IN5550 – Spring 2024 — Obligatory 1. Document Classification

Sushant Gautam and **Fernando Vallecillos Ruiz**

2024/02/23

1 Dataset and pre-processing

1.1 Pre-processing

To pre-process the the given dataset before using it for down-stream tasks, several steps were performed.

1. NLTK's WordNet lemmatizer was employed to reduce words in abstracts to their base or dictionary form, aiding in standardizing the text for analysis.
2. NLTK's stopwords list for English was used to filter out common words that typically do not carry significant meaning in text analysis. These stopwords were then removed from each abstract to improve the relevance of the remaining words.
3. Regular expressions were utilized to substitute non-word characters and digits with spaces, ensuring consistency in the structure of the text.
4. All remaining words were converted to lowercase to avoid duplications due to case sensitivity.

The resulting processed abstracts were then for all the Tasks below, effectively preparing the textual data for further analysis or modeling tasks.

1.2 Data Splitting

The training and validation datasets were obtained using the `train_test_split` function from the scikit-learn library, where the abstract served as the features and label served as the target labels. The splitting was performed with a test size of 30% and a random seed of 42 to ensure reproducibility. Additionally, the `stratify` parameter was set to label to maintain the same distribution of labels in both the training and validation sets. This ensured that each class had representative samples in both datasets.

1.3 Dynamic TextClassifier

The `TextClassifier` class could dynamically constructs its recurrent and fully connected layers based on the input size and the provided layer mapping. This dynamic nature allows for flexibility in defining the architecture of the classifier in experimentation. Upon initialization, the class iterates through the layer map, creating linear layers sequentially according to the specified sizes as well as the recurrent layer, if specified, with the desired number of size and layers. The number of layers, recurrent layer type, bi-directional nature, their sizes and the dropout rate depend on the input parameters provided during instantiation. This dynamism enables the model to adapt to various input dimensions and layer configurations, making it suitable for experimentation. This class also could instantiate a embedding layer, if required by experiment, as per the input parameter with option to freeze or fine-tune.

ReLU was used as the activation function of choice all over the experiments in the linear layer. Cross-entropy loss was employed to calculate loss as well as measure the performance of a classification model in each epoch. The Adam optimizer was used to minimize the loss function during the training.

Also, on the loss curves plotted below, the **validation loss is lower than training** on the curve because of the dropout in training, which only penalizes training and not validation process.

2 Part 1: Bag of words baseline

2.1 Vectorization

The document vectors were obtained using three different vectorization techniques: **Count Vectorization** represents each document as a vector, where each element corresponds to the frequency of a term in the document. **Binary Vectorization**, on the other hand, simply represents the presence or absence of a term in the document, assigning 1 if the term is present and 0 otherwise. **TF-IDF Vectorization** takes into account not only the frequency of a term in a document but also its importance in the entire corpus, giving higher weights to terms that are frequent in a document but rare in the corpus. These techniques transformed the abstracts from the dataset into numerical vectors, enabling further analysis and modeling of text data.

2.2 Experiments

2.2.1 Different variations of BoW features

Three experiments were conducted to evaluate the performance of different Bag-of-Words feature representations for a text classification task as shown in Figure 1. All experiments were trained for 50 epochs, with a single linear layer of 128 hidden size, with slight variations of learning rate. The experiments were evaluated using a classifier model trained on the train-split and validated on our validation set. The loss curves are shown in in Figure 1.

Vectorizer	Precision	Recall	F1(macro)	Comment
(a) Binary *lr=0.0005*	0.831730	0.832194	0.831756	each feature represents the presence or absence of a term in the document
(b) Count *lr=0.0005*	0.827573	0.827820	0.827534	each document as a vector of term frequencies
(c) TF-IDF *lr=0.001*	0.838165	0.837962	**0.837931**	weighs the importance of each term in a document relative to its frequency across the corpus

Table 1: Experiment results with different variations of BoW features reported at epoch 50 for each.

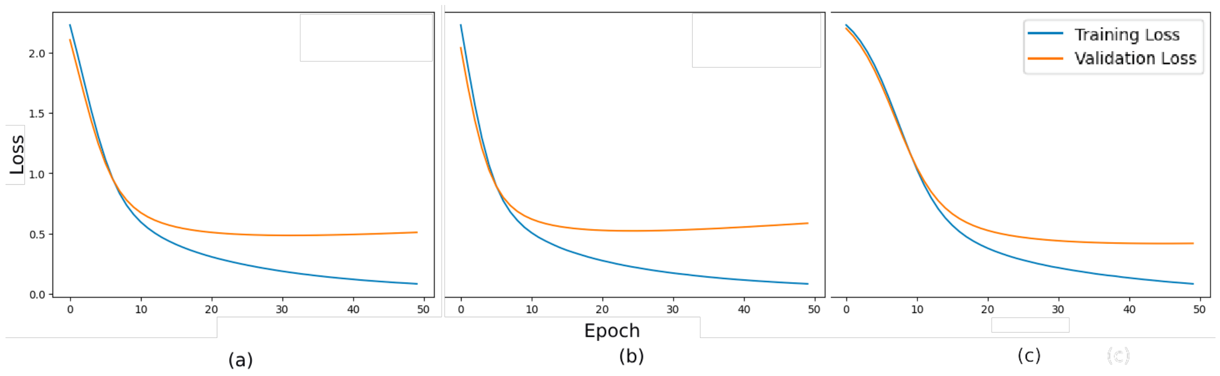

Figure 1: Loss curves for experiments with different configurations of BoW features as in Table 1

