Investigating Propaganda Detection

Candidate: 260037

Advanced Natural Language Processing (968G5)
University of Sussex

May 10, 2024

Abstract

This report investigates different approaches to text classification in the context of two propaganda detection tasks. The investigation will be carried out through experimentation and analysis of two different machine learning approaches for each task, with performance analysed with reference to wider empirical considerations.

Owing to their prominence in many contemporary natural language applications [2] and their use in many of the best performing models in the original paper from which the datasets in this assignment are derived [3], both approaches investigated used will utilise vectorised word-embeddings. The first approach will use pre-trained static word embeddings trained as specified in [4], whereas the second will use contextualised word embeddings encoded by the BERT transformer model as described in [1]. In both approaches these embeddings will form the as inputs to a multi-layer perceptron (MLP) classifier model.

The findamental differences between embeddings, model performance on the different tasks and comparison between them shall be used as a basis for discussion of the approached as well as representation and classification in natural language processing more generally.

Words: 4094

1 Introduction/Propaganda

Propaganda can be defined as:

> "Information, ideas, opinions, or images, often only giving one part of an argument, that are broadcast, published, or in some other way spread with the intention of influencing people's opinions." [5]

As Da San Matrino et al point out [3], being able to identify text as being propaganda an increasingly important, but also particularly challenging machine learning task.

It is important because identifying whether information is neutral and purely informative [1], or if it has an intention to influence or persuade, can help people make more informed decisions, and draw more objective conclusions rather than being swayed by the intention of the person or organisation presenting that information.

As the above deinition points out - the medium for propaganda is media - 'information, ideas, opinions, or images' and increasingly people are immersed in media every day, perhaps to a degree never seen before, it is increasingly important. Furthermore, as noted by [3] this proliferation of media has led to an increased exposure to the phenomenon of 'fake news' which has led to more research on propaganda detection in this area. - important because of proliferation of media and so increased exposure - volume of it make it hard for experts to point it out - stirring of hatred based on ignorance

Propaganda detection is a particularly difficult task though, because, as [3] point out, "[p]ropaganda is most successful when it goes unnoticed by the reader...it often takes some training for people to be able to spot it". As such it can be genuinely difficult to distinguish the thing we are looking for from something authentic as it can often deliberately try to ape the form of genuine text. It can also be difficult as there are many different forms of propoganda. They list many but boil them down to 'a series of rhetorical and psychological techniques' - subjective nature?

Thus it can be seen that propoganda detection is clearly an important but challenging machine learning task.

Many of the deeper challenges of propoganda detection are mitigated in this particular assignment because we are using prelabeled data - where the issues of deciding what is and isn't propoganda in the first place has been done by those creating the dataset (for more details of the datset creation see section 3.1 in [3]) propoganda in raw text.

As such any deeper problems are built in and despite good performance on this data, one would have to rigorously any models built on this or even a similar dataset to be confident that they are in fact able to detect 'propaganda'

2 Problem Outline

The broad aim of the assignment is to build and evaluate 2 approaches to two different but similar classification tasks.

A more detailed description of the data, and how it was used in the solutions to the problems can be found below, but in brief; the data provided was in the form of a table of label and text pairs, where the label either described a type of propaganda found in the text (for example 'flag-waving' or 'loaded language') or identified the text as being 'not propaganda'. Each text item was a passage of text which contained segment start and end tags ('BOS' and 'EOS'). The label referred to the particular type of propaganda contained in the segment[2].

The first task (T1) was to build a binary classifier which could classify whether a passage of text contained a propaganda segment or not. The second task (T2) was a multi-class classification problem. In this task all non-propaganda instances were removed, and the task was to apply the correct specific propaganda-type-label to the given propaganda snippet.

[1] An entire article could be written about the possibility of 'neutral' objective truth, however there is little scope for this debate here. All that shall be said on this matter is that the many arguments that could be had about objective truth, and the subjective nature of the labelling required for the creation of the datasets from which the data used here derives all just goes to show how difficult a task real-world 'propaganda' detection is - as one has to first decide where and how to draw that line, and then try to train a model to learn that line

[2] and in depth description of the different propaganda type can be found in section 2 of [3]

3 Method

In this section a general introduction to the data, its distribution and the pre-processing carried out for both task shall be described. More detail will then be given on the two approaches used, and finally on the MLP classifier used.

As the same two approached were used for both tasks they shall be described in detail here, grouped by approach, whereas comparison in performance – both practically and with empirical reflection, shall be grouped by task, as each task brings out interesting nuances of each approach for discussion.

3.1 Data Analysis and Prepossessing

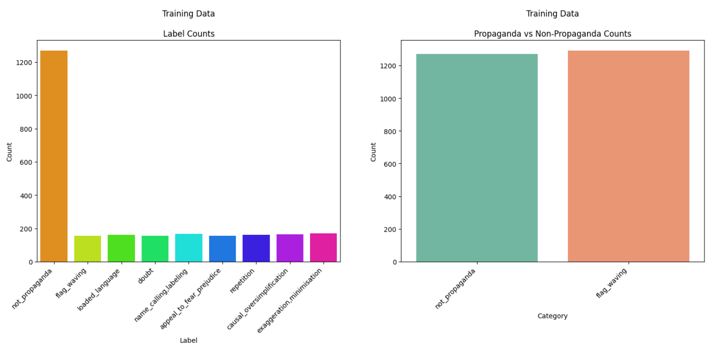
3.1.1 Data

The data for these tasks is a subset of that created as part of the Propaganda Techniques Corpus (the 'PTC-SemEval20 corpus') [3]. It was reated from 'a sample of news articles from .. mid-2017...to early 2019 [from] 13 propoganda and 36 non-propaganda news outlets". Labels were applied manually by 'professional annotators" and "consisted of both spotting a propaganda snippet and ...labelling it with a specific propoganda technique" [3] (pp1381) [3].

