Facebook Topics Extraction System (FTES)

Khoa Tran
khoatran@berkeley.edu
INFO 256 Fall 2013

December 15, 2013

Contents

1 Introduction

Working with social network data and extracting information from them is increasingly becoming a major topic in Natural Language Processing and Data Mining. In recent years, Twitter datastream is used to predict stocks[2], as well as flu outbreaks[5], among other things. In this final project, Facebook feeds are explored to build a simple search system and to determine popular topics being discussed in real time.

1.1 Project Goals

- To extract information related to a search query, and compare the result with Facebook's search
- To identify popular topics in a given facebook page

1.2 Motivation

Facebook search uses different metrics, including but not limited to, the number of views, likes, and comments. Based on these counts, it then decides which posts would be most relevant to a given search query. While popularity is certainly an interesting metric, oftentimes the search result doesn't carry the relevant information. For example, a sarcastic response with completely opposite information, which is often the case on social network, may receive many 'likes' from other users. To that end, one of the project goals is to determine how well the NLP approach - a bag of words model with TFIDF weighting and Cosine Distance Similarity - would perform compare to Facebook search.

Apart from search, there's much to learn from Facebook feeds. One common example would be what popular topics are being discussed. Twitter introduced "trendings" a few years back, which allows its users to see what everyone else around the world is talking about. Given a long list of posts, how would a user determine the popular topics without scrolling through every single post? The second goal of this project is to answer this question.

1.3 Why Facebook?

There are certainly many interesting datasets out there, but nothing comes close to the excitement from working with live data that one sees everyday. Why was Facebook chosen over Twitter, Github, or other social networking sites? First, Facebook feeds have not been explored much compared to other social networks, mostly due to the complexity and not well-designed documentation of the API. Second, the FB API doesn't offer official

support to some popular programming languages such as Python, Ruby, etc. This is mostly because Facebook Apps developers tend to prefer Javascript or PHP over aforementioned languages for different reasons. Hence, tackling this project using the unofficial Python Facebook SDK is a fantastic and challenging problem to crack.

Furthermore, unlike Twitter who requires a 140-character limit in every tweet, there is no limit to the length of a Facebook post. This is actually good news for the NLP approach because words are more often spelled out in full on Facebook than on Twitter.

2 API Usage, Dataset, and External Dependencies

2.1 Access Token and API Usage

To connect to Facebook and acquire the feeds data, the first step is to generate a Facebook Access Token[1]. Most Facebook applications will have this generated on the fly whenever a user logins to the application, which allows it to acquire the user's data. The biggest downside of Facebook Access Token is that it expires every two hours, so it is very important to locally cache the data.

Facebook API offers many different services, ranging from querying data from individual profile to Ads and Facebook Chat integration. However, it is quite cumbersome to connect to Facebook the traditional way by sending a GET request since there are a lot of different URLs with different parameters to remember. To that end, the unofficial Facebook SDK for Python is born to make this process easier. It's suprisingly easy to retrieve a list of one's friends using just two lines of code:

```
graph = facebook.GraphAPI(ACCESS_TOKEN)
friends = graph.get_connections("me", "friends")
```

`get_connections` is a very special function because it allows one to easily retrieve any connection between two different "Facebook objects". More information can be found on the Python Facebook SDK[2], as well as the official Graph API documentation[3]. In the next subsection, we will see how to acquire the dataset using this very method.

[1] `https://developers.facebook.com/tools/explorer`

[2] `http://facebook-python-library.docs-library.appspot.com/facebook-python/library-manual.html`

[3] `https://developers.facebook.com/docs/reference/apis/`

2.2 Dataset

The corpus used in this project came from the UC Berkeley Computer Science Facebook group, which was last updated on Thursday, December 12, at 5pm PST. To acquire the group's feed, we use the `get_connections` method as followed:

```
graph.get_connections(id, 'feed')
```

, where `graph` is the Facebook graph from Section 2.1. How would one find out a Facebook group ID?

