THE UNIVERSITY OF SYDNEY

QBUS6850 Machine Learning for Business (2024S2)

Group Project

Due date: Monday 28 October 2024

Group Number: Group 09

Group Members:

530179737

530137070

530213754

Duty Declaration

Student Number	Task Conducted
530179737	Task A, B, C, D
530137070	Task A, B, C, D
530213754	Task A, B, C, D

Abstract

The aim of this project is to build a sentiment classification model based on a news dataset to accurately categorize news text into five categories (Business, Entertainment, Politics, Sports and Technology). We used traditional machine learning, deep learning, and Transformer models to explore the performance of different approaches in sentiment analysis. The text data was transformed into numerical representations through feature engineering suitable for the models, and hyperparametric tuning was performed for each model. The results show that the DistilBERT model performs best in terms of accuracy and weighted F1 scores, with significant text context understanding. This study provides a multi-model comparison for the task of sentiment analysis, revealing the superiority of the Transformer model.

1.0 Introduction

Sentiment analysis is widely applied in areas like business analytics, market research, and public opinion monitoring, helping companies better understand user feedback and public sentiment. This project provides a set of news datasets including five categories of news: business, entertainment, politics, sports and technology, and the goal is to predict the sentiment tendency of these news by building a classification model. In addition to the classification task, the project also involves feature engineering and numerical representation of text, including traditional and deep learning text embeddings, as well as Transformer-based embeddings, to improve the related model performance in sentiment analysis.

2.0 Literature review

2.1 Sentiment analysis in real life

Sentiment analysis (SA) is a fundamental technique in the study of Natural Language Processing (NLP) which is used to identify and classify the underlying expression from text information (Valarmathi et al., 2024). In this digital era, customers and users can be more convent in sharing their opinions and feelings, which is crucial for companies to keep tracking of what people feels and make improvements in quick reaction (Valarmathi et al., 2024). In this case, based on sufficient sentiment data, current machine learning (ML) and deep learning (DL) algorithms can be applied to extract and analyze such information.

2.2 Word embedding methods

Since ML algorithms cannot work with raw text directly, converting textual information into numerical data is crucial and several methods can be taken. As the most basic method, by counting the word frequency and constructing the document-term matrix, the Bag of Words (BoW) is easier to use but assuming words are independent and non-contextual relationships (Hauschild & Eskridge, 2024). Unlike BoW, the TF-IDF method adds weights to reduce the common words' influence and highlights rare words, since rare words may also carry important information (Silge & Robinson, 2022, as cited in Hauschild & Eskridge, 2024).

Compared to the previous two methods, which may have a higher sparsity and non-contextual assumption, Word2Vec, which was developed by Mikolov et al. (2013), is designed to represent words as continuous low-dimensional vectors. This may transfer the sparse representation to the dense, which can better capture semantic information based on the local context. Compared to Word2Vec, the Global Vectors for Word Representation (GloVe), which was developed by Pennington et al. (2014), focuses more on capturing the global co-occurrence relationship of words in the entire corpus rather than local contextual. However, these two approaches have the drawback of using a single vector per word. It is not practical to use one representation when encountering different meanings of the same word (Sun & Platoš, 2023). To address these limitations, Bidirectional Encoder Representations from Transformers (BERT) introduced the Transformer architecture, which processes the entire sequence at once (Devlin et al., 2018). It creates dynamic word embeddings using bidirectional context information so that polysemous words may have different representations depending on the context.

2.3 Traditional machine learning (ML) algorithms

There are many types of traditional ML algorithms. Anderson et al. (2024) used the dataset containing 38576 tweets, where all labels are equally weighted, to train and validate various traditional ML algorithms. Although Logistic Regression assumes linear relationships in the text, it achieved 96.75% accuracy, second only to random forest among the traditional ML models, including SVM, decision tree and Naive Bayes. This kind of algorithm has a significant advantage of training time efficiency, which only needs one second to train with high accuracy. Compared to Logistic regression, which assumes linear relationships in text, tree models including Random Forest (RF) can help detect the variable interaction and cope with non-linear relationships (Hassan et al., 2022). An improved random forest algorithm (IRFTC), proposed by Jalal et al. (2022), can simultaneously apply feature sorting and determine the optimal number of trees. IRFTC outperforms other models like Logistic Regression and standard RF regarding classification accuracy, particularly in feature selection and tree optimization.

Another model that can be applied is the boosting model. Boosting, including Gradient Boosting (GB), is one category of ensemble technique that is considered more advanced and powerful than another type, bagging, like RF (Jun, 2021). One benefit of the GB approach is the bias reduction which uses previous models' residuals to train sequent models (Petropoulos & Siakoulis, 2021). One GB algorithm called XGBoost is proposed by Chen & Guestrin (2016). Its strength is derived from scalability, which is due to the application of weighted quantile sketch procedure for handling instance weights in approximate tree learning with parallel and distributed computing. This can maximize the prediction performance and processing speed. Sawarn et al. (2020) have done a competitive analysis on RF and XGBoost based on Amazon Fine Food Reviews data which contains 568,454 reviews. XGBoost had better performance than RF. And the XGBoost with TF-IDF featurization performs the best.

2.4 Deep learning (DL) algorithms in sentiment analysis

Websites offer vast amounts of unstructured information, including numerous opinions and reviews, thus DL is well-suited to cope with such information and recognize sentiment in large datasets (Suhartono et al., 2023). Feedforward Neural Network (FNN) might be a cornerstone of modern data analysis due to the ability to learn complex patterns from large datasets and through backpropagation (a process of adjusting weights based on residuals) (Bellary et al., 2024). Different from FNN, inputs and outputs are independent of one another and only knowing the previous state can determine the subsequent state, one network called Recurrent

Neural Network (RNN) has a hidden state to remember information from the prior state (Prabakaran et al., 2023). This may be the reason that is commonly applied in SA. One variant of RNN called Long Short-Term Memory (LSTM) is developed to address the limitation of RNN, which is unable to capture sequential data's long-term dependencies. LSTM introduces three gates to store, update and discard information, which enhances the ability to model complex temporal relationships (Wu et al., 2024). According to Prabakaran et al. (2023), the validation accuracy of LSTM outperformed standard RNN based on a Reddit sentiment dataset.

Currently, large pre-trained language models based on the Transformer architecture are dominating other traditional approaches in SA. By using larger pre-trained data during the pre-training phase, models may have better language understanding (Xiao et al., 2024). The Transformer was proposed by Vaswani et al. (2017), which is entirely based on attention mechanisms. This can model relationships regardless of their distance in input or output sequences. One drawback is it requires huge data (Mao et al., 2024). BERT (Devlin et al., 2018) is one of the pre-trained models, used in classification tasks and achieved better performance.

2.5 Evaluation metrics

Mao et al. (2024) reviewed several evaluation metrics to evaluate performance in SA models and proposed that the main evaluation indicators are accuracy, F1 score, precision, and recall.

2.6. Literature gap and research objectives

After reviewing the literature, much of the existing work focuses on an academic perspective, with limited attention given to real business applications. Most studies research in model design and performance evaluation are academic, which may disconnect from real-world business scenarios. These academic discussions typically emphasize metrics such as accuracy but less considering the deployment and application of real business environments

This report not only evaluates several models, both traditional ML algorithms and DL techniques but also assesses their performance through a business lens to fill that gap.

3.0 The reasons of model choice

Several main types of models were chosen for the project: the traditional machine learning model chosen as Logistic Regression and XGBoost, the deep learning modes FNN, LSTM, RNN, and the Transformer-based DISTRIBERT model. These models represent a wide range of applications from classical to modern approaches, and the advantages as well as the reasons for choosing these models are explained in detail below. FNN and Logistic Regression act as benchmarks to compare the lifting effect of other complex models.

3.1 Traditional ML model: Why XGBoost is preferred over RF

3.1.1 Ability to handle text features

Sentiment analysis is a classic text classification task. In this project, text is converted to numerical features through feature engineering techniques like TF-IDF or Bag of Words. XGBoost outperforms decision trees in handling sparse features (Ming, 2020).

News articles often contain sparse data, and XGBoost efficiently handles this by automatically skipping computations for missing values and utilizing a block-based structure. This is particularly suited to high-dimensional, sparse-feature tasks in our project, where features are numerous, but individual values are often sparse. XGBoost's weighted quantile sketching algorithm efficiently manages non-zero entries in the feature matrix (Ashtari, 2024). Although Random Forest can manage high-dimensional data, its adaptability to sparse features is limited. In high-dimensional sparse datasets, Random Forest's tree splits may be more random, which will lead to lower stability and performance. So Random Forest is less suitable for our project.

3.1.2 Efficiency and resource utilization

XGBoost supports multithreading and parallel processing, making it faster for processing large datasets, which is well fitted in this project. While Random Forest takes longer to train on large datasets because each tree is generated independently. XGBoost is also more optimized in memory management, particularly when handling large-scale text datasets in sentiment analysis, as it can use memory resources more effectively.

3.1.3 Advantage in parameter tuning

XGBoost is more customizable, allowing for fine-tuning of parameters like learning rate, tree depth, and subsample ratio, which can be tailored to the specifics of a sentiment analysis task for improved performance. In contrast, Random Forest offers fewer hyperparameters, limiting its flexibility and precision in adjustments. For a complex sentiment classification task like this one, XGBoost's tuning capabilities offer a distinct advantage.

3.2 Deep learning: Neural Network model selection and architectures

3.2.1 FNN

At the early stages of this project, FNN (Feedforward Neural Network) was chosen as a quick way to explore data and model feasibility. FNNs are efficient in computation, fast to train, and can provide stable performance in high-dimensional data settings. Compared to more complex models, FNNs are also less prone to overfitting (Naz et al., 2021). Additionally, FNNs work well in contexts involving traditional feature engineering. After applying TF-IDF or similar transformations, the FNN effectively learns relationships between features in the project.

The FNN architecture is a standard neural network with an embedding layer, with a first fully connected layer with ReLU activation, and a second fully connected layer. The embedding layer output undergoes average pooling to convert inputs of different sequence lengths into a fixed global representation, which allows key sequence features to be captured within a fixed dimension and reduces the risk of data loss. A dropout layer is also included to randomly drop neurons, which minimizes overfitting and enhances the model's robustness.

3.2.2 LSTM and Vanilla RNN

For this project, Vanilla RNN and LSTM were chosen as primary models for sentiment analysis. The Vanilla RNN is simpler in structure and generally suited for shorter sequence tasks, such as classifying short texts in sentiment analysis. RNNs retain information through state transfer but struggle with long-sequence data due to issues like gradient vanishing and exploding, making it challenging for models to learn long-range dependencies. This limitation affects the ability to capture and utilize contextual information from previous inputs, which is essential in sentiment analysis, where emotional context often relies on prior content.

To address its limitations with long-sequence retention, we applied layer normalization to embedding vectors within the RNN architecture. Given the diverse sentence lengths and content in this project, layer normalization ensures consistent scaling across different layers, mitigating issues like gradient vanishing and exploding, thus aiding in faster convergence. It also enables the model to balance information across all sequence positions, capturing comprehensive input sequence information. This approach avoids the model's bias toward outputs at specific time steps, enhancing its capacity to represent news sequences. Additionally, the RNN's hidden state is reinitialized for each forward pass using Xavier initialization, ensuring the initial state does not affect learning and improving training effectiveness.

For longer input sequences, however, earlier input information may gradually dissipate during propagation, weakening the model's comprehension of later inputs (Li et al., 2021).

LSTM, an enhanced version of RNN, incorporates specialized units with an input gate (deciding the extent of new information to add to the cell state), a forget gate (controlling the removal of old information from the cell state), and an output gate (regulating the amount of information output to the next layer). The gating mechanism allows LSTM to store, discard, and transfer information effectively, enabling better performance on long-sequence data.

In sentiment analysis, LSTM identifies shifts in sentiment and maintains contextual coherence. Li et al. (2021) also noted that LSTMs outperform other models in tasks involving long-term dependencies due to their superior memory management capabilities.

Unlike RNNs, we applied max pooling along the time dimension after LSTM outputs, extracting the most significant features across the sequence. This fixed output dimension enhances classification efficiency for identifying distinct features in news articles while minimizing the impact of sequence length variability.

Despite LSTM's advantages, gradient vanishing or exploding remains a risk as network depth increases. To address this, we added ResNet to the LSTM structure. This method allows certain layers to skip directly over others, preserving original input information, thereby resolving gradient issues and training difficulties associated with increased depth. Residual connections also encourage better information flow between features across different news categories, enabling LSTM to capture both similarities and differences among categories more effectively.

Unlike traditional sentiment analysis tasks, we opted not to use Bidirectional RNN for this project. The primary goal of sentiment analysis here is to understand the text information to classify news categories, where the central information often revolves around events, facts, and viewpoints. The project does not require the Bidirectional RNN's function to understand Information both before and after the word helps to judge tone and emotion more accurately.

LSTM is sufficient for capturing the sequence's order; adding a Bidirectional RNN could introduce redundant information and reduce model interpretability.

3.3 DistilBERT based on FNN

DistilBERT is a lightweight Transformer model derived from BERT through a distillation process (Sanh et al., 2020). It consists of 6 Transformer blocks, each containing a Multi-Head Self-Attention layer. This layer allows the model to simultaneously focus on various positions in a sequence, enhancing its contextual understanding. DistilBERT includes 12 attention heads, where each head captures different positional information across the sequence. Each token's representation relies not only on its information but also considers the influence of other words within the sentence. Each attention head has a dimensionality of 64. After each attention layer, a Feedforward neural network performs nonlinear transformations on the output from the attention layer, typically through two linear transformations with an activation function (e.g., ReLU). Like LSTM, each attention and feed-forward layer in DistilBERT incorporates residual connections, which help gradient flow and mitigate vanishing gradient issues during training.

The attention mechanisms remain a core component in DistilBERT. The self-attention mechanism allows the model to assign varying weights to each word in the input sequence, effectively capturing context-dependent relationships.

3.3.1 Reasons for selecting DistilBERT based on FNN

DistilBERT was pre-trained on a large text corpus, allowing it to learn rich language representations. The extensive vocabulary coverage enables the model to recognize and process various linguistic nuances, making it particularly effective for tasks like sentiment analysis that require capturing subtle contextual differences. The choice of an FNN-based DistilBERT model reflects its contextual comprehension needs. While RNN and LSTM may encounter gradient vanishing or exploding issues with long sequences, the Transformer architecture captures information across any sequence position more effectively via the self-attention mechanism. Additionally, the substantial textual data in this project aligns well with DistilBERT's extensive vocabulary, removing the need for RNN or LSTM's sequential processing.

4.0 Experiment detail and result analysis

4.1 Data preprocessing

In the data preprocessing stage, we first deleted duplicate values. Duplicate values will cause the model to learn certain features and ignore other important features, resulting in overfitting. However, it cannot handle highly similar data (since 'drop duplicate' can only identify variables that are exactly the same) at present, we will do further cleaning in feature engineering. Second, we used 'spacy' to do the basic word tokenization, stopping word, lowercase conversion and lemma. This can clean and standardize text data and help the model learn more useful features.

In addition, we also deleted the title variable in the dataset, which means that our model will use the content of the text as input. There are two reasons: 1. The title is usually highly summarized and concise. It is intended to arouse the reader's interest rather than fully describe the entire news. The title cannot fully represent the key point of the news. In contrast, the text contains more comprehensive information and topics. 2. News titles sometimes appear ambiguous to attract readers' attention, which will increase the Type One Error, while the text usually eliminates these ambiguities.

4.2 EDA

In EDA, we mainly made word clouds and checked the problem of category imbalance. As shown in Appendix Figure 1, apart from the most common 'say', each category has corresponding core words. These core words well reflect the corresponding category. For example, business mainly revolves around company and firm; politics mainly revolves around company and firm. Each field has its unique focus. These core vocabularies can effectively reveal the topics of each category and will play a very important role in the model.

