# Advanced AI for Data Analytics

« LEADING A DATA SCIENCE PROJECT USING ADVANCED AI TOOLS »

CERTIFICATION RNCP 34126BC02

Ghassen ABDEDAYEM

## Table of content

## 1. Introduction

The aim of this project is to apply advanced Deep Learning techniques to solve a link prediction problem. The problem involves analyzing a graph network with over a hundred thousand nodes, representing scientific papers. The edges in the graph represent links between scientific papers, indicating if one paper cites another. In addition to the graph structure, we are given the abstracts and authors of each paper.

To tackle this challenging task, we can employ various techniques, including Graph Neural Networks, Natural Language Processing, and Deep Learning. Moreover, we need to consider techniques to handle the large volume of data involved in this problem.

In this report, I will present the approaches and techniques used to achieve the goal of link prediction in the context of graph and NLP. I will also describe the difficulties encountered during the project and suggest possible improvements.

## 2. Data specificities

When working with complex datasets, it can be challenging to know where to start. To help us get oriented, let's begin by exploring some of the defining features of our data. So first, we verified the number of nodes and edges of the graph as well as the distribution of connections in some parts of the graph.

Size of the graph

```
Number of nodes = 138,499
Number of the edges = 1,091,955
```

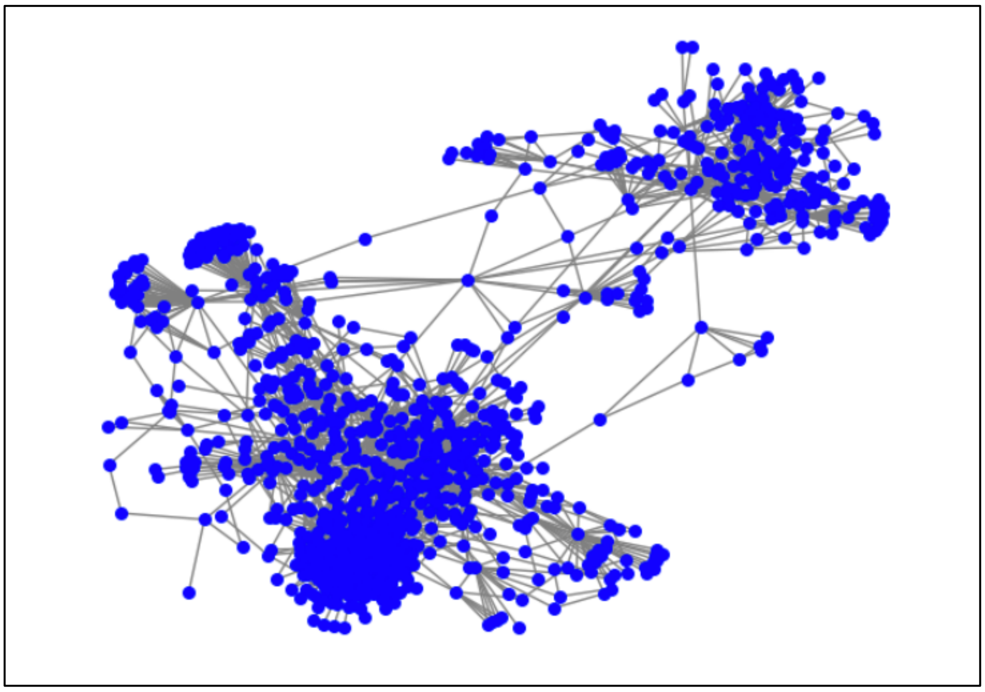


**Figure 1:** 1000 first nodes of the graph

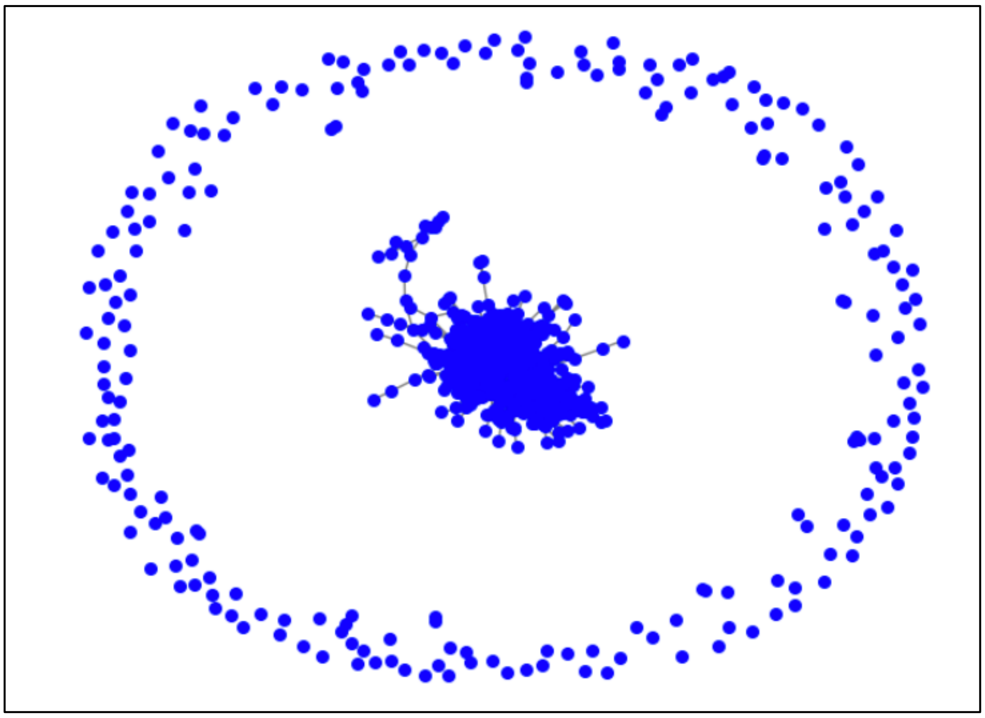


**Figure 2:** 1000 second nodes of the graph

So, as we can see on the figures, the distribution is not homogeneous across the graph, and this is validated by the table of the number of edges and of isolated nodes in the subgraphs of the graph.