It turns out that Facebook offers no easy way to achieve this. One option would be to click on 'View Page Source' and look for the group ID. Since the source code of a Facebook page is huge, this approach is much like looking for a needle in a haystack. In this project, I use a third-party service to get the Facebook Group ID[4].

Once the group ID is found, the function call above would return the data containing Facebook posts in JSON format. The data is then cleaned up using several different methods, including converting from Unicode to ASCII, dropping newline ('`\n`') and carriage ('`\r`') characters, etc. to make the data more processable.

2.3 External Dependencies

One of the thing that makes Python a great language for Data Mining and Natural Language Processing is its rich number of powerful, yet simple, packages and libraries. Apart from the Python Facebook SDK, the non-exhaustive list below contains some important packages used in this project:

Connecting to Facebook:

- `requests` is a powerful library for making RESTful requests. Born to simplify the complexity in built-in modules like `urllib` or `urllib2`, `requests` is quicky becoming the de facto way of making web requests in Python applications
- `json` and `simplejson` are used to convert back and forth between JSON data and Python's dictionary, as well as printing multi-level nested data in a nicely indented format

Numerical Computation and Visualization:

- `numpy` and `matplotlib` are the standard way for any numerical computation and plotting in Python

[4] `http://wallflux.com/facebook_id/`

- `networkx`, a package built on top of matplotlib, is used to visualize the connections in a Facebook social graph
- `pytagcloud`, a Wordle-inspired package, is used to generate word cloud completely in Python

Natural Language Processing:

- `nltk` - with 12 years of experience since its first release in 2001, there's no deny that the Natural Language Toolkit is the best friend of any Natural Language researchers working with Python
- `pattern` is a web mining module developed by CLiPS, the Computational Linguistics and Psycholinguistics Research Center. Beside web mining, `pattern` comes equipped with modules for database, web search, vectorized computation, and graphing

Last but not least, the entire project was built inside an IPython Notebook. As Professor Philip Guo said, "everything related to my analysis is located in one unified place"[5], thanks to the IPython Notebook. This simplifies a lot of things as I don't have to maintain a separate presentation for the final project demonstration, and I can document every step along the way right on the notebook, which helps a lot when I come back and type up this final report.

2.4 Viewing the Demo and Reproducing the Result

The Python code is generated using IPython Notebook's automatic code generation, which often includes more extraneous comments than a standard Python file. The best way to view the code without installing any dependency is to use `nbviewer`[6], a static IPython Notebook viewer. The entire notebook, which includes all code, documentation, and relevant images, can be found at `http://bit.ly/19Fzr5p`.[7]

To reproduce the project in a development environment, a file called `requirements.txt` is included. By running `pip install -r requirements.txt`, all external packages and dependencies will be installed. It is best to execute this command in a virtual machine or a separate `virtualenv` environment to avoid any potential dependency conflicts.

Please contact the author for any questions regarding the project and reproducing the result.

[5] `http://www.pgbovine.net/ipython-notebook-first-impressions.htm`

[6] `http://nbviewer.ipython.org/`

[7] The unshortened link is `http://nbviewer.ipython.org/github/kqdtran/FTES/blob/master/ftes.ipynb`

3 Simple Search Engine

3.1 High-level Idea

After retrieving the feeds from Facebook, the data is converted from JSON format to a Python dictionary. I decided to treat comments equally as the original posts, since they more or less carry information of the same quality. For simplicity, each original post and comment will be referred to as "post" from this point forward.

Each post is then converted into an unordered bag-of-words representation, which is represented under the hood as a dictionary of `(word,count)` tuples. A collection of posts, or documents, is a bag-of-words model. One common way to represent this model is to treat each document as a vector of length n, where n is the number of unique words in the entire model. Given a vector `vec`, `vec[i]` would return how many times a word with id i appears in the document represented by `vec`. Hence, the entire model is a matrix of size $m \times n$, with m being the total number of documents and n retains the same definition above.

It is very common to "offset" the simple term frequency model above since stopwords like "the" could appear many times, yet doesn't give us any useful information. The Term Frequency - Inverse Document Frequency weighting schema, better known as TFIDF, is often used in such cases. This value would increase proportionally to the frequency of a term in a given document, but is then offset by the same term's frequency in the entire corpus[7].

