CS 722 Project 2

Kyle Goodwin and Sam Harris

1. Dataset

Our map reduce function is run via an HTTP Post method to its URL (local host or azure link) with the endpoint https://url/api/BookReduce_HttpStart with the body being a JSON format where books = {list of books and Project Gutenberg links}.

The books we used are as follows:

```
books = @(
    @{
        title = "Pride and Prejudice"
        url = "https://www.gutenberg.org/files/1342/1342-0.txt"
    },
    @{
        title = "The Picture of Dorian Gray"
        url = "https://www.gutenberg.org/files/174/174-0.txt"
    },
    @{
        title = "Moby Dick"
        url = "https://www.gutenberg.org/files/2701/2701-0.txt"
    },
    @{
        title = "Dracula"
        url = "https://www.gutenberg.org/files/345/345-0.txt"
    },
    @{
        title = "Sherlock Holmes"
        url = "https://www.gutenberg.org/files/1661/1661-0.txt"
    },
    @{
        title = "Tale of Two Cities"
        url = "https://www.gutenberg.org/files/98/98-0.txt"
    },
    @{
        title = "Frankenstein"
        url = "https://www.gutenberg.org/files/84/84-0.txt"
    },
    @{
```

```
        title = "Great Gatsby"
        url = "https://www.gutenberg.org/files/64317/64317-0.txt"
    },
    @{
        title = "Alice in Wonderland"
        url = "https://www.gutenberg.org/files/11/11-0.txt"
    },
    @{
        title = "Wuthering Heights"
        url = "https://www.gutenberg.org/files/768/768-0.txt"
})
```

2. Analysis of the Inverted Index

Some results were very predictable, for example the word “I” being seen hundreds of times per book or the word “because” being seen in most if not all buckets with varying frequency. One thing we did not account for nor expect were special characters like the underscore character “_” being seen before words or used consistently in general. This is an interesting result likely caused by the metadata tags or characters that were not parsed correctly in the Project Gutenberg .txt files, and were not removed by the tokenizer used to parse the text. We caught a lot of them and tried to parse some. What was tricky was trying to leave the possessive “’s” that many nouns have. Since our parsing removed punctuation and splits words by punctuation and other characters, the “s” was usually the most common word if it was left in.

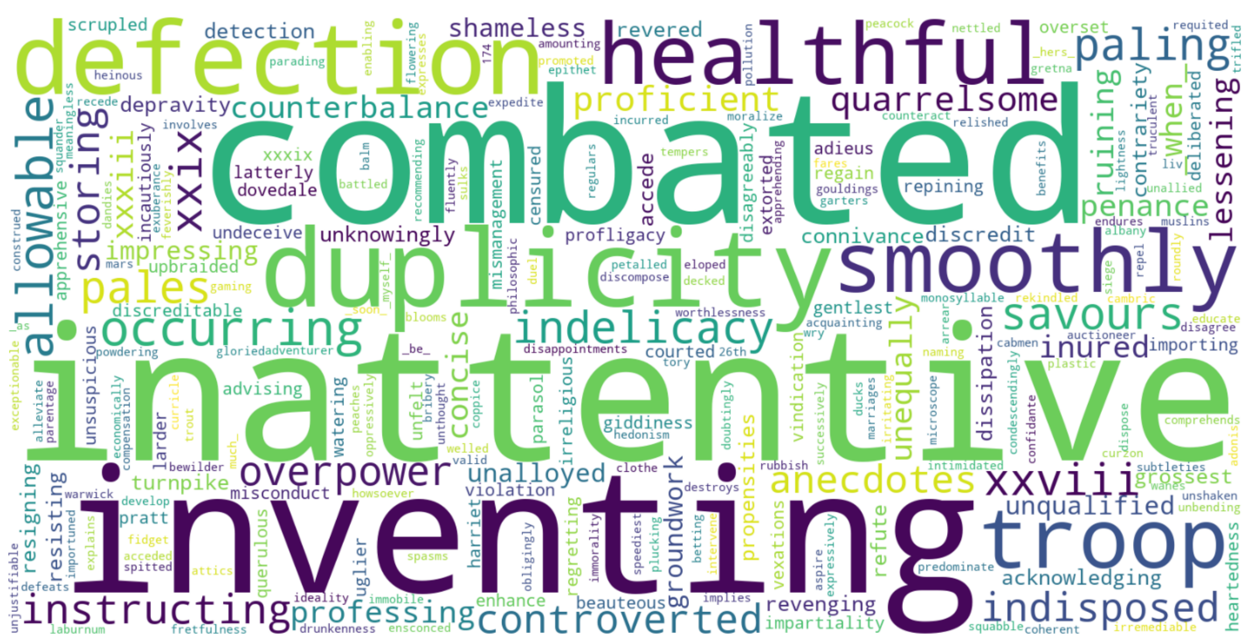
There were also many interesting, rare terms, many being numbers or dates (for example “144” in Moby Dick bucket 30, or “15th” in Pride and Prejudice bucket 5. The most interesting and least common words came from Moby Dick and Wuthering Heights which makes sense as both books are known for being long and difficult. It was also clear that Moby Dick is the longest book as it is so ubiquitous in the results which makes sense since is the longest of all the books by having tens, if not hundreds, of thousands more words than the other books.

