Comparative Analysis of DeBERTaV3, LSTM with BERT Word Embeddings, and SVM in Propaganda Detection

Kemal Bayik

Abstract

In this study, three different language models were run in two different propaganda detection tasks and the results were compared. The models used are DeBERTaV3, LSTM with BERT Word Embeddings (LBWE) and SVM. By selecting these models, a transformer-based model, a model whose word embedding is provided by a transformer-based model but uses LSTM in classification, and a traditional machine learning model that works with word2vec word embeddings were compared. The first task to which these models are applied is to classify whether the sentence contains propaganda or not. The second task is to classify which type of propaganda the sentences containing propaganda contain. In the results obtained, while DeBERTaV3 showed the best performance in the first task, the LBWE model showed the best performance in the second task. In both tasks, SVM achieved the lowest F1 and accuracy scores.

1. Introduction

Propaganda is a set of pre-planned messages intended to influence the thoughts and behaviors of a large number of people. Propaganda presents information from a one-sided perspective and tries to turn people's perspective in that direction. It is a very dangerous situation that this propaganda does not reflect the facts and manipulates information. This situation can mislead the public, polarize society and harm democracy.

Nowadays, the internet and social media occupy a large part of people's lives. In this way, people can easily access information that was difficult to access in the past. However, they are also exposed to propaganda much more than before. For this reason, propaganda detection has become much more important than in the past.

A lot of work is being done to detect propaganda in the field of NLP [1][2][3][4], but this task is quite challenging. There are several reasons for this. The first of these is that the language is very complex and words have multiple meanings. This makes detection of propaganda quite difficult, especially considering that propaganda generally uses figurative meanings. Another reason is that propaganda can be customized according to people's beliefs and tendencies. In this case, the success of models prepared according to general propaganda methods will be low. The last reason that can be mentioned is that propaganda techniques are constantly changing.

This study offers solutions to two different tasks regarding propaganda detection. The first task is to determine whether the given sentences contain propaganda. The second task is to determine what type of propaganda the sentences containing propaganda contain. A total of 3 models were created and these 3 models were tested in two tasks. The first of these three models is DeBERTaV3 [5], the second is LSTM with BERT Word Embeddings (LBWE) and the third is Support Vector Machine (SVM). The architectures and working principles of these models are explained in detail in the Model Architecures section.

2. Model Architectures

In this section of the report, the model architectures used in Task-1 and Task-2 are explained. The three methods described in this section were used in both tasks.

First of all, I will give information about how the DebertaV3 model works. In order to better explain the DeBERTaV3 model, the structure of the Transformer, BERT and DeBERTa model will be mentioned before examining the detailed structure of the DeBERTaV3 model.

2.1. Transformer

Transformer[6] is a deep learning architecture introduced in 2017 and based on the multi-head self-attention mechanism. The self-attention mechanism helps understand the relationship of each word with other words in a sentence and multi head self attention performs more than one self attention process in parallel at the same time and this architecture allows the model to learn different features from the same input and create richer contextual representations. In addition, Transformer models use embedding vectors for each word or token, but these embeddings do not include the order of the words. Therefore, positional encoding is used to help the model understand word order and syntactic context.

2.2. BERT

BERT (Bidirectional Encoder Representations from Transformers) [7] is a transformer-based language model introduced by Google in 2018. It uses only the encoder block of the transformer model. Each layer has two sub-layers. These are the multi-head self-attention layer and the position-wise fully connected layer. BERT uses positional encoding to keep track of the order of words.

BERT uses Next Sentence Prediction (NSP) or Masked Language Model (MLM) during the training phase. In the NSP method, the model is given two sentences and tries to predict whether the second sentence is a continuation of the first sentence. In the MLM method, randomly selected tokens from the text given as input are masked with a certain probability and the model tries to predict the masked token. During this prediction, the model takes into account the words to the right and left of the masked token. This is one of the mechanisms that provide the bidirectional feature of BERT. In addition, the fact that the self-attention mechanism calculates the attention score of a token by using all other tokens is another factor that provides this bidirectional feature.

2.3. DeBERTa

DeBERTa (Decoding-enhanced BERT with disentangled attention) is an NLP model introduced by Microsoft in 2020 [8]. The aim of DeBERTa is to improve the BERT and RoBERTa models and they used two new techniques for this. These methods are disentangled attention mechanism and an enhanced mask decoder.

2.3.1. Disentangled Attention

In Deberta, words are represented by two vectors. These are content and position vectors. Attention weights of words are calculated using disentangled matrices according to their content and position, respectively. The reason for using this method is that the relationships between words do not only depend on the content of the words, but also the positions of the words relative to each other are effective in this relationship. [8]

2.3.2. Enhanced Mask Decoder

DeBERTa uses MLM like BERT during the training phase, but in addition to BERT, the enhanced mask decoder method is also used. This means that, method incorporates the absolute position embeddings of the words in all transformer layers, just before the softmax layer. Assuming that the disentangled attention mechanism takes into account the content and relative positions of the words, it is of great importance to know the absolute positions of the words.

2.4. DeBERTaV3

The DeBERTaV3 model is a pre-trained language model introduced by Microsoft in 2021 and improves the DeBERTa model. DeBERTaV3 uses two new techniques over DeBERTa. The first of these is ELECTRA-style training, that is, using replaced token detection (RTD) instead of masked language modeling (MLM). The second is the new gradient-disentangled embedding sharing method (GDES).

Algorithm 1 DeBERTaV3 Fine Tuning

```
Load tokenizer and model with DeBERTa-v3-base
function Tokenize(data)
    return Tokenize data, pad to max length, truncate
end function
Map tokenize function to train and test datasets
Define batch size and epochs
Setup training arguments
Initialize Trainer with model, datasets, and arguments
Train the model
```

2.4.1. Replaced Token Detection

Replaced token detection was first introduced and used in the ELECTRA model in 2020 [9]. A model using RTD uses two transformer encoders. The first of these is the generator and the second is the discriminator. Generator is trained with MLM, just like BERT. Discriminator, on the other hand, is trained with a token-level binary classifier. The generator tries to predict the masked words in an input, and this output is given as input to the discriminator. In this way, some of the masked words will be correct and some will be incorrect. Discriminator, on the other hand, tries to determine whether the words in the input it receives belong to the original text or have been changed by the generator.

2.4.2. Gradient-Disentangled Embedding Sharing

While the output produced in the generator is transferred to the discriminator, it passes through an embedding layer used jointly by the generator and discriminator. This is called token embedding sharing (ES). In the RTD used in DeBERTaV3, certain changes were made to ES and new gradient-disentangled embedding sharing method (GDES) was introduced. GDES is used jointly by the generator and discriminator, like the ES used in ELECTRA. The difference is that in ES, token embeddings are updated by balancing the gradients of the two tasks, while in GDES, the gradients of the generator and discriminator are kept separate from each other. In this way, both generator and discriminator can optimize the learning process and speed independently of the influence of the other layer.

2.4.3. Fine Tuning Process of DeBERTaV3

In this project, microsoft/deberta-v3-base was used as the pretrained DeBERTaV3 model. The model and tokenizer were created using Hugging Face's transformers library [10]. In addition, the TrainingArguments function of the transformer library was used to set the hyperparameters of the model, and the Trainer function of the transformer library was used to fine tune the model. Cross entropy loss was used as the loss function. The code was written with assistance from Hugging Face's "Fine-tune a pretrained model" article[11]. The pseudocode of

the training and evaluation phases of DeBERTaV3 is shown in Algorithm 1.

