John Kubas | Logan T Graham

2842254 | 2864014

CIS 492 Big Data

Final Project Report

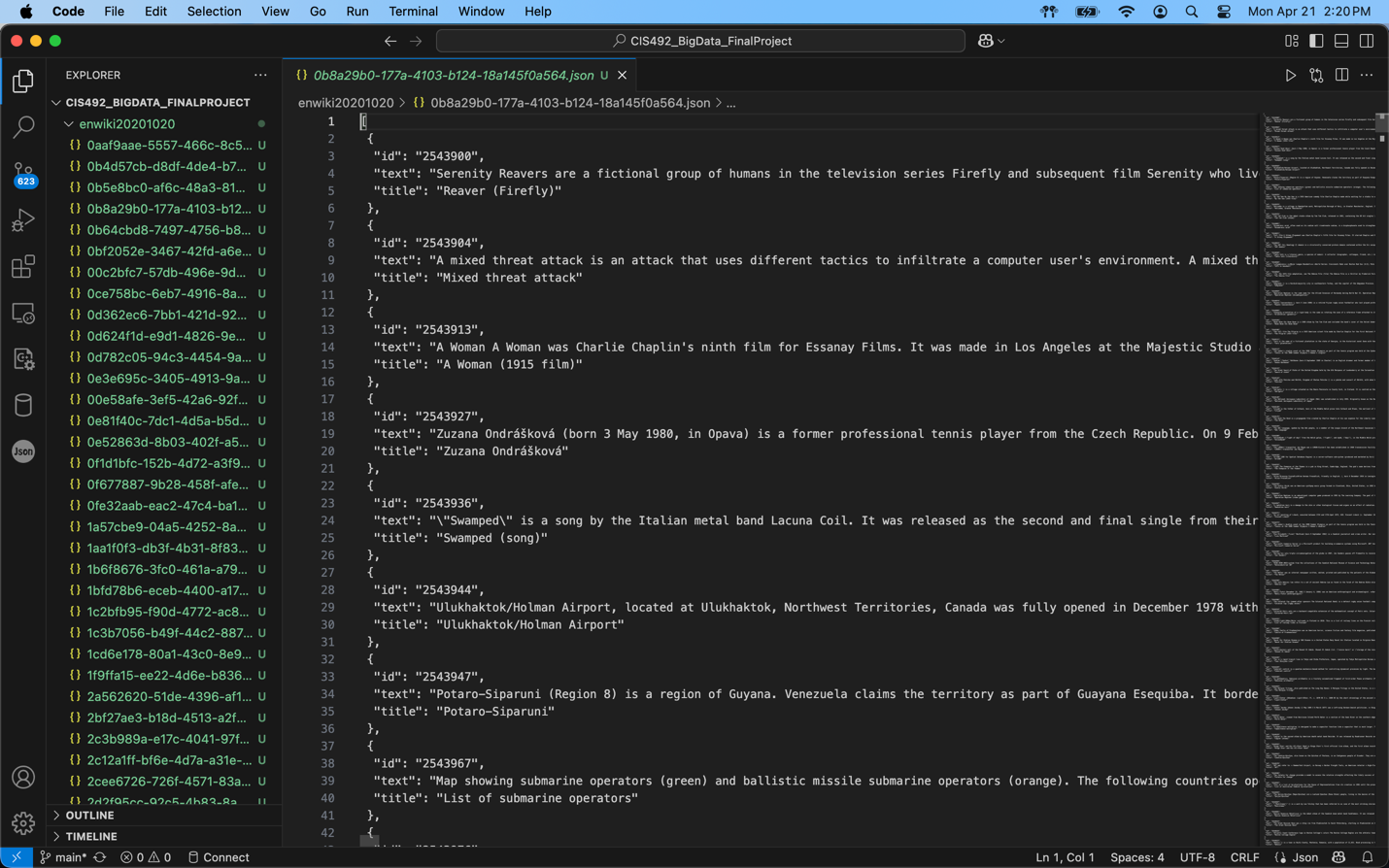
Intelligent Wikipedia Search Engine with Inverted Index and TF-IDF Ranking Using Cosine Similarity

## Data Collection, and Pre-Processing:

## For this project, we are using a Wikipedia data dump publicly available at <https://dumps.wikimedia.org/enwiki/20250401/#:~:text=enwiki%2D20250401%2Dpages%2Darticles.xml.bz2> . This data set contains a semi-structured XML file containing ~ 25,000,000 articles. The XML file is 22.1GB compressed and 100.8 GBs uncompressed. For testing and validation we will use a smaller partition of the full data dump found at <https://dumps.wikimedia.org/enwiki/20250401/#:~:text=enwiki%2D20250401%2Dpages%2Darticles1.xml%2Dp1p41242.bz2> which is approximately 1GB in size compressed and 5GB uncompressed, containing ~ 2,500,000 articles.

Each article contains an article title, an article ID, the body text, category tags, and re-direct links to other articles and references. Our pre-processing phase will consist of parsing the XML file and extracting documents as an object in JSON format. Each JSON object will contain the fields “title”, “BodyText” and “ArticleID”. Significantly we will be skipping over metadata, re-direct links, and category tags (we will develop our own later). To do this, we used an existing python library “wiki text” available at <https://pypi.org/project/wikitextparser/> or with “pip install wikitextparser” (This was approved by Dr. Sunnie in a conversation after class on Wednesday April 16th). The functions we used from this library were wikitext.parse(), wikitext.plaintext(), and htt(string). Wikitext.parse() turns the entire <page> tag into a tree representation and wikitext.plaintext() turns the tree back into plain text. These two functions are called in sequential order on the body text of a given page, resulting in a string which represents the body text of a given article. Htt(string) removes any remaining html elements from the body text string.

To gather all articles, we iterate through the entire data dump and when we encounter <page> we then grab the <title>, <id> and <text> tags. The <text> tag is the one we pass through the parser, then we use all 3 tags to create a JSON object for the article. Now that the articles are stored in JSON format we can process them in our AI application.An example of the formatted JSON file:



## Building Our Big Data/AI Application

Now that the data is processed, the first step in building our application is to read each json object, extract the ‘text’ field, and pass it through our NLP pipeline. The pipeline consists of the following steps:

1. Lowercase to avoid case mismatches: text\_lower = text.lower()
2. Punctuation removal: re.sub(r'[^a-z\s]', '', text\_lower)
3. Tokenization: tokens = nltk.word\_tokenize(text\_cleaned)
4. POS tagging: pos\_tags = nltk.pos\_tag(tokens)
5. NER chunking: ner\_tree = nltk.ne\_chunk(pos\_tags)
6. Stop Word Removal: from nltk.corpus import stop\_words ….. if word not in stop\_words
7. Lemmatization: from nltk.stem import WordNetLemmatizer….lemmatizer = WordNetLemmatizer()….lemma = lemmatizer.lemmatize(word)

\*\*write about important design choice, early iterations had very complete collections but had noisy/ meaningless terms such as ‘0’ and ‘zz999zz’\*\*

\*\*will removing nlp on a user query result in faster queries without lowering accuracy?\*\*