UCB MIDS W205 Project Documentation
Fall 2016 - Section 4
12/06/2016

Shuang Chan
David Skarbrevik

Streaming Twitter Data for Sentiment Analysis of 2016 U.S. Presidential Candidates

[If you are simply looking for instructions on running our application, please see the README.md file of this project's associated github repo: https://github.com/dskarby/MIDSW205_Project]

Introduction

The main aim of our project is to determine the political sentiments of the U.S. population during the period of the 2016 presidential election. To achieve this aim, we have chosen to analyze streaming Twitter data. Specifically, tweets that mention Hillary Clinton or Donald Trump.

There are many ways that our aim could be approached, however, Twitter is an ideal platform to pursue our aim because Twitter's data is composed of short reactionary text messages from a large sample of the U.S. population. Not only is Twitter data well suited for sentiment analysis, it is also a great source of real-time information. Twitter users post their feelings and thoughts as soon as an event occurs. This gives the advantage of being able to strongly link changes in Twitter data patterns to specific events.

Just as significant, some sources quote that, as of 2016, there are 67 million American Twitter users with 300 million tweets being sent per day (worldwide). This staggering volume of users/tweets means there is no shortage of data for a wide range of topics (including politics).

Further, it should be noted that another overwhelming advantage of using Twitter is the richness of their data. Beyond just the textual message that a tweet contains, Twitter attaches a large amount of metadata (e.g. time/location of tweet, # of re-tweets, # of followers) that can help to answer a variety of questions.

This project has the obvious practical benefit of helping to understand the political feelings and motivations of US citizens as a presidential election approaches. It may help to uncover certain biases in the way Americans approach politics. It may help show how political opinions are influenced. In a novel application, this project may help predict who would become president of the United States in 2016. Ultimately, we hope to make our analysis publicly available so that

anyone can benefit from this project and possibly apply our analysis in new and interesting ways.

Design Considerations

The volume and velocity of the Twitter's streaming data presents a huge technical challenge. A horizontally scalable solution is needed. As noted in the previous diagram, the data injection, storage and processing layers are all based on a distributed environment. This design allows us to scale up as more computational power is needed without any major system refactoring.

Furthermore, the volume and velocity of the Twitter data concerning politics varies greatly over time. For instance, we expect high volume and velocity after an election debate or a major political incident. Such unpredictability warrants a solution that requires a small initial investment and very flexible scalability as demand ramps up or down.

Since we are only concerned about streaming Twitter data, Kappa architecture is a suitable design for its simplicity to handle real time event based processing.