Second, as shown in Appendix Figure 2, although there are differences in the amount of news in each category, the overall difference is not particularly significant. The amount of news in all categories is within an acceptable range and the data distribution is quite reasonable.

4.3 Feature engineering

To delete highly similar news, we further use 'cosine similarity' to quantify the similarity between them. First, each token is converted into a numerical vector through 'Word2Vec'. Then, cosine similarity is used to calculate the distance between vectors. We regard similar documents as duplicates and delete them. The purpose of this was to ensure that part of the data (i.e., the test set) had not been learned by the model. The model was used to make predictions on the test set to simulate the model's ability to predict unknown data in real life. The performance of the test set represents the generalization ability of the model. Based on this, we divided the training set into a training set and a validation set again, to select our optimal model and tune the hyperparameters through the validation set. We divided the data set into a ratio of 0.49:0.21:0.3 for the training validation test set.

Secondly, we used the word representation methods of Bag of words, bi-grams, tri-grams, TF-IDF and Word2Vec to convert the data from text variables into vector forms that can be learned by the model. We used Logistic Regression as our base model and tried the performance of the above methods on the model to select the optimal word representation method. Appendix Table 1 shows the performance of Logistic Regression using various word representation methods. As shown in the figure, the logistic model under TF-IDF achieves the highest result. TF-IDF places more emphasis on words that appear frequently in a certain type of document but less frequently in other types of documents. This means that in this text classification task, the frequency of certain words is crucial for class distinction. TF-IDF can better capture the distinguishing ability of these words, thereby improving the classification effect.

Vector data converted by word representation methods such as TF-IDF usually faces the problem of high-dimensional sparsity. High-dimensional sparse data will significantly increase the amount of computation and make it difficult for the model to learn useful features. One solution is dimension reduction. In this project, we chose Singular Value Decomposition (SVD) as our dimension reduction method. SVD is widely used in text analysis, especially when dealing with sparse high-dimensional data (Dumais, 2004). It is worth noting that SVD essentially compresses high-dimensional features into a low-dimensional space, which will linearly combine the original features. The model is actually training with new features. This will affect the interpretability of the model. Variable importance and SHAP value will no longer work. However, in this project, our task is to predict the topic of the news. Unlike fields such as medical or financial which require high interpretability and transparency, the core goal of news topic prediction is to improve the prediction accuracy of the model. Therefore, we choose to sacrifice a certain degree of interpretability in this task in exchange for higher prediction performance. After SVD dimension reduction, the two features may have large numerical differences. To avoid the poor performance of the model, we normalized the data to ensure that the mean of each feature is 0 and the variance is 1, to further improve the stability of the model.

Finally, since the vocabulary all from the training set, there will inevitably be words that have not appeared in the training set during validation. To address this situation, we added an 'unk' column to ensure that when words that do not appear in the train dataset appear during validation and testing, these words will be automatically added to the 'unk' column.

For neural networks, since the first layer of our network is designed as an embedding layer, it can automatically reduce the dimension of the data and convert it into a vector format, and standardize it through 'LayerNorm', so there is no additional feature engineering step.

4.4 Hyperparameter optimization

We optimized the hyperparameters of XGBoost and deep learning models by using 'Optuna'. Optuna is based on the Tree-structured Parzen Estimator (TPE) algorithm, which can dynamically adjust the search range of hyperparameters according to the results of each experiment, making the search more targeted.

Appendix Table 2 shows the hyperparameters selection of FNN. In this project, we did not tune FNN because FNN does not consider the sequence nature of the data. In contrast, XGBoost, RNN and LSTM are more suitable for this project, so we put more effort into these three models.

First, we conducted hyperparameter research on XGBoost, the result is shown in Appendix Table 3. After obtaining the hyperparameters, we recorded the train and validation loss of the training process. The results are shown in Appendix Figure 3. The model performs poorly, there may be underfitting. It did not learn some features. Therefore, we increased the depth of the tree, reduced the min child weight and lambda and limited its regularization ability. After updating the hyperparameter range, we performed a second Optuna optimization and obtained new results, as shown in Appendix Figure 4. Now the model has learned enough features, but there are signs of overfitting. Despite this, the performance of the model is basically satisfied and we decided to spend more time on Deep learning.

For RNN and LSTM, we performed multiple optimizations and obtained the results. As shown in Appendix Table 4. For RNN, we set the commonly used hyperparameter range and used Optuna for optimization and recorded the training loss and validation loss of each epoch. As shown in Appendix Figure 5, the model showed strong overfitting and the validation loss fluctuated greatly. Hence, we reduced the range of learning rate, weight decay and increased the dropout rate. After the update, we used Optuna again to find hyperparameters and train. The results are shown in Appendix Figure 6. The model performance has been improved but still shows overfitting. We think this is within the acceptable range, so we keep these results.

In the initial tuning of the LSTM model, we also set the commonly used hyperparameter range. Based on the optimal hyperparameters found by Optuna, we trained the model and the results are shown in Appendix Table 5. However, based on the train and validation loss represented in Appendix Figure 7, the model training triggered the early stopping after a small number of epochs. By observing the loss curve, we can draw the following conclusions: 1. The model faces a serious overfitting problem. It cannot learn complex data relationships. 2. The fluctuation of the validation loss is very serious. In view of the above situation, we increased the number of neurons in the embedding layer and the hidden layer to enhance the learning ability of the model and increased the range of the dropout rate and weight decay to improve the model regularization to prevent overfitting. The learning rate was reduced to ensure a more stable model iteration process. In addition, we found that in cases with better performance, the number of LSTM layers is mostly 1 layer, which may imply that our data is relatively simple, one layer of LSTM is sufficient, so we also changed the range of LSTM layers to [1, 2] layers. After optimizing again with Optuna using the updated hyperparameter range, we trained a new model. The training process is shown in Appendix Figure 8. Although the new model still has some overfitting and validation loss fluctuation problems, these phenomena have been significantly alleviated and are within an acceptable range. Most importantly, the overall performance of the model has been significantly improved compared to the initial tuning.

In this project, we did not perform in-depth hyperparameter tuning for BERT. Considering that BERT, as a pre-trained Transformer model, has shown very strong performance in NLP, we think it is not necessary to do too much hyperparameter tuning. Therefore, we only selected a basic hyperparameter range and performed a simple round of Optuna optimization and model training. The optimization results of BERT are as shown in Appendix Table 6.

4.5 Model selection and result analysis

In model selection and evaluation, we will use weighted F1 score and accuracy as our evaluation criteria, and mainly judge based on weighted F1. The main reasons are as follows: 1. F1 combines precision and recall, it can comprehensively evaluate the performance of the model in each category. Besides, Weighted F1 considers the problem of class imbalance, which can assign weights according to the number of samples in each category, so as to more accurately reflect the overall model performance and avoid the impact of minority classes on the evaluation results being amplified or ignored. 2. As a measure of model performance, accuracy is simple and easy to explain, especially suitable for showing the overall correct prediction ratio. Although the data has a certain class imbalance, accuracy can also well measure the correctness of the model in the overall major categories, which is suitable for evaluating the overall reliability of the final prediction results. The following Table 1 shows our model selection results and comparisons.

Type	Model	Weighted F1	Accuracy
Machine learning	Logistic	0.963	0.963
	XGBoost	0.930	0.930
Deep learning	FNN	0.952	0.952
	RNN	0.953	0.952
	LSTM	0.960	0.960
Transformer	BERT	0.987	0.987

Table 1. *Model selection results of all models*

BERT performs best among all models, it leads other models in all loss metrics. The best performing model in Deep Learning is LSTM, which shows that it can better capture sequence information and has an advantage in this task. The best performing model in Machine Learning is Logistic Regression, but since the core goal of the task is to predict unknown data through deep learning models, traditional machine learning models are not the best choice.

Given that the task requires using deep learning models to predict unknown data, I merged the current training and validation datasets to enable the LSTM model to learn more data features. The final evaluation was then conducted on the test dataset to assess the model's performance. The following Table 2 is the model evaluation of LSTM.

Model	Weighted F1	Accuracy
LSTM	0.961	0.962

Table 2. *Model evaluation of Long Short-Term Memory*

After learning more features, the performance of LSTM has been slightly improved, which also shows that the generalization ability of the model has been proven and basically meets the requirements of the task.

5.0 Conclusion

Based on the experimental results, our best models are DistilBERT, LSTM, and Logistic Regression. DistilBERT performs the best, with its pre-trained Transformer architecture capturing contextual information efficiently and outperforming other models in terms of prediction accuracy. This makes DistilBERT particularly suitable for business scenarios, helping organizations make more accurate judgments in sentiment analysis and user feedback.

However, despite DistilBERT's significant advantages in prediction accuracy, there are limitations in terms of its interpretability and computational cost. The complex architecture of DistilBERT makes its prediction mechanism more difficult to understand compared to simple models such as Logistic Regression or LSTM. In application scenarios that require a high degree of interpretability (e.g., explaining the model's decision-making process to stakeholders), Logistic Regression may be more advantageous in intuitively explaining feature significance.

In terms of computational cost, Logistic Regression has the lowest computational overhead and is suitable for fast analysis and simple tasks, while although DistilBERT performs best in terms of accuracy, it requires more computational resources and time.

Therefore, for applications that focus on prediction accuracy and have sufficient computational resources, it is recommended that DistilBERT be preferred as the primary scoring prediction model. In scenarios that require a balance between interpretability and computational efficiency, traditional models such as LSTM or Logistic Regression can be chosen to find the right balance between predictive validity and efficiency.

Reference

Anderson, T., Sarkar, S., & Kelley, R. (2024). Analyzing public sentiment on sustainability: A comprehensive review and application of sentiment analysis techniques. *Natural Language Processing Journal*, *8*, 100097. https://doi.org/10.1016/j.nlp.2024.100097

Ashtari, H. (2024). *XGBoost vs. Random Forest vs. Gradient Boosting: Differences | Spiceworks*. Spiceworks. https://www.spiceworks.com/tech/artificial-intelligence/articles/xgboost-vs-random-forest-vs-gradient-boosting/

Bellary, S., Bala, P. K., & Chakraborty, S. (2024). Exploring cognitive-behavioral drivers impacting consumer continuance intention of fitness apps using a hybrid approach of text mining, SEM, and ANN. *Journal of Retailing and Consumer Services*, *81*, 104045. https://doi.org/10.1016/j.jretconser.2024.104045

Chen, T., & Guestrin, C. (2016). XGBoost: a Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, 785–794. https://doi.org/10.1145/2939672.2939785

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional Transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dumais, S. T. (2004). Latent semantic analysis. *Annual Review of Information Science and Technology*, 38(1), 188–230.

Hassan, S. U., Ahamed, J., & Ahmad, K. (2022). Analytics of machine learning-based algorithms for text classification. *Sustainable Operations and Computers*, *3*, 238–248. https://doi.org/10.1016/j.susoc.2022.03.001

Hauschild, J., & Eskridge, K. (2024). Word embedding and classification methods and their effects on fake news detection. *Machine Learning With Applications*, *17*, 100566. https://doi.org/10.1016/j.mlwa.2024.100566

Jalal, N., Mehmood, A., Choi, G. S., & Ashraf, I. (2022). A novel improved random forest for text classification using feature ranking and optimal number of trees. *Journal of King Saud University - Computer and Information Sciences*, *34*(6), 2733–2742. https://doi.org/10.1016/j.jksuci.2022.03.012

Jun, S. (2021). Technology integration and analysis using Boosting and Ensemble. *Journal of Open Innovation Technology Market and Complexity*, *7*(1), 27. https://doi.org/10.3390/joitmc7010027

Li, D., Liu, S., Lyu, Z., Xiang, W., He, W., Liu, F., & Zhang, Z. (2021). Use mean field theory to train a 200-layer vanilla GAN. *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, *33*, 420–426. https://doi.org/10.1109/ictai52525.2021.00068

Mao, Y., Liu, Q., & Zhang, Y. (2024). Sentiment analysis methods, applications, and challenges: A systematic literature review. *Journal of King Saud University - Computer and Information Sciences*, 102048. https://doi.org/10.1016/j.jksuci.2024.102048

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *arXiv*. https://doi.org/10.48550/arxiv.1310.4546

Ming, J. (2020). *Sentiment Classification using XGBoost - Artificial Intelligence in Plain English*. Medium; Artificial Intelligence in Plain English. https://ai.plainenglish.io/sentiment-classification-using-xgboost-7abdaf4771f9

Naz, H., Ahuja, S., Kumar, D., & Rishu. (2021). DT-FNN based effective hybrid classification scheme for twitter sentiment analysis. *Multimedia Tools and Applications*, *80*. https://doi.org/10.1007/s11042-020-10190-3

Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. https://doi.org/10.3115/v1/d14-1162

Petropoulos, A., & Siakoulis, V. (2021). Can central bank speeches predict financial market turbulence? Evidence from an adaptive NLP sentiment index analysis using XGBoost machine learning technique. *Central Bank Review*, *21*(4), 141–153. https://doi.org/10.1016/j.cbrev.2021.12.002

Prabakaran, N., Anbarasi, A., Deepa, N., & Pandiaraja, P. (2023). Enabling an On-demand Access to Community Sentiments using LSTM RNNs Web Service Architecture. *Procedia Computer Science*, *230*, 584–597. https://doi.org/10.1016/j.procs.2023.12.114

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2020). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. https://arxiv.org/pdf/1910.01108

Sawarn, A., Ankit, & Gupta, M. (2020). Comparative Analysis of Bagging and Boosting Algorithms for Sentiment Analysis. *Procedia Computer Science*, *173*, 210–215. https://doi.org/10.1016/j.procs.2020.06.025

Suhartono, D., Purwandari, K., Jeremy, N. H., Philip, S., Arisaputra, P., & Parmonangan, I. H. (2023). Deep neural networks and weighted word embeddings for sentiment analysis of drug product reviews. *Procedia Computer Science*, *216*, 664–671.

https://doi.org/10.1016/j.procs.2022.12.182

Sun, Y., & Platoš, J. (2023). A method for constructing word sense embeddings based on word sense induction. *Scientific Reports*, *13*(1). https://doi.org/10.1038/s41598-023-40062-3

Valarmathi, B., Gupta, N. S., Karthick, V., Chellatamilan, T., Santhi, K., & Chalicheemala, D. (2024). Sentiment Analysis of Covid-19 Twitter Data using Deep Learning Algorithm. *Procedia Computer Science*, *235*, 3397– 3407. https://doi.org/10.1016/j.procs.2024.04.320

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *arXiv*. https://doi.org/10.48550/arxiv.1706.03762

Wu, X., Wang, B., & Li, W. (2024). Comparing the accuracy of ANN with transformer models for sentiment analysis of tweets related to COVID-19 Pfizer vaccines. *Chaos Solitons & Fractals*, *185*, 115105. https://doi.org/10.1016/j.chaos.2024.115105

Xiao, Y., Yang, Y., Xu, H., & Li, S. (2024). Empirical insights into the interaction effects of groups at high risk of depression on online social platforms with NLP-based sentiment analysis. *Data and Information Management*, 100080. https://doi.org/10.1016/j.dim.2024.100080

