CSCI 8360 Data Science Practicum
Department of Computer Science
University of Georgia

Project 0: Introduction to CSCI 8360

DUE: Tuesday, January 16 by 11:59:59pm

Out January 9, 2018

1 Overview

There are a lot of technologies we'll be using this semester that may be new to you. This first "project" is meant as a gentle introduction to each of them, as you'll be using them for each of the four projects that follow.

This first project **will also be done INDIVIDUALLY**, so everyone starts from the same baseline. This will **not** be the case for any of the subsequent projects!

2 Your Mission: Word Count

Your mission for this project, then, is to implement a basic word counter in Apache Spark and submit the resulting word count dictionary to AutoLab. Download and install Spark on your local machine. You can use any of the Spark APIs (Scala, Python, Java, R) for this project, though some will make your work easier than others.

Natural language processing (NLP) is a huge component of data science: using statistics and linear algebra to make sense of language, which is messy and ambiguous and full of self-contradictions. In short, the best and worst of what machine learning can do.

One of the most basic (but nonetheless successful) ways of "getting a feel for" unstructured text is to perform word counting. This is both intuitive to us–to some extent, we associate importance with frequency, and so the words that are most frequent will provide a nice summary of the text on a whole–while also having some foundation in statistical

theory–the distributions over the words, as determined by their relative frequencies, are directly linked to identifying topics and discerning meaning.

Word counting proceeds pretty much how it sounds: you split the text into "tokens" (usually individual words), and count the number of times each unique word appears. You can often organize this structure in a dictionary / hash-map / key-value pair, where the "key" is the word and the "value" is the count of that word.

For experienced functional programmers, you could probably whip this up in one, maybe two lines of code. In a distributed environment, like the one we'll be simulating using Apache Spark, it gets a little trickier.

2.1 Data: Project Gutenberg

For this project, we're using freely available data from Project Gutenberg: https://www.gutenberg.org/ . Specifically:

1. *Pride and Prejudice*, by Jane Austen
2. *Alice's Adventures in Wonderland*, by Lewis Carroll
3. *Ulysses*, by James Joyce
4. *Leviathan*, by Thomas Hobbes
5. *The Iliad*, by Homer
6. *The War of the Worlds*, by H.G. Wells
7. *The Republic*, by Plato
8. *Little Women*, by Louisa May Alcott

You can download them all via the `handout.zip` archive posted to the course GitHub repository. This URL should also be mirrored through the Project 0 page on AutoLab.

2.2 Code: GitHub

Develop your code in a GitHub repository (under the DSP-UGA organization). Give it a name that follows the scheme: `<username>-p0` so I know who it belongs to. Make sure you include a basic `README` file that explains what it does and how it works, and any problems you ran into. Include a `CONTRIBUTORS` file with the names of everyone who worked on the project and what they did (yes, I know this will just be you, but it's good to get into the habit for future projects). Finally, include all the code you develop to address the project (document it!).

2.3 Goal: Output

Your Spark program should 1) tokenize the input, 2) conduct basic text preprocessing (see below), 3) count up the remaining words, and 4) serialize the word counts as output to a JSON file.

Subproject A: You'll need to generate a dictionary / hash-map of the top 40 words across all documents with the largest counts (to clarify: this is the top-40 across *all* documents, meaning you're basically ignoring specific documents and throwing all their words into one big bag, then pulling out the top 40). The only filtering step you'll perform is to drop words with a total count less than 2 (this is purely for performance, since the top 40 words will have counts well above 2; but also to give you practice with Spark). Your word counts should be case-insensitive, but otherwise you don't need any additional preprocessing. The JSON file should contain a list of key-value pairs, where each key is the word and the value is the count. Name the JSON file `sp1.json`.

Subproject B: Implement a list of stopwords (use the `stopwords.txt` file). These are words like "the", "an", and "is" that are common across all written works and therefore do not carry much meaning about the specific text. Re-generate the list of top 40 words across all documents, dropping words entirely that are found in the provided list of stopwords. As before, your counts should be case-insensitive, but otherwise no additional preprocessing is needed (*HINT*: you will need to look at broadcast variables to do this). The JSON file should contain a list of key-value pairs, where each key is the word and the value is the count. Name the JSON file `sp2.json`.

