set of original accomplishments that represent a contribution to the scientific community as a whole. Our application of Boolean logic to this field has resulted in an effective framework for adding together separate Mixture Models. This process has allowed us to extract trends in data that would otherwise remain hidden, and allows us a method for making predictions for data points we have no information on, with the data we have. On a computer science level, our output format allows other developers to use the jMEF mixture model library along with Hadoop MapReduce without having to write their own format or deal with the extremely space inefficient text output that comes default with Hadoop. Our single largest contribution is the overall program architecture we devised to solve our problem. We offer an inexpensive method for individuals without easy access to a cluster or other high power computing hardware to conduct an analysis that requires both. The streamlined tweet collection process we implemented allows users the ability to collect tweets for a long period of time by just executing a single script. Once a user has a sufficient amount of tweets the MapReduce script can easily be executed on Amazon cloud with only a couple keystrokes. Finally, the result analysis program allows users to conduct all the desired post processing on whatever n-grams they have in mind, and then outputs the results in a format easily that can be easily visualized in QGIS. Thus we have created a tool which allows users to analyze large sets of Twitter data for trends, unite these online trends with geography, and visualize the results, without having to wade through the huge time investment making such a tool would normally require.

Acknowledgements

First, we would like to thank the New Mexico Supercomputing Challenge for offering a venue in which projects like this can be proposed and succeed. Next we would like to thank our mentors Reid Preidhorsky and Venkat Dasari who helped us with; understanding the math and computer science behind the project, researching the problem, giving us helpful ideas and insight, and proofreading the final report. Our team would also like to thank our teacher Lee Goodwin who provided encouragement throughout the year, and put time and effort into organizing the club at our school. He is retiring this year and we would like to recognize the work he did as the supercomputing coordinator and physics teacher at Los Alamos High School. We would also like to acknowledge the team of programmers and mathematicians responsible for making jMEF, the library we used to generate GMMs. Furthermore, we would like to thank the maker of Windoop, a Windows implementation of Hadoop, which greatly simplifies that amount of setup work one would normally have to go through to get a working small scale Hadoop environment for debugging and developing. Finally, we would like to thank those who evaluated our project at the High School, and Eleanor Walther who reviewed our interim report.

Discussion of Team

Our team is composed of two junior students from Los Alamos High School. All members have extensive experience with the Supercomputing Challenge. This is Sudeep Dasari's 6^{th} year competing, and Colin Redman's 7^{th}. Prior experience was extraordinarily helpful in planning and executing this year's project. In past years this team has done work with agent based modeling, specifically modeling the flow of goods in shipping facilities, and last year we did work on a mathematical model of tornadoes. This is the first time we have stepped into the field of social media, computational techniques such as MapReduce, and machine learning algorithms. This unique experience was educational, stimulating, and no doubt will give us skills we will use later in life.

Bibliography

1. M. Dredze, How social media will change public health, Intelligent Systems, 27(4)81-84, 2012.
2. C. A. Davis Jr. et al. Inferring the location of Twitter messages based on user relationships. Transactions in GIS, 15(6), 2011.10.1111/j.1467-9671.2011.01297.x.
3. J. Eisenstein et al. A latent variable model for geographic lexical variation. In Proc. Empirical Methods in Natural Language Processing, 2010. http://dl.acm.org/citation.cfm?id=1870782.
4. H. Chang et al. @Phillies tweeting from Philly? Predicting Twitter user locations with spatial word usage. In Proc. Advances in Social Networks Analysis and Mining (ASONAM), 2012.10.1109/ASONAM.2012.29.
5. Z. Cheng et al. You are where you tweet: A content-based approach to geo-locating Twitter users. In Proc. Information and Knowledge Management(CIKM), 2010.10.1145/1871437.1871535.
6. S. Chandra et al. Estimating Twitter user location using social interactions – A content based approach. In Proc. Privacy, Security, Risk and Trust (PASSAT), 2011.10.1109/PASSAT/SocialCom.2011.120
7. E. Cho et al. Friendship and mobility: User movement in location-based social networks. In Proc. Knowledge Discovery and Data Mining (KDD), 2011.10.1145/2020408.2020579.
8. Mathioudakis, Michael, and Nick Koudas. "TwitterMonitor: Trend Detection over the Twitter Stream." *ACM* (n.d.): n. pag. *Google Scholar*. Web. 1 Apr. 2014. <http://scholar.google.com/scholar?hl=en&q=TwitterMonitor%3A+Trend+Detection+over+the+Twitter+Stream&btnG=&as_sdt=1%2C32&as_sdtp=>.
9. Zuanich, Jon. "Cloudera." *Cloudera Developer Blog*. Cloudera, 15 Dec. 2010. Web. 01 Apr. 2014. <http://blog.cloudera.com/blog/2010/12/a-profile-of-hadoop-mapreduce-computing-efficiency-continued/>.
10. Ray, William. "Radio New Zealand." *Radio New Zealand*. Radio New Zealand, n.d. Web. 01 Apr. 2014.<http://www.radionz.co.nz/news/national/239864/cause-of-swine-flu-outbreaks-unclear/>.
11. 11. "Another H5N1 Avian Flu Outbreak in South Cambodia - The Poultry Site." *The Poultry Site*. N.p., n.d. Web. 01 Apr. 2014. <http://www.thepoultrysite.com/poultrynews/31826/another-h5n1-avian-flu-outbreak in-south-cambodia>.

12. "Weekly US Map: Influenza Summary Update." *Centers for Disease Control and Prevention*. Centers for Disease Control and Prevention, 28 Mar. 2014. Web. 31 Mar. 2014. <http://www.cdc.gov/flu/weekly/usmap.htm>.

13. "Multivariate Gaussian.png." *Wikimedia Commons*. Wikimedia, n.d. Web. 1 Apr. 2014. <http://www.mathworks.com/matlabcentral/fileexchange/screenshots/3544/original.jpg>.

