Experiment 2

1. Enumeration of Text corpora Utilised

The following links and points explain the various text files and csv files used as a part of our second experiment.

1. **"static.txt":** This corpus includes around 26000 static idioms from the EPIE corpus

   Text Corpus Link here

2. **"bbc-news-data.csv":** This corpus includes 2226 text files from bbc news sources. The bbc.ipynb notebook contains the code used to preprocesses a dataset containing news articles from the BBC, saving it in two formats: a pickled Python list (bbc.pkl) and a plain text file (bbc.txt). It first reads a CSV file, extracts and concatenates the text data, splits it into sentences, filters out short and numeric sentences, and finally saves the processed data into the specified formats.

3. **"generic.txt:"** This corpus includes around 1 million generic sentences sorted alphabetically.

   The generic.ipynb is used to download a dataset from a TSV file and load it into a pandas DataFrame using pandas. We extracted the fourth column of the DataFrame, assuming it's indexed as 3 (0-based index). This column contains a collection of sentences. We converted this column into a list of strings and found that it contains a total of 1,020,868 sentences.

   For example, one of the sentences in the list is: "Koalas are nocturnal marsupials famous for spending most of their lives asleep in trees."

   Next, we decided to save a subset of these sentences into a text file named "generic.txt." We opened the file in write mode and used the writelines method to write every third sentence from the list into the text file. This process allowed us to create a new dataset containing a subset of sentences for further use.

   Generic Corpus Link here

4. **"1_combined.txt":** This corpus contains 54000 random sentences with idioms and their translations too from the EPIE corpus as well as the preprocessed text from the bbc.txt corpus. This was manually created from individual files.

   EPIE Corpus Link here

2. First attempt at K-Means clustering and subsequent visualization

Building experience with clustering and higher dimensional visualization of vectors.

1. Creating Data Points: We start by defining a set of 3-dimensional data points (`vec_a` to `vec_h`) that we want to cluster in 3D space. These data points represent our dataset.

2. K-Means Clustering: We use the `KMeans` clustering algorithm from scikit-learn to cluster our data points into three clusters. The `n_clusters` parameter specifies the number of clusters we want to create, and `n_init` controls the number of times the algorithm will be run with different centroid seeds.

3. Setting Environment Variable (Windows Only): Due to a known memory leak issue on Windows with the MKL library, we set the `OMP_NUM_THREADS` environment variable to '1' to mitigate the issue.

4. Fitting the K-Means Model: We fit the K-Means model to our data points, and the algorithm identifies cluster centers.

5. Plotting in 3D Space: We create a 3D scatter plot using Matplotlib to visualize our data points. The blue points represent our original data points, and the red 'x' markers represent the cluster centers. We set labels and a title for the plot.

6. Displaying the Plot: Finally, we display the 3D plot, showing the clusters and their centers.

This code demonstrates how to perform K-Means clustering on a set of data points in 3D space and visualize the results using Matplotlib.

3. Utilizing Sentence Embeddings for Text Augmentation

We explain the code in the embeddings_script.py file as well as the equivalent jupyter notebook file, embeddings.ipynb.

1. Library Imports: The code begins by importing essential Python libraries. It includes:
   - `numpy` for numerical operations.
   - `random` for generating random values.
   - `time` for measuring execution time.
   - `pickle` for working with serialized data.
   - `SentenceTransformer` for utilizing a pre-trained sentence embedding model.
   - `nltk` for natural language processing tools.
   - `spacy` for loading a pre-trained English language model.
   - `sklearn` for implementing the k-nearest neighbors algorithm.
2. Loading Pre-trained Models: The code initializes a Sentence Transformer model (`sbert_model`) with the 'bert-base-nli-mean-tokens' pre-trained model. It also loads a pre-trained English language model using SpaCy (`nlp`).
3. Sentence Embedding Functions: Two functions for sentence embedding are defined:
   - `embed_1(s)` encodes a single sentence using the Sentence Transformer model.
   - `embed(s)` encodes a sentence by splitting it into trigrams and averaging their embeddings.
4. Synonym Retrieval: Functions like `find_closest_synonyms`, `remove_underscores`, and `find_most_sim` are defined to find synonyms for words in a sentence.
5. Text Augmentation: Functions `select_random`, `replace_w_aug`, `select_random_2`, and `replace_w_aug_2` are provided for word replacement in a sentence with synonyms.
6. k-nearest Neighbors (k-NN): The `knn` function uses k-NN to find the nearest sentences in the embedding space.
7. Similarity Score Calculation: The `s_score` function calculates the number of common elements between sets of indices returned by k-NN for different augmented sentences.
8. Sentence Index Retrieval: The `indices` function finds the indices of nearest sentences in the embedding space for a given input sentence.
9. File Handling: Functions `read_text`, `save_arr`, and `main` handle reading data from files and executing the main logic of the code.
10. Main Execution: The `main` function loads pre-trained sentence embeddings, performs augmentation on a sample sentence, and calculates similarity scores.