| Subgraph nodes | Number of edges | Number of isolated nodes |
|---|---|---|
| 0- 9999 | 56226 | 47 |
| 10,000 – 19,999 | 58756 | 172 |
| 20,000 – 29,999 | 38085 | 712 |
| 30,000 – 39,999 | 35383 | 680 |
| 40,000 – 49,999 | 29300 | 1294 |
| 50,000 – 59,999 | 21488 | 1956 |
| 60,000 – 69,999 | 16846 | 2253 |
| 70,000 – 79,999 | 10622 | 3228 |
| 80,000 – 89,999 | 7739 | 4044 |
| 90,000 – 99,999 | 7984 | 4165 |
| 100,000 – 109,999 | 27856 | 3179 |
| 110,000 – 119,999 | 8038 | 3707 |
| 120,000 – 129,999 | 5133 | 4494 |
| 130,000 – 138,499 | 1510 | 6278 |

**Table1:** Connection of the data by slice of the Graph

Then, when it comes to the abstracts, the length of the text can vary from tens of words to more than one thousand words. So, I list down the findings of the abstracts, and of the cleaned and normalized abstracts. The techniques of cleaning and normalization are explained in a dedicated section later in this report.

Characteristics of the abstracts before normalization

- `Empty abstracts = 7249`
- `Long abstracts more than 128 words = 82,394`
- `Very long abstracts more than 256 words = 4171`
- `Huge abstracts more than 512 words = 65`
- `Longest sentence = 1,462 words`
- `Number of words = 345,570 words`

After text normalization:

- `Empty abstracts = 7249`
- `Long abstracts more than 128 words = 11,217`
- `Very long abstracts more than 256 words = 95`
- `Huge abstracts more than 512 words = 12`
- `Longest sentence = 915 words`
- `Number of words = 188,891 words`

## 3. Data processing

In order to use the data in our machine learning models, we needed to process and prepare it for training. To accomplish this, we applied various operations, which are listed in the subsections below.

### 3.1. Training and validation split

We split the data into training and validation sets to test the model during development. A function called **read_train_val_graph** reads the edgelist file and creates these sets. The validation set is set at a ratio of 0.1 using the val_ratio parameter.

| | Complete set | Training set |
|---|---|---|
| Number of nodes | 138499 | 138499 |
| Number of edges | 1091955 | 982430 |

**Table 2:** Training and validation split

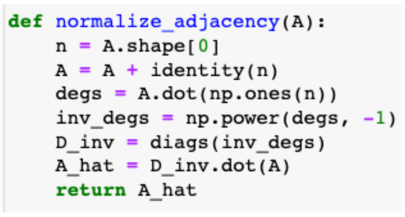
The function first reads the edgelist using the NetworkX **read_edgelist** method. It then creates a dictionary **node_to_idx** to map nodes to integers. It then creates the validation set by iterating through all edges in the graph and randomly removing a proportion **val_ratio** of edges, which are then added to the validation set. The remaining edges are kept as the training set.

It also creates a **y_val** array of labels for the validation set, with **1** indicating that the edge is in the validation set and **0** indicating that it is in the training set.

### 3.2. Graph adjacency matrix

The adjacency matrix is a fundamental representation of a graph that encodes the relationships between its nodes. It is particularly useful in the context of graph neural networks (GNNs) because it provides a concise and efficient way to store and manipulate the graph structure. However, the adjacency matrix can have varying degrees of sparsity, which can lead to numerical instability and poor performance when used directly in GNNs. To address this, we normalized the adjacency matrix.

The **normalize_adjacency** function was used to normalize the adjacency matrix in our project. This function takes an adjacency matrix **A** as input and returns a normalized adjacency matrix **A_hat**. The normalization process involves adding the identity matrix to the adjacency matrix, computing the degree of each node in the graph, and then taking the reciprocal of the degree values. The resulting diagonal matrix is then used to normalize the adjacency matrix. Specifically, each row of the normalized adjacency matrix is divided by the corresponding diagonal element of the diagonal degree matrix.



```python
def normalize_adjacency(A):
    n = A.shape[0]
    A = A + identity(n)
    degs = A.dot(np.ones(n))
    inv_degs = np.power(degs, -1)
    D_inv = diags(inv_degs)
    A_hat = D_inv.dot(A)
    return A_hat
```

**Figure3:** Adjacency matrix normalization function

### 3.3. Random walks features

We incorporated additional features into our model by generating random walks on the graph and applying the word2vec algorithm to the resulting sequences. This approach allows us to capture more nuanced relationships between nodes in the graph, as the random walks provide a way to sample different paths and capture the context in which nodes appear. The dimensionality of the resulting embeddings is reduced using word2vec, allowing us to efficiently incorporate these features into our model. Overall, this technique provides a way to enrich our model with more information about the graph structure, potentially improving its performance.