Appendix

Figure 1. *Word Cloud of among categories*

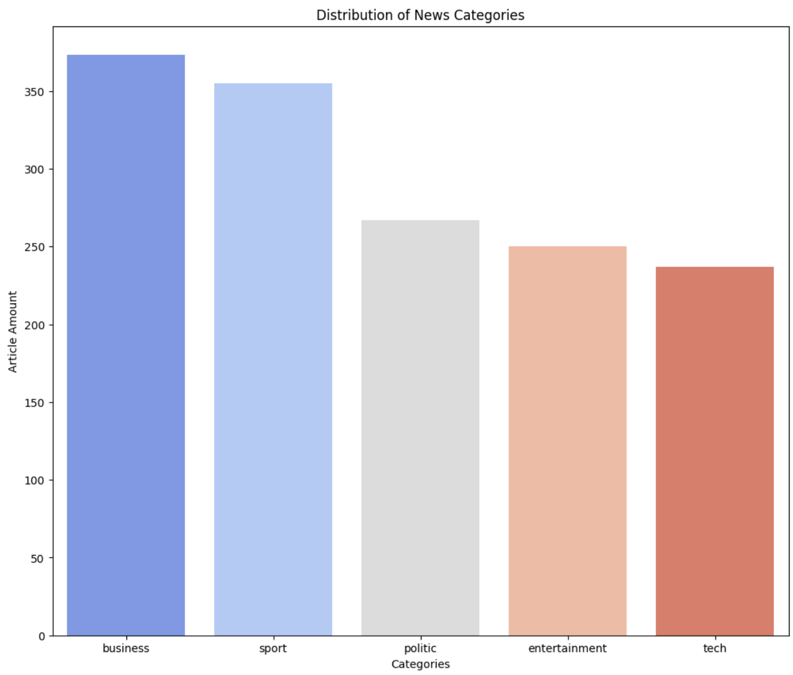

Figure 2. *Distribution of News Categories*

Text representation	Accuracy (Logistic regression)	Accuracy (L1 Logistic)	Accuracy (L2 Logistic)
Word2Vec	0.945	0.952	0.949
Bag of words	0.963	0.945	0.956
Bi-gram	0.850	0.824	0.839
Tri-gram	0.473	0.440	0.477
TF-IDF	0.963	0.956	0.956

Table 1. *Results of various text representation methods on Logistic Regression*

Hyperparameters	Range	Result
Embedding dimension	[16, 32]	25
Hidden dimension	[16, 30]	20
Dropout rate	[0.1, 0.5]	0.251
Learning rate	[1e-4, 1e-2]	0.010

Table 2. *Hyperparameters results of Feedforward Neural Network*

Hyperparameters	First range	Second range	Result
eta	[0.2, 0.4]	[0.2, 0.4]	0.300
num_boost_round	100000	1000	1000
max_depth	[2,5]	[3, 10]	6
subsample	[0.5, 1]	[0.5, 1]	1
colsample_bytree	[0.5, 1]	[0.5, 1]	1
gamma	[0, 10]	[0, 10]	0
min_child_weight	[2, 5]	[0.1, 10]	1
lambda	[2, 5]	[0.1, 10]	1
alpha	[0, 10]	[0, 10]	0

Table 3. *Hyperparameters results of XGBoost*

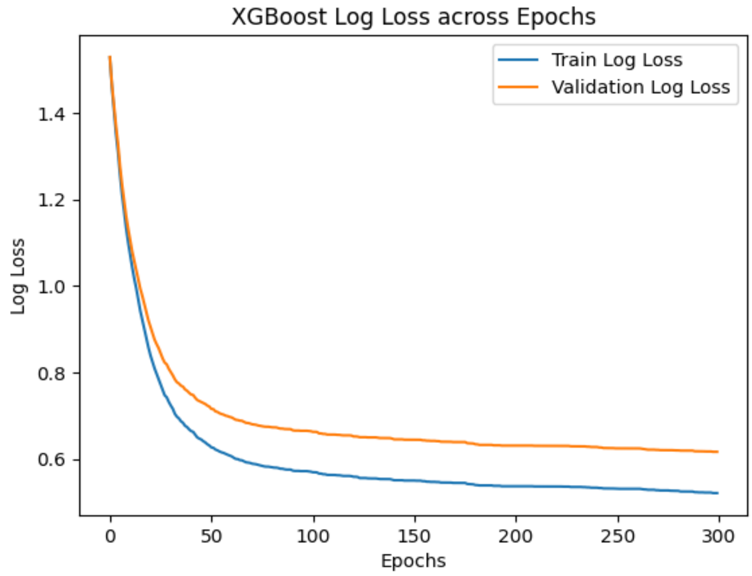

Figure 3. *XGBoost training and validation loss in first tuning*

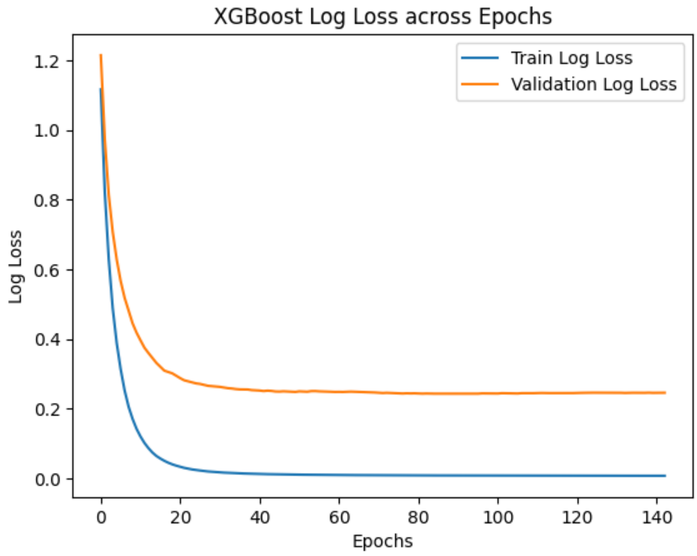

Figure 4. *XGBoost training and validation loss in second tuning*

Hyperparameters	First range	Last range	Result
Embedding dimension	[64, 128]	[64, 128]	72
Hidden dimension	[32, 96]	[32, 96]	94
Dropout rate	[0,1 0.5]	[0.35, 0.7]	0.476
Learning rate	[1e-4, 1e-1]	[1e-4, 1e-2]	0.005
Weight decay	[1e-4, 1e-1]	[1e-4, 1e-2]	0.001
Number of LSTM layer	[1, 3]	[1, 2]	1

Table 4. *Hyperparameters results of Recurrent Neural Network*

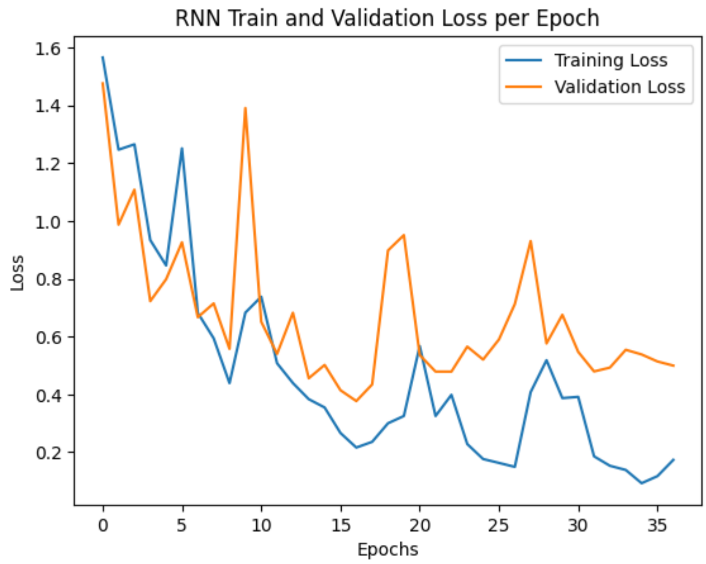

Figure 5. *RNN training and validation loss in first tuning*

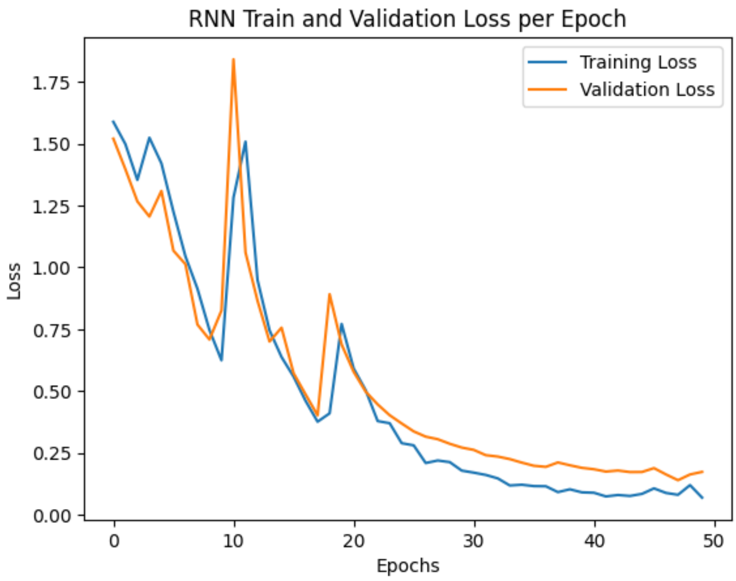

Figure 6. *RNN training and validation loss in second tuning*

Hyperparameter	First range	Last range	Result
Embedding dimension	[64, 128]	[86, 158]	102
Hidden dimension	[32, 96]	[64, 158]	102
Dropout rate	[0,1 0.5]	[0.45, 0.65]	0.538
Learning rate	[1e-4, 1e-2]	[1e-5, 1e-3]	0.001
Weight decay	[1e-4, 1e-1]	[1e-4, 1e-2]	0.007

Fully connected dimension	[64, 128]	[86, 158]	125
Number of LSTM layer	[1, 3]	[1, 2]	1

Table 5. *Hyperparameters results of Long short-term memory*

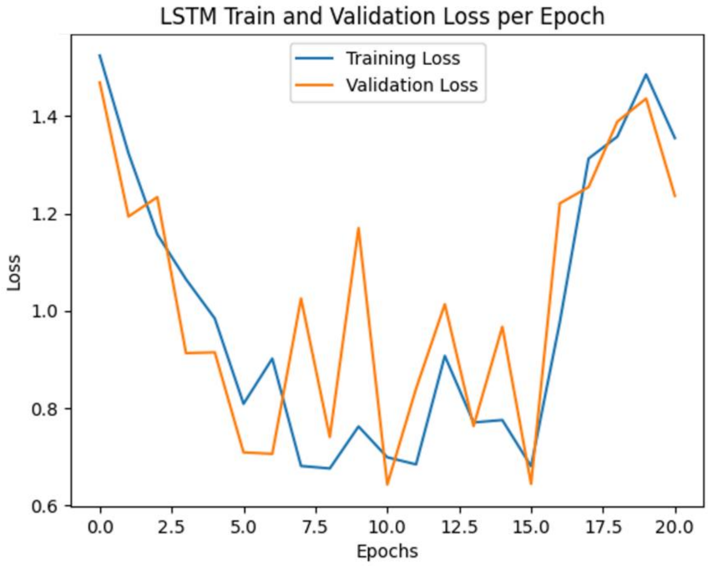

Figure 7. *LSTM training and validation loss in first tuning*

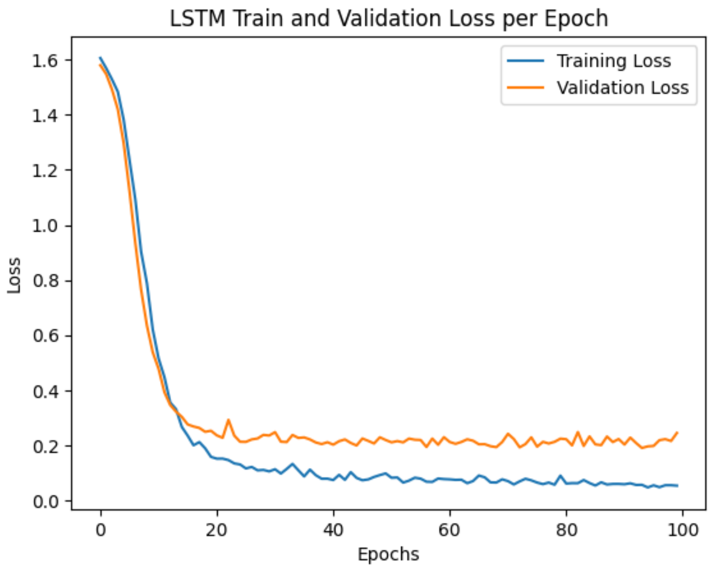

Figure 8. *LSTM training and validation loss in second tuning*

Hyperparameters	Range	Result
hidden_size	[8, 64]	14
dropout_rate	[0.1, 0.4]	0.362
learning_rate	[1e-4, 1e-2]	0.003
weight_decay	[1e-5, 1e-2]	0.003