Architecture Overview

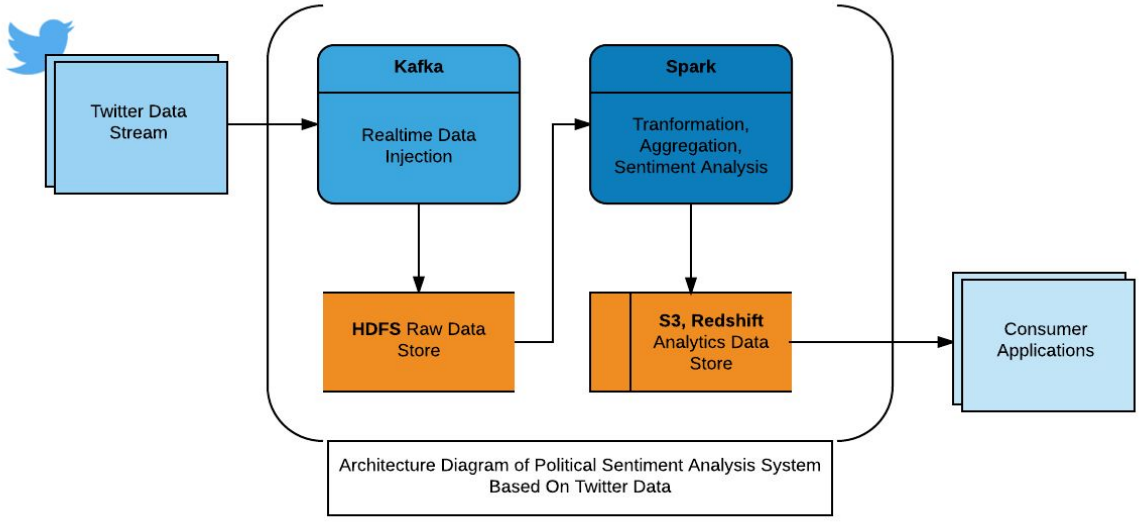


Figure 1: Overview of Application Architecture

What we propose is a simple Kappa architecture. All Twitter streaming data is processed in near real time into an event layer implemented in Kafka. Using Kafka also has the added benefit of horizontal scaling as stream size may vary over time. The data is persisted in a distributed file system (HDFS) for fault tolerance and scalability. Since the data is persisted, we enable re-processing of historical data as new use cases of the data are developed.

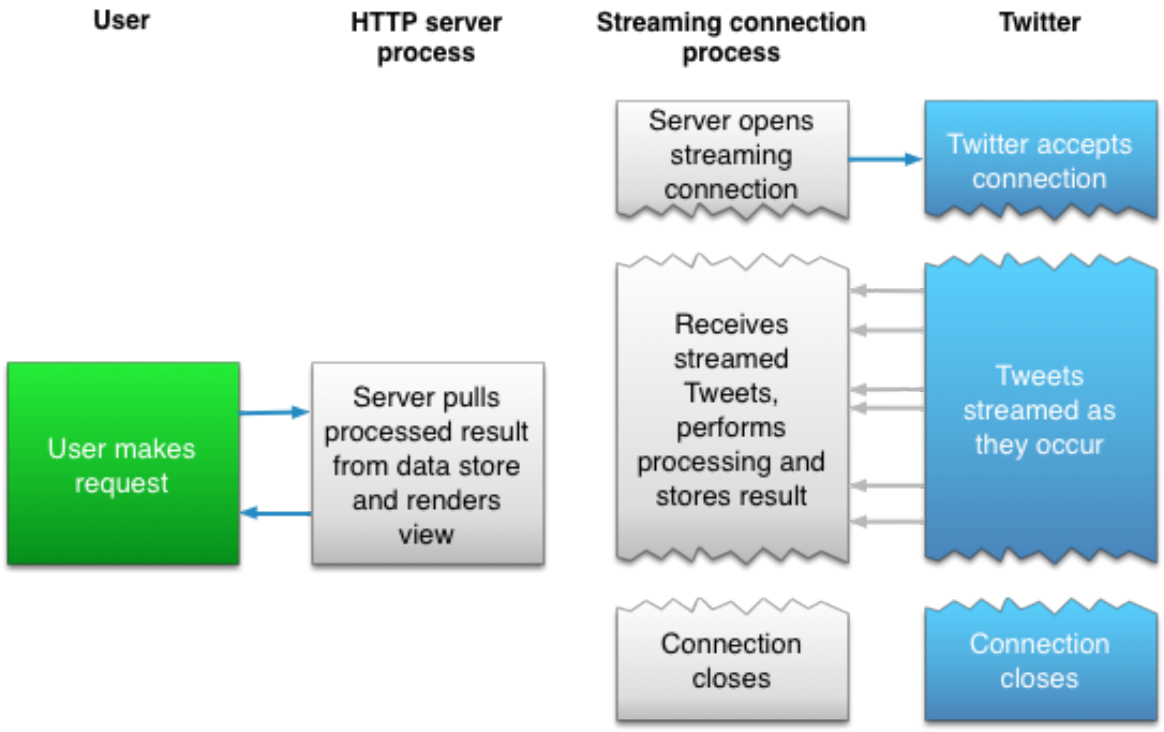
The processing layer will be implemented in Apache Spark as it provides high computational performance in a distributed framework. The core data processing of transformation, aggregation and sentiment analysis will all be done in this layer. Since Spark is a distributed computing environment, we can easily scale horizontally as more computational power is needed.

The analytical output will be stored in an object storage (AWS S3) for consumer applications such as news websites, mobile apps and other social media platforms. For real time consumers, the data will also be available in a high performance data warehouse environment (AWS Redshift).