14. Roughan, Matthew. "Gaussian_mixture_model.m." *MathWorks*. Matlab, 28 Oct. 2009. Web. 1 Apr. 2014. <http://www.mathworks.com/matlabcentral/fileexchange/24867 gaussianmixturemodelm>.

Appendix A: Mixture Models

The following data are mixture models listed next to the n-grams they describe. Information regarding the format of the GMMs can be found in the results section

Oscars Mixture Models

Model for n-gram “won”

	Weight	Parameter	Covariance Matrix
Element 1	0.283629	(39.216909, -87.719256)	| 11.425847 3.655604 |
			| 3.655604 12.769624 |
Element 2	0.294047	(38.699793, -75.666681)	| 36.253347 12.977894 |
			| 12.977894 7.527747 |
Element 3	0.144542	(34.302502, -112.10022)	| 0.349569 0.933831 |
			| 0.933831 56.665398 |
Element 4	0.277783	(34.899676, 30.599833)	| 854.842090 -1415.973077 |
			| -1415.973077 2948.650812 |

Model for n-gram “frozen”

	Weight	Parameter	Covariance Matrix
Element 1	0.056338	(-10.960299, 124.2924)	| 619.729946 -586.936747 |
			| -586.936747 568.075760 |
Element 2	0.193431	(27.018890, -96.868353)	| 82.745502 -39.370091 |
			| -39.370091 58.811901 |

Element 3	0.531548	(40.417987, -78.899205)	| 22.165259 -10.672431 |
			| -10.672431 42.448704 |
Element 4	0.106007	(40.316249, -120.158919)	| 55.240952 -21.341384 |
			| -21.341384 12.753838 |
Element 5	0.112676	(12.765117, -23.944568)	| 1314.159040 803.046198 |
			| 803.046198 493.140253 |

Model for n-gram "#oscars2014"

	Weight	Parameter	Covariance Matrix
Element 1	0.061986	(22.643633, -98.996661)	| 17.890713 3.687642 |
			| 3.687642 2.615671 |
Element 2	0.242083	(41.200014, -75.546376)	| 3.689556 0.458958 |
			| 0.458958 7.794292 |
Element 3	0.091787	(34.000658, -116.143347)	| 0.304730 -0.573493 |
			| -0.573493 7.958441 |
Element 4	0.038304	(7.328290, -73.616856)	| 10.660829 5.622863 |
			| 5.622863 25.890078 |
Element 5	0.106624	(-27.480502, -67.635179)	| 82.213872 -31.211718 |

			| -31.211718 68.893281 |
Element 6	0.049471	(35.377545, -97.473002)	| 39.107297 2.535514 |
			| 2.535514 7.959359 |
Element 7	0.052921	(29.434532, -81.437114)	| 12.136037 -1.621435 |
			| -1.621435 0.421997 |
Element 8	0.082126	(40.33613, -122.515615)	| 19.791168 -2.622869 |
			| -2.622869 0.616867 |
Element 9	0.149758	(42.700673, 17.200252)	| 237.956092 -572.889895 |
			| -572.889895 1638.299926 |
Element 10	0.124941	(40.619182, -86.416267)	| 4.440867 -0.817139 |
			| -0.817139 3.565035 |

Model for n-gram “oscars”

	Weight	Parameter	Covariance Matrix
Element 1	0.193077	(40.795455, -74.096286)	| 1.066725 1.358281 |
			| 1.358281 2.517885 |
Element 2	0.051947	(28.384751, -81.581887)	| 6.592463 -3.405599 |
			| -3.405599 2.002350 |
Element 3	0.079672	(8.648718, -74.841865)	| 279.209241 -240.489505 |

| -240.489505 299.865231 |

Element 4	0.088229	(30.945984, -96.354602)	| 12.370475 -4.882967 |
			| -4.882967 4.076348 |
Element 5	0.104918	(36.21048, -119.559103)	| 8.908806 -4.310021 |
			| -4.310021 11.456088 |
Element 6	0.223980	(39.76521, -86.346488)	| 8.908806 -4.310021 |
			| -4.310021 11.456088 |
Element 7	0.131762	(41.529338, -78.652696)	| 5.635968 -2.289280 |
			| -2.289280 1.817384 |
Element 8	0.094613	(32.324833 , 39.032817)	| 415.170287 -624.463151 |
			| -624.463151 2798.710540 |
Element 9	0.031803	(45.399057, -72.684026)	| 0.390252 1.446795 |
			| 1.446795 12.663165 |

Model for n-gram “frozen won”

	Weight	Parameter	Covariance Matrix
Element 1	1.000000	(39.913643, -73.478554)	| 46.236170 137.524437 |
			| 137.524437 735.343411 |

Malaysia Mixture Models

Model for n-gram “malaysia airlines”

	Weight	Parameter	Covariance Matrix
Element 1	0.461538	(-1.250416, 103.339616)	| 27.542807 -21.295840 |
			| -21.295840 18.701594 |
Element 2	0.538462	(11.624666, -69.069520)	| 983.571720 -463.873504 |
			| -463.873504 1297.984191 |

Model for n-gram “mh370”

	Weight	Parameter	Covariance Matrix
Element 1	0.199978	(45.866735, -21.597188)	| 69.891936 287.693947 |
			| 287.693947 1609.642768 |
Element 2	0.724482	(-0.789572, 107.990169)	| 159.056521 -64.543983 |
			| -64.543983 114.347420 |
Element 3	0.075540	(-17.752512, -8.107306)	| 326.476808 305.232291 |
			| 305.232291 1414.540265 |

Earthquake Mixture Models

Each of these following mixture models describe the n-gram “earthquake” on separate days

March 17:

	Weight	Parameter	Covariance Matrix
Element 1	0.630351	(34.051611, -118.337015)	| 0.020965 -0.013613 |
			| -0.013613 0.062253 |