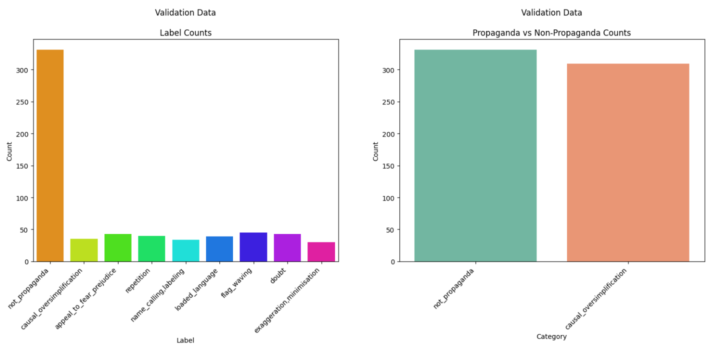
This subset of data were in the form of training and validation datasets in *.tsv* file format. Althgouh split in this way the datasets were structurally identical with just two columns 'label' and 'tagged in context' two examples of which can be seen in figure 1. The training dataset had 2560 entries and the validation dataset has 640 entries so the validation set is equactly 1/4 the size of the training data.

The labels in this task comprised of 9 possibilities...BLAH BLAH BLAH

Usually in a classification task such as this, one would reserve a test dataset in order to get a 'true' measure of performance on data that has been completely withheld during training. The validation set here, for example, was used to tune all of the hyperparameters discussed in section... However, the assignment brief makes clear that (absolute) performance is not the desired focus, and so given the helpful balance between training and validation provided, it was decided to use these two sets only for investigating the prposed methods.



(a) Training data distribution



(b) Validation data distribution

Figure 1: Data distribution in the training and validation sets

As can be seen in Fig 2, both datasets are relatively balanced with an almost even split between prpogranda and not-propaganda labels for binary classification, as well as a balanced splitbetween the individual propoganda labels for class two. There were no Null or Nan values in the dataset - it was clean and complete.

3.1.2 Preprocessing

Very little pre-processing was required owing to the straighforward and complete nature of the data provided. Approach-specific pre-processing is described in the relevent section below, however prior to

[3]again, the issues with this approach shall not be explored here, but there are many

this there was some basic transfomations applied to the data for each task.

For T1 a new binary column was added to the data with a 0 value for any text item with a label of 'non propoganda', and 1 for any other columns. This created a new target variable which was 1 for all instances where the text item contained propoganda, and 0 when it did not. As well as this for T1 a new feature column for this task was created with the original full context text with the snippet tags removed as seen in Fig ....

For T2 any rows with the 'not propaganda' label were removed leaving 1291 egs for training and 309 for validation. The remaining labels were given numerical categorical labels in accordance with a dictionary that reamined consistent across all runs seen in Fig 4. For T2 the classifier is required to classify the propoganda based on the snipped, so for this task a new feature column was created with just the text from inside the ¡BOS¿ ¡EOS¿ tags resulting in a transformation viewable in Fig 5

For T1 then, a training dataset of length 2560 and a validation dataset of length 640 both resembling Fig 1 was constructed.

For T2 the training data was 1291 and the validation 309 and resembled Fig 5. As shall be discussed below, further preprocessing was required for each of the methods, and all datasets were converted into pytorch datasets and dataloaders before the models were trained. Details can be found below.

label	**tagged in context**
loaded_language	This is why it is so <BOS> heinous <EOS>.
not_propaganda	No, <BOS> he <EOS> will not be confirmed.
flag_waving	Where is that authorized in <BOS> our Constitution <EOS>?

Figure 2: Example table - original

Propaganda	**detagged in context**
1	This is why it is so heinous.
0	No, he will not be confirmed.
1	Where is that authorized in our Constitution?

Figure 3: Example table - original

class label	**snippet**
6	heinous
0	our Constitution

Figure 4: Example table - original

Original Label	**Class label**
flag waving	0
exaggeration, minimisation	1
causal oversimplification	2
name calling, labeling	3
repetition	4
doubt	5
loaded language	6
appeal to fear prejudice	7

Figure 5: Example table - original

3.2 Word2Vec

3.2.1 Overview

Althoguh usage varies, word2vec is actually the name of a piece of software available as part of the foogle code archive (https://code.google.com/archive/p/word2vec/) which turns a corpus of text into a set of n-dimensional embeddings for a desired subset of the vocabulary of that corpus. it works by implimenting to a the pair of algorithms desribed by Mikolov et al [4] which "[compute] continuous vector representations of words from very large data sets"[4]. In this paper they demonstrate two approached to learning these dense vector embeddings -a continuous bag of words approach and a skip gram model approach.

The 'word2vec' approach used for both T1 and T2 (and outlines in fig 6) utilised pre-trained word embeddings which were trained as part of the Milov [4]. These 300 dimensional embedding are knows as the 'GoogleNews' kbedding and are publicly available word vectors trained on a large corpus of Google News articles. These embeddings were introduced by Mikolov et al. in their 2013 paper "Efficient Estimation of Word Representations in Vector Space" [4]. The embeddings were trained using the word2vec algorithms described in the paper, specifically the skip-gram model with negative sampling SGNS [4, 2].

The GoogleNews dataset used for training consists of approximately 100 billion words from various news articles. The resulting pre-trained embeddings contain 300-dimensional vectors for 3 million words and phrases. The large size of the training corpus allows the embeddings to capture rich semantic and syntactic relationships between words

Broadly, the two approaches to learning these dense representational embeddings are as follows. The CBOW model learns word embeddings by predicting a target word based on its surrounding context words. It ”uses the average of the context word vectors as the prediction for the target word” [4]. In contrast, the skip-gram model learns word embeddings by predicting the surrounding context words given a target word. It ”uses each target word as an input to a log-linear classifier with continuous projection layer and predicts words within a certain range before and after the current word” [4].