Data Ingestion (Twitter to Kafka)

Step 1) Twitter API connection

To stream data from Twitter's API with Apache Kafka, we used Twitter's "hosebird client" (hbc) which is a Java package that allows us to implement the Twitter API into our Kafka Producer file. The implementation is simple as you can just add a dependency in the Apache Maven pom.xml file of your Java project/program (our Kafka Producer file) to pull the hbc packages. Below is a summarizing diagram of how Twitter's streaming API allows us to make requests for data:



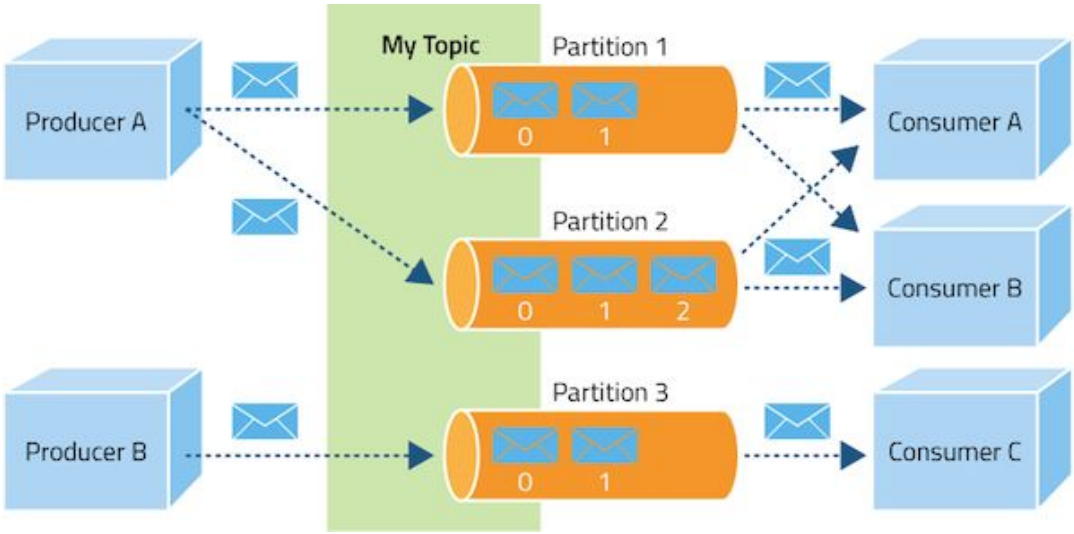
Twitter 2016

Figure 2: Twitter API diagram

It is worth noting that to successfully connect to the Twitter's streaming API, one must have valid Twitter Oauth tokens. This is a simple process that can be completed at https://dev.twitter.com/.

Step 2) Ingesting Twitter stream via Apache Kafka

These Twitter requests are packaged in a Kafka "producer" that contains a "topic" (in this case, our specific Twitter search terms). The producer sends topics to a "broker" that partitions topics and ultimately sends their messages to a "consumer". In our case, we used LinkedIn's Camus (explained in step 3) as a consumer that sends data to HDFS. Below is a general graphical representation of Kafka's workflow:



Cloudera 2016

Figure 3: Kafka workflow diagram

As explained in the "Design Consideration" section of the documentation, using Kafka allows for horizontal scalability as incoming Twitter data may be more or less at any given time. It also allows for fault tolerance as issues in a stream arise and via its partitioning system allows for efficient processing.

Step 3) Kafka to HDFS using LinkedIn's "Camus"

From Kafka, messages need to be consumed by HDFS. To facilitate this transfer, we have implemented LinkedIn's Camus project which was designed specifically to handle the transfer of Kafka streams to HDFS. The advantage of using Camus is that it adds fault tolerances and automates certain aspects like offset/timestamp recognition and Hadoop job creation that could be highly time consuming to implement from scratch. That being said, many adjustments were made to Camus for the purpose of this application. At it's core, Camus has two important classes, the decoder and the writer (basically, how it reads messages from Kafka and how it outputs them to HDFS). A custom decoder class was produced for this project to ensure that the data written into HDFS would be a CSV format for the convenience of our sentiment analysis script. To clarify, the Json format is still mostly preserved, but within a CSV format (example row = "twitter topic","tweet info in Json format").