Element 2	0.334644	(33.787447, -86.332352)	| 154.654317 -195.093975 |
			| -195.093975 1462.265117 |
Element 3	0.035005	(50.751588, -125.837520)	| 45.461720 -46.390022 |
			| -46.390022 47.417675 |

March 18:

	Weight	Parameter	Covariance Matrix
Element 1	1.000000	(41.028669, 4.911066)	| 74.072700 -1.467219 |
			| -1.467219 11219.073164 |

March 19:

	Weight	Parameter	Covariance Matrix
Element 1	1.000000	(35.239937, -102.371224)	| 2.411554 -12.365129 |
			| -12.365129 245.283288 |

March 20:

	Weight	Parameter	Covariance Matrix
Element 1	1.000000	(27.929069, -72.482389)	| 381.018932 -1767.645711 |
			| -1767.645711 8718.352991 |

March 21:

	Weight	Parameter	Covariance Matrix
Element 1	1.000000	(36.928320, -64.698390)	| 8.741101 174.280746 |
			| 174.280746 5628.362164 |

Flu Test Case

Model for n-gram “flu”

	Weight	Parameter	Covariance Matrix
Element 1	0.333332	(-7.640210, 114.516163)	| 175.623701 -234.428422 |
			| -234.428422 333.925805 |
Element 2	0.666668	(32.357113, -62.455293)	| 695.643907 14.271422 |
			| 14.271422 1351.391583 |

Model for n-gram “cold”

	Weight	Parameter	Covariance Matrix
Element 1	0.061418	(41.456606, -72.819166)	| 0.737509 1.121888 |
			| 1.121888 2.496312 |
Element 2	0.049883	(54.430003, -5.375916)	| 2.374523 2.228594 |
			| 2.228594 3.664087 |
Element 3	0.134776	(39.724818, -77.576680)	| 5.158106 -0.893069 |
			| -0.893069 4.314380 |
Element 4	0.047456	(-2.050306, 122.396478)	| 316.198585 -193.129973 |
			| -193.129973 428.106749 |
Element 5	0.125123	(-28.605167, 24.883556)	| 14.708817 17.711015 |
			| 17.711015 21.330968 |
Element 6	0.049471	(32.15760,9 -82.205819)	| 22.776763 -16.349981 |
			| -16.349981 24.837485 |
Element 7	0.147997	(41.396185, -86.517625)	| 5.170202 -3.423225 |
			| -3.423225 12.107286 |
Element 8	0.113147	(32.413003, -95.209405)	| 9.334871 -1.739155 |

			-1.739155 8.191983
Element 9	0.112971	(35.253642, -114.721054)	6.052082 3.129762
			3.129762 16.801914
Element 10	0.012125	(44.859315, -73.244495)	0.487539 0.008562
			0.008562 6.022588
Element 11	0.099294	(52.192233, -0.884441)	1.722468 -0.812850
			-0.812850 3.811406
Element 12	0.010616	(-18.070790, -57.184716)	91.103144 -114.074715
			-114.074715 168.849644
Element 13	0.011436	(27.901405, 32.489563)	156.400562 -41.952162
			-41.952162 504.094872
Element 14	0.012729	(46.388272, -63.498023)	8.825673 -4.004170
			-4.004170 4.034620
Element 15	0.054661	(43.318841, -120.803186)	28.145600 7.284567
			7.284567 8.467314

Model for n-gram “sick”:

	Weight	Parameter	Covariance Matrix
Element 1	0.082346	(40.686242, -83.273595)	3.266447 1.331067
			1.331067 4.725487
Element 2	0.029924	(46.266313, -120.774759)	4.872820 0.386119
			0.386119 8.304257
Element 3	0.037806	(35.961071, -108.252690)	20.826393 2.431134
			2.431134 10.331019
Element 4	0.049855	(34.294547, -96.673435)	9.311568 1.042318

			1.042318 4.610360
Element 5	0.015628	(32.233922, 17.045488)	295.005865 -87.067472
			-87.067472 1245.276825
Element 6	0.005721	(-29.949887, -50.500818)	65.426380 59.086426
			59.086426 54.664648
Element 7	0.043106	(34.389048, -79.671114)	2.563862 1.918569
			1.918569 2.828366
Element 8	0.017165	(-33.380248, 156.606242)	29.846084 -32.158513
			-32.158513 121.732795
Element 9	0.099742	(35.312172, -118.513665)	3.729422 -2.615563
			-2.615563 4.555643
Element 10	0.174104	(52.880942, -1.737575)	2.673839 -1.430368
			-1.430368 4.441950
Element 11	0.073490	(41.715385, -73.363272)	2.942613 3.255420
			3.255420 7.619254
Element 12	0.058781	(29.792344, -96.220969)	91.103144 -114.074715
			-114.074715 168.849644
Element 13	0.071249	(42.166452, -88.757066)	6.193555 -5.444155
			-5.444155 10.967296
Element 14	0.052501	(-0.578450, 104.807007)	57.026506 -63.126673
			-63.126673 90.079489
Element 15	0.089958	(40.130734, -76.549791)	2.731265 -0.207763
			-0.207763 3.177700
Element 16	0.034915	(32.879734, -86.055892)	4.335035 3.234598
			3.234598 5.225993

Element 17	0.048485	(26.726157, -80.432815)	7.076740	-7.240533
			-7.240533	9.152393
Element 18	0.015222	(15.389554, 125.086943)	62.769102	36.391843
			36.391843	78.165441

Appendix B: Code:

The following section contains the final versions of code developed throughout the course of the project. While other code was developed through the course of the year, they merely build up the final versions below. Thus it makes little sense to include them. The code contained within this sections is a combination of python and Java. The TweetCombination.java program was responsible for post processing, TweetGeolocate.java was the MapReduce code, and the python code was used for tweet capture.

