GAN-BERT

Course: Deep Learning
Year: 2023-2024

Group 17

Full Name	Student ID
Amirabbas Afzali	400100662
Amirreza Velae	400102222
Hesam hosseini	400101034
Mahshad moradi	400109373

Tutor: Dr.Fatemi

Contents

1 | Introduction

1.1 | Background

GAN-BERT is a novel approach to improve the performance of BERT by using Generative Adversarial Networks (GANs). BERT is a state-of-the-art model for natural language processing (NLP) tasks, which is based on the transformer architecture. The transformer architecture is a type of neural network that is designed to handle sequential data, such as text. BERT has achieved impressive results on a wide range of NLP tasks, including question answering, text classification, and named entity recognition. However, BERT has some limitations, such as its inability to generate new text and its reliance on large amounts of labeled data. GAN-BERT aims to address these limitations by using GANs to generate new text and by using unsupervised learning to pre-train the model.

1.2 | Problem Statement and Dataset

The main idea behind GAN-BERT is to use GANs to generate new text that is similar to the input text. Input text contains data with 6 live classes, human, chatGPT, cohere, davinci, bloomz, and dolly. The GAN is trained to generate new text that is similar to the input text, and the generated text is then used to pre-train BERT. This allows BERT to learn from unlabeled data, which can improve its performance on downstream NLP tasks. In this report, we describe the architecture of GAN-BERT and present experimental results that demonstrate its effectiveness. We also compare GAN-BERT to other pre-training methods and discuss its potential applications in NLP.

1.3 | The Generators

Two main architectures are used in this project, G_1 and G_2. G_1 gets BOW or bag of words as input and generates a text. G_2 gets a simple gaussian noise z $N(0,1)$ and passes it through multiple layer perceptrons (MLP) to generate a text. Generated text from G_1 and G_2 are labeled as fake with a label of 6.

1.4 | The Discriminator

The Discriminator is a MLP that gets a text and classifies it as one of the 6 classes. The discriminator is trained to classify the generated text as fake and the input text as real.

1.5 | Goal of the Project

The goal of the generator is to generate text that is similar to the input text, and the goal of the discriminator is to distinguish between real and fake text. The generator and discriminator are trained in an adversarial manner, where the generator tries to fool the discriminator and the discriminator tries to detect the generated text. This adversarial training process helps the generator to learn the distribution of the input text and generate new text that is similar to the input text.

1.6 | Optimizer

We used AdamW optimizer to train the model. AdamW is an optimization algorithm commonly used in deep learning for training neural networks. It is an extension of the popular Adam optimizer, which combines the benefits of adaptive learning rates and momentum. The ”W” in AdamW stands for weight decay, which is a regularization technique used to prevent overfitting in machine learning models. Weight decay adds a penalty term to the loss function during training, encouraging the model to have smaller weights. This helps to prevent the model from becoming too complex and overfitting the training data. In AdamW, the weight decay is applied in a slightly different way compared to the original Adam optimizer. In AdamW, the weight decay is decoupled from the adaptive learning rate, meaning that the weight decay is applied directly to the weights during the parameter update step, rather than being incorporated into

the adaptive learning rate calculation. This decoupling helps to improve the generalization performance of the model. By using AdamW, you can benefit from the adaptive learning rate capabilities of Adam, while also effectively applying weight decay to prevent overfitting. This can lead to improved performance and better generalization of your neural network models.
In summary:

- AdamW is an optimization algorithm used for training neural networks.
- It is an extension of the Adam optimizer.
- The "W" stands for weight decay, a regularization technique to prevent overfitting.
- AdamW decouples weight decay from the adaptive learning rate calculation.
- It helps improve the generalization performance of the model.
- By using AdamW, you can benefit from adaptive learning rates and effective weight decay simultaneously.

2 | Preprocessing

First we need to preprocess the data before we can use it to train the model. First we need to get to know the data and understand it. We need to know the number of classes, the number of samples in each label, data manipulation, etc.

2.1 | Data Exploration

The dataset contains 5 classes, and the number of samples in each class is shown in the table below:

Class	Number of samples
1	11997
1	11995
2	11336
3	11999
4	11998
5	11702

Table 2.1: The number of samples in each class.

Sequence lenght of text is as follows:

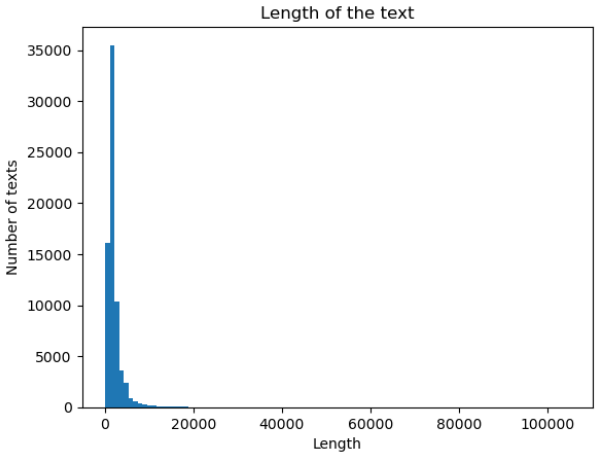


Figure 2.1: Sequence length of text

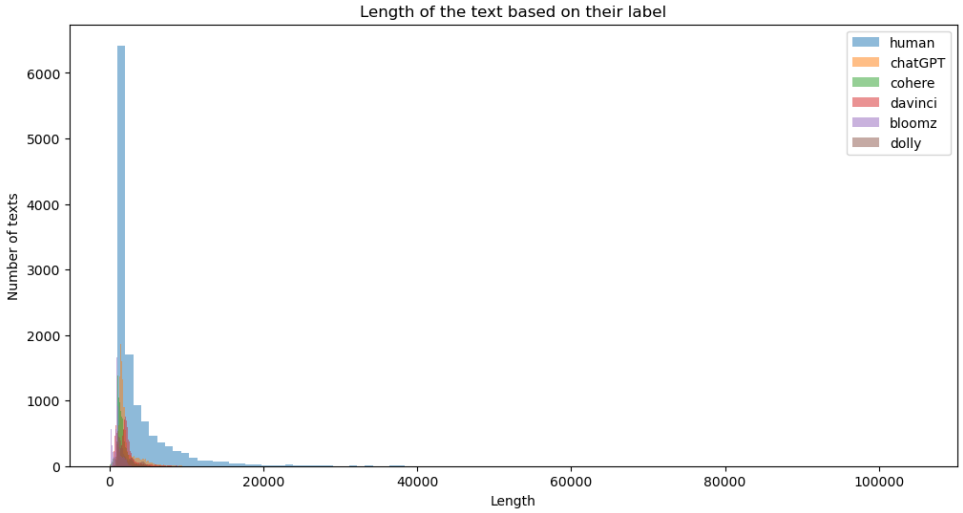


Figure 2.2: Sequence length of text based on class

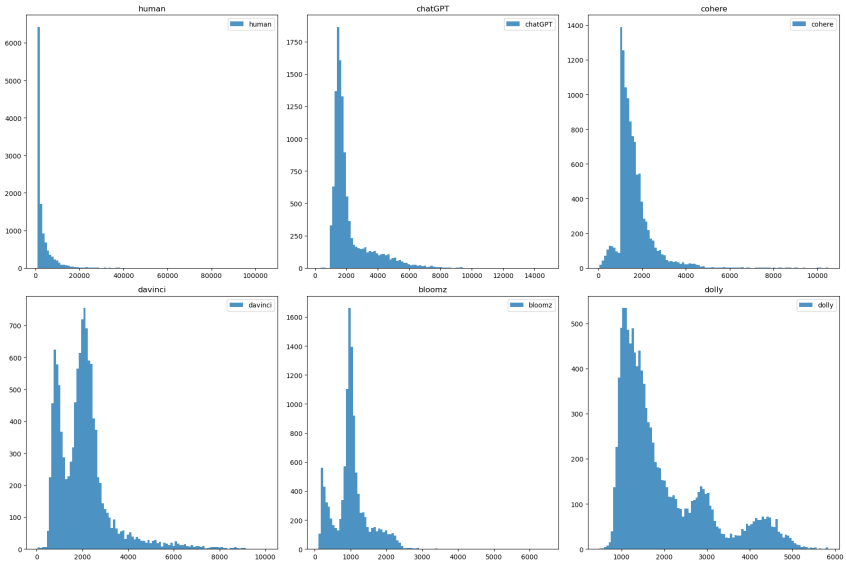


Figure 2.3: Sequence length of text based on class

2.2 | Tokenization

Tokenization is the process of converting a sequence of characters into a sequence of tokens. The BERT model requires the input to be tokenized in a specific way. We use the tokenizer provided by the Hugging Face library.
After tokenization, the input is converted into a sequence of tokens, and the tokens are then converted into a sequence of input IDs. The input IDs are then used as input to the BERT model.
Token size of text is as follows:

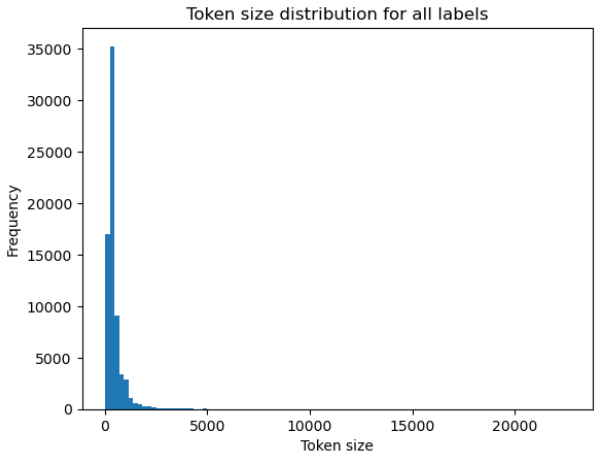


Figure 2.4: Token size of text

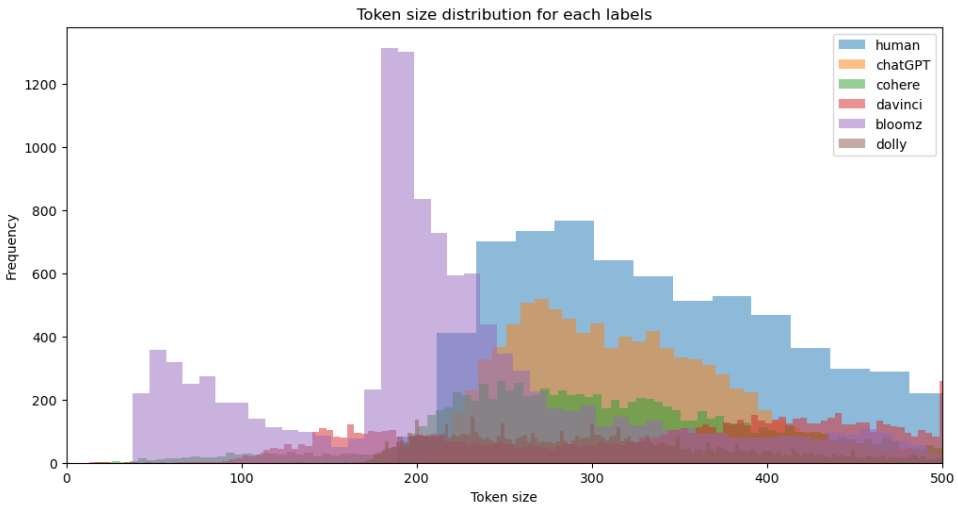


Figure 2.5: Token size of text based on class

Becuse of the limitation of RAM we have to limit the token size to 256, which makes sence according to the following figure:

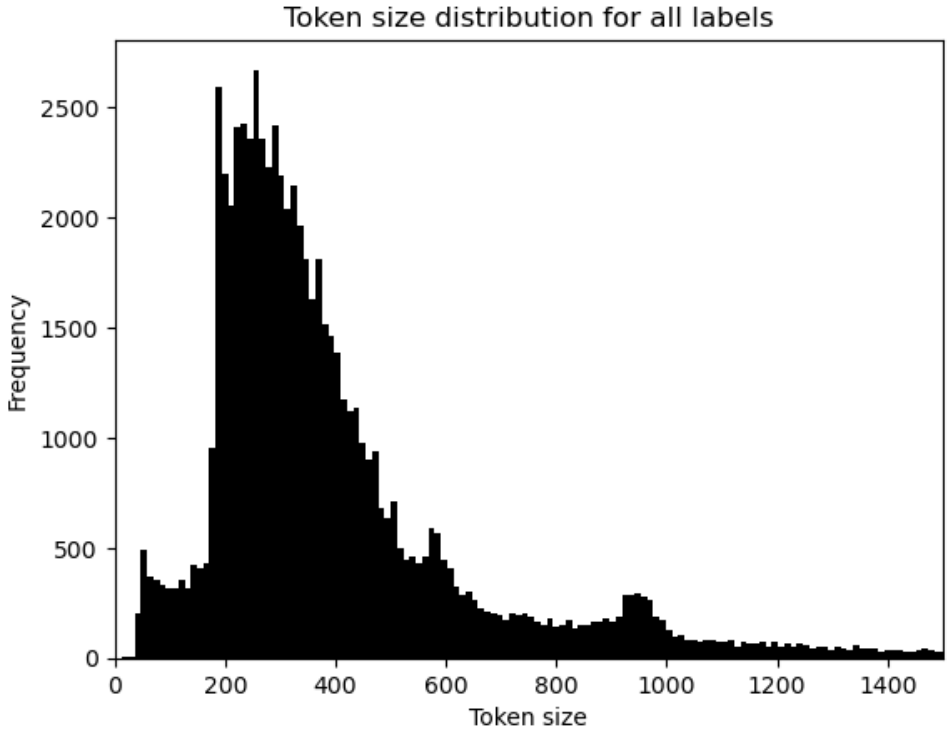


Figure 2.6: Token size of text based on class

2.3 | Data Cleaning

The dataset contains some non-expanded words (e.g. ”do not” is written as ”don’t”). We need to expand these words to make the dataset more consistent. We made a list of words named”contractions_dict” and expanded the words in the dataset using this list.
The dataset also contains some special characters and numbers. We need to remove these characters and numbers from the dataset to make it more consistent. We used the regular expression library to remove these characters and numbers from the dataset.
The dataset also contains lowercase and uppercase letters. We need to convert all the letters in the dataset to lowercase to make it more consistent. We used the lower() function to convert all the letters

in the dataset to lowercase.

2.4 | Word Cloud

We created a word cloud for every class to see the most frequent words in the dataset. The word clouds are shown in the figures below:

Figure 2.7: Word cloud for text generated by human

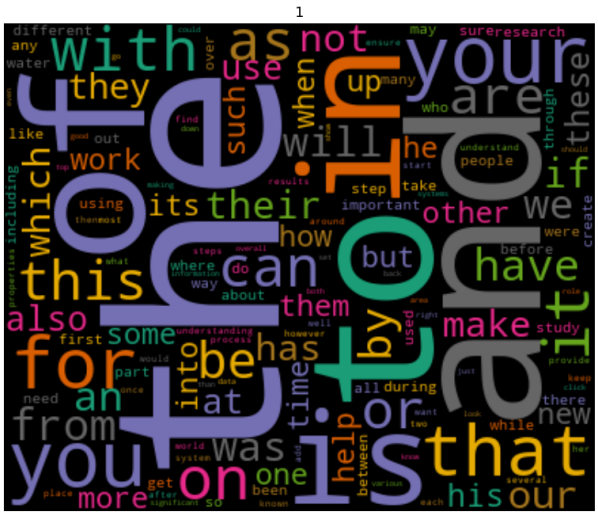


Figure 2.8: Word cloud for text generated by chatGPT

Figure 2.9: Word cloud for text generated by cohere

Figure 2.10: Word cloud for text generated by davinci

Figure 2.11: Word cloud for text generated by bloomz

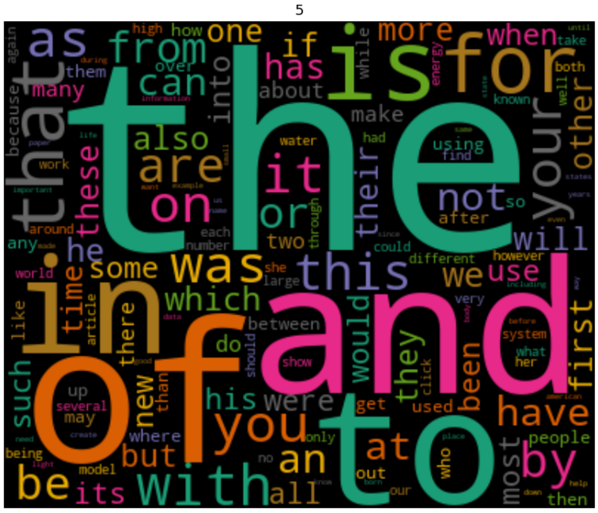


Figure 2.12: Word cloud for text generated by dolly

2.5 | Most Frequent Words

One of the most important steps in preprocessing the data is to find the most frequent occurring N-grams in the dataset. N-grams are contiguous sequences of n items from a given sample of text or speech. The most frequent occurring N-grams in the dataset are shown in the figures below:

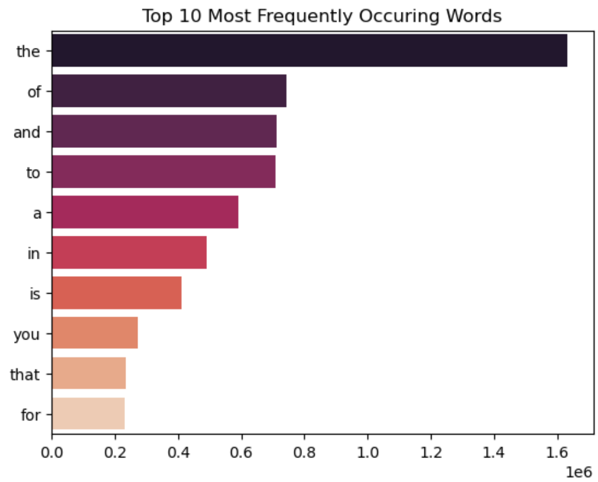


Figure 2.13: Most frequent occurring words in the dataset

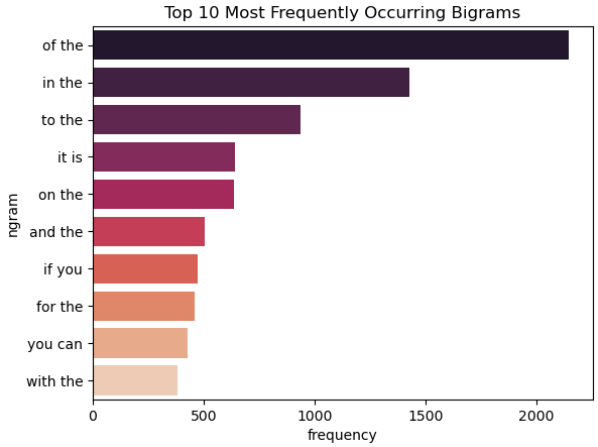


Figure 2.14: Most frequent occurring N-grams in the dataset

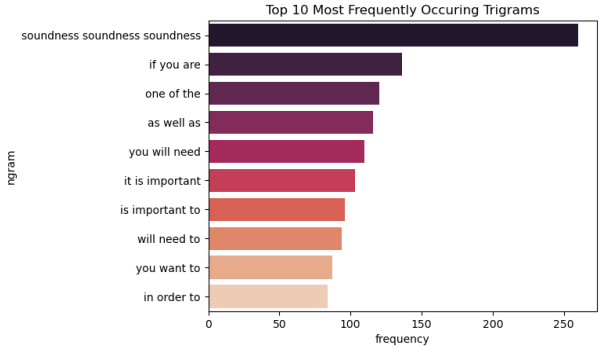


Figure 2.15: Most frequent occurring N-grams in the dataset

2.6 | Data Manifold

We used the PCA algorithm to reduce the dimensionality of the dataset to 2D. The 2D representation of the dataset is shown in the figure below:

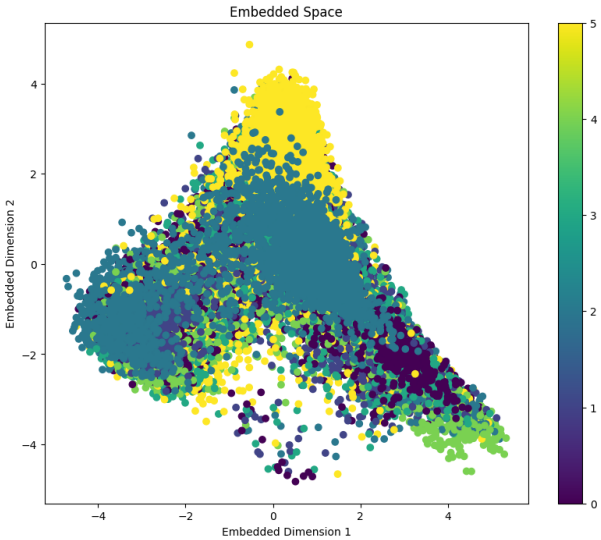


Figure 2.16: 2D representation of the train set using PCA

And the 2D representation of the test set is shown in the figure below:

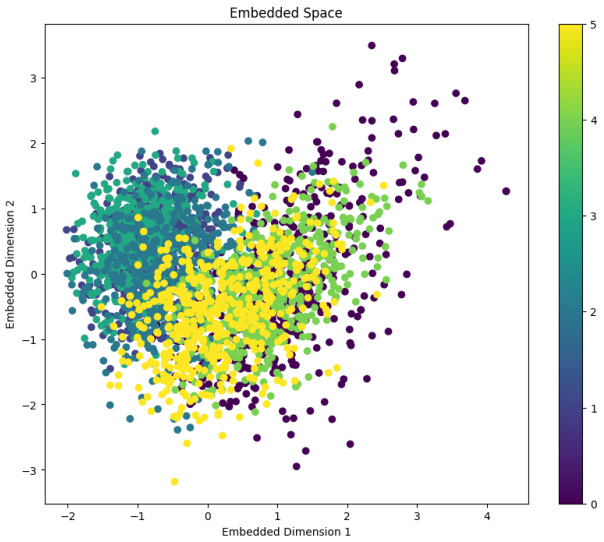
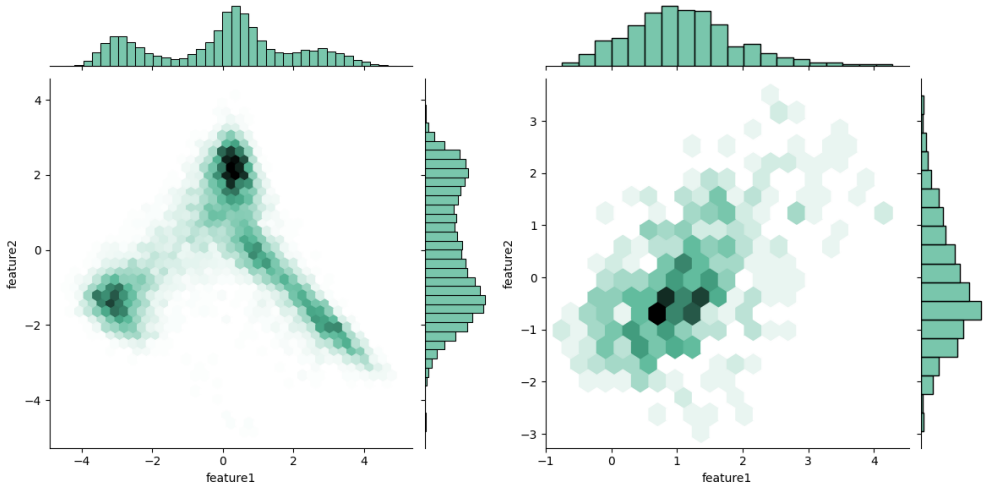


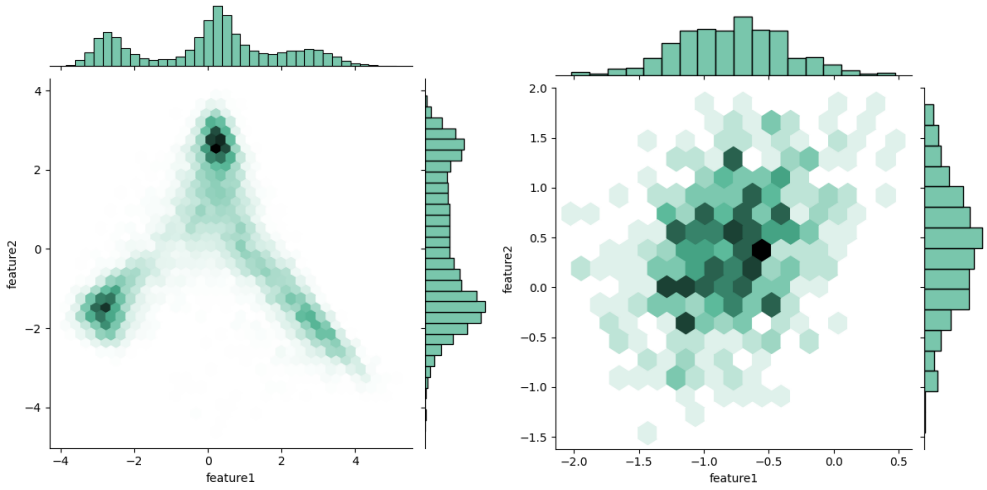
Figure 2.17: 2D representation of the test set using PCA

Also hezplot of the data based on class is as follows:



(a) human class train

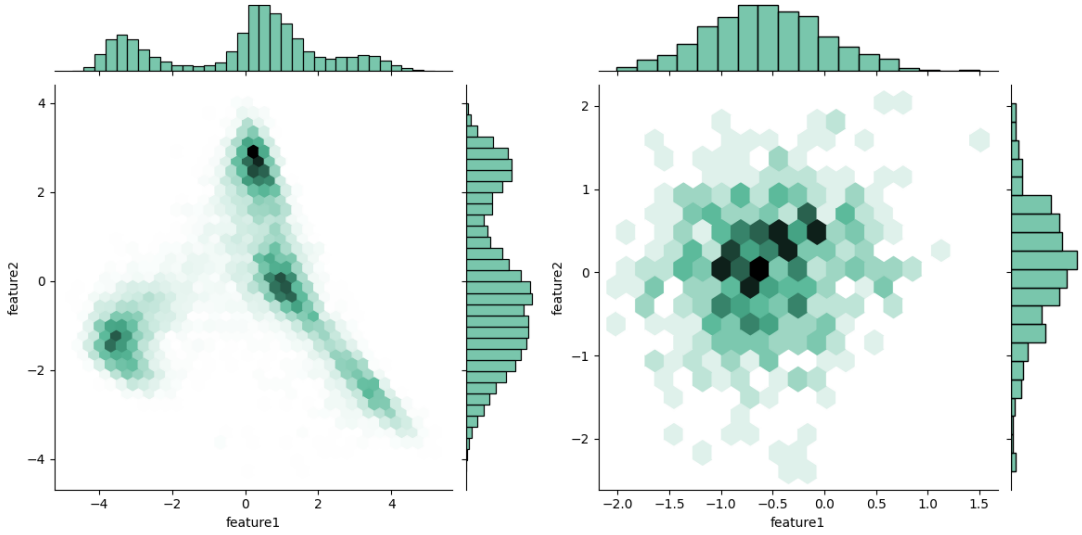
(b) human class test



(c) chatGPT class train

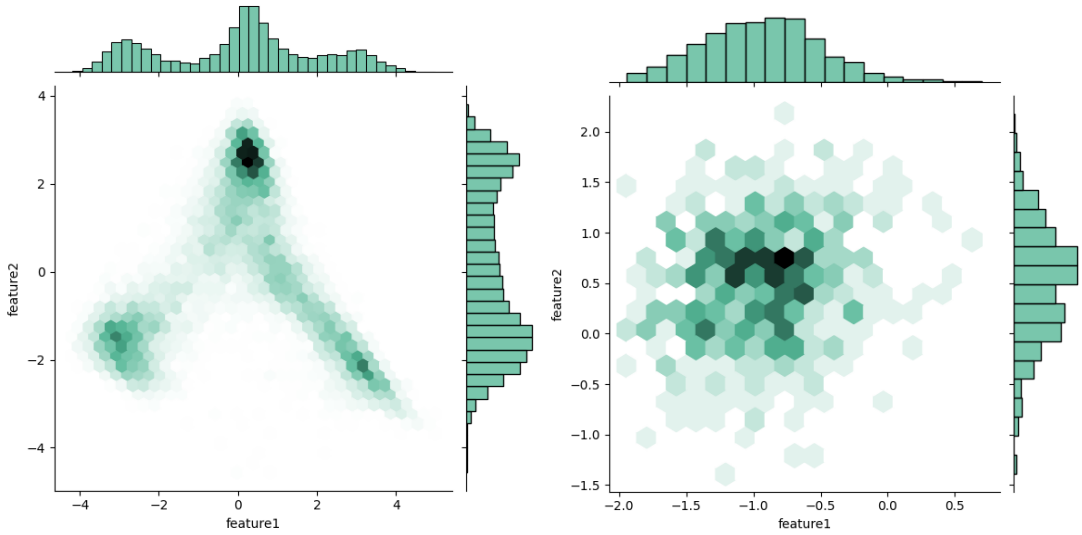
(d) chatGPT class test

Figure 2.18: Hex plot of the data based on class and train/test part i



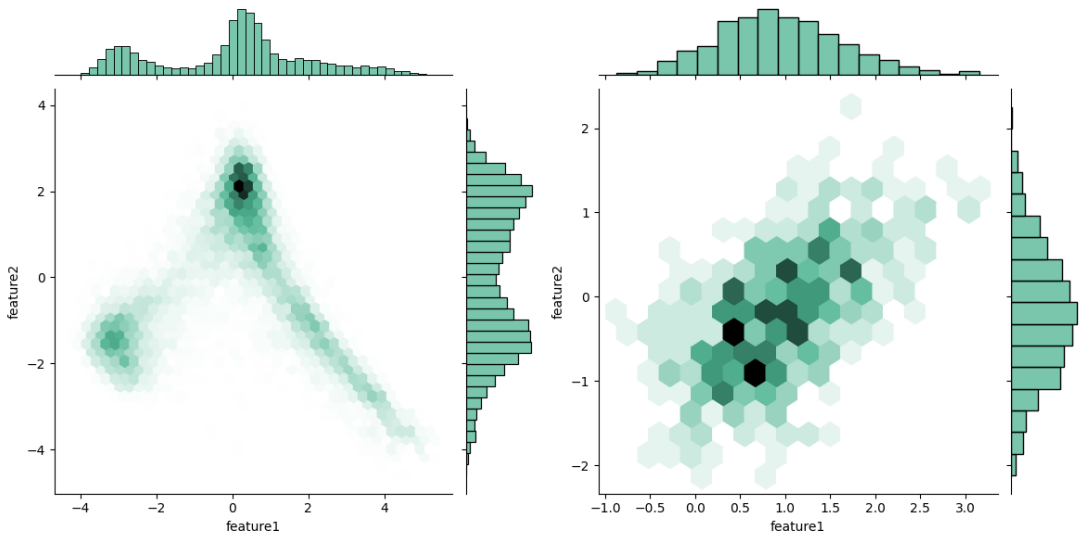
(a) cohere class train

(b) cohere class test



(c) davinci class train

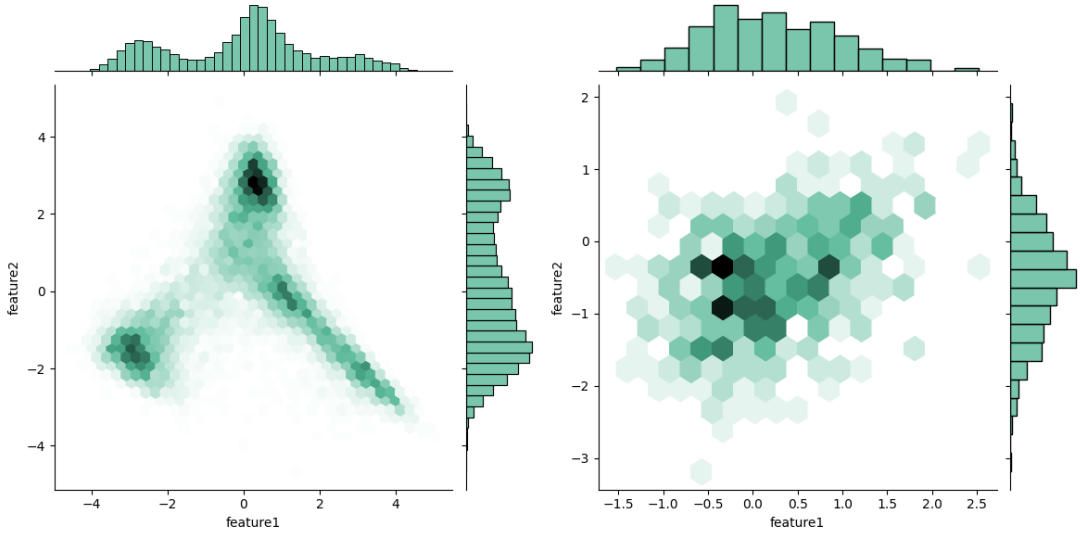
(d) davinci class test



(e) bloomz class train

(f) bloomz class test

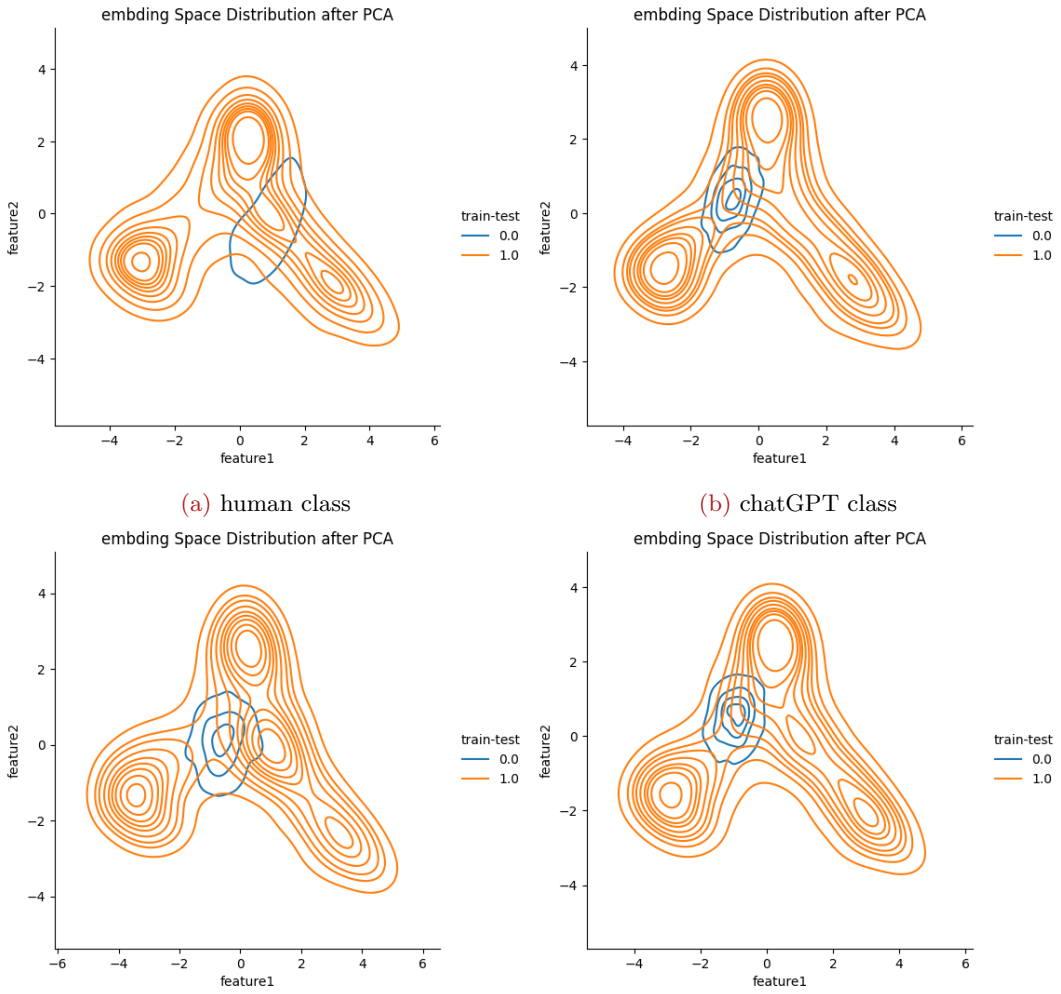
Figure 2.19: Hex plot of the data based on class and train/test part *ii*



(a) dolly class train (b) dolly class test

Figure 2.20: Hex plot of the data based on class and train/test part ***iii***

Also KL divergence of the train and test data based on class is as follows:



(a) human class (b) chatGPT class

(c) cohere class (d) davinci class

Figure 2.21: KL divergence of the train and test data based on class ***i***

(a) bloomz class (b) dolly class

Figure 2.22: KL divergence of the train and test data based on class *ii*

As we can see from the figures above, the manifold of the train and test data is not similar by any means. This is a sign of that model may not perform well on the test data. Also the KL divergence of the train and test data is not similar by any means. This is also a sign of that model may not perform well on the test data.
As we know, considering Variance and Bias, best possible hypothesis -represented by g- sometimes may not be the best hypothesis for the test data- represented by h-, and this is called overfitting.
in other words, models loss can be represented as follows:

$$L_h = L_g + \mathrm{KL}(p_{data} || p_{model}) \tag{2.1}$$

where L_h is the loss of the model on the test data, L_g is the loss of the model on the train data, p_{data} is the distribution of the train data and p_{model} is the distribution of the model. As we can see from the figures above, the KL divergence of the train and test data is not similar by any means. This fact ensures that the model may not perform well on the test data.