To better view the words please enjoy some word clouds. Considering the vast number of unique words, we had Claude Sonnet 4.5 build a quick python script that takes the inverted index and creates a word cloud of the 250 most common words, the 250 median common words, and the 250 least common words from the JSON results file. There are some very interesting words in the least common cloud.

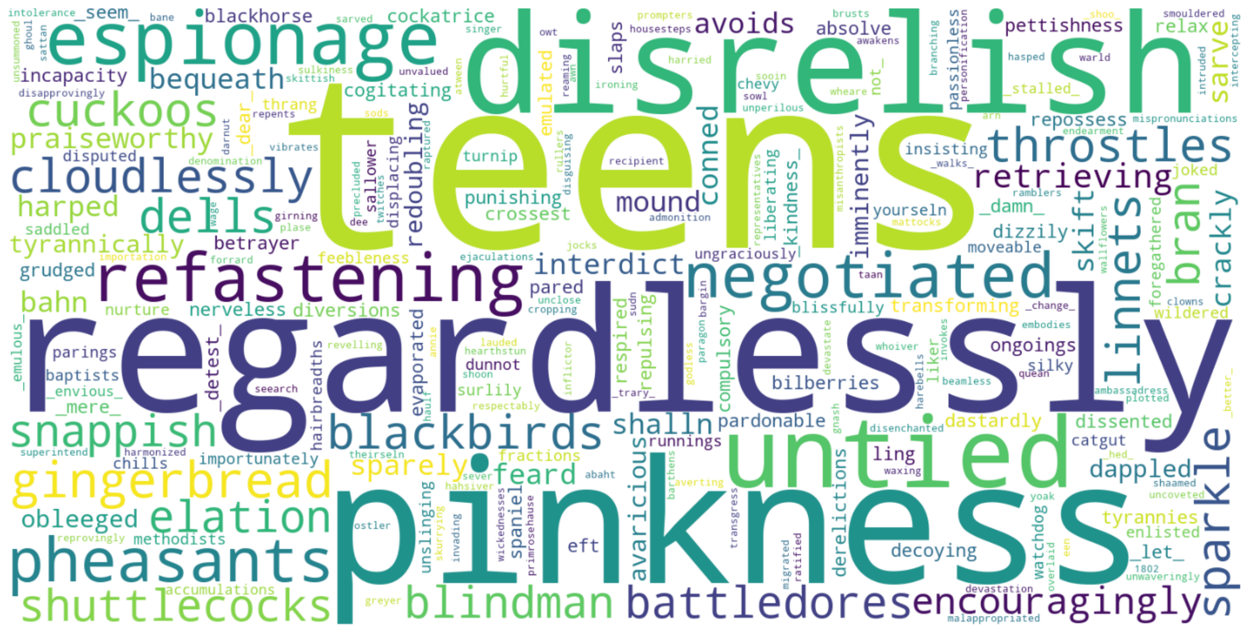
And interestingly a lot of them are odd old-timey verbs or adverbs. A lot of them are from Moby Dick and Wuthering Heights so that does make sense. Most of the common words are very common and an argument could be made to add many to the ignore list.

Top 250 Most Common Words

Middle 250 Most Common Words



250 Least Common Words



3. AI Use Disclosure

We were able to use both the existing map reduce example, the fan out/fan in tutorial, and the existing Azure Function App template from Visual Studio to build out most of the project. We did use Claude Sonnet 4.5 to build out the parsing that is done by the mappers in MapperAsync. Prompting that "I am trying to split a text document of a book into individual words, ignoring punctuation, new lines, and spaces". This worked well because we haven't done much with regular expressions and it provided a good pipeline from the document to a cleaned version of the content. We had to tweak a bit to get the results we wanted and try to account for some of the weird outputs we saw with "_" and the possessive s. But there were a lot of different edge cases, and we covered as much as we could. To test the parsing, we ran it a few times with single books and more books and then looked at the output to determine what was being included and what wasn't. Then we were able to adjust what was being included and what was being cut. We tried messing with it to keep the possessive "'s" by transforming "'s" into "possessive" and then back after the punctuation had been trimmed but for some reason, we couldn't get that to work.

The other thing we had that LLM build out was the GetResults functions. This is the HTTP function that provides users with a way to download the JSON results that are stored ephemerally in the Function App. So, when the reducer is finished we use the GetTempPath() function to save the JSON results somewhere on the VM that is hosting the Function App and then to get those results to my local machine we needed an endpoint to access that. We prompted Claude Sonnet 4.5 to "create an HTTP function that will serve the results.json file from the temporary storage using the GetTempPath() function". It created pretty much exactly what we wanted. We adjusted the endpoint route since the LLM had added an extra identifier so you could include the orchestration instance but that

wasn’t necessary. Then to test it, we ran the function with some books and called the GetResults endpoint which worked, and we were able to view the results in JSON format and save them to my machine.