EEE 586: Final Report for the Term Project Text Classification with Distil-BERT Enhanced GNNs

Tuna Alikaşifoğlu*, Arda Can Aras†
*†Dept. of Electrical and Electronics Engineering
*†Bilkent University
{*t.alikasifoglu, †can.aras}@bilkent.edu.tr

***Abstract*—In recent years, the amount of text based complex documents increased significantly, along with the importance of ability to classify texts as efficiently and accurately as possible. We had traditional algorithms to tackle the text classification problem, but with the progressive computational power, many machine learning, especially deep learning, based solutions surpassed the human capabilities in this area. Lately, BERT based models overshadow the remaining approaches in the text classification area. On the other hand, we have the trending topic of graph neural networks (GNNs). They are geometric extensions, i.e., extensions of traditional neural network architectures to the graph domain, and graph-structured data. In recent years, there has been some developments in the text classification area, especially using graph convolutional networks (GCNs). In this project, a brief overview of text classification problem and of graph neural networks are provided. Text classification overview covers the fundamental steps of a text classification task with the indication of GNN integration, where the GNN overview provides the reasons, challenges and how-tos of utilizing GNNs. Then, we provide an overview of the related work that tackles the text classification problem with the GNNs, while comparing several approaches. Finally in this project, we present a text classification approach that combines the BERT models, that provide semantic and contextual information, with the GCN models, that provide structural and global information.**[§]

***Index Terms*—Text Classification, Graph Neural Networks (GNNs), Graph Convolutional Networks (GCNs), BERT, Distil-BERT.**

I. Introduction

In this project, we have scrutinize the classic natural language processing task of text classification along with the trending approach of graph neural networks (GNNs). In this context, we present a text classification approach that combines the BERT models, that provide semantic and contextual information, with the GCN models, that provide structural and global information. In this project, we first provide a comprehensive related work analysis in Section II. Then, in Section III, we present our proposed method of combining BERT based approaches with GNN based approaches in the context of text classification. Then, we present our results and compare them with the baselines in Section IV. Finally, we provide a discussion and possible future directions in Section V.

We have divided the related work analysis into four parts in Section II. First, we provide an overview regarding the traditional text classification (Section II-A) and we provide an overview to graph neural networks (Section II-B). Both of these overviews are provided based on several survey papers for text classification [1]–[3] and for graph neural networks [4]–[7]. In Section II-A, we provide a basic definition for text classification, then we disintegrate text classification process to five steps to analyse the overall process, and we point out where the integration of graph neural networks can be made. In Section II-B, we provide an introduction to graph neural networks. Then, we analyse why we need graph neural networks, we provide several challenges on the route of obtaining graph based deep learning frameworks, and finally we provide bare-minimum steps in order to obtain a graph neural network based architecture, based on the aforementioned surveys, in this subsection. After this step, we are at a stage that we have provided

[§]The source code of the project is provided in its GitHub Page background information both on the preliminary aspects of the overall term project, which are text classification and graph neural networks.

In Section II-C, we provide previous work directly related to our own topic of graph neural networks related to solution of the classical natural language processing task of text classification. Section II-C of the survey aims to investigate the related and previous work in the text classification with graph neural network field possibly with a historical order. All the papers mentioned in this section related with each other by references. Surprisingly, this structure can also be viewed as graph where each paper is node and and edges of the graphs as citation. There are also studies on this topic to predict structured citation trend [8]. Most of the papers in the Section II-C did not directly proposed to solve text classification task. However, nearly all of the papers presented Section II-C used text classification bench mark data sets to evaluate model performance. Finally, in Section II-D, we present the similar approaches that combine BERT architecture with GNNs for text classification [9]–[13].

In Section III, we present our approach, and give further details that how we implemented the proposed method, by also comparing with the related work discussed in Section II. We provide how we are generating both BERT & GCN embeddings and how we are combining these embeddings to generate a better representation for documents to obtain a text classification model. Then, we present the results of the proposed model in Section IV, and compare them with the baselines discussed in Section II.

Finally, in Section V, we provide a discussion that we comment on the obtained results, issues that we had encountered, and possible future directions for the project.

II. Related Work

A. Text Classification

In [2], *text classification* (text categorization) is defined as the "procedure of designating predefined labels for the text". The task is to assign labels or tags to the text based knowledge, i.e., textual units such as sentences, paragraphs and documents, where the labels are usually defined by humans, but can also be defined by the machine. This task is a fundamental part of Natural Language Processing (NLP), and it is significant to its applications such as sentiment analysis, question answering, text summarization, etc.. Text classification task can be partitioned into five phases as preprocessing, feature extraction, dimensionality reduction (optional), classifier selection and evaluation:

1) Preprocessing: Text preprocessing is a crucial prerequisite for a successful feature extraction, and summarized in [1] as follows. The input of the text classification frameworks consists of raw text data, which are in the form of a sequence of sentences. In this step, "cleaning" of the text datasets is performed to transform the data into a form that is suitable for feature extraction. The cleaning process is usually performed by tokenization, capitalization, slang and abbreviation handling, noise removal, spelling correction, stemming and lemmatization.

2) Feature Extraction: After preprocessing step, another crucial step, feature extraction step is necessary. In [1], this step explained as follows. Two common methods of text based feature extraction are weighted word and word embedding techniques. In the weighted word aspect, we have old techniques like bag-of-words and term frequency-inverse document frequency (TF-IDF). In the relatively recent aspect, we have the word embedding techniques like *word2vec*, *GloVe*, *FastText*, etc.

3) Dimensionality Reduction: The dimensionality reduction is an optional step of a text classification task, but based on the size of the dataset, it may be a must to have a computable result. In this aspect of the task, we try to reduce the dimensionality of the feature space while preserving the information of the original features space. Some possible dimensionality reduction techniques provided in [1] include (principal / independent) component analysis, linear discriminant analysis, non-negative matrix factorization, random projection, autoencoder and stochastic neighbor embedding.

4) Classifier Selection: As it is stated in [2], selecting the optimal classifier is the most important aspect of a text classification task. Currently we have both traditional and deep learning oriented classifiers. The traditional classifiers are based on the statistical analysis of the training data, and the deep learning classifiers are based on the neural networks. The main distinction between the traditional and deep learning based approaches can be described as follows: Good feature extraction methodology is crucial for the traditional classifiers. They obtain sample features by artificial methods and then make classifications based on these features. Hence, the performance of the traditional classifiers are mainly restricted by feature extraction. On the other hand, by making feature mapping via nonlinear transformations a part of the learning process, deep learning based classifier selection can integrate feature extraction aspect into the model fitting process.

Examples of both traditional and deep learning based approaches are provided in [1]–[3]: Some traditional classifiers are logistic regression, (kernel) support vector machine, Naive Bayes, k-nearest neighbors, decision tree, random forest, etc. On the other hand, the deep learning classifiers are usually based on the neural networks, such as deep feed forward neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), lately attention and transformer based models such as BERT [14] variations and fine tuning pre-trained language models [15], and finally what we will focus on, graph neural network (GNN) based models.

5) Evaluation: Evaluation is step that we understand how the our model performs under the given text classification task. As it is provided in [2], [3], there are several evaluation metrics that can be used to evaluate the performance of a supervised technique. The most common metrics are accuracy, F_β-score, micro/macro-averaging. Although we also have metrics like Matthews correlation coefficient and receiver operating characteristics (ROC). In order to evaluate the performance of our model, based on the provided techniques, we need to use labeled data, i.e., we need benchmark datasets like GLUE [16], TweetEval [17], among others.

B. Graph Neural Networks

In recent years, deep learning based solutions surpassed any approach on machine learning tasks such as image classification, video processing, speech recognition and natural language processing. In these tasks, the underlying data are usually represented in the Euclidean domain. However, each day the amount of non-Euclidean data increases, which are represented as represented by graphs to capture the underlying the complex relationships and interdependency between objects. Therefore, a need for deep learning methods that can manage graph structured data has emerged. In this context, the graph neural networks (GNNs) are born, and many of the deep learning approaches are converted to graph domain such as recurrent GNNs, convolutional GNNs (ConvGNNs) or graph convolutional networks (GCNs), graph autoencoder (GAE), graph reinforcement learning (GRL), graph adversarial methods and spatial-temporal GNNs, as summarized in [4]–[7].

1) Reasons to use GNNs: Hidden patterns residing under Euclidean data can be effectively obtained by traditional deep learning techniques. However, the increasing number of applications based on a non-Euclidean data structure enforces the necessity of graph based solutions. In this aspect, the following examples can be used to illustrate the benefits of having a graph based deep learning framework [5]:

- In e-commerce, highly accurate recommendation system can be achieved by using graph based deep learning techniques, since the interactions between users and products are a textbook example of graph structured data.
- For drug discovery in chemistry, we need to obtain the bioactivity of the molecules, where the molecules are modeled as graphs.
- Categorization of articles in a citation network, where the articles are linked to each other via "citationships", i.e., forming a graph structure.

2) Challenges to use GNNs: In order to have a graph domain deep learning framework, we need to overcome several challenges imposed by the complexity of the graph data. Due to the nature of graphs, when they are compared with Euclidean data, they can be irregular, they can have unordered nodes with different number of neighbors. Hence, many basic operations defined in Euclidean domain can be challenging to apply to the graph domain, e.g., convolution operation. In addition, one of the fundamental assumption we have in the existing machine learning algorithms is that the instances are independent of each other, although this assumption is not valid for graph data since each instance (node) is related to others by links of various types [5]. Some of the main challenges can be categorized as follows [6]:

a) Irregular structures of graphs: We have the *geometric deep learning problem* which is the inability to define basic operations like convolution and pooling in the graph domain, which are essential aspects of traditional CNNs.

b) Heterogeneity and diversity of graphs: We have many different properties that a single graph can have: graphs can be homogenous or heterogeneous, they can be weighted or unweighted, they can be directed or undirected, and they can be signed or unsigned. Furthermore, the tasks may consist of node-level problems such as node classification, link prediction or they can consist of graph-level problems such as graph classification or graph generation. Therefore, we need a spectrum of architectures to tackle all these problems one-by-one.

c) Large-scale graphs: As in the case of e-commerce and social networks, graph structured data can have a large number of nodes and edges. However, we still need appropriate algorithms to work on the graph structure without increasing the computational and time complexity too much.

d) Incorporating interdisciplinary knowledge: Graph structured data sometimes traces back to other disciplines such as biology, chemistry and social sciences. The interdisciplinary nature helps to leverage domain knowledge to solve specific problems, but it can also complicate model designs. For the case of molecular graph generation, the chemical constraints and the generation's objective function are often non-differentiable. Hence, gradient based training methods are out of the picture.