It is important to emphasis that whilst the embedding learnt via this process are trained on prediction tasks involving different contxts, the aim of the algorithm is to derive or to extract from the corpus being used dense representational vectors that themselves capture the semantic meaning of the words themselves as a whole. As such the embeddings produced by this process are considereed to be ’static’ and, in terms of representation, represent the multiple meaning of a word.

THey can be thought of as a devekioment of the idea of sparse distributional embeddings, but where that distibutional data has been compacted into a smaller, denser, more abstract representational space.

3.2.2 implimentation

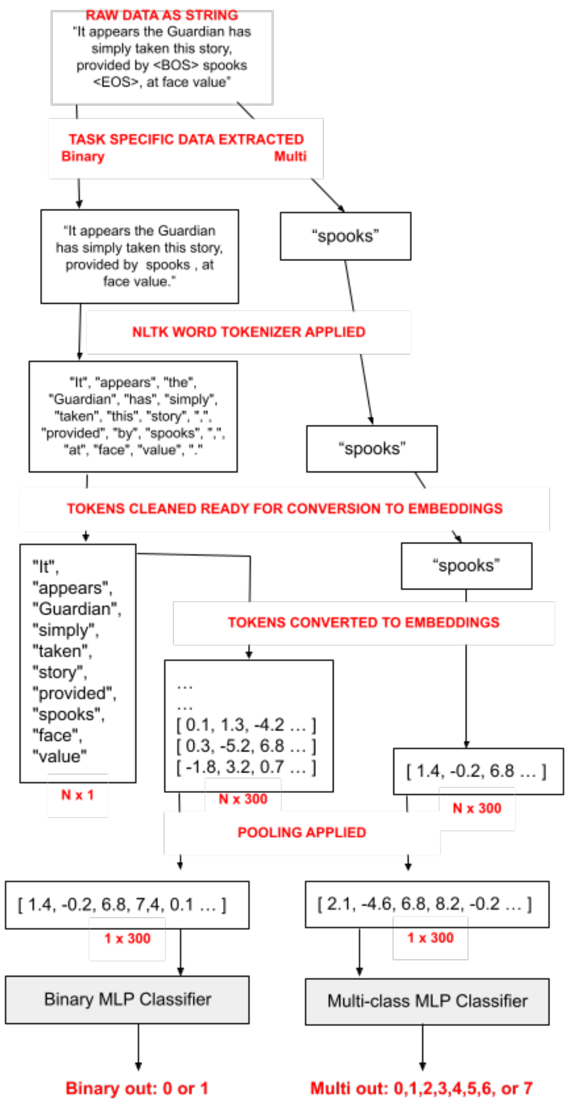

RAW DATA AS STRING
"It appears the Guardian has simply taken this story, provided by <BOS> spooks <EOS>, at face value"
TASK SPECIFIC DATA EXTRACTED
Binary
Multi
"It appears the Guardian has simply taken this story, provided by spooks , at face value."
"spooks"
NLTK WORD TOKENIZER APPLIED
"It", "appears", "the", "Guardian", "has", "simply", "taken", "this", "story", ",", "provided", "by", "spooks", ",", "at", "face", "value", "."
"spooks"
TOKENS CLEANED READY FOR CONVERSION TO EMBEDDINGS
"It", "appears", "Guardian", "simply", "taken", "story", "provided", "spooks", "face", "value"
"spooks"
TOKENS CONVERTED TO EMBEDDINGS
...
...
[0.1, 1.3, -4.2 ...]
[0.3, -5.2, 6.8 ...]
[-1.8, 3.2, 0.7 ...]
[1.4, -0.2, 6.8 ...]
N x 1
N x 300
N x 300
POOLING APPLIED
[1.4, -0.2, 6.8, 7,4, 0.1 ...]
[2.1, -4.6, 6.8, 8.2, -0.2 ...]
1 x 300
1 x 300
Binary MLP Classifier
Multi-class MLP Classifier
Binary out: 0 or 1
Multi out: 0,1,2,3,4,5,6, or 7

Figure 6: W2V CHART

As shown in Fig 6, Each sentence in string form was tokenised using the popular nltk wordtokenizer. Short and common words were removed along with punctuation as as a preprocessing step before obtaining the word2vec embeddings. This preprocessing is commonly performed to remove noise and focus on the most informative words in the text.

Removing short words, typically those with fewer than two or three characters, helps to eliminate articles, prepositions, and other less meaningful words that may not contribute significantly to the semantic content of the sentences. Similarly, filtering out common words, also known as stop words, such as ”the,” ”is,” ”and,” etc., reduces the dimensionality of the vocabulary and allows the word2vec model to concentrate on learning meaningful representations for the more informative words.

Punctuation is also often removed during preprocessing because it does not typically carry semantic information and can introduce unnecessary complexity into the word2vec model. By stripping away punctuation, the focus remains on the words themselves and their relationships within the text.

This preprocessing step of tokenization, removing short and common words, and eliminating punctuation helps to streamline the input data for the word2vec model. It reduces the vocabulary size, speeds up the training process, and allows the model to learn more effective and meaningful word embeddings that capture the semantic and syntactic relationships between the important words in the text.

Each Sentence was therefor trasnformed into a list of ’tokens’ of length n. The pretrained word2Vec embedding for each words is retrieves and so the sentence is then represented as $nx300$ dimensional matrix. As each setnence has a potentially different n, and a consistent input was needed for the classification model. Here there is a choice of options in order to pool these these representations were pooled using either max-pooling or mean-pooling. Both were experimented with as described in the section $_{b}elow.$

A further condieration was how to handle out of vocabulary words - where the vocabilary here is the word 2 vec imbedding keys. Two approached were tried - stemming and skipping. Again results below.

The flow chart in Fig shows the process by which a 300 dimensional dense sentence embedding was created for each sentence in the training and validation datasets.

It was this 300 dimensional representation which was fed to the neural net.