2.4.4. Hyperparameters of DeBERTaV3

While selecting hyperparameters, the hyperparameters shared in the article where DeBERTaV3 was introduced were taken into account and they were kept the same in both tasks. 8,16 and 32 batch sizes were tried. Additionally, the model with each batch size was fine tuned for 5 and 10 epochs and the results were obtained. Also the learning rate is not constant in the model. It was initialized as 2e-5 and Adam optimizer was used to optimize the learning rate and the model was not fine tuned for different learning rate values. Constant hyperparameters are as follows.

- Weight Decay: 0.01
- Warmup Steps: 0
- Max Sentence Length: 128. When values larger than 128 are used, the training time of the model becomes considerably longer. Since the average length of the sentences in the data sets used in the two tasks is approximately 20, I think the value of 128 is reasonable.

2.5. LSTM with BERT Word Embeddings (LBWE)

As the second approach in the project, LSTM with BERT word embeddings was used. In this model, BERT's task is to create word embeddings. The created word embeddings (last_hidden_state) are given as input to LSTM. The task of LSTM is to make the classification.

Algorithm 2 LSTM with BERT Word Embeddings

function __INIT__(*self*, *n_classes*)
    Initialize the base class (nn.Module)
    Define self.bert as pre-trained BERT model
    Define self.drop as dropout layer with dropout rate of 0.3
    Define self.lstm as LSTM layer with input size from BERT and output size 128
    Define self.out as linear layer from 128 to *n_classes*
end function
function FORWARD(*self*, *input_ids*, *attention_mask*)
    Compute BERT outputs for *input_ids* and *attention_mask*
    Extract the last hidden state from BERT outputs
    Pass the last hidden state through the LSTM layer
    Apply dropout to the output from the LSTM layer
    Squeeze the output from the LSTM layer to remove unnecessary dimensions
    Compute final output by passing LSTM output through linear layer
    return the final output
end function

The reason why BERT is used to produce word embeddings in this model is that it has advantages over methods that produce static word embeddings such as word2vec or GloVE. One of these is that BERT creates contextual embeddings. The second is that BERT can be fine tuned in every epoch. Although not shown in this report, LSTM was first tested with word2vec word embeddings, and accuracy and F1 scores remained below 50 percent for Task-2. Considering this, it can be said that BERT exhibits superior performance compared to static word embedding methods.

LSTM(Long Short-Term Memory) was introduced by Hochreiter and Shcmidhuber in 1997[12] and is a member of the RNN (Recurrent Neural Networks) family and is specifically designed to model long-term dependencies. It uses a special structure to solve the vanishing gradient problem that RNN encounters during training. In this structure, LSTM consists of cells, and these cells contain the cell state and three gates. These are the forget gate, input gate and the output gate. The forget gate decides what information to delete from the cell state, the input gate decides what new information to add to the cell state, and the output gate decides which output to send to the next layer. The advantages of LSTM are that it can learn long-term dependencies and successfully understand the context of the sentence. The reason why LSTM was chosen in this project is its widespread use in NLP and these advantages. LSTM architecture is shown in Figure-1.

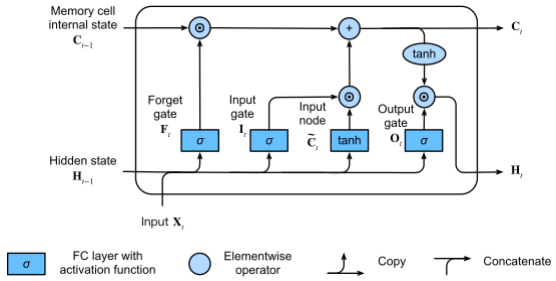


Figure 1: LSTM Architecture [15]

2.5.1. Training Process of LBWE

In this model, bert-base-uncased was used as the pretrained BERT model. The tokenizer and the model were loaded using the transformers library. Before training the model, the data is first processed with the BERT tokenizer. During training of the model, the data is first passed through the BERT layer and BERT produces an embedding vector for each token. These embedding vectors are passed to LSTM. LSTM processes the data and obtains the final cell state. This cell state is given to the dropout layer and then to a linear layer and the output is obtained. Cross entropy loss was used as the loss function. The code was written with assistance from this PyTorch article [13], this Kaggle code [14] and Hugging Face's "Fine-tune a pretrained model" article [11]. The pseudocode of the LBWE model is shown in Algorithm 2.

2.5.2. Hyperparameters of LBWE

LBWE's hyperparameters were kept the same in both tasks. 16,32,48 and 64 batch sizes were tried. Additionally, the model with each batch size was trained for 5 and 10 epochs and the results were obtained. Also the learning rate is not constant in

the model. It was initialized as 2e-5 and Adam optimizer was used to optimize the learning rate and the model was not trained for different learning rate values. Constant hyperparameters are as follows.

- Weight Decay: 0
- Dropout Rate: 0.3
- Warmup Steps: 0
- Max Sentence Length: 128. Also in this model when values larger than 128 are used, the training time of the model becomes considerably longer. Since the average length of the sentences in the data sets used in the two tasks is approximately 20, I think the value of 128 is reasonable.
- Number of LSTM layers: 1
- LSTM Hidden Size: 128

2.6. SVM (Support Vector Machines)

SVM is a supervised learning model that is widely used in classification and regression problems. It creates a plane that separates the classes in the data set and aims to have this plane at the maximum distance from all data. Sometimes, data points cannot be separated just by a straight line. In such cases, SVM can still be effective by using something called the "kernel trick". Kernel trick makes linear separation possible by transforming the data points to higher dimensions. To do this, it multiplies the data with certain kernel functions.

Word embeddings were given as input to SVM and Word2vec was used to obtain these word embeddings. Word2vec was developed by Tomáš Mikolov and his team at Google in 2013 and is a method for converting words into vectors [16]. It consists of two basic structures. The first of these is Continuous Bag of Words (CBOW) and the second is Skip-gram. The task of CBOW is to try to guess the token by looking at the words around the token. Skip-gram, on the other hand, is the opposite of CBOW and tries to guess the words around token based on the token.

2.6.1. Training Process of SVM

First of all, the inputs are normalized. For the normalization process, tokens were converted to lowercase, punctuation marks were removed, and stop words were removed. Afterwards, word embeddings were obtained from the normalized tokens using Word2vec. These resulting word embeddings were given as input to SVM. "GoogleNews-vectors-negative300.bin" was used as the Word2vec model. The SVM model was coded with assistance from this article[17] and this Kaggle code [18]. The pseudocode of the SVM model is shown in Algorithm 3.

Algorithm 3 SVM

```
function PreprocessText(text)
    Convert text to lowercase
    Remove punctuation from text
    Remove stopwords from text
    return processed text
end function
function NormaliseDataframes()
    Apply PreprocessText(text) to train and test dataframes
end function
function CreateSentenceVector(word2vec_model, sentence)
    Split sentence into words
    Vectorize words using word2vec_model
    return average of vectors
end function
function CreateTrainTest()
    NormaliseDataframes()
    Create sentence vectors for all documents
    Prepare x_train, y_train, x_test, y_test
    return x_train, y_train, x_test, y_test
end function
function Train(x_train, y_train, x_test, y_test)
    Setup GridSearch with parameters
    Perform GridSearch on training data
    return GridSearch results
end function
```

2.6.2. Hyperparameters of SVM

SVM hyperparameters were optimized separately in two tasks. Hyperparameter optimization was done using GridSearchCV[19]. GridSearchCV uses the cross validation method and the model is trained for each combination. Thus, the performance of the model is calculated for all parameter sets. GridSearchCV's cv parameter represents the number of folds and is fixed at 5. All tried SVM hyperparameters are as follows.