Subproject C: You may have noticed by now that some of the top words in the previous subprojects contain trailing punctuation, like periods or commas. Implement an additional bit of preprocessing that strips out periods (`.`), commas (`,`), colons (`:`), semicolons (`;`), single quotes (`'`), exclamation points (`!`), or questions marks (`?`), but only if they are the *first* or *last* character of the word (this will leave contractions like "can't" or "won't" unaffected). You may notice that in doing so, you have to intrinsically discard all words that have only 1 character; do that as well. Submit the resulting top 40 words. The JSON file should contain a list of key-value pairs, where each key is the word and the value is the count. Name the JSON file `sp3.json`.

Subproject D: You're on a roll! This subproject is the most difficult, however. Up until now, you have been computing what is known as *term frequency*: word counts are measures of frequency of the words (terms), but they are an imperfect measure of importance, as you have already discovered with stopwords. What is more informative is *term frequency inverse document frequency*, or TF-IDF. This weights the term frequency by the inverse number of documents the word appears in; a word that appears a lot in one document, but never in any of the others, is very clearly an important word in that document! Contrast that with a stopword, which appears a lot, but appears in all documents. This would, as a result, be downweighted.

The IDF term looks like this: $\log \frac{N}{n_t}$, where N is the number of documents (in this case, 8 books), and n_t is the number of documents the specific word t appears in (therefore, this should be between 1 and 8, inclusive). You will also have to compute *document-specific* term frequencies (i.e., word w_i appears n_j times in document j, but n_k times in document k; you'll need both those counts!). Then, you multiply the IDF term by each document-specific TF term, and that gives you the TF-IDF score for each document.

In this case, your output should still contain 40 words, but should contain the top-5 words *in each document* by their TF-IDF rank (don't worry, none of the top-5 words are the same between documents). The JSON file should contain a list of key-value pairs, where each key is the word and the value is its TF-IDF score. Name the JSON file `sp4.json`.

2.4 Hints

All text processing logic MUST happen inside Spark executors (yes, even finding the top 40 words!). Conversely, the **ONLY** logic permitted in the driver is performing the file I/O for the 40 words in each subproject to JSON files (obviously the driver will orchestrate RDD operations; this is not considered part of the text processing "logic").

The only hard-coded values that are allowed is the number 40 (all subprojects), the number 5 (subproject D), the list of punctuation to strip out of words (subproject C), and the JSON output filenames (see the "Submissions" section below). **All other values must be determined programmatically** (yes, even the document IDs for subproject D; take a look at the Spark documentation for reading in files if you need some help here). Use command-line arguments to direct your program to the text files.

Spend as much time as possible reading the Spark documentation. Also, ask questions in the Slack chat!

You can read in text files using `textFile`, or `wholeTextFiles` to grab a bunch at once.

`map` is your friend for operations that take one input and generate one output. If you're generating any number of outputs for a given input, then you want `flatMap`.

Know the difference between `reduce` and `reduceByKey`.

`collect` and `collectAsMap` are extremely useful functions, but be careful; if you just want to peek at your data in the RDD to make sure it's formatted correctly, then `take` is your friend.

`broadcast` will send a read-only copy of a variable to all the nodes in the cluster. When you're actively pulling it out, access the variable using its `.value` attribute.

If you're stuck on the TF-IDF portion, consider a generalization of the key-value paradigm: keep the words as keys in the RDD, but instead of a single count for the value, store an *array* of counts (one for each document).

3 Tools Involved

I'll give you a brief rundown of some of the tools we went over in Lecture 1 that you'll be making use of here.

3.1 Course Website

Our class website is located here: https://dsp-uga.github.io/sp18. All materials, links, and scheduling will be mirrored here.

Yep, that's the website. So that's cool.