3.3 Bert

3.3.1 Overview

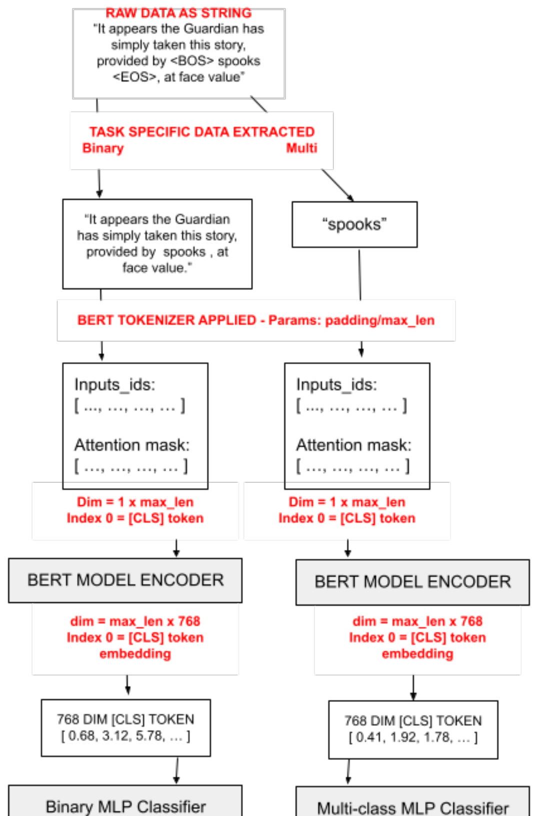
Bidirectional Encoder Representations from Transformers (BERT) is a state-of-the-art pre-trained language model developed by Google AI Language [?]. BERT is designed to learn deep bidirectional representations of text by jointly conditioning on both left and right context in all layers.

The model is pre-trained on a large corpus of unlabeled text, including the BooksCorpus (800M words) and English Wikipedia (2,500M words), using two training objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP).

In the MLM objective, a portion of the input tokens is randomly masked, and the model is trained to predict the original vocabulary id of the masked word based on its context. This allows BERT to learn a deep bidirectional representation of the text. In the NSP objective, the model is trained to predict whether two sentences follow each other in the original text, allowing BERT to capture long-range dependencies and understand sentence relationships.

One of the key innovations of BERT is the introduction of the [CLS] token, which is added to the beginning of every input sequence. The [CLS] token is used for classification tasks and serves as an aggregate representation of the entire input sequence. During fine-tuning for downstream tasks, the final hidden state corresponding to the [CLS] token is used as the sentence-level representation.

3.3.2 Implimentation



In the preprocessing step, the sentences were tokenized using the BERT tokenizer, and the [CLS] token was added to the beginning of each tokenized sequence. The sequences were then padded or truncated to a fixed length to ensure a consistent input size for the classification model. The pre-trained BERT model was used to encode each sentence, and the final hidden state of the [CLS] token was extracted as the sentence representation. These [CLS] representations were then used as input features for training the downstream classification model.

The process by which BERT was used can be seen in FIG 7. To obtain the BERT-encoded [CLS] representation for a sentence, the sentence is first tokenized using the BERT tokenizer. The Bert tokenizer actually does a number of task - - which splits the text into subword units called WordPieces [?] - gives them an index code for embeddings. The tokenized sequence is then prepended with the special [CLS] token (in cases where required) and appended with the [SEP] token, which is used to separate sentences.

The input sequence is then fed into the pre-trained BERT model, and the final hidden state of the [CLS] token is extracted as the sentence representation. The [CLS] token representation captures the contextual information of the entire sentence, taking into account the bidirectional context and long-range dependencies learned during pre-training. This representation can be used as input to downstream classification tasks, such as sentiment analysis, text classification, or natural language inference.

The effectiveness of using BERT-encoded [CLS] representations for sentence classification was evaluated and compared to the performance of using word2vec embeddings with different pooling strategies. The results of these experiments are presented and discussed in the following sections.

Considerations with Bert encoding were lessoned by the convenience bert tokeniser handling input tokenisation. However, it was still possible to choose a max len value and to choose which layer to take the CLS from.

The flow chart in Fig shows the process by which a 768 dimensional dense sentence embedding was created for each sentence in the training and validation datasets.

It was this 768 dimensional representation which was fed to the neural net.

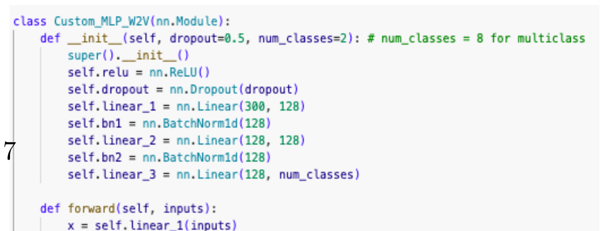
3.4 MLP

3.4.1 Overview

A Multi-Layer Perceptron (MLP) Classifier is a feedforward artificial neural network that consists of an input layer, one or more hidden layers, and an output layer. In this implementation, the MLP Classifier takes word embeddings as input and learns to classify the input into predefined categories. The input to the MLP Classifier is a sequence of word embeddings, which are dense vector representations of words. These embeddings can be obtained from pre-trained models like Word2Vec, GloVe, or FastText, or they can be learned jointly with the classifier during training. The word embeddings are passed through one or more fully connected hidden layers, where each neuron in a layer is connected to all neurons in the previous layer. The hidden layers apply a non-linear transformation to the input, allowing the model to learn complex patterns and relationships in the data. The output layer of the MLP Classifier has a specified output dimension, which corresponds to the number of target classes or categories. The output of the final hidden layer is fed into the output layer, which applies a softmax activation function. The softmax function converts the output scores into a probability distribution over the target classes. During training, the MLP Classifier is optimized using the categorical cross-entropy loss function. The categorical cross-entropy loss measures the dissimilarity between the predicted probability distribution and the true class distribution. The goal of training is to minimize this loss by adjusting the weights of the network through backpropagation and gradient descent. The MLP Classifier is a simple yet effective model for text classification tasks. It can learn to capture the semantic information present in the word embeddings and map them to the corresponding categories. The flexibility of the MLP architecture allows for the addition of multiple hidden layers and the adjustment of the output dimension based on the specific requirements of the task. Overall, the MLP classifier provides a simple yet effective approach to classify sentences based on their Word2Vec embeddings. By learning a non-linear mapping from the input embeddings to the target classes, the MLP can capture complex patterns and relationships in the data, enabling accurate classification of sentences.