Table 6. *Hyperparameters results of BERT*

Code:

```
# -*- coding: utf-8 -*-
"""Group09QBUS6850_ML&DL_2024S2

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1_5SSGcfYgVWKecf0kdAznJJyQb5gpFay

# This program is our data preprocessing, EDA, feature engineering, ML and DL except
Competition and BERT. Please run this program with CPU first.
# Then run Competition program with CPU.
# Finally run BERT program with GPU.

# Import data
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install optuna
!pip install shap
!pip install optuna-integration[xgboost]
!pip install torchtext==0.16.0

!python -m spacy download en_core_web_lg

import warnings
warnings.filterwarnings('ignore')

import pickle
import json
import re
import shap
import torch
import spacy
import random
import numpy as np
import pandas as pd
import seaborn as sns
from wordcloud import WordCloud
import matplotlib.pyplot as plt
from collections import Counter
```

```
import optuna
import xgboost as xgb
import optuna.visualization as vis
from sklearn.decomposition import TruncatedSVD
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, Normalizer,
LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, classification_report, log_loss


from gensim.models import Word2Vec
import torch.nn as nn
import torch.optim as optim
from torch import utils
from torchtext.data.utils import get_tokenizer
from torch.utils.data import DataLoader, Dataset
from torchtext.vocab import build_vocab_from_iterator
import collections
from torch.nn.utils.rnn import pad_sequence
import torch.nn.functional as F
from transformers import DistilBertTokenizer, DistilBertModel
from torch.utils.data import DataLoader, TensorDataset


seed_value = 42
random.seed(seed_value)
np.random.seed(42)
torch.manual_seed(42)
torch.cuda.manual_seed(42)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

checkpoint = torch.load('/content/drive/MyDrive/6850/checkpoint.pth')

assert checkpoint.get('device') == 'cpu', "Checkpoint was saved on a CPU, but is being
loaded elsewhere."

data = pd.read_csv('/content/drive/MyDrive/6850/news-dataset.csv',sep='\t')
```

```
"""# 1.0 Data preprocessing"""

data

data.describe()

data.info()

data.nunique()

dup_content = data[data.duplicated(subset='content', keep=False)]
dup_content.head(20)

content_clean = data.drop_duplicates(subset='content')
content_clean.describe()
## We found there remain some duplicate content(ie content with higher similarity but
not identify as duplicate)

title_dup = content_clean[content_clean.duplicated(subset='title', keep=False)]
title_dup.head(20)

nlp = spacy.load('en_core_web_lg') #create an object

def tokenizer(text):
    text = re.sub(r'#(\S+)', 'xxhashtag ' + r'\1', text) # hashtag

    text = re.sub(r'\s{2,}', ' ', text)

    doc = nlp(text)

    tokens = []

    for token in doc:

        word = token.lemma_.lower()

        if not token.is_stop:

            if word == '!':
                tokens.append('!')

            elif (not token.is_punct) and word != '':
                tokens.append(word)
```

```
    return tokens

corpus = content_clean.copy()
for i in range(len(corpus.columns)):
     corpus.iloc[:,i] = corpus.iloc[:,i].apply(tokenizer)

content_clean = corpus.copy()

corpus

"""# 2.0 EDA"""

content_clean['category'] = content_clean['category'].apply(lambda x: ', '.join(x) if
isinstance(x, list) else x)

content_clean['category'].unique()

def generate_wordcloud(text, category):
    wordcloud = WordCloud(width=1000, height=500, background_color='white').generate('
'.join(text))
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(f"Word Cloud for {category}")
    plt.axis('off')
    plt.show()

categories = content_clean['category'].unique()

for category in categories:
    category_text = content_clean[content_clean['category'] ==
category]['content'].str.join(' ')
    generate_wordcloud(category_text, category)

overall_text = content_clean['content'].str.join(' ').tolist()
generate_wordcloud(overall_text, 'Overall')

content_clean.isnull().sum()

cate_count = content_clean['category'].value_counts()

print(cate_count)

# Caculate category percentage
category_counts = content_clean['category'].value_counts()
```

```
category_percentage = (category_counts / category_counts.sum())

category_percentage

plt.figure(figsize=(12, 10))
sns.barplot(x=category_counts.index, y=category_counts.values, palette='coolwarm')
plt.title('Distribution of News Categories')
plt.xlabel('Categories')
plt.ylabel('Article Amount')
plt.xticks
plt.show()

content_clean['title_length'] = content_clean['title'].str.len()
content_clean['content_length'] = content_clean['content'].str.len()

stats = content_clean[['title_length', 'content_length']].agg(['min', 'max', 'mean'])

print(stats)

"""# 3.0 Feature engineering

## 3.1 Word Embedding
"""

sentences = corpus['content'].tolist()

word2vec_model = Word2Vec(sentences, vector_size=300, window=9, min_count=2,
workers=1, negative = 10, alpha = 0.025, min_alpha = 0.0001, sg =1,seed = 42)
# After identify remaining duplicate content, we choose to use cosine similarity based
on the distence between contents first we use word embedding to convert token into
words vector.

def sentence_to_vector(sentence, model):
   word_vectors = [model.wv[word] for word in sentence if word in model.wv]
   if len(word_vectors) > 0:
      return np.mean(word_vectors, axis=0)
   else:
      return np.zeros(model.vector_size)

corpus_vectors = np.array([sentence_to_vector(sentence, word2vec_model) for sentence
in sentences])

print("Corpus shape: ", corpus_vectors.shape)
```

```
# We define a function to convert into vector, this may help to calculate the
similarity then.

"""### **3.1.1 Cosine similarity**"""

similarity_matrix = cosine_similarity(corpus_vectors)

print("The minimum:", np.min(similarity_matrix))
print("The maximum:", np.max(similarity_matrix))
print("The mean:", np.mean(similarity_matrix))

threshold = 0.994
to_remove = set()

for i in range(len(similarity_matrix)):
    for j in range(i + 1, len(similarity_matrix)):
        if similarity_matrix[i][j] > threshold:
            to_remove.add(j)

to_remove_list = list(to_remove)

to_remove_array = np.array(to_remove_list)
to_remove_array.shape

corpus = corpus.drop(corpus.index[to_remove_array]) # we drop the content which has
high similarity

corpus.info()

corpus.head(30)

"""## 3.2 Train test split"""

index_train, index_test = train_test_split(corpus.index,stratify=corpus['category'],
train_size=0.7, random_state = 42)

train = corpus.loc[index_train, :].copy()
test = corpus.loc[index_test, :].copy()

index_train1, index_valid1 = train_test_split(train.index, stratify =
train['category'], train_size=0.7, random_state=42)

sub_train = train.loc[index_train1, :].copy()
sub_valid = train.loc[index_valid1, :].copy()
```

```
y_train = sub_train['category'].copy()
y_valid = sub_valid['category'].copy()
y_test = test['category'].copy()

category_distribution = y_train.value_counts()
category_distribution

sentences_train = sub_train['content'].tolist()
sentences_valid = sub_valid['content'].tolist()

word2vec_train = Word2Vec(sentences=sentences_train, vector_size=300, window=9,
min_count=2, workers=4, negative = 10, alpha = 0.025, min_alpha = 0.0001, sg =1,seed =
42)
vocab = set(word2vec_train.wv.index_to_key)
vocab.add("<UNK>")

processed_valid = []
for sentence in sentences_valid: # the words not seen in train we give the <unk>
    processed_sentence = [word if word in vocab else "<UNK>" for word in sentence]
    processed_valid.append(processed_sentence)

def sentence_to_vector(sentence, model):
    word_vectors = []
    for word in sentence:
        if word in model.wv:
            word_vectors.append(model.wv[word])
        else:
            word_vectors.append(np.zeros(model.vector_size))

    if word_vectors:
        return np.mean(word_vectors, axis=0)
    else:
        return np.zeros(model.vector_size)

trainvector_word2vec = np.array([sentence_to_vector(sentence, word2vec_train) for
sentence in sentences_train])
validvector_word2vec = np.array([sentence_to_vector(sentence, word2vec_train) for
sentence in processed_valid])


print("Train shape: ", trainvector_word2vec.shape)
print("Valid shape: ", validvector_word2vec.shape)
```

```
trainvector_word2vec = Normalizer().fit_transform(trainvector_word2vec) #Normalization
(mean = 0, st dev = 1)
validvector_word2vec = Normalizer().transform(validvector_word2vec)

"""## 3.3 Encoding"""

# The tokens are reconnected with spaces to ensure that the subsequent word
representation method can recognize the complete text data.
sub_train['content'] = sub_train['content'].apply(lambda x: ' '.join(x))
sub_valid['content'] = sub_valid['content'].apply(lambda x: ' '.join(x))
test['content'] = test['content'].apply(lambda x: ' '.join(x))

"""### 3.3.1 Bag of Words"""

vectorizer = CountVectorizer()

x_bow_train = vectorizer.fit_transform(sub_train['content'])
x_bow_valid = vectorizer.transform(sub_valid['content'])

x_bow_train = Normalizer().fit_transform(x_bow_train) #Normalization (mean = 0, st dev
= 1)
x_bow_valid = Normalizer().transform(x_bow_valid)

feature_names = vectorizer.get_feature_names_out()

bagofwords_train = pd.DataFrame(x_bow_train.toarray(), columns=feature_names)
bagofwords_valid = pd.DataFrame(x_bow_valid.toarray(), columns=feature_names)

print(bagofwords_train.shape)
print(bagofwords_valid.shape)

svd = TruncatedSVD(n_components=100)
x = np.arange(100)

x_bow_train = svd.fit_transform(x_bow_train) #Singular Value decomposition, i.e.
dimension reduction
x_bow_valid = svd.transform(x_bow_valid)

"""### 3.3.2 N-gram(bigram)"""

vectorizer = CountVectorizer(ngram_range=(2, 2))

x_bigram_train = vectorizer.fit_transform(sub_train['content'])
x_bigram_valid = vectorizer.transform(sub_valid['content'])
```

```
x_bigram_train = Normalizer().fit_transform(x_bigram_train) #Normalization (mean = 0,
st dev = 1)
x_bigram_valid = Normalizer().transform(x_bigram_valid)

feature_names = vectorizer.get_feature_names_out()

bigram_train = pd.DataFrame(x_bigram_train.toarray(), columns=feature_names)
bigram_valid = pd.DataFrame(x_bigram_valid.toarray(), columns=feature_names)

print(bigram_train.shape)
print(bigram_valid.shape)

svd = TruncatedSVD(n_components=100)
x = np.arange(100)

x_bigram_train = svd.fit_transform(x_bigram_train) #Singular Value decomposition, i.e.
dimension reduction
x_bigram_valid = svd.transform(x_bigram_valid)

"""### 3.3.3 N-gram(trigram)"""

vectorizer = CountVectorizer(ngram_range=(3, 3))

x_trigram_train = vectorizer.fit_transform(sub_train['content'])
x_trigram_valid = vectorizer.transform(sub_valid['content'])

x_trigram_train = Normalizer().fit_transform(x_trigram_train) #Normalization (mean =
0, st dev = 1)
x_trigram_valid = Normalizer().transform(x_trigram_valid)

feature_names = vectorizer.get_feature_names_out()

trigram_train = pd.DataFrame(x_trigram_train.toarray(), columns=feature_names)
trigram_valid = pd.DataFrame(x_trigram_valid.toarray(), columns=feature_names)

print(trigram_train.shape)
print(trigram_valid.shape)

svd = TruncatedSVD(n_components=100)
x = np.arange(100)

x_trigram_train = svd.fit_transform(x_trigram_train) #Singular Value decomposition,
i.e. dimension reduction
```

```
x_trigram_valid = svd.transform(x_trigram_valid)

"""### 3.3.4 TF-IDF"""

tfidf_vectorizer = TfidfVectorizer(norm='l2', smooth_idf=False)

x_tfidf_train = tfidf_vectorizer.fit_transform(sub_train['content'])
x_tfidf_valid = tfidf_vectorizer.transform(sub_valid['content'])

x_tfidf_train = Normalizer().fit_transform(x_tfidf_train) #Normalization (mean = 0, st
dev = 1)
x_tfidf_valid = Normalizer().transform(x_tfidf_valid)

feature_names_tfidf = tfidf_vectorizer.get_feature_names_out()

tfidf_train = pd.DataFrame(x_tfidf_train.toarray(), columns=feature_names_tfidf)
tfidf_valid = pd.DataFrame(x_tfidf_valid.toarray(), columns=feature_names_tfidf)

print(tfidf_train.shape)
print(tfidf_valid.shape)

svd = TruncatedSVD(n_components=100)
x = np.arange(100)

x_tfidf_train = svd.fit_transform(x_tfidf_train) #Singular Value decomposition, i.e.
dimension reduction
x_tfidf_valid = svd.transform(x_tfidf_valid)

"""## 3.4 Label Encoding"""

y_train_cleaned = y_train.apply(lambda x: x[0].strip('[]'))
y_valid_cleaned = y_valid.apply(lambda x: x[0].strip('[]'))
y_test_cleaned = y_test.apply(lambda x: x[0].strip('[]'))

category_to_label = {
    'business': 0,
    'sport': 1,
    'politic': 2,
    'entertainment': 3,
    'tech': 4
}
y_train_encoded = y_train_cleaned.map(category_to_label)
y_valid_encoded = y_valid_cleaned.map(category_to_label)
y_test_encoded = y_test_cleaned.map(category_to_label)
```

```
print(y_train_encoded)
print(y_valid_encoded)

"""# 4.0 Model building (Machine learning)

## 4.1 Logistic

### 4.1.1 Word2Vec logit

#### 4.1.1.1 Word2Vec No penalty

>
"""

# multinomial is suitable for our task, 5 class, this model is based on word2vec
process and with no penalty
logit_word2vec = LogisticRegression(penalty= None, multi_class='multinomial',
solver='lbfgs', random_state=3407)
logit_word2vec.fit(trainvector_word2vec, y_train_encoded)

logit_pred_word2vec = logit_word2vec.predict(validvector_word2vec)
acc_logit_w2v = accuracy_score(y_valid_encoded, logit_pred_word2vec)
print("Accuracy_Score_w2v_logit", acc_logit_w2v)

"""#### 4.1.1.2 Word2Vec L1"""

alphas = np.logspace(-5, 5, 50, base=10)

# Create a logistic regression model with L2 regularization
# Cs are the inverse of regularization strengths
l1_word2vec = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l1',
solver='liblinear', multi_class='ovr', random_state=3407)

l1_word2vec.fit(trainvector_word2vec, y_train_encoded)

l1_pred_word2vec = l1_word2vec.predict(validvector_word2vec)
acc_l1_w2v = accuracy_score(y_valid_encoded, l1_pred_word2vec)
print("Accuracy_Score_w2v_l1", acc_l1_w2v)

"""#### 4.1.1.3 Word2Vec L2"""

# Create a logistic regression model with L2 regularization
# Cs are the inverse of regularization strengths
```

```
l2_word2vec = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l2', solver='lbfgs',
multi_class='multinomial', random_state=3407)

l2_word2vec.fit(trainvector_word2vec, y_train_encoded)

l2_pred_word2vec = l2_word2vec.predict(validvector_word2vec)
acc_l2_w2v = accuracy_score(y_valid_encoded, l2_pred_word2vec)
print("Accuracy_Score_w2v_l2", acc_l2_w2v)

"""### 4.1.2 BOW logit

#### 4.1.2.1 BOW No penalty
"""

logit_bow = LogisticRegression(penalty= None, multi_class='multinomial',
solver='lbfgs', random_state=3407)
logit_bow.fit(x_bow_train, y_train_encoded)

logit_pred_bow = logit_bow.predict(x_bow_valid)
acc_logit_bow = accuracy_score(y_valid_encoded, logit_pred_bow)
print("Accuracy_Score_bow_logit", acc_logit_bow)

"""#### 4.1.2.2 BOW L1"""

l1_bow = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l1', solver='liblinear',
multi_class='ovr', random_state=3407)
l1_bow.fit(x_bow_train, y_train_encoded)

l1_pred_bow = l1_bow.predict(x_bow_valid)
acc_l1_bow = accuracy_score(y_valid_encoded, l1_pred_bow)
print("Accuracy_Score_bow_l1", acc_l1_bow)

"""#### 4.1.2.3 BOW L2"""

l2_bow = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l2', solver='lbfgs',
multi_class='multinomial', random_state=3407)
l2_bow.fit(x_bow_train, y_train_encoded)

l2_pred_bow = l2_bow.predict(x_bow_valid)
acc_l2_bow = accuracy_score(y_valid_encoded, l2_pred_bow)
print("Accuracy_Score_bow_l2", acc_l2_bow)

"""### 4.1.3 2-grams logit
```

```
#### 4.1.3.1 2-grams No penalty
"""

logit_bigram = LogisticRegression(penalty= None, multi_class='multinomial',
solver='lbfgs', random_state=3407)
logit_bigram.fit(x_bigram_train, y_train_encoded)

logit_pred_bigram = logit_bigram.predict(x_bigram_valid)
acc_logit_bigram = accuracy_score(y_valid_encoded, logit_pred_bigram)
print("Accuracy_Score_bigram_logit", acc_logit_bigram)

"""#### 4.1.3.2 2-grams L1"""

l1_bigram = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l1', solver='liblinear',
multi_class='ovr', random_state=3407)
l1_bigram.fit(x_bigram_train, y_train_encoded)

l1_pred_bigram = l1_bigram.predict(x_bigram_valid)
acc_l1_bigram = accuracy_score(y_valid_encoded, l1_pred_bigram)
print("Accuracy_Score_bigram_l1", acc_l1_bigram)