3) Ways to use GNNs: In [4], a general design to pipeline of GNNs is proposed. The following steps are necessary to obtain a graph-based deep learning framework:

a) Finding a graph structure: Based on the application in hand, we need to find out the underlying graph structure. There are two possibilities. First one is that we have an explicit graph structure, in the application such as social network, physical system or knowledge graph. The other possibility is that the underlying graph is implicit, and we need to build the graph from the task, such as obtaining a fully-connected "word" graph for text or obtaining a scene graph for an image. Then, we can obtain an optimal GNN model for the the graph we obtained either from explicit information or from the task.

b) Design a loss function: Based on the task in hand and the training setting, a loss function needs to be determined, the loss function can be node-level, edge-level or graph-level, depending on the training setting of supervised, semi-supervised or unsupervised learning.

c) Build model using computational modules: Finally, we need computational modules to build and train our model. Based on the definition provided in [4], we need a module to conduct convolution and recurrent operations to propagate information between nodes to capture the underlying feature and topological information. We need a sampling module, and we need a pooling module. With the combination of these modules a typical architecture of GNN model can be built.

C. Text Classification with GNN

Some of the earliest success achieved on deep learning with graphs relied on finding proper ways to embed nodes into vectors using an encoder function [18]. One question arises from that definition is the *"What is a good representation?"*. We want these nodes embeddings to preserve interesting structures of the graphs. There are unsupervised graph representations learning algorithms like *node2vec* [19], *DeepWalk* [20] and LINE [21] which are trained prior to graph neural networks. These algorithms aimed to learn representative embeddings for nodes to preserve interesting structures of the graphs

Aforementioned algorithms inherently capture local similarities. Further studies find that Convolutional Graph Neural Networks (ConvGNNs) summarizes local patches of the graphs and shows that neighboring nodes tends to highly overlap [18]. Therefore, a ConvGNNs enforce similar features for neighboring nodes by its nature without needing pre-training for node embeddings. This phenomenon was also mentioned in Text Graph Convolutional Network (Text GCN) [22]. Results of this paper on multiple benchmark data sets demonstrate that a vanilla Text GCN without any external word embeddings or knowledge outperforms state-of-the-art methods for text classification. On the other hand, Text GCN also learns predictive word and document embeddings jointly.

In [22] they evaluate Text GCN on two experimental tasks. First they seek the answer of whether their model achieve satisfactory results in text classification, even with limited labeled data and then they test whether their model learn predictive word and document embeddings. They compare their method with several state-of-art models. The suggested Text GCN may produce high text classification results and train predictive document and word embeddings, according to the experimental results. However, a major limitation of this study is that the GCN model is inherently transductive [22], in which test document nodes (without labels) are included in GCN training. Thus Text GCN could not quickly generate embeddings and make prediction for unseen test documents.

To improve the weakness of Text GCN in text classification task, [23] and [24] was mentioned in future work section of [22]. In [23] a novel algorithm Graph Attention Networks (GATs) was proposed. It was mentioned in [23] that GATs are new convolutional-style neural networks that operate on graph-structured data and use masked self-attentional layers. The graph attentional layer used in these networks is computationally efficient. Attentional layers do not require expensive matrix operations and they are parallelizable across all nodes in the graph. This structure allows for (implicitly) assigning different importance to different nodes within a neighborhood while dealing with different sized neighborhoods, and does not require knowing the entire graph structure. The experimental results yields that their attention-based models outperformed or matched state-of-the-art performance in four well-known node classification benchmarks, both transductive and inductive tasks [23].

Fast Graph Convolutional Neural Network [24] was also mentioned in the future work section of [22]. In [24] it was mentioned that, GCN in [25] represented as a useful graph model for semi-supervised learning. This model was created with the intention of being taught with both training and test data. Furthermore, for training with large, dense graphs, the recursive neighborhood expansion across layers faces time and memory issues. [24] interpret graph convolutions as integral transforms of embedding functions under probability measures to relax the condition of simultaneous availability of test data. As a result of this interpretation, Monte Carlo techniques may be used to consistently estimate the integrals, leading to a batched training scheme like FastGCN, which is proposed [24].

After further development on top of Text GCN, Simplifying Graph Convolutional Networks [26] was proposed to overcome unnecessary complexity and redundant computation in the previous work of Fast GCN and GATs. GCNs and their variants have received a lot of attention and have become the defacto methods for learning graph representations. GCNs are primarily inspired by modern deep learning methodologies, and as a result, they may inherit extra complexity and redundant processing. In [26] they eliminate the unnecessary complexity in this paper by reducing non-linearities one by one and collapsing weight matrices between layers. The resulting linear model is theoretically analyzed and shown to correspond to a fixed low-pass filter followed by a linear classifier in. The test results in [26] shows that these simplifications have no negative influence on accuracy in a wide range of downstream applications. Furthermore, the resulting model scales to bigger datasets, is intuitively interpretable, and outperforms FastGCN by up to two orders of magnitude.

D. BERT-GNN Architectures

After coming up with the idea of combining BERT models with GNN based models to increase text classification performance, we have encountered a few similar approaches that tries to accomplish a similar task. The first significant approach in combining BERT and GNN architectures is the *VGCN-BERT* proposed in [9]. However, in this approach the authors generate a vocabulary graph to produce word embeddings using the GCN architecture, then they supply these embeddings as input to the BERT architecture. This approach is different than what we are trying to achieve, since we are proposing aggregation of these embeddings since they both embody different aspects of the information.

In addition to this approach, in [10], *BEGNN* model is proposed, which aggregates graph embeddings with BERT embeddings similar to out approach, although they are generating graph embeddings for each document separately, so graph embeddings that are used in BEGNN are limited to the global information of that specific document. This approach can be limited, since with graph embeddings we try to convey the global and structural information of the texts, and we propose an extension to this manner by generating graph embeddings using whole training set that can embody structural and global information in a more generic sense.

After our project proposal is finalized, there has been several publications regarding the text classification with combination of GNN and BERT architectures [11]–[13]. In these publications, there are some similar approaches to our project, although we still have some differences in our methods, when it is compared to these fresh publications.

III. Methods

A. General Structure

We propose a model to obtain structural and semantic embeddings for each of the documents and then use this information to make classification. Graph Neural Network is used to obtain structural embedding and Distil-BERT is used for retrieving semantic embeddings from text. Then these two embeddings combined with 3 different approaches that mentioned in Section III-G. Finally, the obtained document embedding passed to classification layer to get predictions for each document. The Fig. 1 is the summary of general architecture.

B. Dataset Description

The dataset of this term project is *20 News Group* (20NG) [27] from huggingface.co. Dataset contains 18,846 documents evenly categorized into 20 different categories. In total, 11,314 documents are in the training set and 7,532 documents are in the test set.

C. Graph Generation from Data

To represent documents and words in a graph structure, we identify the vocabulary for the whole dataset without explicitly dividing it into test and training. For GCN structure, PyG asks user to pass whole dataset at once and then later identify the train and test indices. Therefore, we treat whole dataset as single corpus and obtained vocabulary out of it. Later, we put each document and words in the nodes and represent their connections with adjacency matrix $\boldsymbol{A}$. Also each node can have an embedding in arbitrary dimension. To not effect learning of the structural features, we did not initialize these node embeddings by using popular algorithms like [28], [29], since they enforce node to start with a semantic information. Therefore, we used one-hot vector representation for both document and word embeddings. It yields the matrix $\boldsymbol{X} \in \mathbb{R}^{(n \times n)}$, where $n = n_{doc} + |V|$. We define the entries of the adjacency matrix as follows:

$$A_{ij} = \begin{cases} \mathrm{PMI}(i,j) & i,j \text{ are words, } \mathrm{PMI}(i,j) > 0 \\ \text{TF-IDF}_{ij} & i \text{ is document, } j \text{ is word} \\ 1 & i = j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

D. Learnable Adjacency Matrix

We also propose learnable adjacency matrix to have further find interesting structures in data. We define new adjacency matrix and tune the learnable part of it by α. It can be given as follows:

$$\widetilde{\boldsymbol{A}} = \alpha \widehat{\boldsymbol{A}} + (1-\alpha)\boldsymbol{A} \tag{2}$$

where we took gradient of loss function only with respect to $\widehat{\boldsymbol{A}}$ and $\alpha \in (0,1)$.

E. GNN Embeddings

To implement Graph Neural Network, PyTorch Geometric library have been utilized. The aforementioned learnable adjacency matrix $\widetilde{\boldsymbol{A}}$ have been utilized instead of default $\boldsymbol{A}$. Generic equations of the GNN structure in our model can be given as follows:

$$\boldsymbol{L}^{(j+1)} = \rho(\widetilde{\boldsymbol{A}}\boldsymbol{L}^{(j)}\boldsymbol{W}_j), \tag{3}$$

Our model has 2 layers:

$$\boldsymbol{Z} = \mathrm{softmax}\left(\widetilde{\boldsymbol{A}}\ \mathrm{ReLU}(\widetilde{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}_0)\,\boldsymbol{W}_1\right) \tag{4}$$

and the cross-entropy error over all labeled documents:

$$\mathcal{L} = -\sum_{d \in \mathcal{Y}_D} \sum_{f=1}^{F} Y_{df} \ln Z_{df} \tag{5}$$

Where $\mathcal{Y}_D$ is the set of documents indices that have labels and F is the dimension of the output features.

