Course Title: Programming for Data Applications
Course Code: LDSCI7234
Course Leader: Dr. Jiri Motejlek
Student ID: 23220031
NB: Please also download text file, alongside the submission, thank you:)
Test data: 11 literature from Fyodor Dostoevsky, 14 from Charles Dickens, 11 from VirginiaWoolf, 11 from Leo Tolstoy.
Link to git repository: https://github.com/JENWH/AE2-Programming-for-Data-Applications.git

Task Documentation and Report:

Analysis:
This project's goal is to develop a basic interface using the Flask Library, implementing a trained word vector embeddings model using the Word2Vec technique and the Gensim library. The project unfolded in four distinct phases: initially selecting training data; then developing a function to cleanse this data in preparation for model training; subsequently creating and initiating the training process of the model; and ultimately designing a web interface with backend capabilities that retrieves antonyms of user-input words using the trained embeddings.

We used 25 literary works by Fyodor Dostoevsky and Charles Dickens as training data, chosen for their thematic consistency and coherence. These texts, sourced from Project Gutenberg in TXT format, formed the foundation of our training material. However, the model's effectiveness is currently sub-optimal, largely due to the significant increase in its size with added data, complicating its upload to GitHub. Despite these challenges, the project successfully demonstrates the logical process of training a word vector embeddings model and its deployment via a web interface. To enhance the model, potential improvements include expanding the training dataset, experimenting with different parameter settings, and exploring alternatives to both Word2Vec and the Gensim library.

A key objective was deploying these trained embeddings on a web interface for interactive user engagement and antonym retrieval. An intriguing insight emerged during the project: up to the 14th training session, the model accurately identified words similar to 'sad', but it incorrectly labeled 'pleasant' as its antonym. This highlights a crucial aspect of the English language—its words often possess multidimensional meanings. Therefore, when vectorizing words to identify opposites, a linear approach may not be entirely appropriate. This is exemplified by words like 'love' and 'hate', which, while seemingly antithetical, could both find their true opposite in a concept like 'indifference', denoting an absence of emotion.

In summary, the project, while adhering to a straightforward methodology to achieve its aims, the accuracy of the model is subpar, revealed several nuanced factors that could refine the model's design and training process.

Design
During the first phase of data collection, I chose to focus on authors such as Fyodor Dostoevsky and Charles Dickens to ensure a consistent quality in the data. Literature, particularly of this caliber, often employs sophisticated wordplay and can convey multiple meanings, typically with a melancholic undertone. Selecting multiple works from these

renowned authors was strategic, aiming to maintain a level of consistency critical for fine-tuning parameters like vector_size, min_count, and sg.

Given the rich context and narrative depth in the works of Dostoevsky and Dickens, I opted for a larger vector_size of 230 to adequately capture these subtleties. Their classical literature boasts an extensive vocabulary, hence my decision to set the min_count to a lower threshold of 3, ensuring that even less frequent words are included. The complexity of this literature also made the Skip-gram model (set to 1) a suitable choice for capturing semantic relationships.

In the training phase of the model, I meticulously analyzed the results using Dostoevsky's works, specifically focusing on the top 5 words akin to 'sad'. This was instrumental in evaluating the effectiveness of my parameter adjustments.

Once satisfied with the parameter settings, I expanded the dataset by incorporating all of Charles Dickens' literature, thus enriching the training material.

Regarding data cleaning, my initial step was to scrutinize the format of texts obtained from Project Gutenberg. These often begin with a standard Gutenberg header and include irrelevant numerical data and special symbols like '---', '***', and '…'. My cleaning process, therefore, went beyond the usual steps of case normalization, contraction expansion, tokenization, punctuation removal, and stopwords elimination. It also included the removal of irrelevant numerical data and special symbols.

Before consolidating these cleaning steps into a singular function, I meticulously outlined each step's logic, applying them to a single TXT file. This process of creating functions with direct references to these steps ensured that any overlooked errors were captured.

Having developed the clean_and_convert_text function, I applied it to my dataset to ready it for training.

