EECS 485 Project 6: The Next Great Search Engine

Due 9pm Monday, April 18, 2016

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 deep summarization.

In Projects 4 and 5, we computed tf-idf and PageRank but we have not done so together. In Project 6, we will fix this oversight and build a search engine based on the Wikipedia graph you saw in project 5.

You will continue to use your Project 4 codebase and modify it to account for the new Pagerank scores and new web endpoints. However, instead of using the Indexer you wrote in project 4, you will write a new Indexer for the Wikipedia dataset using MapReduce and Hadoop.

BEFORE YOU READ ANY FURTHER: Remember and promise 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.

CONSEQUENTLY: 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.

ADDITIONALLY: For this project, gunicorn and Index Server can take a long time to start up due to large input files, so as a result, you will be assigned new EECS servers before the release of the autograder.

ONE MORE THING: Don't use the old servers for running your Index Server or the MapReduce code. Kill the process if you accidently do it. You will be using new servers that can handle the computationally expensive processes in this project. Even then, we recommend running MapReduce code locally and only running the IndexServer code on the EECS servers.

ANOTHER ONE: Nah, just kidding. 4 important things are enough.

Part 1: MapReduce Indexing

Instead of using the C++ Indexer you wrote in project 4, you will now be building and using a MapReduce based indexer in order to process the incredibly 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 C++ or Python for this project so that the course staff can provide better support to students.

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 find the discussion slides from last week very helpful.

You should reimplement your inverted index software using the MapReduce framework, i.e. use the new dataset and generate a similar inverted index. 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.

In discussion, we introduced you to the Hadoop streaming interface used in this project. If you were not present for that, please follow the tutorial given in the discussion folder as that will be the basis of your MapReduce code. Go through the tutorial thoroughly, and only once you completely understand it attempt to write your own indexing code. First get the sample MapReduce code running, which does a basic word count based on the given input (which is not how our input is). We recommend that you continue to use the Vagrant setup shown in discussion because installing Hadoop locally is a very messy process. We have provided a whole set of starter files, along with suggested file structure in Google Drive.

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 programs and run them in a pipeline to be able to generate the inverted index. The first mapper/reducer set will get input and generate output. The second one will receive the first's output as its input and so on. We left a sample of this in the run.sh file (redo your Vagrant setup to see the changes). We have also modified the run.sh file to run with multiple mappers and multiple reducers.

Hint: Your first job should be a document count mapper/reducer in order to calculate idf and store that result in a file so that future mapper(s)/reducer(s) can use that value.

Hadoop Input

The input of your MapReduce program is given in Google Drive (map_reduce_input_data). Your input is a modified version of mining.articles.xml. The XML file defines a set of documents in the Wikipedia graph and each document has three properties: article_id, article_title, and article_body. If you remember, your mapper code will be receiving input via standard in and line-by-line. As a result, we created an alternate copy of the input which is newline separated and each document is represented by 3 lines: 1st is the ID, 2nd is the title and the 3rd is the content. This modification will allow you to read the input in your mapper function more easily.

Remember that the input for Hadoop programs is an input *directory*, not file. As you will notice, our input is in one large file but your code runs with multiple mappers and each mapper gets one input file. Consequently, part of the project is writing a custom (recommended Python) script to break the large input file with over 3,000 documents (3 lines each) into smaller files (starting point should be 300 documents per file) and use this newly created directory of documents as your Hadoop input. For the autograder, you will be given an input directory with the same name (/hadoop/mapreduce/input) with an unknown number of files and unknown number of documents per file.

For testing, we recommend you split the big file into a single test one with 10-20 documents. Once you know your code works with one input file, try breaking the input into more files and using more mappers.