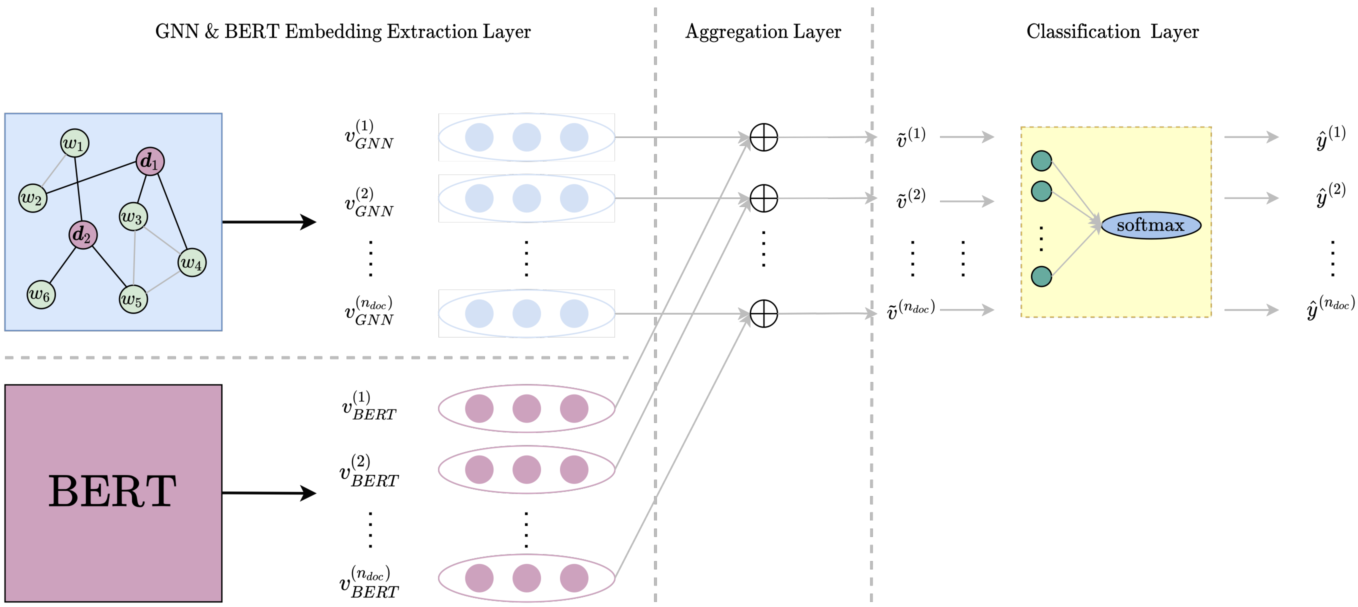

Fig. 1: Model Architecture

Finally, we can obtain embeddings from the layer equations as follows:

$$\boldsymbol{E}_1 = \widetilde{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}_0 \tag{6}$$

$$\boldsymbol{E}_2 = \widetilde{\boldsymbol{A}}\ \mathrm{ReLU}(\widetilde{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}_0)\,\boldsymbol{W}_1 \tag{7}$$

Where $\boldsymbol{W}_0 \in \mathbb{R}^{(n\times 200)}$ and $\boldsymbol{E}_1 \in \mathbb{R}^{(n\times 200)}$ which gives 200 dimensional representation for both documents and words. We used those 200 dimensional embedding vector and represented as $\boldsymbol{v}_{\mathrm{GNN}}$.

F. Distil-BERT Embeddings

To obtain embeddings from Distil-BERT, we fine-tuned the pre-trained model on 20NG dataset. Default parameters have 13 stacked attention layers so it has a output of dimension $(13, 768)$ for each document. There is no pre-defined way to obtain a single 768 dimensional vector from that stacked representation. We choose the 13rd head of the attention layer as document representation. It is also possible to choose different head of the attention or one can also take element wise max of each layer etc. The obtained 768 dimensional document embeddings will be represented as $\boldsymbol{v}_{\mathrm{BERT}}$.

G. Aggregation of Embeddings

There are several possible ways to aggregate the embeddings retrieved from GNN and Distil-BERT. We will show the structure of each of them. Each of the document embedding vector will be represented as $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$. The dimension of the $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$ will be generically represented as d_{doc}

1) Concatenation: We simply concatenate $\boldsymbol{v}_{\mathrm{BERT}}$ at the end of $\boldsymbol{v}_{\mathrm{GNN}}$ to obtain 968 dimensional document representation.

$$\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}} = [\boldsymbol{v}_{\mathrm{GNN}} || \boldsymbol{v}_{\mathrm{BERT}}], \widetilde{\boldsymbol{v}}_{\boldsymbol{doc}} \in \mathbb{R}^{(968\times 1)} \tag{8}$$

2) Element-wise Sum: To sum the embeddings, they must have same dimension. Therefore, we apply PCA to $\boldsymbol{v}_{\mathrm{BERT}}$ to reduce its dimension to 200. Then we simply sum them and obtain $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$ as follows:

$$\widetilde{\boldsymbol{v}}_{\mathrm{BERT}} = \mathrm{PCA}_{200}\{\boldsymbol{v}_{\mathrm{BERT}}\} \tag{9}$$

$$\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}} = \boldsymbol{v}_{\mathrm{GNN}} + \widetilde{\boldsymbol{v}}_{\mathrm{BERT}}, \quad \widetilde{\boldsymbol{v}}_{\boldsymbol{doc}} \in \mathbb{R}^{(400\times 1)} \tag{10}$$

3) Trade-Off: The trade-off version of the embedding approach is aimed to control the contribution of GNN and BERT embedding with trainable parameter λ. It can be done for both summation and concatenation strategies $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$ is given as follows:

$$\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}} = [\lambda \boldsymbol{v}_{\mathrm{GNN}} \,||\, (1-\lambda)\boldsymbol{v}_{\mathrm{BERT}}] \in \mathbb{R}^{(968\times 1)} \tag{11}$$

$$\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}} = \lambda \boldsymbol{v}_{\mathrm{GNN}} + (1-\lambda)\widetilde{\boldsymbol{v}}_{\mathrm{BERT}} \in \mathbb{R}^{(400\times 1)} \tag{12}$$

H. Classification Layer for Document Embeddings

As a final step, we passed $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$ to classification layer to obtain accuracy results. The classification layer equation can be easily given as follows:

$$\boldsymbol{z}_{doc} = \text{softmax}(\boldsymbol{W}\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}), \boldsymbol{W} \in \mathbb{R}^{n_{doc} \times d_{doc}} \quad (13)$$

IV. Results

The result section will have 4 different parts. In the first part we will investigate the performance of GNN structure by its own. The second part will include the performance of Distil-BERT only. Individual results explanations will help us to compare their performance when they are combined with aggregation layer. At the third part we will consider the classification results of obtained $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$ from the Section III-G section. We will investigate the each aggregation setting mentioned in Section III-G. As a last part, we will investigate the performance of our model when it is compared with the different SOTA algorithms on the 20NG dataset.

A. GNN Results

In this part, we only train the GNN part of the algorithm and get the prediction results as a by product of embedding extraction procedure. We will use the best results from Table I to compare with SOTA results. For the GNN results with learnable adjacency matrix, we only use the best $\alpha = 0.13$ result for convenience. For convenience, we used some naming in our models. These are as follows: $\text{GCN}_{i,j,k} \triangleq$ GCN with i, j, k hidden layers and $\text{L-GCN}_{i,j,k} \triangleq$ GCN with i, j, k hidden layers using learnable adjacency matrix structure.

TABLE I: GNN Results

Models	Train Accuracy (%)	Test Accuracy (%)
$\text{GCN}_{200,20}$	100	66.50
$\text{L-GCN}_{200,20}$	100	67.50
$\text{GCN}_{2000,200,20}$	100	60.80
$\text{L-GCN}_{2000,200,20}$	100	60.70

B. BERT Results

To train Distil-BERT on our dataset, we used the transformers library. It allows us to further fine-tune our model on pre-trained Distil-BERT. Prediction results of the Distil-BERT model are in Table II.

TABLE II: Distil-BERT Training Results

Epoch	Training Loss	Validation Loss	Accuracy (%)
1	1.1764	1.217	65.64
2	0.7607	1.174	68.12
3	0.5118	1.440	67.90
⋮	⋮	⋮	⋮
28	0.0684	3.372	69.98
29	0.0817	3.397	70.05
30	0.0799	3.404	69.99

C. Combined $\widetilde{\boldsymbol{v}}_{\boldsymbol{doc}}$ Results

The given combined results are find by using the method in Section III-G1. This method gave the best results for all models. Therefore, only this aggregation approach results are mentioned.

TABLE III: GNN+BERT Combined Results

Models		20NG Test Accuracy (%)
$\text{GCN}_{200,20}$	+ Distil-BERT	67.20
$\text{L-GCN}_{200,20}$	+ Distil-BERT	69.70
$\text{GCN}_{2000,200,20}$	+ Distil-BERT	64.90
$\text{L-GCN}_{2000,200,20}$	+ Distil-BERT	63.20

D. Comparison with SOTA Algorithms

The results of the SOTA algorithms for the 20NG dataset along with our results are provided in Table IV. The models are sorted according to their accuracies, and our results are provided in color.