- C: 0.1, 1, 10, 100
- kernel: linear, rbf, poly, sigmoid
- gamma: scale, auto, 0.1, 1, 10

3. Data Preperation

3.1. Dataset

The data set used in this project consists of sentences and the propaganda type labels they contain, or the not_propaganda label if they do not contain propaganda. There are 8 different types of propaganda in total and these are a subset of those identified in the Propaganda Techniques Corpus [1]. These types of propaganda are as follows;

- flag waving
- appeal to fear prejudice
- causal simplification
- doubt

- exaggeration, minimization
- loaded language
- name calling, labeling
- repetition

One-hot encoding was applied to these labels during training of all models. There are 2560 data in total in the train set and 640 data in the test set. A part of each sentence is enclosed in <BOS>and <EOS>tags. The part between these tags represents the part of the sentence that contains propaganda. These tags are also found in sentences that do not contain propaganda. The words between these tags do not mean anything in these sentences. Since DeBERTaV3 and BERT's tokenizers apply normalization to the texts, no additional normalization is applied to DeBERTaV3 and LBWE.

3.2. Task-1

Since the aim of the first task is to find out whether the sentence contains propaganda or not, it does not matter which type of propaganda the sentence contains. For this reason, propaganda types labels (flag_waving, doubt, etc.) have been changed to "propaganda". Additionally, <BOS>and <EOS>tags have been removed from the sentences. The minimum, maximum and average word counts of sentences in the data sets are shown in Table 1. In addition, the class distributions in the training and the test datasets can be seen in Figure 2.

Dataset	Minimum	Maximum	Average
Train	1	142	25.80
Test	2	116	24.4

Table 1: Task-1 Train and Test Set Word Counts

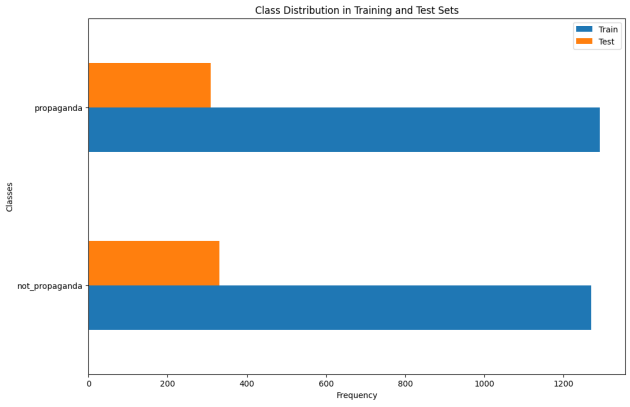


Figure 2: Task-1 Training and Test Sets Class Distribution

3.3. Task-2

Since the purpose of Task 2 is to find out which propaganda type the sentences contain, only words between the <BOS>and <EOS>tags were used. In addition, data with the label not_propaganda was removed from the dataset. Since these data were removed from the data set, the training set of Task-2 contains 1291 data and the test set contains 309 data. The minimum, maximum and average word counts of sentences in the data sets are shown in Table 2. In addition, the class distributions in the training and the test datasets can be seen in Figure 3.

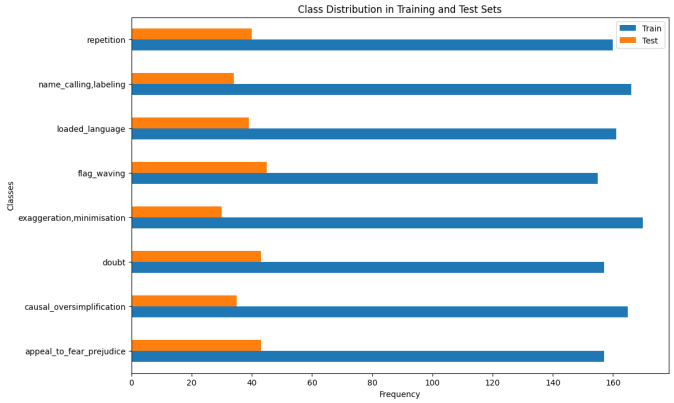


Figure 3: Task-2 Training and Test Sets Class Distribution

Dataset	Minimum	Maximum	Average
Train	1	141	11.04
Test	1	71	11.58

Table 2: Task-2 Train and Test Set Word Counts

4. Results and Analyses

In this section, the results obtained by the models for Task-1 and Task-2 are shown and analyzed.

In this study, accuracy and macro F1 score was used as the performance metrics since there were no unbalanced class distributions in the training sets of the two tasks. Additionally, Weights & Biases were used to store and graph the obtained models and results [20]. Figure 4 shows the F1 scores and confusion matrices obtained by the models in Task-1. Additionally, the train losses of DeBERTaV3 and LBWE models are shown. Figure 5 shows the F1 scores and confusion matrices obtained by the models in Task-2. Additionally, the train losses of DeBERTaV3 and LBWE models are shown.

Model	Task-1 F1 Score	Task-2 F1 Score
DeBERTaV3	0.91	0.6253
LBWE	0.788	0.6361
SVM	0.740	0.502

Table 3: Best F1 scores of the models in Task-1 and Task-2

Model	Task-1 Accuracy	Task-2 Accuracy
DeBERTaV3	0.9103	0.6272
LBWE	0.7914	0.6487
SVM	0.734	0.514

Table 4: Best accuracies of the models in Task-1 and Task-2

4.1. Task-1

Looking at the results of DeBERTaV3 shown in Figure 4/a, Table 3 and Table 4, it can be seen that the version with **32 batch size** and **5 epoch** fine tuning achieved very successful results such as 0.91 F1 and 0.9103 accuracy scores. In addition, if we look at the confusion matrix in Figure 4/g, we can say that the predictions are evenly distributed. In the train loss graph in Figure 4/c, it can be seen that the model is trained stably. Looking at the F1 score graph, it can be said that the model gives better performance in this task with higher batch numbers.

In the results of the LBWE model in Figure 4/b Table 2 and Table 3, it is seen that the version with **64 batch size** and **10 epoch** trained achieved an F1 score of 0.788 and an accuracy score of 0.7914. It is obvious that this result is worse than the results of the DeBERTaV3 model. In addition, when we look at the confusion matrix in Figure 4/f, we see that it predicts the not_propaganda class much better than the propaganda class. This result is also supported by the fact that F1 and accuracy scores are not very close to each other. In the train loss graph in Figure 4/e, it is seen that the decrease of the loss is regular but slower than DeBERTaV3. When looking at the F1 score graph of this model, it is not possible to observe the effect of the number of batches and the number of epochs on the performance of the model in this task.

Looking at the SVM results in Task-1, it is seen that the SVM model achieved an F1 score of 0.740 and an accuracy of 0.734 with the parameters **rbf**, **c = 1** and **gamma = 1**. It is clearly seen that this result is weaker than the other two models. In addition, it can be said that the prediction distributions in the confusion matrix in Figure 4/h are more balanced than those in LBWE.

As a result, it is clear that the most successful model in Task-1 is DeBERTaV3, both in terms of F1 and accuracy score and in terms of balance of prediction distributions. Although it has similar performance to SVM, LBWE ranks second and SVM ranks third. In addition, the imbalance of class distributions of LBWE stands out as a major negative.

4.2. Task-2

Looking at DeBERTaV3's performance in Task-2, it can be seen that the best score it achieved was the version fine tuned in **16 batches** and **10 epochs**, its F1 score was 0.6253 and its accuracy score was 0.6272. In the confusion matrix, there is a balanced distribution in the predictions, as in Task-1. In the train loss chart, it can be seen that the loss decrease is much slower compared to Task-1, and as the batch numbers decrease, the loss decreases faster. Looking at the F1 score graph, it can be said that the model gives better performance in this task with smaller batch numbers and it is not possible to observe the effect of the number of epochs on the performance of the model in this task.