A higher TFIDF score indicates that the corresponding word is more important. After producing a matrix which contains TFIDF weights, we can measure how similar two documents are by calculating the cosine distance between them. Given vectors v_1 and v_2, one can calculate the cosine similarity between them by dividing their dot product by the product of their norms, $cos\theta = \frac{v_1 \cdot v_2}{||v_1|| \cdot ||v_2||}$[6].

To answer the question "which post is most related to a search query?", one simply calculate the cosine similarity between the input query vector and every document vector currently in the model. The highest ranked results are then returned as the most related posts to the search query.

3.2 Text Processing

Let us slightly backtrack and discuss the result of several text procesing techniques. The raw JSON data in Unicode was first converted to ASCII and all newline and carriage return characters are stripped out. The following techniques are all tried out to determine if there is any improvement over processing the raw ASCII text:

- Lowercasing
- Stemming
- Lemmatization
- Stripping punctuations
- Spelling correction

Of the five techniques, it turns out that lowercasing and lemmatization work best. Stemming did not work so well because most stemming techniques like the Porter stemmer tend to turn grammatically correct English words to incorrect ones with its greedy approach. The stemmed document sees drop in the similarity score because the words do not match the input query as often anymore.

Stripping punctuations was an interesting approach. While punctuation in general seems to "stuck" with a token, there are interesting ones like "you're" where stripping punctuations would quite mess things up, whether we replace the apostrophe with a whitespace or not. Perhaps the best approach to take here would be to translate such terms to its full words like "you are", though this was not implemented in the project.

Finally, spelling correction was also tested out. Two approaches were taken, though neither yield better search results. The first approach is to build a list of possible "suggestions", and from there, choose the word with the minimum edit distance. If there are ties, either choose the first result or choose one at random. The second, more sophisticated approach comes from Peter Norvig's famous blog post[4], which describes a model powered under the hood by Bayes Theorem, along with a training corpus consisting of several public books from Project Gutenberg and many other sources.

3.3 Coding

Pattern, the web mining module described in Section 2.3, convenient came with a built-in Document-Term model and TFIDF weighting. Each Facebook post is treated as a `pattern`'s Document object, and lemmatization as well as stopwords removal are applied whilst constructing the document. To make the result more meaningful and easily comparable with Facebook search, the permanent link to the post is embedded as a description of the document, so that we can later retrieve the link with dot notation syntax like `doc.description`. All the documents are then put into a Model object; in other words, a Model is just a collection of Documents.

To calculate similarity between an input term and every document in the downloaded Facebook feeds, we treat the input term as a separate Document object, and find the top documents whose similarity scores are highest to the input document. Using the `neighbors`

method with an optional parameter `top` which indicates how many results to retrieve, this can be done very quickly. A `pattern`'s Document or Model object offer many more interesting functionalities - please refer to the official documentation[8] for further reading.

The final result is then printed as a prettytable, which also includes links to the original Facebook posts for comparison. One of my initial project proposal goals is to build a system where users could compare and contrast the NLP-based search and Facebook's search side by side. Unfortunately, that did not happen because the Facebook API offers no method for searching within a group[9]. Facebook group's search has to be run directly on Facebook itself, and the result is then compared against the NLP approach, which shall be presented in the next subsection.

3.4 Result

To compare the result between the two approaches, I decided to come up with different queries related to the CS major at UC Berkeley with respect to the working corpus. For example, trying out the query "declaring major early" yields the following top results:

Post	Sim
What are the requirements for declaring the C...	0.82
In terms of petitioning for the major, I had ...	0.23
Hey guys, I'm an intended LSCS major. The LS-...	0.23
Is it true that LnS CS majors are relatively ...	0.23
Just for clarification... to declare early (w...	0.17
Hello CS Majors, Here's an updated graphic of...	0.17
I am currently a freshman and was trying to m...	0.16
with the new cs major requirement...cs162 is ...	0.16
So as an undeclared CS major finishing my pre...	0.12
hey guys! so i am a cog sci major cs minor, d...	0.12

, where the top result (with a significantly higher score than the rest) corresponds to

Ryan Peden

What are the requirements for declaring the CS major early?