The training phase itself was relatively straightforward once the parameters were set. I monitored the effectiveness of the training by reviewing the first 14 sessions' results. Subsequently, the model was saved as "./model_test" for deployment in the web application.

The web interface design encompassed two key aspects. Firstly, I developed the web page using HTML, focusing on a simple user request to find the opposite of a given word. Secondly, I worked on the model's inference and deployment within the application. I selected the word pair 'wealthy' and 'poverty' for their appropriateness to the literary style under study.

In the Word2Vec Model, words are represented as vectors in a multidimensional space, where closer vectors signify similar meanings. To identify antonyms, I employed a formula: word1 - word2 + word3 = word4, with word4 being the target prediction. This approach aimed to find a word (word4) that bears a similar relationship to word3 as word1 does to word2.

Using the command opposite_result = model.wv.most_similar(positive=[user_input, reference_pair[1]], negative=[reference_pair[0]], topn=1), I directed the model to discover a word linked to the user's input, analogous in context to 'wealthy' and 'poverty'. Essentially, the

model executes vector arithmetic by subtracting the vector of 'wealthy' from the user's word and adding the vector of 'poverty', thereafter returning the outcome of this computation. Another interesting observation is that when I added four literary works by Ernest Hemingway, the model's accuracy decreased when I attempted to input words. This will be explained in the 'lessons learned' section.

Implementation:
The training phase was executed with a focus on simplicity. Once the parameters were set, I monitored the training by reviewing the outcomes of the first 14 sessions. This led to the model being saved as "./model_test," which I then integrated into the web application.

The web interface's design is bifurcated into two key components. Firstly, the interface, crafted using HTML, straightforwardly prompts users to input a word for which they seek an antonym. Secondly, the backend involves scripting the inference logic and embedding the model within the application. For this, I selected the word pair 'wealthy' and 'poverty', as these terms align well with the literary style of the source material.

Within the framework of the Word2Vec Model, words are depicted as vectors in a multidimensional space. This spatial arrangement is designed so that words bearing similar meanings are positioned in closer proximity. To ascertain opposites, the model employs a formula: word1 - word2 + word3 = word4. Here, word4 is the target word that the model aims to predict, representing a word that bears a similar relation to word3 as word1 does to word2.

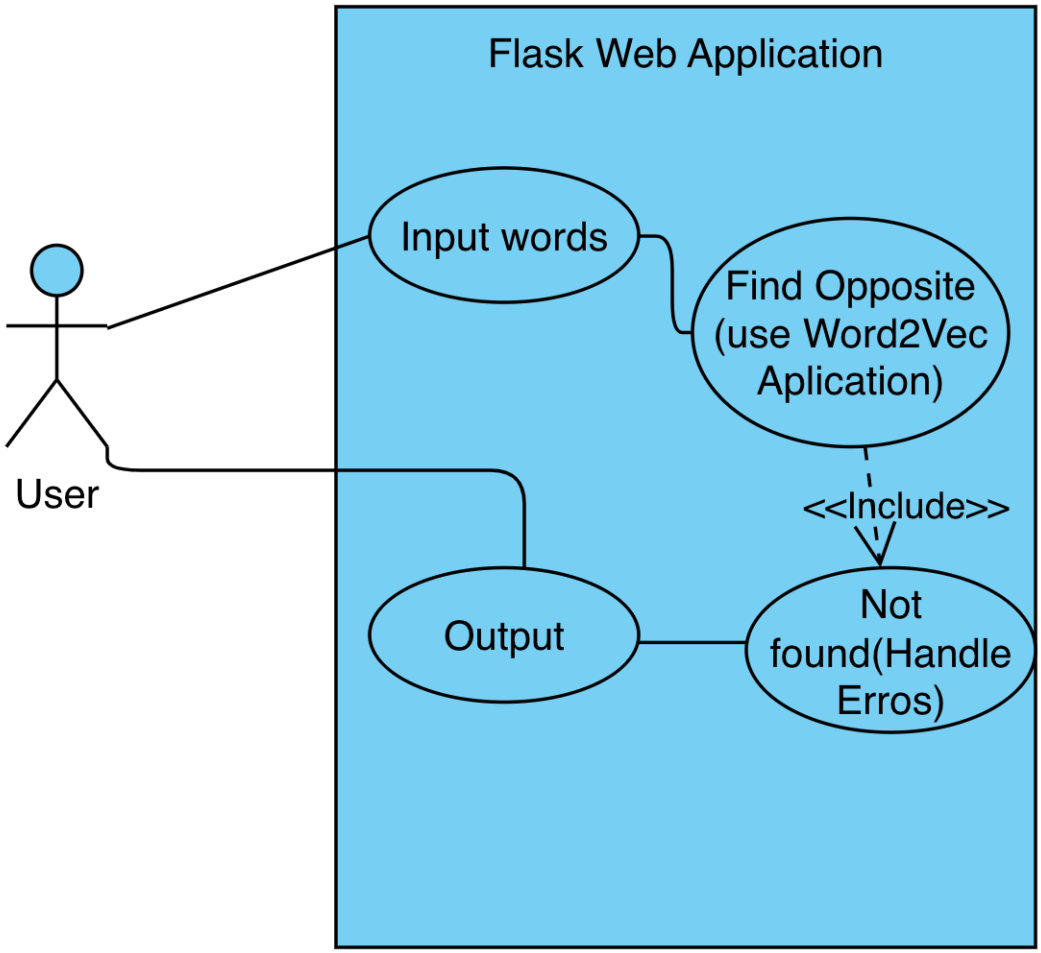
The process involves executing the command opposite_result = model.wv.most_similar(positive=[user_input, reference_pair[1]], negative=[reference_pair[0]], topn=1). Essentially, this instructs the model to identify a word that maintains the same relational dynamics with 'wealthy' and 'poverty' as the user's input word. This is achieved through vector arithmetic, where the model subtracts the vector for 'wealthy' from the user's input word and adds the vector for 'poverty', subsequently outputting the result of this operation.

