Ben Brattain
I427

Final Project

IMPORTANT NOTE: In order to pass this class and graduate, I unfortunately have to do slightly better than perfect on the final score of this project. On top of that, I was in a similar position with some of my other classes this semester because of a family crisis. As such, I had to maximize my points with not as much time as I wanted.

So, my search engine does not work perfectly. I figure in building this, it takes 90% of the time to get 90% of the way there, and then another 90% of the time to fix the final tweaks, add unit testing etc. Since, I didn't think that was the best way to maximize my points with my limited time, I did most of everything with some small errors and then pivoted right away to extra credit. Unfortunately, that breaks all the test cases which is why I'm writing this!

Speaking to Professor Nematzadeh, I was told that it was possible to get up to an additional 20% (on the final grade, not just the final project so VERY large amount) of extra credit if I could successfully implement Pagerank using Mapreduce. So, I chose to focus on that instead. I figure if I even get partial credit on it, I'm more likely to get the grade I need to get going this route than ironing out the final problems in my normal search engine. I'm really sorry if this causes any problems with grading.

In order to make it easier for you, I'll tell you where I observed there to be breaking points:

1. My index.py file is the module that builds my inverted index, amongst other things. What is really unfortunate is the document to word assignment in the index is not perfect. Some pages get through that should not be getting through, which completely breaks the tf-idf scoring and the final rank scoring which build off of the index. So, my output will be wrong even if everything else is right.
2. I could not get the myscript.cgi file to work on burrow. I'm not sure why because it's very similar to the demo.cgi file which did work, but I suspect it's because I run a python program from there and it's causing server issues. I can't even debug the server effectively so I chose to move on to Mapreduce. Needless to say, there's a page that loads if you go to my url, but it does not work. The myscript.cgi file is still included in this repo with the work I did, but I didn't bother to include the index.html file since it's so bare bones.

End Note.

Project:
Since I didn't get the final web gui to work and it's trivial to get input, I went ahead and created a wordlist for you. If you run python start.py, it will start the whole program and print out the computed ranks (again wrong because of the broken invindex). However, if you change your mind, you can go to the file, comment out wordlist under main and then uncomment the user input section. It should be clear.

I also have code in here that is commented out to work with burrow. It's labelled appropriately. The idea was to build the invindex and pagerank pickle file locally, upload those to burrow, and then get the input, compute the tfidf, and then the final rank from there. I left that code in so it can be evaluated.

To run the crawler you currently just need to type crawl.py. It's currently set to scrape 200 pages from The Washington Post using BFS. I decided to follow etiquette and scrape 2 pages a second like you recommended in the past assignments, but that can all be edited. Everything, but the speed can be changed in the craw file, and the speed can be changed in settings.py. I chose to use BFS because it was more robust in preventing errors and not getting caught in a loop.

I noticed the tradeoff in between BFS and DFS is that BFS is very stable and good and collecting lots of information over a small number of topics. However, if you want a very large set of diverse documents, it might not be the best design choice since most likely a lot of pages will link to other pages that are similar to them. If you're scraping all of the links on a page, you're more likely to have a lot of overlap. With DFS, it's really nice if you want lots of different topics in a smallish sample size. I don't know the exact rumor, but another professor told me in the past that you can pretty much get to anywhere on Wikipedia within 13 (estimate and maybe off) links. DFS is really good at arbitrarily picking a link and going down to random spots, getting you a good diverse set of documents. However, it's really bad about catching header links and getting caught in loops, or running out of links quickly if you don't allow duplicates in your stack. Since the assignment clearly said we should specialize, and BFS didn't get caught in headers, I chose to go with BFS.

To run the indexer by itself, uncomment the main function at the bottom and just run the program. You'll need to have run the crawler or have a set of pages, index.dat already made under the pages/ directory. This file takes care of the vast majority of getting all of the information that was scraped, cleaning it, putting them in the appropriate data structures for pagerank, and building the docs.dat and invindex.dat files. This program basically gets me all of the information I need for both the tfidf and pagerank programs. I decided it would be a lot faster if I just build the invindex and gathered the pagerank info all at once since they come from the same place. I use BeautifulSoup to scrape everything I need off of the pages. This file as I mentioned before, does not create the invindex perfectly.