The chosen parameters of the random walks features are:

- `num_walks = 10`
- `walk_length = 15`
- `wv_vector_size = 64`

## 3.4. Authors features

In our project, we also have the authors' names for each abstract. To capture the common authors between pairs of papers (nodes), we developed a function that counts the number of common authors, for a given pair of nodes. This function takes in pairs of papers and the corresponding authors of each paper. It then computes the number of common authors and a binary value indicating whether the papers have at least one common author using the intersection function.

The idea behind this feature was to add the number of common authors or the binary value as a feature to concatenate with the embedded representation of pairs at the end of the model. However, I didn't see any improvement in the results. Hence, I decided to multiply pairs features with the result of our function plus one to give more weight to the importance of common authors. This approach will be explained in the Model section of this report.

## 3.5. Abstracts text processing

### 3.5.1. Text cleaning and normalization

The first step before processing the abstracts involves cleaning and normalizing the text. This is achieved by removing all special characters, i.e., non-alphanumeric characters. We then apply the stop word function to eliminate meaningless words such as "the", "and", "of", "to", etc., which frequently occur in a language but generally lack significant meaning on their own. Next, we normalize the text using the WordNetLemmatizer 'wordnet' instead of the SnowballStemmer. This is because we aim to reduce words to their base form and subsequently locate the resultant words in used dictionaries and vocabularies. This operation reduced the size of the vocabulary from 345,570 words to 188,891 words.

### 3.5.2. Create the vocabulary

The second step consists in creating a Vocabulary class, which is intended to handle the set of cleaned and normalized abstracts. It manages a dictionary that assigns a distinct index to each distinct word in the collection while simultaneously keeping track of the word count and the nodes or document identifiers in which they appear. The Vocabulary class also provides functions to retrieve a word's index, a given index's word, and a list of all the words in the vocabulary.

### 3.5.3. Mean words word2vec embedding

To obtain word embeddings of the abstracts' words, I utilized the word2vec algorithm, a shallow neural network-based algorithm that employs unsupervised learning of word embeddings from large corpus of text data. The algorithm operates on the basis of the distributional hypothesis, which states that words that occur in similar contexts tend to have similar meanings. It functions by predicting the probability of a target word given a surrounding context of words. During training, the weights of the network are adjusted to maximize the likelihood of predicting the correct target word. Once trained, the network's weights generate a dense vector representation, or embedding, for each word in the vocabulary.

The first approach was to use a pre-trained **Google News 300-dimensional** word2vec model (goog300) to obtain word embeddings of the abstracts. I checked each word in the abstracts' vocabulary against the vocabulary of the pre-trained model, and retrieved corresponding word embeddings for those found. For those not found, I attempted to generate random embeddings for these words, but the processing time proved too high. Consequently, I omitted them from the calculations. Then, averaging the word embeddings for all words in each abstract allowed us to obtain a single vector representation that captured its semantic meaning, leveraging the rich semantic information from the pre-trained word2vec model. Notably, the goog300 embedding produces an output vector of dimension 300, which provides a comprehensive representation of the abstracts' meaning.

The second approach involved creating a **local word2vec** model trained on the vocabulary of the abstracts to ensure that each word had its embedded representation. However, we were unable to capture similarities between words unless they appeared in the same abstracts. Similar to the goog300 approach, we applied mean pooling across the word embeddings of the abstracts to obtain a single representation for each abstract. The size of the output vector of this approach was set to 300 as well to be able to compare the result to goog300 approach.

#### 3.5.4. TF-IDF matrix

In our study, we also implemented the generation of the term frequency-inverse document frequency (tf-idf) matrix for the input corpus of abstracts. This method is widely used in natural language processing to identify the most important words in a document based on their frequency and occurrence across the corpus. We utilized the Python library scikit-learn to generate the tf-idf matrix with normalization and log frequency. Normalization ensures that the values in the matrix are scaled between 0 and 1, while log frequency scales down the importance of high-frequency words. This approach results in a sparse matrix, where each row represents a document and each column represents a term in the corpus vocabulary. The value in each cell represents the tf-idf score for the corresponding term in the corresponding document.

And because our task is a link prediction between abstracts, we reduced the size of TF-IDF matrix by keeping only words that occurs at least in two abstracts and removed the words that occurs in only one abstract. Even if this aspect is already taken into account in the scores of the tf-idf matrix, this reduction would help in the computation performance, the number of kept words is only 62,907 over the total size of the vocabulary of the reduced text which was 188,891.

Then, to reduce the dimensionality of the tf-idf matrix, we used a dense layer in our model rather than PCA. This approach allows for end-to-end learning of feature representations in the model, potentially improving model performance. The dense layer learns a lower-dimensional representation of the tf-idf matrix through the use of non-linear activation functions and backpropagation during training.

### 3.5.5. BART embedding

In this study, we utilized the pre-trained BART (Bidirectional and Auto-Regressive Transformer) model and tokenizer provided by the Hugging Face transformers library. BART is a transformer-based sequence-to-sequence model that excels at tasks such as text generation, summarization, and translation. Our implementation employed the **get_bart_embeddings** function, which generates embeddings using the BART model on input text. Initially, the BART tokenizer splits the input text into a sequence of subword tokens.

By default, the maximum text length accepted by the BART model is 1024 tokens. This can be changed by modifying the max_length parameter in the encoded_input dictionary when tokenizing the text. If the input text is shorter than 1024 tokens, it is padded with zeros up to the maximum length of 1024 tokens. Padding ensures that all inputs have the same length, a necessary requirement for batch processing in deep learning models. The padding token is typically set to 0. This results in a torch tensor of size 1024, representing an embedded representation of each abstract.

