POLITICAL POSTS CLASSIFICATION USING GRAPH NEURAL NETWORKS AND TEXTUAL EMBEDDINGS: DISTINGUISHING CONSERVATIVE AND LIBERAL PERSPECTIVES

STUDENTS: Alessio Mattiace, Claudio Saponaro

INTRODUCTION:

This project aims to predict political orientation labels based on graph-structured data. The methodology employs a transformer-based technique to generate embeddings from blog posts within the dataset. These embeddings, in conjunction with edge information from the graph, serve as input to a Graph Convolutional Network (GCN).

The resulting dataset, consisting of the aforementioned embeddings, was partitioned into training, validation, and test sets. The training and validation sets were utilized to optimize the model's hyperparameters, including learning rate and weight decay, through a fine-tuning process. Subsequently, the test set was employed to generate predictions and compute relevant performance metrics.

This approach leverages both textual content and network structure to enhance the accuracy of political orientation prediction, demonstrating the potential of combining natural language processing techniques with graph-based machine learning methods.

This combination allows for a more comprehensive analysis, where the political orientation of a blog is inferred not only from its content but also from its connections to other blogs in the network.

STATE OF THE ART

Graph Convolutional Networks (GCN):

This section is dedicated to an in-depth study of the state of the art regarding the model we have used in our project, based on the paper [1] which marked a breakthrough in this field.

We chose to use this architecture due to its widespread adoption and effectiveness in node classification tasks.

The key operation in GCN is graph convolution, which can be viewed as a message passing operation between adjacent nodes, like in the standard Graph Neural Network (GNN). This operation aims to aggregate information from neighboring nodes for each node in the graph.

GCNs are typically characterized by two main steps: node aggregation and node update.

1. Node Aggregation: The node aggregation step collects information from neighboring nodes, weighted by the edge connections and normalized by node degrees. It can be represented as:

$$x'_i{}^{(l)} = \sum_{j \in N(i)} \frac{e_{ji}}{\sqrt{d_j}\sqrt{d_i}} * h_j{}^{(l-1)}$$

   Where:

   - $x'_i{}^{(l)}$: The aggregated feature representation for node i at layer l.
   - N(i): The set of neighbors of node i.
   - e_{ji}: The weight of the edge from node j to node i.
   - d_j and d_i: The degrees of nodes j and i, respectively, including any self-loops.
   - $h_j{}^{(l-1)}$: The feature vector of node j from the previous layer.

   The normalization factor $\sqrt{(d_j d_i)}$ ensures stability in the learning process by preventing the scale of features from growing or shrinking exponentially with network depth.

2. Node Update: The node update step applies a learnable transformation to the aggregated features and passes them through a non-linear activation function:

$$h_i{}^{(l)} = \sigma\left(W^{(l)} x'_i{}^{(l)}\right)$$

   Where:

   - $h_i{}^{(l)}$: The updated feature representation for node i at layer l.
   - $W^{(l)}$: A matrix of learnable weights for layer l.
   - σ: A non-linear activation function (e.g. ReLU or sigmoid).

This two-step process allows the GCN to learn hierarchical representations of nodes by iteratively aggregating and transforming neighborhood information across multiple layers.

ChebNet:

The second model we have tested is the ChebNet, which is explained with detail in [2].

The main idea of ChebNet implementation is to use Chebyshev polynomials to approximate the convolutional filter, thus reducing computational complexity and allowing more efficient localization of the filter in the spatial domain.

These polynomials are computed as follows:

- $T(0) = 1$
- $T(1) = X$
- $T(n+1) = 2 \cdot X \cdot T(n) – T(n-1)$ for $n > 1$.

It is supported in PyTorch Geometric with the ChebConv module.

Graph Attention Networks (GAT):

The last model we have implemented is based on Graph Attention Networks (GATs), which utilize the attention mechanism.