3.2 Apache Spark

Apache Spark (http://spark.apache.org) is an in-memory distributed processing system. It competes directly with MapReduce (*not* Hadoop), and is considerably more flexible through its embrace of many functional primitives and their implementation on a general distributed compute engine. Best of all, it's open source and completely free!

However, programming in Spark can be tricky, especially if you're not familiar with functional or distributed programming. You will first need to familiarize yourself with functional primitives like `map`, `reduce`, `groupBy`, `filter`, and others (these are functional constructs that are *not unique to Spark*, but rather which are implemented *on Spark's distributed engine*). A great starting point are the basic Spark examples: https://spark.apache.org/examples.html

Spark used to be hard to become familiar with, but particularly as of v2.0 their documentation has been vastly improved. Installation is a snap, and the program docs are excellent: https://spark.apache.org/docs/latest/

For this class, we'll be using **Spark 2.2.1** (released December 2017).

3.3 git and GitHub

`git` is a concurrent, distributed versioning system, similar in a lot of ways to Subversion, Mercurial, Perforce, CVS, or others you may be familiar with. If you have never used a versioning system before, I highly recommend at least one of the following tutorials:

1. https://try.github.io is an interactive, web-based tutorial for git. I would highly recommend this one before any other.

2. https://git-scm.com/docs/gittutorial is a good reference for git commands and basic use.

3. https://www.atlassian.com/git/tutorials has a wide variety of tutorials available and is also a good reference.

Git is cross-platform and easy to install on most machines; **everyone will need to install it!**

GitHub, by comparison, is the most popular online repository for code, and it uses git as its synchronization tool. We in CSCI 8360 have our own organizational account, which can be found here: https://github.com/dsp-uga. As part of this assignment, you will need to

- Install git on your local machine
- Create an account on GitHub
- Ping me with your GitHub account name so I can add you to the DSP-UGA organization

ALL code generated in this course will go in the aforementioned GitHub organization. You will have the ability to create repositories and edit them, as well as make them visible only to you and your team during each of the projects. These repositories will then be the focus of grading for each project.

3.4 AutoLab

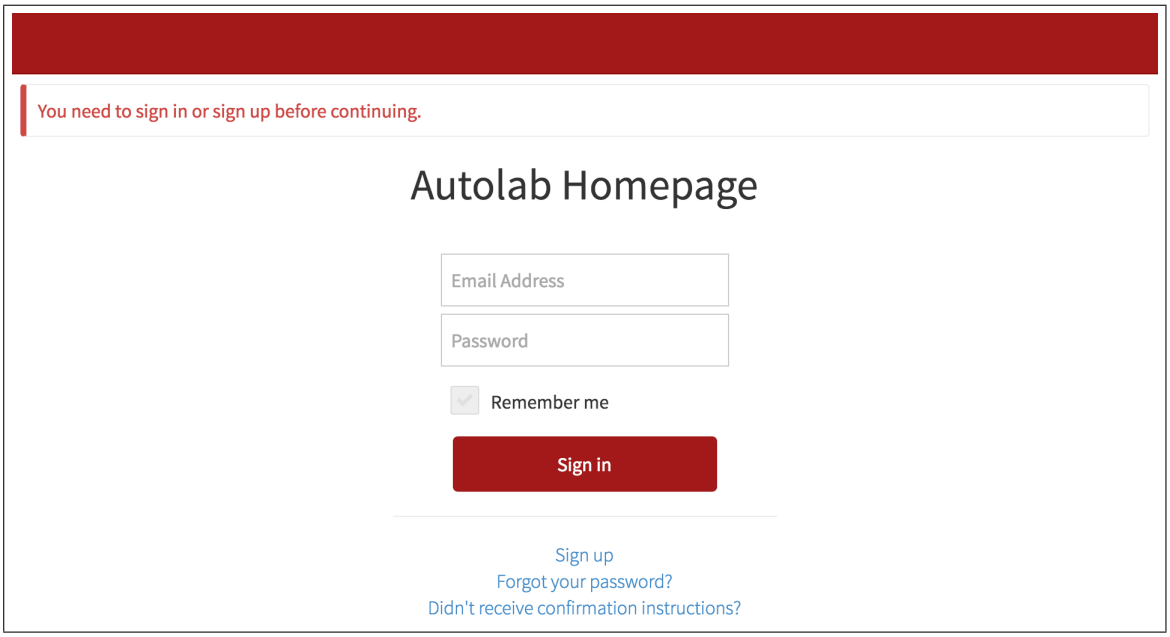
AutoLab is a project out of Carnegie Mellon that attempts to provide a nice web front-end to course management in addition to incorporating customizable autograders. It also has the ability to run arbitrary programs (in the form of Docker containers) on cloud services like AWS (see next subsection). While we won't make use of this feature, we will make use of another particularly competitive switch: the ability to implement a *leaderboard*.