On top of a custom decoder class, changes were made to the configuration file to compensate for the fact that we did not have the Kafka stream running continuously during testing/development phases of this project. Below is a detailed diagram of the workflow Camus performed to transfer messages from Kafka to HDFS:

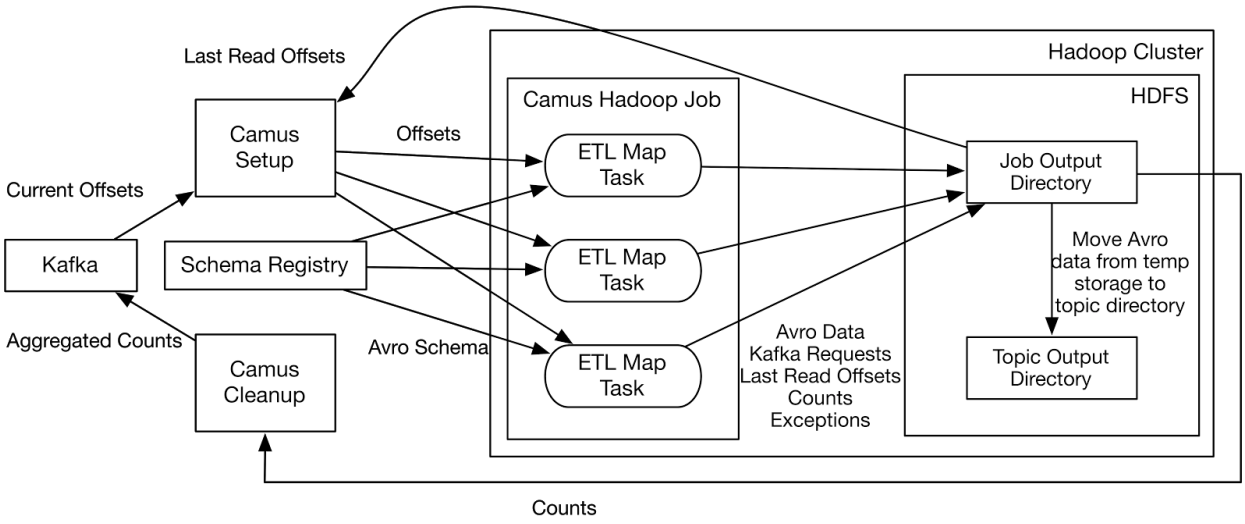


Figure 4: Camus workflow diagram

The diagram uses Avro data as an illustrative example, however essentially any format of data is supported by Camus as long as an appropriate “decoder” class is made.

Note that as of 12/2016, Camus is no longer under active support and was replaced by “Gobblin” (still maintained by LinkedIn). Camus is still a suitable solution for our application, but a future improvement of this application will be to implement Gobblin as programs like Hadoop and Kafka continue to update and require new dependencies.

Data Storage and Schema

The data layer of the system is Hadoop based. HDFS is a highly scalable distributed file system. It provides a very cost effective way to horizontally scale across cheap utility servers.