The pagerank.py program can be run automatically if you uncomment the main lines on the bottom. As I mentioned, it relies on the index.py program to get the data I need to manipulate it. The program is pretty straightforward since I collected everything in index.py. I have to manipulate the data little bit, but basically I already have all of the inbound at this point. As such, I can find all of the documents since I passed them over in a hash, get the pageranks, and run the formula for each document. I iterate 50 times, and then write my output to a pickle file. Just so we're clear, this is the formula that I used:

Pagerank(document) = .15 + .85 * (Pagerank of inbound link doc/ num of out links that doc has … for each inbound link doc).

I chose to use .85 as my damper because that seemed like the norm everywhere I read. I also have a smaller PageDoc class so I can store all the pagerank information with less details than I had in the index.py file.

To run the tfidf doc, just uncomment the main section. I created a wordlist for you already. Feel free to change it if you want, but we already talked about the incorrect output anyways so that's sort of a waste of your time in my opinion. The way I created the hit list is simple. I have a class Word, for each word that is in my search query. Each word has a list of all the documents that it appears in from the invindex. For my hitlist, I made sure each document had all of the words, so I had a program where I pruned off documents from each word object that was not shared amongst all of the other words. You can see "make_hit_list" to see how this works.

After I make all of the words from the invindex and make the hit list of docs, I create a Documents object for each doc in the first word (same documents for each one so it doesn't matter) and the documents class computes the tf, idf, and tfidf scores on initialization. After that, I do some cleaning, write the info to a pickle file just in case, and return it so I can get it in my start function for the final ranking.

The last program is the start file. As I mentioned, I can get input here, and you can run the whole chain of programs from here. I also have a MergedRank class which computes my final ranking. I did a simple multiplication of the pagerank and tfidf scores. The reason I chose to do this was so that the pagerank and tfidf have the same average weight regardless of what query is inputted and the pages scraped. As I mentioned, the crawler relies on BFS so it's likely that all of the pages are going to be on average in a few categories. Depending on the query chosen, tfidf will either weigh a ton, or not much at all depending on the prevalence of the query. Since I don't control your input, I preferred to just average it out with the pagerank so tfidf doesn't have too much or too little impact.

Before I get on to the extra credit, I want to make sure I'm ok in the plagiarism department. I didn't end up submitted a scraper on my own for hw3. After that, I had Kyle Neely walk me through his since I figured I wasn't getting credit anyways. I ended up programming it on my own, but there's a good chance that the walkthrough influenced my implementation a decent amount. So, I'm letting you know now that I walked through his before I built mine.

Since I know my invindex is broken, and I switched to pagerank instead, I also did the same with robustness and prewritten tests. I don't have any. I'm hoping the extra credit from Mapreduce greatly outweighs the points lost from the lack of user tests, the broken invindex. It has impacted hw's 4 (got a 0 for being too late) and 5 (relied on it) so I'm hoping the invindex being broken doesn't have too large of a weight here since it's been graded a ton already. I think it's mostly ok, but the stemming or something of that sort might throw it off and if it's a little off, the whole program breaks. It should be mostly ok.

Extra Credit:

For extra credit, I implemented Pagerank using Mapreduce. I decided to make it a skeleton program, so I could work on it in as much isolation as possible. Since I rely on data gathered in index.py, that program remained. However, I wrote a new Mapreduce program and if you just run it from terminal, it should work. The pages/ directory are the files that you gave out for the old hw assignment so I was confident they would work.

I initially restructure my data a little bit into the appropriate key,value form for mapreduce. This was definitely the most difficult part of the assignment for me. It was infuriating and took a lot of trial and error since I'm not too experienced with mapreduce. I finally got it to work though.

The way it ran is pretty simple. I took all of the document urls and pageranks as a key with the values being all of the outbound links. I think paired the pagerank of each doc/outbound links and created pairs for each of those and created a large intermediary array.

I then sorted all of the intermediate values so the urls were grouped by all of their pageranks and I could reduce them into the final pagerank score. After that, I printed everything out. This took me right about until now (I had permission to work until the presentation so I was unable to do anything else).