Arin Schwartz

6/4/2013

CMSC123

Cluster Analysis of Twitter Investment Community

The goal of this program is to explore characteristics of Twitter users who tweet about stocks. The investment community is surprisingly vibrant on Twitter, and being able to categorize these users could give valuable insight into Twitter's potential effect on real markets. The first goal was to categorize users based on a number of Twitter statistics to learn more about characteristics of potentially impactful behavior. The investment sub-community of Twitter was chosen for a number of reasons. First, Twitter requires a user to filter the tweet stream in order to collect anything, so appropriate search criteria were needed. Second, stock tickers on Twitter are perpended with a dollar sign, making them an easily searchable category. Third and maybe most apparent, many possibilities exist for analyzing the Twitter investment community's effect on real markets.

The program is written in R and Python, at times using the module *rpy* to interact with R from Python. It is divided into three main parts; a python class, TweetCollector, for the collection and management of tweet data, another python class, TwitterUser, to provide a custom representation of a Twitter user, and a k-means clustering algorithm in R. Results have not been groundbreaking by any stretch so far, but the groundwork has been laid for much more sophisticated analysis. In this paper, I will first cover the design and implementation of the program, along with the datasets generated and methodology used. I will then discuss results, followed by a brief discussion of some common pitfalls and future plans for continued analysis.

Classes

TweetCollector uses a wrapper for the Twitter API called *python-twitter.* The class uses the search API to mimic a tweet stream by periodically performing searches filtered by time, and storing the results. A TweetCollector object has many attributes, including most importantly the search string, as well as the refresh rate for periodic searches. This rate must not be too short for fear of exceeding Twitter's rate limit, and cannot be too long to avoid missing tweets. With a default refresh rate of three-hundred seconds (5min), TweetCollector will search Twitter for any new tweets containing its assigned search string and write them to a file. Each time TweetCollector begins a session, a new file is created with a time stamp. Using this class, I collected all tweets made during market hours over a week-long period about the top ten market cap stocks in the S&P500, as well as Fannie Mae and Freddie Mac. Fannie Mae and Freddie Mac have been creating some buzz as of late and it seemed appropriate to include them to collect some reactionary users. For every Tweet that it detects, TweetCollector records and writes the search string, screen name of the tweeting user, the time stamp, the contents of the tweet, and the tweet's unique ID number. This dataset will be valuable later on as I will discuss, but for the purpose of this first analysis I was mainly concerned with the users themselves.

The TwitterUser class is the central class in the program. This collects and organizes all important user-centric data, and prepares it for analysis in R. For each file of tweets collected by TweetCollector, TwitterUser parses them and generates a list of all users, along with metadata. It uses Twitter's API to retrieve some user statistics: the user's screen name, a unique ID number, total number of statuses posted, number of followers, number of friends, number of lists this user appears on, number of favorites they have, and the average amount of times one of their tweets is retweeted. TwitterUser stores these attributes in a class system for use in the interactive shell, but also contains a list attribute, which packages the user statistics into a convenient form to begin coercion into R objects. The class sytem allows easy exploration of a single user-case, and the list form allows larger analysis to run more smoothly.

When a list of users is generated from a file, the data is then ready to be ported into R. The module *rpy* is very robust, but has stringent conventions for type conversion between R and Python as well as cumbersome syntax. A good deal of data-coercion had to be done to align my datasets with R's formatting requirements.

The last piece of the program is the k-means analysis in R. This takes a column-major matrix or data frame and returns another matrix with an extra row appended representing the final cluster assignment for each data point. K-means.R is run entirely from the python class TwitterUser and the objects returned are accessible as both python objects and R objects.

Results So Far & Next Steps

Results have been preliminary, but informative for further steps. Many Twitter users are smaller-time investment bloggers and go not give consistently high numbers in their statistics. However there do exist some extremely heavy outliers like CNBC and Forbes, who sport very a high average retweet rate as well as a following that dwarfs that of individuals. Through quantitative categorization of these users, we may be able to see whether these “high-impact” users have an influence on the market with their tweets.

For this analysis we will have to return to the original tweet dataset and use the time-stamps recorded. We could explore the relationship between the timing of stock-related tweets and market events, such as large spikes in volume or a large fluctuation in price. With the S&P500, it will probably be impossible to be sure if any market activity is directly related to tweets, since the noise level of these high-volume price series are extremely high. However the S&P500 was a sure-fire way to collect users which made it the perfect candidate for the cluster analysis. When analyzing the timing of tweets versus price movement, it may help to turn to lower-volume stocks whose large adjustments are more easily attributable to a human cause.