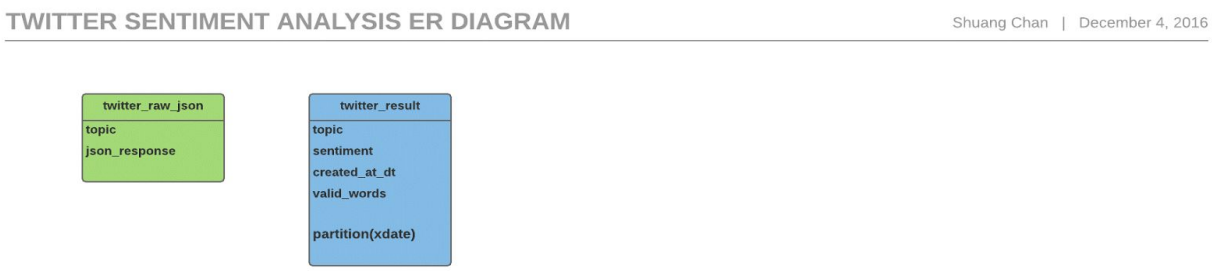


Figure 5: Sentiment Analysis ER Diagram

The database schema consists two entities: *twitter_raw_json* and *twitter_result*. The *twitter_raw_json* table is used to store unaltered Twitter JSON messages captured by the data injection layer. The *twitter_result* table is used to store the outputs from the analysis processing layer.

HIVE DDL

```
--Table to store the raw Twitter JSON data
drop table IF EXISTS twitter_raw_json;
create EXTERNAL table twitter_raw_json
(
	topic string,
	json_response string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
"separatorChar" = ",",
"quoteChar" = "",
"escapeChar" = '\\'
)
STORED AS TEXTFILE
LOCATION '/user/w205/twitter/topic/';

--Table to store the results of the sentiment analysis
drop table IF EXISTS twitter_result;
CREATE EXTERNAL TABLE twitter_result
(
	topic string,
	sentiment string,
	created_at_dt timestamp,
	valid_words string
)
partitioned by (xdate date)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
"separatorChar" = ",",
"quoteChar" = "",
"escapeChar" = '\\'
)
STORED AS TEXTFILE
LOCATION '/user/w205/twitter/twitter_result';
```

Sentiment Corpus

In order to create a model to classify Twitter message by sentiment, a sentiment corpus is required. The dataset should include training data and the assigned sentiment labels. Below are some examples:

Label,Data
positive,i'm just messing with you politics got you tense can understand doe have nice evening
negative,epic hour overnight challenge trump tower houur overnight sneaking into trump tower
positive,electrify your trump sign for instant justice
negative,much would give anything for her push out trump its gonna long shot sadly

To manually collect and classify Twitter messages for a sentiment corpus suitable for machine learning model is extremely costly and time-consuming. Our approach is to adapt a method introduced by Read[1], in which the emoticons embedded in the messages are used for auto-classification. For instance, :) indicates positive emotions, and :(indicates negative emotions.

We use the Tweepy API to create a simple Python program to collect and classify Twitter message by emotion based on the following emoticons.

```
# Define positive and negative emoticons
pos_emoticons = [":)", ":-)", ": )", ":D", "=)"];
neg_emoticons = [":(", ":-(", ": ("];
```

The program also allows us to create sentiment corpus based on different topics. For the purpose of the project. We use the following key words.

```
# Specify key words
keywords = ["hillary", "trump", "presidential election", "politics"]
```

[1] J. Read. Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In Proceedings of ACL-05, 43nd Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, 2005.

Sentiment Analysis

Once the sentiment corpus is created, the analysis program is relatively straightforward. It basically includes four stages: 1) Load sentiment corpus in memory. 2) Train a model. 3) Retrieve the Twitter messages and use the model to classify the sentiments. 4) Store the results.