3.4.2 MLP for word2vec

The Multi-Layer Perceptron (MLP) classifier for Word2Vec embeddings was implemented using



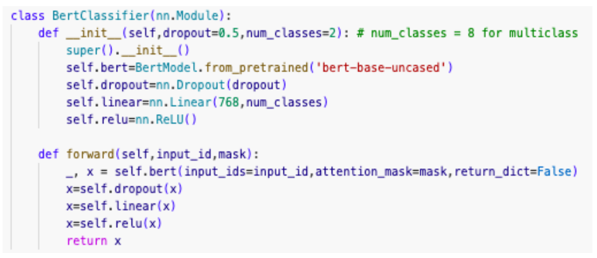
```python
class Custom_MLP_W2V(nn.Module):
    def __init__(self, dropout=0.5, num_classes=2): # num_classes = 8 for multiclass
        super().__init__()
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout)
        self.linear_1 = nn.Linear(300, 128)
        self.bn1 = nn.BatchNorm1d(128)
        self.linear_2 = nn.Linear(128, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.linear_3 = nn.Linear(128, num_classes)

    def forward(self, inputs):
        x = self.linear_1(inputs)
```

the PyTorch deep learning framework. The architecture of the MLP consists of three linear layers with ReLU activation and dropout regularization. The input to the MLP is a 300-dimensional Word2Vec embedding, which is passed through the first linear layer (self.linear 1) with an output dimension of 128. This layer is followed by a batch normalization layer (self.bn1) to normalize the activations and stabilize the training process. The ReLU activation function is then applied to introduce non-linearity into the model. To prevent overfitting and improve generalization, dropout regularization (self.dropout) is applied after the ReLU activation. Dropout randomly sets a fraction of the input units to 0 during training, which helps the model learn more robust features and reduces overfitting. The output of the dropout layer is then passed through the second linear layer (self.linear 2) with an output dimension of 128, followed by another batch normalization layer (self.bn2) and ReLU activation. Dropout is applied again to further regularize the model. Finally, the output of the second dropout layer is passed through the third linear layer (self.linear 3) with an output dimension equal to the number of target classes (num classes). The output of this layer represents the logits or raw predictions of the model. During training, the MLP is optimized using the categorical cross-entropy loss function. The categorical cross-entropy loss measures the dissimilarity between the predicted class probabilities and the true class labels. The goal of training is to minimize this loss by adjusting the weights of the MLP using backpropagation and gradient descent optimization algorithms. The forward pass of the MLP is defined in the forward() method. It takes the input Word2Vec embeddings (inputs) and applies the linear layers, batch normalization, ReLU activation, and dropout in the specified order. The final output of the forward pass is the logits, which can be passed through a softmax function to obtain class probabilities if needed.

3.4.3 MLP for bert

The BERT-based classifier for sentence classification was implemented using the PyTorch deep learning framework and the Hugging Face Transformers library. The architecture of the classifier consists of the pre-trained BERT model followed by a dropout layer and a linear classification layer.

The pre-trained BERT model (self.bert) is loaded from the 'bert-base-uncased' checkpoint using the `BertModel.from_pretrained()`. This model has been pre-trained on a large corpus of unlabeled text and provides a powerful foundation for downstream classification tasks. The input to the BERT-based classifier is a pair of tensors: `input_id` and `mask`. The `input_id` tensor represents the tokenized input sequence, where each token is mapped to its corresponding vocabulary index. The mask tensor is a binary mask that indicates which tokens are valid (non-padded) in the input sequence. During the forward pass, the `input_id` and mask tensors are passed through the pre-trained BERT model (self.bert). The BERT model processes the input sequence and generates a sequence of hidden states. The `return_dict=False` argument is used to obtain the last hidden state of the [CLS] token (x) and the pooled output, which is not used in this classifier. To regularize the model and prevent overfitting, dropout (self.dropout) is applied to the last hidden state of the [CLS] token. Dropout randomly sets a fraction of the input units to 0 during training, which helps the model learn more robust features and reduces overfitting. The output of the dropout layer is then passed through a linear classification layer (self.linear) with an output dimension equal to the number of target classes (num classes). This layer maps the BERT-encoded representation to the class logits. Finally, the ReLU activation function (self.relu) is applied to the output of the linear layer to introduce non-linearity and obtain the final class predictions. During



```python
class BertClassifier(nn.Module):
    def __init__(self,dropout=0.5,num_classes=2): # num_classes = 8 for multiclass
        super().__init__()
        self.bert=BertModel.from_pretrained('bert-base-uncased')
        self.dropout=nn.Dropout(dropout)
        self.linear=nn.Linear(768,num_classes)
        self.relu=nn.ReLU()

    def forward(self,input_id,mask):
        _, x = self.bert(input_ids=input_id,attention_mask=mask,return_dict=False)
        x=self.dropout(x)
        x=self.linear(x)
        x=self.relu(x)
        return x
```

Figure 9: BERT

training, the BERT-based classifier is optimized using the categorical cross-entropy loss function. The categorical cross-entropy loss measures the dissimilarity between the predicted class probabilities and the true class labels. The goal of training is to minimize this loss by fine-tuning the pre-trained BERT model and adjusting the weights of the classification layer using backpropagation and gradient descent optimization algorithms. The forward() method defines the forward pass of the BERT-based classifier. It takes the input id and mask tensors and applies the pre-trained BERT model, dropout, linear classification layer, and ReLU activation in the specified order. The output of the forward pass is the class logits, which can be passed through a softmax function to obtain class probabilities if needed. During training, the BERT-based classifier is fine-tuned on the training dataset. The pre-trained BERT model is used as a feature extractor, and its weights are updated along with the classification layer to adapt to the specific task. The performance of the trained model is evaluated on a separate validation or test dataset to assess its generalization ability. The hyperparameters of the BERT-based classifier, such as the dropout rate and the number of target classes, can be adjusted based on the specific requirements of the task. Fine-tuning techniques, such as gradient accumulation and learning rate scheduling, can be employed to optimize the training process and improve the model's performance. Overall, the BERT-based classifier leverages the power of pre-trained language models to achieve high accuracy in sentence classification tasks. By fine-tuning the pre-trained BERT model and adding a classification layer on top, the classifier can effectively capture the semantic information present in the input sentences and map them to the corresponding target classes.

