W205 Final Project - Summer 2014
Rahul Bansal
Lisa Kirch
Joe Morales
Christopher Walker

Correlating Stock Prices with Predictions from Twitter

The Problem

Twitter is a well-known messaging platform that allows users to express their sentiments on nearly any topic worldwide. With 271 million active users monthly and 500 million tweets sent daily, a fairly sizable corpus of sentiment is available for analysis. Several companies have products that leverage this sentiment to provide information to investors, and there are a variety of algorithms that may be employed to mine relevant sentiment from Twitter data. One such company is Downside Hedge and their algorithm analyzes tweets made by specifically traders and investors about trades and outlook. This strategy requires knowledge of specific traders and investors and involves targeted searching for Tweets which relate to specific securities. Another company with a slightly different strategy is Dataminr. They use proprietary algorithms to analyze the entire Twitter stream (the "Firehose") looking for Tweets that will turn out to be indicators of breaking news or trends. This information is sold to both financial and news clients.

George Box was a statistician made famous for this memorable quote, "All models are wrong, but some are useful." (http://en.wikipedia.org/wiki/George_E._P._Box). Peter Norvig, the Director of Research at Google, caused a stir in 2008 when he put forth a new version of the Box quote, "All models are wrong, and increasingly you can succeed without them." (http://archive.wired.com/science/discoveries/magazine/16-07/pb_theory). Norvig's assertion is that you don't need a model if you have all the data and that it is becoming increasingly more possible to gather, retain, and analyze *all* of the data. The algorithms described if the first paragraph employ sophisticated models to make financial predictions from Twitter data. Our goal with this project was to use information retrieval (IR) search mechanisms and sentiment analysis to look for correlations between Twitter data and stock performance in the absence of a model; a small test of Norvig's assertion.

The Process

We started by building an IR system containing information about companies, specifically information contained in the companies' annual reports to the SEC. To scope the project, we limited companies to those that are in the Standard and Poor's 500 Index (S&P 500). We gathered unfiltered Tweets over a period of time and determined the relevance of each Tweet by performing a text search against the IR system using words from the Tweet which resulted in a relevance score for each company for each Tweet. Tweets would be relevant to a company if they contained words that were also contained in the company's annual report. These words include the company's name, stock ticker symbol, products, officers' names, and other less obvious information. We also ran a sentiment analysis algorithm on each Tweet, resulting in sentiment score of -1 for the most negative Tweets to + 1 for the most positive Tweets. Multiplying the relevance score by the sentiment score for each Tweet for each company resulted in an overall score that represents the sentiment value for that company at that point in time; we called this score the TweetShift. We analyzed the stream of TweetShift data alongside the stream of market price data, specifically the change in the market price, which we called price shift. In order to scope the analysis, we analyzed aggregated TweetShift and price shift data at 10 minute intervals.

Table 1: Tools and Methods

Purpose	**Tools Used**
Code Base	Github - https://github.com/jmorales4/W205-RJCL
Version Control	SourceTree by Atlassian
Storage	S3 - http://rjcl-tweets.s3.amazonaws.com/ http://rjcl-stockquotes.s3.amazonaws.com/ http://10k-clean-data.s3.amazon.com
Twitter Sentiment Analysis	Python sentiment analysis courtesy Alex Davies http://alexdavies.net/
Search / Relevance Scoring	Apache Solr
Hadoop Platform	Amazon Elastic MapReduce Hadoop streaming using Python
Languages	Python (with tweepy, boto, BeautifulSoup, numpy, and solrpy) Perl R
APIs	Twitter Solr Yahoo Finance
Other	XML JSON Tableau

Twitter Data

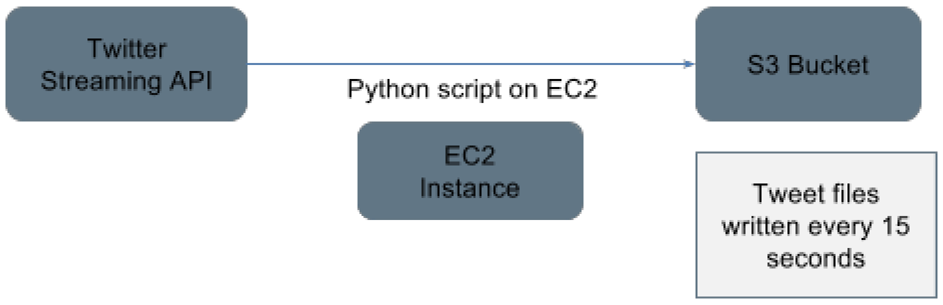


Figure 1: Twitter Data

Twitter data was collected from the Twitter streaming API using the Python library Tweepy (http://www.tweepy.org/). Access to the full Twitter "Firehose" is limited to Twitter partners, so we were only able to retrieve a sample of the full stream, approximately 4000 tweets per minute. A process was run on an Amazon Web Services (AWS) Elastic Cloud Compute (EC2) virtual machine. This process remained subscribed to the sample stream and created a file for every 1000 received Tweets (approximately every 15 seconds). These files were stored on AWS Simple Storage Service (S3). The process ran for approximately three weeks and stored about 97 million Tweets in 97,000 files totaling 269 GB of data. The Tweet files are available at http://rjcl-tweets.s3.amazonaws.com/.

Stock Quote Data

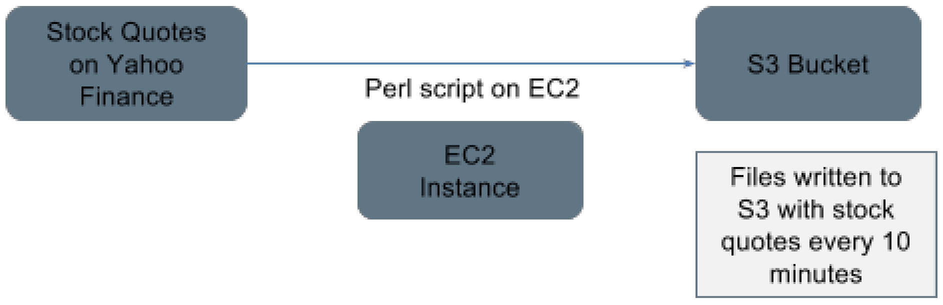

Stock Quotes on Yahoo Finance
Perl script on EC2
S3 Bucket
EC2 Instance
Files written to S3 with stock quotes every 10 minutes

Figure 2: Stock Quote Data

The data flow diagram above shows how we went about collecting stock quote data for our project. We developed a Perl script that utilized the Yahoo API to download stock quotes every 10 minutes for the S&P 500 tickers. The Perl script was deployed on an EC2 instance and wrote files to an S3 bucket every 10 minutes. Each file was approximately 82 KB and we generated 144 files per day for 2 weeks, resulting in 2016 files or 161.4 MB of data. The files were pipe-delimited text files with one row per stock and we captured data elements such as the stock ticker, current price, price change, volume, last trade time, bid price, ask price, and the date's range. For the purposes of this analysis, we mostly used the current price and the price change.

A listing of the files that were generated through this process can be found at - http://rjcl-stockquotes.s3.amazonaws.com/

Annual Report (10-K) and Company Data

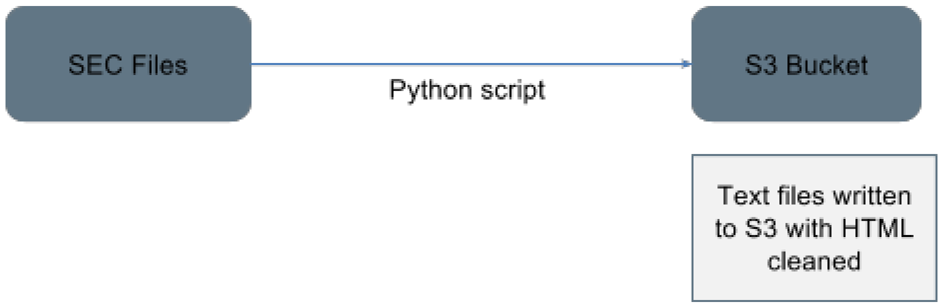


Figure 3: Annual Report (10-K) Data

We started with the Wikipedia list of the S&P 500 companies. This list includes the ticker symbol, security name, links to SEC filings over in EDGAR at the Securities Exchange Commission, the Global Industry Classification Standard (GICS) Sector and Sub Industry, Headquarters Address, Date First Added, and the Central Index Key (CIK). Unfortunately, due to the way the SEC website is set up, it was a manual process to collect each of the annual report (10-K) files for each company in the S&P 500. File names and CEO information were appended to this data. No 10-Ks were available in the EDGAR database for Navient Corp (NAVI) or Perrigo (PRGO). SEC files are available in XBRL format or HTML format. Since not all data is required to be filed in the XBRL format, we opted to use the complete submission text file. The text files contain HTML and in some cases binary data which had to be cleaned. In order to clean the data, a python script was written that employed the BeautifulSoup package to strip off the HTML. Originally, we were going to convert the files to JSON, but found it would be more work than just cleaning the text files.

The files were run through BeautifulSoup twice to get them mostly clean. Once the files were cleaned by the script, they were FTPed to an S3 bucket over at AWS, http://10k-clean-data.s3.amazon.com.