TABLE IV: SOTA Results

Models	20NG Test Accuracy (%)
PV-DM	51.10
LSTM	65.70
$\text{L-GCN}_{200,20}$	67.60
$\text{L-GCN}_{200,20}$ +Distil-BERT	69.70
Our Distil-BERT	70.05
RoBERTa	83.80
BERT	85.30
TextGCN	86.30
RoBERTaGAT	86.50
BertGAT	87.40
SGC	88.50
BertGCN	89.30
RoBERTaGCN	**89.50**

V. Discussion & Conclusion

A. Obtained Results

Unfortunately, obtained results did not surpass the SOTA algorithms that mentioned in Section IV-D. However, we are aware of the problems that occurred during our implementation. These issues are mentioned in the Section V-B. Our model still can pass some of the other algorithms.

B. Issues

During our implementation of the algorithm, we have discovered several problems and solved most of them. The very first problem that we faced with was the instability of the 20NG dataset in different websites. This dataset is used in both supervised and semi-supervised manner in the literature. The graph convolutional network proposed in [25] is for semi-supervised learning problem. [22] also proposed for only semi-supervised learning problem. [22] treats 20NG dataset as a semi-supervised learning problem. In that version of 20NG, they do not use labels of every data instance. However, in huggingface.co, all of the data are labeled. In our structure, we tried to treat problem in inductive manner since we know all of the labels. However, we used the GCN architecture of the PyTorch Geometric which only accepts the semi-supervised learning tasks and it asks to pre-define the labeled data to do back propagation only on these labeled data. Since we know all of the labels, we pre-define the labeled data as whole training set.

C. Future Directions

To solve the mentioned problems in Section V-B, we can treat whole problem in inductive manner and use more appropriate GNN structures. One of the candidate for inductive learning problem is [24]. It solves the problem totally in inductive manner which also makes computation faster. Another candidate architecture for the GNN side is [30]. It also enables us to learn more interesting features since it uses attention mechanism rather simple matrix multiplications in Eq. (3).

References

[1] Kowsari, J. Meimandi, Heidarysafa, Mendu, Barnes, and Brown, "Text classification algorithms: A survey," *Information*, vol. 10, no. 4, p. 150, Apr. 2019, ISSN: 2078-2489. DOI: 10.3390/info10040150. [Online]. Available: http://dx.doi.org/10.3390/info10040150.

[2] Q. Li, H. Peng, J. Li, *et al.*, "A survey on text classification: From shallow to deep learning," *CoRR*, vol. abs/2008.00364, 2020. arXiv: 2008.00364. [Online]. Available: https://arxiv.org/abs/2008.00364.

[3] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning based text classification: A comprehensive review," *CoRR*, vol. abs/2004.03705, 2020. arXiv: 2004.03705. [Online]. Available: https://arxiv.org/abs/2004.03705.

[4] J. Zhou, G. Cui, S. Hu, *et al.*, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020, ISSN: 2666-6510. DOI: https://doi.org/10.1016/j.aiopen.2021.01.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666651021000012.

[5] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, Jan. 2021, ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.2978386. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[6] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *CoRR*, vol. abs/1812.04202, 2018. arXiv: 1812.04202. [Online]. Available: http://arxiv.org/abs/1812.04202.

[7] L. Sun, J. Wang, P. S. Yu, and B. Li, "Adversarial attack and defense on graph data: A survey," *CoRR*, vol. abs/1812.10528, 2018. arXiv: 1812.10528. [Online]. Available: http://arxiv.org/abs/1812.10528.

[8] D. Cummings and M. Nassar, "Structured citation trend prediction using graph neural networks," *CoRR*, vol. abs/2104.02562, 2021. arXiv: 2104.02562. [Online]. Available: https://arxiv.org/abs/2104.02562.

[9] Z. Lu, P. Du, and J. Nie, "VGCN-BERT: augmenting BERT with graph embedding for text classification," *CoRR*, vol. abs/2004.05707, 2020. arXiv: 2004.05707. [Online]. Available: https://arxiv.org/abs/2004.05707.

[10] Y. Yang and X. Cui, "Bert-enhanced text graph neural network for classification," *Entropy*, vol. 23, 2021.

[11] Y. Lin, Y. Meng, X. Sun, *et al.*, *Bertgcn: Transductive text classification by combining gcn and bert*, Mar. 2022. DOI: 10.48550/ARXIV.2105.05727. [Online]. Available: https://arxiv.org/abs/2105.05727.

[12] X. She, J. Chen, and G. Chen, "Joint learning with bert-gcn and multi-attention for event text classification and event assignment," *IEEE Access*, vol. 10, pp. 27 031–27 040, 2022. DOI: 10.1109/ACCESS.2022.3156918.

[13] F. Zeng, N. Chen, D. Yang, and Z. Meng, "Simplified-boosting ensemble convolutional network for text classification," *Neural Processing Letters*, May 2022. DOI: 10.1007/s11063-022-10843-4. [Online]. Available: https://doi.org/10.1007/s11063-022-10843-4.

[14] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805.

[15] J. Howard and S. Ruder, "Fine-tuned language models for text classification," *CoRR*, vol. abs/1801.06146, 2018. arXiv: 1801.06146. [Online]. Available: http://arxiv.org/abs/1801.06146.

[16] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," arXiv preprint 1804.07461, 2018. [Online]. Available: https://gluebenchmark.com/.

[17] F. Barbieri, J. Camacho-Collados, L. Espinosa-Anke, and L. Neves, "TweetEval: Unified Benchmark and Comparative Evaluation for Tweet Classification," in *Proceedings of Findings of EMNLP*, 2020.

[18] P. Veličković, *Theoretical foundations of graph neural networks*, 2021. [Online]. Available: https://petar-v.com/talks/GNN-Wednesday.pdf.

[19] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," *CoRR*, vol. abs/1607.00653, 2016. arXiv: 1607.00653. [Online]. Available: http://arxiv.org/abs/1607.00653.

[20] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk," *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug. 2014. DOI: 10.1145/2623330.2623732. [Online]. Available: http://dx.doi.org/10.1145/2623330.2623732.

[21] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line," *Proceedings of the 24th International Conference on World Wide Web*, May 2015. DOI: 10.1145/2736277.2741093. [Online]. Available: http://dx.doi.org/10.1145/2736277.2741093.

[22] L. Yao, C. Mao, and Y. Luo, *Graph convolutional networks for text classification*, 2018. arXiv: 1809.05679 [cs.CL].

[23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph attention networks*, 2018. arXiv: 1710.10903 [stat.ML].

[24] J. Chen, T. Ma, and C. Xiao, "Fastgcn: Fast learning with graph convolutional networks via importance sampling," *CoRR*, vol. abs/1801.10247, 2018. arXiv: 1801.10247. [Online]. Available: http://arxiv.org/abs/1801.10247.

[25] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, 2017. arXiv: 1609.02907 [cs.LG].

[26] F. Wu, T. Zhang, A. H. S. Jr., C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," *CoRR*, vol. abs/1902.07153, 2019. arXiv: 1902.07153. [Online]. Available: http://arxiv.org/abs/1902.07153.

[27] *20 News Group DataSet*, https://huggingface.co/datasets/SetFit/20_newsgroups, Accessed: 2022-5-24.

[28] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. [Online]. Available: https://aclanthology.org/D14-1162.

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, 2013. DOI: 10.48550/ARXIV.1301.3781. [Online]. Available: https://arxiv.org/abs/1301.3781.

[30] Z. Guo, Y. Zhang, and W. Lu, "Attention guided graph convolutional networks for relation extraction," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 241–251. DOI: 10.18653/v1/P19-1024. [Online]. Available: https://aclanthology.org/P19-1024.

Appendix A
Contribution of Members

A. *Tuna Alikaşifoğlu*

Tuna was responsible for several aspects of the project. First, he was responsible for data preprocessing with (Distil-) BERT tokenization techniques to convert the documents to list of input IDs, which corresponds to the text representations that are both used in generation of BERT and GNN embeddings. He was also responsible for generation of (Distil-) BERT embeddings to be used in the aggregation step. Finally, he implemented an efficient way to generate graph adjacency matrix $\boldsymbol{A}$ with sparse representation.

B. *Arda Can Aras*

Arda was responsible for researching and understanding the novel GNN algorithms and their implementations in text classification. He has implemented Graph Neural Network architecture, with different setups and proposed a learnable adjacency matrix $\boldsymbol{A}$ as a novelty. He also implemented the three different aggregation layer strategies and classification layer. Arda was also responsible of fine-tuning the Distil-BERT on 20NG dataset.

Appendix B
Source Code

A. generic.py

```
import time
import pickle
from pathlib import Path


def get_time(format: str = "%Y_%m_%d_%H_%M"):
    return time.strftime(format)


def pickle_dump(obj: object, path: Path):
    with open(path, "wb") as f:
        pickle.dump(obj, f)


def pickle_load(path: Path) -> object:
    with open(path, "rb") as f:
        return pickle.load(f)


def picklize(func, path: Path, *args, enforce: bool = False, **kwargs):
    if enforce or not path.exists():
        result = func(*args, **kwargs)
        pickle_dump(result, path)
    else:
        result = pickle_load(path)
    return result


def batch_iterable(iterable, batch_size=1):
    l = len(iterable)
    for i in range(0, l, batch_size):
        yield iterable[i : min(i + batch_size, l)]
```

B. adjacency.py

```
from typing import Tuple, Dict, List
import numpy as np
from numba import njit, vectorize
from numba import float64 as numba_float64
from numba import int64 as numba_int64
from numba.core import types as numba_types
from numba.typed import Dict as NumbaDict
from itertools import chain, combinations
from tqdm import tqdm
from pathlib import Path
import scipy.sparse as sps

from eee586 import PKL_DIR
from eee586.word_embedding import get_token_encodings
from eee586.utils.generic import picklize

WORDS_OCC_KEY_TYPE = numba_types.int64
WORD_PAIRS_OCC_KEY_TYPE = numba_types.Tuple((numba_types.int64, numba_types.int64))
WORD_OCC_VAL_TYPE = numba_types.int64


@vectorize([numba_float64(numba_int64, numba_int64)])
def _idf(df: numba_int64, N: numba_int64) -> numba_float64:
    return np.log(N / (1 + df))


@njit()
def count_nonzero(vector: np.ndarray) -> int:
    count = 0
    for value in vector:
        if value != 0:
            count += 1
    return count