This leaderboard will be used to rank submissions, similarly to how you would see the rankings on a Kaggle competition. Use this to see how your peers are doing, and if you could perhaps tweak your code to produce a better submission!

This is the AutoLab URL we'll be using: https://autolab.cs.uga.edu/ If you can't access it, **please let me know!**

3.5 Google Cloud Platform (GCP)

GCP is a cloud computing service offered by our benevolent one-world government, Google. Its offerings have expanded over the years, but at the core it is still an on-demand elastic computing service: you can spin up virtual compute "instances", perform



some work, and spin them down when finished Definitely have a look at all their individual product offerings: `https://cloud.google.com/products/`.

I would highly recommend checking out the developer documentation and installing the Google Cloud SDK, which usually includes a command-line utility for interacting with the various products from the comfort of your command prompt. However, we won't be making explicit use of GCP in this project, so don't stress about this too much just yet.

You are welcome to use a cloud compute service besides GCP! If you prefer Amazon Web Services (AWS), Microsoft's Azure platform, or even the DataBricks cloud (the corporate support behind Apache Spark), you're more than welcome to.

4 Submissions

Submit your three (or four) JSON files under **Project 0** on AutoLab. **IMPORTANT**: They should follow the prescribed format of being named `sp1.json`, `sp2.json`, `sp3.json`, and `sp4.json`. They should also be in proper key-value format. Finally, they should be archived using the `tar` utility:

```
> tar cvf submission.tar *.json
```

(the `tar` file can be named whatever you want, as AutoLab will rename it anyway, but the JSON files **must be named correctly** or the autograder will fail)

If the file is correctly archived, and the JSON files properly named and formatted, the submission and its score will show up on the leaderboard. If not, you should be able to click the job link to see what went wrong. You MUST submit all your JSON files each time; the leaderboard will blank out any submissions that are missing, leading to an incorrect ranking! Of course this is fine while you're developing each one at a time–you can submit them just to make sure they're correct. But for your final submission, make sure they're all included to get the maximum possible score.

<*smarmy*> While it's certainly possible there's a bug in the autograder, it's much more likely there's a bug in your code.</*smarmy*> Pay attention to any part of the job failure reports that reference lines in your submission file.

If you think you can do better than what shows up (particularly to beat out everyone else) make another submission! There's no penalty for additional submissions, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is programmed to close submissions *promptly* at 11:59pm on January 16, so give yourself plenty of time!

DIRE WARNING: If you do not adhere to this output format and the autograder crashes as a result, your entire team will be forthwith banished from Athenshire! Seriously though, please adhere to this output format.

5 Summary

In summary, these are the things I will look over when determining your team grade:

1. **Code**: Organization, structure, style, and clarity. Strategy and theory will also be a big component (what did you implement? was it implemented correctly?).
2. **Documentation**: This includes comments in the code, but also in the repository itself (GitHub wiki, `README` instructions). How easy would it be for someone to get up and running with your code, or to submit a bugfix?
3. **Accuracy**: Testing accuracy as submitted on AutoLab, and how far you are from the very top.
4. **Extras**: These include things like continuous integration tools, project websites, theoretical novelty (within the constraints of the project), unit tests with good coverage, an open-source license on the repository (with accompanying `LICENSE` file), among others. Use your imagination!