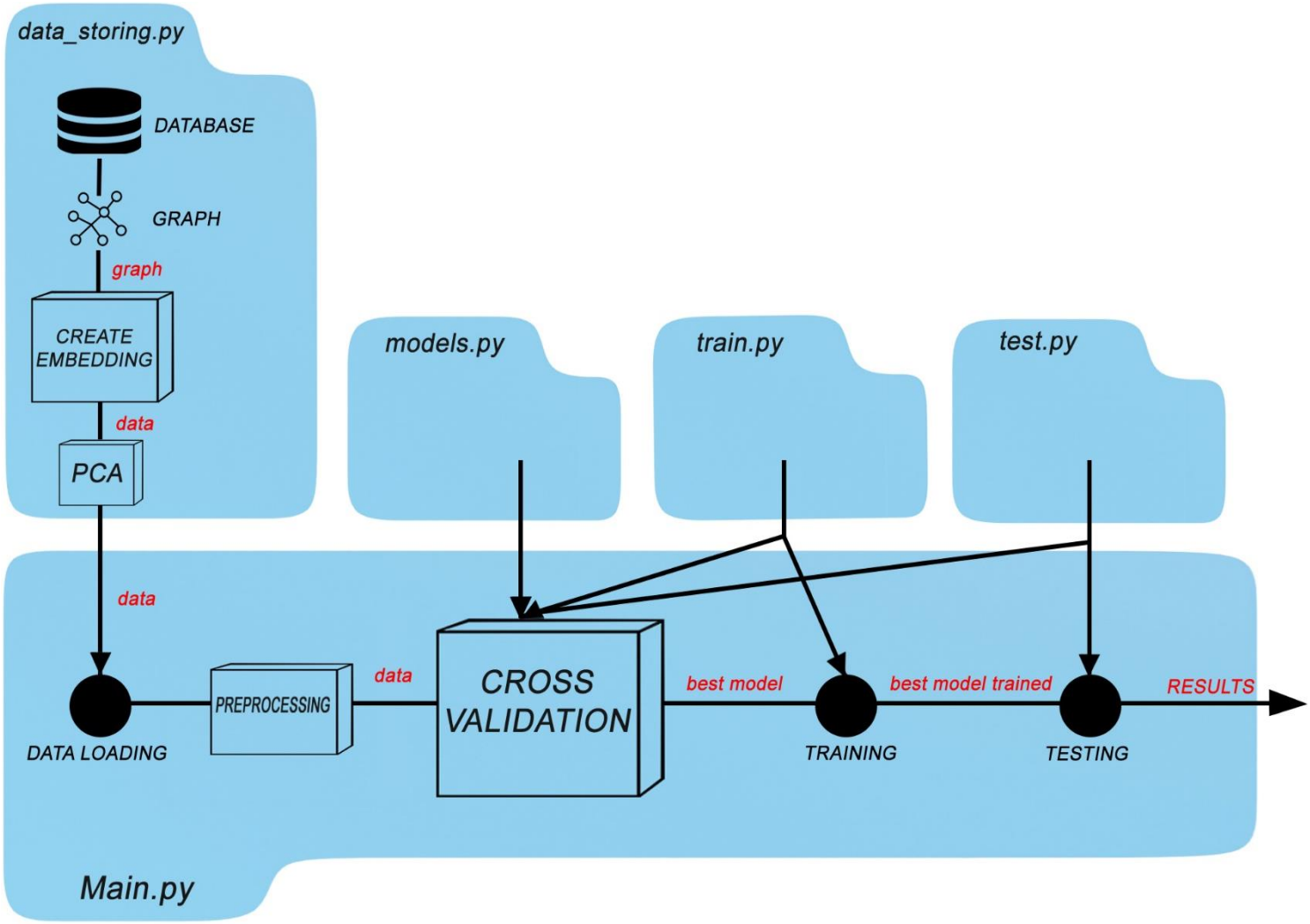
The fundamental concept of GATs is to employ an attention mechanism to assign weights to neighboring nodes, enabling the network to focus on the most relevant nodes during information aggregation. This approach addresses certain limitations of Graph Convolutional Networks (GCNs), where edge weights are fixed.