With the cleaned documents ready, we needed a place to store them. We installed Apache Solr on an Amazon EC2 instance, using a freely available public AMI from Bitnami. The Solr instance was configured to treat the various meta data elements (CIK, ticker symbol, etc.) and the full 10-K document text as separate facets. In the beginning, we considered incorporating the different facets in our search and relevance scoring strategy. This excerpt from the *schema.xml* configuration file shows the definitions for the various facets:

Figure 4: schema.xml Excerpt for Facet Definitions

```
	<field name="_version_" type="long" indexed="true" stored="true"/>
	<field name="cik" type="text_general" indexed="true" stored="true"/>

	<field name="ticker_symbol" type="text_general" indexed="true" stored="true"/>
	<field name="company_name" type="text_general" indexed="true" stored="true"/>
	<field name="sector" type="text_general" indexed="true" stored="true"/>
	<field name="sub_industry" type="text_general" indexed="true" stored="true"/>
	<field name="address" type="text_general" indexed="true" stored="true"/>
	<field name="date_first_added" type="text_general" indexed="true"
stored="true"/>
	<field name="html_file_name" type="text_general" indexed="true" stored="true"/>
	<field name="xbrl_file_name" type="text_general" indexed="true" stored="true"/>
	<field name="ceo_name" type="text_general" indexed="true" stored="true"/>
	<field name="ten_k_text" type="text_general" indexed="true" stored="true"/>

	<uniqueKey>cik</uniqueKey>
```

With the facets established, we wrote a Python ingestion script that used the *solrpy* library to communicate with the Solr API to post documents to the collection.

Figure 5: Excerpt of Python ingestion script for posting documents to the collection

```
        ten_k_url = 'http://10k-clean-data.s3.amazonaws.com/' + item['html_file_name']
        ten_k_url = ten_k_url.replace(".txt", "-clean.txt")
        print ten_k_url

        try:
                response = urllib2.urlopen(ten_k_url)
        except Exception, e:
                print "ERROR: Failed to retrieve 10-K document for " +
item['company_name'] + "(" + ten_k_url + ")"
                continue

        html = response.read()
        soup = BeautifulSoup(html)
        item['ten_k_text'] = ''.join(soup.findAll(text=True)).strip()
        item['ten_k_text'] = item['ten_k_text'].replace("\0", " ")
        item['ten_k_text'] = ''.join(s for s in item['ten_k_text'] if s in
string.printable)

        try:
                s.add(item)
                s.commit()
        except Exception, e:
                print "ERROR: Failed on add document to Solr for " + item['company_name']
                continue
```

During our exploration of different search strategies, we tried assigning different weights for the search facets using Solr's relevance score boosting settings. The excerpt below, from our *solrconfig.xml* configuration file, shows some of the boosting settings we tried during these tests. In this iteration, we gave the highest weight to facets that were most likely to be unique identifiers for a company (CIK, ticker symbol, and company name).

Figure 6: solrconfig.xml Excerpt for Relevance Score Boosting

```
<requestHandler name="/query" class="solr.SearchHandler">
   <lst name="defaults">
     <str name="echoParams">explicit</str>
     <str name="wt">json</str>
     <str name="indent">true</str>
     <str name="defType">edismax</str>
     <str name="qf">
        cik^5.0 ticker_symbol^5.0 company_name^5.0 sector^0.3 sub_industry^0.5
        address^0.2 date_first_added^0.01 html_file_name^0.1 xbrl_file_name^0.1
        ceo_name^3.0 ten_k_text^3.0
     </str>
     <str name="df">ten_k_text</str>
   </lst>
</requestHandler>
```

Unfortunately, tweets are not easily parsed into these types of facets. While we might have employed a complicated query syntax to test each tweet against all possible facets, the *ten_k_text* field already contained all of the same information that was present in the other facets. The information was held within a much larger body of text, but the test searches that we tried returned appropriate results when searching for things such as CEO names.

We then began testing actual tweets as queries. Solr initially treated our queries as stricter "and" searches, requiring the presence of all search terms. As a result, most of the test Tweets we tried as queries returned no results. We adjusted our code to produce "or" queries (as shown below), with much better results.

```
ten_k_text:diet OR ten_k_text:coke OR ten_k_text:right OR ten_k_text:meow
```

In the proof-of-concept phase, we used a *t2.micro* sized EC2 instance because they were available to run at no cost. Java consistently ran out of memory on these instances after a few searches, and it was clear that they would not keep up with our planned query volume. We migrated the collection to an *m3.large* instance, allocated substantially more memory to the Java runtime, and tested with an Elastic MapReduce cluster of 10 machines. Once we verified that Solr's performance was satisfactory, we extended to 20 machines for the remainder of the MapReduce processing.

Our Solr document collection is available for simple browsing and searching. Our primary goal was to produce good quality search results with appropriate relevance scores, so many of the user interface options may not work as expected. However, the search query handling and relevance scoring are the same as those used in our final iteration of the project. Source code for our query testing Python script is also available in our project Github repository (*sectenk_query.py*).

m3.large instance:
http://ec2-54-187-252-24.us-west-2.compute.amazonaws.com:8983/solr/sectenk/browse

t2.micro instance:
http://ec2-54-186-141-116.us-west-2.compute.amazonaws.com:8983/solr/sectenk/browse

Evaluating and Visualizing Results

Once all of our data was collected, we made a conscious decision to limit our analysis to one week of data, the week of August 4th (0000 GMT) through August 8th (0000 GMT). The final dataset for analysis consisted of a CSV file with the eight columns defined below:

1. Datetime - this was every 10-minute interval in a YYYY-MM-DD HH24:MI:SS format
2. Ticker - the company stock ticker symbol
3. Stock Price - the stock price at the given time interval
4. Price Shift - the actual price shift after the 10-minute interval
5. Tweet Shift - the estimated shock shift as a calculation of tweet relevance to the given company and the sentiment score of the given tweet
6. Tweet Volume - the volume of tweets relevant to the company within the 10-minute interval
7. Text of Most Relevant Tweet - the text of the tweet with the highest relevant score for the company within the 10-minute interval
8. Score of the Most Relevant Tweet - the relevant-sentiment combination score of the most impactful tweet within the 10-minute interval

We performed our exploratory and statistical analysis in Tableau and R. During the week of August 4th, here are the top 10 S&P 500 stocks that were Tweeted about per our Solr search algorithm.

Table 2: Top 10 S&P 500 Stocks Tweeted

Ticker	**Tweet Volume**
PPL	3733
SJM	1497
T	1018
GME	855
TSN	789
VAR	609
EA	606
TWX	603
CCL	550
MAT	547

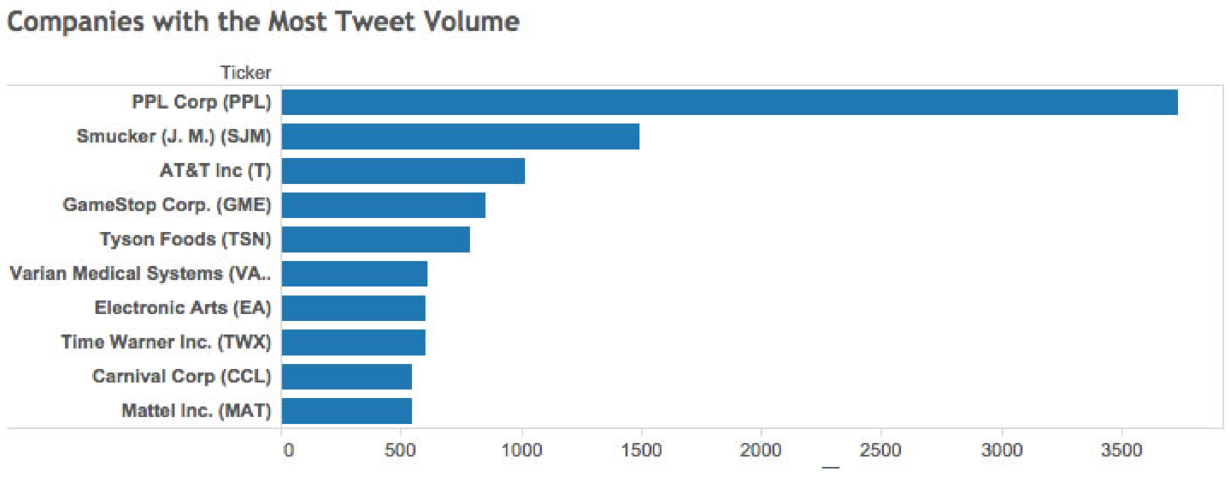


Figure 7: Top 10 S&P 500 Stocks Tweeted

We also took a look at the top 10 tweets with the highest relevancy scores to the companies in our database and they all matched against SJM which is the JM Smuckers Jam Company. All of these tweets had the word ‘empty’ in them signifying that ‘empty’ was a major part of the Smuckers 10-K report. Here are a listing of the top 10 tweets in our dataset.

Table 3: Top 10 Tweets

Company	**Datetime (GMT)**	**Tweet Text**	**Relevance Score**
SJM	8/4/2014 1:00 AM	Empty .	1.5580
SJM	8/4/2014 7:40 AM	empty	1.5580
SJM	8/5/2014 5:20 AM	EMPTY ! :))	1.5580
SJM	8/5/2014 7:00 AM	(Empty)	1.5580
SJM	8/6/2014 7:40 AM	empty.	1.5580
SJM	8/6/2014 8:50 PM	Empty	1.5580
SJM	8/7/2014 2:20 PM	Empty me	1.5580
SJM	8/5/2014 3:20 AM	And empty	1.5458
SJM	8/5/2014 4:20 PM	So empty.	1.5209
SJM	8/6/2014 5:50 AM	So empty	1.5209

In addition to these top 10 charts, we looked at some summary statistics of the relevancy and tweet shift scores. Recall, the tweet shift score is a score that combines the relevancy of the tweet to a given company and the sentiment score of a given tweet.