The code utilizes various natural language processing techniques, including sentence embeddings and synonym replacement, to enhance and augment text data. The goal of the code is to find the effect in similarity score of embeddings when single adjective/nouns/verbs are replaced with close synoynms. The hypothesis is that when we replace words for idioms/informal sentences the meaning will shift a lot therefore drastically reducing the similarity score.

4. Experimenting with Universal Sentence Encoder (USE)

1. Library Imports: The code starts by importing necessary Python libraries:
   - `tensorflow` and `tensorflow_hub` for using the Universal Sentence Encoder (USE) model.
   - `pickle` for handling data serialization.
   - `sklearn` for implementing the k-nearest neighbors algorithm.
   - `numpy` for numerical operations.
2. Loading Universal Sentence Encoder (USE): The Universal Sentence Encoder model is loaded from TensorFlow Hub using the provided module URL.
3. File Handling Functions: Functions `save_arr` and `read_text` are defined for saving data as pickle files and reading text data from a file, respectively.
4. k-nearest Neighbors (k-NN): The `knn` function uses k-NN to find the nearest sentences in the embedding space using the Universal Sentence Encoder.
5. Similarity Score Calculation: The `s_score` function calculates the number of common elements between sets of indices returned by k-NN for different sentences.
6. Sentence Index Retrieval: The `indices` function finds the indices of nearest sentences in the embedding space for a given input sentence using the Universal Sentence Encoder.
7. Data Preparation: The code reads a large text corpus from a file using the `read_text` function and loads it into the `sentences` variable. It then uses the Universal Sentence Encoder (`use_model`) to obtain sentence embeddings for all sentences in the corpus.
8. Sample Sentence and Index Calculation: A sample sentence (`s`) is chosen, and the `indices` function is used to find the nearest sentences to this sample sentence in the embedding space.
9. Similarity Score Calculation: The `s_score` function is used to calculate the similarity scores between the sample sentence and two other variations of it.
10. Output: The code prints the sentences that are most similar to the sample sentence, along with the number of common elements in their k-NN results.

The code utilizes the Universal Sentence Encoder to generate sentence embeddings and measure similarity between sentences using k-nearest neighbors. It provides insights into the semantic similarity of sentences in the provided text corpus. As with the previous experiment we just are trying different embeddings to find the ones that most suit our hypothesis.

5.Attempting tuning using Annoy Index

This part of the code sets up and utilizes the Annoy index for efficient nearest neighbor search in the high-dimensional embedding space, providing a more computationally efficient way to find similar embeddings.

1. Initializing Annoy Index: The code begins by determining the dimensionality of the embeddings (`embedding_dim`) based on the shape of the embeddings array.
2. Creating Annoy Index: An Annoy index is created with the same dimensionality as the embeddings, and the metric used for distance calculation is set to 'angular'.
3. =Adding Items to Annoy Index: The code iterates through each embedding in the `embeddings` array, and for each embedding, it adds an item to the Annoy index with its corresponding index (`i`) and the embedding itself.
4. Building the Annoy Index: After all items are added, the Annoy index is built using the `build` method with a specified number of trees (`n_trees=10`).
5. Finding Nearest Neighbors: The code then performs nearest neighbor searches using the Annoy index for three different input sentences.
6. Retrieving Nearest Neighbors: The variable `ind` stores the indices of nearest neighbors for the first sentence, `i1` for the second, and `i2` for the third.
7. Extracting Sentences: The code extracts the actual sentences corresponding to the indices found by the Annoy index.
8. Calculating Similarity Scores: The `s_score` function is used to calculate similarity scores between the sets of indices.
9. Printing Results: The code prints the sentences that are most similar to the input sentences based on the calculated similarity scores.

We notice slightly better results as Annoy index is needed for high-dimensional embeddings comparison because it efficiently approximates nearest neighbors in high-dimensional spaces, reducing computational complexity and memory usage, which is crucial when working with large-scale text data and high-dimensional embeddings like those from deep learning models, making the search for similar embeddings much faster and practical.