Listed below is the source code for the program which performed post processing work and handled Boolean logic math.

```
import jMEF.MixtureModel;
import jMEF.MultivariateGaussian;
import jMEF.PVector;
import jMEF.PVectorMatrix;
import jMEF.Parameter;

import java.awt.Color;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.StringTokenizer;
import java.util.zip.GZIPInputStream;

import javax.swing.JFrame;

import org.math.plot.Plot2DPanel;

public class TweetCombiner {
	//array of all n-grams
	private static ArrayList<String> Keys = new ArrayList<String>();
	//array of all GMMs
	private static ArrayList<MixtureModel> Models = new ArrayList<MixtureModel>();
	//number of times n-grams occured
	private static ArrayList<Integer> Occurence = new ArrayList<Integer>();
	//sets level of n-gramming, eg ngrams=2 tokenizes phrase to 2-grams
	private final static int ngrams =2;
	private  static ArrayList<MixtureModel> phraseModels;
	private  static ArrayList<String> finalPhrase;
	private  static ArrayList<Integer> finalOccurence;
	public static void main(String[] args) {
		phraseModels= new ArrayList<MixtureModel>();
		finalPhrase= new ArrayList<String>();
	    finalOccurence = new ArrayList<Integer>();
		// path to MapReduce output files
```

```
		reader("path");
		//gets input phrase/n-grams from users and tokenizes phrase into n-grams
		Scanner input = new Scanner(System.in);
		System.out.print("Enter target phrase: ");
		ArrayList<String> phrase= tokenize(input.nextLine());

		//gets models associated with n-grams user inputted
		for(String s:phrase){
			MixtureModel mm= pullModel(s);
			if(mm!=null&&!(finalPhrase.contains(s))){
				phraseModels.add(mm);
				finalPhrase.add(s);
				finalOccurence.add(Occurence.get(pullIndex(s)));
			}
		}
		//performs operations on each of the n-grams of user input phrase (can
modify operations easily by changing method call)
		for(int i=0;i<finalPhrase.size();i++){
			densityMapAndExport(i,finalPhrase.get(i));
		}
		//combines n-grams through either AND or OR boolean (change to
combineOR() for OR logic)
		combineAND();

		//outputs models with optional ability to graph model using javaPlot
using graphModel(String, MixtureModel, int) method

		for(int i=0;i<phraseModels.size();i++){

	graphModel(phraseModels.get(i),finalPhrase.get(i),finalOccurence.get(i));
			System.out.println(finalPhrase.get(i)+ " "+ phraseModels.get(i));

		}
	}

	//draws random points from GMM and outputs for heatmap generation
	private static void randomPointsAndExport(MixtureModel m, String name){
		PVector points[] = m.drawRandomPoints(numPoints);
		try{
			FileWriter writer = new
FileWriter("C:/Users/User/Desktop/results/output/"+name+" randpoints.csv");
			writer.append("latitude,longtitude");
			writer.append("\n");
			writer.flush();
			for(int i=0;i<numPoints;i++){
				writer.append(""+points[i].array[0]);
				writer.append(",");
				writer.append(""+points[i].array[0]);
				writer.append("\n");

			}
			writer.close();
		}
		catch(IOException e){
			e.printStackTrace();
```

```
			}

		}
		//creates a density map for a GMM
		private static void densityMapAndExport(int index,String name){
			//latitude +-90, longitude +- 180
			double [][] density= new double[180][360];
			for(int i=0;i<180;i++){
				for(int j=0;j<360;j++){
					PVector v= new PVector(2);
					v.array[0]=i-90;
					v.array[1]=j-180;
					density[i][j]=phraseModels.get(index).density(v);

				}
			}
			try{
				FileWriter writer = new
FileWriter("C:/Users/User/Desktop/results/output/"+name+"density.csv");
				writer.append("latitude,longtitude,density");
				writer.append("\n");
				writer.flush();
				for(int i=0;i<180;i++){
					for(int j=0;j<360;j++){
						writer.append(""+(i-90));
						writer.append(",");
						writer.append(""+(j-180));
						writer.append(",");
						writer.append(""+density[i][j]);
						writer.append("\n");
						writer.flush();
					}
				}
				writer.close();
			}
			catch(IOException e){
				e.printStackTrace();
			}
		}
		//variable used to set minimum accuracy when choosing which tokens to use
during combination
		private final static int COMBACC=10;
		//combines n-grams into phrases using AND boolean logic
		private static void combineAND(){
			ArrayList<String> combinePhrase=(ArrayList<String>) finalPhrase.clone();
			ArrayList<MixtureModel> combineModels=new ArrayList<MixtureModel>();
			for(String s: finalPhrase){

				for(int i=0;i<finalPhrase.size();i++){


		if(finalPhrase.get(i).contains(s)&&!(finalPhrase.get(i).equals(s))){

					if(finalOccurence.get(i)>COMBACC){
						System.out.println(finalOccurence.get(i));
```

```
                                        combinePhrase.remove(s);
                                }
                                else
                                        combinePhrase.remove(finalPhrase.get(i));

                        }
                }
        }

        for(String s: combinePhrase){
                combineModels.add(pullModel(s));
                System.out.println(s);
        }

        double [][] density= new double[180][360];
        for(int i=0;i<180;i++){
                for(int j=0;j<360;j++){
                        PVector v= new PVector(2);
                        v.array[0]=i-90;
                        v.array[1]=j-180;
                        double value=1;
                        for(int f=0;f<combinePhrase.size();f++)
                                value*=combineModels.get(f).density(v);
                        density[i][j]=value;

                }
        }


        try{
                FileWriter writer = new
FileWriter("C:/Users/User/Desktop/results/output/phrasedensity.csv");
                writer.append("latitude,longtitude,density");
                writer.append("\n");
                writer.flush();
                for(int i=0;i<180;i++){
                        for(int j=0;j<360;j++){
                                writer.append(""+(i-90));
                                writer.append(",");
                                writer.append(""+(j-180));
                                writer.append(",");
                                writer.append(""+density[i][j]);
                                writer.append("\n");
                                writer.flush();
                        }
                }
                writer.close();
        }
        catch(IOException e){
                e.printStackTrace();
        }


}
//combines n-grams into phrases using OR boolean logic
```

```
	private static void combineOR(){
		ArrayList<String> combinePhrase=(ArrayList<String>) finalPhrase.clone();
		ArrayList<MixtureModel> combineModels=new ArrayList<MixtureModel>();
		for(String s: finalPhrase){

			for(int i=0;i<finalPhrase.size();i++){


	if(finalPhrase.get(i).contains(s)&&!(finalPhrase.get(i).equals(s))){

					if(finalOccurence.get(i)>COMBACC){
						System.out.println(finalOccurence.get(i));
						combinePhrase.remove(s);
					}
					else
						combinePhrase.remove(finalPhrase.get(i));

				}
			}
		}

		for(String s: combinePhrase){
			combineModels.add(pullModel(s));
			System.out.println(s);
		}

		double [][] density= new double[180][360];
		for(int i=0;i<180;i++){
			for(int j=0;j<360;j++){