The results showed that the TF-IDF vectorizer achieved the highest F1 score, followed by the binary vectorizer, and the count vectorizer. These results suggest that **incorporating the TF-IDF into the feature representation yields better performance** compared to binary or count-based representations.

2.2.2 Different number of layers

As results from previous experiments suggested the use of TF-IDF into the feature representation, it will be used in this part of the experiment with varying number of layers in the classifier as shown in Table 2.

Overall, it seems that a single hidden layer of size 256 yielded the best balance between model complexity and performance. Also it is worth noting that all experiments overfitted after around epoch

Hidden-Layer Shape	**Precision**	**Recall**	**F1(macro)**	**Comment**
(a) [256] *lr=0.001*	0.831001	0.830927	**0.830839**	
(b) [4096] *lr=0.0005*	0.824972	0.825111	0.824911	lr decreased as model complexity increased
(c) [4096, 128] *lr=0.0005*	0.818387	0.818362	0.818255	overfitted after epoch 18 with **max F1: 0.845**
(d) [512, 128, 16] *lr=0.0005*	0.827404	0.828442	0.827452	

Table 2: Experiment results with different variations of layer shape and size

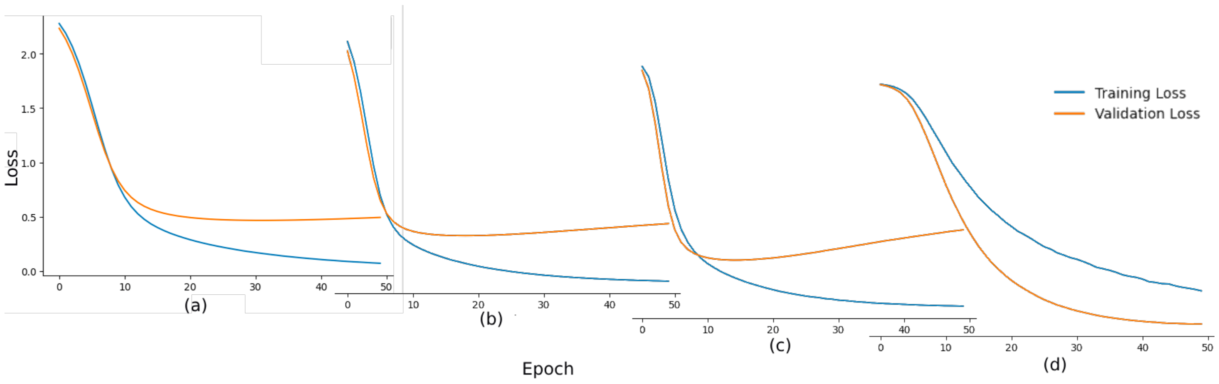

Figure 2: Loss curves for experiments with different configurations of hidden layers as in Table 2.

18, as show in Figure 2, and the max F1, before it started to overfit, was at around 0.845 for last two experiments with multiple layers. But, scores at epoch 50 is reported for fair comparison. This indicates that optimizing other aspects of the model or dataset or trying newer architectures might be more fruitful for improving performance rather than solely focusing on increasing model complexity.

3 Part 2: Word Embeddings

3.1 Pretrained Word Embedding Models

Embedding Model	Vocab	Unique Coverage	Total Coverage
GloVe-Twitter-25	1,193,514	43.01%	92.57%
Word2Vec-Google-News-300	3,000,000	40.23%	94.35%
GloVe-Wiki-Gigaword-300	400,000	**52.06%**	**96.78%**
FastText-Wiki-News-Subwords-300	999,999	47.39%	96.40%

Table 3: Token coverage of Pretrained Word Embedding Models in Gensim to our corpus.

The total number of tokens in our corpus is 7,950,716 and the number of unique tokens is 85,756. Table 3 shows the token coverage of pretrained word embedding models in Gensim to our corpus. From the table, we observe that each pretrained word embedding model has varying coverage of unique tokens and total tokens from our corpus. The GloVe-Wiki-Gigaword-300 model has the highest coverage of unique tokens (52.06%) and total tokens (96.78%) from our corpus among the models listed. However, it has a relatively smaller total vocabulary size compared to Word2Vec-Google-News-300, which has the largest vocabulary size. Considering the trade-off between coverage and vocabulary size, **the GloVe-Wiki-Gigaword-300 model seems to be the most suitable for our application** as it covers a significant portion of our corpus.