The token sequence obtained from the input text is then passed through the BART model to generate contextualized word embeddings. The resulting tensor is obtained by averaging the hidden states of the last layer of the BART model and flattening it into a one-dimensional vector. Our implementation utilized the default embedding dimension for the BART model, which is 1024. However, it is possible to modify the embedding dimension by using a different BART model or fine-tuning the model with a specific embedding dimension.

### 3.5.1. Truncated and padded words embeddings

Finally, to be able to apply CNN and LSTM models, we need to have all the sequence of words embeddings. However, it is required to have a fixed size of the texts to be able to process it into the Deep Learning model. To do so, and for computation and performance purposes, I decided to apply the truncation at the beginning of the processing.

Next, I applied the word2vec model trained on the vocabulary of the abstracts to get a list of 128 words embeddings of 192-vector-size for each abstract. These words embeddings would be used in then in CNN and LSTM models.

Regrettably, the resulting size exceeded the capacity of a single tensor of shape (138,499, 128, 192) and required more than 100 GB of RAM. Although there are techniques to shuffle and batch the data to address this issue, I was unable to implement them at the time.

## 4. Model

To achieve the task of link prediction, I initially implemented a GNN model that relied solely on the graph. I then proceeded to augment this model by integrating additional features and assessing their impact on the overall score. However, given hardware limitations and configuration constraints, I had to limit the model's complexity. As a result, I couldn't incorporate as many features and layers as desired, which could have potentially affected the model's performance. Despite these limitations, I utilized the available resources to the best of my ability and still managed to attain favorable outcomes.

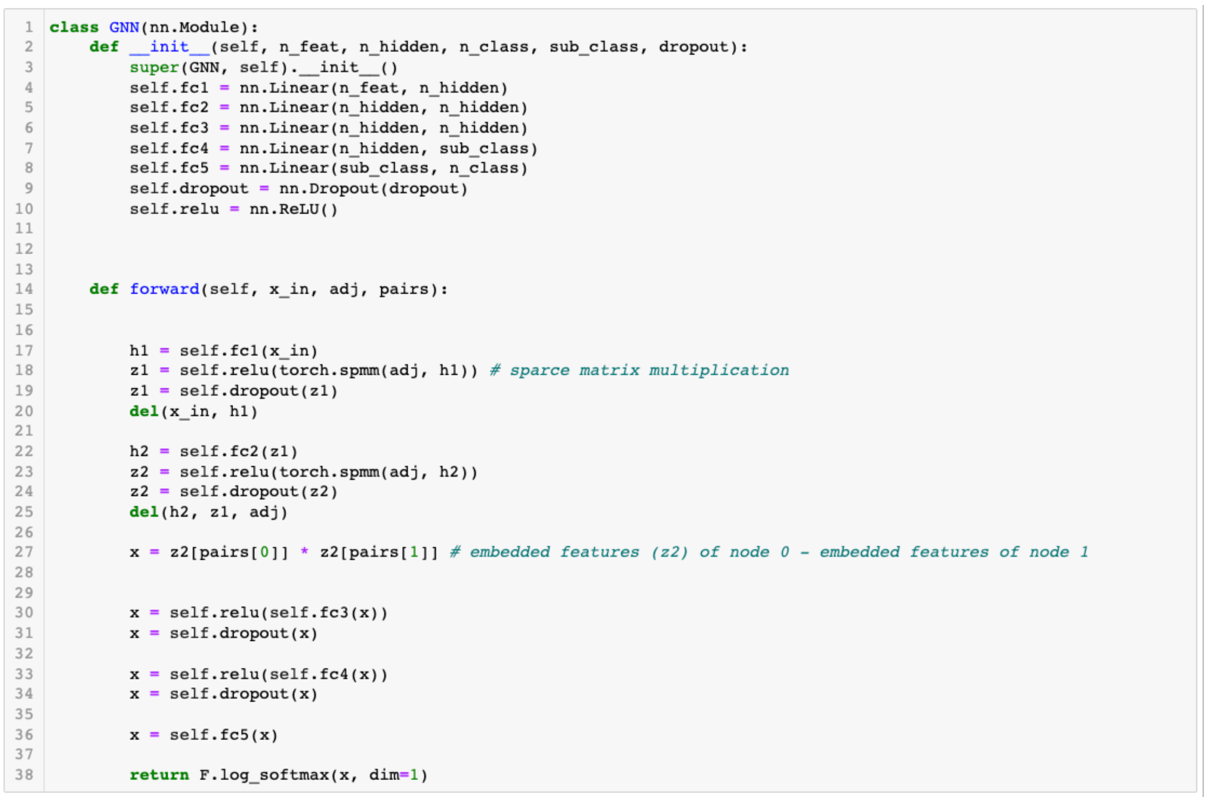
### 4.1. GNN model

The used model is a PyTorch implementation of a Graph Neural Network of a message passing layer. The forward method of this model takes in the input tensor x_in, the adjacency matrix adj, and the pairs tensor, which contain pairs of nodes indices for which we want to predict the presence or absence of a link.

The input tensor is first fed through the first fully connected layer (fc1) and then the result is multiplied by the adjacency matrix using sparse matrix multiplication (spmm), which generates the hidden representation of the nodes (z1). The hidden representation is then passed through a ReLU activation function and a dropout layer.