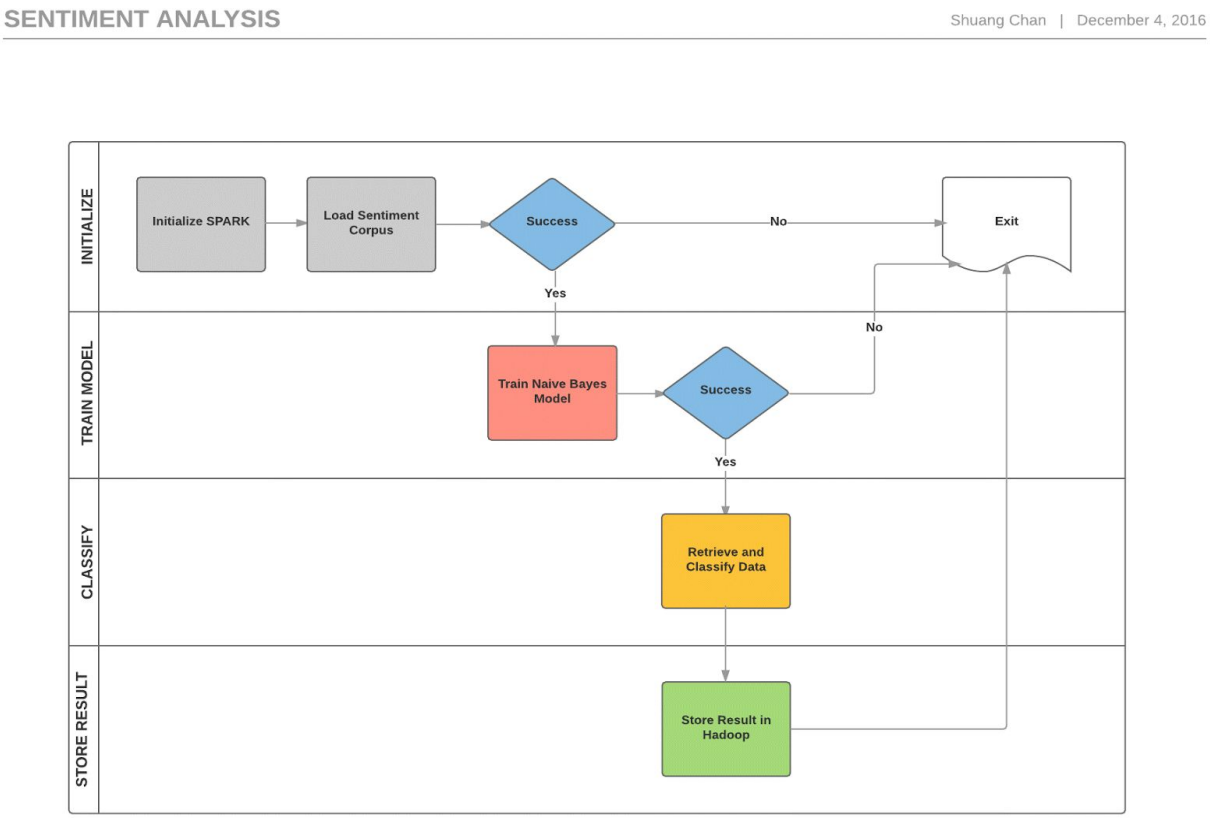


Figure 6: Sentiment Analysis Program Flow Chart

The analysis program is written in Python, and is designed to run within the Apache Spark framework. Spark is much faster than MapReduce due to its superior in-memory and parallel processing capability. It also has built-in support of data streaming, and works alongside with Hadoop.

The machine learning model is built using the NLTK (http://www.nltk.org/) library, a natural language process tookit. With the help of the library, the implementation of a Naive Bayes model is greatly simplified.

```
training_set = [(extract_features(d), c) for (d,c) in sentiment_corpus]
classifier = nltk.NaiveBayesClassifier.train(training_set)
```

Data Serving

The sentiment analysis data is made available to the authorized users in AWS S3 and Redshift.

AWS S3

The new data is uploaded to AWS S3 on a daily basis. Typically the analysis data on S3 is one day behind. This is most suitable for consumers who do not require near real time update. S3 is a very robust and low cost object storage solution. Serving data on S3 allows the processing layer be decoupled from the data serving layer which enhances resiliency and security.

AWS Redshift

For the consumers who require near real time update, the data is available in AWS Redshift. Consumers can access the data via a direct ODBC connection. The data lag is usually seconds behind. AWS Redshift is a very high performance storage solution suitable for data with high velocity and volume. It can also be scaled up horizontally.

Concluding Data

Unfortunately, we could not get our project fully operational in time for the 2016 presidential election, thus missing a valuable stream of Twitter data. It is possible to mine past tweets, however this uses methods outside the Twitter API and also would not be in line with the goals of the project (to stream data). Still we were able to analyze a stream of tweets in the aftermath of the 2016 election.

Below are graphs (made in R) showing the number of tweets a candidate received as well as the percentage of those tweets that were viewed as containing a positive sentiment by our sentiment analysis model. The data used is a sample stream taken on 12/04/2016 (well after the 2016 U.S. presidential election).