"""#### 4.1.3.3 2-grams L2"""

l2_bigram = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l2', solver='lbfgs',
multi_class='multinomial', random_state=3407)
l2_bigram.fit(x_bigram_train, y_train_encoded)

l2_pred_bigram = l2_bigram.predict(x_bigram_valid)
acc_l2_bigram = accuracy_score(y_valid_encoded, l2_pred_bigram)
print("Accuracy_Score_bigram_l2", acc_l2_bigram)

"""### 4.1.4 3-grams logit

#### 4.1.4.1 3-grams no penalty
"""

logit_trigram = LogisticRegression(penalty= None, multi_class='multinomial',
solver='lbfgs', random_state=3407)
logit_trigram.fit(x_trigram_train, y_train_encoded)

logit_pred_trigram = logit_trigram.predict(x_trigram_valid)
acc_logit_trigram = accuracy_score(y_valid_encoded, logit_pred_trigram)
print("Accuracy_Score_trigram_logit", acc_logit_trigram)
```

```
"""#### 4.1.4.2 3-grams L1"""

l1_trigram = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l1', solver='liblinear',
multi_class='ovr', random_state=3407)
l1_trigram.fit(x_trigram_train, y_train_encoded)

l1_pred_trigram = l1_trigram.predict(x_trigram_valid)

acc_l1_trigram = accuracy_score(y_valid_encoded, l1_pred_trigram)
print("Accuracy_Score_trigram_l1", acc_l1_trigram)

"""#### 4.1.4.3 3-grams L2"""

l2_trigram = LogisticRegressionCV(Cs=1/alphas, cv=5, penalty='l2', solver='lbfgs',
multi_class='multinomial', random_state=3407)
l2_trigram.fit(x_trigram_train, y_train_encoded)

l2_pred_trigram = l2_trigram.predict(x_trigram_valid)

acc_l2_trigram = accuracy_score(y_valid_encoded, l2_pred_trigram)
print("Accuracy_Score_trigram_l2", acc_l2_trigram)

"""### 4.1.5 TF-IDF logit

#### 4.1.5.1 TF-IDF No Penalty
"""

logit_tfidf = LogisticRegression(penalty= None, multi_class='multinomial',
solver='lbfgs', random_state=3407)
logit_tfidf.fit(x_tfidf_train, y_train_encoded)

logit_pred_tfidf = logit_tfidf.predict(x_tfidf_valid)

acc_logit_tfidf = accuracy_score(y_valid_encoded, logit_pred_tfidf)
print("Accuracy_Score_TFIDF_logit", acc_logit_tfidf)

"""#### 4.1.5.2 TF-IDF L1"""

l1_tfidf = LogisticRegressionCV(cv=5, penalty='l1', solver='liblinear',
multi_class='ovr', random_state=3407)
l1_tfidf.fit(x_tfidf_train, y_train_encoded)

l1_pred_tfidf = l1_tfidf.predict(x_tfidf_valid)
```

```
acc_l1_tfidf = accuracy_score(y_valid_encoded, l1_pred_tfidf)
print("Accuracy_Score_TFIDF_L1", acc_l1_tfidf)

"""#### 4.1.5.3 TF-IDF L2"""

l2_tfidf = LogisticRegressionCV(cv=5, penalty='l2', solver='lbfgs',
multi_class='multinomial', random_state=3407)
l2_tfidf.fit(x_tfidf_train, y_train_encoded)

l2_pred_tfidf = l2_tfidf.predict(x_tfidf_valid)

acc_l2_tfidf = accuracy_score(y_valid_encoded, l2_pred_tfidf)
print("Accuracy_Score_TFIDF_L2", acc_l2_tfidf)

"""### 4.1.6 Text Presentation selection"""

Logit = {
   'Text presentation': ['Word Embedding', 'Bag of words', 'Bigraam', 'Trigram','TF-
IDF'],
   'Accuracy Score using Logistic regression': [acc_logit_w2v, acc_logit_bow,
acc_logit_bigram, acc_logit_trigram, acc_logit_tfidf]
}
Logit_l1 = {'Text presentation': ['Word Embedding', 'Bag of words', 'Bigraam',
'Trigram','TF-IDF'],
   'Accuracy Score using Logistic regression L1': [acc_l1_w2v, acc_l1_bow,
acc_l1_bigram, acc_l1_trigram, acc_l1_tfidf]
}
Logit_l2 = {'Text presentation': ['Word Embedding', 'Bag of words', 'Bigraam',
'Trigram','TF-IDF'],
   'Accuracy Score using Logistic regression L2': [acc_l2_w2v, acc_l2_bow,
acc_l2_bigram, acc_l2_trigram, acc_l2_tfidf]
}

df = pd.concat([pd.DataFrame(Logit),pd.DataFrame(Logit_l1),pd.DataFrame(Logit_l2)],
axis=1)

df

print(classification_report(y_valid_encoded,logit_pred_tfidf, digits=3))

"""## 4.2 Gradient Boosting Decision Tree

### 4.2.1 XGBoost
```

```
#### Optuna Optimization
"""

# This is the optuna optimization.

# from sklearn.metrics import accuracy_score
# def objective(trial):
#     # Define the search space for hyperparameters
#     param = {
#         'objective': 'multi:softprob',  # Multi-class classification
#         'eval_metric': 'mlogloss',  # Use logloss/merror for multi-class
classification
#         'eta': trial.suggest_float('eta', 0.2, 0.4),
#         'num_boost_round': 300,  # Fix the boosting round and use early stopping
#         'max_depth': trial.suggest_int('max_depth', 3, 10),
#         'subsample': trial.suggest_float('subsample', 0.5, 1.0),
#         'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
#         'gamma': trial.suggest_float('gamma', 0.0, 10.0),
#         'min_child_weight': trial.suggest_float('min_child_weight', 0.1, 10.0),
#         'lambda': trial.suggest_float('lambda', 0.1, 10.0),
#         'alpha': trial.suggest_float('alpha', 0.0, 10.0),
#         'num_class': 5,  # Number of classes
#     }

#     # Convert the data into DMatrix format
#     dtrain = xgb.DMatrix(tfidf_train, label=y_train_encoded)
#     dvalid = xgb.DMatrix(tfidf_valid, label=y_valid_encoded)

#     # Define the pruning callback for early stopping
#     pruning_callback = optuna.integration.XGBoostPruningCallback(trial, 'validation-
mlogloss')

#     # Train the model with early stopping
#     evals = [(dtrain, 'train'), (dvalid, 'validation')]
#     model = xgb.train(param, dtrain, evals=evals, num_boost_round=300,
early_stopping_rounds=50, callbacks=[pruning_callback])

#     # Make predictions on the validation set
#     y_pred_proba = model.predict(dvalid)
#     y_pred = np.argmax(y_pred_proba, axis=1)  # Convert probabilities to class labels

#     # Calculate accuracy and logloss
#     accuracy = accuracy_score(y_valid_encoded, y_pred)
#     logloss = log_loss(y_valid_encoded, y_pred_proba)
```

```
#     # Return or print the losses and accuracy
#     print(f"Train Loss: {model.eval(dtrain)}")
#     print(f"Validation Loss: {model.eval(dvalid)}")
#     print(f"Validation Accuracy: {accuracy}")

#     return accuracy  # Or logloss if you want to minimize that

# # Create an Optuna study and optimize the objective function
# study = optuna.create_study(direction='maximize')  # Maximize accuracy for multi-
class classification
# study.optimize(objective, n_trials=75)

# # Print the best hyperparameters and the best accuracy
# best_params = study.best_params
# best_accuracy = study.best_value
# print("Best Hyperparameters: ", best_params)
# print("Best Accuracy: ", best_accuracy)

# # history graph of optimization
# history_plot = vis.plot_optimization_history(study)
# history_plot.show()

# # Different importance weights from optimization
# importance_plot = vis.plot_param_importances(study)
# importance_plot.show()

"""#### Training XGBoost"""

# best_params.update({
#     'objective': 'multi:softprob',
#     'eval_metric': 'mlogloss',
#     'num_class': 5,
#     'seed': 3407
# })
# print(best_params)

dtrain = xgb.DMatrix(tfidf_train, label=y_train_encoded)
dvalid = xgb.DMatrix(tfidf_valid, label=y_valid_encoded)

loaded_model_xgb = checkpoint['XGB_model']
y_pred_proba = loaded_model_xgb.predict(dvalid)
y_pred = np.argmax(y_pred_proba, axis=1)
```

```
accuracy = accuracy_score(y_valid_encoded, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")

# epochs = len(evals_result['train']['mlogloss'])
# x_axis = range(0, epochs)

# # Drawing a line chart to display the train and validation loss
# fig, ax = plt.subplots()
# ax.plot(x_axis, evals_result['train']['mlogloss'], label='Train Log Loss')
# ax.plot(x_axis, evals_result['eval']['mlogloss'], label='Validation Log Loss')
# ax.legend()
# plt.xlabel('Epochs')
# plt.ylabel('Log Loss')
# plt.title('XGBoost Log Loss across Epochs')
# plt.show()

print(classification_report(y_valid_encoded, y_pred, digits=3))

"""# 5.0 Model building (Deep learning)"""

torch.set_rng_state(checkpoint['rng_state'])
np.random.set_state(checkpoint['numpy_rng_state'])
random.setstate(checkpoint['random_state'])

"""## 5.1 Dataloader"""

train_for_nn = sub_train.copy()
val_for_nn = sub_valid.copy()
test_for_nn = test.copy()

# Because the data in content is in token, we combine the token into list
train_for_nn['category'] = train_for_nn['category'].apply(lambda x: ', '.join(x))
val_for_nn['category'] = val_for_nn['category'].apply(lambda x: ', '.join(x))
test_for_nn['category'] = test_for_nn['category'].apply(lambda x: ', '.join(x))

le = LabelEncoder()
train_target = le.fit_transform(train_for_nn['category']) # convert five categories
into integers
val_target = le.transform(val_for_nn['category'])
test_target = le.transform(test_for_nn['category'])

train_for_nn['category'] = train_target
val_for_nn['category'] = val_target
test_for_nn['category'] = test_target
```

```
print(le.classes_) # this shows which index maps to which class

train_data = train_for_nn[['category','content']]
val_data = val_for_nn[['category','content']]
test_data = test_for_nn[['category','content']]

class TextDataset(utils.data.Dataset):
    def __init__(self, myData):
        """
        myData should be a dataframe object containing both y (first col) and X (second
col)
        """
        super().__init__()
        self.data = myData

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return (self.data.iloc[idx,0], self.data.iloc[idx,1])

# build torch datasets
train_torch = TextDataset(train_data)
val_torch = TextDataset(val_data)
test_torch = TextDataset(test_data)

tokenizer = get_tokenizer('basic_english')

from torchtext.vocab import build_vocab_from_iterator

# ===== Build vocabulary =====
# an unknown token is added for all unknown words outside the documents
# you may specify the min_freq to filter out infrequent words
vocabulary = build_vocab_from_iterator(
    [tokenizer(msg) for msg in train_for_nn['content']],
    specials=["<unk>"],
    min_freq = 3, # filter out all words that appear less than three times
)
# Set to avoid errors with unknown words
vocabulary.set_default_index(vocabulary["<unk>"])

# define a function that converts a document into tokens (represented by index)
def doc_tokenizer(doc):
```

```
    return torch.tensor([vocabulary[token] for token in tokenizer(doc)],
dtype=torch.long)

def collate_batch_advanced(batch):

    target_list, text_list = [], []

    # loop through all samples in batch
    for idx in range(len(batch)):

        _label = batch[idx][0]
        _text = batch[idx][1]

        target_list.append( _label )
        tokens = doc_tokenizer( _text )
        text_list.append(tokens)

    # convert to torch tensor
    target_list = torch.tensor(target_list, dtype=torch.int64)

    return target_list, text_list


# define the evaluate function, notice we need to pad each document with 0
def evaluate_adv(dataloader, model):
    y_pred = torch.tensor([], dtype=torch.float32)
    y_true = torch.tensor([], dtype=torch.float32)
    model.eval()
    with torch.no_grad():
        for label, text in dataloader:
            y_pred_batch = model(text)
            if y_pred_batch.dim() == 1:
                y_pred_batch = y_pred_batch.unsqueeze(0)
            y_pred = torch.cat((y_pred, y_pred_batch), dim=0)
            label = label.view(-1)
            y_true = torch.cat((y_true, label), dim=0)
    return y_pred, y_true

batchSize = 16
train_loader = utils.data.DataLoader(train_torch, batch_size=batchSize, shuffle=True,
collate_fn=collate_batch_advanced)
val_loader = utils.data.DataLoader(val_torch, batch_size=batchSize, shuffle=False,
collate_fn=collate_batch_advanced)
test_loader = utils.data.DataLoader(test_torch, batch_size=batchSize, shuffle=False,
```

```
collate_fn=collate_batch_advanced)


class EarlyStopping:
    def __init__(self, patience=10, verbose=False, min_delta=0.0003):
        """
        Args:
            patience (int): The number of epochs allowed when validation loss is not
improving.
            verbose (bool): If True, print the information.
            min_delta (float): the minimum threhold in improving the validation loss, if
less than the threshold identify as no improvement
        """
        self.patience = patience
        self.verbose = verbose
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        if self.best_loss is None or val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
        else:
            self.counter += 1  # If no improvement, add 1 to counter
            if self.counter >= self.patience:
                self.early_stop = True  # early stopping
                if self.verbose:
                    print("Early stopping triggered")

def train_and_validate(model, train_loader, valid_loader, learning_rate,
weight_decay):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay =
weight_decay)
    valid_losses = []
    valid_accuracies = []
    train_losses = []
    early_stopper = EarlyStopping(patience=10, verbose=True)

    for epoch in range(10):
        model.train()
        train_loss = 0.0  # Initialize train loss for each epoch
        total_samples = 0
```

```
        for label, text in train_loader:
            optimizer.zero_grad()
            text = pad_sequence(text, batch_first=True, padding_value=0)

            outputs = model(text)
            loss = criterion(outputs, label)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * label.size(0)  # Multiply loss by batch size
            total_samples += label.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)  # Append average loss for the epoch

        y_pred_val, y_val = evaluate_adv(valid_loader, model)
        loss_val = criterion(y_pred_val, y_val.long())

        _, pred_labels = torch.max(y_pred_val, 1)
        valid_accuracy = (pred_labels == y_val).float().mean().item() * 100

        valid_losses.append(loss_val.item())
        valid_accuracies.append(valid_accuracy)

        print(f'Epoch [{epoch+1}/{100}], Train Loss: {train_losses[-1]:.4f}, Valid
Loss: {valid_losses[-1]:.4f}, Valid Accuracy: {valid_accuracies[-1]:.2f}%')

        early_stopper(loss_val / len(valid_loader), model)
        if early_stopper.early_stop:
            print(f"Early stopping at epoch {epoch + 1}")
            break
    return max(valid_accuracies), train_losses, valid_losses

def predict(model, data_loader):
    model.eval()
    all_preds = []
    all_probs = []
    all_labels = []
    with torch.no_grad():
        for labels, texts in data_loader:
            outputs = model(texts)
            probabilities = F.softmax(outputs, dim=1)
            predicted_classes = torch.argmax(probabilities, dim=1)

            all_preds.extend(predicted_classes.numpy())
```

```
            all_probs.extend(probabilities.numpy())
            all_labels.extend(labels.numpy().flatten())

    return np.array(all_preds), np.array(all_probs), np.array(all_labels)

"""## 5.2 Feedforward prepare data"""

vocab_size = len(vocabulary)

class FeedforwardNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim,
dropout_rate):
        super(FeedforwardNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.fc1 = nn.Linear(embedding_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        x = pad_sequence(x, batch_first=True, padding_value=0)
        embedded = self.embedding(x)
        embedded = embedded.mean(dim=1)
        out = self.fc1(embedded)
        out = torch.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
        return out

def evaluate_model(model, val_loader, criterion):
    model.eval()
    total_loss, correct, total = 0, 0, 0
    with torch.no_grad():
        for label, text in val_loader:
            output = model(text)
            loss = criterion(output, label)
            total_loss += loss.item()

            _, predicted = torch.max(output, 1)
            correct += (predicted == label).sum().item()
            total += label.size(0)

    avg_loss = total_loss / len(val_loader)
    accuracy = correct / total
    return avg_loss, accuracy
```

```
"""###5.2.1 FNN optuna

"""

# def objective(trial):

#     embedding_dim = trial.suggest_int('embedding_dim', 16, 32)
#     hidden_dim = trial.suggest_int('hidden_dim', 16, 30)
#     dropout_rate = trial.suggest_float('dropout_rate', 0.1, 0.5)
#     learning_rate = trial.suggest_loguniform('lr', 1e-4, 1e-2)
#     patience = 10

#     model = FeedforwardNN(len(vocabulary), embedding_dim, hidden_dim,
len(le.classes_), dropout_rate)

