EECS 485 Project 5: Wikipedia Search Engine

Project due 9pm Monday, April 17, 2017

You have now learned enough to build a scalable search engine that is similar to commercial systems. So, in this assignment, you will build an integrated web search engine that has several features:

- Indexing implemented with MapReduce so it can scale to very large corpus sizes
- Information retrieval based on both tf-idf and PageRank scores
- A new search engine interface with two special features: user-driven scoring and summarization.

Remember not to commit the Hadoop library, vagrant folders or large files (like the Wikipedia input or your inverted index) on your Github! It's 100s of MBs and you will not only be adding headaches for the staff but also for yourself as syncing that with your teammates will be painful. So don't do it. Just don't. Update your .gitignore ASAP.

Since your large input files won't be on Github, use the scp command line program to copy large input files and your new inverted index to the server.

Don't use the EECS server you were assigned to run MapReduce. Please run your MapReduce code locally and only run the IndexServer code on the EECS server. If you accidentally start running your MapReduce code on the EECS server, kill it.

Table of Contents

Part 0: Setup

You will create a new virtual machine for this project. The vagrant.sh for this new VM will download Hadoop, its dependencies and all of the starter files for this project. Download the `Vagrantfile`, `vagrant.sh`, and other starter files from the GitHub repo into an empty directory and run the command "vagrant up" to create the virtual machine. This will create the project with the following main files (all located in the `/vagrant` directory):

- `.gitignore`: This is very important! We are dealing with lots of very large files that should not be committed to github; they will slow down your pushes and pull tremendously, so make sure to add this gitignore to your github
- `/hadoop`: This folder will be created as a result the Vagrant setup. It should contain all of your mapreduce code. See Part 1 for details

- `/index_server`: This folder will have all of the Python code for the index server. See Part 2 for details.
- `/search_interface_server`: This folder should have your search interface app. See Part 3 for details.

Part 1: MapReduce Indexing

You will be building and using a MapReduce based indexer in order to process the large dataset for this project (over 300 MB!). You will be using Hadoop's command line streaming interface that lets you write your Indexer in the language of your choice instead of using Java (which is what Hadoop is written in). However, you are limited to using Python 3 so that the course staff can provide better support to students. **In addition, all of your mappers must output both a key and a value (not just a key).**

There is one key difference between the MapReduce discussed in class and the Hadoop streaming interface implementation: In the Java Interface (and in lecture) one instance of the reduce function was called for each intermediate key. In the streaming interface, one instance of the reduce function may receive multiple keys. **However, each reduce function will receive all the values for any key it receives as input.**

You will not actually run your program on hundreds of nodes; it's possible to run a MapReduce program on just one machine: your local one. However, a good MapReduce program that can run on a single node will run fine on clusters of any size. In principle, we could take your MapReduce program and use it to build an index on 100B web pages.

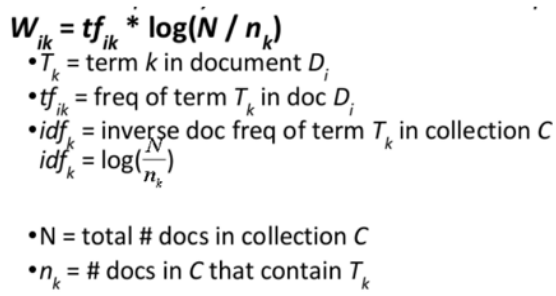
After you have your environment setup, examine the `hadoop/` directory. You will notice that there are many subdirectories, most of which are used by the hadoop library, and you will not need to modify them. These subdirectories are `bin/`, `etc/`, `include/`, `lib/`, `libexec/`, `sbin/`, and `share/`, and are all included in the `.gitignore`. The subdirectories that you will be working with are `data/` and `mapreduce/`. The `data/` directory contains the `map_reduce_input_data`. The `mapreduce/` directory is where the MapReduce executables and the `input/` subdirectory belong; the purpose of the `input/` subdirectory is explained in the Splitter section. There are a starter `map.py` and `reduce.py` in the `mapreduce/` directory which implement WordCount on the files in the `mapreduce/sampleInput/` directory. We can run this job by executing `run.sh`. Now open the `run.sh` file and examine the contents to see how the script works. **It is imperative that you understand this before proceeding!**