3.4.4 Training

The MLP classifier is trained on the training dataset using the Word2Vec embeddings as input features. The model learns to map the input embeddings to the corresponding target classes by minimizing the categorical cross-entropy loss. The performance of the trained model is evaluated on a separate validation or test dataset to assess its generalization ability.

3.4.5 Training

The training process for the models (Word2Vec-based MLP and BERT-based classifier) followed a similar approach using the PyTorch deep learning framework. The main steps involved in training the models are as follows:

1. Data Preparation: - The training and validation data were loaded from the respective DataFrames (train df and val df) using custom dataset classes (CustomPropagandaDataset_vanilla). - The dataset classes handled text preprocessing, such as stemming (if stem_v is True), normalization (if norm_v is True), and pooling (based on the pool_v parameter). - The preprocessed data were then converted into PyTorch DataLoader objects (train_dataloader and val_dataloader) with specified batch sizes (bn) and shuffling options.

2. Model Initialization: - The models (Custom_MLP_W2V for Word2Vec-based MLP and BertClassifier for BERT-based classifier) were initialized with the desired hyperparameters, such as the number of classes (n_classes) and dropout rate (dr). - The models were moved to the appropriate device (CPU or GPU) using the .to(device) method.

3. Loss Function and Optimizer: - The CrossEntropyLoss criterion was used as the loss function for both models. - The Adam optimizer was used to optimize the model parameters, with a specified learning rate (lr).

4. Training Loop: - The models were trained for a specified number of epochs (epochs). - In each epoch, the models were set to training mode using model.train(). - The training data were iterated over in batches using the train_dataloader. - For each batch, the input data (train_input) and labels (train_label) were moved to the appropriate device. - The forward pass was performed, and the model outputs (output_1) were obtained. - The loss (batch_loss_1) was computed using the criterion and the model outputs. - The gradients were computed using backward propagation (batch_loss_1.backward()), and the optimizer updated the model parameters (optimizer.step()). - The training accuracy and loss were accumulated for each batch.

5. Validation Loop: - After each training epoch, the models were set to evaluation mode using model.eval(). - The validation data were iterated over in batches using the val_dataloader. - For each batch, the input data (val_input) and labels (val_label) were moved to the appropriate device. - The forward pass was performed, and the model outputs (output_2) were obtained. - The loss (batch_loss_2)

was computed using the criterion and the model outputs. - The validation accuracy and loss were accumulated for each batch. - The true labels (val_label) and predicted labels (output_2.argmax(dim=1)) were stored for later evaluation.

6. Model Evaluation and Saving: - The training and validation accuracies and losses for each epoch were stored in lists (train_acc_list, train_loss_list, val_acc_list, val_loss_list). - The best validation accuracy (best_val_acc) and the corresponding epoch (best_epoch) were tracked. - If the current validation accuracy exceeded the best validation accuracy, the model state was saved (best_model_state) along with the true and predicted labels for that epoch.

7. Visualization and Metrics: - The training and validation accuracy and loss curves were plotted using matplotlib. - The confusion matrix for the best-performing model was visualized using ConfusionMatrixDisplay. - The classification report, including precision, recall, and F1-score, was computed for the best-performing model.

8. Results Saving: - The training results, including accuracies, losses, hyperparameters, and evaluation metrics, were saved in a dictionary format (results_dict). - The results dictionary was then saved as a JSON file with a unique model ID.

This training process was repeated for different hyperparameter configurations, such as learning rate (lr), batch size (bn), and pooling strategy (pool_v), to find the best-performing models.

The training procedure aimed to optimize the models' performance by minimizing the cross-entropy loss and maximizing the accuracy on the validation set. The best-performing models were selected based on their validation accuracy and were further evaluated using metrics like confusion matrix and classification report.

4 Hyper-parameter tuning

There were a number of hyperpaprameters explored over many different runs. For evalluation and analysis the best performing models were selected, but an overview of parameters experimented with are presented here.

4.0.1 W2V

As can be seen in the chart in fig 2, the main choices for hyperparamaters in the W2V approach were whether to use stemming or not, and which type of pooling to use.

If stemming was set to be true, then before skipping a word which was found not to be in the word2vec keyed vocabulary, a packaged stemmer (both porter stemmer and snowball stemmer were tried) was applied and the stemmed version of the token was tried against the word2vec vocab. If the stem was present the stemmed version was used. If stemming was false, then any words would just be skipped and so not vectorised.

For pooling if Max Pooling: Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be n vectors, each with 300 dimensions. The max pooling operation selects the maximum value for each dimension across all the vectors:

$$MaxPool(\mathbf{x}1, \mathbf{x}2, \ldots, \mathbf{x}n) = (\max i \in 1, 2, \ldots, nxi1, \max i \in 1, 2, \ldots, nx_{i2}, \ldots, \max_{i \in 1,2,\ldots,n} x_{i300})$$

where x_{ij} represents the j-th element of the i-th vector. Mean Pooling: Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be n vectors, each with 300 dimensions. The mean pooling operation calculates the average value for each dimension across all the vectors:

$$MeanPool(\mathbf{x}1, \mathbf{x}2, \ldots, \mathbf{x}n) = \left(\frac{1}{n} \sum i = 1^n xi1, \frac{1}{n} \sum i = 1^n x_{i2}, \ldots, \frac{1}{n} \sum_{i=1}^{n} x_{i300} \right)$$

where x_{ij} represents the j-th element of the i-th vector. In these formulas:

$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ represent the n input vectors, each with 300 dimensions. x_{ij} represents the j-th element of the i-th vector. n is the number of vectors.