3 | Fine-tuning BERT

Fine-tuning is a common technique used to adapt pre-trained models to specific tasks. In the context of BERT, fine-tuning involves training the model on a labeled dataset to learn task-specific information. The fine-tuning process typically involves updating the weights of the pre-trained model using backpropagation and gradient descent. Fine-tuning BERT has been shown to be effective for a wide range of NLP tasks, including question answering, text classification, and named entity recognition devlin2019bert [1]. In this section, we describe the fine-tuning process for BERT and discuss some best practices for fine-tuning BERT on specific tasks.

3.1 | Fine-tuning process

The fine-tuning process for BERT involves several steps, including data preprocessing, model configuration, and training. The following steps are typically involved in the fine-tuning process:

3.2 | Data preprocessing

The first step in the fine-tuning process is to preprocess the labeled dataset. This typically involves tokenizing the text, converting it to the input format expected by BERT, and splitting the dataset into training, validation, and test sets. Tokenization is the process of splitting the text into individual words or subwords, and converting each word or subword into an integer index that corresponds to a token in the BERT vocabulary. The input format expected by BERT consists of three parts: the input ids, the attention mask, and the token type ids. The input ids are the integer indices of the tokens in the BERT vocabulary, the attention mask is a binary mask that indicates which tokens are padding tokens and which are not, and the token type ids are used to distinguish between different segments of the input, such as the question and the answer in a question answering task.
The dataset given is imbalanced and contains 5 classes. For training more efficeintly and effectively, we seprarate the dataset into 5 different datasets, each containing one class. Then we used 0.01%, 0.05%, 0.1% and 0.5% of every class to train the model.

3.3 | Model configuration

The next step in the fine-tuning process is to configure the BERT model for the specific task. This typically involves selecting the pre-trained BERT model to use, adding a classification layer on top of the BERT model, and setting hyperparameters such as the learning rate, batch size, and number of training epochs.
We used the ruberta-large model from the transformers library, we added a classification layer named BertForClassification on top of the BERT model.

3.4 | Training

The final step in the fine-tuning process is to train the BERT model on the labeled dataset. This typically involves using backpropagation and gradient descent to update the weights of the BERT model, and monitoring the model's performance on the validation set to prevent overfitting. Note that here we used SubtaskB_Dev as the validation set.

Psudo code of the fine-tuning process is shown in the algorithm below:

```
Input: Pre-trained BERT model, Training data
Output: Fine-tuned BERT model
Initialize BERT model parameters;
Initialize AdamW optimizer with learning rate;
for each epoch do
    for each batch in training data do
        Compute loss using BERT model and batch;
        Compute gradients of loss with respect to BERT model parameters;
        Update BERT model parameters using AdamW optimizer;
    end
end
```

Algorithm 1: Fine-tuning BERT with AdamW

We used the AdamW optimizer with a learning rate of $1e-6$, a batch size of 33, and trained the model for 10 epochs. It's worth mentioning after we saw the model is overfitting, we used early stopping to prevent it and trained the model for 7 epochs.

3.5 | Results

We trained the model on the dataset and tested it on the test set. The results are shown in the table below:

Class	0.01%	0.05%	0.1%	0.5%
1	26.00 %	51.07 %	62.70 %	0.0
2	39.43 %	60.10 %	62.83 %	0.0
3	43.47 %	53.60 %	58.47 %	0.0
4	44.20 %	60.30 %	61.60 %	0.0
5	47.20 %	58.63 %	62.90 %	0.0
6	48.87 %	61.87 %	60.63 %	0.0
7	50.03 %	60.10 %	62.10 %	0.0
8	54.77 %	Nan	Nan	Nan
9	55.57 %	Nan	Nan	Nan
10	54.73 %	Nan	Nan	Nan

Table 3.1: The accuracy of the model on the test set on every epoch for different percentages labeled dataset.

Results of the model on the test set are shown in the figures below:

```
Validation Accuracy: 0.5473333333333333
Classification Report:
              precision    recall  f1-score   support

           0       0.58      0.61      0.60       481
           1       0.16      0.68      0.26       117
           2       0.86      0.34      0.49      1277
           3       0.00      0.00      0.00        49
           4       0.82      0.73      0.77       562
           5       0.86      0.84      0.85       514

    accuracy                           0.55      3000
   macro avg       0.55      0.53      0.49      3000
weighted avg       0.77      0.55      0.60      3000
```

Figure 3.1: The accuracy and F1 score of the model on the test set for 0.01 percent of the labeled dataset.

```
Validation Accuracy: 0.601
Classification Report:
              precision    recall  f1-score   support

           0       0.39      0.92      0.55       213
           1       0.72      0.72      0.72       502
           2       0.89      0.43      0.58      1033
           3       0.05      0.08      0.06       324
           4       0.96      0.86      0.91       558
           5       0.59      0.79      0.68       370

    accuracy                           0.60      3000
   macro avg       0.60      0.63      0.58      3000
weighted avg       0.71      0.60      0.62      3000
```

Figure 3.2: The accuracy and F1 score of the model on the test set for 0.05 percent of the labeled dataset.

```
Validation Accuracy: 0.621
Classification Report:
              precision    recall  f1-score   support

           0       0.43      0.98      0.59       217
           1       0.69      0.88      0.77       393
           2       0.91      0.43      0.58      1053
           3       0.04      0.05      0.04       328
           4       1.00      0.83      0.90       604
           5       0.67      0.82      0.74       405

    accuracy                           0.62      3000
   macro avg       0.62      0.67      0.61      3000
weighted avg       0.73      0.62      0.64      3000
```

Figure 3.3: The accuracy and F1 score of the model on the test set for 0.1 percent of the labeled dataset.

```
Validation Accuracy: 0.6633333333333333
Classification Report:
              precision    recall  f1-score   support

           0       0.38      0.99      0.55       193
           1       0.90      0.82      0.86       547
           2       0.86      0.45      0.59       948
           3       0.05      0.12      0.07       231
           4       1.00      0.82      0.90       607
           5       0.79      0.83      0.81       474

    accuracy                           0.66      3000
   macro avg       0.66      0.67      0.63      3000
weighted avg       0.79      0.66      0.70      3000
```

Figure 3.4: The accuracy and F1 score of the model on the test set for 0.5 percent of the labeled dataset.

Plots of the accuracy and F1 score of the model on the validation set for different percentages of the labeled dataset are shown in the figures below:

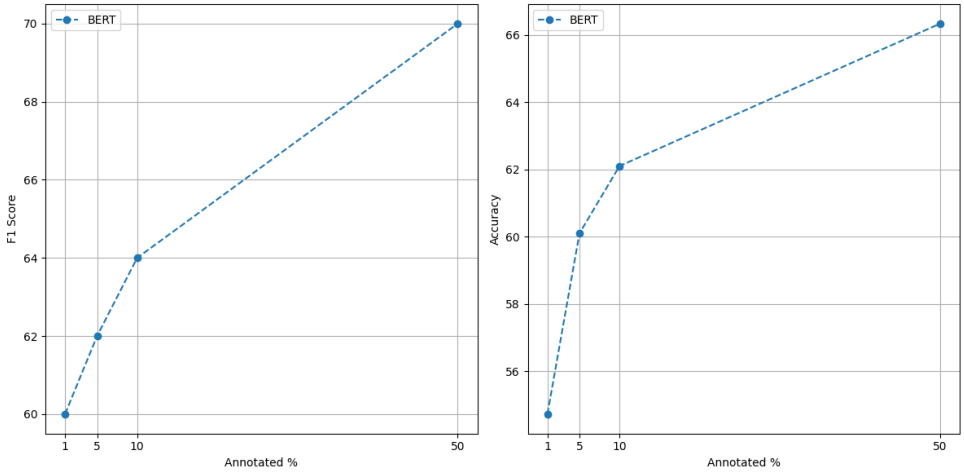


Figure 3.5: The accuracy and F1 score of the model on the validation set for different percentages of the labeled dataset.

