Text Mining and Natural Language Processing

2024-2025

Pratelli Pietro 535548

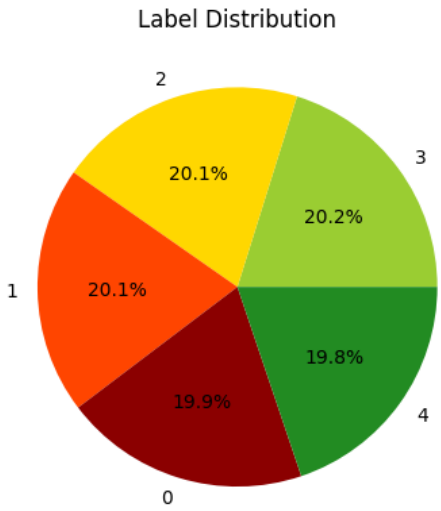
Ruggeri Sara 535570

SHREK

Sentiment Harvesting for Review Evaluation via Knowledge

Introduction

We chose a dataset including Yelp reviews and the corresponding rating from 1 to 5 given by a user. Our goal was to predict how many stars a review would be just from the text itself, so the task was one of ordinal encoding.

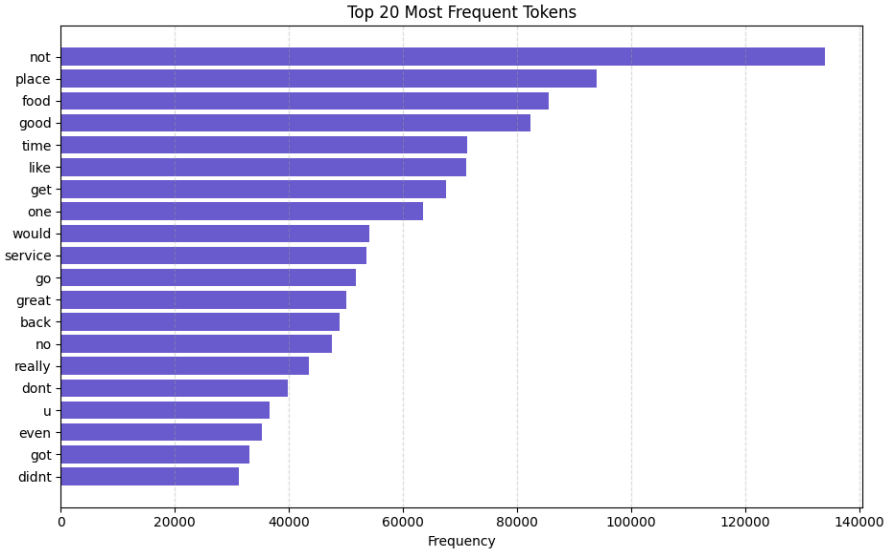


Data

The dataset we chose is constructed by Xiang Zhang (xiang.zhang@nyu.edu) from the Yelp Dataset Challenge 2015. It was first used as a text classification benchmark in the paper *Character-level Convolutional Networks for Text Classification. Advances in Neural Information Processing Systems 28 (NIPS 2015)* by Xiang Zhang, Junbo Zhao, Yann LeCun.

The original dataset was already divided into test and training sets, with 650,000 rows for training and 50,000 for test. As our computational resources were very limited, we decided to only keep a smaller portion of the two: in particular, we maintained 100,000 rows for training and 10,000 for the test.

In the dataset, a typical data-point comprises a yelp review and its corresponding score in stars (from 0 to 4 instead of 1 to 5, for simplicity). After reducing the dataset we also



checked the review score label distribution to make sure it was balanced. The result can actually be seen in the pie chart above and shows how they are, in fact, balanced.

After reducing the original dataset, we proceeded with the new one we had created. The distribution of words was analysed after some basic preprocessing steps, and we can observe from the word cloud and the histogram that some words stand out a lot more than others in our corpus.

Methodology

Given that the input consists entirely of textual data, one of the most important components of the system is the preprocessing and tokenization pipeline, which prepares the text to be input into different types of models.