				PVector v= new PVector(2);
				v.array[0]=i-90;
				v.array[1]=j-180;
				double value=0;
				for(int f=0;f<combinePhrase.size();f++)
					value+=combineModels.get(f).density(v);
				density[i][j]=value;

			}
		}


		try{
			FileWriter writer = new
FileWriter("C:/Users/User/Desktop/results/output/phrasedensity.csv");
			writer.append("latitude,longtitude,density");
			writer.append("\n");
			writer.flush();
			for(int i=0;i<180;i++){
				for(int j=0;j<360;j++){
					writer.append(""+(i-90));
					writer.append(",");
					writer.append(""+(j-180));
					writer.append(",");
					writer.append(""+density[i][j]);
					writer.append("\n");
```

```
                        writer.flush();
                    }
                }
                writer.close();
            }
            catch(IOException e){
                e.printStackTrace();
            }


    }
    //number of points drawn for heatmap/graphing
    private static int numPoints=1000;
    //graphs input GMM using javaPlot library
    private static void graphModel(MixtureModel mm, String name, int occ){
        PVector points[] = mm.drawRandomPoints(numPoints);
        double x[]= new double[numPoints];
        double y[]= new double[numPoints];
        for(int i=0;i<numPoints;i++){
            x[i]=points[i].array[0];
            y[i]=points[i].array[1];
        }
        //System.out.println(name+" "+occ+"\n"+mm);
        Plot2DPanel graph = new Plot2DPanel();
        graph.addScatterPlot(name, Color.red, x, y);
        JFrame frame = new JFrame(name);
        frame.setSize(600, 600);
        frame.setContentPane(graph);
        frame.setVisible(true);
    }

    //returns array index at which string s exists in Keys
    private static int pullIndex(String s){
        for(int i=0;i<Keys.size();i++){
            if(s.equalsIgnoreCase(Keys.get(i)))
                return i;
        }
        return (Integer) null;
    }
    //returns mixture model associated with Key s from Model array
    private static MixtureModel pullModel(String s){
        int i=0;
        for(String key: Keys){
            if(s.equalsIgnoreCase(key)){
                MixtureModel mm= Models.get(i);
                if(paramCheck(mm))
                    return mm;
                else
                    return null;
            }
            i++;
        }
        return null;
    }
    //checks parameters of Mixture Model and returns if they are null or not
```

```
private static boolean paramCheck(MixtureModel mm) {
	Parameter param[]=null;
	try {
		 param = mm.param;
	} catch (java.lang.NullPointerException e) {
		return false;
	}
	for (Parameter p : param) {
		if (new Double(p.InnerProduct(p)).equals(Double.NaN))
			return false;
	}
	return true;
}
//tokenizes input user phrase
private static ArrayList<String> tokenize(String line){
	ArrayList<String> tokens= new ArrayList<String>();
	try {
		String output = "";

		StringTokenizer tokenizer = new StringTokenizer(line);
		ArrayList<String> nGrams = new ArrayList<String>();
		while (tokenizer.hasMoreElements()) {

			nGrams.add(clean((String) tokenizer.nextElement()));
		}
		for (int nCounter = ngrams; nCounter >= 1; nCounter--) {
			for (int i = 0; i < nGrams.size(); i++) {

				int j = i + (nCounter - 1);
				j = testLastIndex(j, nGrams.size());
				for (int f = i; f <= j; f++) {
					if (f != j)
						output = output + nGrams.get(f) + " ";
					else
						output = output + nGrams.get(f);
				}
				tokens.add(output.trim());

				output = "";
			}
		}
	} catch (Exception e) {

	}
	return tokens;
}
//helper method used during tokenization
private static int testLastIndex(int j, int length) {
	int f = j;
	if (!(f < length)) {
		f--;
		f = testLastIndex(f, length);
	}
	return f;
}
```

```
	//cleans input phrase to remove upperCase and remove select characters
	private static String clean(String line) {
		String element = line;
		element = element.replace("!", "");
		element = element.replace('?', '\0');
		element = element.replace(":", "");
		element = element.replace(".", "");
		element = element.replace(",", "");
		element = element.replace("=", "");
		element = element.replace(";", "");
		element = element.replace("\"", "");
		element = element.toLowerCase();
		return element;
	}
	//iterates through output folder and opens every valid output file
	private static void reader(String path) {
		File folder = new File(path);
		String output = "";
		for (File file : folder.listFiles()) {
			FileInputStream fis;
			try {
				fis = new FileInputStream(file);
				ObjectInputStream ois = new ObjectInputStream(
						new GZIPInputStream(fis));
				output += (String) ois.readObject();
				ois.close();
				fis.close();
			} catch (FileNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (IOException e) {
				//System.out.println(file.getAbsolutePath());
				// TODO Auto-generated catch block
				//e.printStackTrace();
				//triggered when reading _SUCCESS file, ignore
			} catch (ClassNotFoundException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}

		}
		read(output);
	}
	//reads output file, and inputs keys with valid GMMs into Keys and Models
array. Also populates Occurrence array with amount of times each Key was heard
	private static void read(String input) {
		String models[] = input.split("><");
		// System.out.println(models.length);
		for (int i = 0; i < models.length; i++) {
			//System.out.println(models[i]);
			String modelData[] = models[i].split(":");
			// do stuff with key modelData[o]
			try {

				MixtureModel mm = new MixtureModel(
```

```
                                    Integer.parseInt(modelData[1]));
                        // System.out.println(modelData[0]+" "+modelData[1]+"
"+modelData[2]);
                        for (int j = 3; j < modelData.length; j++) {
                              String elementLine = modelData[j];
                              String elements[] = elementLine.split(",");
                              PVectorMatrix param = new PVectorMatrix(2);
                              mm.weight[j - 3] = Double.parseDouble(elements[0]);
                              param.v.array[0] = Double.parseDouble(elements[1]);
                              param.v.array[1] = Double.parseDouble(elements[2]);
                              param.M.array[0][0] =
Double.parseDouble(elements[3]);
                              param.M.array[0][1] =
Double.parseDouble(elements[4]);
                              param.M.array[1][0] =
Double.parseDouble(elements[5]);
                              param.M.array[1][1] =
Double.parseDouble(elements[6]);
                              mm.param[j - 3] = param;
                              // System.out.println(mm);
                        }
                        mm.EF=new MultivariateGaussian();
                        Keys.add(i, modelData[0]);
                        Models.add(i, mm);
                        Occurence.add(i,Integer.parseInt(modelData[2]));
                  } catch (java.lang.NumberFormatException e) {
                        // e.printStackTrace();
                        Keys.add(i, modelData[0]);
                        Models.add(i, null);
                        Occurence.add(i,0);
                  }
            }
      }

}
```