```

```python

@njit()
def get_tfidf_coo_vectors(
    document_count: int,
    all_vocab: np.ndarray,
    tf_dict: NumbaDict,
    df_dict: NumbaDict,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:

    data_size = len(tf_dict)
    row = np.zeros((data_size,))
    col = np.zeros((data_size,))
    data = np.zeros((data_size,))

    word_idx = {word: idx for idx, word in enumerate(all_vocab)}
    for i, ((doc_id, word), tf) in enumerate(tf_dict.items()):
        df = df_dict[word]
        idf = _idf(df, document_count)
        curr_word_idx = word_idx[word]

        row[i] = doc_id
        col[i] = curr_word_idx
        data[i] = tf * idf
    return row, col, data


def get_tfidf_matrix(
    dataset_dir: Path,
    documents: List[List[int]],
    all_vocab: np.ndarray,
    enforce_recompute: bool = False,
):
    docs = [np.array(doc) for doc in documents]
    tfidf_matrix_path = Path(dataset_dir, "tfidf_matrix.pkl")
    tfidf_matrix = picklize(
        _get_tfidf_matrix,
        tfidf_matrix_path,
        docs,
        all_vocab,
        enforce=enforce_recompute,
    )
    return tfidf_matrix


def _get_tfidf_matrix(
    documents: List[np.ndarray],
    all_vocab: np.ndarray,
) -> np.ndarray:
    tf_dict = NumbaDict.empty(
        key_type=WORD_PAIRS_OCC_KEY_TYPE,
        value_type=numba_types.float64,
    )

    df_dict = NumbaDict.empty(
        key_type=WORDS_OCC_KEY_TYPE,
        value_type=WORD_OCC_VAL_TYPE,
    )

    for word in tqdm(all_vocab, desc="TF-IDF", unit="word"):
        for doc_id, doc in enumerate(documents):
            tf = count_nonzero(doc == word) / len(doc)
            if tf > 0:
                tf_dict[(doc_id, word)] = tf
                df_dict[word] = df_dict.get(word, 0) + 1

    N = len(documents)
    row, col, data = get_tfidf_coo_vectors(N, all_vocab, tf_dict, df_dict)
    return sps.coo_matrix((data, (row, col)), shape=(N, len(all_vocab)))


def _convert_dict_to_numba_dict(d, key_type, value_type) -> NumbaDict:
    nd = NumbaDict.empty(
        key_type=key_type,
```

```python
        value_type=value_type,
    )
    for k, v in d.items():
        nd[k] = v
    return nd


def _get_windows(
    documents: List[List[int]],
    window_size: int = 20,
    stride: int = 1,
):
    corpus = np.array(list(chain(*documents)))
    N = len(corpus)
    try:
        assert stride < window_size
        assert np.mod(N - window_size, stride) == 0
    except AssertionError as e:
        print("Window size and stride is not compatible with the corpus size")
        raise e

    doc_vocabs = [set(doc) for doc in documents]
    all_vocab = np.array(list(set.union(*doc_vocabs)))
    all_vocab = np.sort(all_vocab)

    windows = (
        np.array(
            [corpus[i : i + window_size] for i in range(0, N - window_size + 1, stride)]
        )
        if window_size < N
        else np.array([corpus])
    )
    return windows, corpus, all_vocab


def get_words_occurrence(
    dataset_dir: Path,
    documents: List[List[int]],
    enforce_recompute: bool = False,
    window_size: int = 20,
    stride: int = 1,
) -> Dict[int, int]:
    windows_path = Path(dataset_dir, "windows.pkl")
    windows, *_ = picklize(
        _get_windows,
        windows_path,
        documents,
        window_size=window_size,
        stride=stride,
        enforce=enforce_recompute,
    )

    word_occurrence_path = Path(dataset_dir, "words_occurrence.pkl")
    words_occurrence = picklize(
        _get_words_occurrence,
        word_occurrence_path,
        windows,
        enforce=enforce_recompute,
    )

    return words_occurrence


def _get_words_occurrence(windows) -> Dict[int, int]:
    words_occurrence = {}
    for window in tqdm(windows, desc="Words Occurrence", unit="window"):
        window_words_unique = np.unique(window)
        for word in window_words_unique:
            words_occurrence[word] = words_occurrence.get(word, 0) + 1
    return words_occurrence


def get_word_pairs_occurrence(
```

```python
    dataset_dir: Path,
    documents: List[List[int]],
    enforce_recompute: bool = False,
    window_size: int = 3,
    stride: int = 1,
) -> Dict[Tuple[int, int], int]:
    windows_path = Path(dataset_dir, "windows.pkl")
    windows, *_ = picklize(
        _get_windows,
        windows_path,
        documents,
        window_size=window_size,
        stride=stride,
        enforce=enforce_recompute,
    )

    word_pairs_occurrence_path = Path(dataset_dir, "word_pairs_occurrence.pkl")
    word_pairs_occurrence = picklize(
        _get_word_pairs_occurrence,
        word_pairs_occurrence_path,
        windows,
        enforce=enforce_recompute,
    )

    return word_pairs_occurrence


def _get_word_pairs_occurrence(windows) -> Dict[Tuple[int, int], int]:
    word_pairs_occurrence = {}
    for window in tqdm(windows, desc="Words Occurrence", unit="window"):
        window_words_unique = np.unique(window)
        window_word_pairs = combinations(window_words_unique, 2)
        for pair in window_word_pairs:
            word_pairs_occurrence[pair] = word_pairs_occurrence.get(pair, 0) + 1
    return word_pairs_occurrence


@njit()
def pmi(n_i: int, n_j: int, n_ij: int, n_win: int, relu: bool = True) -> float:
    if n_i <= 0 or n_j <= 0 or n_ij <= 0 or n_win <= 0:
        pmi = 0
    else:
        pmi = np.log(n_ij * n_win / (n_i * n_j))
    result = np.maximum(pmi, 0) if relu else pmi
    return result


def get_pmi_matrix(
    dataset_dir: Path,
    documents: List[List[int]],
    enforce_recompute: bool = False,
    window_size: int = 20,
    stride: int = 1,
) -> np.ndarray:
    windows_path = Path(dataset_dir, "windows.pkl")
    windows, _, all_vocab = picklize(
        _get_windows,
        windows_path,
        documents,
        window_size=window_size,
        stride=stride,
        enforce=enforce_recompute,
    )

    words_occurence_path = Path(dataset_dir, "words_occurrence.pkl")
    words_occurrence = picklize(
        _get_words_occurrence,
        words_occurence_path,
        windows,
        enforce=enforce_recompute,
    )
    word_pairs_occurrence_path = Path(dataset_dir, "word_pairs_occurrence.pkl")
    word_pairs_occurrence = picklize(
```

```python
        _get_word_pairs_occurrence,
        word_pairs_occurrence_path,
        windows,
        enforce=enforce_recompute,
    )
    print("Converting Numba Dict")
    words_occurrence = _convert_dict_to_numba_dict(
        words_occurrence,
        key_type=WORDS_OCC_KEY_TYPE,
        value_type=WORD_OCC_VAL_TYPE,
    )
    word_pairs_occurrence = _convert_dict_to_numba_dict(
        word_pairs_occurrence,
        key_type=WORD_PAIRS_OCC_KEY_TYPE,
        value_type=WORD_OCC_VAL_TYPE,
    )

    print("Computing PMI")
    pmi_matrix_path = Path(dataset_dir, "pmi_matrix.pkl")
    pmi_matrix = picklize(
        _get_pmi_matrix,
        pmi_matrix_path,
        all_vocab,
        words_occurrence,
        word_pairs_occurrence,
        window_size=window_size,
        enforce=enforce_recompute,
    )

    return pmi_matrix


@njit()
def _get_pmi_coo_vectors(
    all_vocab: np.ndarray,
    words_occurrence: Dict[int, int],
    word_pairs_occurrence: Dict[Tuple[int, int], int],
    window_size: int,
) -> np.ndarray:
    n_vocab = len(all_vocab)
    pmi_dict = {}
    for i in range(n_vocab):
        for j in range(i + 1, n_vocab):
            word1 = all_vocab[i]
            word2 = all_vocab[j]

            n_i = words_occurrence.get(word1, 0)
            n_j = words_occurrence.get(word2, 0)
            n_ij = word_pairs_occurrence.get((word1, word2), 0)

            pmi_score = pmi(
                n_i=n_i,
                n_j=n_j,
                n_ij=n_ij,
                n_win=window_size,
                relu=True,
            )
            if pmi_score > 0:
                pmi_dict[(i, j)] = pmi_score

    data_size = len(pmi_dict)
    row = np.zeros((data_size,))
    col = np.zeros((data_size,))
    data = np.zeros((data_size,))

    for k, ((i, j), pmi_score) in enumerate(pmi_dict.items()):
        row[k] = i
        col[k] = j
        data[k] = pmi_score
    return row, col, data


def _get_pmi_matrix(
```

```python
    all_vocab: np.ndarray,
    words_occurrence: Dict[int, int],
    word_pairs_occurrence: Dict[Tuple[int, int], int],
    window_size: int,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    row, col, data = _get_pmi_coo_vectors(
        all_vocab,
        words_occurrence,
        word_pairs_occurrence,
        window_size,
    )
    n_vocab = len(all_vocab)
    pmi_matrix_upper = sps.coo_matrix((data, (row, col)), shape=(n_vocab, n_vocab))
    pmi_matrix = pmi_matrix_upper + pmi_matrix_upper.T + sps.identity(n_vocab)
    return pmi_matrix


def generate_adj_matrix(
    documents: List[List[int]],
    enforce_recompute: bool = False,
    dataset_name: str = "SetFit/20_newsgroups",
    window_size: int = 20,
    stride: int = 1,
) -> np.ndarray:
    doc_vocabs = [set(doc) for doc in documents]
    all_vocab = np.array(list(set.union(*doc_vocabs)))

    n_docs = len(documents)
    n_vocab = len(all_vocab)
    print(
        "n_docs: "
        + str(n_docs)
        + " n_vocab: "
        + str(n_vocab)
        + " n nodes : "
        + str(n_vocab + n_docs)
    )

    dataset_dir = Path.joinpath(
        PKL_DIR,
        f"{dataset_name.replace('/', '_')}",
        f"win{window_size}_s{stride}",
    )
    Path.mkdir(dataset_dir, parents=True, exist_ok=True)
    adj_matrix_path = Path(dataset_dir, "adj_matrix.pkl")
    adj_matrix = picklize(
        _generate_adj_matrix,
        adj_matrix_path,
        dataset_dir,
        documents,
        all_vocab,
        window_size=window_size,
        stride=stride,
        enforce=enforce_recompute,
    )
    return adj_matrix


def _generate_adj_matrix(
    dataset_dir: Path,
    documents: List[List[int]],
    all_vocab: np.ndarray,
    enforce_recompute: bool = False,
    window_size: int = 20,
    stride: int = 1,
) -> np.ndarray:
    tf_idf_matrix = get_tfidf_matrix(
        dataset_dir,
        documents,
        all_vocab,
        enforce_recompute,
    )
    pmi_matrix = get_pmi_matrix(
```

```
        dataset_dir,
        documents,
        enforce_recompute,
        window_size,
        stride,
    )

    upper_left = sps.identity(len(documents))
    upper_right = tf_idf_matrix
    lower_left = tf_idf_matrix.T
    lower_right = pmi_matrix
    adj_matrix = sps.bmat(
        [
            [upper_left, upper_right],
            [lower_left, lower_right],
        ],
    )
    return adj_matrix


def main(
    dataset_name: str = "SetFit/20_newsgroups",
    window_size: int = 20,
    stride: int = 1,
    pkl_dir: Path = PKL_DIR,
):
    dataset_dir = Path.joinpath(
        pkl_dir,
        f"{dataset_name.replace('/', '_')}",
        f"win{window_size}_s{stride}",
    )
    Path.mkdir(dataset_dir, parents=True, exist_ok=True)
    train_token_enc = get_token_encodings("train")
    documents = train_token_enc.get("input_ids")
    documents = documents[:2000]

    A = generate_adj_matrix(
        documents=documents,
        dataset_name=dataset_name,
        window_size=window_size,
        stride=stride,
    )

    print(type(A))
    print(f"Shape, Size: {A.shape}, {A.size}")


if __name__ == "__main__":
    main()
```