To handle various models, we chose to consider two different pipelines: a **word-level tokenization** technique, suitable for LSTM and Keras embedding layers and in general more classical deep learning models, and **Byte-Pair Encoding** (BPE) for architectures based on transformers.

The word-level tokenization process begins with a preprocessing phase with the objective of standardizing and cleaning the text. Each review is first converted into lowercase, to ensure consistency across words with case-only differences. Then, any URLs are removed, as usually they do not contribute to the actual content of the review, and the same applies to HTML tags and entities, which are also taken out.

Punctuation is then removed to prevent meaningless tokens (commas, full stops etc. which would be inevitably considered as distinct features). Following this, we remove only some stopwords, while keeping others, in particular those we think might influence meaning, such as ‘not’, ‘no’ and others expressing negation. In fact, we know that “not good” is different from just “good”, especially when reviewing a restaurant or a dish, this might

change the meaning of the entire review. Finally, lemmatization is applied in order to reduce words to their base or dictionary form. This reduces the total number of tokens, but also groups together those words which are semantically equivalent.

Once the texts have been cleaned, a tokenizer is trained on the preprocessed corpus using Keras's Tokenizer class. This tokenizer learns a mapping from words to unique indices based on their frequency. We also defined an out-of-vocabulary token to handle words that may appear while performing the task, but that hadn't been encountered during the training phase. The preprocessed texts are then transformed into a sequence of integer tokens, which will then be padded or truncated to a fixed length to guarantee uniform input dimensions.

Secondly, we implemented a subword-level tokenizer using BPE. This technique is preferable especially for transformer-based models. In this case, we used the BertWordPieceTokenizer.

The BPE works by splitting words into subword units based on their frequency, so its handling of OOV words will definitely be better than word-level. For this pipeline, a lighter preprocessing is used: only lowercasing, URL and HTML cleaning are applied. This is because transformer based models benefit from the inclusion of punctuation and stopwords, as they do have an impact on syntactic meaning. Excessive preprocessing would worsen performance, likely discarding information the model is expecting to receive.

The BPE is trained using raw training text and is configured to learn a fixed size vocabulary, and to discard those subwords which appear too frequently. We also explicitly included special tokens which are required by the transformer model directly into the vocabulary. Just like before, the tokenizer encodes each input into a sequence of subword indices, which are then padded/truncated to a fixed length.

All models take as input tokenized and padded sequences derived from raw review texts. To explore the impact of different text representation strategies, we design our experimental setup around the **type of embedding layer used** (randomly initialized vs. pretrained) and the **class of model architecture** (RNN-based vs. Transformer-based).

For the initialization of the embedding layer, we compared two strategies:

1. Standard Keras Embedding Layer: this layer is initialized randomly and trained along with the model itself, so it actually allows the model to learn task-specific embeddings from scratch.
2. Pretrained GloVe Embeddings: these embeddings, contrary to the previous, exploit global statistical information, and are able to incorporate prior semantic knowledge. We specifically used the glove.6B.100d file, which contains 100-dimensional representations of English words, trained on a very large Wikipedia corpus. These vector representations are mapped into the model's vocabulary using an embedding matrix, while those not found in the GloVe dictionary are assigned to 0

vectors. The resulting embedding matrix is then used as non-trainable fixed weights in the Embedding layer to prevent overwriting pretrained information.

The first model consists of a Bidirectional LSTM with **randomly initialized** trainable embeddings. After the input layer, the tokens are passed through an Embedding layer, followed by a Bidirectional LSTM layer, made of 32 units. The embedding dimension was set at 100, a relatively small size that we thought could, given the limited dataset and training time, contain overfitting.

A dropout layer is added after the LSTM to reduce overfitting by deactivating randomly 50% of neurons during training. The final output is then passed through a Dense layer with a softmax to produce the probabilities. The model is trained using Adam optimizer and sparse categorical cross entropy as loss function.

The second model has an identical structure, except we use a **GloVe** based embedding instead of a random one. The size of the GloVe embeddings, 100, was chosen to match one of the standard configurations available in the GloVe 6B corpus, but also to match the one in the first model and to stay relatively computationally efficient.