Deliverables

- Write a custom script to convert the one giant input file into multiple smaller files.
- Build a pipeline of mappers and reducers (either in C++ or Python) that go from newline-separated input to an inverted index with the same exact format from Project 4.
- A bash script that runs the mappers/reducers and generates the inverted index (similar to run.sh from discussion). We will run this exact file, where the first input will be in the /hadoop/mapreduce/input folder and the final output should be in the /hadoop/mapreduce/output folder. If you have intermediate outputs, please name them differently.

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

Part 2: Integrated Ranking

Your new 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 parameter like the query in your project 4 C++ server. 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.

In Project 4, your C++ server received a query string **q** as input and emitted a ranked list of documents as its response. In this project, you will continue to treat query **q** as a simple AND, non-phrase query as you did in Project 4. However, your new C++ server should also take a parameter **w**, and use the new formula above for search retrieval.

You will continue to use the Index Server from Project 4, but you will modify it a bit to handle PageRank and **w**. Remember, you will continue to handle stop words and build a case-insensitive inverted index in your Index Server. Integrating PageRank scores will require creating a second index, which maps each document id to its corresponding precomputed PageRank score (you can do this where you read the inverted index). This index should be accessed at query time by your C++ server.

In order to create this new PageRank file, you will be using the mining.edges.xml file from Google Drive. We highly recommend writing a Python script to convert the XML file to Pajek format and then use your P5 code to calculate the new PageRank scores (using d=0.85 and 50 iterations). Your output should have 30,109 lines. We will also give you a sanity test case to make sure you are doing PageRank right.

Warning: After finishing part 2 with the new data set, you will get a huge inverted index, which will result in a larger start up time for your Index Server.

Deliverables

- Modify your Index Server to accept a **w** parameter, in addition to the query parameter.
- Use the mining.edges.xml file to create a new PageRank file needed by the Index Server (preferably by going from XML to Pajek format using a custom script).
- Modify your Index Server to read PageRank scores on initialization and then use those scores in your search retrieval algorithm.

Part 3: The New Search Interface

The third component of your project is a new HTML interface for the search engine. You will continue to use your project 4 website and build a new /wikipedia endpoint. This new page will be similar to the /search page in project 4 but instead of showing image results, it will show Wikipedia links based on the new dataset for this project.

However, your /wikipedia page will have two features that mark it as new.

1. **Slider:** The slider to control the parameter **w**, as described in Part 1. You will use the <input type="range"> element for a slider. The values range from 0 to 1 with a 2 decimal precision (step size of 0.01). This is in addition to the query input field.
2. **Deep Page Summary**: Unlike standard search engines, you know that your search engine will be used on Wikipedia pages. Because Wikipedia pages have a more regular structure than standard Web pages, your summary page can be much more interesting than what Google can do for a standard page.

MySQL Updates

Remember how in project 4 your data was captions and images? Well, now it's data about Wikipedia. In the Google Drive folder, you will find a folder of various XML files, containing data about Wikipedia documents. You will be creating a new 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 data given is divided into various XML files as follows:

- mining.articles.xml: Contains **docid** and **title** for each document. This document contains every single article in the corpus.
- mining.category.xml: Contains **categories** for each document. Note that the same document can have multiple categories and you will store a comma separated list in the categories column of the SQL table (cat1, cat2, cat3).
- mining.imageUrls.xml: Contains images for each document. You will store the first image for each document in the **image** column of the table. Not every document has to have an image.
- mining.infobox.xml: the <eecs485_article_summary> node in this XML file is the **summary** column in the SQL database.

You will be creating a new SQL file as follows:

- wikipedia.sql: this file should create the table above and add the data.

We highly suggest writing a custom Python script to iterate through all the files, build an in-memory dictionary and then create a SQL file dynamically using Python. When initially building your dictionary, you should start by loading mining.articles.xml; this file contains every single document (and corresponding docid).

Note: We will not be testing your SQL files directly due to the large datasets but will be testing them indirectly via your website.