The same process is repeated for the second fully connected layer (fc2) to generate the final hidden representation of the nodes (z2). Then, the embedded features (z2) of the two nodes in each pair are multiplied to create a feature vector for each pair. This vector is passed through two additional fully connected layers (fc3 and fc4) with ReLU activation functions and a dropout layer. Finally, the output is passed through a final fully connected layer (fc5) with a log softmax activation function.

This implementation is considered as a message passing layers because it applies the same operation (i.e., linear transformation and activation function) to each node's features and its neighbors' features during the forward pass. Specifically, the **torch.spmm** function performs a sparse matrix multiplication between the adjacency matrix and the node features, resulting in a matrix that contains aggregated information from each node's neighbors. This matrix is then passed through two fully connected layers, which can be seen as message passing layers, before producing the final output.



```python
class GNN(nn.Module):
    def __init__(self, n_feat, n_hidden, n_class, sub_class, dropout):
        super(GNN, self).__init__()
        self.fc1 = nn.Linear(n_feat, n_hidden)
        self.fc2 = nn.Linear(n_hidden, n_hidden)
        self.fc3 = nn.Linear(n_hidden, n_hidden)
        self.fc4 = nn.Linear(n_hidden, sub_class)
        self.fc5 = nn.Linear(sub_class, n_class)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()



    def forward(self, x_in, adj, pairs):


        h1 = self.fc1(x_in)
        z1 = self.relu(torch.spmm(adj, h1)) # sparce matrix multiplication
        z1 = self.dropout(z1)
        del(x_in, h1)

        h2 = self.fc2(z1)
        z2 = self.relu(torch.spmm(adj, h2))
        z2 = self.dropout(z2)
        del(h2, z1, adj)

        x = z2[pairs[0]] * z2[pairs[1]] # embedded features (z2) of node 0 - embedded features of node 1


        x = self.relu(self.fc3(x))
        x = self.dropout(x)

        x = self.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.fc5(x)

        return F.log_softmax(x, dim=1)
```

**Figure 4:** First GNN model

This model will be called “GNN n_hidden- n_hidden- n_hidden- sub_class- n_class”.

First, we tested it with the following parameters:

- N_hidden = 64
- Sub_class = 8
- N_class = 2

And with a random vector of size 64 as a node feature. The obtained result was 0.4640. Then we tested various combinations of features inputs in the next sections. All the results with the different tested configurations will be consolidated in one table as appendices at the end of this report.

### 4.2. Adding the first features

To choose between the different features, I tested the model with different configurations.

The training parameters were almost the same for all these configurations:

- Optimizer: Adam
- Learnnig rate: 0,01 (and for some cases we devided the lr by 10, but there is no real impact on the results at this stage)
- Epoch: 200

Parameters:

- Pairs features: how we construct the pair features from the nodes features (difference, mean, concatenation or multiplication of the pairs embedding)
- Model: the number of perceptions of each layer, and if we applied dense layer to large sparce inputs.
- Node features:
  - Walks_wv: the random walks then reduced with word2vec
  - Goog300: mean of the words' embeddings obtained with Google News 300 model
  - Local_wv_300: mean of the embeddings obtained with a word2vec trained on the abstracts
  - Tf-idf: tf-idf matrix first densified with one or two dense layers in the model
  - Bart1024: abstract embedding obtained with Bart model

| Model | node features | pairs features | log_loss_val |
|---|---|---|---|
| GNN 64-64-64-8-2 | walks_wv | Mean | 0,2464 |
| GNN 64-64-128-8-2 | walks_wv | Concatenation | 0,2495 |
| GNN 64-64-64-8-2 | walks_wv | Multiplication | 0,2562 |
| GNN 64-64-128-8-2 | local_wv_300 | Concatenation | 0,2942 |
| GNN 64-64-128-8-2 | goog300 | Concatenation | 0,2979 |
| Dense 128- GNN 64-64-128-8-2 | bart1024 | Concatenation | 0,3208 |
| Dense 1024- GNN 64-64-64-8-2 | tf-idf | Concatenation | 0,3310 |
| Dense 1024-128- GNN 64-64-128-8-2 | tf-idf | Concatenation | 0,3442 |
| GNN 64-64-64-8-2 | local_wv_300 | Mean | 0,3450 |
| GNN 64-64-128-8-2 | random | Concatenation | 0,4358 |
| Dense 128- GNN 64-64-64-8-2 | bart1024 | Mean | 0,4469 |
| GNN 64-64-64-8-2 | goog300 | Mean | 0,4615 |
| GNN 64-64-64-8-2 | random | Multiplication | 0,4640 |
| GNN 64-64-64-8-2 | bart1024 | Multiplication | 0,4813 |
| Dense 128- GNN 64-64-64-8-2 | bart1024 | Multiplication | 0,4818 |
| GNN 64-64-64-8-2 | walks_wv | Difference | 0,6239 |

**Table 3:** Scores of the first GNN model with other features

The results indicate that the walks_wv feature has the most significant positive impact on the model's performance, followed by other text features that also enhance the model's score, albeit only when used in combination with concatenated pairs of features. Moving forward, we will focus on refining our model by retaining the walks_wv feature and combining it with one or two text features. Additionally, we plan to incorporate authors' features in the subsequent step.

## 4.3. Applying authors features

For this task, I prepared a function that takes as input the list of authors for each paper and a pair of nodes (papers) and returns two outputs: a binary of if the two papers of the pair have at least one author and the number of common authors. This function is called during the training to provide the pairs features corresponding to the authors.