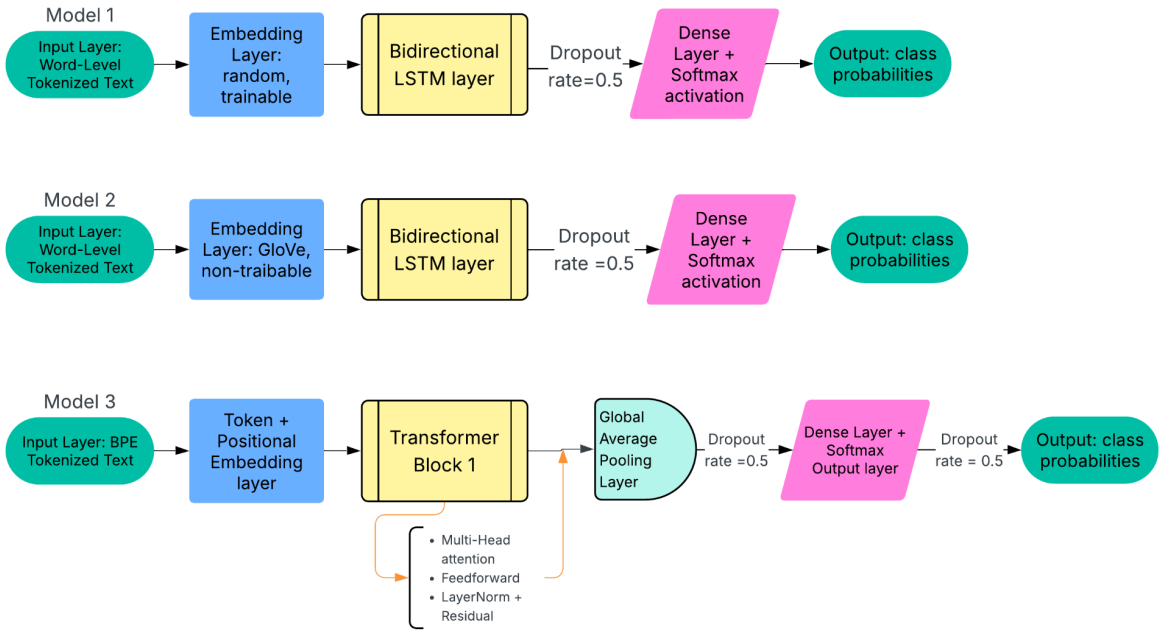
The embedding matrix is built by overlapping the GloVe and the dataset's vocabularies. So, words present in both receive their corresponding pretrained vectors, while others are initialized as zero basically. Also, as the embedding layer is 'non-trainable', so the weights that are pretrained are preserved throughout training, this allows us to measure the effect of semantic knowledge on model performance, by keeping all other parameters the same.

The third and final model adopts a **transformer-based** architecture. Unlike LSTM, which processes text sequentially, it uses self-attention to capture dependencies and relationships between tokens at a more global level, so it's expected to exploit context more efficiently than RNNs.

The input sequences, tokenized with BPE, are first passed through a custom `TokenAndPositionEmbedding` layer, which combines standard token embeddings with learnable positional encodings, so as to allow the model to maintain the order of tokens. The embedding size is set at 32, as well as the feedforward layer size, to keep the model more compact and reduce complexity as well as the risk for overfitting.

The core of the model consists of a transformer block, with its own multi-head self-attention, feedforward layer and residual connections, also considering layer normalization and dropout for regularization and stability. After the transformer layer, a global average pooling layer is used to compress the sequence into a fixed-size vector.

This is then passed through a Dense layer with ReLu activation and L2 regularization, which penalizes weights which are too large, and finally a softmax to output class probabilities. This model has the same objective function and optimizer as the LSTM variants.



We can say the first two models allow a direct comparison between embeddings from scratch versus pre-trained ones, while the third model explores the shift in performance when using an attention-based mechanism and subword-based vocabularies.