3.2 Corpus for Word Embedding

In the pursuit of increasing the unique token coverage, we wanted to train our own word embedding model, for which we sought a suitable corpora. While exploring options like Gensim corpora and the Fox Corpora directory, we found them unrelated to our training dataset, comprising arXiv abstracts. Consequently, we opted for a large arXiv abstract dataset available on HuggingFace (gfissore/arxiv-abstracts-2021), encompassing 2 million abstracts. Augmenting this dataset with our existing 80,000 abstracts, we accumulated a corpus of 2,080,000 samples. Following the preprocessing steps detailed in Section 1.1, we standardized the data, resulting in a corpus containing **180 millions total word tokens** and 471 thousands unique tokens across the 2,080,000 samples. With this comprehensive corpus prepared, we are now poised to commence training our word embedding model.

3.3 Training Custom Word Embedding Model

The process of training our own word embedding model began with the selection of the FastText model from the gensim library. This choice was made due to its recent advancements and its ability to provide superior embeddings for morphologically rich languages compared to older models. We configured the model with parameters including a vector size of 100, a window of 5, a minimum count of 5, 10 epochs, and skip-gram architecture.

However, it became apparent that Gensim's FastText implementation lacked loss tracking functionality. Despite attempts to monitor the training loss using *model.get_latest_training_loss()*, the function consistently returned zero. Undeterred, we switched our approach to Word2Vec, which offered built-in loss logging within the Gensim library. Two models were trained, one with a vector size of 300 and 10 epochs, and another for 20 epochs. Subsequent evaluation through manual similarity searches revealed that the trained Word2Vec model with a vector size of 300 and trained for 20 epochs yielded the best results. It was chosen for further experimentation as our custom embedding model, boasting an impressive 99.8% unique token coverage. The training loss curve of that model is shown in Figure 3.

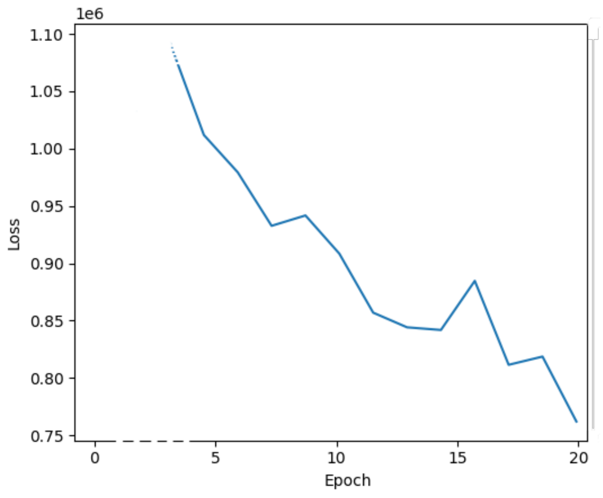

Figure 3: Loss curves for training Word2Vec model on the custom corpus.

3.4 Experiments

3.4.1 Merging Sequence Embeddings

Experiment	Operation	Hidden Shape	Parameters	Precision	Recall	F1
A-1	Average	[64]	19,914	0.85	0.85	0.85
A-2	Max	[512]	159,242	0.79	0.78	0.78
A-3	Sum	[512]	159,242	0.84	0.84	0.84
A-4	Concat	[512]	466,442	0.84	0.83	0.83

Table 4: Evaluating different strategies for Merging Sequence Embeddings

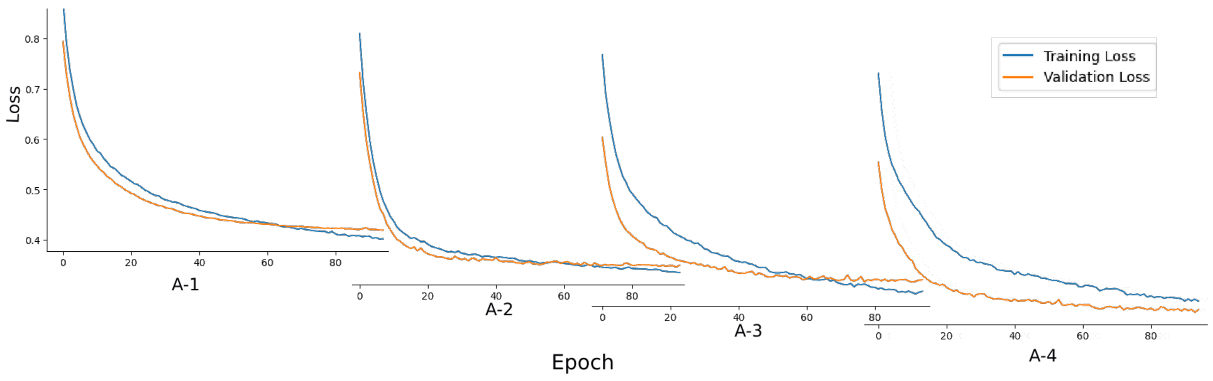

Figure 4: Loss curves for different strategies for Merging Sequence Embeddings as in Table 4