Listed below is the source code for the program which handled tweet collection from Twitter.

```
import sys

import tweepy

import oauth2

import logging

#data needed for authentication

consumerKey =

consumerSecret =
```

```
accessTokenKey =
accessTokenSecret =

#sets up authentication
auth1 = tweepy.OAuthHandler(consumerKey, consumerSecret)
auth1.set_access_token(accessTokenKey, accessTokenSecret)

#gets filepath for output, sets tweetlimit to user input
path =str(raw_input("Enter your filepath: "))
tweetLimit=int(raw_input("Enter max tweets: "))

#opens file
file = open(path,'w')

#receives tweet objects from twitter, stores them in output format
#overwrites tweepy's StreamListener class
#ends streaming after collecting max tweets
#disregards non-geotagged tweets
class StreamListener(tweepy.StreamListener):
    def __init__(self, maxnum):
        super(StreamListener, self).__init__()
        logging.propogate=False
        self.max=maxnum
        self.c=0
```

```
def _start(self, async):
    print "here"
    self.running = True
    if async:
        Thread(target=self._run).start()
    else:
        self._run()

def on_status(self, tweet):

    if(self.c>self.max):
        return False
    if not (tweet.geo==None):
            if(self.c%(self.max*0.01)==0):
                print self.c
            text = tweet.text.encode('utf-8')
            text=text.replace('\n','')
            textList=text.split("http")
            geo = str(tweet.geo)
            geoList=geo.split("[")
            coordList=geoList[1].split(",")
            lat=coordList[0]
            longitude=(coordList[1].split("]"))[0].replace(' ','')
            tweetid=tweet.id
```

```
            time= tweet.created_at
            print >>file, textList[0],"\t",lat,"\t",longitude,"\t",tweetid,"\t",time
            self.c=self.c+1




    def on_error(self, status_code):
        #print 'Error: ' + str(status_code)
        return False

#uses try catch in order to prevent errors from stopping stream
# reinitiates stream if error is thrown
def start(myListener):
    try:
        streamer = tweepy.Stream(auth=auth1, listener=myListener)
        streamer.sample()
    except:
        start(myListener)
#disables logging to ignore non important tweepy whining
logging.disable(logging.CRITICAL)
myListener= StreamListener(maxnum=tweetLimit)
#starts stream
start(myListener)
```