Table 4: Summary Statistics for Tweet Shift

Min	1st Qtl	Median	Mean	3rd Qtl	Max
.10	.11	.13	.19	.19	2.74

First thing to note here is that the tweet shift is never negative. We found that our sentiment algorithm never generated enough negative tweets to ever let a 10-minute interval of tweet scores go below zero. Next, we sought to correlate tweet shifts with actual price shifts in the market on a 10-minute interval. When running the correlation test, we actually get a non-significant p-value (p= .81) indicating that we cannot reject the null hypothesis that the correlation is indeed zero. The correlation coefficient we get is r = .0028 which has very little practical significance. This result is supported by the graph below showing no real relationship between our tweet shift metric and actual price changes in the market.

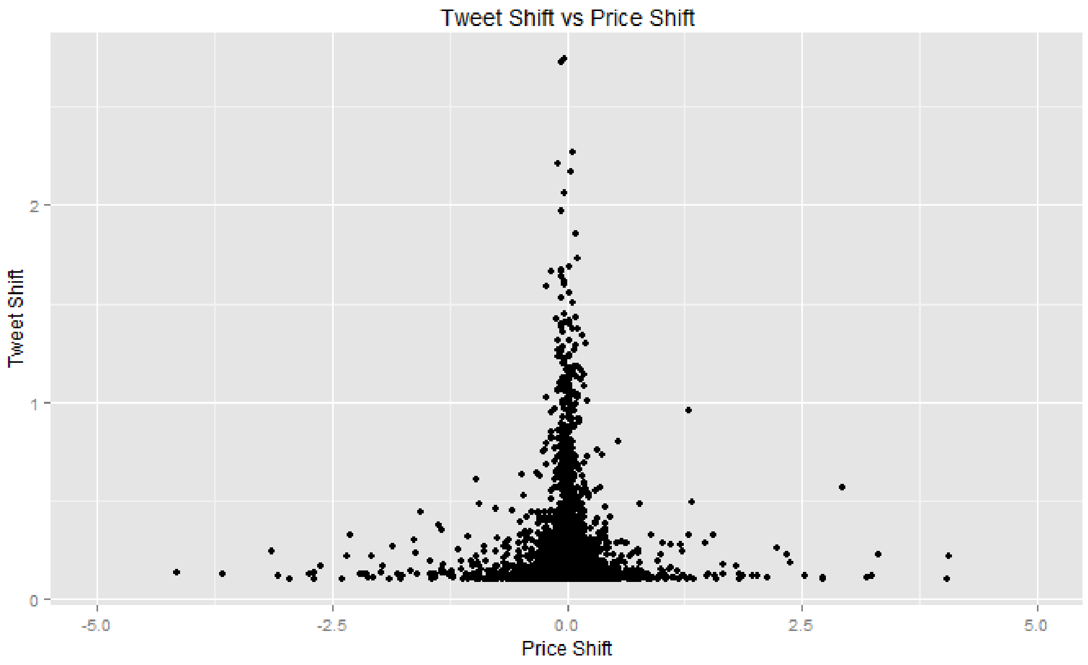


Figure 8: Tweet Shift vs Price Shift

After looking at the data at the 10-minute interval level, we decide to roll up the price changes and the predicted tweet changes at the hour-level to see if reducing the noise would produce a stronger correlation. However, we found that the results were equally scattered if not more.

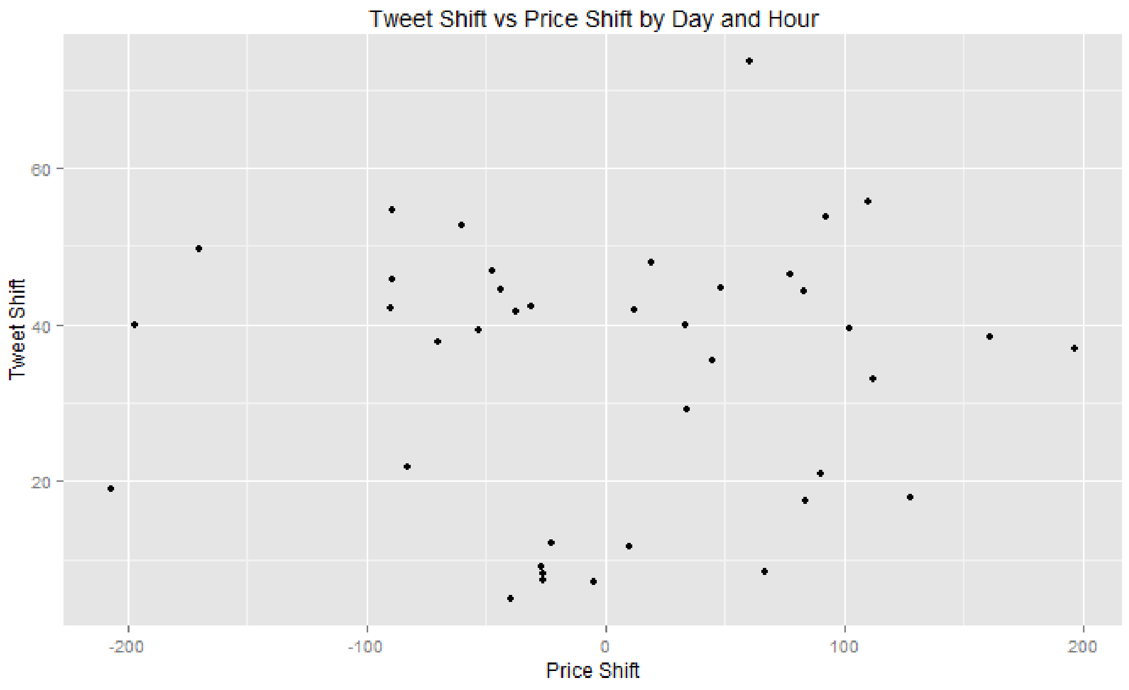


Figure 9: Tweet Shift vs Price Shift by Day and Hour

The hour-level continued to produce a non-significant correlation coefficient of r = .02. Finally, we decided to pull up at the day-level but to look at each stock on each day separately. The results from this analysis were similar actually to the results from the first analysis at the 10-minute interval level. This relationship produced a correlation coefficient of r=-0.014 (see graph directly below). The final relationship we tested before manipulating any data was at the overall day-level with all stocks rolled up. Here, since we only had 5 days of quotes data, we were really only working with 5 data points so the correlation we achieved is really meaningless with this small of a sample size. Regardless, the overall day-level numbers produced a significant correlation with r = -.895. This means that, at the day-level, the more positive tweets are about S&P 500 companies, the worse they will do on that given day.

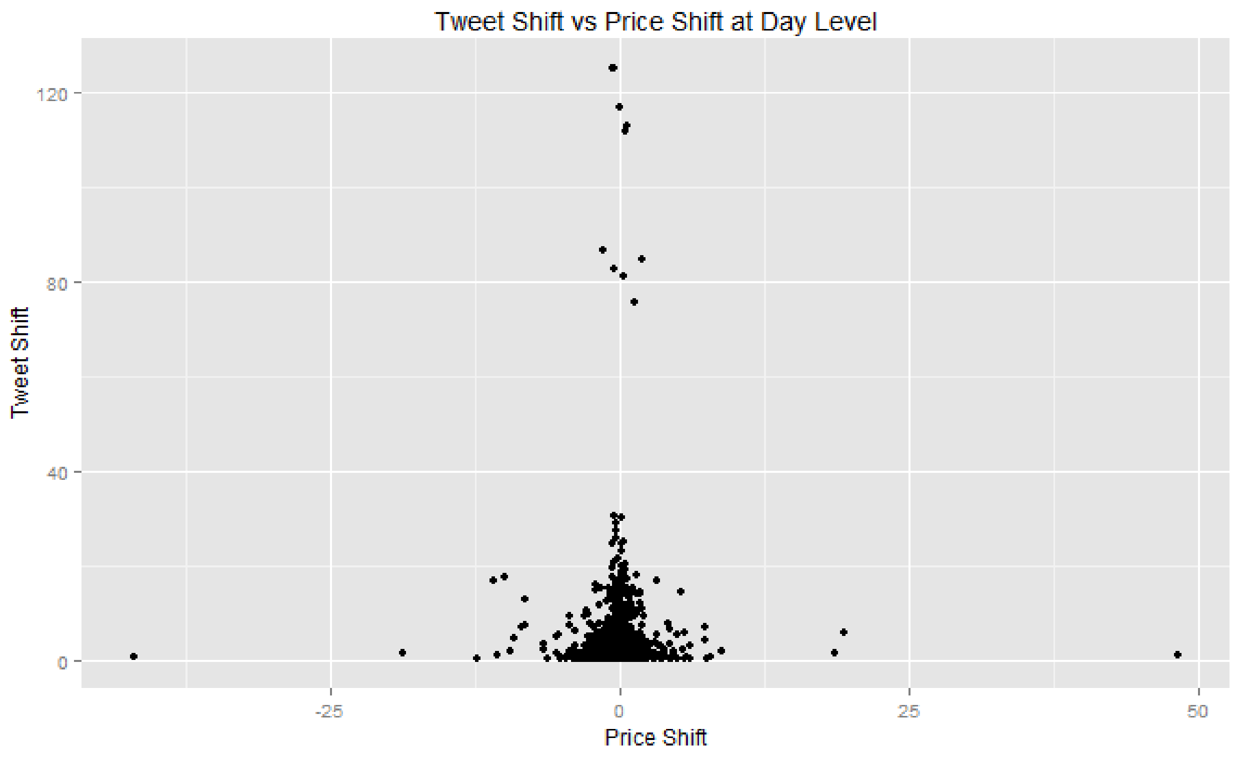


Figure 10: Tweet Shift vs Price Shift at Day/Stock Level

The last thing we decided to check was whether we could find a relationship in the data at the 10-minute interval level if we did a 10-minute time shift. That is, we correlated stock price shifts 10 minutes after the tweet shifts. Here we were trying to test if there was a lag in the time it took for the stock market to react to mentions in the Twitter stream.