We see that LBWE has the best F1 score of 0.6361 and accuracy score of 0.6487, and these scores were achieved with the version trained with **16 batches** and **5 epochs**. It is possible to say that in the Confusion matrix, as in Task-1, it predicts some classes very well and in others it is quite bad. On the other hand, in the train loss chart in Figure 5/e, it can be easily said that the loss decrease is slower compared to Task-1. By looking at the F1 score graph, it can be said that the model performs better at smaller batch numbers. In addition, although it achieved the best results at 5 epochs, it generally appears to give good performance at high batch numbers.

We can see that SVM achieved its best scores of 0.502 F1 and 0.514 accuracy scores on **rbf**, **c = 10** and **gamma = 0.1** hyperparameters. Based on these results, it would not be wrong to say that the SVM gave weaker performance than the other two models, as in the Task-1. It is obvious that there is a more unbalanced distribution in the Confusion matrix compared to Task-1. Especially the percentage of correct predictions for classes exaggeration, minimization, casual_oversimplification, loaded_language and flag_waving being below 50 can be considered as a negative result.

It is also seen that the models show similar characteristics in predicting classes. For example, all three models successfully predict the appeal_to_fear_prejudice class, but they also predict the "exaggeration,minimization" class just as badly. Although there are similarities in the distributions of the predictions of the three classes, the prediction distributions of LBWE and SVM are closer to each other, while the prediction distributions of DeBERTaV3 are more balanced and less similar to the other two models.

When we look at the results in general, it seems that the best performing model in Task-1 is DeBERTaV3, the best performing model in Task-2 is LBWE, and the worst performing model in both tasks is SVM. The reason why DeBERTaV3 performs worse in Task-2 may be due to its more complex structure. The fact that the data set in Task-2 contains less data than Task-1 and the sentence lengths are shorter may have caused DeBERTaV3 to overfit the data. The opposite situation is valid for Task-1. The fact that DeBERTaV3 is a more complex model may have contributed to its high success in Task-1. There may be several different reasons why SVM is less successful than both models in both tasks. The first of these is word embeddings. DeBERTaV3 and LBWE models better capture the contextual relationships of words. In addition, word2vec is a static model, but the BERT used in this study is fine tuned in every epoch, which seriously affects the performance of the model. In addition, in SVM, data are treated as independent features, that

is, each word is modeled independently. This causes SVM to ignore the contextual relationships of words, and this is one of the biggest reasons for the lower performance of SVM.

5. Conclusion

In this study, three different models were created for two different propaganda detection tasks and the performances of these models in these tasks were compared. The results obtained allowed analyzing the advantages and disadvantages of the models relative to each other. It also gave insight into what types of models could perform better in challenging NLP tasks such as propaganda detection. Nowadays, transformer-based models are much more preferred in solving NLP problems and in new technologies, and they perform better in most cases. In this study, DeBERTaV3, a completely transformer-based model, gave the best results in Task-1, and the LSTM model, whose word embeddings were created with BERT, achieved the most successful results in Task-2. From these results, it can be concluded that transformer based models produce much better word embeddings than static word embedding models like word2vec. They can also classify complex data sets more successfully than deep learning methods and traditional machine learning methods like SVM. However, it should be noted that the fact that DeBERTaV3 performs worse than LBWE in Task-2, may mean that the performance of this model may decrease as the dataset becomes smaller. For this reason, in NLP problems with small data sets, the use of other transformer-based models or models in which the transformer-based model produces only word embeddings, as in this study, should be considered.

6. Further Work

As seen in this study, DeBERTaV3 gives a very successful result in Task-1, that is, in classifying whether a sentence contains propaganda or not. However, it seems that all three models give unsuccessful results in Task-2, that is, in classifying the propaganda type. For this reason, it would be logical to try and improve models that can work more optimally in smaller data sets and classification problems containing more than two classes. In addition, in the results obtained in this project, the models were trained once because the training of the models took too long. However, creating and interpreting graphs by running the models multiple times and averaging the results will be much more consistent and logical. As another study, the number of layers and hidden dimensions of LSTM can be increased and the performance change of the model can be observed. Finally, the SVM model can be fed with BERT word embeddings instead of word2vec word embeddings.

References

[1] Giovanni Da San Martino, Alberto Barron-Cedeno, Henning Wachsmuth, Rostislav Petrov, and Preslav Nakov. 2020. SemEval-2020 task 11: Detection of propaganda techniques in news articles.

[2] Martinkovic, M., Pecar, S., & Šimko, M. (2020, December). NLFIIT at SemEval-2020 Task 11: Neural network architectures for detection of propaganda techniques in news articles. In Proceedings of the Fourteenth Workshop on Semantic Evaluation (pp. 1771-1778).

[3] Jurkiewicz, D., Borchmann, Ł., Kosmala, I., & Graliński, F. (2020). ApplicaAI at SemEval-2020 task 11: On RoBERTa-CRF, span CLS and whether self-training helps them. arXiv preprint arXiv:2005.07934.

[4] Martino, G., Barrón-Cedeño, A., & Nakov, P. (2019). Findings of the NLP4IF-2019 Shared Task on Fine-Grained Propaganda Detection. ArXiv, abs/1910.09982. https://doi.org/10.18653/v1/D19-5024.

[5] He, P., Gao, J., & Chen, W. (2021). Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing. arXiv preprint arXiv:2111.09543."

[6] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

[7] 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.

[8] He, P., Liu, X., Gao, J., & Chen, W. (2020). Deberta: Decoding-enhanced bert with disentangled attention. arXiv preprint arXiv:2006.03654.

[9] Clark, K., Luong, M. T., Le, Q. V., & Manning, C. D. (2020). Electra: Pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555.

[10] https://pypi.org/project/transformers/

[11] https://huggingface.co/docs/transformers/en/training

[12] Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[13] https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html

[14] https://www.kaggle.com/code/christofhenkel/bert-embeddings-lstm

[15] https://d2l.ai/chapter_recurrent-modern/lstm.html

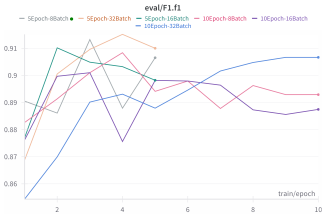
[16] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

[17] https://www.geeksforgeeks.org/svm-hyperparameter-tuning-using-gridsearchcv-ml/

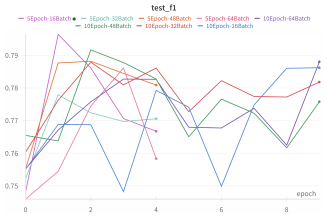
[18] https://www.kaggle.com/code/mehmetlaudatekman/tutorial-word-embeddings-with-svm