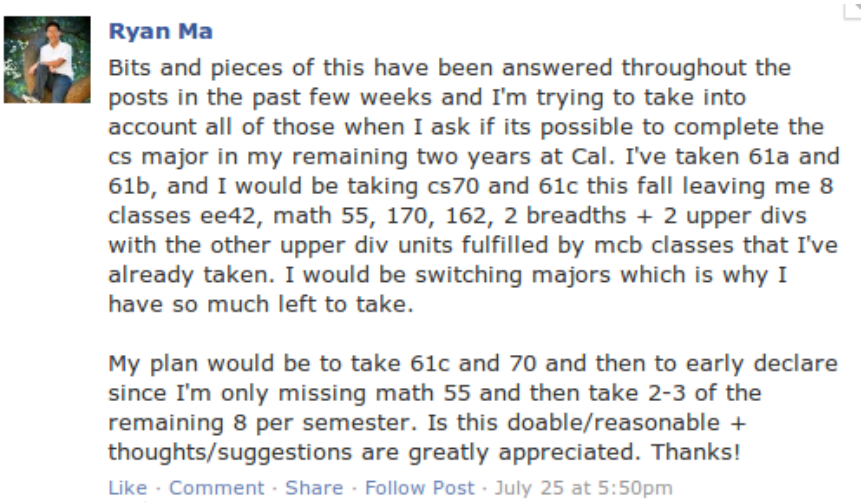
Like · Comment · Share · Follow Post · October 3 at 9:38pm near Berkeley, CA

. On the other hand, searching for the same query on Facebook returns

[8] `http://www.clips.ua.ac.be/pages/pattern-vector`

[9] `https://developers.facebook.com/docs/reference/api/group/`

, where the top result is



Let's dive into this a bit further. Based on the content of the posts, why is there a

discrepancy?

3.5 Analysis

One thing that immediately took my attention was the fact that the top result from the NLP approach is very short and concise. Since the input query is also very short, this comes out to a very high similarity score. On the other hand, the top result from Facebook search seems very descriptive and covers a very special case that few students would fall into. In other words, this post seems to be much richer in content. However, since the difference in length and in term frequency is very large, this post has an unsurprisingly low similarity score with respect to the asked query in the NLP approach. In fact, it was not even in the top 10 results using TFIDF weighting and cosine distance.

After many similar experiments, I conclude that the difference in text length is inversely proportional to the similarity score in the NLP approach. The larger the difference, the more sparse the input query vector is, and the lower the similarity score. This is understandable, since the input vector has a lot of 0s in it, while an information-rich document includes many nonzero entries, which leads to a smaller dot product and a bigger norm, both of which imply a significant decrease in the cosine similarity scores.

Another thing that may lead to the discrepancy in result was the amount of data collected. Facebook imposes a limit of 500 posts per request, hence the amount of data used to build this simple search engine depends on these 500. To improve the accuracy, a pagination implementation is necessary, so that the request can span over multiple pages and retrieve more than just 500 posts.

While the NLP approach is not a powerhouse search system with different ranking metrics like that of Facebook, it certainly performs fairly well just by looking at a post content.

4 Popular Topics Analysis

4.1 Introduction

Unlike Twitter who introduced the concept of "what's trending now?", it's relatively difficult to determine what people are talking most about on Facebook given the current UI. With Natural Language Processing, this becomes a goal completely within reach. After all, since this project has "topics extraction" in its name, it is now time to introduce a simple Popular Topics Analysis system using Part of Speech Tagging and Chunking.

The idea behind this system is to identify and extract keywords (noun phrases) from a Facebook post, keep track of the frequency, and finally determine the most talked about

topic. Once again, we do not take into account other metrics like the number of likes; everything is extracted completely based on the raw text.