Also confusion matrices of the model on the test set for different percentages of the labeled dataset are shown in the figures below:

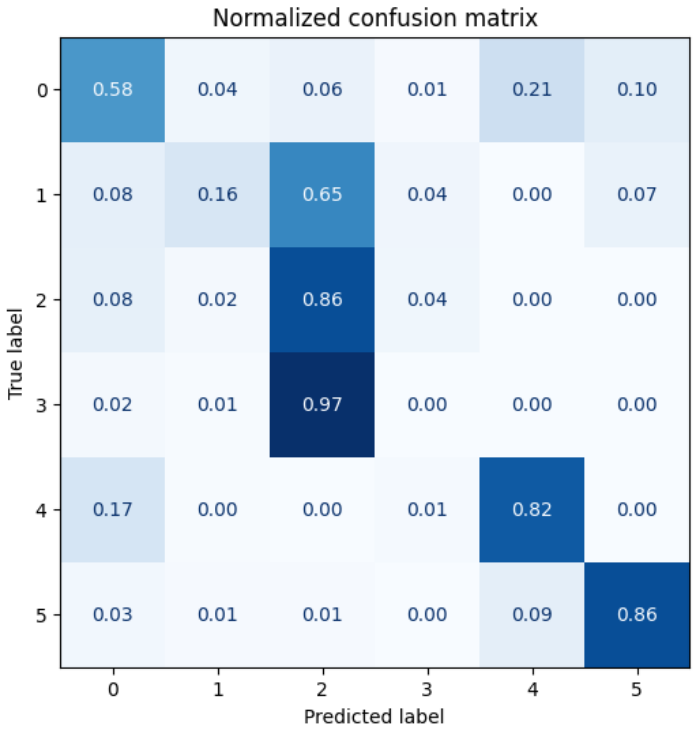


Figure 3.6: The confusion matrix of the model on the test set for 0.01 percent of the labeled dataset.

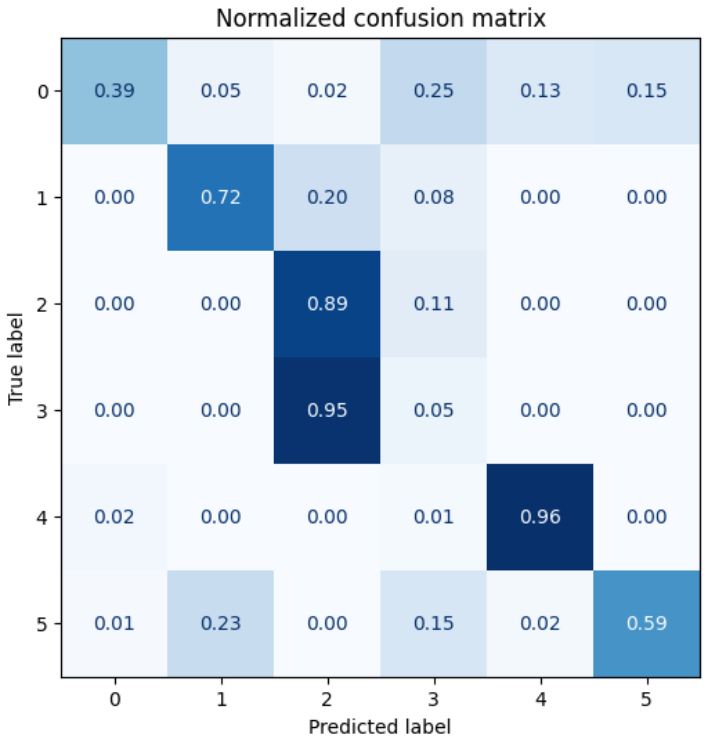


Figure 3.7: The confusion matrix of the model on the test set for 0.05 percent of the labeled dataset.

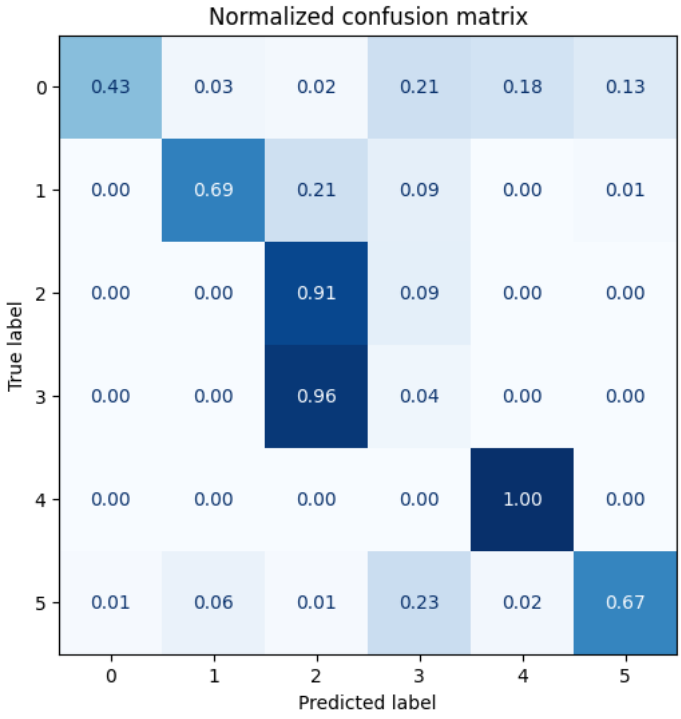


Figure 3.8: The confusion matrix of the model on the test set for 0.1 percent of the labeled dataset.

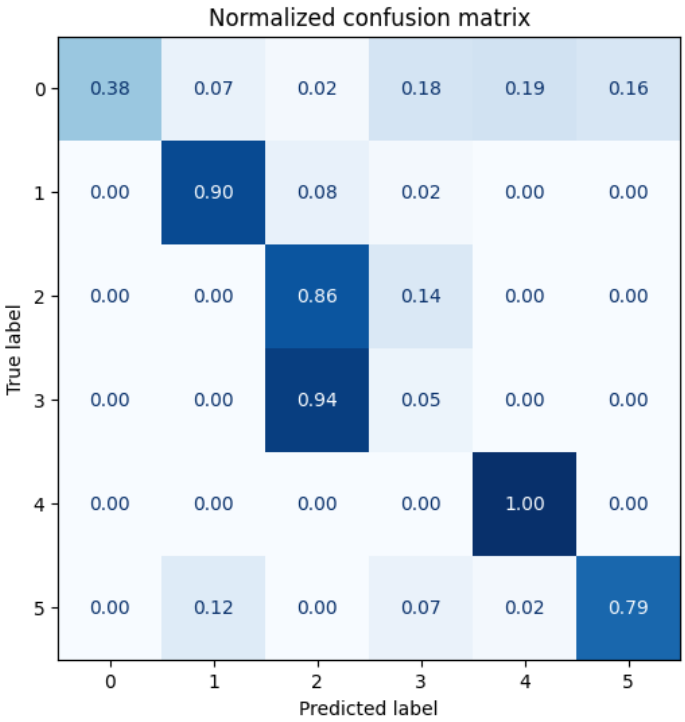


Figure 3.9: The confusion matrix of the model on the test set for **0.5** percent of the labeled dataset.

4 | Adaptor

Adapters are a method of fine-tuning transformer models, such as BERT, that are more parameter-efficient than fine-tuning the entire model. Instead of updating all the parameters of the model, adapters only update a small fraction of the parameters, which are inserted in between the layers of the model. The main effects of adding adapters to BERT models are :

- Parameter Efficiency: Adapters are a more parameter-efficient way of fine-tuning models, as they only update a small fraction of the parameters.
- Faster Training: Adapters allow for faster training, as they only update a small fraction of the parameters.
- Better Generalization: Adapters allow for better generalization, as they are trained on a wide range of tasks and can be transferred to new tasks.

4.1 | Adaptor Architecture

First introduced by houlsby2019parameterefficient[2], the adapter architecture is a method of fine-tuning transformer models that is more parameter-efficient than fine-tuning the entire model. Instead of updating all the parameters of the model, adapters only update a small fraction of the parameters, which are inserted in between the layers of the model. The adapter architecture consists of two main components: the adapter module and the adapter bottleneck. The adapter module is a small neural network that is inserted in between the layers of the model, and the adapter bottleneck is a set of parameters that are shared across all the layers of the model. The adapter module is used to fine-tune the model on a specific task, and the adapter bottleneck is used to share the parameters of the adapter module across all the layers of the model. This allows for faster training and better generalization, as the adapter module can be transferred to new tasks and the adapter bottleneck can be shared across all the layers of the model. Then we add a classification layer on top of the BERT model to classify the text into one of the 6 classes. The adapter architecture in python looks like this:

```
def BertWithAdapter():

    model = AutoAdapterModel.from_pretrained(model_name, config=config).to(device)
    # Add a new adapter
    model.add_adapter(adapter_name)

    model.set_active_adapters = adapter_name

    # Add a matching classification head
    model.add_classification_head(adapter_name, num_labels=num_classes,
                          overwrite_ok=True)
    # Activate the adapter
    model.train_adapter(adapter_name)

    return model
```

4.2 | Results

We trained the model on the dataset and tested it on the test set. The results are shown in the tables below:

Class	0.01%	0.05%	0.1%	0.5%
1	28.83 %	55.10 %	56.53 %	61.60
2	52.96 %	63.20 %	61.46 %	72.16
3	51.73 %	59.20 %	63.26 %	70.66
4	56.76 %	60.06 %	63.23 %	70.00

Table 4.1: The accuracy of the model on the test set on every epoch for different percentages labeled dataset.

Class	0.01%	0.05%	0.1%	0.5%
1	15.53 %	55.45 %	54.38 %	58.79
2	46.39 %	59.38 %	56.28 %	66.82
3	46.95 %	56.96 %	60.39 %	67.54
4	52.06 %	58.17 %	60.00 %	66.42

Table 4.2: The F1 score of the model on the test set on every epoch for different percentages labeled dataset.