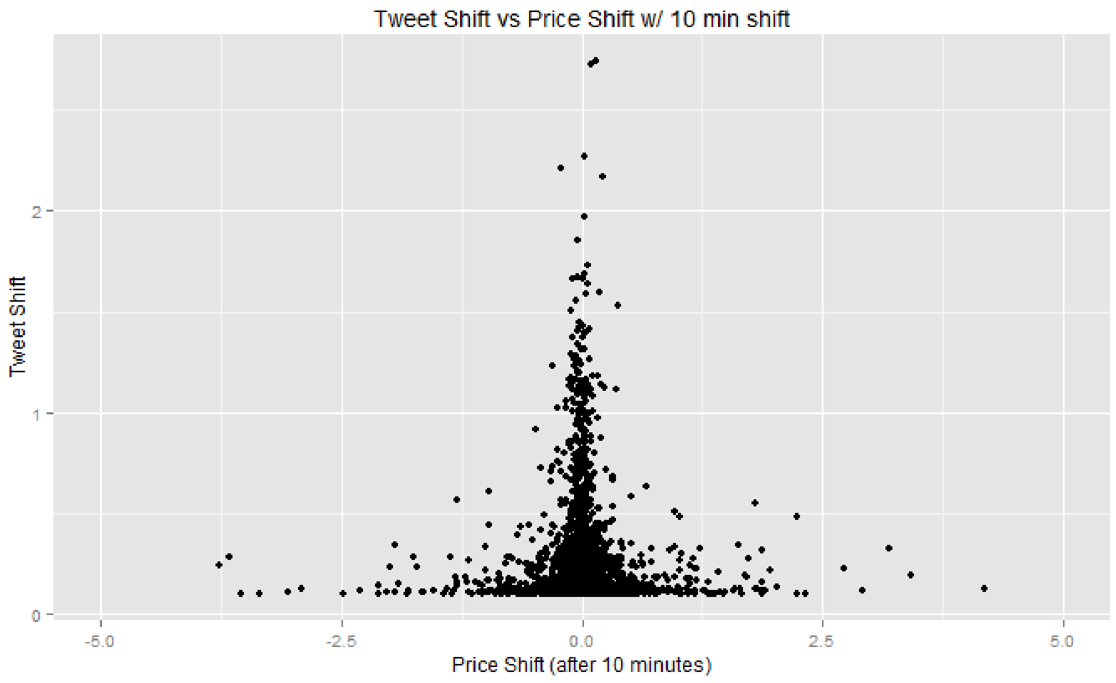


Figure 11: Tweet Shift vs Price Shift with 10 Minute Time Shift

The above graph, again, shows no real correlation even with this time shift (r = -0.12, p = .34) So, in sum, we did not find any real relationships between stock price changes and the Twitter predictions we generated through a combination of sentiment scoring and company relevance scoring for the S&P 500. There is some promise at the day-level of showing a meaningful negative correlation but we need to collect a lot more data to determine if we can conclude anything from that.

We eliminated the data for WDC, ACT, AXP, NWSA, GMCR, EL, and EIX from our analysis as we noticed that these particular files contained stray &gt; HTML entities, which were being picked up as positive sentiment even though they were truly just HTML code.

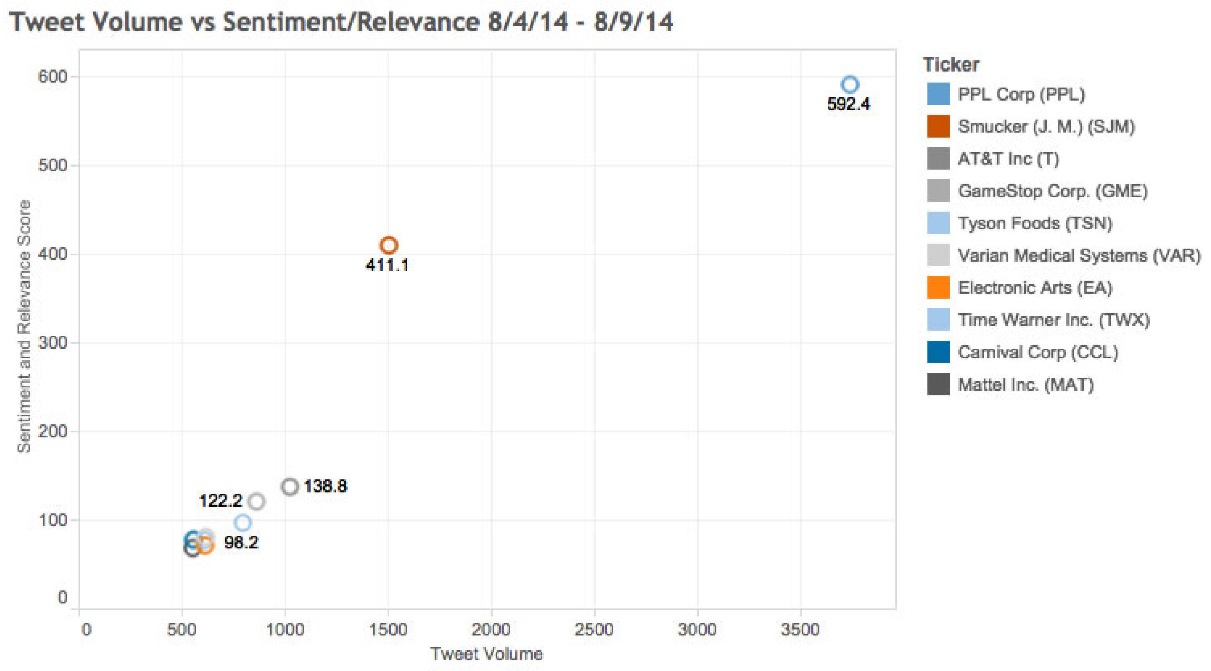
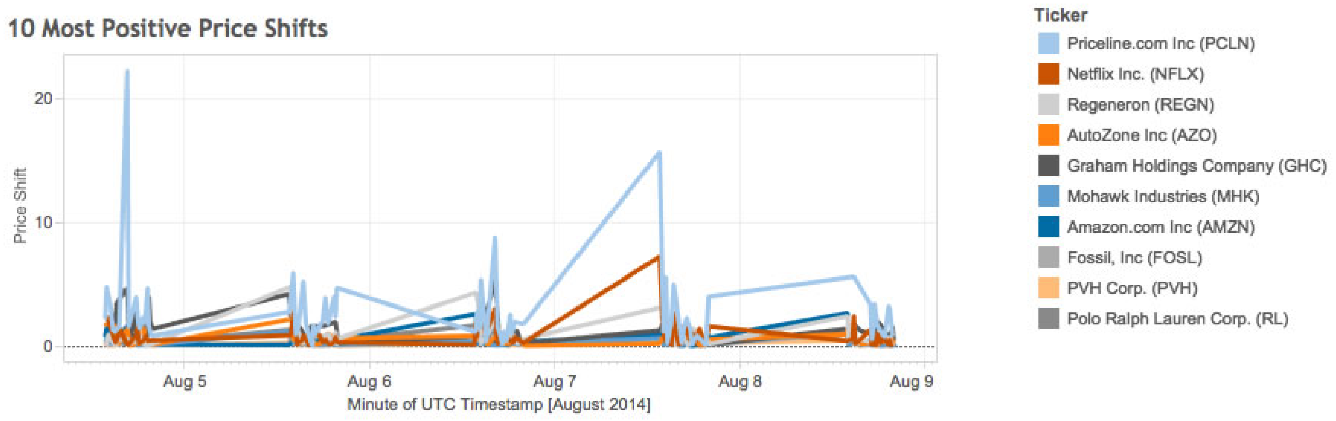


Figure 12: Tweet Volume vs Sentiment/Relevance

Above are the top 10 most tweeted companies and the sum of their sentiment/relevance score. PPL Corp (PPL) and Smucker (SJM) stand out as being markedly higher in their cumulative sentiment/relevance score. Upon further investigation when looking at the text of the tweets, we got false positives on PPL since some people use “ppl” as a text abbreviation for people. In the case of Smuckers, they had the word “empty” repeatedly throughout their HTML code in their style sheet comments, thus again, another false positive.