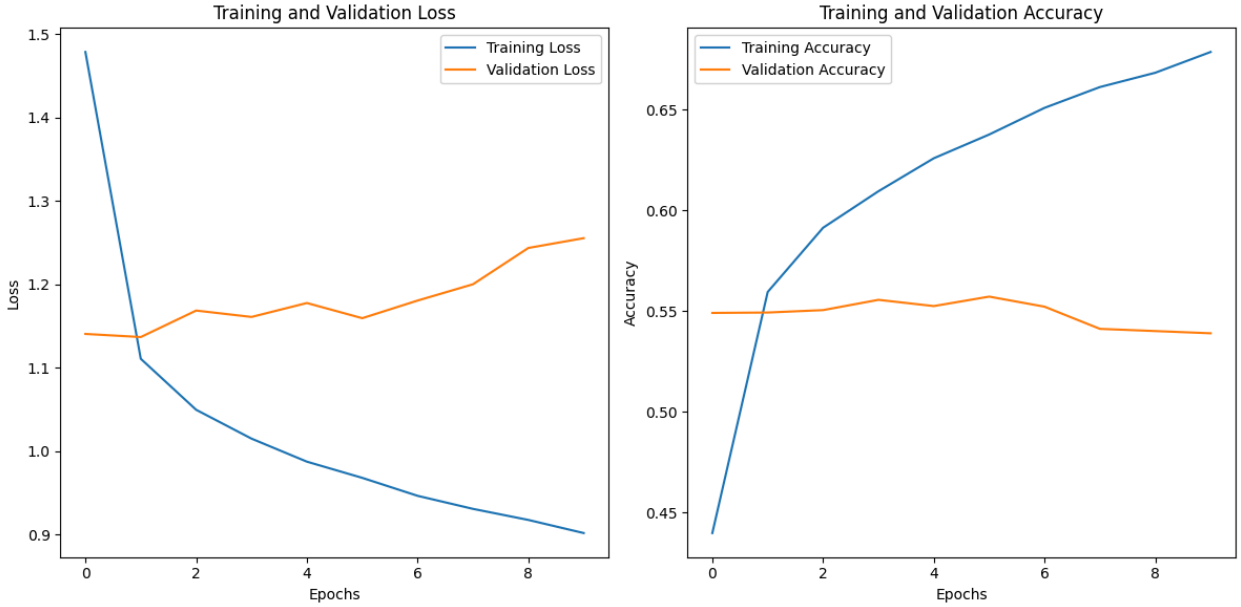
Result and Analysis

To assess the performance of the proposed models, we trained three distinct architectures over 10 epochs with a batch size of 128. The relatively small number of training epochs was chosen due to constraints in computational resources, which made longer training very demanding. Each model was saved to disk to avoid redundant retraining and to allow reproducibility.

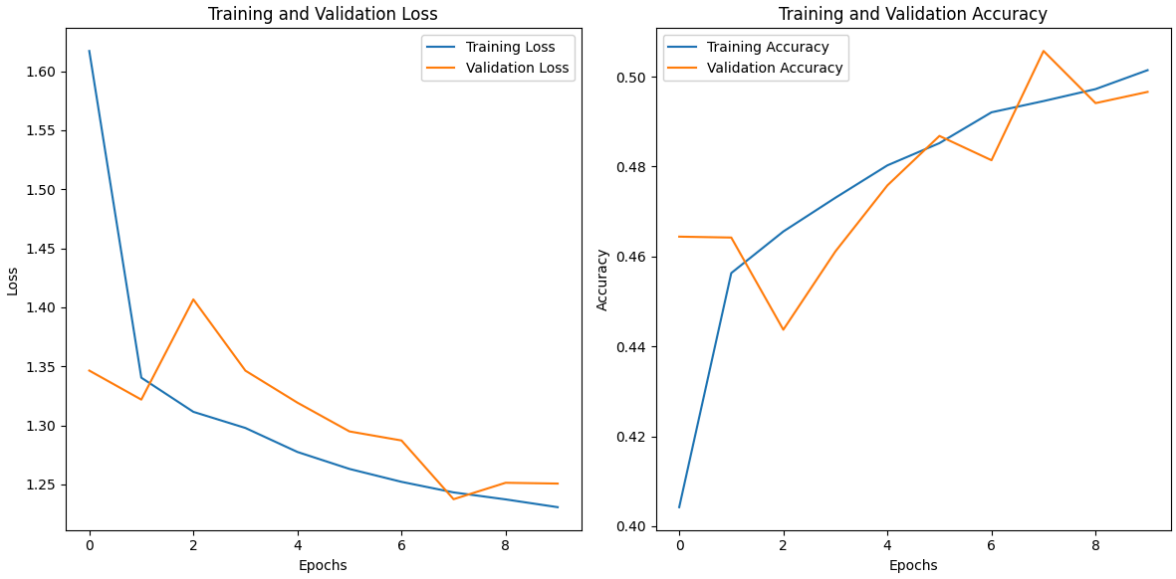
Model	Test accuracy	Test loss
Model 1	0.5230	1.2512
Model 2	0.4767	1.2647
Model 3	0.4966	1.4579