For this project you will create an inverted index for the documents in `map_reduce_input_data` through a series of MapReduce jobs. This inverted index will contain the idf, term frequency and document normalization factor as specified in the information retrieval lecture slides. The format of your inverted index must follow the format, with each data element separated by a space:

word -> idf -> doc_id -> number of occurrences in doc_id -> doc_id's normalization factor (before sqrt)

Sample of formatted output can be found in `/hadoop/sample.txt`. You can also use this sample output to check the accuracy of your inverted index.

For your reference, here are the definitions of term frequency and idf:



And here is the definition for normalization factor for a document:

$$\sqrt{\sum_{k=1}^{t} (tf_{ik})^2 [\log(N / n_k)]^2}$$

Part A: Splitter Script

Each document has three properties: **doc_id**, **doc_title**, and **doc_body**. Your mapper code will be receiving input via standard in and line-by-line. As a result, the input is newline separated and each document is represented by 3 lines: 1st is the **doc_id**, 2nd is the **doc_title**, and the 3rd is the **doc_body**. This will allow you to read the input in your mapper function easily.

Input for Hadoop programs is an input directory, not file. As you will notice, our input is in one large file called `map_reduce_input_data` but your code runs with multiple mappers, and each mapper gets one input file. Consequently, you must write a custom (recommended Python) script to break the large input file with over 3,000 documents (3 lines each) into smaller files and places these files into the `mapreduce/input/` directory. This splitting script will *not* be submitted to the autograder and thus you can name it whatever you please. Do not split the input file into one file per document; this will make your MapReduce pipeline run very slowly! We do not require a specific number of smaller input files, but ~30 is fine.

Part B: Document Count Job

The first MapReduce job that you will create counts the total number of documents. This should be run with 30 mapper workers and only ONE reducer worker. The mapper and reducer executables should be named `map0.py` and `reduce0.py` respectively. The reducer should save the total number of documents in a file called `total_document_count.txt`. The only data in this file will be an integer representing the total document count. This job will be executed using `run.sh` and thus the `total_document_count.txt` file will be created in the `hadoop/` directory, not the `hadoop/mapreduce/` directory.

Part C: Pipeline Jobs

You will be going from large datasets to an inverted index, which involves calculating quite a few values for each word. As a result, you will need to write several MapReduce jobs and run them in a pipeline to be able to generate the inverted index. The first MapReduce job in this pipeline will get input from `mapreduce/input/` and write its output to a directory that you specify in `run.sh`. All future jobs in this pipeline will use the output directory from the previous job as the input directory for the current job, until the inverted index is constructed. Inspect the sample `run.sh`, which shows how to pipe the output from one MapReduce job into the next one.

To test your MapReduce program, we recommend that you make a new test file, with only 10-20 of the documents from the original large file. Once you know that your program works with a single input file, try breaking the input into more files, and using more mappers.

Each of your MapReduce jobs in this pipeline should have 30 mappers and 30 reducers. However, your code should still work when the number of mappers/reducers is changed. You may only use a maximum of 9 MapReduce jobs in this pipeline (but the inverted index can be produced in fewer). The first job in the pipeline (the document counter) must have mapper and reducer executables named `map0.py`, `reduce0.py` respectively, the second job should be `map1.py`, `reduce1.py`, etc.

The format of your inverted index must follow the format, with each data element separated by a space:

word -> idf -> doc_id -> number of occurrences in doc_id -> doc_id's normalization factor (before sqrt)

Sample of formatted output can be found in `/hadoop/sample.txt`. The order of words in the inverted index does not matter. If a word appears in more than one doc it does not matter which **doc_id** comes first in the inverted index. Note that the log for **idf** is computed with base 10 and that some of your decimals may be slightly off due to rounding errors. In general, you should be within 5% of these sample output decimals.