Figure 13: 10 Most Positive Price Shifts



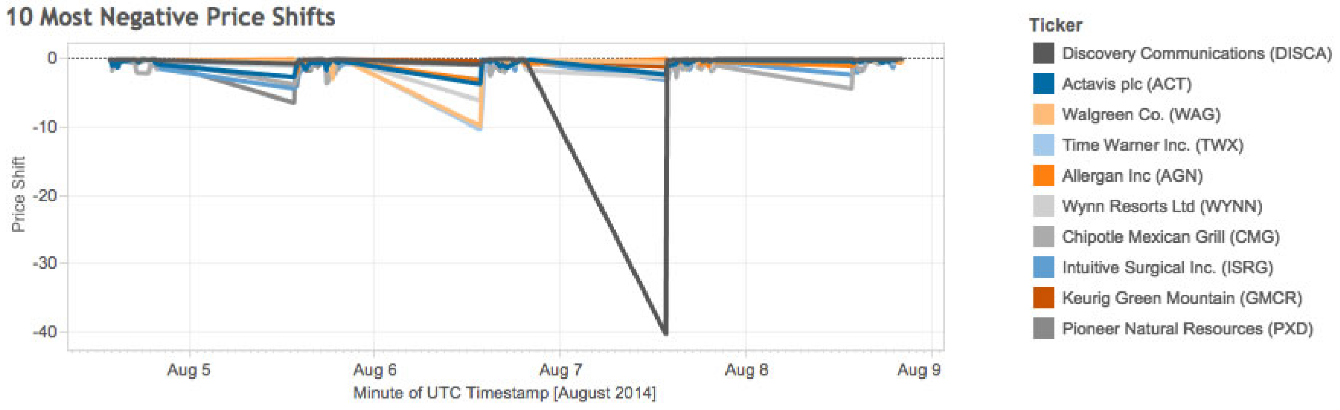
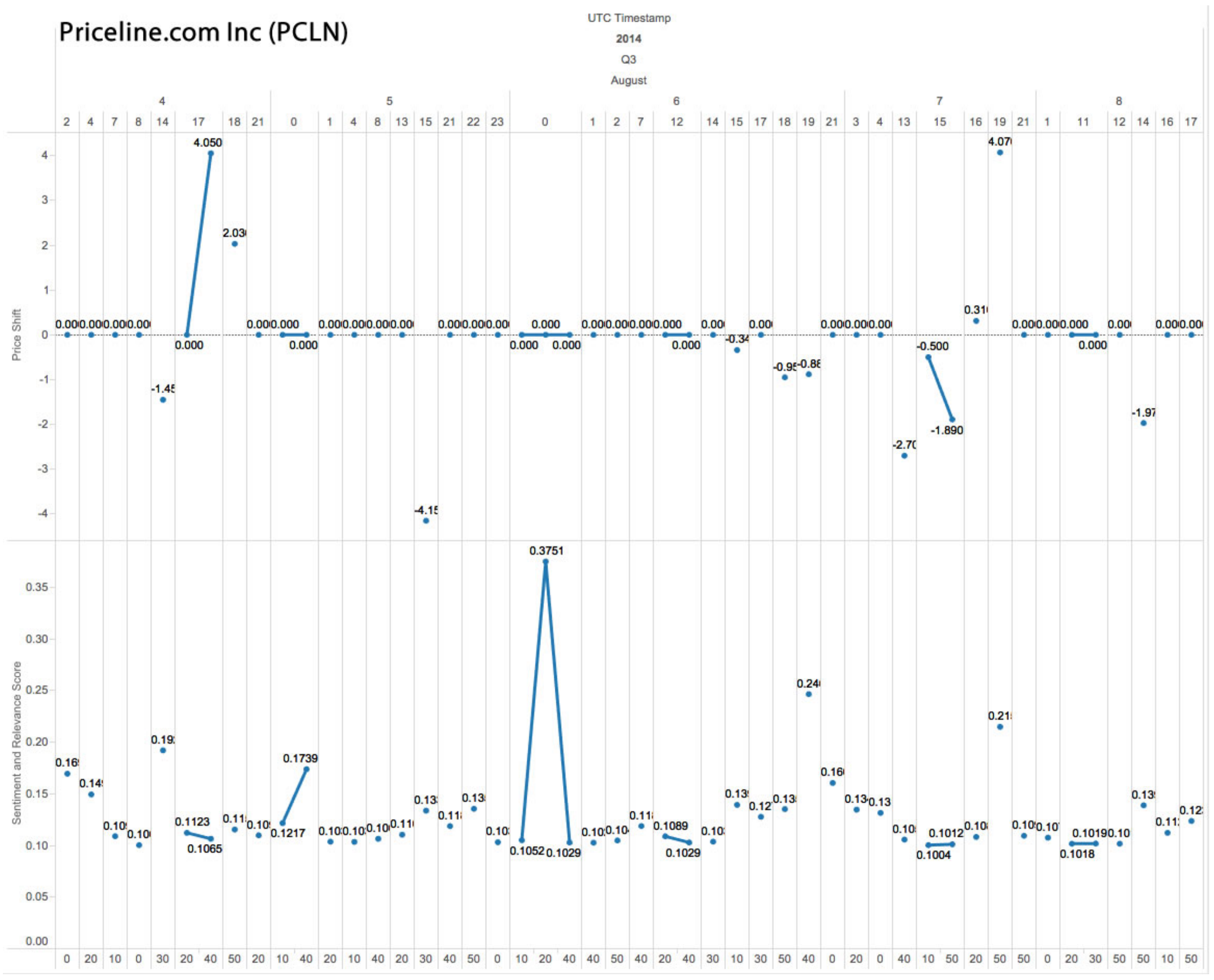


Figure 14: 10 Most Negative Price Shifts

We examined the top 10 stocks with the greatest positive and negative price shifts (these were not converted to percentage of stock price to see relative stock price shifts) and found that Priceline.com had the largest positive price shift. Reviewing the news and press releases surround Priceline.com on August 5th, we found that Priceline.com released Q2 earnings results on August 8th and saw a spike early in the week due to increased August volatility in the S&P 500 in general as well as outlook on earnings. Discovery Communications, Inc. (DISCA) released their financials online to the SEC on August 8th. Although Bernie Ecclestone settled with a Munich court by writing a $100 million check in return for ending his alleged bribery to maintain control of Formula One trial midweek, Discovery Communications was looking to buy out private equity group CVC's shares in Formula One (a 49% stake). This large investment and negative press may have pulled down the price of Discovery Communications. In addition, Time Warner plunged on August 6th which may also have dragged the price of Discovery Communications down and often there is price volatility prior to an earnings release.

Figure 15: Priceline.com (PCLN) Price Shifts and Corresponding Sentiment/Relevance Score Shifts Over the Week of August 4 - 9, 2014



When looking at Priceline.com’s price shifts and corresponding tweet sentiment/relevance, we see that there does not appear to be any correlated pattern between the two.

Lessons Learned and Recommendations for Future Work

- Number of tweets captured were a small part of the Twitterverse: the theory that we were testing was that you don't need a model if you have all of the data, but we didn't have all of the data. In the absence of a connection to the full Twitter Firehose, we believe that target Tweet searches based on sender, content, and other metadata would produce better results.
- Sentiment package used did not produce reliable scores: it was observed that the package produced significantly more positive sentiment results than negative sentiment results. It is possible that there are in fact significantly more positive Tweets than negative Tweets, but a sampling revealed that some Tweets that were graded as positive in fact seemed negative. Furthermore, the few negative Tweets that existed seemed much less likely to result in matches against the Solr 10k IR system, resulting in low relevance and subsequent filtering.
- Solr relevance scores did not always make intuitive sense: for example, searching for Big Mac yielded McDonald's as the third most relevant company with a relevance score that was <1% of the relevance score of the top relevant company, Macerich. While explainable by Macerich's ticker symbol (MAC), this does reveal a flaw in the method.
- Markets open 9:30 AM - 4 PM ET, but people tweet 24 hours a day: this results in an inability to reliably test correlations below the day level. One better way to capture correlations would be to perhaps include Asian market data.
- Due to time and processing constraints, we only have one week of stock quote data processed. Perhaps trends would become more apparent over a longer period of data.
- Idiosyncrasies in the Twitterverse sometimes produce unexpected results: for example, a large number of Tweets abbreviated "people " as "ppl" resulting in false relevance to PPL Corp.

Appendix A: Source code

The following code is available at https://github.com/jmorales4/W205-RJCL-Public

Phase 1: Data Collection

twitterStream.py
Connect to the Twitter Stream API using Tweepy, create 1000 tweet files, and store in Amazon S3 using Boto

get_stock_quotes.pl
Perl Script that reads Stock Quotes from Yahoo Finance API and writes them to AWS S3 every 10 minutes

clean-secdata-bs4.py
Python script that uses BeautifulSoup to clean HTML from SEC 10K files

sectenk_ingest.py
Retrieve SEC 10k files from S3 and import into Solr

Phase 2: Map Reduce

mapStockPrices.py
Map file used by Hadoop: input = stock quote file, output key = time rounded down to nearest 10 minutes, output val = csv of ticker|price pairs

redStockPrices.py
Reduce file used by Hadoop: input = stock prices as ticker|price pairs, keyed by time, output = time, ticker, value csv lines

mapTweets.py
Map file used by Hadoop: input = tweet files containing 1000 tweets
- score sentiment. Tweets with neutral sentiment are discarded (abs(sent) < 0.25)
- score relevance against the 10k Solr system. Discard low hits (relevance < 0.1)
- calculate the tweetShift score for each relevant ticker = sentiment * relevant

output key = time, output val = csv of tweet + ticker|tweetShift pairs

redTweets.py
Reduce file used by Hadoop: input = ticker|score pairs, keyed by time (10 minute periods),
output = time, ticker, aggregated sum of tweetShift, aggregated count of Tweets, and the text and tweetShift of the most significant Tweet for the period

Phase 3: Dataset Construction

tenMinutePrices.py

Read Hadoop StockPrices output and produce individual csv files for each Ticker/10 minute period containing time, ticker, and price

priceShift.py

Read in individual csv files containing price data and calculate the priceShift by comparing the price to the previous price. Also and fill in missing points (e.g. when markets were closed) by copying previous points.

tweetShift.py

Read Hadoop Tweets output and apply tweetShift to existing priceShift files.

makeOneFile.py

Gather up all the individual files produced above and combine them into a single csv file.