The outputs of this function are then combined with the random walks features by multiplication of concatenation.

| model | features | pairs features | authors | epochs | lr | score |
|---|---|---|---|---|---|---|
| GNN 64-64-128-8-2 | walks_wv | concatenation | X 2 if same auth | 200 | 0,01 | 0,2442 |
| GNN 64-64-128-8-2 | walks_wv | concatenation | X 0 if not same auth | 200 | 0,01 | 0,2482 |
| GNN 64-64-66-8-2 | walks_wv | mean | concatenate with number of auth and binary | 200 | 0,01 | 0,2565 |
| GNN 64-64-128-8-2 | walks_wv | concatenation | X nb of common auth + 1 | 200 | 0,01 | 0,2573 |
| GNN 64-64-128-8-2 | walks_wv | concatenation | X 0,5 if not same auth | 191 | 0,01 | 0,2579 |
| GNN 64-64-64-8-2 | walks_wv | mean | X 2 if same auth | 200 | 0,01 | 0,2627 |
| GNN 64-64-130-8-2 | walks_wv | concatenation | concatenate with number of auth and binary | 186 | 0,01 | 0,2677 |
| GNN 64-64-64-8-2 | walks_wv | mean | X nb of common auth | 200 | 0,01 | 0,2844 |
| GNN 64-64-64-8-2 | walks_wv | mean | X 0,5 if not same auth | 200 | 0,01 | 0,2928 |
| GNN 64-64-64-8-2 | walks_wv | mean | X nb of common auth + 1 | 200 | 0,01 | 0,3085 |
| GNN 64-64-64-8-2 | local_wv_300 | concatenation | X nb of common auth + 1 | 200 | 0,01 then 0,001 | 0,3476 |
| GNN 64-64-64-8-2 | local_wv_300 | concatenation | X nb of common auth | 200 | 0,01 then 0,001 | 0,3634 |
| GNN 64-64-64-8-2 | local_wv_300 | mean | X nb of common auth | 200 | 0,01 then 0,001 | 0,3855 |
| GNN 64-64-64-8-2 | goog300 | concatenation | X nb of common auth + 1 | 200 | 0,01 then 0,001 | 0,4464 |
| GNN 64-64-64-8-2 | walks_wv | mean | X 0 if not same auth | 26 | 0,01 | 0,6929 |

**Table 4:** Scores of the GNN model with features combined with authors

Although we employed this method, the model scores did not show significant improvement. However, a slight enhancement was noticed when we combined the concatenated pairs of walks features by multiplying them by 2 in cases where there was at least one common author.

Although there are other techniques that could be used, such as using a sparse representation of authors and densifying it through a Dense layer or creating a graph of authors and learning graph representation from it, these approaches were not explored due to computational challenges and time constraints.

Moving forward, we will maintain the current combination of parameters of walks_wv features, and concatenation of the node features embeddings to generate pairs embeddings, multiplied by 2 if there is at a shared author between the two papers. Our next step is to improve the model by incorporating additional abstracts features.

### 4.4. Adding abstracts features

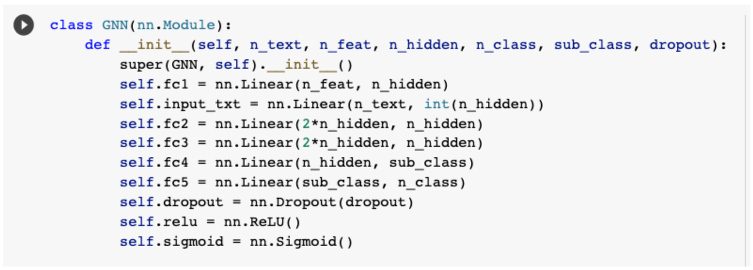
In the previous section, we defined the GNN model. Now, we will adapt it to incorporate abstract text features in addition to the walks_wv features. Since the model is becoming larger, we increased the number of epochs to 600. To prevent overfitting, we implemented an early stopping function that reduces the learning rate by 10 each time the validation loss does not improve.

Based on the tests of the last sections, we will use walks_wv features multiplied by 2 for shared authors between the papers. We additionally integrate the local_wv_300, the mean of words embeddings obtained with a Word2vec trained on the vocabulary of the abstracts.

The finetuning task now consists in adapting the model and defining at which stage the local_wv_300 features is combined with the walks_wv and the GNN (by the multiplication with the adjacency matrix).

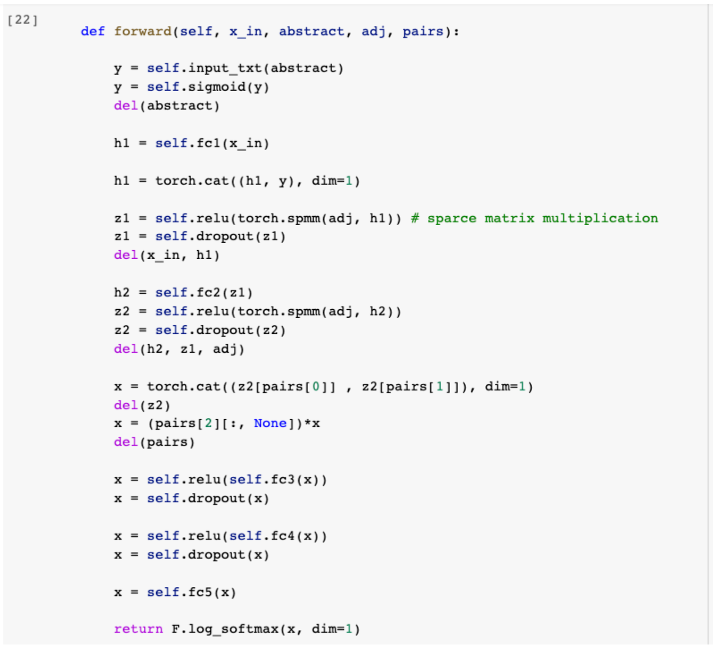
#### 4.4.1. Applying Dense then cat node features then GNN