In order to derive a single representation from a sequence of embeddings, several strategies were employed, each tailored to capture different aspects of the sequence's information. The used strategies are as follows:

- **Average:** This method computes the mean of all non-padding embeddings in a sequence. It is particularly useful for obtaining a general representation of the sequence while ignoring the effects of padding.
- **Sum:** Summation aggregates the embeddings by summing them, capturing the total information present in the sequence without normalization.
- **Max:** This operation takes the maximum value across embeddings for each dimension, providing a representation that captures the most salient features of the sequence.
- **Concat:** In an attempt to create a richer representation, the average, sum, and max embeddings are concatenated. This combines the benefits of each individual method, potentially leading to a more informative representation.

The results are tabulated in Table 4 and the loss curves are shown in Figure 4 for our best custom Word2Vec word embedding model. Experiment A-1 with an average merge operation achieves the highest precision, recall, and F1 score of 0.85, indicating strong overall performance with relatively fewer trainable parameters (19914) and a simpler model layer map [64]. Experiment A-2 and A-3 with max and sum merge operations respectively also perform well, although with slightly lower scores and a larger number of trainable parameters (159242). Experiment A-4, employing concatenation, shows a comparable performance but with a notably higher number of trainable parameters (466442), suggesting potential overfitting or inefficiency in parameter utilization. Overall, A-1 presents the most efficient and effective approach among the tested merge operations.

3.4.2 Handling Out-of-Vocabulary Words

To address the issue of Out-of-Vocabulary (OOV) words, different strategies were explored to assign embeddings to words not present in the pre-trained model vocabulary:

- **Mean:** Assigns the mean of all embeddings in the model to the OOV word. This provides a neutral, average representation for unknown words.

$$\text{word\_model}[''[UNK]''] = \text{mean}(\text{word\_model.vectors})$$

- **Random:** Generates a random embedding within a specified range. This introduces variability for OOV words, potentially capturing diverse unknown contexts.

$$\text{word\_model}["[UNK]"] = \text{random}(\text{low} = -0.3, \text{high} = 0.3)$$

- **Zero:** Assigns a zero vector to the OOV word. This method effectively neutralizes the unknown word's contribution to the model's predictions.

$$\text{word\_model}["[UNK]"] = \text{zeros}(\text{word\_model.vector\_size})$$

Experiment	**Precision**	**Recall**	**F1 Score**
B-1: OOV handling- Mean	0.50	0.49	0.49
B-2: OOV handling- Random	0.50	0.50	0.50
B-3: OOV handling- Concat	0.49	0.49	0.49

Table 5: Evaluating different Out-of-Vocabulary (OOV) words handling strategies

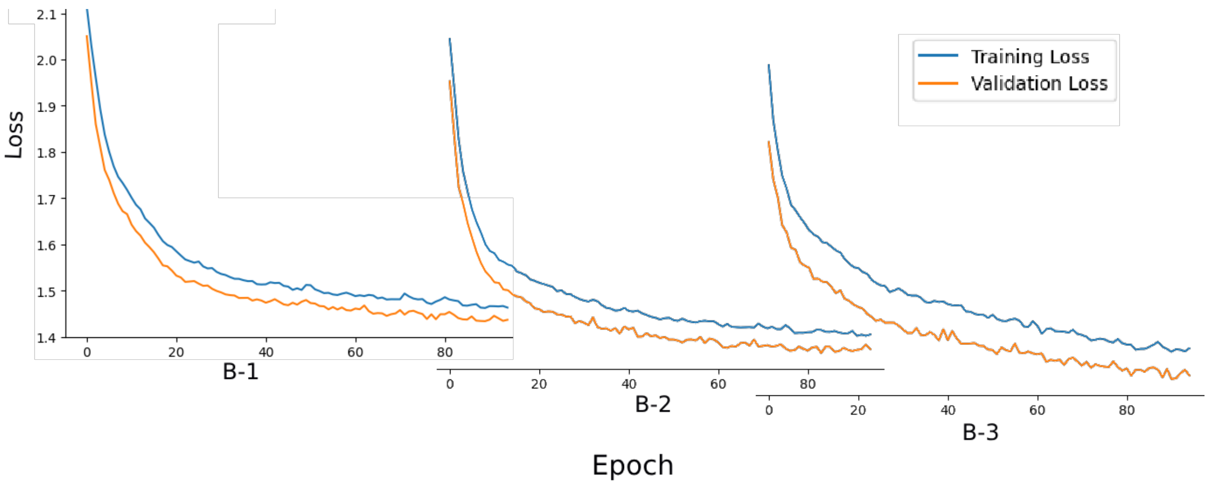

Figure 5: Loss curves for different Out-of-Vocabulary (OOV) words handling strategies as in Table 4

These strategies were implemented to enhance the model's robustness to variations in input data, ensuring that it can handle sequences with varying lengths and compositions, as well as accommodate words that are not found in its pre-existing vocabulary. The results are tabulated in Table 5 and the loss curves are shown in Figure 5, all trained with **lr= 0.01** and 44,042 numbers of trainable parameters over "glove-twitter-25" word embeding from Gensim.