Phase 4: Analysis

analysis_final.R

Correlate stock price changes with predictions from Twitter

```
# ! /usr/local/bin/python
#
# twitterStream.py: Connect to the Twitter Stream API using Tweepy,
# create 1000 tweet files, and store in Amazon S3 using Boto
#

from tweepy.streaming import StreamListener
from twitterCredentials import *
from awsCredentials import *
from time import gmtime, strftime
from boto.s3.connection import S3Connection
from boto.s3.key import Key
import sys
import os

class TwitterStream(StreamListener):
    file = None
    count = 0
    s3 = None

    def __init__(self):
        StreamListener.__init__(self)
        self.s3 = S3Connection(aws_access_key, aws_secret_key)
        self.start_new_tweet_file()


    def start_new_tweet_file(self):
        old_file = self.file

        filename = strftime("%Y%m%d%H%M%S.tweets", gmtime())
        self.file = open(filename, "w")
        self.count = 0

        print filename

        if old_file is not None:
            old_file.close()
            self.post_to_s3(old_file.name)

    def on_data(self, data):
        try:
            self.file.write(data[:-1])
            self.count += 1
            if self.count >= 1000:
                self.start_new_tweet_file()
            return True

        except Exception, e:
            pass
    def post_to_s3(self, filename):
        bucket = self.s3.get_bucket('rjcl-tweets')
        entry = Key(bucket)
        entry.key = filename
        b = entry.set_contents_from_filename(filename)
        print "{}: wrote {} bytes to s3".format(filename, b)
        os.remove(filename)

if __name__ == '__main__':
    print "Startup"
    while True:
        try:
            stream1 = tweepy.Stream(auth, TwitterStream())
            stream1.sample()
        except:
            print "Unexpected error:", sys.exc_info()[0]

    print "Shutdown"
```

```
#
# get_stock_quotes.pl: Perl Script that reads Stock Quotes from Yahoo Finance API and writes them to AWS S3
every
# 10 minutes
#
use Finance::YahooQuote;
use Amazon::S3;
use strict;

### Global params
$Finance::YahooQuote::TIMEOUT = 60;
my $aws_access_key_id = '';
my $aws_secret_access_key = '';

### Open S3 connection
my $s3 = Amazon::S3->new(
{
        aws_access_key_id     => $aws_access_key_id,
        aws_secret_access_key => $aws_secret_access_key,
        retry                 => 1,
}
);
my $bucket = $s3->bucket('rjcl-stockquotes');

### read in stock ticker symbols of interest
open INPUT, "stock_tickers.txt";
my @tickers;
foreach (<INPUT>)
{
        chomp($_);
        push(@tickers,$_);
}

close INPUT;

while(1)
{
        my $date_string = get_date_string();
        my $filename = "stock_quotes_".$date_string.".txt";
        my $filepath = "//home//ubuntu//rjcl_stockquotes//$filename";
        open OUTPUT, "> $filepath";
        print OUTPUT "0 Symbol|1 Company Name|2 Last Price|3 Last Trade Date|4 Last Trade Time|5
Change",
          "|6 Percent Change|7 Volume|8 Average Daily Vol|9 Bid|10 Ask|11 Previous Close|12 Today's Open",
          "|13 Day's Range|14 52-Week Range|15 Earnings per Share|16 P/E Ratio|17 Dividend Pay Date",
          "|18 Dividend per Share|19 Dividend Yield|20 Market Capitalization|21 Stock Exchange\n";
        foreach my $symbol (@tickers)
        {
                my @quote = getonequote $symbol; # Get a quote for a single symbol
                print OUTPUT join("|",@quote),"\n";
        }
        close OUTPUT;

        $bucket->add_key_filename(
                $filename,$filename,
                {   content_type        => 'text/plain'}
                );
        sleep 600;
}

exit();

sub get_date_string
{
        my @date = localtime;
        my $day = $date[3];
        my $month = $date[4]+1;
        my $year = $date[5]+1900;
        my $hour = $date[2];
        my $min = $date[1];
        my $sec = $date[0];
        $day = "0".$day if(length($day) == 1);
        $month = "0".$month if(length($month) == 1);
        $hour = "0".$hour if(length($hour) == 1);
        $min = "0".$min if(length($min) == 1);
        $sec = "0".$sec if(length($sec) == 1);
        my $date_string = $year.$month.$day.$hour.$min.$sec;

        return $date_string;
}
```

```python
#! /usr/local/bin/python
__author__ = 'lkirch'
#
# clean-secdata-bs4.py: Python script that uses BeautifulSoup to clean HTML from SEC 10K files
#

import codecs
import os

inputDir = 'data/'
outputDir = '../clean/'

from bs4 import BeautifulSoup, Tag

# Vasilis http://stackoverflow.com/questions/753052/strip-html-from-strings-in-python
def stripHtmlTags(self, htmlTxt):
    if htmlTxt is None:
        return None
    else:
        return ' '.join(BeautifulSoup(htmlTxt).findAll(text=True))

if __name__ == '__main__':
    for root, dirs, files in os.walk('/Users/lkirch/PycharmProjects/convertSECdatatoJSON/data/'):
        for name in files:
            print("Processing Input File: " + name)
            outputFile = name.rstrip('.txt') + '-clean.txt'
            print("Output File Will Be: " + outputFile)
            with codecs.open('data/' + name, 'r', encoding='utf-8') as sec_file, \
                    codecs.open('clean/' + outputFile, 'w+', encoding='utf-8') as clean_file:
                clean_data = stripHtmlTags(sec_file, sec_file)
                clean_data2 = stripHtmlTags(clean_data, clean_data)
                clean_file.write(clean_data2)
```

```
#
# sectenk_ingest.py:  Retrieve SEC 10k files from S3 and import into Solr
#

import solr
import simplejson
from BeautifulSoup import BeautifulSoup
import urllib2
import string


def clean_strip(item):
    item['cik'] = str(item['cik'])

    for key in item:
        item[key] = unicode(item[key]).strip()
        item[key] = ''.join(s for s in item[key] if s in string.printable)

    return item


# Start by wiping the existing document catalog in Solr
urllib2.urlopen(

'http://ec2-54-186-141-116.us-west-2.compute.amazonaws.com:8983/solr/sectenk/update?stream.body=<delet
e><query>*:*</query></delete>')
urllib2.urlopen(

'http://ec2-54-186-141-116.us-west-2.compute.amazonaws.com:8983/solr/sectenk/update?stream.body=<com
mit/>')


# Create a connection to a solr server
s = solr.Solr('http://ec2-54-186-141-116.us-west-2.compute.amazonaws.com:8983/solr/sectenk')

f = open('companies.json', 'r')
json_data = simplejson.loads(f.read())
f.close

counter = 0
for item in json_data:
    print 'Importing: ' + item['company_name']

    item = clean_strip(item)
    counter += 1


    # Grab 10-K from Lisa's S3 bucket
    ten_k_url = 'http://10k-clean-data.s3.amazonaws.com/' + item['html_file_name']
    ten_k_url = ten_k_url.replace(".txt", "-clean.txt")
```

```
print ten_k_url

try:
   response = urllib2.urlopen(ten_k_url)
except Exception, e:
   print "ERROR: Failed to retrieve 10-K document for " + item['company_name'] + "(" + ten_k_url + ")"
   continue

html = response.read()
soup = BeautifulSoup(html)
item['ten_k_text'] = ''.join(soup.findAll(text=True)).strip()
item['ten_k_text'] = item['ten_k_text'].replace("\0", " ")
item['ten_k_text'] = ''.join(s for s in item['ten_k_text'] if s in string.printable)

try:
   s.add(item)
   s.commit()
except Exception, e:
   print "ERROR: Failed on add document to Solr for " + item['company_name']
   continue
```

```
#!/usr/bin/env python
#
# mapStockPrices.py: Map file used by Hadoop
# input = stock quote file
# output key = time rounded down to nearest 10 minutes
# output val = csv of ticker|price pairs
#
import sys
from datetime import datetime, timedelta

prices = {}

input = sys.stdin
for line in input:
    try:
        # StockQuote schema and sample line
        #
        # 0 Symbol|1 Company Name|2 Last Price|3 Last Trade Date|4 Last Trade Time|5 Change|6 Percent Change|
...
        # A|Agilent Technolog|56.07|8/1/2014|4:04pm|-0.02|-0.04%|1601356|1808860|N/A|N/A|56.09|55.77| ...

        line = line.strip()
        words = line.split('|')
        if len(words) < 5: continue  # not enough data in this line, ignore

        ticker = words[0]
        if ticker == '0 Symbol': continue  # header row

        price = words[2]

        t = datetime.strptime(words[3] + ' ' + words[4], '%m/%d/%Y %I:%M%p')
        nearest10Minutes = (t.minute / 10) * 10  # time rounded down to nearest 10 minutes
        t = t.replace(minute=nearest10Minutes) + timedelta(hours=4)
        trade_time = t.strftime('%Y-%m-%d %H:%M')

        if not trade_time in prices:
            prices[trade_time] = []

        # Build up a price list of ticker|price pairs for all stocks for each 10 minute period
        priceList = prices[trade_time]
        priceList.append(ticker + '|' + price);

    except:
        continue

# Emit the price lists: key = time val = csv of ticker|price pairs
for trade_time in prices:
    priceList = prices[trade_time]
    print '%s\t%s' % (trade_time, ','.join(priceList))