#writes tweets to file and ends program

print "done with streaming, writing to file"

file.close()

Listed below is the source code for the MapReduce program which ran on top of Amazon cloud.

```
package org.team63;

import jMEF.*;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Random;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.zip.GZIPOutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.mortbay.log.Log;

public class TwitterGeolocate {
	// constant which sets level to which messages are broken into n-grams, for
	// example ngrams=2 will break messages into 2-grams (can be modified by
	// command line input)
	public static int ngrams = 2;
	// if set to 0 only certain characters will be removed during tokenization,
	// else all non-alphabet characters will be removed (can be modified by
	// command line input)
	public static int fullClean = 0;

	/*
	 * creates job with user paramters, runs job, exists with 0 (success) 1
	 * (failure) requires command line input inputpath outputpath ngramlevel
	 * fullclean
	 */
	public static void main(String[] args) throws Exception {
		// gets input for n-gram level from command line, if invalid defaults to
		// 2
		try {
```

```
		ngrams = Integer.parseInt(args[2]);

		if (!(ngrams > 0))
			ngrams = 2;
	} catch (Exception e) {
		ngrams = 2;
		System.out.println("ngrams parsing failed");
	}
	// gets input for parsing "clean" level from command line, if invalid
	// defaults to off
	try {
		int dat = Integer.parseInt(args[3]);
		if (dat == 0 || dat == 1)
			fullClean = dat;
		else
			throw new java.lang.IllegalArgumentException();
	} catch (Exception e) {
		fullClean = 0;
		System.out.println("defaulting to no fullclean");
	}
	// creates configuration for job
	Configuration conf = new Configuration();
	System.out.println("starting: " + ngrams);
	// sets input/output path to values from command line
	Path inputPath = new Path(args[0]);
	Path outputPath = new Path(args[1]);

	// creates new job
	Job job = new Job(conf, "Tweet Geolocation");
	job.setJarByClass(TwitterGeolocate.class);

	// sets input format to default TextInputFormat
	job.setInputFormatClass(TextInputFormat.class);

	// sets Mapper to defined map class
	// defines output of Mapper
	job.setMapperClass(TokenizerMapper.class);
	job.setMapOutputKeyClass(Text.class);
	job.setMapOutputValueClass(Text.class);

	// sets reducer class
	job.setReducerClass(GMMReduce.class);

	// sets inputpath path
	FileInputFormat.addInputPath(job, inputPath);

	// notifies hadoop of our custom output format usage
	job.setOutputFormatClass(GMMOutputFormat.class);

	// job.setOutputKeyClass(Text.class);
	// job.setOutputValueClass(Text.class);

	// sets output path
	FileOutputFormat.setOutputPath(job, outputPath);
```

```
		// Submits job, exits with 0 if success, 1 if failure
		System.out.println("submitting");
		job.submit();
		if (job.waitForCompletion(true)) {
			System.out.println("success");
			System.exit(0);
		} else {
			System.out.println("failure");
			System.exit(1);
		}
	}

	/*
	 * Mapper class gets tweets from input outputs Key Value pair <ngram,
	 * <longitude, latitude>> ngrams are outputted as text longitude latitude
	 * outputted as one string separated by ,
	 */
	public static class TokenizerMapper extends
			Mapper<Object, Text, Text, Text> {
		// performs map function described above
		public void map(Object key, Text value, Context context)
				throws IOException, InterruptedException {
			StringTokenizer itr = new StringTokenizer(value.toString(),
"\n");
			while (itr.hasMoreTokens()) {
				String line = itr.nextToken();

				try {
					String output = "";
					String[] tokens = line.split("\t");
					Double longitude = Double.parseDouble(tokens[1]);
					Double latitude = Double.parseDouble(tokens[2]);
					StringTokenizer tokenizer = new
StringTokenizer(tokens[0]);
					ArrayList<String> nGrams = new ArrayList<String>();
					while (tokenizer.hasMoreElements()) {
						nGrams.add(clean((String)
tokenizer.nextElement()));
					}
					for (int nCounter = ngrams; nCounter >= 1; nCounter-
-) {
						for (int i = 0; i < nGrams.size(); i++) {

							int j = i + (nCounter - 1);
							j = testLastIndex(j, nGrams.size());
							for (int f = i; f <= j; f++) {
								if (f != j)
									output = output +
nGrams.get(f) + " ";
								else
									output = output +
nGrams.get(f);
							}
							context.write(new Text(output.trim()),
new Text(
```

```
                                                        longitude + "," +
latitude));
                                                output = "";
                                        }
                                }
                        } catch (Exception e) {

                        }

                }
        }

        // helper class which removes formatting during tokenization
        private String clean(String line) {
                String element = line;
                if (fullClean == 0) {

                        element = element.replace("!", "");
                        element = element.replace("?", "");
                        element = element.replace(":", "");
                        element = element.replace(".", "");
                        element = element.replace(",", "");
                        element = element.replace("*", "");
                        element = element.replace("=", "");
                        element = element.replace(";", "");
                        element = element.replace("\"", "");
                        element = element.toLowerCase();
                        return element;
                }
                element = element.replaceAll("\\W", "");
                element = element.toLowerCase();
                return element;
        }

        // helper for tokenization
        private int testLastIndex(int j, int length) {
                int f = j;
                if (!(f < length)) {
                        f--;
                        f = testLastIndex(f, length);
                }
                return f;
        }

}

/*
 * test reducer function that counts instances of n-grams outputs <n-gram, #
 * times occurred> n-grams is a text field # of times occurred in number
 * stored in string
 */
public static class TestRed extends Reducer<Text, Text, Text, Text> {
        public void reduce(Text key, Iterable<Text> values, Context context)
                        throws IOException, InterruptedException {
                int n = 0;
```

```
                for (Text value : values) {
                        n++;
                }
                context.write(key, new Text("" + n));
        }
}

/*
 * real reducer method used gets <ngram, <longitude, latitude>> key value
 * pairs from mapper fits GMM to each n-gram outputs <ngram, GMM> uses
 * custom output format
 */
public static class GMMReduce extends
                Reducer<Text, Text, Text, MixtureModel> {
        // checks paramters of mixture models, returns null if GMM contains null
        // values
        private boolean paramCheck(MixtureModel mm) {
                Parameter param[] = null;
                try {
                        param = mm.param;
                } catch (java.lang.NullPointerException e) {
                        return false;
                }
                for (Parameter p : param) {
                        if (new Double(p.InnerProduct(p)).equals(Double.NaN))
                                return false;
                }
                return true;
        }

        // performs reduce task described above
        public void reduce(Text key, Iterable<Text> values, Context context)
                        throws IOException, InterruptedException {
                ArrayList<Double> longitude = new ArrayList<Double>();
                ArrayList<Double> latitude = new ArrayList<Double>();

                for (Text val : values) {
                        String[] tokens = val.toString().split(",");
                        longitude.add(Double.parseDouble(tokens[0]));
                        latitude.add(Double.parseDouble(tokens[1]));

                }
                PVector[] points = new PVector[longitude.size()];
                for (int i = 0; i < points.length; i++) {
                        PVector p = new PVector(2);