In Experiment B-1 with mean OOV handling, the scores are slightly lower. Experiment B-2, utilizing random OOV handling, shows slightly improved performance. Experiment C-1, employing concat OOV handling, exhibits results almost comparable to B-1. These findings suggest that the choice of OOV handling method has marginal impact on model performance, possibly indicating robustness against variations in handling strategies.

3.5 Pre-trained Embedding Models

Experiment Name	Unk Tokens	Class. Train. Params.	Precision	Recall	F1 Score
C-1: glove-twitter-25	7.42%	44042	0.49	0.49	0.49
C-2: word2vec-google-news-300	5.64%	466442	0.76	0.76	**0.76**
C-3: glove-wiki-gigaword-300	3.21%	466442	0.77	0.77	**0.76**

Table 6: Evaluating different Embedding Models

In the experiments comparing different pretrained embedding models as shown in Table 6, we observed varying levels of performance in terms of achieving validation accuracy. Specifically, embeddings trained on larger corpora, like word2vec-google-news-300 and glove-wiki-gigaword-300, displayed lower

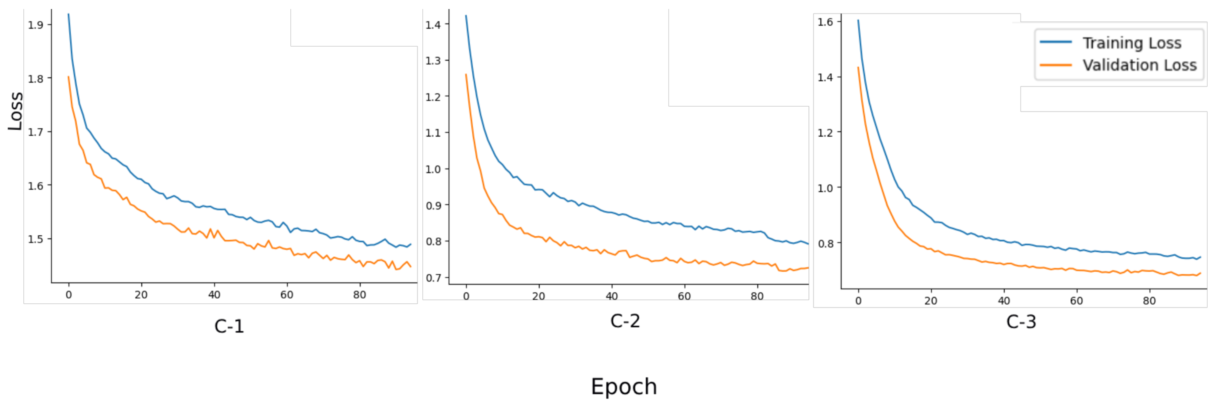

Figure 6: Loss curves for different Embedding Models as in Table 6

percentages of unknown tokens and higher accuracy compared to embeddings from smaller corpora, such as glove-twitter-25. Despite having the same hidden layer shape of [512] and learning rate of 0.01 across all experiments, the models benefited differently from the richness and diversity of the embedding model, resulting in varied performance metrics.

3.6 Pre-trained vs. Self-trained Embeddings

Experiment Name	Unk Tokens	Hidd. Layer Shape	Precision	Recall	F1
C-2: word2vec-google-news-300	5.64(%)	[512]	0.76	0.76	0.76
A-4: custom best Word2Vec_300	0.20(%)	[512]	0.84	0.83	**0.83**
D-1: custom FastText_100	0.20(%)	[256]	0.82	0.82	0.82