4.2 Extracting Keywords

Before we can identify a noun phrase, a Part of Speech (POS) Tagger is necessary. In this project, a simple backoff POS Tagger was built based on the material presented in Chapter 5 of the online NLTK book[1]. It trains 90% of the tagged sentences in the Brown Corpus[10], and the default tagger would tag everything as a Noun. The second level tagger would look for unigrams, whereas the next and final one would tag bigrams. The reason a backoff tagger work well is because in the event we cannot assign a tag, we can always fall back one level and use a less robust tagger instead. This POS tagger has an evaluation accuracy of 84.49% on the remaining 10% of the Brown Corpus. The tagger is then pickled into a small binary file, so that we don't have to keep on building a new one everytime we want to tag a sentence.

With that said, let's move on to the chunker[11]. Two regular expression rules are used to extract the noun phrases:

```
grammar = r'''
        NBAR:
                {<NN.*|JJ>*<NN.*>}

        NP:
                {<NBAR>}
                {<NBAR><IN><NBAR>}'''
```

The first one would look for a combination of nouns and adjectives, terminated with nouns. The second one finds everything found in the first rule, along with noun phrases from the first rule that are connected with preposition or subordinating conjunction, for example, "in", "of", etc.

Once we have defined the rules, we can finally extract noun phrases from a given text. The document, i.e. Facebook feed, is processed in the same way as in the previous section: words are lowercases and lemmatized, and stopwords are filtered out. We build a Part of Speech tree and iteratively construct noun phrases from leaf node whose tag is "NP", or noun phrase.

For every noun phrase extracted from a Facebook post, we add one to its frequency count to later on check if it is among the most talked-about topics. The result will be sorted in

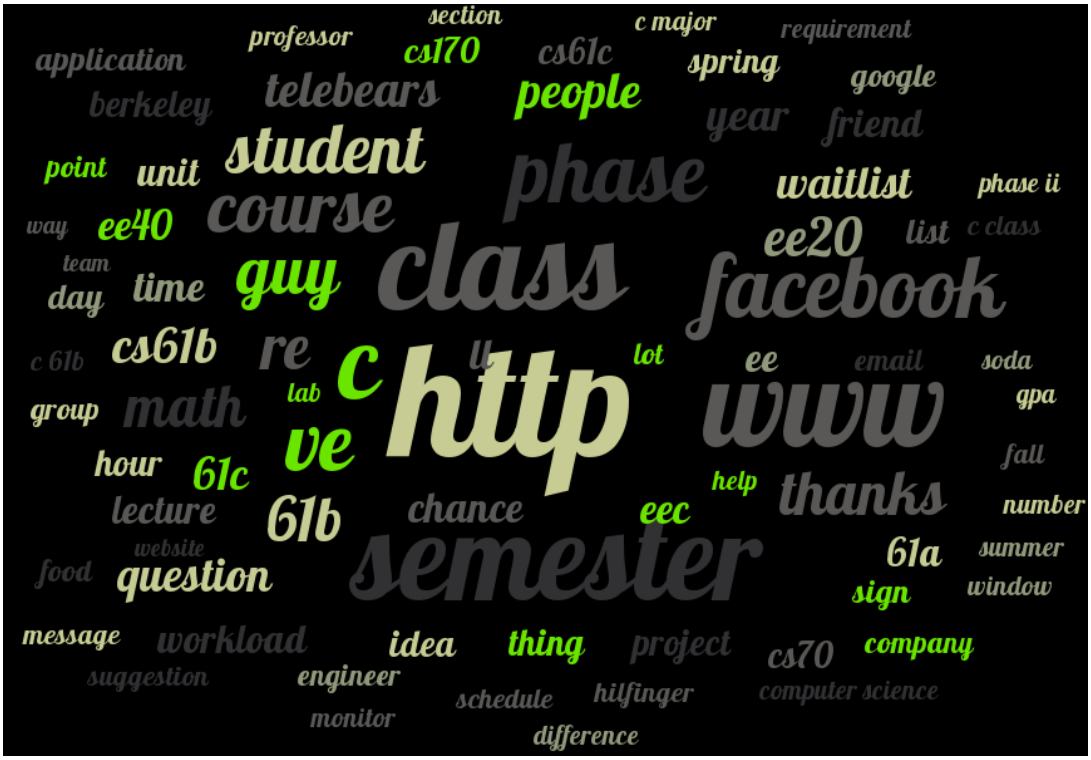
[10] `https://en.wikipedia.org/wiki/Brown_Corpus`

[11] Chunker adapted from `https://gist.github.com/alexbowe/879414`

descending, and can be used as input for many visualization software packages.