Another direction for further analysis would be to attempt to find a more sophisticated metric for determining a user's influence. For each user, all of their followers could be found; from there, we could see important statistics about idea-diffusion such as what percentage of people pass the messages they receive from this user on to others. Large companies may have the most followers, but smaller bloggers may have higher credibility among smaller niches in the stock market. For large entities, this would require a parallel design to run efficiently.

For very heavy analysis, it will be beneficial to parallelize our k-means clustering. This would require distribution of the distance calculations across multiple machines, coupled with a message passing interface to aid in communication. We would accomplish this by dividing the space into partitions, giving one to each computer. Distance calculations for the partition could easily be made to all centroids, and message passing between machines would allow clusters to be assigned correctly. However, parallelization is not the only hurdle in the way of heavier analysis.

Pitfalls and Possible Solutions

Twitter's API is heavily rate limited, and special permissions are required to exceed this limit. The fundamental design of the TwitterUser class requires user info to be pulled from the searc API. For this reason, though I had collected a lot of users, analysis was slow at times as I ran into the 350-call-per-hour cap. More research will need to be done to see if there is a way I can further minimize API calls, but for any real large-scale analysis a major redesign of the program would be needed, or we would need permission to exceed the rate limit.

Another difficult aspect of this project was *rpy* itself. While admittedly a comprehensive module, some of the conversions between python and R datatypes were inconsistent. *Rpy* can sometimes require one to manage variables in Python and R simultaneously, and the syntax around this can be extremely time-consuming to get used to. Using *rpy* was a new experience, and something I had avoided in the past; while I am glad I finally learned it, I feel a better solution could have been designed from the beginning, either avoiding *rpy* entirely, or embracing it entirely.

Preliminary results of the cluster analysis were mildly disappointing, but expected – this was partly due to the rate limit, as at this stage more sheer volume of users was needed to determine significant clusters. While every permutation of collected data so far has been clustered upon, more time is needed to collect more data points, as well as aggregate all collected data for a far more significant analysis.

A potential solution for the rate limits as well as *rpi* would require a reworking of how the program's pieces interact with one another. Instead of loading in user data for analysis all at once, the rest of the program could mimic the TweetCollector class. That is, it could wait one hour, process a file of collected tweets, write all gathered user data to another file, close down and wait for the rate limit to reset itself. This way, we would be slowly collecting larger amounts of data over time. The data-sparsity problem caused by the rate limit would be handled this way as well, as we would be reading from static files for larger analysis.

Another somewhat obvious solution would be to augment the TweetCollector class to use the streaming API instead of the search API. This has potential for larger scaling with faster analysis. It seems that the design of a program like this would benefit from an “all-or-nothing” approach: either go entirely with large files exchanged between components, or go entirely streaming, with information being shared and collected real-time. This middle-ground approach did not lend itself to scalability, as user-data generation proved to be a huge bottleneck in gaining enough datapoints for robust analysis.

Conclusion

I set up a decent object-oriented architecture for the collection and first-analysis of Twitter user data. Its interactive components are currently its strongest aspects, as data can be explored very finely on a low level. While this was great for debugging, some design choices early on inhibited scalability quite a bit, and slowed down the generation of large user-centric datasets. *Rpy* is definitely useful in some contexts, and it would be incredibly useful in a program built around a real-time data stream. However for processing of static files it proved a bit cumbersome, and standalone R would work just as well if not better for reading in static files. Further steps for analysis will involve an iterative upgrade to smooth the API calls of the TwitterUser class so that more local data can be generated. Also, a first analysis of price data could yield some very interesting user credibility metrics. Once all local data has been aggregated and analyzed, potential exists for a switch to a filly-streaming system which could be used to even further understand credibility among users. We could also parallelize k-means, so there is plenty of room for scaling in theory. This first analysis gave rise to more questions than answers, but it seems with a project such as this, how could it not? Aside from the technical skills I learned, I became aware of the veritable ocean that is Twitter. The impact of these small status updates is definitely measurable at some level, and there are endless questions to ask as well as endless possibilities to explore.