#     optimizer = optim.Adam(model.parameters(), lr=learning_rate)
#     criterion = nn.CrossEntropyLoss()

#     best_val_loss = float('inf')
#     best_epoch = 0
#     num_epochs = 30
#     early_stop = False

#     for epoch in range(num_epochs):
#         if early_stop:
#             break

#         model.train()
#         total_train_loss = 0
#         for label, text in train_loader:
#             optimizer.zero_grad()
#             output = model(text)
#             loss = criterion(output, label)
#             loss.backward()
#             optimizer.step()
#             total_train_loss += loss.item()

#         avg_train_loss = total_train_loss / len(train_loader)



#         val_loss, val_accuracy = evaluate_model(model, val_loader, criterion)


#         if val_loss < best_val_loss:
```

```
#           best_val_loss = val_loss
#           best_epoch = epoch
#       elif epoch - best_epoch >= patience:
#           early_stop = True

#       print(f"Epoch [{epoch+1}/{num_epochs}] | Train Loss: {avg_train_loss:.4f} |
Validation Loss: {val_loss:.4f} | Validation Accuracy: {val_accuracy:.4f}")

#       # Prune trial if no improvement
#       trial.report(val_loss, epoch)

#       if trial.should_prune():
#           raise optuna.exceptions.TrialPruned()

#    return best_val_loss

# study = optuna.create_study(direction='minimize')
# study.optimize(objective, n_trials=30)

# print("Best trial:")
# trial = study.best_trial
# print("  Value: {}".format(trial.value))
# print("  Params: ")
# for key, value in trial.params.items():
#    print("    {}: {}".format(key, value))

"""###5.2.2 Load FNN best parameter"""

FNN_best_params = checkpoint['FNN_best_params']

FNN =  FeedforwardNN(
   vocab_size=len(vocabulary),
   embedding_dim=FNN_best_params['embedding_dim'],
   hidden_dim=FNN_best_params['hidden_dim'],
   output_dim=len(le.classes_),
   dropout_rate=FNN_best_params['dropout_rate']

)
train_loader = DataLoader(train_torch, batch_size=FNN_best_params['BATCH_SIZE'],
shuffle=True, collate_fn=collate_batch_advanced)
val_loader = DataLoader(val_torch, batch_size=FNN_best_params['BATCH_SIZE'],
shuffle=False, collate_fn=collate_batch_advanced)

"""###5.2.3 Train FNN"""
```

```
# class TextDataset(torch.utils.data.Dataset):
#     def __init__(self, myData):
#         super().__init__()
#         self.data = myData

#     def __len__(self):
#         return len(self.data)

#     def __getitem__(self, idx):
#         return (self.data.iloc[idx, 0], self.data.iloc[idx, 1])


# tokenizer = get_tokenizer('basic_english')
# vocabulary = build_vocab_from_iterator(
#     [tokenizer(msg) for msg in train_for_nn['content']],
#     specials=["<unk>"],
#     min_freq=3,
# )
# vocabulary.set_default_index(vocabulary["<unk>"])

# def doc_tokenizer(doc):
#     return torch.tensor([vocabulary[token] for token in tokenizer(doc)],
dtype=torch.long)

# def collate_batch(batch):
#     target_list, text_list = [], []
#     for label, text in batch:
#         target_list.append(label)

#         if isinstance(text, list):
#             text = ' '.join(text)
#         tokens = doc_tokenizer(text)
#         text_list.append(tokens)
#     target_list = torch.tensor(target_list, dtype=torch.int64)
#     text_list = pad_sequence(text_list, batch_first=True, padding_value=0)
#     return target_list, text_list


# batch_size = 16
# train_loader = DataLoader(TextDataset(train_data), batch_size=batch_size,
shuffle=True, collate_fn=collate_batch)
# val_loader = DataLoader(TextDataset(val_data), batch_size=batch_size, shuffle=False,
collate_fn=collate_batch)
```

```
def train_and_validate_FNN(model, train_loader, valid_loader, learning_rate):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    valid_accuracies = []
    train_losses = []
    valid_losses = []
    valid_f1_scores = []

    device = torch.device('cpu')

    for epoch in range(50):
        model.train()
        train_loss = 0.0
        total_samples = 0

        for batch in train_loader:
            optimizer.zero_grad()

            labels, input_data = batch
            labels = labels.to(device)
            input_data = input_data.to(device)

            outputs = model(input_data)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * labels.size(0)
            total_samples += labels.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)



        model.eval()
        y_pred_val = []
        y_val = []
        valid_loss = 0.0
        total_valid_samples = 0

        with torch.no_grad():
            for batch in valid_loader:
```

```
            labels, input_data = batch
            labels = labels.to(device)
            input_data = input_data.to(device)

            outputs = model(input_data)



            loss = criterion(outputs, labels)
            valid_loss += loss.item() * labels.size(0)
            total_valid_samples += labels.size(0)

            _, preds = torch.max(outputs, 1)
            y_pred_val.extend(preds.cpu().numpy())
            y_val.extend(labels.cpu().numpy())


        average_valid_loss = valid_loss / total_valid_samples
        valid_losses.append(average_valid_loss)

        val_accuracy = (np.array(y_pred_val) == np.array(y_val)).mean() * 100
        valid_accuracies.append(val_accuracy)


        f1 = f1_score(y_val, y_pred_val, average='weighted')
        valid_f1_scores.append(f1)

        print(f'Epoch [{epoch+1}/50], Train Loss: {train_losses[-1]:.4f}, Valid Loss:
{valid_losses[-1]:.4f}, Valid Accuracy: {valid_accuracies[-1]:.2f}%, F1 Score:
{f1:.4f}')


    return train_losses, valid_losses, valid_accuracies, valid_f1_scores

# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# train_losses, valid_losses, valid_accuracies, valid_f1_scores =
train_and_validate_FNN(
#     FNN, train_loader, val_loader,
#     learning_rate=FNN_best_params['lr'])

"""###5.2.4 Load thje final trained FNN Model and parameter"""

FNN.load_state_dict(checkpoint['FNN_state_dict'])
```

```
y_pred, y_probs, all_labels = predict(FNN, val_loader)

accuracy = accuracy_score(all_labels, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")

# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# x_axis = range(0, len(train_losses))
# fig, ax = plt.subplots()
# ax.plot(x_axis, train_losses, label='Training Loss')
# ax.plot(x_axis, valid_losses, label='Validation Loss')
# ax.set_xlabel('Epochs')
# ax.set_ylabel('Loss')
# ax.set_title('RNN Train and Validation Loss per Epoch')
# ax.legend()
# plt.show()

print(classification_report(all_labels, y_pred, digits=3))

"""## 5.3 Vanila Recurrent

"""

class RNNClassifier(nn.Module):
   def __init__(self, vocab_size, embedding_dim, hidden_dim, num_RNNLayer, dropout):
      super(RNNClassifier, self).__init__()

      self.hidden_dim = hidden_dim
      self.num_RNNLayer = num_RNNLayer
      # Embedding layer
      self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
      # Use layer normalization to normalize
      self.embed_norm = nn.LayerNorm(embedding_dim)
      self.rnn = nn.RNN(input_size=embedding_dim,
                     hidden_size=hidden_dim,
                     num_layers= num_RNNLayer,
                     batch_first=True)
      # Dropout
      self.dropout = nn.Dropout(dropout)
      self.linear = nn.Linear(hidden_dim, 5)

   def forward(self, x):
```

```
        x = pad_sequence(x, batch_first=True, padding_value=0)
        h = torch.zeros(self.num_RNNLayer, x.size(0), self.hidden_dim)
        torch.nn.init.xavier_normal_(h)

        out = self.embedding(x)
        out = self.embed_norm(out)
        out, h = self.rnn(out, h)
        # Average pooling
        out = torch.mean(out, dim=1)
        out = self.dropout(out)
        out = self.linear(out)
        return out

def RNN_train_and_validate(model, train_loader, valid_loader, learning_rate,
NUM_EPOCHS, weight_decay):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay =
weight_decay)
    valid_losses = []
    valid_accuracies = []
    train_losses = []
    early_stopper = EarlyStopping(patience=10, verbose=True)

    for epoch in range(NUM_EPOCHS): # loop through epochs
        model.train()
        train_loss = 0.0  # Initialize train loss for each epoch
        total_samples = 0
        for label, text in train_loader:
            optimizer.zero_grad()
            text = pad_sequence(text, batch_first=True, padding_value=0)

            outputs = model(text)
            loss = criterion(outputs, label)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * label.size(0)  # Multiply loss by batch size
            total_samples += label.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)  # Append average loss for the epoch

        y_pred_val, y_val = evaluate_adv(valid_loader, model)
        loss_val = criterion(y_pred_val, y_val.long())
```

```
        _, pred_labels = torch.max(y_pred_val, 1)
        valid_accuracy = (pred_labels == y_val).float().mean().item() * 100

        valid_losses.append(loss_val.item())
        valid_accuracies.append(valid_accuracy)

        print(f'Epoch [{epoch+1}/{NUM_EPOCHS}], Train Loss: {train_losses[-1]:.4f},
Valid Loss: {valid_losses[-1]:.4f},  Valid Accuracy: {valid_accuracies[-1]:.2f}%')

        early_stopper(loss_val / len(valid_loader), model)
        if early_stopper.early_stop:
            print(f"Early stopping at epoch {epoch + 1}")
            break
    return max(valid_accuracies), train_losses, valid_losses

# This is the optuna optimization.

# def objective(trial):
#     vocab_size = len(vocabulary)
#     embedding_dim = trial.suggest_int('embedding_dim', 64, 128)
#     hidden_dim = trial.suggest_int('hidden_dim', 32, 96)
#     num_RNNLayer = trial.suggest_int('num_RNNLayer', 1, 2)
#     dropout = trial.suggest_float('dropout', 0.35, 0.7)
#     learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2)
#     weight_decay = trial.suggest_float('weight_decay', 1e-4, 1e-2)
#     BATCH_SIZE = 32,
#     NUM_EPOCHS = 50

#     train_loader = DataLoader(train_torch, batch_size=BATCH_SIZE, shuffle=True,
collate_fn=collate_batch_advanced)
#     val_loader = DataLoader(val_torch, batch_size=BATCH_SIZE, shuffle=False,
collate_fn=collate_batch_advanced)

#     model = RNNClassifier(vocab_size, embedding_dim, hidden_dim, num_RNNLayer,
dropout)
#     valid_accuracies, _, _= RNN_train_and_validate(model, train_loader, val_loader,
learning_rate, NUM_EPOCHS, weight_decay)
#     return valid_accuracies

# study = optuna.create_study(direction='maximize')
# study.optimize(objective, n_trials=50)

# best_params = study.best_params
# best_accuracy = study.best_value
```

```
# print("Best Hyperparameters: ", best_params)
# print("Best accuracy: ", best_accuracy)

# history_plot = vis.plot_optimization_history(study)
# history_plot.show()

# importance_plot = vis.plot_param_importances(study)
# importance_plot.show()

RNN_best_params = checkpoint['RNN_best_params_state_dict']

RNN_model = RNNClassifier(
    vocab_size=RNN_best_params['vocab_size'],
    embedding_dim=RNN_best_params['embedding_dim'],
    hidden_dim=RNN_best_params['hidden_dim'],
    num_RNNLayer=RNN_best_params['num_RNNLayer'],
    dropout=RNN_best_params['dropout']
)

def RNN_train_and_validate_update(model, train_loader, valid_loader, learning_rate,
NUM_EPOCHS, weight_decay, model_save_path):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay =
weight_decay)
    valid_losses = []
    valid_accuracies = []
    train_losses = []

    early_stopper = EarlyStopping(patience=50, verbose=True)

    best_accuracy = 0.0
    best_epoch = 0

    for epoch in range(NUM_EPOCHS):
        model.train()
        train_loss = 0.0
        total_samples = 0
        for label, text in train_loader:
            optimizer.zero_grad()
            text = pad_sequence(text, batch_first=True, padding_value=0)

            outputs = model(text)
            loss = criterion(outputs, label)
            loss.backward()
```

```
            optimizer.step()
            train_loss += loss.item() * label.size(0)
            total_samples += label.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)

        y_pred_val, y_val = evaluate_adv(valid_loader, model)
        loss_val = criterion(y_pred_val, y_val.long())

        _, pred_labels = torch.max(y_pred_val, 1)
        valid_accuracy = (pred_labels == y_val).float().mean().item() * 100

        valid_losses.append(loss_val.item())
        valid_accuracies.append(valid_accuracy)

        print(f'Epoch [{epoch+1}/{NUM_EPOCHS}], Train Loss: {train_losses[-1]:.4f},
Valid Loss: {valid_losses[-1]:.4f}, Valid Accuracy: {valid_accuracies[-1]:.2f}%')
        # Save the best model and calculate the best accuracy
        if valid_accuracy > best_accuracy:
            best_accuracy = valid_accuracy
            best_epoch = epoch + 1
            torch.save(model.state_dict(), model_save_path)

        early_stopper(loss_val / len(valid_loader), model)
        if early_stopper.early_stop:
            print(f"Early stopping at epoch {epoch + 1}")
            break
    print(f'Best Validation Accuracy: {best_accuracy:.2f}% at epoch {best_epoch}')

    return best_accuracy, train_losses, valid_losses

# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# RNN_train_loader = DataLoader(train_torch, batch_size=RNN_best_params['BATCH_SIZE'],
shuffle=True, collate_fn=collate_batch_advanced)
# RNN_val_loader = DataLoader(val_torch, batch_size=RNN_best_params['BATCH_SIZE'],
shuffle=False, collate_fn=collate_batch_advanced)
# _, train_losses, valid_losses = RNN_train_and_validate_update(RNN_model,
RNN_train_loader, RNN_val_loader, RNN_best_params['learning_rate'],
RNN_best_params['NUM_EPOCHS'], RNN_best_params['weight_decay'],
model_save_path='/content/drive/MyDrive/6850/final_rnn_model_state_dict.pth')
```

```
RNN_model.load_state_dict(checkpoint['RNN_model_state_dict'])

y_pred, y_probs, all_labels = predict(RNN_model, val_loader)

accuracy = accuracy_score(all_labels, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")

# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# x_axis = range(0, len(train_losses))
# fig, ax = plt.subplots()
# ax.plot(x_axis, train_losses, label='Training Loss')
# ax.plot(x_axis, valid_losses, label='Validation Loss')
# ax.set_xlabel('Epochs')
# ax.set_ylabel('Loss')
# ax.set_title('RNN Train and Validation Loss per Epoch')
# ax.legend()
# plt.show()

print(classification_report(all_labels, y_pred, digits=3))

"""## 5.4 LSTM

### 5.4.1 Optimization with Optuna
"""

import torch
import torch.nn as nn
from torch.nn.utils.rnn import pad_sequence

class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_LstmLayer, dropout_rate,
fc1_dim):
        super().__init__()

        self.hidden_dim = hidden_dim
        self.num_LstmLayer = num_LstmLayer

        # 1. Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        # 2. Layer Normalizaion after embedding
        self.embed_norm = nn.LayerNorm(embed_dim)
        # 3. LSTM layer
```

```
        self.lstm = nn.LSTM(input_size=embed_dim, hidden_size=hidden_dim,
num_layers=num_LstmLayer, batch_first=True)
        # 4. Dropout layer (moved after LSTM)
        self.dropout = nn.Dropout(p=dropout_rate)
        # 5. First fully connected layer
        self.Linear1 = nn.Linear(hidden_dim, fc1_dim)
        # 6. Second fully connected layer
        self.Linear2 = nn.Linear(fc1_dim, 5)

        # For residual connection in the fully connected layers
        if hidden_dim != fc1_dim:
            self.residual_fc = nn.Linear(hidden_dim, fc1_dim)
        else:
            self.residual_fc = None

    def forward(self, x):
        x = pad_sequence(x, batch_first=True, padding_value=0)

        # Initialize hidden state and cell state
        h = torch.zeros((self.num_LstmLayer, x.size(0), self.hidden_dim)).to(x.device)
        c = torch.zeros((self.num_LstmLayer, x.size(0), self.hidden_dim)).to(x.device)

        # 1. Embedding layer
        out = self.embedding(x)
        # 2. Nrom after Embedding
        out = self.embed_norm(out)
        # 3. LSTM layer
        out, (hidden, cell) = self.lstm(out, (h, c))
        # 4. Layer Normalization and Dropout
        out, _ = torch.max(out, dim=1)
        out = self.dropout(out)

        # Residual connection starts here
        # 5. First fully connected layer with possible residual
        residual = out  # Save for the residual connection