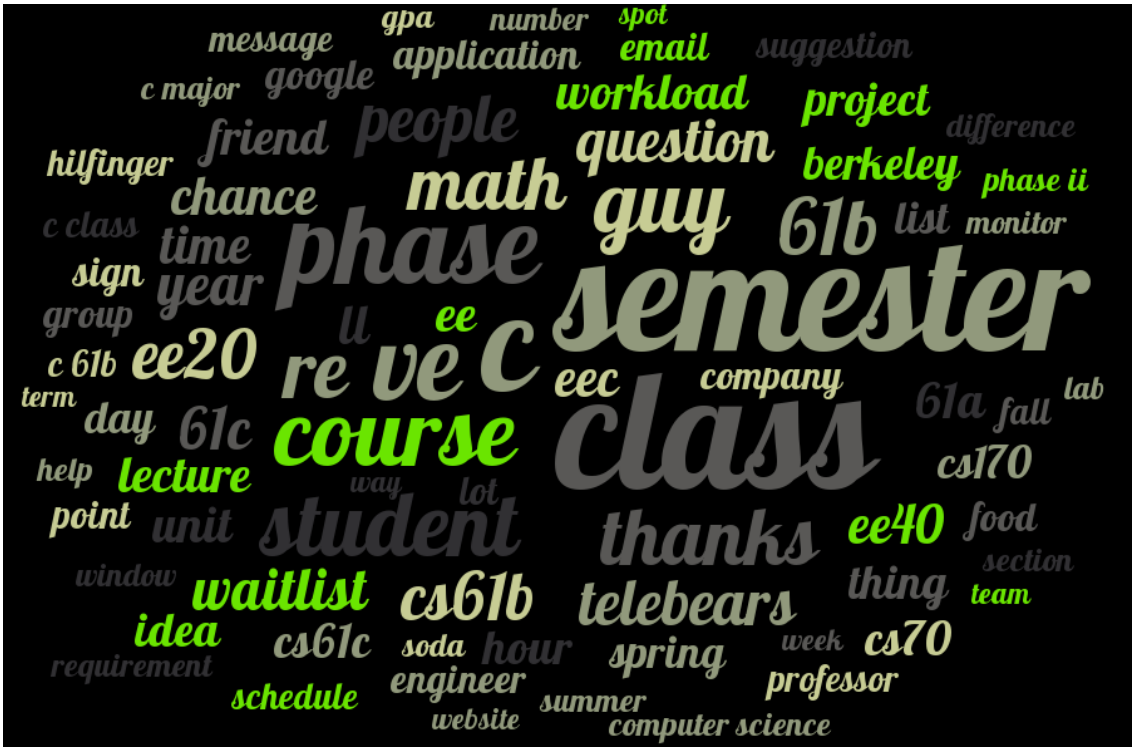
4.3 Result

PyTagCloud, a Wordle-inspired package, is used in this project to output the popular topics as a word cloud. Applying topics extraction on the same corpus from the previous section, the UC Berkeley Computer Science Facebook Group, we have some very fascinating result.



4.4 Analysis and Takeaways

It appears that there are quite some noises in the final result. This is not surprising, however, especially when words like “http”, “www”, “facebook” come from the fact that people usually share links with others on Facebook. The Regular Expression parser used in this project stripped out some punctuations, so as a result, a typical URL would be broken up into multiple pieces containing the aforementioned words. By filtering out some noises, we got the updated word cloud:



This is extremely consistent with the majority of the topics usually being discussed on the CS Facebook Group! Questions about the CS 61 series, Telebears, the workload of different courses, etc. pop up every now and then. That said, there are quite a few strange terms floating around like "eec" or "c class". This is an extremely mind-boggling point: the lemmatizer thinks of "CS" as in "C" in plural, so it strips out the "S" in every noun phrase where "CS" is a separate word!