For **Model 1** (LSTM with a randomly initialized embedding layer), training loss dropped very quickly and accuracy rose to around 68%, which shows that the model learned the training data pretty well. However, the validation loss actually increased over time, and the validation accuracy stayed around 55%. This indicates overfitting: the model did very well

on the training data but failed to generalize to new, unseen data. This behavior was probably caused by the lack of regularization in the embedding layer and the LSTM layer's ability to memorize patterns in the training too much, which leads to bad performance in the test.

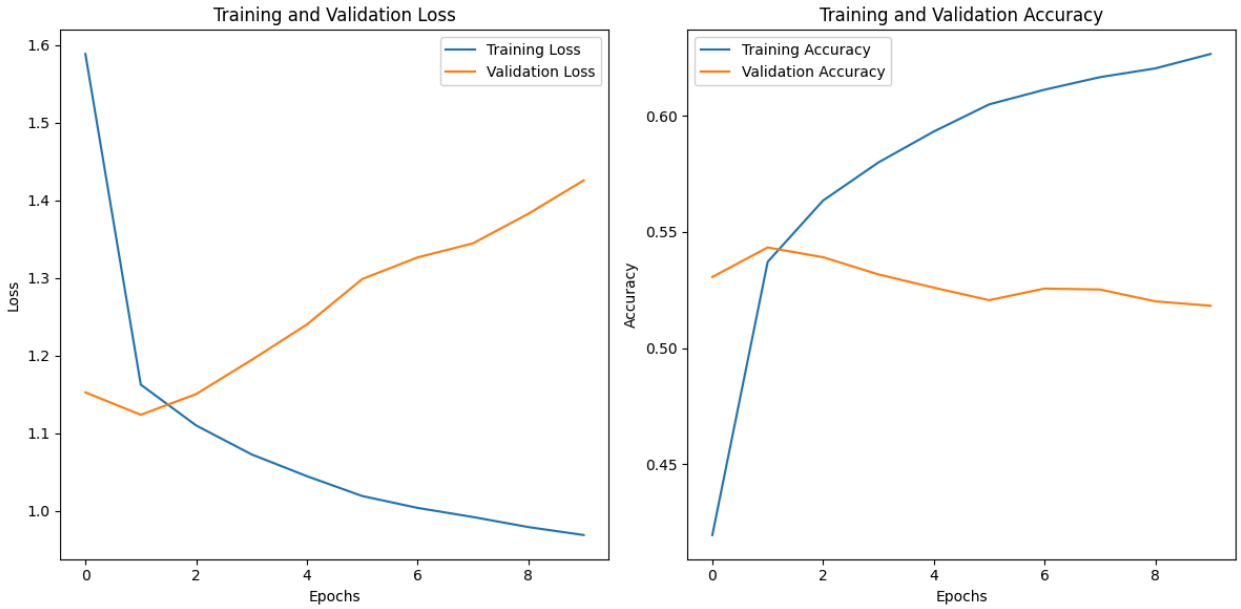


In **Model 2** (LSTM with pretrained GloVe embeddings), both training and validation loss went down fast, and training accuracy reached around 50%. The validation accuracy also improved slightly and reached about 47%, which is a bit worse than Model 1. Even though the training accuracy was lower than in Model 1, the more stable validation results suggest that the model with GloVe embeddings generalized better and didn't overfit as much. This shows that using pre-trained word vectors can help guide the model and make learning more stable, even if we don't train the embeddings further.