[19] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearch

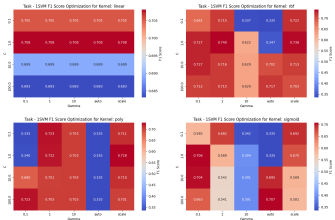
[20] https://wandb.ai/site



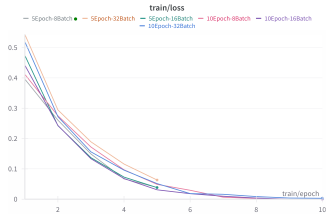
(a) Task-1 DeBERTaV3 F1 Scores



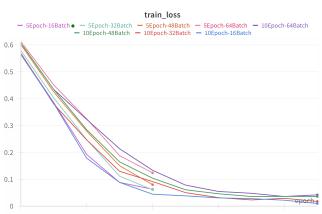
(b) Task-1 LBWE F1 Scores



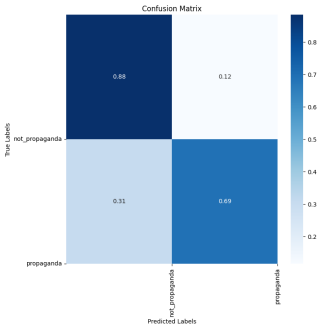
(c) Task-1 SVM F1 Scores



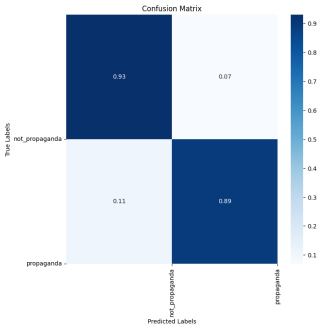
(d) Task-1 DeBERTaV3 Train Loss



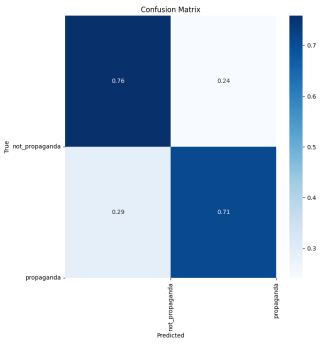
(e) Task-1 LBWE Train Loss



(f) Task-1 LBWE 10 Epoch 64 Batch Size Confusion Matrix

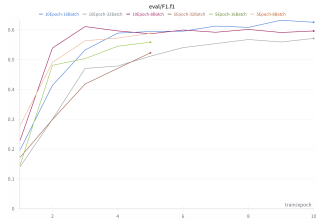


(g) Task-1 DeBERTaV3 5 Epoch 32 Batch Size Confusion Matrix

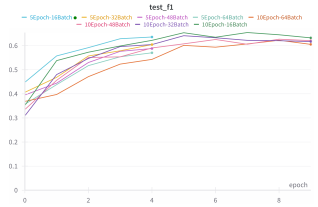


(h) Task-1 SVM C:1 Gamma:1 RBF Kernel Confusion Matrix

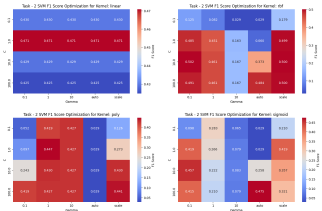
Figure 4: Task-1 Results



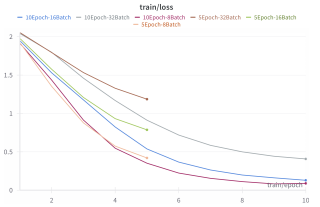
(a) Task-2 DeBERTaV3 F1 Scores



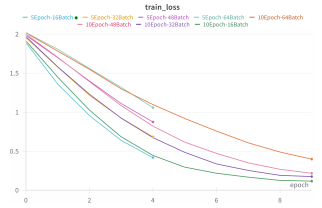
(b) Task-2 LBWE F1 Scores



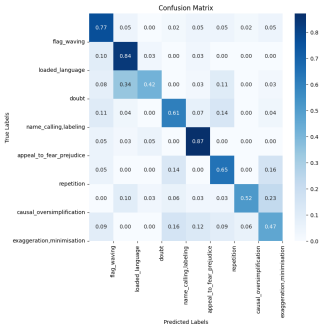
(c) Task-2 SVM F1 Scores



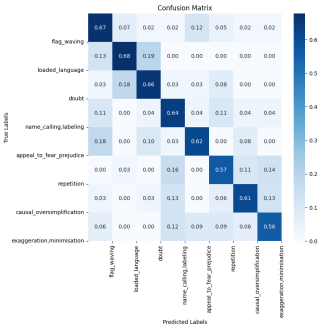
(d) Task-2 DeBERTaV3 Train Loss



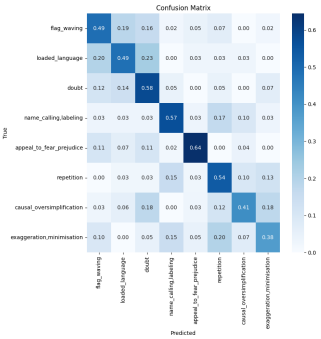
(e) Task-2 LBWE Train Loss



(f) Task-2 LBWE 5 Epoch 16 Batch Size Confusion Matrix



(g) Task-2 DeBERTaV3 10 Epoch 16 Batch Size Confusion Matrix



(h) Task-2 SVM C:10 Gamma:0.1 RBF Kernel Confusion Matrix

Figure 5: Task-2 Results

7. Appendix

Importing libraries

```
import pandas as pd
import numpy as np
import torch
from transformers import BertModel, BertTokenizer, DebertaV2Tokenizer,
DebertaV2ForSequenceClassification, Trainer, TrainingArguments
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
from matplotlib import pyplot as plt
import wandb
import torch.nn as nn
from sklearn.preprocessing import LabelEncoder
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, f1_score
from gensim.models import KeyedVectors
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from datasets import Dataset, load_metric
from tqdm import tqdm
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

nltk.download('punkt')
nltk.download('stopwords')
```

Loading Datasets

Task 1

```
task1_train_df = pd.read_csv('propaganda_train.tsv', delimiter='\t',
quotechar='\'')
task1_train_df.loc[task1_train_df['label'] != "not_propaganda",
'label'] = "propaganda" # All 8 Propaganda labels are changed to
"propaganda" and the total number of classes is reduced to 2
task1_train_df['tagged_in_context'] =
task1_train_df['tagged_in_context'].str.replace('<BOS>', '',
regex=False)
task1_train_df['tagged_in_context'] =
task1_train_df['tagged_in_context'].str.replace('<EOS>', '',
regex=False)

task1_test_df = pd.read_csv('propaganda_val.tsv', sep='\t',
quotechar='\'')
task1_test_df.loc[task1_test_df['label'] != "not_propaganda", 'label']
= "propaganda" # All 8 Propaganda labels are changed to "propaganda"
and the total number of classes is reduced to 2
```

```
task1_test_df['tagged_in_context'] =
task1_test_df['tagged_in_context'].str.replace('<BOS>', '',
regex=False)
task1_test_df['tagged_in_context'] =
task1_test_df['tagged_in_context'].str.replace('<EOS>', '',
regex=False)

label_encoder = LabelEncoder()
task1_train_df['label'] =
label_encoder.fit_transform(task1_train_df['label'])
task1_test_df['label'] =
label_encoder.transform(task1_test_df['label'])

display(task1_train_df)
display(task1_test_df)
```

Task 2

```
task2_train_df = pd.read_csv('propaganda_train.tsv', delimiter='\t',
quotechar='\'')
task2_test_df = pd.read_csv('propaganda_val.tsv', delimiter='\t',
quotechar='\'')

task2_train_df = task2_train_df[task2_train_df['label'] !=
'not_propaganda']
task2_test_df = task2_test_df[task2_test_df['label'] !=
'not_propaganda']

for index, row in task2_train_df.iterrows():
    bos_index = row['tagged_in_context'].index("<BOS>")
    eos_index = row['tagged_in_context'].index("<EOS>")
    task2_train_df.loc[index,'tagged_in_context'] =
row['tagged_in_context'][bos_index + 5:eos_index]

for index, row in task2_test_df.iterrows():
    bos_index = row['tagged_in_context'].index("<BOS>")
    eos_index = row['tagged_in_context'].index("<EOS>")
    task2_test_df.loc[index,'tagged_in_context'] =
row['tagged_in_context'][bos_index + 5:eos_index]

label_encoder = LabelEncoder()
task2_train_df['label'] =
label_encoder.fit_transform(task2_train_df['label'])
task2_test_df['label'] =
label_encoder.transform(task2_test_df['label'])


display(task2_train_df)
display(task2_test_df)
```