Deployment:
With the model deployed and the web interface established, users can access the application through the provided link. Here, they are prompted to enter a word to discover its antonym. If the input word is not part of the model's vocabulary, the interface will display a message: "Word not in database." However, as I have observed, the model is not entirely accurate, and the returned words are not consistently the exact opposites, sometimes, the model even returns words that are similar, rather than antonyms. This issue will be addressed in the section below.

Lesson Learned:
Throughout this project, I gained several valuable insights. The first lesson pertains to data collection: the importance of maintaining consistency in data selection and employing a clear naming system for each file. This becomes particularly vital when handling extensive datasets, as efficient file location is key. Especially when there is too much data and it is difficult to clean the data properly, it becomes an important issue. For instance, after adding Leo Tolstoy's literature, if I type 'sad,' the result is 'Mihalovna,' which isn't an English word but rather signifies 'daughter of Mikhail.' This term appears in Tolstoy's novel, 'Resurrection,'

and it underscores the importance of paying attention to certain nuances and challenges in English literature during the text cleaning process. Secondly, a crucial preparatory step before initiating data cleaning is to thoroughly examine the file. A clear understanding of where and how to begin is essential to avoid starting blindly and making unnoticed errors. Thirdly, an intimate knowledge of the chosen data is fundamental in effectively tuning the model. Observing the initial training results provided crucial feedback, indicating whether the parameter adjustments were effective. Fourthly, in the realm of web interface design, the selection of reference pairs should be thoughtfully aligned with the available database. In this instance, the 'wealthy' and 'poverty' pair were aptly chosen, resonating with the literary style and context. Looking forward, I am considering the possibility of selecting reference pairs that more accurately reflect the literary context. Lastly, one interesting thing is that the accuracy of the model does not necessarily increase when I add more data for training. This may be due to issues such as overfitting, data quality and balance issues, and may also require adjustments to the parameters. All points mentioned can be considered to adjust and improve the model.

Task Documentation:
Basic UML class diagram:



This simple UML class diagram explains the basic logic of the web interface “Find the Opposite Word!”. As illustrated, within the Flask web application, a user can input a word. The Word2Vec application model will find the opposite using the pre-trained model. If the

word is not included in the library database, it will process an error. Then, the processed output is returned to the user.