Accuracy and F1 score of the model on the test for Bert model with adapter are shown in the figures below:

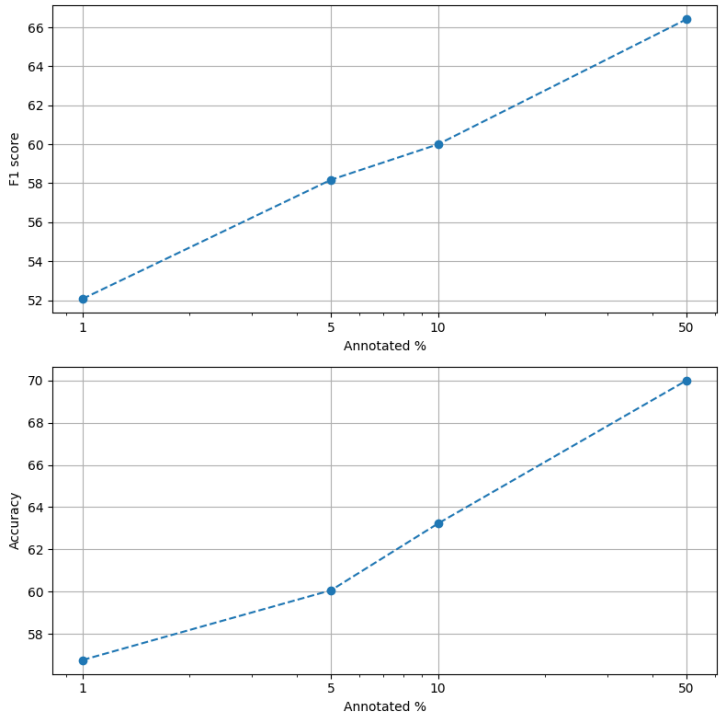


Figure 4.1: The accuracy and F1 score of Bert model with adapter

5 | GAN-BERT Implementation

In this section, we describe the implementation of GAN-BERT. We first describe the architecture of the generator and the discriminator. We then describe the training process and the hyperparameters used in the experiments.

5.1 | Architecture of the Generator 1

First generator is a simple feedforward neural network. The input is fed to a linear layer followed by a LeakyReLU activation function. Then we use dropout to prevent overfitting. The output of the dropout layer is fed to another linear layer. Dropout rate is set to 0.2 and the LeakyReLU slope is set to 0.2. The output of the second linear layer is the output of the generator, which is a tensor of size 768 given to the discriminator.

5.1.1 | Input

The input to the generator is a gaussian noise vector of size 100.

5.1.2 | Hidden Layers

The generator has two hidden layers. The first hidden layer gets a input of size 100 and outputs a tensor of size 256. This tensor is then fed to the second hidden layer, which outputs a tensor of size 768, which is the same as the output size of the bert_based model.

5.2 | Architecture of the Generator 2

The generator is a BERT model that is fine-tuned on the data. The input to the generator bag of words (BoW) representation of the input text.

5.2.1 | BoW Representation

The BoW representation is obtained by tokenizing the input text using the BERT tokenizer. First, the input text is tokenized into subwords. Then, the subwords are converted into word embeddings using the ”distilbert-base-uncased” model. Then for input of Genereator, we took samples from the tokenized text and converted them into BoW representation.

5.2.2 | Fine-tuning BERT

The fine-tuning process is similar to the one described in devlin2019bert [1].

5.3 | Architecture of the Discriminator

The discriminator is a simple feedforward neural network.
The architecture of the discriminator is as follows:

5.3.1 | Input

The input to the discriminator is a tensor of size 768, which is first fed to dropout layer to prevent overfitting.

5.3.2 | First Hidden Layer

The output of the dropout layer is then fed to a linear layer with 512 output features, followed by a LeakyReLU activation function. Then we use dropout to prevent overfitting.

5.3.3 | Second Hidden Layer

The output of the dropout layer is then fed to a linear layer with 256 output features, followed by a LeakyReLU activation function. Then we use dropout to prevent overfitting.

5.3.4 | Third Hidden Layer

The output of the dropout layer is then fed to a linear layer with 256 output features, followed by a LeakyReLU activation function. Then we use dropout to prevent overfitting.

5.3.5 | Output Layer

The output of the dropout layer is then fed to a linear layer with 7 output features, followed by a sigmoid activation function. The output of the sigmoid function represents the probability that the input is real or fake and the class of the input.

5.4 | Training Process

The training process involves training the generator and the discriminator in an alternating fashion. Psudo code of the training process is shown in the algorithm below:

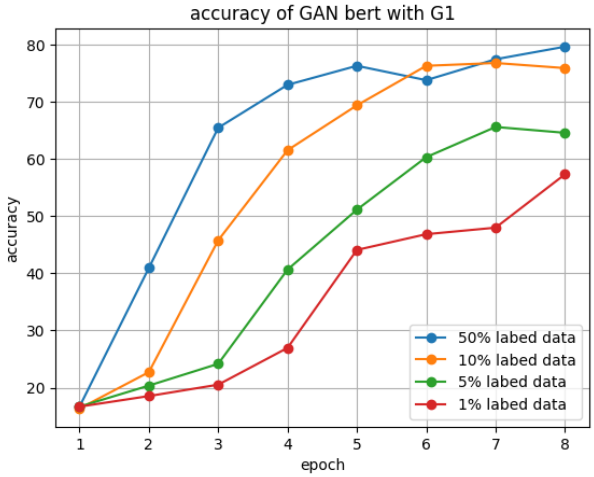
```
Input: Pre-trained BERT model, Training data
Output: Fine-tuned BERT model
Initialize BERT model parameters;
Initialize AdamW optimizer with learning rate;
for each epoch do
    for each batch in training data do
        Compute loss using BERT model and batch;
        Compute gradients of loss with respect to BERT model parameters;
        Update BERT model parameters using AdamW optimizer;
    end
end
```

Algorithm 2: Fine-tuning BERT with AdamW

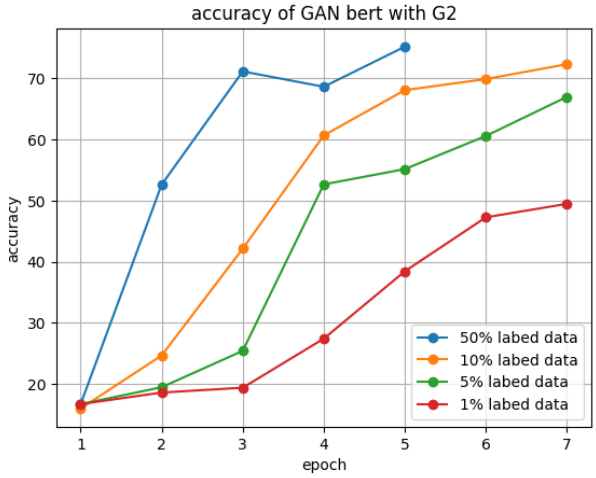
5.5 | Rsults

5.6 | Validation

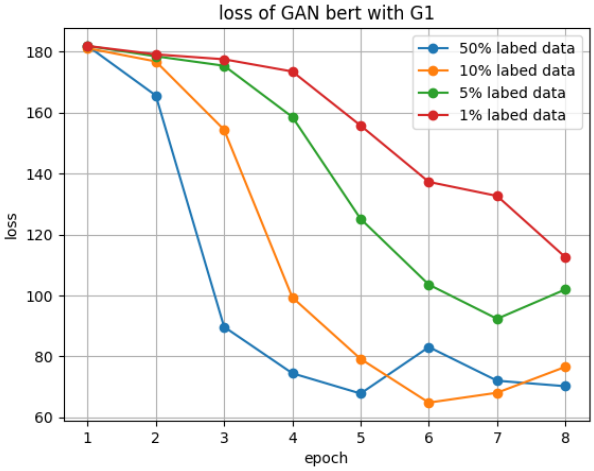
Accuracy, loss and mcc of the model with generator 1 and generator 2 on validation data is shown in the figures below:



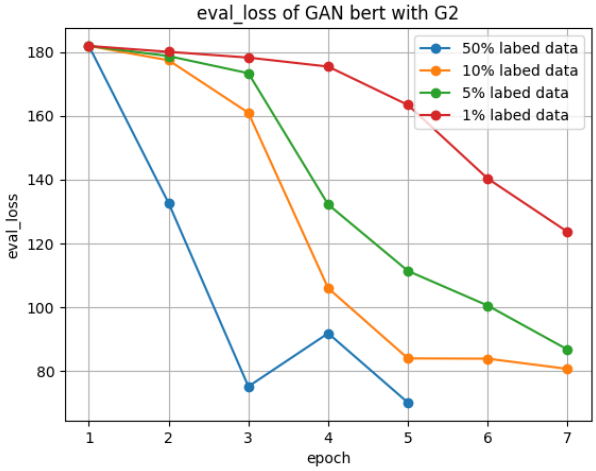
(a) Accuracy of the model with generator 1