TASK-1 EDA

```
# Plotting class distributions

class_counts_train =
task1_train_df['label'].value_counts().sort_index()

class_counts_test = task1_test_df['label'].value_counts().sort_index()

df = pd.DataFrame({'Train': class_counts_train, 'Test':
class_counts_test})

plt.figure(figsize=(12, 8))
df.plot(kind='barh', ax=plt.gca())
plt.title('Class Distribution in Training and Test Sets')
plt.xlabel('Frequency')
plt.ylabel('Classes')
plt.show()

# Plotting maximum, minimum and average word counts of the datasets

task1_train_df['word_count'] =
task1_train_df['tagged_in_context'].apply(lambda x: len(x.split()))
task1_test_df['word_count'] =
task1_test_df['tagged_in_context'].apply(lambda x: len(x.split()))

train_min = task1_train_df['word_count'].min()
train_max = task1_train_df['word_count'].max()
train_mean = task1_train_df['word_count'].mean()

test_min = task1_test_df['word_count'].min()
test_max = task1_test_df['word_count'].max()
test_mean = task1_test_df['word_count'].mean()

stats = pd.DataFrame({
    'Train': [train_min, train_max, train_mean],
    'Test': [test_min, test_max, test_mean]
}, index=['Minimum', 'Maximum', 'Average'])

stats.plot(kind='bar', figsize=(10, 6))
plt.title('Word Counts in Train and Test Sentences')
plt.ylabel('Word Count')
plt.xticks(rotation=0)
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

TASK-2 EDA

```
# Plotting class distributions
```

```
class_counts_train =
task2_train_df['label'].value_counts().sort_index()

class_counts_test = task2_test_df['label'].value_counts().sort_index()

df = pd.DataFrame({'Train': class_counts_train, 'Test':
class_counts_test})

plt.figure(figsize=(12, 8))
df.plot(kind='barh', ax=plt.gca())
plt.title('Class Distribution in Training and Test Sets')
plt.xlabel('Frequency')
plt.ylabel('Classes')
plt.show()

# Plotting maximum, minimum and average word counts of the datasets

task2_train_df['word_count'] =
task2_train_df['tagged_in_context'].apply(lambda x: len(x.split()))
task2_test_df['word_count'] =
task2_test_df['tagged_in_context'].apply(lambda x: len(x.split()))

train_min = task2_train_df['word_count'].min()
train_max = task2_train_df['word_count'].max()
train_mean = task2_train_df['word_count'].mean()

test_min = task2_test_df['word_count'].min()
test_max = task2_test_df['word_count'].max()
test_mean = task2_test_df['word_count'].mean()

stats = pd.DataFrame({
    'Train': [train_min, train_max, train_mean],
    'Test': [test_min, test_max, test_mean]
}, index=['Minimum', 'Maximum', 'Average'])

stats.plot(kind='bar', figsize=(10, 6))
plt.title('Word Counts in Train and Test Sentences')
plt.ylabel('Word Count')
plt.xticks(rotation=0)
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

DeBERTaV3

```
from datasets import Dataset
class DebertaV3():

    def __init__(self,train,test,n_labels):

        """
```

```
            Creates Dataset from train and test datasets. Creates model
and tokenizer.
            :param train: Train set
            :param test: Test set
            :param n_labels: Number of classes
        """

        self.n_labels = n_labels
        # Creating dataset, loading DeBERTaV3 model and tokenizer
        self.train_dataset = Dataset.from_pandas(train)
        self.test_dataset = Dataset.from_pandas(test)

        device = torch.device("mps")
        model_name = "microsoft/deberta-v3-base"
        self.tokenizer =
DebertaV2Tokenizer.from_pretrained(model_name)
        self.model =
DebertaV2ForSequenceClassification.from_pretrained(model_name,
num_labels=n_labels)
        self.model.to(device)

    def tokenize(self, data):
        """
            It tokenizes the data given as parameters.
            :param data: Train or test data
            :return: Tokenized data
        """
        return self.tokenizer(data['tagged_in_context'],
padding="max_length", truncation=True, max_length=128)

    def create_datasets(self):
        """
            Tokenizes train and test sets
        """
        self.train_dataset = self.train_dataset.map(self.tokenize,
batched=True)
        self.test_dataset = self.test_dataset.map(self.tokenize,
batched=True)


    def evaluate(self, eval_pred):
        """
            Calculates Accuracy and F1 score. Creates confusion
matrix.
            :return: Accuracy and F1 values with dictionary
        """
        accuracy_metric = load_metric("accuracy")
        f1_metric = load_metric("f1")

        logits, labels = eval_pred
```

```
        predictions = np.argmax(logits, axis=-1)
        accuracy = accuracy_metric.compute(predictions=predictions,
references=labels)
        f1 = f1_metric.compute(predictions=predictions,
references=labels, average='macro')

        cm = confusion_matrix(labels, predictions)
        np.save("confusion_matrix.npy", cm)

        return {"Accuracy" : accuracy, "F1" : f1}

    def train(self, b_size, epoch):

        """
            Trains and evaluates the model. Logs results to wandb.
            :param b_size: Batch size
            :param epoch: Number of epochs
        """

        #wandb.init(project="deberta-propaganda-detection-trainer") #
Logging to wandb

        self.create_datasets()

        batch_size = b_size
        epochs = epoch
        torch.mps.empty_cache()
        log_time = int(self.train_dataset.num_rows / epochs)

        training_args = TrainingArguments(
            output_dir="./results",
            evaluation_strategy="epoch",
            learning_rate=2e-5,
            logging_strategy="epoch",
            logging_steps=log_time,
            per_device_train_batch_size=batch_size,
            per_device_eval_batch_size=batch_size,
            num_train_epochs=epochs,
            weight_decay=0.01,
            use_mps_device=True,
            report_to="wandb"
        )

        trainer = Trainer(
            model=self.model,
            args=training_args,
            train_dataset=self.train_dataset,
            eval_dataset=self.test_dataset,
            compute_metrics=self.evaluate
        )
```

```
        trainer.train()

        results = trainer.evaluate()
        print(results)

    def plot_confusion_matrix(self):

        """
            Plots confusion matrix and accuracy by class graph
        """

        # Loading confusion matrix saved by evaluate function
        cm = np.load("confusion_matrix.npy")
        cm_norm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

        df = pd.read_csv('propaganda_train.tsv', delimiter='\t')

        if self.n_labels == 2:
            df.loc[df['label'] != "not_propaganda", 'label'] =
"propaganda"
        else:
            df = df[df['label'] != 'not_propaganda']

        labels = df['label'].unique()
        plt.figure(figsize=(8, 8))

        sns.heatmap(cm_norm, annot=True, fmt=".2f", cmap='Blues')
        plt.xticks(range(1, self.n_labels + 1), labels=labels,
rotation ='vertical')
        plt.yticks(range(1, self.n_labels + 1), labels=labels,
rotation ='horizontal')
        plt.ylabel('True Labels')
        plt.xlabel('Predicted Labels')
        plt.title('Confusion Matrix')
        plt.margins(0.2)
        plt.subplots_adjust(bottom = 0.15)
        plt.show()


        # Calculating accuracy per class
        class_accuracy = 100 * cm_norm.diagonal() /
cm_norm.sum(axis=1)

        plt.figure(figsize=(10, 8))
        sns.barplot(x=labels, y=class_accuracy, palette='viridis')
        plt.xlabel('Classes')
        plt.ylabel('Accuracy (%)')
        plt.title('Model Accuracy by Class')
        plt.ylim(0, 100)
```

```
        plt.xticks(rotation=90)
        plt.show()

# Run Task 2
# debertav3 = DebertaV3(task2_train_df, task2_test_df,
len(task2_train_df["label"].unique()))
# debertav3.train(32, 1)
# debertav3.plot_confusion_matrix()

# Run Task 1
debertav3 = DebertaV3(task1_train_df, task1_test_df,
len(task1_train_df["label"].unique()))
debertav3.train(32, 1)
debertav3.plot_confusion_matrix()
```