Table 7: Comparison of Pretrained Word Embedding models with Custom Models

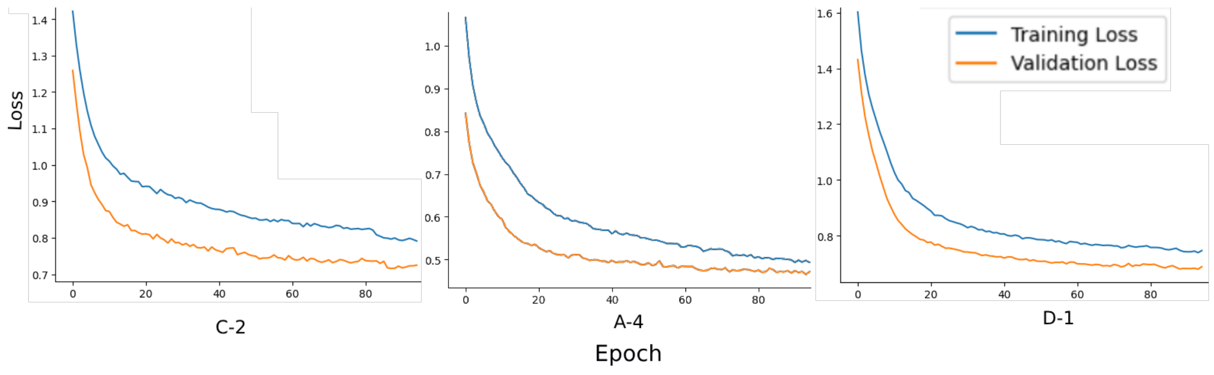

Figure 7: Loss curves for different custom Embedding Models as in Table 7

The experiments, as outlined in 7, show that the custom word embeddings outperform the pretrained embeddings consistently across different architectures and datasets, indicating that fine-tuning embeddings on specific data can lead to better performance. Despite the pretrained embeddings demonstrating significantly lower rates of unknown tokens, the custom embeddings achieve higher accuracy metrics, suggesting a more effective representation of the underlying semantic information in the data.

3.7 Word Embeddings Fine-tuning

The experiments compare frozen embedding (A-4) with fine-tuned embedding (E-1), as shown in Figure 8, both employing a hidden layer size of 512. Despite the considerable difference in trainable parameters, fine-tuned embedding exhibits only a marginal improvement in performance over frozen embedding across evaluation metrics (0.85 vs. 0.84), suggesting limited benefits of fine-tuning in this scenario.

The marginal improvement in performance despite fine-tuning the embedding can be attributed to the limited complexity of the task or the dataset's characteristics. In scenarios where the pre-trained

Experiment Name	**Trainable Parameters**	**Precision**	**Recall**	**F1**
Experiment A-4: Frozen Embedding	466,442	0.84	0.83	0.83
Experiment E-1: Finetune Embedding	46,259,942	0.85	0.85	0.85

Table 8: Evaluating performance of Frozen vs Fine-tuned embedding layer.

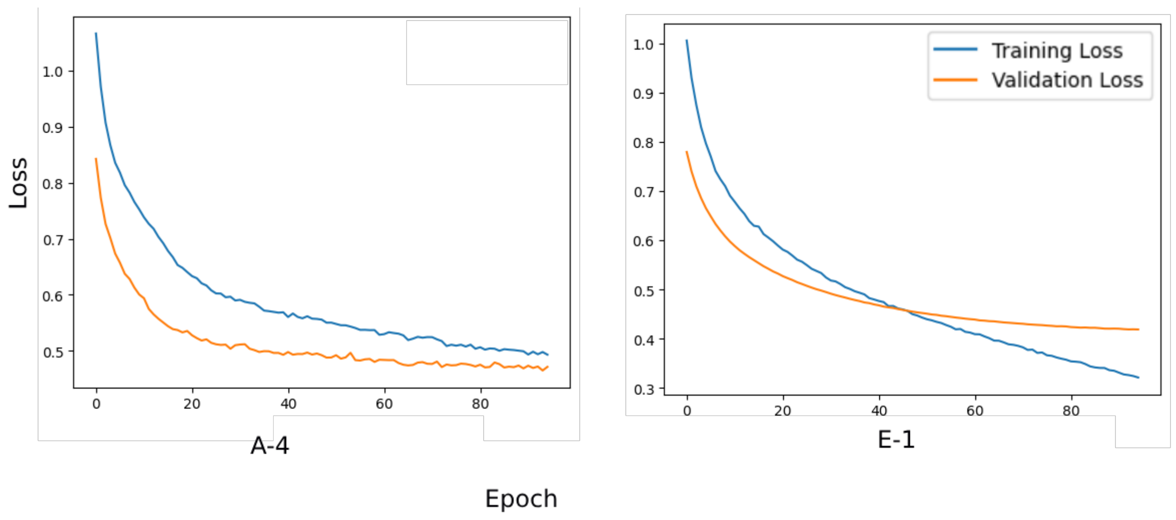

Figure 8: Loss curves for Frozen vs Fine-tuned embedding layer as shown in Table 8

embeddings already capture relevant semantic information, further fine-tuning might not significantly enhance model performance. Additionally, the relatively small hidden layer size might restrict the model's capacity to fully exploit the fine-tuned embeddings, leading to negligible performance gains.

4 Part 3: Recurrent neural networks

Differently from the previous sections. The number of parameters in the following models has a vastly greater number of parameters. In the following models, we vary from 58 thousand trainable parameters if we freeze the embedding weights towards up to 50 million trainable parameters in some of the models without frozen embeddings. The number of parameters vary from feed-forward neural classifiers that have the least to LSTMs that have the highest amount.

The performance achieved is quite similar to the one in the previous section. This can be caused due to the nature of the data. Short sequences can undermine the advantage of RNNs, GRUs, and LSTMs in capturing long-term dependencies. Furthermore, documents may have simple patterns that do not require complex structures.

In the following subsections, the different hyperparameters used in the experiments are described.