The mathematical details of this model are thoroughly presented in the paper [3].

This work inspired the 'GATConv' model implementation in PyTorch Geometric that we decided to use as an option in the project.

PROCESS VISUALIZATION

Here a simple representation of each phase we have followed in our project in pipeline, starting from the dataset and arriving to the results:



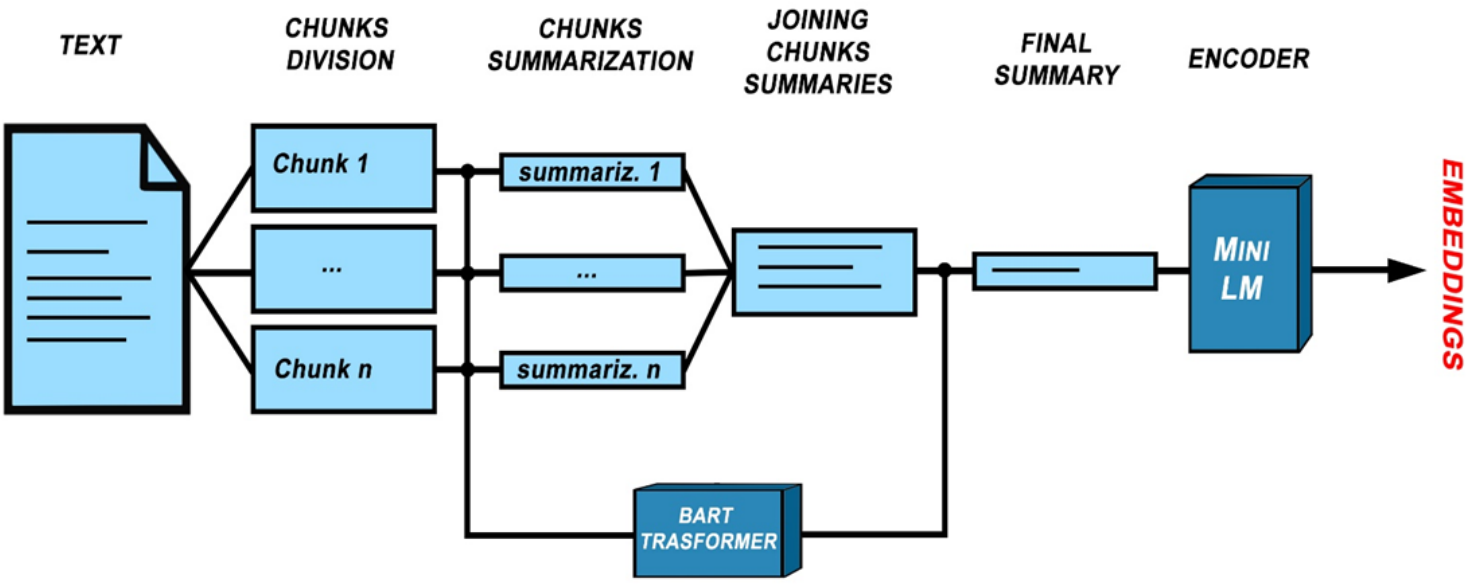
SECTION 1 – DATA STORING

Our first goal has been to create a meaningful graph structure starting from our database ‘polblogs.gml’.

Using networkx library in python, we have created a multigraph structure because of the presence of duplicate edges in the dataset; then we filled a simple directed graph with same nodes and edges in which are allowed directed edges, but not multiple ones: we have considered duplicate edges as a single edge with multiple cardinality.

During the filling process we have checked the nodes having a valid url an http status code lower than 300 and added them to the graph, after that we have added edges only if consistent with nodes.