LSTM with BERT Word Embeddings

```
from torch.utils.data import DataLoader, Dataset

# Creating LSTM with BERT Word Embeddings model
class LSTMWithBERTWordEmbeddings(nn.Module):
    def __init__(self, n_classes, bert_model):
        """
            Initializes model architecture
            :param n_classes: Number of classes
            :param bert_model: BERT model
        """
        super(LSTMWithBERTWordEmbeddings, self).__init__()
        self.bert = bert_model
        self.drop = nn.Dropout(p=0.3)
        self.lstm = nn.LSTM(self.bert.config.hidden_size, 128,
num_layers=1, batch_first=True)
        self.out = nn.Linear(128, n_classes)

    def forward(self, input_ids, attention_mask):
        """
            Processes input through BERT and LSTM to generate class
predictions.
            :param input_ids: Tensor of token IDs from tokenizer
            :param attention_mask: Tensor representing the attention
mask
            :return: Output tensor from the final linear layer
        """
        outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask,
            return_dict=True
        )
        last_hidden_state = outputs['last_hidden_state']
```

```
        _, (hidden, _) = self.lstm(last_hidden_state)  # BERT output
to LSTM
        hidden = self.drop(hidden.squeeze(0))
        return self.out(hidden)


class LSTMDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        """
           Initializes custom dataset.
           :param texts: Sentences
           :param labels: Labels
           :param tokenizer: Tokenizer
           :param max_len: Maximum sentence length
        """
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, idx):
        """
            Encodes and returns data from the given index
            :param idx: Index
            :return: dictionary containing input_ids, attention_mask
and labels
        """
        text = self.texts.iloc[idx]
        label = self.labels.iloc[idx]
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            return_token_type_ids=False,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

class LBWE:
```

```
    def __init__(self, train, test, b_size, epoch, num_classes):

        """
           Initializes variables. Creates Dataset from train and test
datasets. Creates model, tokenizer, optimizer and loss function.
Creates dataloaders.
           :param train: Train set
           :param test: Test set
           :param b_size: Batch size
           :param epoch: Number of epoch
           :param num_classes: Number of classes
        """

        self.num_classes = num_classes

        # Loading BERT Model and Tokenizer
        self.tokenizer = BertTokenizer.from_pretrained('bert-base-
uncased')
        bert_model = BertModel.from_pretrained('bert-base-uncased')

        train_dataset = LSTMDataset(train["tagged_in_context"],
train["label"], self.tokenizer, max_len=128)
        test_dataset = LSTMDataset(test["tagged_in_context"],
test["label"], self.tokenizer, max_len=128)

        # Creating dataloaders, defining model, optimizer and loss
functions
        self.num_epochs = epoch

        self.train_loader = DataLoader(train_dataset,
batch_size=b_size, shuffle=True)
        self.test_loader = DataLoader(test_dataset, batch_size=b_size)

        self.model = LSTMWithBERTWordEmbeddings(num_classes,
bert_model)
        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=2e-5)
        self.criterion = nn.CrossEntropyLoss()

        self.device = torch.device("mps")
        self.model.to(self.device)


    def train(self):

        """
            Trains and evaluates the model. Logs results to wandb.
        """
```

```
        # Training
        for epoch in range(self.num_epochs):
            self.model.train()
            total_loss, total_accuracy = 0, 0

            for data in tqdm(self.train_loader):
                input_ids = data['input_ids'].to(self.device)
                attention_mask =
data['attention_mask'].to(self.device)
                labels = data['labels'].to(self.device)

                outputs = self.model(input_ids, attention_mask)
                loss = self.criterion(outputs, labels)

                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

                total_loss += loss.item()
                total_accuracy += (outputs.argmax(dim=1) ==
labels).sum().item()

            avg_train_loss = total_loss / len(self.train_loader)
            avg_train_accuracy = total_accuracy /
len(self.train_loader.dataset)
            wandb.define_metric("epoch")
            wandb.define_metric("train_loss", step_metric="epoch")

            wandb.log({
                "train_loss": avg_train_loss,
                "epoch": epoch
                })

            wandb.define_metric("train_accuracy", step_metric="epoch")
            wandb.log({"train_accuracy": avg_train_accuracy,
                "epoch": epoch})
            print(f"Epoch {epoch+1}/{self.num_epochs}, Train Loss:
{avg_train_loss:.5f}, Train Accuracy: {avg_train_accuracy:.5f}")

            # Evaluation
            self.model.eval()
            total_test_loss, total_test_accuracy = 0, 0
            predictions = []
            true_labels = []
            with torch.no_grad():
                for data in self.test_loader:
                    input_ids = data['input_ids'].to(self.device)
                    attention_mask =
data['attention_mask'].to(self.device)
                    labels = data['labels'].to(self.device)
```

```
                    outputs = self.model(input_ids, attention_mask)
                    loss = self.criterion(outputs, labels)

                    total_test_loss += loss.item()
                    total_test_accuracy += (outputs.argmax(dim=1) ==
labels).sum().item()

                    _, preds = torch.max(outputs, dim=1)
                    predictions.extend(preds.tolist())
                    true_labels.extend(labels.tolist())

            avg_test_loss = total_test_loss / len(self.test_loader)
            avg_test_accuracy = total_test_accuracy /
len(self.test_loader.dataset)
            f1 = f1_score(true_labels, predictions, average="macro")
            wandb.define_metric("test_loss", step_metric="epoch")
            wandb.log({"test_loss": avg_test_loss,
                "epoch": epoch})
            wandb.define_metric("test_accuracy", step_metric="epoch")
            wandb.log({"test_accuracy": avg_test_accuracy,
                "epoch": epoch})
            wandb.define_metric("test_f1", step_metric="epoch")
            wandb.log({"test_f1": f1,
                "epoch": epoch})
            print(f"Epoch {epoch+1}/{self.num_epochs}, Test Loss:
{avg_test_loss:.5f}, Test Accuracy: {avg_test_accuracy:.5f}")

    def plot_confusion_matrix(self):

        """
            Plots confusion matrix and accuracy by class graph
        """

        y_pred = []
        y_true = []

        self.model.eval()

        with torch.no_grad():
            for data in self.test_loader:
                input_ids = data['input_ids'].to(self.device)
                attention_mask =
data['attention_mask'].to(self.device)
                labels = data['labels'].to(self.device)
                y_true.extend(labels.tolist())

                outputs = self.model(input_ids, attention_mask)
                loss = self.criterion(outputs, labels)
```

```
                _, preds = torch.max(outputs, dim=1)
                y_pred.extend(preds.tolist())


        df = pd.read_csv('propaganda_train.tsv', delimiter='\t')

        if self.num_classes == 2:
            df.loc[df['label'] != "not_propaganda", 'label'] =
"propaganda"
        else:
            df = df[df['label'] != 'not_propaganda']
        labels = df['label'].unique()

        cm = confusion_matrix(y_true, y_pred)
        cm_norm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
# Normalize the confusion matrix

        plt.figure(figsize=(8, 8))

        sns.heatmap(cm_norm, annot=True, fmt=".2f", cmap='Blues')
        plt.xticks(range(1, self.num_classes + 1), labels=labels,
rotation ='vertical')
        plt.yticks(range(1, self.num_classes + 1), labels=labels,
rotation ='horizontal')
        plt.ylabel('True Labels')
        plt.xlabel('Predicted Labels')
        plt.title('Confusion Matrix')
        plt.margins(0.2)
        plt.subplots_adjust(bottom = 0.15)
        plt.show()

        class_accuracy = 100 * cm.diagonal() / cm.sum(axis=1)

        plt.figure(figsize=(10, 8))
        sns.barplot(x=labels, y=class_accuracy, palette='viridis')
        plt.xlabel('Classes')
        plt.ylabel('Accuracy (%)')
        plt.title('Model Accuracy by Class')
        plt.ylim(0, 100)
        plt.xticks(rotation=90)
        plt.show()

# Run Task 2
# lbwe = LBWE(task2_train_df, task2_test_df, 32, 1,
len(task2_train_df["label"].unique()))
# lbwe.train()
# lbwe.plot_confusion_matrix()


# Run Task 1
```

```
lbwe = LBWE(task1_train_df, task1_test_df, 32, 1,
len(task1_train_df["label"].unique()))
lbwe.train()
lbwe.plot_confusion_matrix()
```