Although we expected the transformer model (**Model 3**) to perform better in terms of generalization, the training curves showed differently. The plot shows how the training loss actually decreases steadily, reaching below 1.0, and training accuracy rises to over 64%, indicating that the model is actually learning something from the training.

However, the validation loss starts increasing after just two epochs, rising from around 1.1 to more than 1.4 by the final epoch. Similarly, validation accuracy peaks early (around 55%) and then declines. This suggests that the transformer's higher complexity may require stronger regularization, more training data, or longer training.



Despite the overfitting, to understand how models behaved beyond curves and numbers, we artificially created some reviews to let them predict the number of stars. In this case, the predictions actually show some level of consistency in the results:

Review	Model 1 prediction	Model 2 prediction	Model 3 prediction
"The pasta was fresh and flavorful... Definitely coming back!"	4	4	4
"Terrible experience... rude... cold food..."	0	0	0
"Average place... Not bad, not great."	1	1	1
"It was okay... had to wait an hour, but it was worth it."	1	0	1

“Overpriced and overrated… fries tasted like cardboard.”	0	0	1

These results might be an indication that, for the most part, models agree in their predictions. Some inconsistencies might be a result of some patterns present in the training that the models, which we have seen generally tend to overfit, rely excessively on.

However, the predictions across models remained relatively consistent. Even in the case of more ambiguous reviews, none of the models produced outputs that appeared to be completely random.

Conclusion

In this project, we trained and compared three different models for review sentiment classification. In particular, we tried to explore different strategies in all steps of the model creation process: first with the embeddings, then with the model architecture.

We started with the preprocessing of data: applying lowercasing, URL and punctuation removal, HTML cleaning and lemmatization. These steps were applied specifically for the word-level tokenization path, which was used in the LSTM models.

Next, we implemented two parallel tokenization approaches: a word-level tokenizer, compatible with Keras and GloVe embeddings, and a subword tokenizer using BPE (Byte Pair Encoding) via BertWordPieceTokenizer, to be used in the transformer-based model.

We then prepared and trained three models:

1. **Model 1**: A bidirectional LSTM with a trainable embedding layer.
2. **Model 2**: The same architecture as Model 1 but using frozen pre-trained GloVe embeddings.
3. **Model 3**: A transformer-based model that used token and position embeddings with BPE input.

Who did what

- Pietro Pratelli: report, embeddings, model selection, model evaluation; most difficult parts: GloVe embeddings
- Sara Ruggeri: report, preprocessing, embeddings, evaluation; most difficult parts: GloVe

AI policies

Code assistant from Colab was very useful: it had many time traces of all the previous and successive code blocks. This makes the suggestions very precise, adherent and overall well compatible with the rest of the code. Sometimes it was not perfect, indeed the code had some errors, but they were relatively easy to solve.

Gemini was also useful because, as it's possible to link it to Drive, you can give it the whole notebook directly. Also, the code generated was generally of high quality. However, specifically for the downloading part of GloVe embeddings, it was not very convenient, because the function did not work at all and each time we asked it to solve an error, the newly generated code was as inaccurate as before.

ExplainError was indeed the most useful feature. It is overall very precise and able to solve the issues in the code, not just stopping the 'scan for the cause' at the block that raised the error, but also exploring other possible causes in the entirety of the notebook.