When building the inverted index file, you should use a list of predefined stopwords to remove words that are so common that they do not add anything to search quality ("and", "the", etc). We have given you a list of stopwords to use in `stopwords.txt`. This file is found in the `/hadoop` directory, not the `/hadoop/mapreduce` directory. So when opening it in your MapReduce executables, use the filename '`stopwords.txt`', not '`mapreduce/stopwords.txt`'

When creating your index you should treat capital and lowercase letters as the same (case-insensitive). You should also only include alphanumeric words in your index and ignore any other symbols. If a word contains a symbol, simply remove it from the string. Do this with the following code snippet:

```
>>> import re
>>> re.sub(r'[^a-zA-Z0-9]+', '', word)
```

You should remove non alphanumeric characters before you remove stopwords. Your inverted index should include both **doc_title** and **doc_body** for each document.

To construct the inverted index file used by the index server, 'cat' all the files in the output directory from the final MapReduce job, and put them into a new file (i.e. `cat mapreduce/final_output_dir/* > outfile.txt`) . You can name outfile whatever you like and move it to your `index_server/` directory to be used in the index server.

Deliverables

- Write a custom script to convert the one giant input file into multiple smaller files.
- Build a pipeline of mappers and reducers that go from newline-separated input to an inverted index with the same exact format as sample.txt.
- Submit: mapreduce.tar.gz: This should be a tarball containing:
    - mapreduce/ -- this should be the folder containing mapper & reducer files but do not upload your input or output folders. This folder can also contain stopwords.txt.
        - Each map and reduce file should follow the following naming scheme: map0.py, reduce0.py, map1.py, reduce1.py,...map9.py, reduce9.py. If your MapReduce files have different names, the autograder will not be able to run them.
- Note that your input is from the hadoop/mapreduce/input/ folder and your final reducer should send the inverted index to the hadoop/mapreduce/output/ directory.

Remember: When implementing this part, please do not run MapReduce tasks on your assigned EECS server. Run it locally!

Part 2: Index Server

The index server is a separate service from the search interface that handles search queries and returns a list of relevant results.

Part A: PageRank Integration

In lecture, you learned about PageRank, an algorithm used to determine the relative importance of a website based on the sites that link to that site, and the links to other sites that appear on that website. Sites that are more important will have a higher PageRank score, so they should be returned closer to the top of the search results.

In this project, you are given a set of pages and their PageRank scores in `pagerank.out`, so you do not need to implement PageRank. **However, it is still important to understand how the PageRank algorithm works!**

Your search engine will rank documents based on both the query-dependent tf-idf score, as well as the query-independent PageRank score. The formula for the score of a query q on a single document d should be:

$$Score(q, d, w) = w * PageRank(d) + (1 - w) * tfIdf(q, d)$$

where **w** is a decimal between 0 and 1, inclusive. This value **w** will be a URL parameter. The final score contains two parts, one from pagerank and the other from a tf-idf based cosine similarity score. The **PageRank(d)** is the pagerank score of document **d**, and the **tfIdf(q, d)** is the cosine similarity between the query and the document. Treat query **q** as a simple AND, non-phrase query.

Integrating PageRank scores will require creating a second index, which maps each **doc_id** to its corresponding precomputed PageRank score, which is given to you in **pagerank.out**. You can do this where you read the inverted index. This index should be accessed at query time by your index server.

Part B: Returning Hits

When the Index server is run, it will load the inverted index file, pagerank file, and stopwords file into memory and wait for queries. It should only load the inverted index and pagerank into memory once, when the index server is first started. Every time you send an appropriate request to the index server, it will process the user's query and use the inverted index loaded into memory to return a list of all of the "hit" **doc_ids**. A hit will be a document that is similar to the user's query. When finding similar documents, only include documents that have every word from the query in the document. The index server should not load the inverted index or pagerank into memory every time it receives a query!

