Problem Definition:
To implement a data pipeline that will show the overall sentiment on the U.S. stock market in real-time by analyzing the relevant tweets on Twitter as they are posted. The major premise is that, during a highly volatile market session, overall sentiment would be negative if more people are bearish on stocks while sentiment would be positive if people are generally bullish on stocks. That is, a real-time sentiment reading could be used as a technical indicator for a market participant.

There are several major challenges to this problem: first is the selection criteria of "relevant tweets"; second challenge is the communication method between the different processes of this project; third is the aggregation and transformation of tweets in preparation for sentiment classification; and lastly the challenge of serving the final result to a website.

Methodology:
Three separate processes are designed, each focusing on a different aspect of the major challenges:

A twitter_client process that opens a streaming link with Twitter and actively track up to 400 topics (maximum allowed by Twitter). The biggest challenge here is the specification of "topics" to track - Twitter is filled with irrelevant advertisements and messages, how do I track only the topics that are related to the over U.S. stock market sentiment? At first I constructed a list of 400 symbols of companies that are in the S&P500 index as my list of topics. Initial results were not satisfactory - most of the tweets returned are irrelevant to S&P500 or are simply just spam. I experimented with various search terms, and finally decided to use "$" + symbol + " stock" as a topic (for example, "$AAPL stock").

Furthermore, to make the project scalable and overcome the limit of 400 topics per client, I decided that if required I could deploy multiple instances of the Twitter_client process, each with a set of different Twitter account / credentials. However, this brings up a different challenge where I would have multiple instances of twitter_client streaming tweets in real-time, but only one process to aggregate the tweets. I needed some technology to reliably send and receive messages between all these processes in parallel - enter Kafka. I have set up twitter_client to forward all tweets that it receives via Kafka to the central aggregation process.

The twitter_sentiment process is where the central aggregation happens. A streaming spark context is opened with Kafka via spark-streaming-kafka package, and once every minute the accumulated tweets from twitter_client in Kafka are read, aggregated, and transformed into one huge bulk JSON message. Next, the JSON message is forwarded to a message classification website made by three Stanford graduates in natural language processing, sentiment140.com, where all the tweets within the bulk JSON are classified with a sentiment score. Twitter_sentiment takes the returned answer and only saves the significant tweets per batch (ones with either positive or negative sentiment) in JSON format in the output folder.

The twitter_display process will open the JSON files in the output folder, and build several lists in preparation of outputting to html: a general tweets history, a positive and negative tweets made in the last 24 hours, and positive and negative tweets published during the last hour. Twitter_display then fires up a Flask server to serve the data to html calls.

Finally, a simple html "index.html" is written with two fields for displaying daily and hourly sentiment classification results. This page refreshes once every other minute with the Flask server in twitter_display to automatically get the latest results. This marks the end of the data pipeline.

Problems:
There were a whole host of problems and obstacles that I had to overcome while building this application. This project made extensive use of APIs provided by Twitter and sentiment140.com, but in order to use the APIs my application needs to be certified with proper keys and access tokens. I needed to figure out what terms like OAuth and RESTful mean, and find out how to communicate with these two websites, as I have no prior experience working with websites or APIs.

The next major challenge came with getting Kafka installed and working. First was obtaining the magic Kafka access address of "rcg-hadoop-01:6667", thankfully Mr.Baker was quick to help with that.

Most of the development effort was spent on twitter_sentiment.py. The biggest obstacle in this file was figuring out how to get Kafka to work with Spark Streaming, which took a lot of research plus trial and error. The various transformations from between Kafka input to forwarding tweets to sentiment140.com and the eventual output of final results to disk also took a lot of trial and error. After a few days I was able to get it to work via spark-streaming-kaka package, again with Mr. Baker's help. I eventually decided to go with mostly Spark Dataframes for data manipulation, and JSON format for output to disk. I had to experiment with different ways of saving results to JSON and then subsequently trying to read these results from the twitter_display module.

The biggest problem I encountered, however, was with the twitter_display.py module. This process is supposed to spin up a http server and serve the saved result to a website, which sounds pretty easy. Except for the fact that I have no web development experience and had absolutely no idea how to proceed. Another week of research and book-flipping later, I settled on using Flask under Python for handling http requests server-side, but ran into a huge problem with pyspark and Flask almost immediately - apparently a Flask application would "restart" itself during execution, but when it does, the lines that initialize spark context would get re-run, and the whole process would exit with error. No duplicate spark context allowed.

Because I had originally planned to save results in Parquet in twitter_sentiment, then use spark context to read the Parquet in twitter_display, I needed spark context to run within Flask. Therefore I started a futile quest to look for ways to initiate spark context exactly once with Flask, which took several days of trial and error to no avail. Eventually I gave up on the idea of using spark context within Flask and revisited my file system, and worked out a way to read the files without using Spark. In the end, though, I scrapped the whole idea of setting up a server to serve the JSON data, instead I would save the results to JSON and load from html directly.

Lastly, there is a lingering problem that I never managed to arrive at a satisfactory solution: the search term criteria for tracking 'relevant' tweets for S&P500 and its member stocks. Using "$[name] stock" as track term filtered out some unrelated tweets, but many spam messages still remain in the streams and consequently processed and classified in the final result. Also, duplicate messages by intentional users are sometimes detectable, skewing results.

Outcome:
An almost-real-time data pipeline, from Twitter posts to the final sentiment classification result served on html, is implemented. In building this project, I've learned to:

1) communicate with other websites using RESTful APIs and OAuth authentication.
2) how to install library or package to my directory when a process complains that a library X is missing yet doing a pip on the cluster says that library X is already installed.
3) how to set up and run Kafka to send and receive messages between processes
4) how to set up Spark Streaming to work nicely with Kafka
5) intricacies of sending JSON objects across the internet and saving JSON to disk
6) how to work with the file system in pyspark and Python
7) how to set up a http server with Flask in Python
8) how to write basic html file

I would rate the amount of effort put into the follow tasks as follows:

- Getting the data: Acquiring/gathering/downloading: 4
- ETL: Extract-Transform-Load work and cleaning the data set: 2
- Problem: Work on defining problem itself and motivation for the analysis: 3
- Algorithmic work: Work on the algorithms needed to work with the data, including integrating data mining and machine learning techniques: 0
- Bigness/parallelization: Efficiency of the analysis on a cluster, and scalability to larger data set: 2
- UI: User interface to the results, possibly including web or data exploration frontends: 3
- Visualization: Visualization of analysis results: 1
- Technologies: New technologies learned as part of doing the project: 5