C. __main__.py

```
import click
import torch
from eee586 import word_embedding

from eee586.pretrain import pretrain_bert_model
from eee586.word_embedding import get_token_encodings


@click.group()
def cli():
    pass


@cli.command()
@click.option(
    "--dataset-sample-size",
    default=1000,
    help="Number of samples to use.",
    type=click.INT,
)
def pretrain(dataset_sample_size):
    pretrain_bert_model(
```

```python
        dataset_sample_size=dataset_sample_size,
    )


@cli.command()
def embed():
    train_toke_enc_dict = get_token_encodings("train")
    print(train_toke_enc_dict["input_ids"][0])
    print(train_toke_enc_dict["labels"][0])


@cli.command()
def check_torch_cuda():
    print(torch.cuda.is_available())


# cli()
from eee586.word_embedding import get_doc_embeddings

embed_train = get_doc_embeddings("train")
embed_test = get_doc_embeddings("test")
print(embed_train.shape)
print(embed_test.shape)
print(embed_train[-1])

# get_doc_embeddings("train")
# get_doc_embeddings("test")
```

D. *word_embedding.py*

```python
from transformers import BertTokenizer, BertModel
from pathlib import Path
from datasets import load_dataset
from tqdm import tqdm
from typing import List, Dict, Tuple
import nltk
from nltk.corpus import stopwords
from itertools import chain
import numpy as np
import torch
import torch.nn.functional as F

from eee586 import BERT_DEFAULT_MODEL_NAME, PKL_DIR, BERT_LAST_HIDDEN_OUT_SIZE
from eee586.utils.generic import picklize, batch_iterable


def _pad_batch_to_max(batch: List[torch.Tensor]) -> torch.Tensor:
    max_length = max([doc.numel() for doc in batch])
    batch = [
        F.pad(doc, pad=(0, max_length - doc.numel()), mode="constant", value=0)
        for doc in batch
    ]
    return torch.concat(batch, dim=0)


def _truncate_tensor_last_dim(tensor: torch.Tensor, max_length: int) -> torch.Tensor:
    curr_last_dim = tensor.shape[-1]
    return tensor if curr_last_dim <= max_length else tensor[..., :max_length]


def _get_doc_embeddings(
    sub_dataset_name="train",  # train/test
    *,
    remove_stop: bool = True,
    freq_limit: int = None,
    enforce_recompute=False,
    model_name=BERT_DEFAULT_MODEL_NAME,
    dataset_name="SetFit/20_newsgroups",
    batch_size=1,
    max_embed_length=2048,
) -> np.ndarray:
    if not sub_dataset_name in ["train", "test"]:
        raise ValueError("sub_dataset must be either 'train' or 'test'")

```

```
    sub_token_encodings = get_token_encodings(
        sub_dataset_name=sub_dataset_name,
        remove_stopword=remove_stop,
        freq_limit=freq_limit,
        enforce_recompute=enforce_recompute,
        model_name=model_name,
        dataset_name=dataset_name,
    )

    documents = sub_token_encodings["input_ids"]
    max_embed_length = min(max_embed_length, max([len(doc) for doc in documents]))
    all_vocab = [set(doc) for doc in documents]
    all_vocab = np.array(list(set.union(*all_vocab)))
    all_vocab = np.sort(all_vocab)

    documents = [
        _truncate_tensor_last_dim(torch.tensor([doc]), max_embed_length)
        for doc in documents
    ]
    all_vocab = torch.tensor([all_vocab])

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = BertModel.from_pretrained(
        model_name,
        max_position_embeddings=max_embed_length,
    )
    model.to(device)
    model.eval()

    batches = batch_iterable(documents, batch_size=batch_size)
    embeddings = np.zeros((len(documents), BERT_LAST_HIDDEN_OUT_SIZE))
    with torch.no_grad():
        total_count = len(documents) // batch_size
        for i, batch in enumerate(
            tqdm(
                batches,
                total=total_count,
                desc="Embedding",
                unit="batch",
            )
        ):
            batch = _pad_batch_to_max(batch)
            embedding = model(batch.to(device)).last_hidden_state
            embedding = torch.mean(embedding, dim=1)
            row_range = slice(i * batch_size, (i + 1) * batch_size)
            embeddings[row_range, :] = embedding.cpu().numpy()
    return embeddings


def get_doc_embeddings(
    sub_dataset_name="train",  # train/test
    *,
    remove_stop: bool = True,
    freq_limit: int = None,
    enforce_recompute=False,
    model_name=BERT_DEFAULT_MODEL_NAME,
    dataset_name="SetFit/20_newsgroups",
    batch_size=1,
    max_embed_length=2048,
) -> np.ndarray:
    sub_pkl_dir = Path.joinpath(
        PKL_DIR,
        dataset_name.replace("/", "_"),
        model_name,
        f"maxlen{max_embed_length}_batchsize{batch_size}",
    )
    if remove_stop:
        sub_pkl_dir = sub_pkl_dir.joinpath("wo_stop")
    if freq_limit is not None:
        sub_pkl_dir = sub_pkl_dir.joinpath(f"freq_limit_{freq_limit}")
    Path.mkdir(sub_pkl_dir, parents=True, exist_ok=True)
    sub_pkl_path = sub_pkl_dir.joinpath(f"{sub_dataset_name}_doc_embeddings.pkl")