Search Endpoint

Route: http://{host}:{port}/{secret}/p5/?w=<w>&q=<query>

Your index server should only have one endpoint, which receives the pagerank weight and query as URL parameters **w** and **q**, and returns the search results, including the relevance score for each result (calculated according to the previous section). For example, the endpoint /?w=0.3&q=hello would correspond to a pagerank weight of 0.3, and a query "hello".

Your index server should return a JSON object in the following format:

```
{
  "hits": [ {
            "docid": 0,
            "score": 0.8931
          },
 {
            "docid": 2,
            "score": 0.6408
          }
           ]
}
```

The documents in the hits array must be sorted in order of relevance score, with the most relevant documents at the beginning, and the least relevant documents at the end. If multiple documents have the same relevance score, sort them in order of **doc_id** (with the smallest **doc_id** first).

When you get a user's query make sure you remove all stopwords. Since we removed them from our inverted index, we need to remove them from the query. Also rid the user's query of any non-alphanumeric characters as you did with the inverted index.

Make sure that stopwords.txt is in the same directory as your index server executable.

The index server can be deployed on your EECS server in the same way that you deployed your photo album app in earlier projects, using gunicorn and your assigned port number. For example:

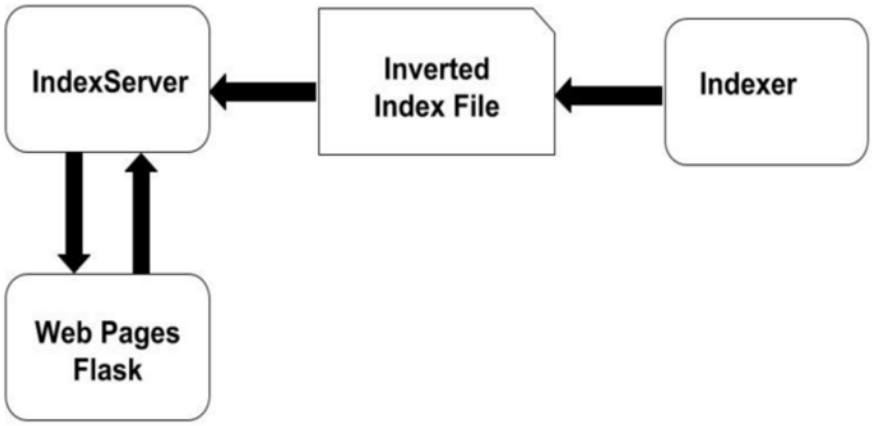
```
gunicorn -b class3.eecs.umich.edu:3000 -w 2 -D app:app
```

Deliverables

- Write an index server which uses the inverted index and pagerank files to return an appropriately formatted JSON object with a vector of documents that are similar to the search query.

Part 3: Search Interface

The third component of the project is an HTML interface for the search engine. The search interface app will provide a GUI for the user to enter a query and specify a Pagerank weight, and will then send a request to your index server. When it receives the search results from the index server, it will display them on the webpage. You should not use any Javascript to implement your search interface.



Part A: MySQL Updates

You will need to create a new database for project 5, with a table called **Documents** as follows:

- **docid**: INT and PRIMARY KEY
- **title**: VARCHAR with max length 100
- **categories**: VARCHAR with max length 5000
- **image**: VARCHAR with max length 200
- **summary**: VARCHAR with max length 5000

The SQL to create this table, and load the necessary data into it is provided in `wikipedia.sql`.

Part B: Routes

The following is a list of the url endpoints that you should create in your application. Be sure to check "Deliverables" below to ensure you have all the appropriate HTML elements and IDs before you submit your project to the autograder.