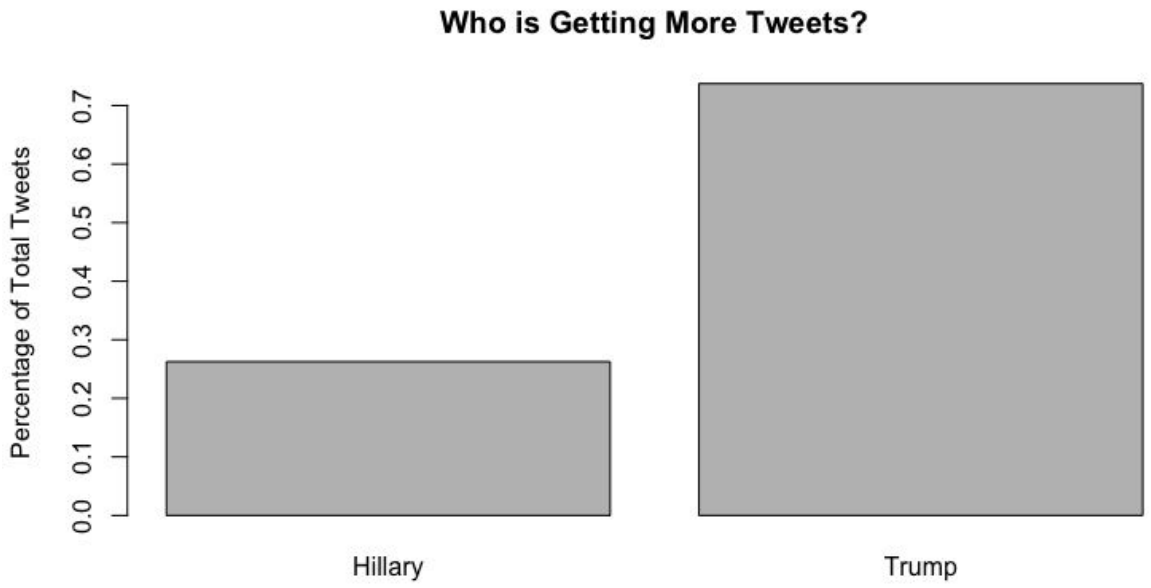


Figure 7: sample graph of twitter data

As would be expected, since Donald Trump has already been decided to be the President, he is currently receiving far more tweets than Hillary Clinton.