SVM

```
class SVM:

    def __init__(self,train,test):

        """
            Initializes train, test sets and word2vec model.
            :param train: Train set
            :param test: Test set
        """

        self.train_df = train
        self.test_df = test
        # Loading word2vec from local
        self.word2vec_model =
KeyedVectors.load_word2vec_format('GoogleNews-vectors-
negative300.bin', binary=True)

    # Data preprocessing
    def preprocess_text(self,text):
        """
            Preprocesses text
            :param text: Sentence
            :return: Processed text
        """
        text = text.lower()
        text = re.sub(r'[^\w\s]', '', text)
        stop_words = set(stopwords.words('english'))
        word_tokens = word_tokenize(text)
        filtered_text = [word for word in word_tokens if word not in
stop_words]
        return ' '.join(filtered_text)

    def normalise_dataframes(self):
        """
            Applies preprocessing to train and test dataframes
        """
        self.train_df['normalized_tagged_in_context'] =
self.train_df['tagged_in_context'].apply(self.preprocess_text)
        self.test_df['normalized_tagged_in_context'] =
self.test_df['tagged_in_context'].apply(self.preprocess_text)

    def create_sentence_vector(self,word2vec_model, sentence):
```

```
        """
            Creates sentence vectors with word2vec
            :param word2vec_model: Word2vec model
            :param sentence: Sentence
            :return: Sentence vector
        """
        words = sentence.split()
        word_vectors = [word2vec_model[word] for word in words if word
in word2vec_model]
        if not word_vectors:
            return np.zeros(word2vec_model.vector_size)
        return np.mean(word_vectors, axis=0)

    def create_train_test(self):

        """
            Creates train and test sets for training
            :param word2vec_model: Word2vec model
            :param sentence: Sentence
            :return: Created train and test sets
        """

        self.normalise_dataframes()

        # Calculates sentence vector for all sentences in the
dataframes
        self.train_df['doc_vector'] =
self.train_df['normalized_tagged_in_context'].apply(lambda doc:
self.create_sentence_vector(self.word2vec_model, doc))
        self.test_df['doc_vector'] =
self.test_df['normalized_tagged_in_context'].apply(lambda doc:
self.create_sentence_vector(self.word2vec_model, doc))

        # Creating x_train, y_train, x_test and y_test. Assigning
vectors to x and assigning labels to y
        x_train = np.vstack(self.train_df['doc_vector'])
        y_train = self.train_df['label'].values
        x_test = np.vstack(self.test_df['doc_vector'])
        y_test = self.test_df['label'].values

        return x_train,y_train,x_test,y_test


    def train(self,x_train,y_train,x_test,y_test):

        """
            Trains the model
            :param x_train: Training sentences
            :param y_train: Training labels
            :param x_tets: Test sentences
```

```
            :param y_test: Test labels
            :return: GridSearchCV results
        """


        #Running GridSearchCV
        param_grid = {
            'C': [0.1, 1, 10, 100],
            'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
            'gamma': ['scale', 'auto', 0.1, 1, 10]
        }

        grid_search = GridSearchCV(SVC(), param_grid, refit=True,
verbose=3, scoring='f1_macro', cv=2)
        grid_search.fit(x_train, y_train)
        results = pd.DataFrame(grid_search.cv_results_)

        return results

    def plot_results(self, results):

        """
            Plots results with hetmap for every kernel
            :param results: GridSearchCV results
        """

        # Creating subplots for every kernel
        kernels = results['param_kernel'].unique()
        num_kernels = len(kernels)
        fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15, 10))
        axes = axes.flatten()

        for idx, kernel in enumerate(kernels):
            kernel_results = results[results['param_kernel'] ==
kernel]

            pivot_table =
kernel_results.pivot_table(values='mean_test_score', index='param_C',
columns='param_gamma', aggfunc=np.max)

            sns.heatmap(pivot_table, ax=axes[idx], annot=True,
fmt=".3f", cmap="coolwarm", cbar_kws={'label': 'F1 Score'})
            axes[idx].set_title(f'Task - 1 SVM F1 Score Optimization
for Kernel: {kernel}')
            axes[idx].set_xlabel('Gamma')
            axes[idx].set_ylabel('C')

        for ax in axes[num_kernels:]:
            fig.delaxes(ax)
```

```
        fig.tight_layout(pad=3.0)
        plt.show()

    def plot_confusion_matrix(self,x_train, y_train, kernel_param,
c_param, gamma_param):

        """
            Plots confusion matrix
            :param x_train: Training sentences
            :param y_train: Training labels
            :param kernel_param: Kernel type
            :param c_param: C value
            :param gamma_param: Gamma value
        """

        # Train with best hyperparameters
        svm_classifier = SVC(kernel=kernel_param, C=c_param,
gamma=gamma_param)
        svm_classifier.fit(x_train, y_train)

        # Predict
        y_pred = svm_classifier.predict(x_test)

        print("Accuracy:", accuracy_score(y_test, y_pred))
        cm_normalized = confusion_matrix(y_test, y_pred,
normalize='true')

        df = pd.read_csv('propaganda_train.tsv', delimiter='\t')
        df.loc[df['label'] != "not_propaganda", 'label'] =
"propaganda"
        labels = df['label'].unique()

        plt.figure(figsize=(8, 8))
        sns.heatmap(cm_normalized, annot=True, fmt=".2f",
cmap='Blues')
        plt.xticks(range(1,len(labels) + 1), labels=labels, rotation
='vertical')
        plt.yticks(range(1,len(labels) + 1), labels=labels, rotation
='horizontal')
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.title('Confusion Matrix')
        plt.show()



# Run Task 1
# svm = SVM(task1_train_df,task1_test_df)
# x_train,y_train,x_test,y_test = svm.create_train_test()
# results = svm.train(x_train,y_train,x_test,y_test)
```

```
# svm.plot_results(results)
# svm.plot_confusion_matrix(x_train,y_train,"rbf",1,1)

# Run Task 2
svm = SVM(task2_train_df,task2_test_df)
x_train,y_train,x_test,y_test = svm.create_train_test()
results = svm.train(x_train,y_train,x_test,y_test)
svm.plot_results(results)
svm.plot_confusion_matrix(x_train,y_train,"rbf",1,1)
```