        out = self.Linear1(out)
        out = torch.relu(out)

        if self.residual_fc is not None:
            residual = self.residual_fc(residual)  # Make dimensions match if needed
        out = out + residual  # Add the residual connection

        # 6. Second fully connected layer
        out = self.Linear2(out)
```

```
        return out

#This is the optuna optimization.

# def objective(trial):
#     embed_dim = trial.suggest_int('embed_dim', 86, 158)
#     hidden_dim = trial.suggest_int('hidden_dim', 64, 158)
#     dropout_rate = trial.suggest_float('dropout_rate', 0.45, 0.65)
#     learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 1e-3)
#     weight_decay = trial.suggest_float('weight_decay', 1e-4, 1e-2)
#     fc1_dim = trial.suggest_int('fc1_dim', 86, 158)
#     num_LstmLayer = trial.suggest_int('num_LstmLayer', 1, 2)

#     # initialize model
#     model = LSTMClassifier(len(vocabulary), embed_dim, hidden_dim, num_LstmLayer,
dropout_rate, fc1_dim)

#     valid_accuracies, _, _= train_and_validate(model, train_loader, val_loader,
learning_rate, weight_decay)

#     return valid_accuracies

# study = optuna.create_study(direction='maximize') # maximize the accuracy
# study.optimize(objective, n_trials=100)

# best_params = study.best_params
# best_accuracy = study.best_value
# print("Best Hyperparameters: ", best_params)
# print("Best accuracy: ", best_accuracy)

"""### 5.4.2 Optimization visualization"""

# history_plot = vis.plot_optimization_history(study)
# history_plot.show()

# importance_plot = vis.plot_param_importances(study)
# importance_plot.show()

"""### 5.4.3 Load pre_trained best parameters and model"""

LSTM_best_params = checkpoint['LSTM_best_params_state_dict']

LSTM_before_concat = LSTMClassifier(
```

```
    vocab_size=LSTM_best_params['vocab_size'],
    embed_dim=LSTM_best_params['embed_dim'],
    hidden_dim=LSTM_best_params['hidden_dim'],
    num_LstmLayer=LSTM_best_params['num_LstmLayer'],
    dropout_rate=LSTM_best_params['dropout_rate'],
    fc1_dim = LSTM_best_params['fc1_dim']
)
train_loader = DataLoader(train_torch, batch_size=LSTM_best_params['BATCH_SIZE'],
shuffle=True, collate_fn=collate_batch_advanced)
val_loader = DataLoader(val_torch, batch_size=LSTM_best_params['BATCH_SIZE'],
shuffle=False, collate_fn=collate_batch_advanced)

"""### 5.4.4 Train the model"""

def train_and_validate(model, train_loader, valid_loader, learning_rate,
weight_decay):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay =
weight_decay)
    valid_losses = []
    valid_accuracies = []
    train_losses = []
    early_stopper = EarlyStopping(patience=10, verbose=True)

    for epoch in range(50):
        model.train()
        train_loss = 0.0  # Initialize train loss for each epoch
        total_samples = 0
        for label, text in train_loader:
            optimizer.zero_grad()
            text = pad_sequence(text, batch_first=True, padding_value=0)

            outputs = model(text)
            loss = criterion(outputs, label)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * label.size(0)  # Multiply loss by batch size
            total_samples += label.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)  # Append average loss for the epoch

        y_pred_val, y_val = evaluate_adv(valid_loader, model)
        loss_val = criterion(y_pred_val, y_val.long())
```

```
        _, pred_labels = torch.max(y_pred_val, 1)
        valid_accuracy = (pred_labels == y_val).float().mean().item() * 100

        valid_losses.append(loss_val.item())
        valid_accuracies.append(valid_accuracy)

        print(f'Epoch [{epoch+1}/{100}], Train Loss: {train_losses[-1]:.4f}, Valid
Loss: {valid_losses[-1]:.4f},  Valid Accuracy: {valid_accuracies[-1]:.2f}%')

        early_stopper(loss_val / len(valid_loader), model)
        if early_stopper.early_stop:
            print(f"Early stopping at epoch {epoch + 1}")
            break
    return max(valid_accuracies), train_losses, valid_losses

# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# _, train_losses, valid_losses = train_and_validate(
#     LSTM_before_concat, train_loader, val_loader,
#     LSTM_best_params['learning_rate'], LSTM_best_params['weight_decay'])

LSTM_before_concat.load_state_dict(checkpoint['LSTM_before_concat_state_dict'])

y_pred, y_probs, all_labels = predict(LSTM_before_concat, val_loader)

accuracy = accuracy_score(all_labels, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")

# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# x_axis = range(0, len(train_losses))
# fig, ax = plt.subplots()
# ax.plot(x_axis, train_losses, label='Training Loss')
# ax.plot(x_axis, valid_losses, label='Validation Loss')
# ax.set_xlabel('Epochs')
# ax.set_ylabel('Loss')
# ax.set_title('RNN Train and Validation Loss per Epoch')
# ax.legend()
# plt.show()

print(classification_report(all_labels, y_pred, digits=3))
```

```
"""# 6.0 Model selection and evaluation

## 6.1 Load pre_trained best model
"""

Train_data = pd.concat([train_data, val_data], axis=0)

Train_torch = TextDataset(Train_data)

batchSize = 16
Train_loader = utils.data.DataLoader(Train_torch, batch_size=batchSize, shuffle=True,
collate_fn=collate_batch_advanced)

class EarlyStopping_for_MS:
   def __init__(self, acc_threshold=0.96, verbose=False):#set threshold to 96%

      self.acc_threshold = acc_threshold
      self.verbose = verbose
      self.early_stop = False

   def __call__(self, val_acc, model):
      if val_acc >= self.acc_threshold:
         self.early_stop = True
         if self.verbose:
            print(f"Early stopping triggered at validation accuracy {val_acc:.2f} (>=
{self.acc_threshold:.2f})")

def train_and_validate_for_MS_1st(model, train_loader, valid_loader, learning_rate,
weight_decay, model_save_path):
   criterion = nn.CrossEntropyLoss()
   optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay =
weight_decay)
   valid_losses = []
   valid_accuracies = []
   train_losses = []
   best_accuracy = 0.0
   best_epoch = 0

   for epoch in range(100):
      model.train()
      train_loss = 0.0  # Initialize train loss for each epoch
      total_samples = 0
      for label, text in train_loader:
```

```
            optimizer.zero_grad()
            text = pad_sequence(text, batch_first=True, padding_value=0)

            outputs = model(text)
            loss = criterion(outputs, label)
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * label.size(0)  # Multiply loss by batch size
            total_samples += label.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)  # Append average loss for the epoch

        y_pred_val, y_val = evaluate_adv(valid_loader, model)
        loss_val = criterion(y_pred_val, y_val.long())

        _, pred_labels = torch.max(y_pred_val, 1)
        valid_accuracy = (pred_labels == y_val).float().mean().item() * 100

        valid_losses.append(loss_val.item())
        valid_accuracies.append(valid_accuracy)

        print(f'Epoch [{epoch+1}/{100}], Train Loss: {train_losses[-1]:.4f}, Valid
Loss: {valid_losses[-1]:.4f}, Valid Accuracy: {valid_accuracies[-1]:.2f}%')

        if valid_accuracy > best_accuracy:
            best_accuracy = valid_accuracy
            best_epoch = epoch + 1  # Update best_epoch
            torch.save(model.state_dict(), model_save_path)

    print(f'Best Accuracy: {best_accuracy:.2f}% at epoch {best_epoch}')
    return best_accuracy, train_losses, valid_losses,

"""## 6.2 Train the final LSTM"""

LSTM_after_concat = LSTMClassifier(
    vocab_size=LSTM_best_params['vocab_size'],
    embed_dim=LSTM_best_params['embed_dim'],
    hidden_dim=LSTM_best_params['hidden_dim'],
    num_LstmLayer=LSTM_best_params['num_LstmLayer'],
    dropout_rate=LSTM_best_params['dropout_rate'],
    fc1_dim = LSTM_best_params['fc1_dim']
)
```

```
# This is the train and validation loss. Since we use the loaded model directly, this
code is commented out.

# best_accuracy, train_losses, valid_losses =
train_and_validate_for_MS_1st(LSTM_after_concat, Train_loader, test_loader,
LSTM_best_params['learning_rate'],
LSTM_best_params['weight_decay'],model_save_path='/content/drive/MyDrive/6850/LSTM_aft
er_concat.pth')

"""## 6.3 Model evaluation"""

LSTM_after_concat.load_state_dict(checkpoint['LSTM_after_concat_state_dict'])

y_pred, y_probs, all_labels = predict(LSTM_after_concat, test_loader)

accuracy = accuracy_score(all_labels, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")

print(classification_report(all_labels, y_pred, digits=3))

"""# 7.0 Save all the models and hyperparameters"""

torch.save({
    'device': 'cpu',
    'LSTM_after_concat_state_dict': LSTM_after_concat.state_dict(),
    'LSTM_before_concat_state_dict': LSTM_before_concat.state_dict(),
    'LSTM_best_params_state_dict': LSTM_best_params,
    'RNN_best_params_state_dict': RNN_best_params,
    'RNN_model_state_dict': RNN_model.state_dict(),
    'FNN_state_dict': FNN.state_dict(),
    'FNN_best_params': FNN_best_params,
    'XGB_model': loaded_model_xgb,

    'rng_state': torch.get_rng_state(),
    'numpy_rng_state': np.random.get_state(),
    'random_state': random.getstate()
}, '/content/drive/MyDrive/6850/checkpoint_ML&DL.pth')

torch.save({
    'model_state_dict': LSTM_after_concat.state_dict(),
    'best_params': LSTM_best_params,
    'vocabulary': vocabulary
}, "/content/drive/MyDrive/6850/Group09QBUS6850_best_2024S2.pt")
```

```
# -*- coding: utf-8 -*-
"""Group09QBUS6850_pred_2024S2

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/13gHk8Lgd7oC0XQWjyzCtz16SY6G-p6Zx

#This program is our competition part. Please run this program with CPU.

# Import data
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install torchtext==0.16.0
!python -m spacy download en_core_web_lg

import pickle
import json
import re
import shap
import torch
import spacy
import random
import numpy as np
import pandas as pd
import seaborn as sns
from wordcloud import WordCloud
import matplotlib.pyplot as plt
from collections import Counter

import xgboost as xgb
from sklearn.decomposition import TruncatedSVD
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, Normalizer,
LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, classification_report, log_loss
```

```
from gensim.models import Word2Vec
import torch.nn as nn
import torch.optim as optim
from torch import utils
from torchtext.data.utils import get_tokenizer
from torch.utils.data import DataLoader, Dataset
from torchtext.vocab import build_vocab_from_iterator
import collections
from torch.nn.utils.rnn import pad_sequence
import torch.nn.functional as F
from transformers import DistilBertTokenizer, DistilBertModel
from torch.utils.data import DataLoader, TensorDataset


seed_value = 3407
random.seed(seed_value)
np.random.seed(3407)
torch.manual_seed(3407)
torch.cuda.manual_seed(3407)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

checkpoint = torch.load("/content/drive/MyDrive/6850/Group09QBUS6850_best_2024S2.pt")

competition = pd.read_csv('/content/drive/MyDrive/6850/news-challenge.csv',sep='\t')

"""# 1.0 Load best model's structure"""

import torch
import torch.nn as nn
from torch.nn.utils.rnn import pad_sequence

class LSTMClassifier(nn.Module):
   def __init__(self, vocab_size, embed_dim, hidden_dim, num_LstmLayer, dropout_rate,
fc1_dim):
       super().__init__()

       self.hidden_dim = hidden_dim
       self.num_LstmLayer = num_LstmLayer

       # 1. Embedding layer
       self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
       # 2. Layer Normalizaion after embedding
```

```
        self.embed_norm = nn.LayerNorm(embed_dim)
        # 3. LSTM layer
        self.lstm = nn.LSTM(input_size=embed_dim, hidden_size=hidden_dim,
num_layers=num_LstmLayer, batch_first=True)
        # 4. Dropout layer (moved after LSTM)
        self.dropout = nn.Dropout(p=dropout_rate)
        # 5. First fully connected layer
        self.Linear1 = nn.Linear(hidden_dim, fc1_dim)
        # 6. Second fully connected layer
        self.Linear2 = nn.Linear(fc1_dim, 5)

        # For residual connection in the fully connected layers
        if hidden_dim != fc1_dim:
            self.residual_fc = nn.Linear(hidden_dim, fc1_dim)
        else:
            self.residual_fc = None

    def forward(self, x):
        x = pad_sequence(x, batch_first=True, padding_value=0)

        # Initialize hidden state and cell state
        h = torch.zeros((self.num_LstmLayer, x.size(0), self.hidden_dim)).to(x.device)
        c = torch.zeros((self.num_LstmLayer, x.size(0), self.hidden_dim)).to(x.device)

        # 1. Embedding layer
        out = self.embedding(x)
        # 2. Nrom after Embedding
        out = self.embed_norm(out)
        # 3. LSTM layer
        out, (hidden, cell) = self.lstm(out, (h, c))
        # 4. Layer Normalization and Dropout
        out, _ = torch.max(out, dim=1)
        out = self.dropout(out)

        # Residual connection starts here
        # 5. First fully connected layer with possible residual
        residual = out  # Save for the residual connection
        out = self.Linear1(out)
        out = torch.relu(out)

        if self.residual_fc is not None:
            residual = self.residual_fc(residual)  # Make dimensions match if needed
        out = out + residual  # Add the residual connection
```

```
        # 6. Second fully connected layer
        out = self.Linear2(out)

        return out

LSTM_best_params = checkpoint['best_params']
Best_model = LSTMClassifier(
    vocab_size=LSTM_best_params['vocab_size'],
    embed_dim=LSTM_best_params['embed_dim'],
    hidden_dim=LSTM_best_params['hidden_dim'],
    num_LstmLayer=LSTM_best_params['num_LstmLayer'],
    dropout_rate=LSTM_best_params['dropout_rate'],
    fc1_dim = LSTM_best_params['fc1_dim']
)

Best_model.load_state_dict(checkpoint['model_state_dict'])

"""# 2.0 Predict"""

vocabulary = checkpoint['vocabulary']

vocabulary.set_default_index(vocabulary["<unk>"])

competition = competition[['content']]

nlp = spacy.load('en_core_web_lg') #create an object
def tokenizer(text):
    text = re.sub(r'#(\S+)', 'xxhashtag ' + r'\1', text) # hashtag

    text = re.sub(r'\s{2,}', ' ', text)

    doc = nlp(text)

    tokens = []

    for token in doc:

        word = token.lemma_.lower()

        if not token.is_stop:

            if word == '!':
                tokens.append('!')
```

```
            elif (not token.is_punct) and word != '':
                tokens.append(word)

    return tokens

competition.iloc[:,0] = competition.iloc[:,0].apply(tokenizer)

competition['content'] = competition['content'].apply(lambda x: ' '.join(x))

class TextDataset_competition(utils.data.Dataset):
    def __init__(self, myData):
        """
        myData: a dataframe object containing only X (input data).
        """
        super().__init__()
        self.data = myData

    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        return self.data.iloc[idx]

competition_torch = TextDataset_competition(competition['content'])

tokenizer = get_tokenizer('basic_english')

def doc_tokenizer(doc):
    return torch.tensor([vocabulary[token] for token in tokenizer(doc)],
dtype=torch.long)

def collate_batch_competition(batch):

    text_list = []

    # loop through all samples in batch
    for idx in range(len(batch)):

        _text = batch[idx]
        tokens = doc_tokenizer( _text )
        text_list.append(tokens)

    # convert to torch tensor

    return text_list
```

```
batchSize = 16
competition_loader = utils.data.DataLoader(competition_torch, batch_size=batchSize,
shuffle=False, collate_fn=collate_batch_competition)

def predict_competition(model, data_loader):
    model.eval()  # evaluate
    all_preds = []  # save the classes
    all_probs = []  # save the probability
    with torch.no_grad():
        for texts in data_loader:
            outputs = model(texts)  # forward propagation
            probabilities = F.softmax(outputs, dim=1)  # calculate probabilities through
softmax
            predicted_classes = torch.argmax(probabilities, dim=1)  # get the class(the
max probabilities)

            all_preds.extend(predicted_classes.numpy())  # save the class into all_preds
            all_probs.extend(probabilities.numpy())  # save the probability into