Max pooling selects the maximum value for each dimension across all the vectors, resulting in a single 300-dimensional vector that contains the maximum values from each dimension. Mean pooling calculates the average value for each dimension across all the vectors, resulting in a single 300-dimensional vector that contains the average values from each dimension. These pooling operations are commonly used to aggregate information from multiple vectors or feature maps in natural language processing tasks or convolutional neural networks.

As can be seen in the figures to the right there was not a huge impact from any of these measures, although using mean pooling slightly improved average performance in T1, and signficantly improved average performance in T2. It should be noted that these avwerages were across many different permutations of neural network models, and whilst are a udesul reflection of the impact of these hyperparamayters, as shall be shown other factor had bigger effects

the biggest impact any of these had was mean pooling which x this had a very small moderate affect on the average performance of models using stemming versus no stemming in both the tasks. Interesting the best performing w2v models did not use stemming, althoguh they did both use mean pooling.

4.0.2 Bert

For bert, the hyperamaters that were accessable to change were the max length of the vectors supplied. This was simply set to the max length of sents in tokens. The other paramters that were tuned were in the neural network layer.

4.0.3 MLP

A number of hyperparamaters were experimented with in the MLP architecture before arriving at the optimal values.

Dropout is a value which sets - default is 0.5 this was found to be optimal Batch size - default 32

The hyperparameters of the MLP, such as the dropout rate and the number of hidden units in each layer, can be tuned to optimize the model's performance. Additionally, techniques like early stopping and model checkpointing can be employed to prevent overfitting and select the best-performing model during training.

Overall, the MLP classifier provides a simple yet effective approach to classify sentences based on their Word2Vec embeddings. By learning a non-linear mapping from the input embeddings to the target classes, the MLP can capture complex patterns and relationships in the data, enabling accurate classification of sentences. Exeriments of hyperparams in each modality and what ended up being best?

5 Evaluation

All models were run trained for at least 50 epochs. Depending on peformance this was sometimes increased. The were trained using mini-batch gradient descent with the batch sizes varying as described above.

After every epoch the model at its current state was evaluated for accuracy against the validation data set. A loss value and accuracy value was then recorded every epoch for both training data and validation data, which for the basis of many of the performance plots used here.

The loss function used was the inbuild categorical cross entropy which actually applies a softmax layer to the outputs of the network in order to clauclate the following loss:

Certainly! The LaTeX formula for the categorical cross-entropy loss calculated by the PyTorch module is as follows: $CategoricalCrossEntropy(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$ where:

y represents the true labels, which are one-hot encoded vectors of shape (N, C). $\hat{y}$ represents the predicted probabilities, which are the output of the softmax function and have the same shape as y. N is the number of samples in the batch. C is the number of classes. y_{ic} is the value of the true label for the i-th sample and c-th class, which is either 0 or 1. $\hat{y}_{ic}$ is the predicted probability for the i-th sample and c-th class.

The categorical cross-entropy loss measures the dissimilarity between the true labels and the predicted probabilities. It encourages the model to assign high probabilities to the correct classes and low probabilities to the incorrect classes. The PyTorch module torch.nn.CrossEntropyLoss combines the softmax activation function and the categorical cross-entropy loss into a single layer. It automatically performs the softmax operation internally before computing the loss, so you don't need to apply the softmax function separately to the model's output. The loss is averaged across all samples in the batch by default. If you want to sum the losses instead of averaging, you can set the reduction parameter to 'sum' when creating an instance of the CrossEntropyLoss module. Note that the PyTorch module expects the input to be raw logits (unnormalized scores) rather than probabilities. The softmax operation is applied internally within the module.

The best perfmance during the total run was also recorded. As there was a general problem with overfitting to the training data this was helpful to capture which paramaters lead to the best perfromance before overfitting. The epoch at which this peak performance was reached was also receorded. And although disc space prevented storing all model runs at this point, in theory it would be possible to save the weights at this point to restore performance to this level.

The confusion matrix valuaes were gathered during the validation evaluation and so are collected only against the relatively small subsection of data (particulally small in the case of multi-class) which should be born in mind.

The best performing models only shall be compared

5.0.1 general observations

Performance on the two tasks can be seen on the right. It is clear that overall the BERT-based model performed significantly better in both tasks, the details of which shall be explored below. However it should be noted that it took signficantly longer to train, and w2V showed decent performance considering.

It became clear from the outset that a significant issue in this task was overfitting, with a typical loss and accuracy plot being the one in fig.

5.0.2 TASK 1

model	best accuracy	best epoch	stemming	pooling	learning rate	batch size
BERT	**0.839063**	17	n/a	n/a	$5e^{-6}$	50
W2V	0.767188	54	False	mean pooling	$5e^{-5}$	100

Figure 10: BINARY CLASSIFICATION TASK

Present the results

Performance of different models in the two tests – take best performing one from each and compare in terms of 'results' but then also maybe chuck in some other results for balance.

Compared to word2vec embeddings, which provide static word-level representations, BERT-encoded [CLS] representations offer several advantages. First, BERT captures the contextual information of words, allowing it to generate different representations for the same word depending on its surrounding context. Second, BERT's bidirectional architecture enables it to learn more powerful and expressive representations by considering both left and right context. Finally, the [CLS] token representation provides a fixed-length, sentence-level embedding that encapsulates the meaning of the entire sentence, making it suitable for sentence-level classification tasks.