The first model consists in densifying the input to the n_hidden = 64 then to concatenate it with the walk_wv features, then to apply the same logic as the previous GNNs. The obtained score of log loss = **0,2414** which is a slight improvement of previous scores.



```
class GNN(nn.Module):
    def __init__(self, n_text, n_feat, n_hidden, n_class, sub_class, dropout):
        super(GNN, self).__init__()
        self.fc1 = nn.Linear(n_feat, n_hidden)
        self.input_txt = nn.Linear(n_text, int(n_hidden))
        self.fc2 = nn.Linear(2*n_hidden, n_hidden)
        self.fc3 = nn.Linear(2*n_hidden, n_hidden)
        self.fc4 = nn.Linear(n_hidden, sub_class)
        self.fc5 = nn.Linear(sub_class, n_class)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
```

**Figure 5:** GNN model with walks_wv fatures and abstract embedding

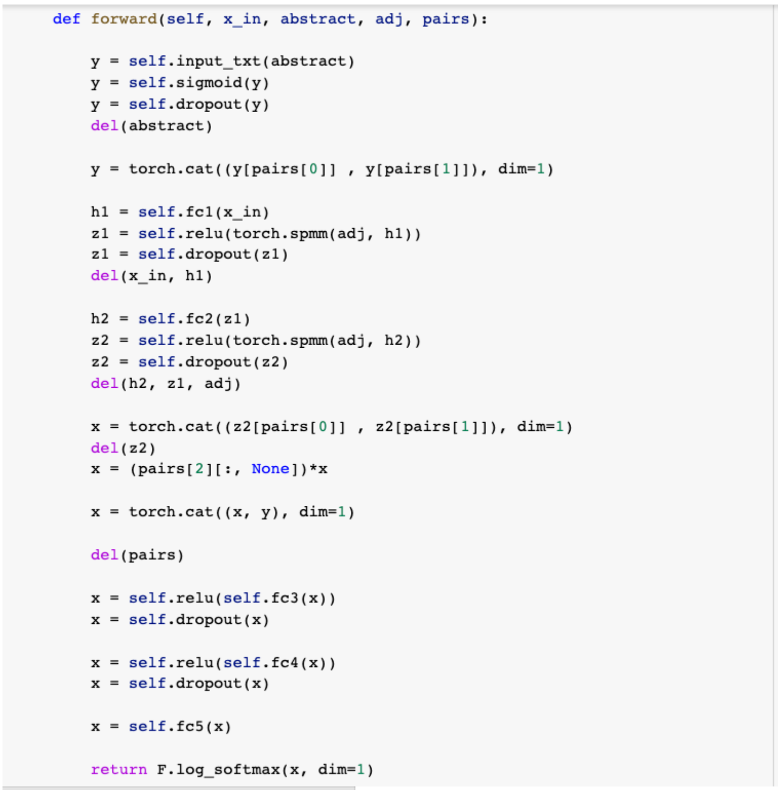


```
[22]
    def forward(self, x_in, abstract, adj, pairs):

        y = self.input_txt(abstract)
        y = self.sigmoid(y)
        del(abstract)

        h1 = self.fc1(x_in)

        h1 = torch.cat((h1, y), dim=1)

        z1 = self.relu(torch.spmm(adj, h1)) # sparse matrix multiplication
        z1 = self.dropout(z1)
        del(x_in, h1)

        h2 = self.fc2(z1)
        z2 = self.relu(torch.spmm(adj, h2))
        z2 = self.dropout(z2)
        del(h2, z1, adj)

        x = torch.cat((z2[pairs[0]] , z2[pairs[1]]), dim=1)
        del(z2)
        x = (pairs[2][:, None])*x
        del(pairs)

        x = self.relu(self.fc3(x))
        x = self.dropout(x)

        x = self.relu(self.fc4(x))
        x = self.dropout(x)

        x = self.fc5(x)

        return F.log_softmax(x, dim=1)
```

**Figure 6:** Forward function of the model

### 4.4.2. Abstract features cat nodes GNN embedding

This approach consists in processing the text with a Dense layer, then concatenate it to the output of the GNN and authors multiplication. This approach showed the best score of our study with a log loss = **0,2027**.



```
def forward(self, x_in, abstract, adj, pairs):

    y = self.input_txt(abstract)
    y = self.sigmoid(y)
    y = self.dropout(y)
    del(abstract)

    y = torch.cat((y[pairs[0]] , y[pairs[1]]), dim=1)

    h1 = self.fc1(x_in)
    z1 = self.relu(torch.spmm(adj, h1))
    z1 = self.dropout(z1)
    del(x_in, h1)

    h2 = self.fc2(z1)
    z2 = self.relu(torch.spmm(adj, h2))
    z2 = self.dropout(z2)
    del(h2, z1, adj)

    x = torch.cat((z2[pairs[0]] , z2[pairs[1]]), dim=1)
    del(z2)
    x = (pairs[2][:, None])*x

    x = torch.cat((x, y), dim=1)

    del(pairs)

    x = self.relu(self.fc3(x))
    x = self.dropout(x)

    x = self.relu(self.fc4(x))
    x = self.dropout(x)

    x = self.fc5(x)

    return F.log_softmax(x, dim=1)
```