                        p.array[0] = longitude.get(i);
                        p.array[1] = latitude.get(i);
                        points[i] = p;
                }
                MixtureModel mm = null;
                // runs kmeans algorithm to break set into clusters, then fits
GMM
                // to clusters
                // starting at maximum estimate and iterating down till valid GMM
is
```

```
                        // made
                        for (int j = ((int) Math.log(points.length / 2)); j > 0; j--) {

                                Vector<PVector>[] clusters = KMeans.run(points, j);
                                // try catch used because null GMMs will cause
ArrayOutOfBounds
                                // and NullPointer Exceptions
                                try {
                                        mm = BregmanSoftClustering.initialize(clusters,
                                                        new MultivariateGaussian());
                                        mm = BregmanSoftClustering.run(points, mm);
                                        mm.numInstances = points.length;
                                } catch (java.lang.ArrayIndexOutOfBoundsException e) {

                                } catch (java.lang.NullPointerException e) {

                                } catch (Exception e) {
                                        Log.info("un-accounted for exception" +
e.toString());
                                }
                                if (paramCheck(mm)) {
                                        // if valid mixture model is made loop is broken and
GMM is
                                        // outputted
                                        j = 0;
                                }
                        }

                        context.write(key, mm);
                }
        }

        /*
         * performs exact same task as above reduce method outputs <ngram, GMM> in
         * text format not used because it is less efficient
         */
        public static class MyTextGMMReducer extends
                        Reducer<Text, Text, Text, Text> {
                private boolean paramCheck(MixtureModel mm) {
                        Parameter param[] = null;
                        try {
                                param = mm.param;
                        } catch (java.lang.NullPointerException e) {
                                return false;
                        }
                        for (Parameter p : param) {
                                if (new Double(p.InnerProduct(p)).equals(Double.NaN))
                                        return false;
                        }
                        return true;
                }

                public void reduce(Text key, Iterable<Text> values, Context context)
                                throws IOException, InterruptedException {
                        ArrayList<Double> longitude = new ArrayList<Double>();
```

```
ArrayList<Double> latitude = new ArrayList<Double>();
Log.info("Starting");
for (Text val : values) {
	String[] tokens = val.toString().split(",");
	longitude.add(Double.parseDouble(tokens[0]));
	latitude.add(Double.parseDouble(tokens[1]));
}
PVector[] points = new PVector[longitude.size()];
for (int i = 0; i < points.length; i++) {
	PVector p = new PVector(2);
	p.array[0] = longitude.get(i);
	p.array[1] = latitude.get(i);
	points[i] = p;
}
MixtureModel mm = null;
System.out.println(key);
for (int j = ((int) Math.log(points.length / 2)); j > 0; j--) {

	Vector<PVector>[] clusters = KMeans.run(points, j);
	// System.out.println("starting breg");
	try {
		mm = BregmanSoftClustering.initialize(clusters,
				new MultivariateGaussian());
		mm = BregmanSoftClustering.run(points, mm);
	} catch (java.lang.ArrayIndexOutOfBoundsException e) {

	}

	if (paramCheck(mm)) {
		j = 0;
	}
}
String output = "";
try {
	output += key + ":" + mm.size;
	for (int i = 0; i < mm.size; i++) {
		output += ":";
		PVectorMatrix param = ((PVectorMatrix) mm.param[i]);
		output += mm.weight[i];
		output += ",";
		output += param.v.array[0];
		output += ",";
		output += param.v.array[1];
		output += ",";
		output += param.M.array[0][0];
		output += ",";
		output += param.M.array[0][1];
		output += ",";
		output += param.M.array[1][0];
		output += ",";
		output += param.M.array[1][1];
	}
	output += "><";
} catch (java.lang.NullPointerException e) {
	output += key + ":null" + "><";
```

```
                }
                context.write(key, new Text(output));
        }
}

/*
 * extend Hadoops RecordWriter class Instance created by below
 * GMMOutputFormatClass Reducers call write to commit their output Once all
 * Reducers on node are done close is called Writes GMM to an
 * ObjectOutputSream Uses our custom format
 */
public static class GMMRecordWriter extends
                RecordWriter<Text, MixtureModel> {
        private ObjectOutputStream out;
        private String output;

        // creates new instance
        public GMMRecordWriter(ObjectOutputStream stream) {
                out = stream;
                output = "";
        }

        @Override
        // outputs final data and closes stream
        public void close(TaskAttemptContext arg0) throws IOException,
                        InterruptedException {
                out.writeObject(output);

                out.close();
        }

        @Override
        // switches GMM to custom format
        public void write(Text key, MixtureModel mm) throws IOException,
                        InterruptedException {
                try {
                        output += key + ":" + mm.size + ":" + mm.numInstances;
                        for (int i = 0; i < mm.size; i++) {
                                output += ":";
                                PVectorMatrix param = ((PVectorMatrix) mm.param[i]);
                                output += mm.weight[i];
                                output += ",";
                                output += param.v.array[0];
                                output += ",";
                                output += param.v.array[1];
                                output += ",";
                                output += param.M.array[0][0];
                                output += ",";
                                output += param.M.array[0][1];
                                output += ",";
                                output += param.M.array[1][0];
                                output += ",";
                                output += param.M.array[1][1];
                        }
                        output += "><";
```

```
                } catch (java.lang.NullPointerException e) {
                        output += key + ":null" + "><";
                }
        }
    }

    /*
     * Overwrites Hadoop's FileOutputFormat class Requires reducers to output
     * <Text,MixtureModel> objects Creates files, creates recordwriters to write
     * reduce output to files Technically can only support one file, but due to
     * overwriting problems in Amazon cloud random int is added to end of
     * filename which allows multiple file output While useful workaround,
     * should switch to actual Multiple File Output Format system in Hadoop
     */
    public static class GMMOutputFormat extends
                FileOutputFormat<Text, MixtureModel> {
        @Override
        public RecordWriter<Text, MixtureModel> getRecordWriter(
                        TaskAttemptContext arg0) throws IOException,
                        InterruptedException {
                // gets path from Hadoop
                Path path = FileOutputFormat.getOutputPath(arg0);
                Random rand = new Random();
                // creates output file in path with random filename to avoid
                // overwrite

                Path fullPath = new Path(path, "result" + rand.nextInt());

                // creates file
                FileSystem fs = path.getFileSystem(arg0.getConfiguration());
                ObjectOutputStream out = new ObjectOutputStream(
                                new GZIPOutputStream(fs.create(fullPath, arg0)));

                // creates new recordwriter
                return new GMMRecordWriter(out);
        }

    }

}
```