Search Page

Route: http://{host}:{port}/{secret}/p5/wikipedia?q=<query>&w=<w>

You will be making a simple search interface for your database of Wikipedia articles which allows users to input a **query** and a **w** value, and view a ranked list of relevant docs. Users can enter their **query** into a text input box, and specify the **w** value using a slider. You can assume anyone can use your search engine; you do not need to worry about access rights or sessions like past projects. Your engine should receive a simple AND, non-phrase query (that is, assume the words in a query do not have to appear consecutively and in-sequence), and return the ranked list of docs. Your search results will be titles of Wikipedia documents, displayed exactly as they appear in the database.

This page must include a slider (`<input type="range">`) that will set the **w** GET parameter in the URL, and will be a decimal value between 0-1, inclusive. You should set the range slider's step value to be 0.01. To summarize, the page will have a `<form>` of type GET with two inputs (**w** and **q**).

If there is no GET query parameter present, only display the search form. On a valid search, you will send the request to your newly modified index server, which will reply back with an array of search results. You will use the **doc_ids** returned from the index server to retrieve the data in your SQL database that corresponds to the same **doc_id**, and show at most 10 results. Each result should link to the corresponding summary page described below.

If there are no search results, display a message saying so in a `<p>` tag.

Summary Page

Route: http://{host}:{port}/{secret}/p5/wikipedia/summary?id=<docid>

Each link on the /wikipedia page should be directed to this dynamic page that uses URL parameters. On this page, you should display each column of the database in a separate `<p>` element (except `<img>` for the image). The categories of the doc should all be displayed together in one `<p>` tag, not separated into an individual `<p>` tag for each category. If a doc does not have an entry for a certain column in the database, then that element should not appear on the page, and the corresponding `<p>` tag should not be present. In addition to having this summary, this page will also have a "**Similar Documents**" section, which will show at most 10 documents using **w=0.15**, and the current document's title as the search query, with the underscores replaced with spaces. For example, if the title of the document is "Saint_Petersburg", the search query should be "Saint Petersburg". Each of these similar documents should link to their respective summary pages. On this summary page, do not display the current document in the "Similar Documents" section.

Routes

Make sure that all these URLS are present in your web application:

- **/wikipedia** Search interface and potential search results
- **/wikipedia/summary** Article summary and list of similar articles

HTML ID Attributes

Make sure that all these element IDs are present in your HTML templates. Additionally, when you view the live page, verify that they are present via (Right Click->View Source).

- /wikipedia
  - The input search bar should have an id "search_bar"
  - The range input for w should have an id "search_w"
  - The submit button to perform the search should have an id "search_button"
  - Each link to a search result's summary page should have an id "result_<docid>_link" and class "search_result"
  - The message displayed when there are no search results should have an id "no_search_results"
- /wikipedia/summary
  - The title of the doc should have an id "doc_title"
  - The categories for the doc should have an id "doc_categories"

- The summary of the doc should have an id “doc_summary”
- The image for the doc should have an id “doc_image”
- Each link to a similar doc’s summary page should have an id “similar_<docid>_link” and class “similar_doc”

In order to deploy the search interface app on your EECS server, you will need to use a different port number, since your assigned port should be used for your index server. Set the port to be your assigned EECS server port + 1.

Submission

We will post a link to the autograder when it is ready for this project. Please submit the following to the autograder:

- Your mappers and reducers, named map1.py, reduce1.py, …, map9.py, reduce9.py.
  - You can have a maximum of 9 mapreduce jobs, but you should not need that many.
  - The files must be named according to this pattern, or the autograder will not be able to test your mapreduce program!

Your index server and search interface will be tested via direct interactions with your deployed endpoint, and interactions with your deployed search interface app. See previous sections for deployment information.

In the README.md at the root of your repository please provide the following details:

- Group Name (if you have one)
- List the contribution for each team member:
  User Name (uniqname): "agreed upon" contributions
- Any need-to-know comments about your site design or implementation.

Please do not modify the files in your git repository or deployment after the project is due! The deployed version of your app also cannot have modifications after the due date.