

Samuel Reade

Professor Hosseinmardi

Communications 188C

May 1st, 2025

### Homework 1 Report

The first chunk of my code implements a Wikipedia article scraper that gets information from articles across four specific topics: History, Sports, Technology, and Politics. It starts by defining a function 'get\_title\_and\_first\_paragraph', which takes the URL of a Wikipedia page, sends an HTTP request, and parses the HTML using the library BeautifulSoup to extract the page title and the first paragraph of content. If the structure is valid and the required elements are found, it returns this information. The 'scrape\_category' function is responsible for collecting a specified number of pages, which was specified to be 50 for each topic. It identifies links to both pages and subcategories to ensure there are 50 pages for each topic. It then scrapes each page using the first function, and if not enough articles are found, it supplements the data by calling 'scrape\_category\_from\_subcategory', which performs a similar scraping process on each subcategory page. Each article is stored with its topic, title, and first paragraph. The final function, 'scrape\_wikipedia', iterates over a dictionary of predefined Wikipedia category URLs corresponding to the four chosen topics. It collects articles from each topic, aggregates all the results, and loads them into a Pandas DataFrame with columns: 'Topic', 'Page\_Title', and 'First\_Paragraph'. The code is designed with error handling and rate limiting using 'time.sleep(1)' to make sure there is a good interaction with the Wikipedia servers. After scraping a chunk of code turns the data frame into a json file named 'wikipedia\_data.json'.

The next step in the process involved preprocessing all the data scraped from Wikipedia. Tokenizing the text in the 'df' data frame was the first thing that needed to be done. The code for this defines a 'simple\_tokenizer' function to tokenize the text by converting it to lowercase and extracting words using regular expressions. It then applies this tokenizer to each article's title and first paragraph in the data frame, creating two new columns: 'Title\_Tokens' and 'Paragraph\_Tokens'. After all the text was tokenized, removing stop words became the next priority. The code for this process defines a set of common stopwords and uses a custom tokenizer named 'clean\_tokenizer' to clean and tokenize the text by removing those stopwords. It then applies this cleaning process to the page titles and paragraphs in the data frame, storing the results in new tokenized columns. Initially, using the 'nltk' library was the most desirable option. However, many difficulties arose involving dependencies with python environments and kernels when trying to access this library. To solve this problem, defining a list of known stopwords manually was the best option.

After processing the data, creating a Document-Term Matrix was a good way to gain insight into what words appeared in each document. The code for this constructs a DTM by counting the frequency of each word in the tokenized paragraphs. First building a vocabulary from all the tokens, and creating a matrix where each row represents an article and each column corresponds to a term's count. At the end it adds the original topic and title information to the matrix. The resulting data was initialized to be stored in the 'dtm\_df' data frame.

Creating a cosine similarity matrix was also a challenging step. This is because the same dependency issues arose while trying to use the 'sklearn' library. To get around this issue manually creating a cosine similarity calculator was the only option, which is defined as 'cosine\_similarity\_manual'. The next step was to access the data from the 'dtm\_df' in order to

calculate cosine similarities. The data was initialized to be stored in the 'tfidf\_values' object. Once these values were accessible the cosine similarity calculator function was applied to the 'tfidf\_values' data and stored in the 'similarity\_matrix' object as an array.

Creating actual similarity matrices was necessary to explore the relationships between the unique pages, and their topics. To do this code iterates through a list of thresholds, 0.2, 0.4, 0.6, 0.8, and creates a binary matrix for each threshold by converting the cosine similarity values in 'similarity\_matrix' to 1 if they are greater than or equal to the threshold, and 0 otherwise. Due to the binary nature of the data being processed all the interactions in the plot were either dark blue, if they were over the similarity threshold, or white, if they were under the threshold. Each plot shows the document index on both axes. This visualization helps observe how document similarity changes based on the selected threshold. The 'Plotly' library was used to make the four plots because the matplotlib library ran into the same dependency issues as 'nltk' and 'sklearn'.

The next step was to create confusion matrices. To do this the 'Topic' column in 'df' had to be converted into a list. Then, the next chunk of code is using the similarity matrix to predict labels for each document based on the label of the second most similar document in the dataset. This assumes that documents with similar content share the same topic. With the previous information a confusion matrix to compare the true labels, 'true\_labels' with the predicted labels, 'predicted\_labels', is generated. It counts how often each predicted category matches the true category and stores the result in a DataFrame 'cm\_df'. To investigate the results further, generating and visualizing three different versions of a confusion matrix using Plotly was ideal. The code first creates a standard confusion matrix, 'fig\_cm', to show the count of correct and incorrect predictions for each category. Then, it creates a row-normalized confusion matrix 'fig\_row\_norm', where each row is scaled to represent the proportion of predictions for that true

category. A column-normalized confusion matrix, 'fig\_col\_norm', is then created, where each column is scaled to show the proportion of true labels for that predicted category. The confusion matrices are displayed as heatmaps, where the color intensity represents the values, making it easier to analyze model performance across different categories. It is clear that the diagonals have the highest values. However, sports and history seem to perform well together too for the all three confusion matrices. Politics and technology also seem to have relatively good performance.

After creating confusion matrices, investigating the relationships between the Wikipedia pages by clustering them by their similarity scores was a good analytical step. The code for this defines two functions and uses them to create clusters based on document similarity and then visualize the most frequent words within each cluster with a bar chart. The 'create\_clusters' function takes a similarity matrix and a threshold as input, iterating over all documents to form clusters of similar documents based on the threshold. It checks each document and groups it with other documents that have a similarity score above the threshold. After trial and error the code was outputting too many clusters with high thresholds. This means many of the documents were not very similar. To handle this, changing the threshold to 0.05 was necessary. Even with this low threshold 42 clusters were returned. The 'generate\_plotly\_word\_bar' function visualizes the most frequent words in each cluster by counting the occurrences of tokens from the documents in the cluster and creating a bar chart using Plotly. The code then applies the functions. First, it generates clusters using the similarity matrix, and then for each cluster, it creates a bar chart of the top 20 words in that cluster, displaying the charts for each cluster sequentially.

To further investigate the tokens that contributed to the clusters creating word clouds for each cluster was the next step in the process. The code defines a function to generate word cloud

visualizations for document clusters using Plotly. The `'generate_wordcloud_plotly'` function takes the indices of documents in a cluster, extracts all word tokens from those documents, and counts their frequencies. It selects the 100 most common words and assigns each a random position on a 2D plane. The font size of each word is scaled proportionally to its frequency. A Plotly scatter plot is then created where each word is displayed as a text element at its assigned position, mimicking the appearance of a traditional word cloud but using Plotly instead of matplotlib because of the dependency issues. A loop then runs through each cluster and calls this function to visualize the word cloud for that cluster.