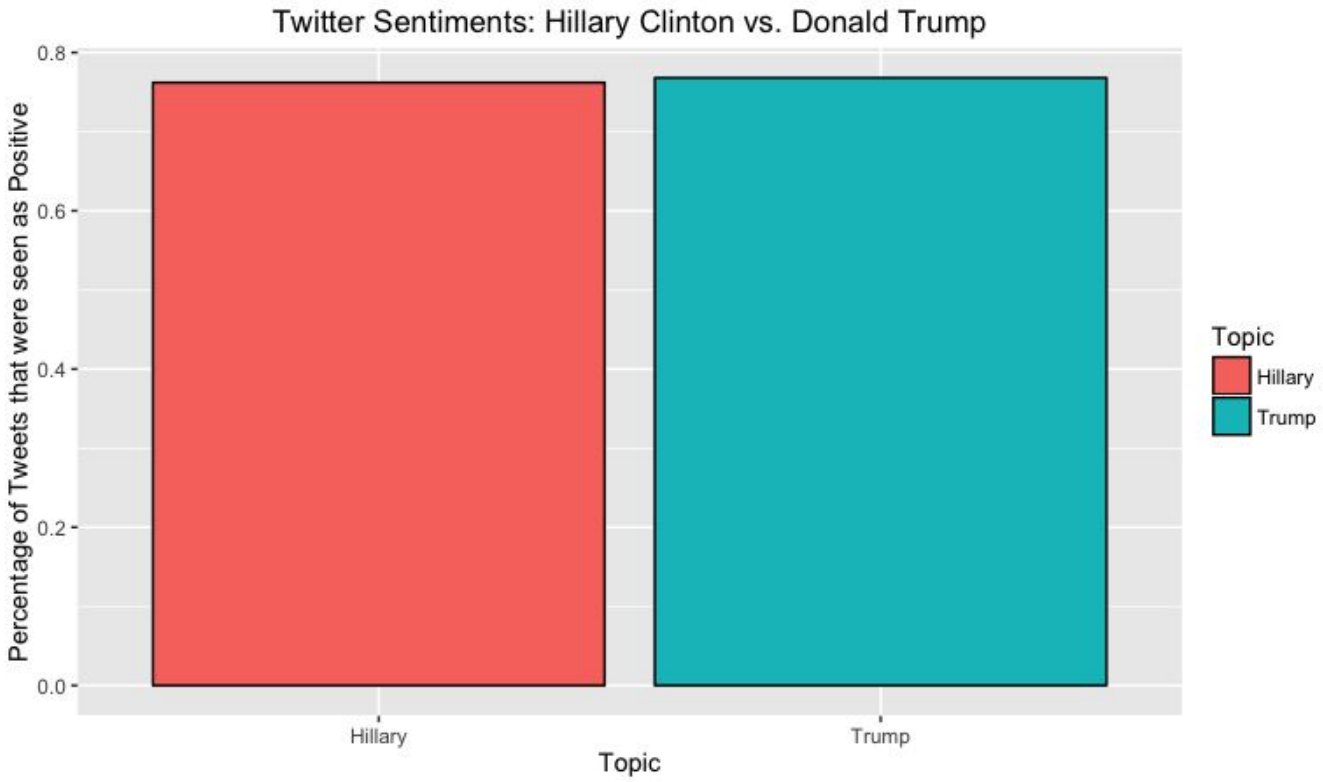


Figure 8: sample graph of sentiment analysis

We see our model rates the percent of positive tweets to be similar for these politicians during the time period of this data.

Consumer Applications

One of the most important aims of this project was to create something that could provide the foundation of future projects for others. In doing so it allows our data to become more meaningful than we could make it by ourselves.

We imagine these applications could manifest in many ways, but as a demonstration we have created a web app using an R package called Shiny. This app is publicly viewable here: https://skarbrevik.shinyapps.io/W205Project/.

Future Enhancements

1. The sentiment analysis process lags behind the data injection. The design to have the data first persisted in HDFS, then have the analysis layer to “pick up” and process the messages can be enhanced by Spark Streaming or Storm, which allows data be streamed directly into the processing layer without an intermediate step.

2. The machine learning algorithm (Naive Bayes) can be further improved by experimenting with different models and better feature selection.

3. As this application was only run over a short period of time during development/testing, scaling volume size was not a major issue. However, because political data gathered today will likely still be useful for analysis of politics even several years from now, ideally, the user of this application will keep it running without a foreseeable termination point. Long term persistence of this application will require considerable storage space in the final storage layer (S3). As S3 will scale horizontally, the implementation takes care of itself, however the financial cost will become considerable over time. To avoid the need to delete older data, one may want to use a cheaper storage solution such as Amazon Glacier ($1 per TB per Month as of 12/6/2016) or initially invest in many large (3+ TB) physical hard drives and create a personal storage server.

4. This application was developed using LinkedIn’s “Camus” platform but this platform is no longer actively supported and thus a future enhancement of our application will be to switch to LinkedIn’s new platform “Gobblin”. A couple improvement claimed by LinkedIn include better performance speeds and more features to monitor jobs as they execute.

5. Create an example real-time consumer application. Currently our example consumer application (an R shiny web application) relies on our batch analysis store (Amazon S3) which is not updated in real time. To use our streaming data to its fullest we would want to demonstrate a use-case that relies on the real-time updates of Amazon RedShift.

For more documentation on the tools used in this project:

Twitter “Hosebird Client” API

Apache Kafka

Camus
Original publication from LinkedIn about Camus

Gobblin - Kafka to HDFS ingestion (not used in this app, but considered alt. to “Camus”)

R shiny (used for example consumer application)