```

```
#!/usr/bin/env python
#
# redStockPrices.py: Reduce file used by Hadoop
# Input: stock prices as ticker|price pairs, keyed by time
# Output: time, ticker, value csv lines
#
import sys
from datetime import datetime

prices={}
current_time = '';

def output():
    for ticker in prices:
        try:
            print '%s,%s,%s' % (current_time, ticker, prices[ticker])
        except:
            continue

input = sys.stdin

for line in input:
    try:
        keyVal1 = line.strip().split('\t')
        time = keyVal1[0]
        vals = keyVal1[1].strip().split(',')

        # Since the keys are ordered, if we get a new time, can output the old time period
        if time != current_time:
            output()
            current_time = time
            prices = {}

        # Build a map with stock tickers and prices for output
        for entry in vals:
            keyVal2 = entry.split('|')
            ticker = keyVal2[0]
            price = keyVal2[1]
            prices[ticker] = price

    except:
        continue

# Output the last time period
output()
```

```
#!/usr/bin/env python
#
# mapTweets.py: Map file used by Hadoop
#
# 1. input = tweet files containing 1000 tweets
# 2. Score sentiment.  Tweets with neutral sentiment are discarded (abs(sent) < 0.25)
# 3. Score relevance against the 10k SOLR IR system. Results with low relevance are discarded (relevance < 0.1)
# 4. Calculate the score for each relevant ticker = sentiment * relevant
# 5. Output
# key = time
# val = csv containing tweet + ticker|score for each relevant ticker
import sys
import json
from datetime import datetime, timedelta
import re
import math
import solr

input = sys.stdin

# Solr connection for 10k IR system
s = solr.SolrConnection('http://ec2-54-187-252-24.us-west-2.compute.amazonaws.com:8983/solr/sectenk')

# These stopwords are removed from the IR query
stopwords = ["@", "a", "about", "above", "after", "again", "against", "all", "am", "an", "and", "any", "..." ]
# REDACTED

# Word values used for sentiment analysis
sentiment = {
    ":)": [-1.97513, -23.69606],
    "...": [-4.50578, -5.13896],
    # REDACTED
}

# Sentiment Analysis courtesy Alex Davies http://alexdavies.net/twitter-emoticon-meanings/
# Used with permission
def classifySentiment(words):
    # Get the log-probability of each word under each sentiment
    tweetSentiment = [sentiment[word] for word in words if word in sentiment]

    # Sum all the log-probabilities for each sentiment to get a log-probability for the whole tweet
    tweet_happy_log_prob = 0
    tweet_sad_log_prob = 0
    for probs in tweetSentiment:
        tweet_happy_log_prob += probs[0]
        tweet_sad_log_prob += probs[1]

    # Calculate the probability of the tweet belonging to each sentiment
    prob_happy = 1.0 / (math.exp(tweet_sad_log_prob - tweet_happy_log_prob) + 1)
    return (prob_happy - 0.5) * 2  # 0 to 1.0 => Happy, -1.0 to 0 => Sad
```

```
for line in input:
    try:
        line = line.strip()
        tweet = json.loads(line)

        # Skip non-tweets or those not in English
        if 'delete' in tweet: continue
        if not 'created_at' in tweet: continue
        if tweet['lang'] != 'en': continue

        # Get the text and split out words with regex
        text = tweet['text']
        words = re.findall(r"[\w']+", text)

        # Get Sentiment and skip neutral tweets
        tweetSentiment = classifySentiment(words)
        if abs(tweetSentiment) < 0.25: continue  #

        # Get time to nearest 10 minutes
        t = datetime.strptime(tweet['created_at'], '%a %b %d %H:%M:%S +0000 %Y')
        nearest10Minutes = (t.minute / 10) * 10;  # Round down to nearest 10 minutes
        t = t.replace(minute=nearest10Minutes, second=0)
        tweet_time = t.strftime('%Y-%m-%d %H:%M')

        # Query Solr for relevance
        words = [word for word in words if word not in stopwords]
        query = '+'.join(words)
        response = s.query('ten_k_text:' + query, 'ticker_symbol,score')
        scores = []
        for hit in response.results:
            score = hit['score']
            if float(score) > 0.1:
                scores.append('%s|%0.4f' % (hit['ticker_symbol'], score))

        # Emit key = time, value = csv(tweet_text + list of ticker|tweetShift pairs)
        if len(scores) > 0:
            score_list = ','.join(scores)
            tweet_text = text.replace(',', ' ').replace('\n',
                                              ' ')  # Remove commas and newlines from tweet_text so it doesn't break csv
            print '%s\t%s,%s' % (tweet_time, tweet_text, score_list)

    except Exception as e:
        print >> sys.stderr, e
        continue
```

```
#!/usr/bin/env python
#
# redTweets.py: Reduce file used by Hadoop
# Input: stock sentiments as ticker|sentiment pairs, keyed by time (10 minute periods)
# Output: time, ticker, aggregated sun of tweetShift, aggregated count of Tweets, and the text and score of the most
# significant Tweet for the period
import sys
from datetime import datetime

data={}
current_time = '';

class Entry:
    def __init__(self):
        self.sum = 0.0
        self.count = 0
        self.hi_tweet = ''
        self.hi_score = 0.0

    def update(self, tweet, score):
        self.sum += score
        self.count += 1
        if abs(score) > abs(self.hi_score):
            self.hi_tweet = tweet
            self.hi_score = score

def output():
    for ticker in data:
        try:
            entry = data[ticker]
            print '%s,%s,%0.4f,%d,%s,%0.4f' % (current_time, ticker, entry.sum, entry.count,
                                                entry.hi_tweet, entry.hi_score)
        except:
            continue

input = sys.stdin

for line in input:
    try:
        keyVal1 = line.strip().split('\t')
        time = keyVal1[0]
        vals = keyVal1[1].strip().split(',')

        # Keys are in order so a new time means we can output the old time period
        if time != current_time:
            output()
            current_time = time
            data = {}
```

```
# Current tweet
        tweet = vals[0]

        # Aggregate each ticker
        for entry in vals[1:]:
            keyVal2 = entry.split('|')
            ticker = keyVal2[0]
            score = float(keyVal2[1])
            if not ticker in data:
                data[ticker] = Entry()

            entry = data[ticker]
            entry.update(tweet, score)


    except Exception as e:
        print >> sys.stderr, e
        continue

# Output the last time period
output()
```

```
#!/usr/bin/env python
#
# tenMinutePrices.py:  Read in Hadoop price output and produce individual csv files for each Ticker/10
# minute period containing time, ticker, and price
#
import sys
from datetime import datetime

time_format = '%Y-%m-%d %H:%M'
start = datetime(2014, 8, 4)
finish = datetime(2014, 8, 9)

input = sys.stdin
if len(sys.argv) > 1:
    input = open(sys.argv[1], 'r')

for line in input:
    try:
        vals = line.strip().strip().split(',')
        time = vals[0]
        ticker = vals[1]
        price = vals[2]

        t = datetime.strptime(time, time_format)
        if t < start or t > finish: continue  # Skip

        filename = 'D:\\TenMinute\\' + ticker + '_' + time.replace(' ', '_').replace(':', '')  + '.twp'
        with open(filename, 'w') as f:
            f.write('%s,%s,%s' % (time, ticker, price))

    except Exception as e:
        print >> sys.stderr, e
        continue
```

```
#!/usr/bin/env python
#
# priceShift.py:  Read in individual csv files containing price data and caluculate the priceShift
# by comparing the price to the previous price.  Also and fill in missing points by copying previous points.
#
import sys
from datetime import datetime, timedelta
from os import listdir

time_format = '%Y-%m-%d %H:%M'
tenMinutes = timedelta(minutes=10)
dir = 'D:\\TenMinute\\'
start = datetime(2014, 8, 4)
finish = datetime(2014, 8, 9)

class FileEntry:
    def __init__(self, tm, ticker, price):
        self.ticker = ticker
        self.time = tm
        self.price = price
        self.priceShift = 0.0

    @classmethod
    def initFromPrevious(cls, previous):
        return cls(previous.time + tenMinutes, previous.ticker, previous.price)

    @classmethod
    def initFromFileLine(cls, line):
        vals = line.strip().split(',')
        return cls(datetime.strptime(vals[0], time_format), vals[1], float(vals[2]))

    # Output csv files with time, ticker, price and priceShift
    def writeFile(self):
        filename = dir + self.ticker + '_' + \
                self.time.strftime(time_format).replace(' ', '_').replace(':', '') + '.twp'
        with open(filename, 'w') as f:
            f.write('%s,%s,%0.2f,%0.2f' % (self.time.strftime(time_format), self.ticker, self.price, self.priceShift))

filenames = listdir(dir)
previous_entry = None