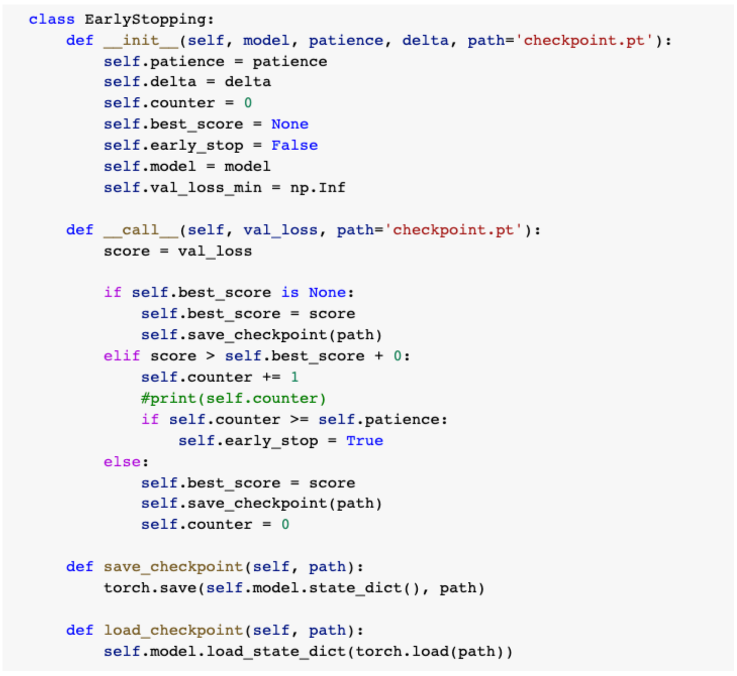
**Figure 7:** Forward function of the Abstract features to GNN embedding

### 4.5. Training parameters

To accomplish the task of training, training functions have been defined. Some parameters have been set to facilitate these heavy operations.

#### 4.5.1. Learning rates and early stopping

After multiple tests and combinations, the learning rate have been set initially to 0.01. During the training, and when the scores stop improving at a certain point, the learning rate is divided by 10 to find better parameters that were not found with big learning rate. A function accomplished this task of determining the moment when we divide the lr by 10. This showed a good improvement for large model with text parameters.



```python
class EarlyStopping:
    def __init__(self, model, patience, delta, path='checkpoint.pt'):
        self.patience = patience
        self.delta = delta
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.model = model
        self.val_loss_min = np.Inf

    def __call__(self, val_loss, path='checkpoint.pt'):
        score = val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(path)
        elif score > self.best_score + 0:
            self.counter += 1
            #print(self.counter)
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(path)
            self.counter = 0

    def save_checkpoint(self, path):
        torch.save(self.model.state_dict(), path)

    def load_checkpoint(self, path):
        self.model.load_state_dict(torch.load(path))
```

**Figure 8:** Early stopping class

## 5. Possible improvements

In this section, we will outline some potential areas for improvement in this project that could be explored in future work.

### 5.1. Authors data

One major area for improvement is the utilization of author information. Although this study attempted to incorporate authors by multiplying the features of pairs with common authors by 2, the results were not significantly improved. However, using a sparse representation of authors and densifying them could potentially create embedded representations of authors and yield better results. Unfortunately, this approach was not explored due to performance limitations and difficulties in handling sparse matrices early in the project. Another possible approach would be to construct an author graph and apply graph neural networks (GNN) to better leverage this information. However, this method was not explored due to its complexity and time constraints.

### 5.2. Abstracts text

Another aspect that could be improved is the use of abstracts. The TF-IDF matrix and the mean of word embeddings did not lead to significant improvements in the results. Although some combinations of max, min, and quartiles were tested, they also did not show any improvement. However, exploring the use of CNN, LSTM or ELMO algorithms to study words in their context could lead to better results. Unfortunately, technical and performance limitations prevented me from handling all the word embeddings of the abstracts. This analysis would require more technical resources, as it is resource-intensive. However, using techniques such as batching or Monte Carlo sampling could help to solve this issue.

### 5.3. Sampling and batching

Another way to improve the analysis would be to shuffle the graph, which would address the issue of unbalanced edge distribution and enable easier data batching without introducing bias. This would also facilitate exploratory analysis on relevant subsets of the data through easy sampling. Ideally, this step should be performed early in the project to streamline subsequent work.

### 5.4. Supervised densification of random walks

The model's performance significantly improved by using the random walks as a feature. We obtained this feature by applying word2vec on the walks to densify them in an unsupervised way. However, we could opt for supervised densification using a Dense layer in our model, even if it could have made the model more complex and difficult to train.

## 6. Conclusion

GNN, which is recognized as a potent method for capturing graph interactions, did not perform well on its own for link prediction in this study. However, combining GNN with other features resulted in significant improvements in model performance. For instance, when GNN was combined with random walks, which are another representation of the graph structure, we observed a considerable enhancement in performance. Additionally, we explored different finetuning parameters, such as multiplying the features of pairs with common authors, which resulted in minor improvements.

However, the use of abstract text features, which were concatenated with the output of the GNN, led to only limited improvements. This is likely due to the fact that the abstract features used in this study relied solely on the average (or mean) of word embeddings, which may not capture the full meaning of the text. To improve the representation of abstracts, other techniques that analyze words in their context may be necessary.