4.1 Types of RNN

The first experiment consists of changing the architecture of the network. This choice has a great impact because it dramatically changes how the final result is calculated. Since the default value of 10 epochs was not conclusive, we decided to increase the number to 30 epochs for this experiment. There is a clear advantage of GRU and LSTM over simple RNN. Analyzing the loss values over the epochs, we can see how the performance varies and starts to degrade. This can be a sign that the model is too simple and not capable of finding a relationship between the pairs of inputs and outputs. On the other hand, the GRU and LSTM models achieve decent results. By graphing the training and validation loss, we can appreciate the crossing point, and therefore start of the overfitting process, around epochs 15 and 20 for the GRU and LSTM, respectively.

Model	Accuracy	Precision	Recall	Macro-F1 Score
RNN	0.51	0.51	0.51	0.50
GRU	0.85	0.85	0.85	0.85
LSTM	0.84	0.85	0.84	0.84

4.2 Hidden states combination

There are different approaches to combine the sequence of hidden states of a recurrent neural network. One of the most simple and common approaches is to take the last hidden state of the recurrent process. On the other hand, we could extract the first hidden state of the sequence. Although less common, it can be useful in scenarios where the beginning of the sequence contains the most important information. In our scenario, the performance degrades to almost half. We can also aggregate the hidden states over the sequence. We have decided to use several approaches for this aggregation, *max pooling* and *addition*. The next approach adds the initial and final state of the sequence which are generally regarded as the most important ones. This achieves a slight increase in performance. We then look for methods capable of aggregate information from all the hidden states. We continue with averaging the values of all hidden states, which improves the performance further. The next approach sums all the hidden without any averaging. The last approach aggregates all hidden states applying a maximum-pooling operation over all time steps. The last two operations achieve the best results. This increase in performance may be due to the fact that we are accumulating information from all time steps.

Model	Combination	Accuracy	Precision	Recall	Macro-F1 Score
RNN	Last	0.64	0.66	0.64	0.63
RNN	First	0.34	0.34	0.34	0.33
RNN	Addition	0.68	0.72	0.68	0.68
RNN	Average	0.79	0.79	0.79	0.79
RNN	Sum	0.84	0.84	0.84	0.84
GRU	Sum	0.85	0.85	0.85	0.84
LSTM	Sum	0.85	0.85	0.85	0.85
RNN	Max	0.83	0.83	0.83	0.83
GRU	Max	0.85	0.85	0.85	0.85
LSTM	Max	0.84	0.85	0.84	0.84

4.3 Bidirectionality

The bidirectionality process in an RNN relates to processing the input from both forward and backward directions. This approach uses two RNNs, which results in two outputs that are typically combined at the last time step. In our case, we have decided to concatenate the last hidden state of both directions. Similarly to Subsection 4.1, the GRU and LSTM architectures have an advantage over simple RNN. However, looking at the loss we can appreciate how in this approach the models converge faster (around epoch 10 in both architectures). This may be due to the model capturing information in both directions and therefore increasing its understanding of the context.

Model	Accuracy	Precision	Recall	Macro-F1 Score
BiRNN	0.56	0.58	0.56	0.56
BiGRU	0.84	0.84	0.84	0.84
BiLSTM	0.84	0.83	0.84	0.83

4.4 Number of recurrent layers

An increased number of recurrent layers depends on the temporal processing capabilities of the model and increases its complexity. This allows it to capture longer sequential dependencies within the data, but it can also result in a model harder to train. In this case, the RNN receives a slight improvement on its capabilities, but it is still unable to converge. This indicates that although the model is more complex, it may not be complex enough to converge successfully yet. On the other hand, the GRU and LSTM models keep very similar results to their previous versions.

Model	Number of recurrent layers	Accuracy	Precision	Recall	Macro-F1 Score
RNN	2	0.53	0.49	0.53	0.48
RNN	3	0.67	0.70	0.67	0.67
GRU	2	0.84	0.84	0.84	0.84
GRU	3	0.83	0.84	0.83	0.83
LSTM	2	0.82	0.83	0.83	0.82
LSTM	3	0.83	0.83	0.83	0.83

4.5 Number of linear layers

An increased number of linear layers offers higher abstraction of features and increases the complexity of the model. However, it can also lead to overfitting and increased computational costs. On the other hand, fewer layers reduce the risk of overfitting and training time. But it may result in a model that is too simple to handle some tasks. Since our default model has only one linear layer, we have decided to add more linear layers. We also increase the number of epochs to 40 to account for the higher complexity of the model. We decide to choose shapes with decreasing size to try to force feature compression. Due to the increased complexity, the model converges more slowly when additional layers are inserted, which was expected. The results show almost no change, therefore we conclude that the linear layers are not holding back the model.