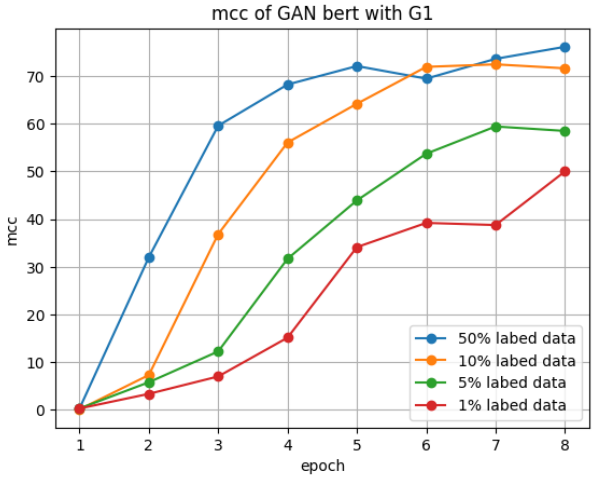
(b) Accuracy of the model with generator 2



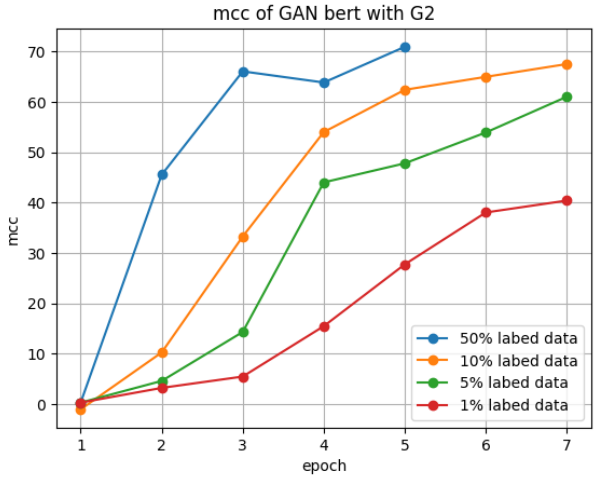
(c) Loss of the model with generator 1



(d) Loss of the model with generator 2



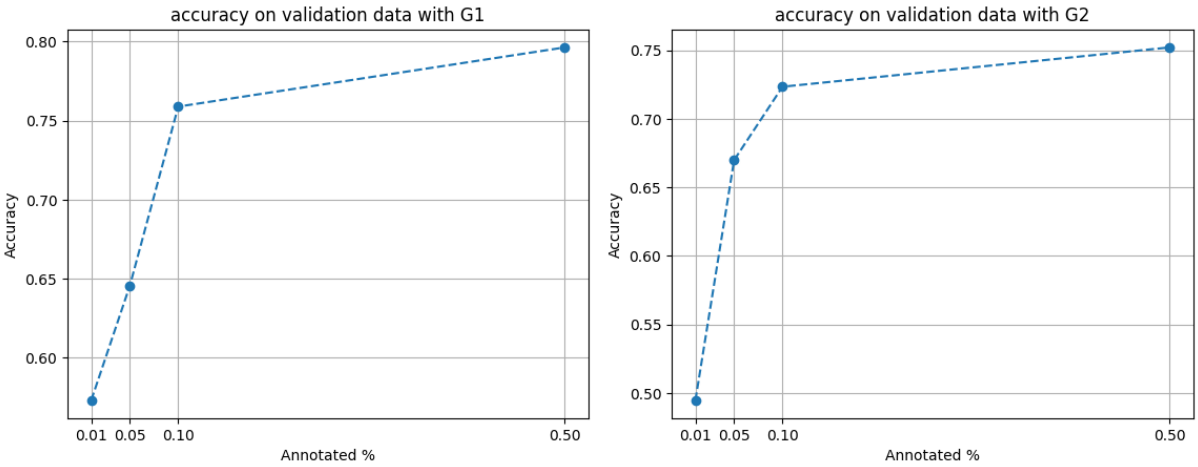
(e) MCC of the model with generator 1



(f) MCC of the model with generator 2

Figure 5.1: Accuracy, loss and mcc of the model with generator 1 and generator 2

Also accuracy on validation based on labeld data is shown in the figure below:

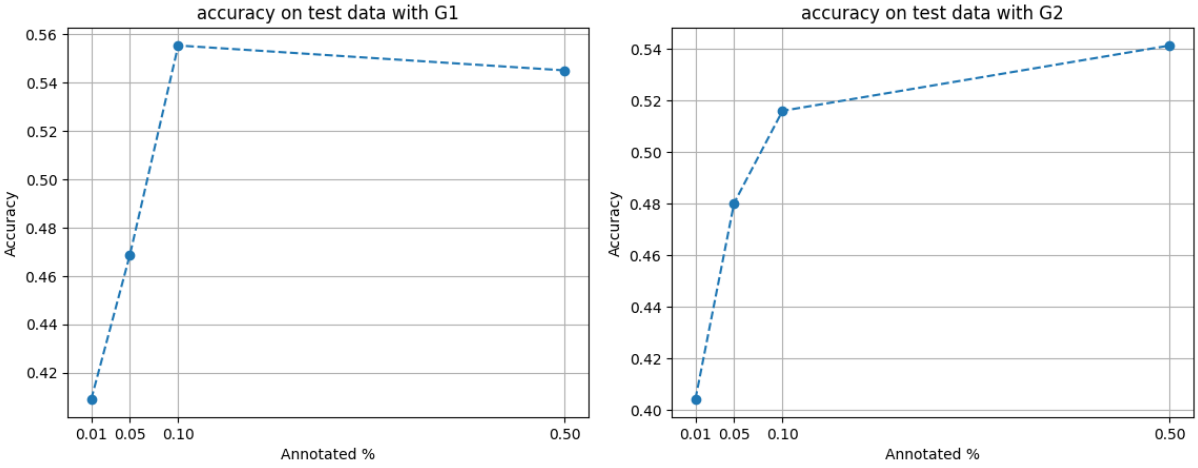


(a) Accuracy on validation based on labeld data (b) Accuracy on validation based on labeld data

Figure 5.2: Accuracy on validation based on labeld data

5.7 | Testing

Accuracy on testing based on labeld data is shown in the figure below:



(a) Accuracy on testing based on labeld data (b) Accuracy on testing based on labeld data

Figure 5.3: Accuracy on testing based on labeld data

MCC The Matthews correlation coefficient , is a measure of the quality of classifications in machine learning. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. It's defined in the range from -1 to 1, with 1 being a perfect prediction, 0 being the result of a random prediction, and -1 indicating total disagreement between prediction and observation

6 | Further Ideas

6.1 | More epochs, more labeled data

We trained the GAN-BERT with generator 1 on 10 epochs, and the results were more than 80% accuracy. We can train the model with more epochs and more labeled data to achieve better results.

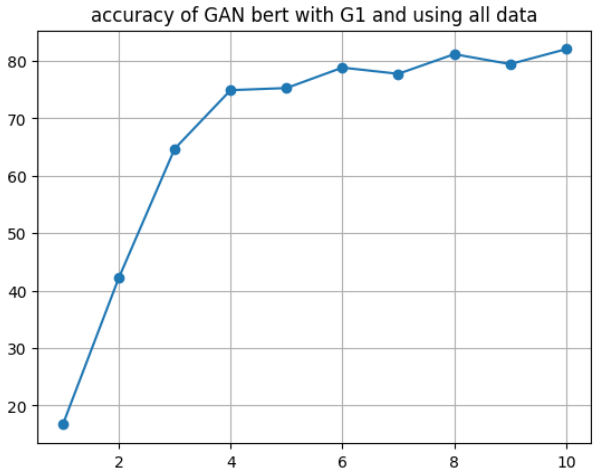


Figure 6.1: More epochs, more labeled data

For achiving better results (mainly 80% accuracy) in the GAN-BERT model, the following ideas can be implemented:

6.2 | Balance between discriminator & generator

In Generative Adversarial Networks (GANs), the discriminator and generator are in a constant competition. Issues like mode collapse and gradient diminishing are often attributed to an imbalance between these two components. Efforts to improve GANs often focus on balancing the loss between the generator and the discriminator, but finding an effective solution has proven challenging.
On the other hand, some researchers challenge the feasibility and desirability of balancing these networks. A well-trained discriminator gives quality feedback to the generator anyway. Also, it is not easy to train the generator to always catch up with the discriminator. Instead, we may turn the attention into finding a cost function that does not have a close-to-zero gradient when the generator is not performing well.

$$-\nabla_{\theta_g} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \to 0 \quad \textit{change to} \quad \nabla_{\theta_g} \log\left(D\left(G\left(z^{(i)}\right)\right)\right)$$

Figure 6.2: Balance between discriminator & generator

Refrence: towardsdatascience.com

6.3 | ALBERT

Al BERT is a language model that is designed to understand and generate human language. It is based on the Transformer architecture and is trained using a large corpus of text. Al BERT has been shown to achieve state-of-the-art performance on a variety of natural language processing tasks, including text classification, named entity recognition, and question answering. The model is also capable of generating human-like text, making it a powerful tool for natural language generation. Al BERT has been used to generate realistic news articles, stories, and poems, and has even been used to write code. The model has also been used to generate human-like responses in chatbots and virtual assistants. Refrence: Al BERT[3]

6.4 | BERT with Adapters

BERT with Adapters is a simple and effective way to improve the performance of BERT. The model is trained on a large corpus of text and then fine-tuned on a smaller dataset. The model is then used to generate predictions for the smaller dataset, which are then used to train a new model. This process is repeated until the model reaches a desired level of performance. The BERT with Adapters model has been shown to improve the performance of BERT on a variety of tasks, including text classification, named entity recognition, and question answering.

7 | References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[2] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp, 2019.

[3] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2020.