New Search Page: /wikipedia?q=happy&w=0.85 [GET only]

This page will be similar to the /search endpoint from project 4. In fact, you can copy that page and start from there. However, due to the different content, your search results will be titles of Wikipedia documents as well as a short 200 character summary (so make a substring of the summary if it is longer than 200 characters, similar to how Google produces their results).

New to this page is the <input type="range"> that will be 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 /wikipedia page will have a <form> of type GET with two inputs (**w** and **q**).

Like project 4, if there is no GET query parameter present, only show the search form. On a valid search, you will send the request to your newly modified Index Server that will again reply back with an array of search results. You will again correlate the responses from the Index Server to the data in your SQL database and show at most 10 results. Each link should go to the deep summary page described below.

If there are no search results, display a friendly message saying so.

Deep Summary Page: /wikipedia/summary/<doc_id>

Each link on the /wikipedia page should be directed to this dynamic page that uses URL parameters, as shown here. On this page, you should display each column of the database in a separate <p> element (except <img> for the image). In addition to having this “deep summary”, this page will ALSO have a “Similar Documents” section, which will show at most 10 documents using the current document’s title as the query (and a w=0.15)! Each of these similar documents should link to their respective deep summary pages.

Element IDs

Please have the following element IDs on the new pages:

- **/wikipedia Page**
    - Input search bar: “wikipedia_search_input”
    - Range input for w: “wikipedia_search_w”
    - Search button: “wikipedia_search_button”
    - Each result link will have an <a> tag that goes to the deep summary page will have the following ID: “result_<docid>_link” where <docid> is document ID of the search result.
    - <p> tag for no search results: “no_search_results”
- **/wikipedia/summary Page: Each of these <p> elements should have the exact content from the database (and should not exist if the column is empty):**
    - <p> element for title with ID: “doc_title”
    - <p> element for categories with ID: “doc_categories”
    - <p> element for summary with ID: “doc_summary”
    - <img> element for the image (if and only if there is an image) with ID: “doc_image”
    - Each similar document link will have an <a> tag that goes to the deep summary page of that document. It will have the following ID: “similar_<docid>_link” where <docid> is document ID of the similar document

Deliverables

- Writing a custom script to go from the XML data to SQL statements (or you can try to use LOAD XML, which will be tough) and update your databases.
- New /wikipedia page with a search bar that shows at most 10 search results based on the query and PageRank weight parameter.
- New /wikipedia/summary/<doc_id> page that shows the deep summary of the document in question and also similar documents, which is essentially the search results using the document’s title as the query and w = 0.85.

Part 4: Starter Files and Project Structure

There's a lot going in this project so we recommend the following file structure:

- `/`: This is the root directory of your project. This folder will be where you put the two vagrant files and then do 'vagrant up'. This directory will then have the hadoop folder.
- `/flask`: This folder should have your Flask app, your P4 directory except the C++ code
- `/index_server`: This folder will have all C++ code for the Index Server
- `/pagerank`: Your pagerank code and related scripts
- `/hadoop`: This folder will be created as a result the Vagrant setup.
- `/data`: All the data given to you and mentioned in this doc. If you ever move this (and you may have to for mapreduce, etc.), make sure to update your gitignore!

We have preserved this file structure and given you starter files. We also left some .gitignore files but please don't commit massive files to Github.

Note: If you use Vagrant for your Flask app, you can avoid setting up two Vagrant images and continue using the VM set up by the Hadoop files (which will also create a virtualenv). You can still keep your Flask app separated in the /flask folder (i.e. you don't NEED to create venv in the folder of your app, you just need to activate the venv shell before working on the app).

Part 5: Submission

We will post a link to the autograder when it is ready for this project, along with a list of what you need to submit to the AG, including the following:

- Your /indexer/hadoop/mapreduce folder without the input or output folders (so only the mappers/reducers with stopwords.txt file, which you <u>will</u> need).
- Your PageRank output for a sanity test case

Your index server and server will be tested via direct interactions with your deployed sites.

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.