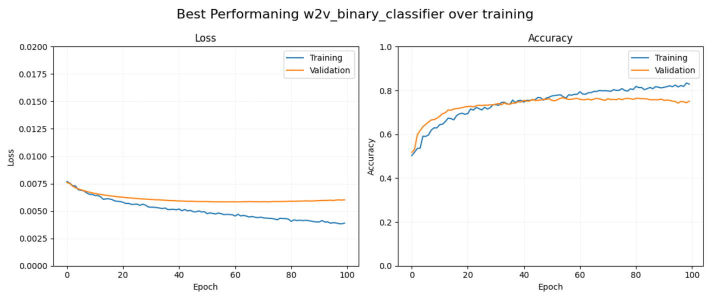


Figure 11: W2V performance

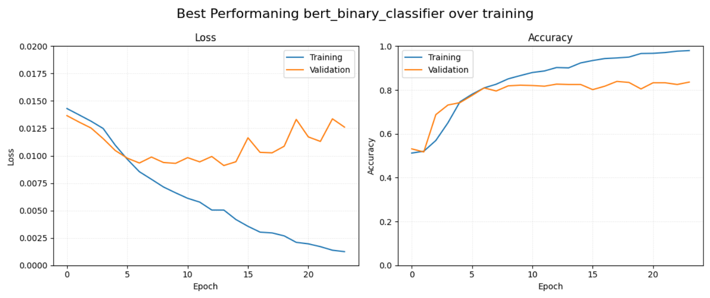
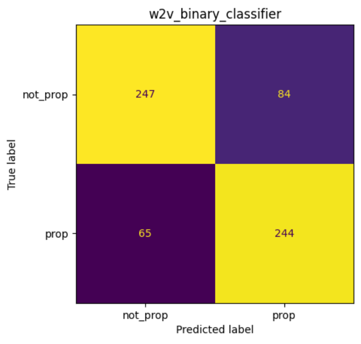
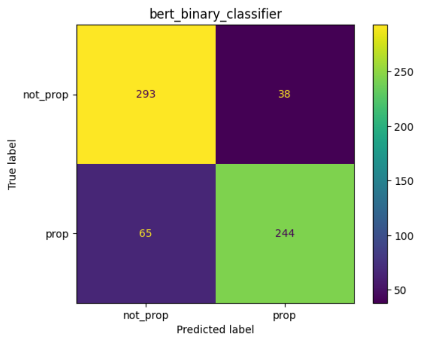


Figure 12: W2V performance



(a) w2v T2



(b) bert T2

Figure 13: Comparison of w2v and bert

5.0.3 Task 2

6 Analysis

7 Conclusion

hReiterate how shit this is, further work would be unsupervised somehow – maybe to detect balance?

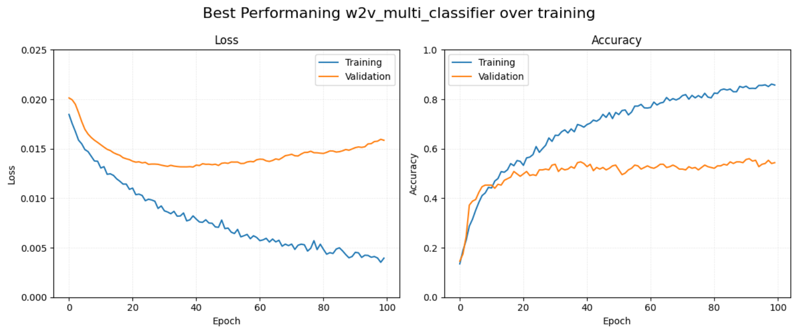


Figure 14: W2V performance

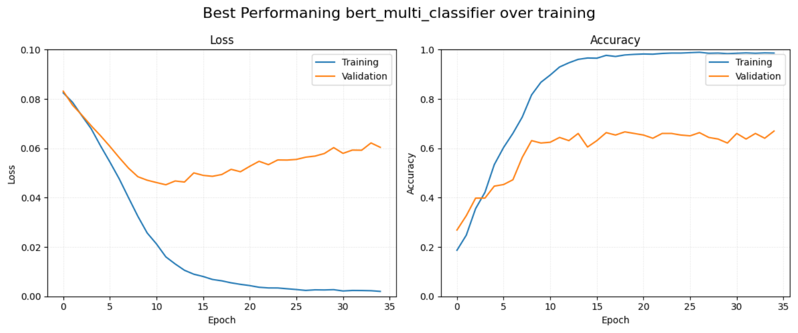
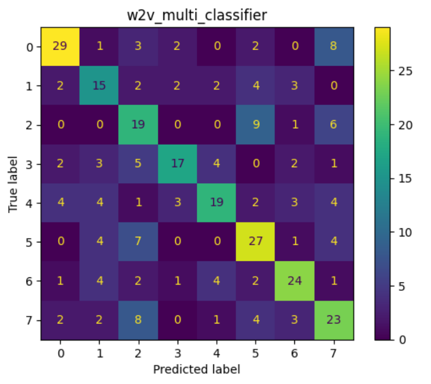
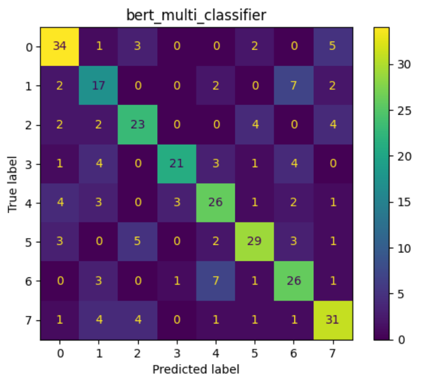


Figure 15: W2V performance



(a) w2v T2



(b) bert T2

Figure 16: Comparison of w2v and bert

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Pearson Prentice Hall, Upper Saddle River, N.J., 2009.

[3] Giovanni Da San Martino, Alberto Barrón-Cedeño, Henning Wachsmuth, Rostislav Petrov, and Preslav Nakov. Semeval-2020 task 11: Detection of propaganda techniques in news articles. *CoRR*, abs/2009.02696, 2020.

[4] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[5] Cambridge University Press. Definition of propaganda from the cambridge advanced learner's dictionary thesaurus.