Prompt	Result	Notes

I want to save my tokenizers, models and glove embeddings so I don't have to retrain them again each time. Make folders to save files and functions to save and get them back. Generate code and also check the already existing one.	`EMBEDDINGS_DIR = './glove_embeddings'`
	`TOKENIZERS_DIR = './saved_tokenizers'`
	`MODELS_DIR = './saved_models'`
	
	`os.makedirs(EMBEDDINGS_DIR, exist_ok=True)`
	`os.makedirs(TOKENIZERS_DIR, exist_ok=True)`
	`os.makedirs(MODELS_DIR, exist_ok=True)`
	`def save_keras_model(model, path):`
	`  model.save(path)`
	`def load_keras_model(path):`
	`  if os.path.exists(path):`
	`  model = load_model(path)`
	`  return model`
	`  else:`
	`  print(f"No model found at {path}")`
	`  return None`
		We also gave Gemini the access to the entire original file, so it was able to devise methods that matched the already existing code.

The suggestion to use arrays was given by Colab AI assistant. Some lines of the remaining code were created by Google Colab's assistant suggestions.	`train_labels_array = np.array(train_labels)`
	`test_labels_array = np.array(test_labels)`
	`plt.subplot(1, 2, 1)`
	`plt.pie(`
	`  [np.sum(train_labels_array == i) for i in range(NUM_CLASSES)],`
	`  labels=[f"Class {i}" for i in range(NUM_CLASSES)],`
	`  autopct='%1.1f%%')`
	`plt.title('Train Set Label Distribution')`
	`plt.subplot(1, 2, 2)`
	`plt.pie(`
	`  [np.sum(test_labels_array == i) for i in`	

	range(NUM_CLASSES)],
	    labels=[f"Class {i}" for i in range(NUM_CLASSES)],
	    autopct='%1.1f%%')
	plt.title('Test Set Label Distribution')
	plt.show()	

Generate a function to check if there is already a word level tokenizer, if not, train it and save it locally.	def get_word_level_tokenizer(texts, vocab_size, oov_token, tokenizer_path):
	    tokenizer_file = os.path.join(tokenizer_path, "word_level_tokenizer.pkl")
	    if os.path.exists(tokenizer_file):
	        print("Loading existing tokenizer...")
	        with open(tokenizer_file, 'rb') as f:
	            tokenizer = pickle.load(f)
	    else:
	        print("Training new tokenizer...")
	        tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_token)
	        processed_texts = [preprocess_text_word_level(text) for text in texts]
	        tokenizer.fit_on_texts(processed_texts)
	        print("Saving tokenizer...")
	        with open(tokenizer_file, 'wb') as f:
	            pickle.dump(tokenizer, f)
	    return tokenizer
		Generated initially by Gemini, but it wasn't working and causing an error, so clicking the ExplainError button on colab helped implement the changes that made it work (before pickle was not being used).

I want to load glove embeddings in my project and to use the embedding layer from keras with Glove. generate functions to use it.	def download_glove_embeddings(embeddings_dir, glove_url, glove_file_name):
	    glove_zip_path = os.path.join(embeddings_dir, os.path.basename(glove_url))
	    glove_extracted_dir = os.path.join(embeddings_dir, os.path.splitext(os.path.basename(glove_url))[0])
	    glove_file_path = os.path.join(glove_extracted_dir, glove_file_name)
	
	    if not os.path.exists(glove_file_path):
	        print(f"Downloading GloVe embeddings from {glove_url} (this may take a while)...")
	        try:
	            _create_unverified_https_context = ssl._create_unverified_context
	        except AttributeError:
	            pass
	        else:
	            ssl._create_default_https_context = _create_unverified_https_context
	        tf.keras.utils.get_file(
	            fname=os.path.basename(glove_url),
		The initial structure caused an error, specifically, the generated download function which pointed to the wrong file path. Colab assistant suggested adding ssl._create_unverified_context to bypass the issues, but eventually we had to put directly the correct path to make it work.

	origin=glove_url, extract=True, cache_dir='.', cache_subdir=embeddings_dir) print("GloVe embeddings downloaded and extracted.") else: print(f"GloVe embeddings ({glove_file_name}) already exist locally.") return glove_file_path	
Decorator suggested by Explain error when unable to retrieve the model because of the impossibility of getting custom blocks.	@keras.saving.register_keras_serializable()	