all_probs

    return np.array(all_preds), np.array(all_probs)  # return category and probability

predictions, probabilities = predict_competition(Best_model, competition_loader)

predictions

category_to_label = {
    'business': 0,
    'entertainment': 1,
    'politic': 2,
    'sport': 3,
    'tech': 4
}
label_to_category = {v: k for k, v in category_to_label.items()}

ori_categories = [label_to_category.get(label) for label in predictions]

print(ori_categories)

predictions_df = pd.DataFrame({
    'ID': range(len(ori_categories)),  # Create an ID column, sorted in order
    'category': ori_categories  # the predicted class (category)
})
```

```
predictions_df.head(30)

output_path = '/content/drive/MyDrive/6850/Group09QBUS6850_2024S2.csv'
# save datafrme into csv file
predictions_df.to_csv(output_path, index=False)

# -*- coding: utf-8 -*-
"""Group09QBUS6850_BERT_2024S2

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1tS4UWeZ79SdagTBCsuPTXjZXBWdxFP48

# This program is our BERT part. Please run this program with GPU.

# Import data
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install optuna
!pip install shap
!pip install optuna-integration[xgboost]
!pip install torch==2.1.0 torchvision==0.16.0 torchtext==0.16.0

import warnings
warnings.filterwarnings('ignore')

import pickle
import json
import re
import shap
import torch
import spacy
import random
import numpy as np
import pandas as pd
import seaborn as sns
from wordcloud import WordCloud
import matplotlib.pyplot as plt
from collections import Counter
```

```
import optuna
import xgboost as xgb
import optuna.visualization as vis
from sklearn.decomposition import TruncatedSVD
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, Normalizer,
LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, classification_report, log_loss

from gensim.models import Word2Vec
import torch.nn as nn
import torch.optim as optim
from torch import utils
from torchtext.data.utils import get_tokenizer
from torch.utils.data import DataLoader, Dataset
from torchtext.vocab import build_vocab_from_iterator
import collections
from torch.nn.utils.rnn import pad_sequence

checkpoint = torch.load('/content/drive/MyDrive/6850/checkpoint_for_BERT.pth')

data = pd.read_csv('/content/drive/MyDrive/6850/news-dataset.csv',sep='\t')

seed_value = checkpoint['seed_value']
random.seed(seed_value)
np.random.seed(seed_value)
torch.manual_seed(seed_value)
torch.cuda.manual_seed(seed_value)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.set_rng_state(checkpoint['rng_state'])
torch.cuda.set_rng_state(checkpoint['cuda_rng_state'])
np.random.set_state(checkpoint['numpy_rng_state'])
random.setstate(checkpoint['random_state'])

"""# 1. Data preprocessing"""

data
```

```
data.describe()

data.info()

data.nunique()

dup_content = data[data.duplicated(subset='content', keep=False)]
dup_content.head(20)

content_clean = data.drop_duplicates(subset='content')
content_clean.describe()

title_dup = content_clean[content_clean.duplicated(subset='title', keep=False)]
title_dup.head(20)

"""# 2.0 DISTRIBERT


"""

from transformers import DistilBertTokenizer, DistilBertModel
from torch.utils.data import DataLoader, TensorDataset

index_train_b, index_test_b = train_test_split(content_clean.index,
stratify=content_clean['category'], train_size=0.7, random_state=42)
train_b = content_clean.loc[index_train_b, :].copy()
test_b = content_clean.loc[index_test_b, :].copy()

index_train1_b, index_valid1_b = train_test_split(train_b.index,
stratify=train_b['category'], train_size=0.7, random_state=42)
sub_train_b = train_b.loc[index_train1_b, :].copy()
sub_valid_b = train_b.loc[index_valid1_b, :].copy()

train_for_nn = sub_train_b.copy()
val_for_nn = sub_valid_b.copy()
test_for_nn = test_b.copy()

le = LabelEncoder()

train_for_nn['category'] = train_for_nn['category'].apply(lambda x: x[0])
val_for_nn['category'] = val_for_nn['category'].apply(lambda x: x[0])
test_for_nn['category'] = test_for_nn['category'].apply(lambda x: x[0])

train_target_br = le.fit_transform(train_for_nn['category'])
```

```
val_target_br = le.transform(val_for_nn['category'])
test_target_br = le.transform(test_for_nn['category'])


train_data_b = train_for_nn[['category', 'content']]
train_data_b['category'] = train_target_br

val_data_b = val_for_nn[['category', 'content']]
val_data_b['category'] = val_target_br

test_data_b = test_for_nn[['category', 'content']]
test_data_b['category'] = test_target_br

class TextDataset(utils.data.Dataset):
    def __init__(self, myData):
        super().__init__()
        self.data = myData

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return (self.data.iloc[idx, 0], self.data.iloc[idx, 1])

tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
distilbert_model = DistilBertModel.from_pretrained('distilbert-base-uncased')

train_texts = [str(text) for text in train_data_b['content']]
val_texts = [str(text) for text in val_data_b['content']]
test_texts = [str(text) for text in test_data_b['content']]

# encoding text
train_encodings = tokenizer(train_texts, truncation=True, padding=True,
max_length=512, return_tensors="pt")
val_encodings = tokenizer(val_texts, truncation=True, padding=True, max_length=512,
return_tensors="pt")
test_encodings = tokenizer(test_texts, truncation=True, padding=True, max_length=512,
return_tensors="pt")

# TensorDataset
train_labels = torch.tensor(train_data_b['category'].values)
val_labels = torch.tensor(val_data_b['category'].values)
test_labels = torch.tensor(test_data_b['category'].values)
```

```
train_dataset = TensorDataset(train_encodings['input_ids'],
train_encodings['attention_mask'], train_labels)
val_dataset = TensorDataset(val_encodings['input_ids'],
val_encodings['attention_mask'], val_labels)
test_dataset = TensorDataset(test_encodings['input_ids'],
test_encodings['attention_mask'], test_labels)

def collate_batch_advanced(batch):
    input_ids_list, attention_mask_list, target_list = [], [], []

    for input_ids, attention_mask, label in batch:
        input_ids_list.append(input_ids)
        attention_mask_list.append(attention_mask)
        target_list.append(label)

    input_ids = torch.stack(input_ids_list)
    attention_masks = torch.stack(attention_mask_list)
    target_list = torch.tensor(target_list)

    return input_ids, attention_masks, target_list

tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = DistilBertModel.from_pretrained('distilbert-base-uncased')

batchSize = 16
max_length = 512


train_loader = utils.data.DataLoader(train_dataset, batch_size=batchSize,
shuffle=True, collate_fn=collate_batch_advanced)
val_loader = utils.data.DataLoader(val_dataset, batch_size=batchSize, shuffle=False,
collate_fn=collate_batch_advanced)
test_loader = utils.data.DataLoader(test_dataset, batch_size=batchSize, shuffle=False,
collate_fn=collate_batch_advanced)

class SentimentClassifier(nn.Module):
    def __init__(self, hidden_size, dropout_rate, num_classes=5):
        super(SentimentClassifier, self).__init__()
        self.bert = DistilBertModel.from_pretrained('distilbert-base-uncased')
        for param in self.bert.parameters():
            param.requires_grad = False

        self.fc1 = nn.Linear(self.bert.config.hidden_size, hidden_size)
        self.relu = nn.ReLU()
```

```
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.last_hidden_state[:, 0]
        x = self.fc1(pooled_output)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

def train_model(model, train_loader, val_loader, learning_rate, weight_decay,
trial_number=None):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")
    model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=weight_decay)

    best_val_accuracy = 0.0
    best_val_loss = float('inf')
    patience = 10
    early_stop_counter = 0

    for epoch in range(30):
        model.train()
        total_train_loss = 0
        for input_ids, attention_mask, labels in train_loader:
            input_ids, attention_mask, labels = input_ids.to(device),
attention_mask.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(input_ids, attention_mask)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()


        average_train_loss = total_train_loss / len(train_loader)
```

```
        model.eval()
        total_val_loss = 0
        correct_predictions = 0
        total_samples = 0

        with torch.no_grad():
            for input_ids, attention_mask, labels in val_loader:
                input_ids, attention_mask, labels = input_ids.to(device),
attention_mask.to(device), labels.to(device)
                outputs = model(input_ids, attention_mask)
                loss = criterion(outputs, labels)
                total_val_loss += loss.item()


                _, preds = torch.max(outputs, dim=1)
                correct_predictions += (preds == labels).sum().item()
                total_samples += labels.size(0)


        average_val_loss = total_val_loss / len(val_loader)
        val_accuracy = correct_predictions / total_samples

        print(f'Epoch [{epoch + 1}/30], Train Loss: {average_train_loss:.4f}, Valid
Loss: {average_val_loss:.4f}, Valid Accuracy: {val_accuracy:.4f}')


        if val_accuracy > best_val_accuracy:
            best_val_accuracy = val_accuracy
            best_val_loss = average_val_loss
            torch.save(model.state_dict(), f'best_model_trial_{trial_number}.pth')
            early_stop_counter = 0
        else:
            early_stop_counter += 1
            if early_stop_counter >= patience:
                print("Early stopping triggered")
                break

    return best_val_loss, best_val_accuracy

def objective(trial):

    hidden_size = trial.suggest_int('hidden_size', 16, 32)
    dropout_rate = trial.suggest_float('dropout_rate', 0.1, 0.4)
    learning_rate = trial.suggest_loguniform('learning_rate', 1e-4, 1e-2)
```

```
    weight_decay = trial.suggest_loguniform('weight_decay', 1e-5, 1e-2)


    model = SentimentClassifier(hidden_size=hidden_size, dropout_rate=dropout_rate)


    best_val_loss, best_val_accuracy = train_model(model, train_loader, val_loader,
learning_rate, weight_decay, trial_number=trial.number)


    return best_val_accuracy

# Optuna study

# study = optuna.create_study(direction='maximize')
# study.optimize(objective, n_trials=50)


# print("Best hyperparameters: ", study.best_params)
# print("Best validation accuracy: ", study.best_value)

"""##2.1 Load pretrain BERT

"""

BERT_best_params = checkpoint['best_params']

BERT_before_concat = SentimentClassifier(
    hidden_size=BERT_best_params['hidden_size'],
    dropout_rate=BERT_best_params['dropout_rate'],
)
train_loader = DataLoader(train_dataset, batch_size=BERT_best_params['BATCH_SIZE'],
shuffle=True, collate_fn=collate_batch_advanced)
val_loader = DataLoader(val_dataset, batch_size=BERT_best_params['BATCH_SIZE'],
shuffle=False, collate_fn=collate_batch_advanced)

"""##2.2 Train the model"""

class EarlyStopping:
    def __init__(self, patience=15, verbose=False, min_delta=0.0003):
        """
        Args:
            patience (int): The number of epochs allowed when validation loss is not
improving.
```

```
            verbose (bool): If True, print the information.
            min_delta (float): the minimum threhold in improving the validation loss, if
less than the threshold identify as no improvement
        """
        self.patience = patience
        self.verbose = verbose
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        if self.best_loss is None or val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
        else:
            self.counter += 1  # If no improvement, add 1 to counter
            if self.counter >= self.patience:
                self.early_stop = True  # early stopping
                if self.verbose:
                    print("Early stopping triggered")

def train_and_validate_BERT(model, train_loader, valid_loader, learning_rate,
weight_decay):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=weight_decay)
    valid_losses = []
    valid_accuracies = []
    train_losses = []
    early_stopper = EarlyStopping(patience=15, verbose=True)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)

    for epoch in range(50):
        model.train()
        train_loss = 0.0  # Initialize train loss for each epoch
        total_samples = 0

        for batch in train_loader:
            optimizer.zero_grad()
```

```
            input_ids, attention_mask, labels = batch['input_ids'].to(device), 
batch['attention_mask'].to(device), batch['labels'].to(device)


            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * labels.size(0)  # Multiply loss by batch size
            total_samples += labels.size(0)

        average_train_loss = train_loss / total_samples
        train_losses.append(average_train_loss)  # Append average loss for the epoch


        model.eval()
        y_pred_val = []
        y_val = []
        val_loss = 0.0
        with torch.no_grad():
            for batch in valid_loader:
                input_ids, attention_mask, labels = batch['input_ids'].to(device), 
batch['attention_mask'].to(device), batch['labels'].to(device)

                outputs = model(input_ids=input_ids, attention_mask=attention_mask)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, preds = torch.max(outputs, 1)
                y_pred_val.extend(preds.cpu().numpy())
                y_val.extend(labels.cpu().numpy())

        val_accuracy = (np.array(y_pred_val) == np.array(y_val)).mean() * 100
        valid_losses.append(val_loss / len(valid_loader))
        valid_accuracies.append(val_accuracy)

        print(f'Epoch [{epoch+1}/50], Train Loss: {train_losses[-1]:.4f}, Valid Loss: 
{valid_losses[-1]:.4f}, Valid Accuracy: {valid_accuracies[-1]:.2f}%')

        early_stopper(val_loss / len(valid_loader), model)
        if early_stopper.early_stop:
            print(f"Early stopping at epoch {epoch + 1}")
            break
```

```
    return max(valid_accuracies), train_losses, valid_losses

#This is our train process.

# _, train_losses, valid_losses = train_and_validate_BERT(
#     BERT_before_concat, train_loader, val_loader,
#     BERT_best_params['learning_rate'], BERT_best_params['weight_decay'])

import torch.nn.functional as F
def predict(model, data_loader):
    model.eval()  # swicth to val model
    all_preds = []
    all_probs = []
    all_labels = []

    with torch.no_grad():
        for input_ids, attention_masks, labels in data_loader:

            outputs = model(input_ids=input_ids, attention_mask=attention_masks)


            probabilities = F.softmax(outputs, dim=1)


            predicted_classes = torch.argmax(probabilities, dim=1)


            all_preds.extend(predicted_classes.cpu().numpy())
            all_probs.extend(probabilities.cpu().numpy())
            all_labels.extend(labels.cpu().numpy().flatten())


    return np.array(all_preds), np.array(all_probs), np.array(all_labels)

"""##2.3 The final test result"""

BERT_before_concat.load_state_dict(checkpoint['model_state_dict'])

y_pred, y_probs, all_labels = predict(BERT_before_concat, val_loader)

accuracy = accuracy_score(all_labels, y_pred)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
# This is the train and validation loss plot. Since we don't train the model, instead
we load the pre-trained BERT, we cant draw the plot.

# x_axis = range(0, len(train_losses))
# fig, ax = plt.subplots()
# ax.plot(x_axis, train_losses, label='Training Loss')
# ax.plot(x_axis, valid_losses, label='Validation Loss')
# ax.set_xlabel('Epochs')
# ax.set_ylabel('Loss')
# ax.set_title('BERT Train and Validation Loss per Epoch')
# ax.legend()
# plt.show()

print(classification_report(all_labels, y_pred, digits=3))

"""# 3.0 Save the model and random state"""

torch.save({
    'seed_value': seed_value,
    'model_state_dict': BERT_before_concat.state_dict(),
    'best_params': BERT_best_params,
    'rng_state': torch.get_rng_state(),
    'cuda_rng_state': torch.cuda.get_rng_state(),
    'numpy_rng_state': np.random.get_state(),
    'random_state': random.getstate()
}, '/content/drive/MyDrive/6850/checkpoint_for_BERT.pth')
```