```

```
    embeddings = picklize(
        _get_doc_embeddings,
        sub_pkl_path,
        sub_dataset_name=sub_dataset_name,
        remove_stop=remove_stop,
        freq_limit=freq_limit,
        enforce_recompute=enforce_recompute,
        model_name=model_name,
        dataset_name=dataset_name,
        batch_size=batch_size,
        max_embed_length=max_embed_length,
    )

    return embeddings


def _prune_words(
    tokenizer: BertTokenizer,
    input_ids: List[List[int]],
    labels: List[int],
    remove_stopword: bool = True,
    freq_limit: int = None,
) -> Tuple[List[List[int]], List[int]]:
    blacklist_words = []
    if remove_stopword:
        blacklist_words += _remove_stopwords(tokenizer)
    if freq_limit is not None:
        blacklist_words += _frequency_limit(input_ids, freq_limit)
    if len(blacklist_words) == 0:
        return input_ids, labels

    input_ids = [
        [j for j in ii if j not in blacklist_words]
        for ii in tqdm(input_ids, desc="Removing stop/rare words")
    ]
    empty_idx = [i for i, ii in enumerate(input_ids) if len(ii) == 0]
    input_ids = [ii for i, ii in enumerate(input_ids) if i not in empty_idx]
    labels = [labels[i] for i in range(len(labels)) if i not in empty_idx]
    return input_ids, labels


def _frequency_limit(input_ids: List[List[int]], limit: int):
    corpus = np.array(list(chain.from_iterable(input_ids)))
    unique, frequency = np.unique(corpus, return_counts=True)
    rare_words = unique[frequency < limit]
    return list(rare_words)


def _remove_stopwords(tokenizer: BertTokenizer) -> List[List[int]]:
    nltk.download("stopwords")
    stop_words = list(set(stopwords.words("english")))
    stop_tokenized_endcoded = tokenizer.batch_encode_plus(stop_words)
    stop_input_ids = stop_tokenized_endcoded["input_ids"]
    stop_input_ids = list(chain(*stop_input_ids))
    return stop_input_ids


def _get_input_ids_and_labels(
    orig_pkl_enc_path: Path,
    tokenizer,
    dataset,
    remove_stop: bool = True,
    freq_limit: int = None,
) -> Dict[str, List[int]]:
    texts_list = [sample["text"] for sample in dataset]
    tokenized_encoded = picklize(
        tokenizer.batch_encode_plus,
        orig_pkl_enc_path,
        tqdm(texts_list, desc="Tokenizing"),
    )
    input_ids = tokenized_encoded["input_ids"]
    labels = [sample["label"] for sample in dataset]
    input_ids, labels = _prune_words(
```

```python
        tokenizer,
        input_ids,
        labels,
        remove_stopword=remove_stop,
        freq_limit=freq_limit,
    )
    token_enc_dict = {
        "input_ids": input_ids,
        "labels": labels,
    }
    return token_enc_dict


def get_token_encodings(
    sub_dataset_name: str,  # train/test
    *,
    remove_stopword: bool = True,
    freq_limit: int = None,
    enforce_recompute: bool = False,
    model_name=BERT_DEFAULT_MODEL_NAME,
    dataset_name="SetFit/20_newsgroups",
):
    if not sub_dataset_name in ["train", "test"]:
        raise ValueError("sub_dataset must be either 'train' or 'test'")

    pkl_enc_path = Path.joinpath(
        PKL_DIR,
        f"{dataset_name.replace('/', '_')}",
        BERT_DEFAULT_MODEL_NAME,
    )

    pkl_enc_path_orig = pkl_enc_path.joinpath(f"{sub_dataset_name}_token.pkl")
    if remove_stopword:
        pkl_enc_path = pkl_enc_path.joinpath("wo_stop")
    if freq_limit is not None:
        pkl_enc_path = pkl_enc_path.joinpath(f"freq_limit_{freq_limit}")
    Path.mkdir(pkl_enc_path, exist_ok=True, parents=True)

    sub_pkl_enc_path = Path.joinpath(pkl_enc_path, f"{sub_dataset_name}_token_enc.pkl")
    if sub_pkl_enc_path.exists() and not enforce_recompute:
        sub_token_enc_dict = picklize(
            None,
            sub_pkl_enc_path,
        )
    else:
        dataset = load_dataset(dataset_name)
        tokenizer = BertTokenizer.from_pretrained(model_name)

        sub_dataset = dataset.get(f"{sub_dataset_name}")
        sub_token_enc_dict = picklize(
            _get_input_ids_and_labels,
            sub_pkl_enc_path,
            pkl_enc_path_orig,
            tokenizer,
            sub_dataset,
            remove_stop=remove_stopword,
            freq_limit=freq_limit,
            enforce=enforce_recompute,
        )
    return sub_token_enc_dict
```

E. *pretrain.py*

```python
from pathlib import Path
from typing import Union, Tuple
from datasets import load_dataset
from datasets.dataset_dict import DatasetDict
from transformers import (
    AutoTokenizer,
    DataCollatorWithPadding,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
)
```

```python

from eee586 import BERT_MODEL_DIR, BERT_DEFAULT_MODEL_NAME
from eee586.utils.generic import get_time


def get_num_label(dataset: DatasetDict) -> int:
    train_dataset = dataset.get("train")
    try:
        labels = [sample["label"] for sample in train_dataset]
    except KeyError as e:
        print(f"Dataset {dataset} does not have label.")
        raise e
    labels_set = set(labels)
    return len(labels_set)


def pretrain_bert_model(
    bert_model_name: str = BERT_DEFAULT_MODEL_NAME,
    dataset_name: Union[str, Tuple[str]] = "imdb",
    output_dir: str = None,
    dataset_sample_size: int = None,
):
    if type(dataset_name) is tuple:
        dataset = load_dataset(*dataset_name)
    else:
        dataset = load_dataset(dataset_name)
    num_labels = get_num_label(dataset)
    tokenizer = AutoTokenizer.from_pretrained(bert_model_name)

    tokenized_dataset = dataset.map(
        lambda x: tokenizer(x["text"], truncation=True), batched=True
    )
    if dataset_sample_size is not None:
        r = range(dataset_sample_size)
        small_train_dataset = tokenized_dataset["train"].shuffle().select(r)
        small_eval_dataset = tokenized_dataset["test"].shuffle().select(r)

    data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
    model = AutoModelForSequenceClassification.from_pretrained(
        bert_model_name, num_labels=num_labels
    )

    if output_dir is None:
        output_dir = Path.joinpath(
            BERT_MODEL_DIR,
            bert_model_name,
            get_time(),
        )
    training_args = TrainingArguments(
        output_dir=output_dir,
        learning_rate=2e-5,
        per_device_train_batch_size=1,
        per_device_eval_batch_size=1,
        num_train_epochs=5,
        weight_decay=0.01,
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=small_train_dataset,
        eval_dataset=small_eval_dataset,
        tokenizer=tokenizer,
        data_collator=data_collator,
    )

    trainer.train()
```

F. __init__.py

```python
from pathlib import Path

__version__ = "0.1.0"

```

```
SRC_DIR = Path(__file__).parent.absolute()
WORK_DIR = SRC_DIR.parent
BERT_MODEL_DIR = Path.joinpath(WORK_DIR, "bert_models")
PKL_DIR = Path.joinpath(WORK_DIR, "pkl")
BERT_DEFAULT_MODEL_NAME = "distilbert-base-uncased"
BERT_LAST_HIDDEN_OUT_SIZE = 768
```

G. *gnn_train_utils.py*

```
import torch
from torch import tensor
from gnn_models import GCN


def train_model(data, model, optimizer, criterion):
    model.train()
    optimizer.zero_grad()
    out = model.forward()
    loss = criterion(out[data.train_idx], data.y[data.train_idx])
    loss.backward()
    optimizer.step()
    return loss


def train_model_mlp(model, data, optimizer, criterion, labels):
    model.train()
    optimizer.zero_grad()
    out = model.forward(data)
    loss = criterion(out, labels)
    loss.backward()
    optimizer.step()
    return loss


def test_model_mlp(model, data, train_labels=None, test_labels=None, type=None):
    model.eval()
    out = model.forward(data)
    pred = out.argmax(dim=1)
    if type == "test":
        correct = pred == test_labels
        acc = int(correct.sum()) / len(test_labels)
    else:
        correct = pred == train_labels
        acc = int(correct.sum()) / len(train_labels)
    return pred, acc * 100


def test_model(data, model, type):
    model.eval()
    out = model()
    pred = out.argmax(dim=1)
    if type == "test":
        correct = pred[data.test_idx] == data.y[data.test_idx]
        acc = int(correct.sum()) / int(data.test_idx.sum())
    else:
        correct = pred[data.train_idx] == data.y[data.train_idx]
        acc = int(correct.sum()) / int(data.train_idx.sum())
    return pred, acc * 100


def get_edge_values(c):
    row, col = tensor(c.row).reshape(-1, 1), tensor(c.col).reshape(-1, 1)
    data = c.data
    edge_index = torch.concat((row, col), dim=1).T.long().contiguous()
    edge_attr = tensor(data).reshape(-1)
    return edge_index, edge_attr


def get_gnn_embeddings(model: GCN, n_train, n_test):
    param_list = []
    for param in model.parameters():
        param_list.append(param)
    W1 = param_list[1]
    gnn_embed_train = W1[:, :n_train]
```

```python
    gnn_embed_test = W1[:, n_train : (n_train + n_test)]
    return gnn_embed_train.T, gnn_embed_test.T
```

H. gnn_train.py

```python
# %%
from torch_geometric.data import Data
import torch
from torch import tensor
import numpy as np

#%matplotlib inline
# import matplotlib.pyplot as plt
from eee586.word_embedding import (
    get_token_encodings,
    get_doc_embeddings,
)
from eee586.utils.adjacency import generate_adj_matrix
from gnn_models import GCN, MLP
from gnn_train_utils import (
    train_model,
    test_model,
    get_edge_values,
    train_model_mlp,
    test_model_mlp,
    get_gnn_embeddings,
)

# %%
def get_graph_data(
    train_encods: dict,
    test_encods: dict,
    n_train: int = None,
    n_test: int = None,
    window_size=20,
    stride=1,
) -> Data:

    train_docs, test_docs = train_encods.get("input_ids"), test_encods.get("input_ids")

    train_labels, test_labels = train_encods.get("labels"), test_encods.get("labels")

    if n_test is None:
        n_test = len(test_docs)

    if n_train is None:
        n_train = len(train_docs)

    documents = train_docs[:n_train] + test_docs[:n_test]
    labels = train_labels[:n_train] + test_labels[:n_test]
    doc_vocabs = [set(doc) for doc in documents]
    all_vocab = list(set.union(*doc_vocabs))
    n_nodes = len(all_vocab) + len(documents)

    c = generate_adj_matrix(
        documents,
        dataset_name=f"SetFit/20_newsgroups_ntrain{n_train}_ntest{n_test}",
        window_size=window_size,
        stride=stride,
    )
    edge_index, edge_attr = get_edge_values(c)
    del c
    x = torch.eye(n_nodes)
    y = tensor(labels + (n_nodes - len(labels)) * [0])

    data = Data(x=x, edge_index=edge_index, edge_attr=edge_attr, y=y)

    data.train_idx = tensor(n_train * [True] + (n_nodes - n_train) * [False])
    data.test_idx = tensor(
        n_train * [False] + n_test * [True] + (n_nodes - (n_train + n_test)) * [False]
    )
    return data, all_vocab


