Deep Learning Part 2

Section 1

The goal of this project is to generate blog content from a provided summary. You will fine-tune a language model to take a summary as input and generate a coherent blog post as output.

The primary objective of this project is to develop a language model capable of transforming a concise summary into a comprehensive blog post. This involves teaching the model to extract meaningful sentences from the passage.

## **Data Collection & Preprocessing**

### **Step 1: Dataset Creation**

**Understanding the Dataset:**

The Kaggle Medium Articles dataset is a valuable resource for text summarization projects. It contains a large collection of Medium blog posts, each with:

* **Title:** The headline of the article.
* **Text:** The full content of the blog post.
* **Summary:** A provided summary of the article.



**Suitability for Text Summarization:**

This dataset is well-suited for your project because:

* **Large scale:** It offers a substantial number of samples, allowing for robust model training.
* **Diverse content:** The articles cover a wide range of topics, ensuring the model is exposed to various writing styles and subject matter.
* **Annotated summaries:** The dataset includes pre-written summaries, providing a valuable ground truth for training and evaluation.

**Data Preprocessing:**

Before using the dataset, you'll need to preprocess the text data to ensure it's suitable for model training. This typically involves:

* **Cleaning:** Removing special characters, punctuation, and stop words.
* **Tokenization:** Breaking down the text into sentences and words.
* **Normalization:** Converting text to lowercase and handling inconsistencies.
* **Handling missing values:** Addressing any missing data points.

**Additional Considerations:**

* **Data quality:** Evaluate the quality of the provided summaries to ensure they accurately represent the content of the articles.
* **Data imbalance:** If the dataset is imbalanced (e.g., some topics have more articles than others), consider techniques like oversampling or undersampling to balance the classes.
* **Data augmentation:** Explore methods like backtranslation or synonym replacement to increase the diversity of the training data.

By leveraging the Kaggle Medium Articles dataset and following these preprocessing steps, you can create a solid foundation for training your text summarization model.

**Global Variable Setup**

* count = 0: Keeps track of how many blogs have been summarized. The generateSummary function increments this each time it's called, indicating the progress during the summarization process.

**Sentence Tokenization**

* **Purpose**: Break down the blog text into individual sentences.
* **Code**: sentences = sent\_tokenize(blog)
* **Explanation**: This step uses NLTK's sent\_tokenize() function to split the blog into a list of sentences. Sentence tokenization allows the algorithm to focus on smaller, meaningful parts of the text for summarization.

**Text Preprocessing**

* **Lowercasing & Punctuation Removal**:
  + **Code**: sentences\_clean = [re.sub(r'[^a-zA-Z\s]', '', sentence.lower()) for sentence in sentences]
  + **Explanation**: This regular expression removes non-alphabetical characters and converts all sentences to lowercase. This reduces variability and simplifies processing, ensuring uniform text.

**Stopword Removal**

* **Purpose**: Eliminate common words (e.g., "the", "is") that don't add meaningful information.

**Explanation**: NLTK’s list of English stopwords is applied, and only meaningful words are retained in the tokenized form of each sentence. Each sentence is split into tokens (individual words), and stopwords are filtered out.

**Word Embeddings (Word2Vec)**

* **Purpose**: Generate numerical representations (embeddings) of words.
* **Code**: w2v = Word2Vec(sentence\_tokens, vector\_size=100, min\_count=1, epochs=1000)
* **Explanation**: A Word2Vec model is trained on the cleaned tokenized sentences. This model converts words into 100-dimensional vectors based on the context in which they appear. The idea is that similar words have similar embeddings.

**Sentence Embeddings**

* **Purpose**: Represent each sentence as a vector by averaging the embeddings of the words in that sentence.
* **Explanation**: This step calculates embeddings for each sentence by taking the vector representation of each word. The embeddings are then padded to ensure all sentences have the same vector length for similarity comparison.

**Similarity Calculation (Cosine Similarity)**

* **Purpose**: Compute the similarity between sentences based on their embeddings.
* **Explanation**: The code calculates cosine similarity between the embedding vectors of every pair of sentences. The result is a similarity matrix where each cell represents how similar two sentences are. A value close to 1 means the sentences are very similar.

**Graph Representation**

* **Purpose**: Build a graph where sentences are nodes and cosine similarity scores are edges.
* **Code**: nx\_graph = nx.from\_numpy\_array(similarity\_matrix)
* **Explanation**: A graph is constructed using NetworkX, where each node is a sentence and edges represent the cosine similarity between pairs of sentences. This graph structure is key to applying the PageRank algorithm.

**PageRank Algorithm**

* **Purpose**: Rank sentences based on their importance in the text.
* **Code**: scores = nx.pagerank(nx\_graph, max\_iter=600)
* **Explanation**: PageRank, traditionally used for ranking web pages, is applied here to rank sentences. Sentences that are more similar to important sentences (high PageRank score) will be ranked higher. The number of iterations (max\_iter=600) ensures the algorithm converges.

**Top Sentence Selection**

* **Purpose**: Select the most important sentences to include in the summary.
* **Explanation**: Sentences are sorted by their PageRank score, and the top 25% of sentences are selected for the summary.

**Summary Generation**

* **Purpose**: Construct the final summary from the selected top-ranked sentences.
* **Explanation**: The top-ranked sentences are concatenated to form the final extractive summary, maintaining the original sentence order.

**Error Handling**

* **Code**: except: return float("NaN")
* **Explanation**: If an error occurs during the summarization process (e.g., insufficient data or tokenization failure), the function will return NaN, ensuring robustness.