Model	Shape of Linear Layers	Accuracy	Precision	Recall	Macro-F1 Score
GRU	[128, 64]	0.84	0.84	0.84	0.84
GRU	[128, 64, 32]	0.83	0.84	0.83	0.83
GRU	[256, 128, 64]	0.84	0.84	0.84	0.84

4.6 Learning rate

The choice of learning rate is extremely important when designing a neural network. A higher learning rate results in faster convergence and the ability to escape local minima. However, overshooting can also cause the model to oscillate and even diverge. On the other hand, small learning rates are useful to achieve a stable convergence, but the process is slower and may get stuck in local minima. We choose to decrease the learning rate by 10 to try to stabilize the convergence of the RNN and maybe achieve better results in the other two architectures. Due to slower convergence, we have increased the number of epochs to 50. Although the results from the RNN have improved, the other two architectures have not seen any improvement. Improvement and convergence in the RNN model indicate that the RNN was, in fact, not too simple, but the lack of convergence was due to the high learning rate.

Model	Learning Rate	Accuracy	Precision	Recall	Macro-F1 Score
RNN	0.001	0.79	0.79	0.79	0.79
GRU	0.001	0.83	0.83	0.83	0.83
LSTM	0.001	0.83	0.83	0.83	0.83

4.7 Combinations

Given the lack of improvement in the results with simple approaches, we decide to use a combination of them. In the following, two to three hyperparameters are modified at once. We aim to identify a combination of hyperparameters that may work to surpass the 0.85 plateau in the Macro-F1 score.

4.7.1 Reduced hidden dimensions and more Layers

A model with more layers but reduced hidden dimensions can be a strategic decision. First, the reduction of dimensionality can help mitigate overfitting and encourage to learn generalized representations. However, it is still necessary to be able to learn complex features, which is achieved through multiple layers. These multiple layers can help extract features at different levels of abstraction. However, the results are not encouraging towards this avenue, since they are similar to more simple models. Similarly to the previous sections, the GRU and LSTM models have a clear advantage over simple RNN. The RNN model is unable to converge successfully, which may still be due to the larger learning rate.

Model	Hidden Dimensions	Number of Recurrent Layers	Accuracy	Precision	Recall	Macro-F1 Score
RNN	128	2	0.47	0.54	0.47	0.46
RNN	64	4	0.59	0.59	0.59	0.56
GRU	128	2	0.84	0.85	0.84	0.84
GRU	64	4	0.83	0.83	0.83	0.83
LSTM	128	2	0.84	0.84	0.84	0.84
LSTM	64	4	0.83	0.83	0.83	0.83

4.7.2 Bidirectionality and reduced learning rate

As previously mentioned, the addition of directionality to an RNN enhances its ability to understand context by processing data forward and backward. Bidirectional RNN doubles the contextual information available to the model, which may lead to improved performance. However, this increase in complexity may result in a more difficult training of the model. Therefore, we have chosen to decrease the learning rate to improve the model's ability to converge to a precise solution. However, the results indicate that the performance of the model is actually degraded in the case of BiGRU and BiLSTM. This may be caused by the model getting stuck on local optima due to the reduced learning rate and the increased noise introduced by the bidirectionality.

Model	Learning Rate	Accuracy	Precision	Recall	Macro-F1 Score
BiRNN	0.001	0.72	0.72	0.72	0.72
BiGRU	0.001	0.78	0.78	0.78	0.78
BiLSTM	0.001	0.81	0.81	0.81	0.81

Model	Number of recurrent layers	Learning Rate	Accuracy	Precision	Recall	Macro-F1 Score
BiRNN	2	0.001	0.79	0.79	0.79	0.79
BiGRU	2	0.001	0.83	0.83	0.83	0.83
BiLSTM	2	0.001	0.83	0.83	0.83	0.82

4.7.3 Bidirectionality, higher recurrent layers and reduced learning rate

Finally, we try one last combination to combine the three different approaches. First, we allow bidirectionality to deepen the understanding of the context. Then we increase the number of recurrent layers to further handle this increase in context correctly. Then we decrease the learning rate to handle the noise created by the increasing amount of context. Once again, the RNN model seem to improved, but the GRU and LSTM models remain with similar results.

5 Evaluation

We have provided script for the evaluation process and have saved the best Torch model into a binary file. Following the provided instructions, we've developed the eval_on_test.py script, designed to accept test data and model arguments. In this script, we've incorporated the class of the best model along with the necessary implementations of CustomDataset and collator functions. Moreover, to ensure consistency, we've applied the same text pre-processing strategy employed during the training phase. This ensures that the evaluation process aligns with the preprocessing methods used for model training. For comprehensive evaluation metrics, we've utilized Scikit-learn's classification_report function. This facilitates the generation of a standard multi-class classification report, detailing precision, recall, and F1 scores for individual classes, as well as an aggregate evaluation for the entire model.

The best model has been securely saved at: **"/fp/projects01/ec30/sushant/best_model.bin"**. This comprehensive evaluation approach ensures a thorough assessment of model performance, providing insights into its efficacy across various classes and overall effectiveness.

Link to Github Repository