```

```python
#%%
def train_strategy(
    train_encods, test_encods, hidden_channels, n_train=None, n_test=None, together=True
):
    """
    If together is True, we will construct single graph for test and train and mask test during training
    """
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    if together:
        data, all_vocab = get_graph_data(
            train_encods,
            test_encods,
            n_train=n_train,
            n_test=n_test,
            window_size=20,
            stride=1,
        )
        data_train = data.to(device)

    # else:
    #     data_train = get_graph_data_train(
    #         train_encods, n_train=100, ratio=0.8, window_size=10, stride=1
    #     ).to(device)
    #     data_test = get_graph_data_test(
    #         test_encods, n_test=40, window_size=10, stride=1
    #     ).to(device)

    model = GCN(layer_no=2, data=data_train).double().to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=0)
    criterion = torch.nn.CrossEntropyLoss().to(device)
    for epoch in range(0, 10):
        loss = train_model(data_train, model, optimizer, criterion)
        if together == True:
            _, train_acc = test_model(data_train, model, type="train")
            _, test_acc = test_model(data_train, model, type="test")
            if epoch % 5 == 0:
                print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
                print(f"Epoch: {epoch:03d}, Loss: {loss:.4f}")

        # else:
        #     _, train_acc = test_model(data_train, model, type="train")
        #     _, test_acc = test_model(data_test, model, type="test")

        #     print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
        #     if epoch % 10 == 0:
        #         print(f"Epoch: {epoch:03d}, Loss: {loss:.4f}")
    return model, all_vocab


# %%
train_encods = get_token_encodings("train")
test_encods = get_token_encodings("test")

n_train = 500
n_test = 50
model, all_vocab = train_strategy(
    train_encods,
    test_encods,
    hidden_channels=[2000, 200, 20],
    n_train=n_train,
    n_test=n_test,
    together=True,
)
# %%
# Run this cell for BERT + GNN embeddings
# continue_with_MLP = False
# if continue_with_MLP:
#     flag = "BERT+GNN embedding"
#     if flag == "BERT+GNN":  # train bert + gnn embeddings with nlp
#         device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#         bert_embed_full_train, bert_embed_full_test = get_doc_embeddings(
#             "train"
```

```
#         ), get_doc_embeddings("test")
#         bert_embed_train = tensor(bert_embed_full_train[:n_train]).to(device)
#         bert_embed_test = tensor(bert_embed_full_test[:n_test]).to(device)
#         gnn_embed_train, gnn_embed_test = get_gnn_embeddings(model, n_train, n_test)

#         embed_out_train = torch.concat((bert_embed_train, gnn_embed_train), dim=1)
#         embed_out_test = torch.concat((bert_embed_test, gnn_embed_test), dim=1)
#         train_labels = tensor(train_encods.get("labels")[:n_train]).to(device)
#         test_labels = tensor(test_encods.get("labels")[:n_test]).to(device)
#     elif flag == "BERT":  # train only bert embeddings with MLP
#         train_encods = get_token_encodings("train")
#         test_encods = get_token_encodings("test")
#         n_train = 10000
#         n_test = 7000
#         device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#         bert_embed_full_train, bert_embed_full_test = get_doc_embeddings(
#             "train"
#         ), get_doc_embeddings("test")
#         embed_out_train = tensor(bert_embed_full_train[:n_train]).to(device)
#         embed_out_test = tensor(bert_embed_full_test[:n_test]).to(device)
#         train_labels = tensor(train_encods.get("labels")[:n_train]).to(device)
#         test_labels = tensor(test_encods.get("labels")[:n_test]).to(device)

#     model_mlp = MLP(input_dim=embed_out_train.shape[1])
#     optimizer = torch.optim.Adam(model_mlp.parameters(), lr=0.005, weight_decay=5e-4)
#     criterion = torch.nn.CrossEntropyLoss()
#     model_mlp = model_mlp.to(device)
#     criterion = criterion.to(device)
#     for epoch in range(0, 10000):
#         loss = train_model_mlp(
#             model_mlp, embed_out_train, optimizer, criterion, train_labels
#         )
#         pred_train, train_acc = test_model_mlp(
#             model_mlp, embed_out_train, train_labels=train_labels, type="train"
#         )
#         pred_test, test_acc = test_model_mlp(
#             model_mlp, embed_out_test, test_labels=test_labels, type="test"
#         )
#         if epoch % 1000 == 0:
#             print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
#             print(f"Epoch: {epoch:03d}, Loss: {loss:.4f}")
#     else:
#         pass
# %%
```

I. *gnn_models.py*

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, GATConv
from torch_geometric.data import Data


class GCN(torch.nn.Module):
    def __init__(self, layer_no, data: Data):
        super().__init__()
        self.data = data
        # self.edge_weight = torch.nn.Parameter(self.data.edge_attr)
        self.edge_weight = self.data.edge_attr
        self.layer_no = layer_no
        if layer_no == 3:
            self.conv1 = GCNConv(data.num_node_features, 2000, cached=True)
            self.conv2 = GCNConv(2000, 200, cached=True)
            self.conv3 = GCNConv(200, 20, cached=True)
        else:
            self.conv1 = GCNConv(data.num_node_features, 200, cached=True)
            self.conv2 = GCNConv(200, 20, cached=True)

    def forward(self):
        x, edge_index = (self.data.x, self.data.edge_index)
        x = x.double()
        x = F.relu(self.conv1(x, edge_index, self.edge_weight))
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv2(x, edge_index, self.edge_weight)
```

```
        x = F.dropout(x, p=0.5, training=self.training)
        if self.layer_no == 3:
            x = self.conv3(x, edge_index, self.edge_weight)
            x = F.dropout(x, p=0.5, training=self.training)
        return x


# model definition
class MLP(torch.nn.Module):
    # define model elements
    def __init__(self, input_dim):
        super(MLP, self).__init__()
        self.linear1 = torch.nn.Linear(input_dim, 100)
        self.linear2 = torch.nn.Linear(100, 20)

    # forward propagate input
    def forward(self, x):
        x = x.float()
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        return x


# def get_graph_data_train(
#     train_encods: dict,
#     n_train: int = None,
#     ratio=0.8,
#     window_size=10,
#     stride=1,
# ) -> Data:

#     """
#     n_train: used to select subset of train dataset to fit the memory
#     ratio: used to divide train set to train and validation
#     """
#     train_docs, train_labels = train_encods.get("input_ids"), train_encods.get("labels")
#     if n_train is None:
#         n_train = len(train_docs)
#     train_docs, train_labels = train_docs[:n_train], train_labels[:n_train]

#     idx = int(len(train_docs) * (ratio))

#     documents, labels = train_docs, train_labels

#     train_docs, val_docs = train_docs[:idx], train_docs[idx:]
#     train_labels, val_labels = train_labels[:idx], train_labels[idx:]
#     n_val, n_train = len(val_docs), len(train_docs)

#     A = generate_adj_matrix(
#         documents,
#         dataset_name=f"SetFit/20_newsgroups_ntrain{n_train}_tv_ratio{ratio}",
#         window_size=window_size,
#         stride=stride,
#     )
#     edge_index, edge_attr, n_nodes = get_edge_values(A)
#     x, y = torch.eye(n_nodes,300), tensor(labels + (n_nodes - len(labels)) * [0])

#     data = Data(x=x, edge_index=edge_index, edge_attr=edge_attr, y=y)

#     data.train_idx = tensor(n_train * [True] + (n_nodes - n_train) * [False])
#     data.test_idx = tensor(
#         n_train * [False] + n_val * [True] + (n_nodes - (n_train + n_val)) * [False]
#     )
#     return data


# def get_graph_data_test(
#     test_encods: dict,
#     n_test: int = None,
#     window_size=10,
#     stride=1,
# ) -> Data:
#     test_docs, test_labels = test_encods.get("input_ids"), test_encods.get("labels")
```

```
#     if n_test is None:
#         n_test = len(test_docs)
#     documents, labels = test_docs[:n_test], test_labels[:n_test]

#     A = generate_adj_matrix(
#         documents,
#         dataset_name=f"SetFit/20_newsgroups_ntest{n_test}",
#         window_size=window_size,
#         stride=stride,
#     )

#     edge_index, edge_attr, n_nodes = get_edge_values(A)

#     x, y = torch.eye(n_nodes), tensor(labels + (n_nodes - len(labels)) * [0])
#     data = Data(x=x, edge_index=edge_index, edge_attr=edge_attr, y=y)
#     data.test_idx = tensor(n_test * [True] + (n_nodes - n_test) * [False])
#     data.train_idx = tensor(n_nodes * [False])

#     return data
```

J. *dataset_trial.py*

```
# %%
from sklearn.datasets import fetch_20newsgroups
from pprint import pprint

newsgroups_train = fetch_20newsgroups(subset="train")

pprint(list(newsgroups_train.target_names))
# %%
print(newsgroups_train.filenames.shape)
print(newsgroups_train.target.shape)
print(newsgroups_train.target[:10])
# %%
from sklearn.feature_extraction.text import TfidfVectorizer

# categories = ['alt.atheism', 'talk.religion.misc',
#               'comp.graphics', 'sci.space']
newsgroups_train = fetch_20newsgroups(subset="train")
vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(newsgroups_train.data)
vectors.shape

# %%
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

newsgroups_test = fetch_20newsgroups(subset="test")
vectors_test = vectorizer.transform(newsgroups_test.data)
clf = MultinomialNB(alpha=0.01)
clf.fit(vectors, newsgroups_train.target)
pred = clf.predict(vectors_test)
metrics.f1_score(newsgroups_test.target, pred, average="macro")
# %%
newsgroups_test = fetch_20newsgroups(
    subset="test", remove=("headers", "footers", "quotes")
)
vectors_test = vectorizer.transform(newsgroups_test.data)
pred = clf.predict(vectors_test)
metrics.f1_score(pred, newsgroups_test.target, average="macro")
# %%
```