current_entry = None

# Write files until finish time
def runOutEntry(entry):
    while entry.time < finish:
        entry = FileEntry.initFromPrevious(entry)
        entry.writeFile()
```

```
for filename in filenames:
   try:
      with open(dir + filename, 'r') as f:
         # Just one line
         for line in f:
            current_entry = FileEntry.initFromFileLine(line)
            break

         # If the ticker symbol changed, run out the previous ticker symbol
         if previous_entry is not None and current_entry.ticker != previous_entry.ticker:
            runOutEntry(previous_entry)
            previous_entry = None

         # If there is no previous entry, calculate the previous  entry from the current entry
         if previous_entry is None:
            previous_entry = FileEntry.initFromPrevious(current_entry)
            previous_entry.time = start
            previous_entry.writeFile()

         # Fill in any gaps
         while (current_entry.time - previous_entry.time) > tenMinutes:
            previous_entry = FileEntry.initFromPrevious(previous_entry)
            previous_entry.writeFile()

         # Write current entry and move to next
         current_entry.priceShift = current_entry.price - previous_entry.price
         current_entry.writeFile()
         previous_entry = current_entry

   except Exception as e:
      print >> sys.stderr, e
      continue

# Run out lst entry to finish time
runOutEntry(current_entry)
```

```
#!/usr/bin/env python
#
# tweetShift.py:  apply tweetShifts to existing files of priceShifts
#
import sys
from datetime import datetime, timedelta
from os import listdir, path
import os

time_format = '%Y-%m-%d %H:%M'
outputDir = 'D:\\TenMinute\\'
inputDir = 'TweetResults'

class FileEntry:
  # Create a FileEntry from Hadoop Reduce output
  def __init__(self, fileline):
    vals = fileline.strip().split(',')
    self.time = vals[0]
    self.ticker = vals[1]
    self.tweetShift = vals[2]
    self.count = vals[3]
    self.tweet = vals[4]
    self.score = vals[5]

  # Write to the end an existing file
  def writeFile(self):
    filename = outputDir + self.ticker + '_' + self.time.replace(' ', '_').replace(':', '') + '.twp'
    if os.path.isfile(filename):
      with open(filename, 'a') as f:
        f.write(',%s,%s,%s,%s\n' % (self.tweetShift, self.count, self.tweet, self.score))
    else:
      print '!!! Missing File: ' + filename

# Walk the inputDir, read each line in each input file, and write to end of files in outputDir
for root, dirs, files in os.walk(inputDir, topdown=False):
  for file in files:
    try:
      with open(os.path.join(root, file), 'r') as f:
        for line in f:
          entry = FileEntry(line)
          entry.writeFile()

    except Exception as e:
      print >> sys.stderr, e
      continue
```

```
#!/usr/bin/env python
#
# makeOneFile.py:  Gather up all the individual files contained in inputDir and combine them
# into a single file, filling in missing carriage returns.
#
# End time is passed as cmd arg, and process ignores records after end-time

import sys
from datetime import datetime
import os

inputDir = 'D:\\TenMinute\\'
end_time = None
time_format = '%Y%m%d%H'
file_time_format = '%Y-%m-%d %H:%M'

# Get the end-time
if len(sys.argv) > 1:
    end_time = datetime.strptime(sys.argv[1], time_format)
else:
    print 'End time required (%s)' % time_format
    exit(1)

# Output filename = end-time
with open(sys.argv[1] + '.twp', 'w') as output:
    # Walk  inputDir
    for root, dirs, files in os.walk(inputDir, topdown=False):
        for file in files:
            try:
                with open(os.path.join(root, file), 'r') as input:
                    for line in input:
                        line = line.strip()
                        if len(line) > 0:
                            time = datetime.strptime(line.split(',')[0], file_time_format)
                            if time >= end_time: continue  # Skip this record
                            output.write(line + '\n')
            except Exception as e:
                print >> sys.stderr, str(e) + " : " + os.path.join(root, file)
                continue
```

```r
##############################################
# RJCL W205 Project
# analysis_final.R:  Correlating stock price changes with predictions from Twitter
# 08/17/2014
##############################################

### Set working directory
getwd();
setwd("C:/Berkeley MIDS/W205/Project/");
list.files();

library(ggplot2)
library(lubridate)


data <- read.csv("2014080900.csv",header=T)
data$dateOnly <- as.Date(data$datetime);
colnames(data)

### Stocks with most Tweets
stockTweetVolume <- tapply(data$tweetVolume,data$ticker,sum,na.rm=T)
stockTweetVolume <- as.data.frame(stockTweetVolume)
stockTweetVolume$ticker <- rownames(stockTweetVolume)
rownames(stockTweetVolume) <- NULL
colnames(stockTweetVolume) <- c("tweetVolume","ticker")

head(stockTweetVolume[order(stockTweetVolume$tweetVolume,decreasing=T),],n=10)
head(stockTweetVolume[order(stockTweetVolume$tweetVolume),],n=100)

### 10 Most Relevant Tweets
head(data[order(data$mostRelevantTweetScore,decreasing=T),],n=10)
dataMinusAmp <- data[!grepl("&",data$mostRelevantTweetText),]
dataMinusAmp <- dataMinusAmp[!is.na(dataMinusAmp$mostRelevantTweetScore),]
head(dataMinusAmp[order(dataMinusAmp$mostRelevantTweetScore,decreasing=T),],n=10)
  ### most relevant tweet with out a "&" has a score of 1.558 ...
  ### maybe we should cap at 2? (would remove about 5000 records)

max(dataMinusAmp$mostRelevantTweetScore, na.rm=T)
summary(dataMinusAmp$mostRelevantTweetScore, na.rm = T)

sum(is.na(data$priceShift))
dataWithPriceShift <- data[data$priceShift != 0,] ### Using data with price shifts as proxy for working hours

### 10-minute interval data
summary(dataWithPriceShift$tweetShift)
qplot(dataWithPriceShift$priceShift,
      dataWithPriceShift$tweetShift,
      xlab = "Price Shift",
      ylab = "Tweet Shift",
      xlim = c(-5,5),
      main = "Tweet Shift vs Price Shift");

cor.test(dataWithPriceShift$priceShift,dataWithPriceShift$tweetShift);


#### Hourly Data
head(dataWithPriceShift)
dataWithPriceShift$datetime2 <- as.character(dataWithPriceShift$datetime)
dataWithPriceShift$datetimeClean <- as.POSIXct(dataWithPriceShift$datetime2, tz="GMT")
dataWithPriceShift <- subset(dataWithPriceShift,select = -c(datetime2))
dataWithPriceShift$hour <- hour(dataWithPriceShift$datetimeClean)

hourPriceShift <- aggregate(dataWithPriceShift$priceShift ~ dataWithPriceShift$dateOnly + dataWithPriceShift$hour,
                            data = dataWithPriceShift,
                            sum)
class(hourPriceShift)
colnames(hourPriceShift) = c("dateOnly","hour","priceShift")
head(hourPriceShift)

hourTweetShift <- aggregate(dataWithPriceShift$tweetShift ~ dataWithPriceShift$dateOnly +
dataWithPriceShift$hour,
                            data = dataWithPriceShift,
                            sum)
colnames(hourTweetShift) = c("dateOnly","hour","tweetShift")
head(hourTweetShift)

hourData <- merge(hourPriceShift,hourTweetShift,by=c("dateOnly","hour"))

qplot(hourData$priceShift,
      hourData$tweetShift,
      xlab = "Price Shift",
      ylab = "Tweet Shift",
      main = "Tweet Shift vs Price Shift");

cor.test(hourData$priceShift,hourData$tweetShift);


### Day-Level Data
dayPriceShift <- tapply(dataWithPriceShift$priceShift, dataWithPriceShift$dateOnly,sum, na.rm = T)
dayPriceShift <- as.data.frame(dayPriceShift)
dayPriceShift$dateOnly <- rownames(dayPriceShift)
rownames(dayPriceShift) <- NULL
dayPriceShift

dayTweetShift <- tapply(dataWithPriceShift$tweetShift,dataWithPriceShift$dateOnly,sum, na.rm = T)
dayTweetShift <- as.data.frame(dayTweetShift)
dayTweetShift$dateOnly <- rownames(dayTweetShift)
rownames(dayTweetShift) <- NULL
dayTweetShift
```

```
dayData <- merge(dayPriceShift,dayTweetShift,by="dateOnly")
head(dayData)

qplot(dayData$dayPriceShift,
    dayData$dayTweetShift,
    xlab = "Price Shift",
    ylab = "Tweet Shift",
    main = "Tweet Shift vs Price Shift");

cor.test(dayData$dayPriceShift,dayData$dayTweetShift);


### 10-minute shifted correlation
dataWithPriceShift$datetime10MinShift <- dataWithPriceShift$datetimeClean+600

help(merge)
colnames(dataWithPriceShift)
shiftedData <- merge(dataWithPriceShift,dataWithPriceShift,
                by.x=c("datetimeClean","ticker"),
                by.y = c("datetime10MinShift","ticker"))

head(shiftedData)

cor.test(shiftedData$priceShift.y, shiftedData$tweetShift.x)

qplot(shiftedData$priceShift.y,
    shiftedData$tweetShift.x,
    xlab = "Price Shift (after 10 min)",
    ylab = "Tweet Shift",
    xlim = c(-5,5),
    main = "Tweet Shift vs Price Shift w/ 10 min shift");
```

Appendix B: Midterm Presentation

https://docs.google.com/presentation/d/1EIqmIHp2KXmHDf7vNuQjchdU4BNrBn9LAiq4vjEAKZI/edit#slide=id.p