If we drop the lemmatizer however, it appears that the word frequency would decrease by a lot. This raises a very interesting question: instead of using the NLTK's lemmatizer, should we code one up ourselves, one that can intelligently deal with very specific edge cases like this?

5 Conclusion

5.1 Project Goals Revisited

Beside the two Project Goals mentioned in Section 1.1, I originally wrote in my project proposal two more goals. One of which is to build a system where users can compare and

contrast Facebook Search and NLP Search side by side. As mentioned in Section 3.3, that did not happen due to the API limitation.

Facebook's rich features like the number of likes and views naturally make it a great corpus to apply Sentiment Analysis on. I also did not get to this point, mainly because I wanted to focus more on the first two goals instead.

Overall, the first two goals presented in Section 1.1 were met, and I definitely have a much better idea about how to build a very simple search engine using just NLP concepts.

5.2 Challenges

Perhaps the most challenging part of this project was looking through the vast Facebook Graph API documentation and figuring out what I really need to acquire the dataset and start analyzing them. Python was also not officially supported, which makes it a bit harder to translate examples from other languages to Python. Coupled with the fact that the final project is open-ended, it is very easy to get lost in the documentation and fail to figure out which part is relevant to the project goals.

Unfortunately, Facebook Access Token expires every two hours. This makes the entire process more painful than it should be, especially whenever I decided to refresh and rerun the IPython Notebook. While regenerating access token doesn't take too long, it can certainly be the most annoying part of the project.

5.3 Suggestions for Future Work

The following items (in no particular order) are some of the improvements that can be made to this project:

- Try out different TFIDF variants[3] (maximum TF normalization, sublinear scaling) and determine if it performs better than the vanilla TFIDF algorithm
- As mentioned in Section 3.5, document's length seems to greatly affect the similarity score. Is there a coefficient that we could scale by that would make the document's length less sensitive to the final output?
- Pagination Implementation that allows the request to span over multiple pages of posts, instead of just retrieving the first 500 posts
- For the Topics Extraction part, is there a better way to identify potential noises (like "http", "facebook", etc.) without having to hardcode them? How would we determine which words tell us more about the popular topics currently being discusses than those who don't? Is there some way we can take the context into account? For

example, a Markov Chain which lookup on the previous words could certainly help to determine whether the noun phrases extracted are in a contributing post?

- Finally, as discussed in Section 4.4, it would be a nice improvement to build a more robust lemmatizer that would be more sensitive to very specific edge cases, which could potentially mess up the meaning of many words (as seen in the "CS" to "C" example)

5.4 Summary and Takeaways

- The search result was fairly accurate compared to Facebook search, but the algorithm tends to favor shorter posts
- For topics extraction, the majority of popular topics seem very consistent with what are being most discussed on Facebook, even in the appearance of noises
- It was certainly a very fun and rewarding project, and once again, "nothing comes close to the excitement from working with live data that one sees everyday."

References

[1] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reily Media, 2009. URL http://nltk.org/book/.

[2] Fast Company. Twitter can predict the stock market, if you're reading the right tweets. Apr 2013. URL http://www.fastcoexist.com/1681873/twitter-can-predict-the-stock-market-if-youre-reading-the-right-tweets.

[3] C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. URL http://nlp.stanford.edu/IR-book/.

[4] P. Norvig. How to write a spelling corrector. URL http://norvig.com/spell-correct.html.

[5] MIT Technology Review. Twitter datastream used to predict flu outbreaks. Oct 2013. URL http://www.technologyreview.com/view/520116/twitter-datastream-used-to-predict-flu-outbreaks/.

[6] Wikipedia. Cosine distance. . URL https://en.wikipedia.org/wiki/Cosine_similarity.

[7] Wikipedia. Tfidf. . URL https://en.wikipedia.org/wiki/Tf%E2%80%93idf.