After managing the graph structure, we implemented an architecture to obtain embedding representations of data in our database; a graphical representation of each step of our architecture follows:



To obtain embedded representations of data, we have chosen ‘MiniLM-v2’ as encoding model, which accepts input lengths no greater than 512 tokens for each blog entry, truncating content beyond this limit.

To mitigate the loss of pertinent information due to truncation, we opted to employ a transformer architecture for initial content summarization.

Specifically, we utilized the ‘facebook-bart-large-cnn’ model provided by Hugging Face. This BART model is pre-trained on the English language and fine-tuned on the CNN/Daily Mail dataset.

Comprehensive information regarding this architecture is documented in [4].

BART is a transformer encoder-decoder (seq2seq) model that combines a bidirectional (BERT-like) encoder with an autoregressive (GPT-like) decoder.

According to the documentation provided by Hugging Face, this model is particularly well-suited for summarization tasks, but works only with quite small inputs of texts, so we divided it in smaller chunks, applying BART summarizer to each of them; after joining them, we opted for another final summarization

to obtain a meaningful result as input to our encoding model in order to have significant embeddings in the next step.

For calculating the embeddings, we have opted to use the 'all-MiniLM-L6-v2' model, an encoder-only architecture that maps sentences into 384-dimensional dense vectors.

These vectors capture semantic information that will subsequently be utilized as node features in the training phase. Lastly, we have used the PCA (principal component analysis) to reduce the number of features, from 384 to 50 principal components, to improve the velocity in the training phase.

SECTION 2 – ARCHITECTURES, CROSS-VALIDATION & TRAINING

We have selected three different models for the cross-validation and training phases, using the different convolutional layers mentioned before.

Each architecture is built in the same way, only changing convolutional layer and the activation function, to standardize the cross-validation process.

The forward function uses three convolutional layers of the same type (with three hidden dimensions passed as an argument by the CV phase) and, after each of them, a non-linear activation function is applied.

OBSERVATION: We used the dropout technique with probability of 0.2; it is very able to decrease overfitting in our task during training.

A fully connected linear layer is applied at the end to reduce the dimension of the output to 1; then, feeding a Softmax function, we obtained the output as probability (Sigmoid is used as Softmax for binary tasks).

The combination of convolutional layers and activation functions used follows:

- GCNConv (with edge index symmetric normalization) + ReLU.
- ChebConv (with symmetric normalization for the edge index, and a polynomial degree K=3) + Leaky ReLU.
- GATConv (with 1 attention head as default) + ELU.

Before validation phase, as preprocessing, we used a robust scaler to reduce outlier influences on our data, then we instantiated k-fold splitter to obtain train and validation indexes for our K-Fold cross-validation (using 5 folds).

We have created a param grid using a dictionary data structure in which we stored a set of parameters to iterate:

- Learning rate
- Weight decay (regularization parameter to prevent overfitting)
- Hidden dimensions

OBSERVATION: learning rate and weight decay are the optimizer’s initial arguments; however, they change dynamically during epochs.

So, we implemented CV with two loops, iterating over parameters combinations and folds; in each loop we train our model on training set and compute accuracy on validation set using test function: this is the score of that specific fold for that specific combination of parameters.

OBSERVATION: during each training epoch, we get the predictions on the whole dataset with a forward step and then we compute the loss on the training samples for the backpropagation.

We store each fold score in a vector and then mean it to compute the params combination score.

As optimizers, we choose to use three options and test them once at a time combined to different models:

- Adam
- Adagrad
- RMSProp

After iterating over all possible combination of parameters, we obtain the best parameters combination based on the score: we use that specific combination and start the final training of the best model.

OBSERVATION: in the training phase we also store each epoch loss in a vector to plot a loss history.

SECTION 3 – EVALUATION PHASE

TESTING:

The last section of our work is dedicated to the testing and the evaluation of the results we have obtained using the architectures previously explained.

We have implemented a test function to calculate the predictions on the test set and compute a set of metrics useful for the results evaluation.

OBSERVATION: the same test function is used both for the validation and test phase.

The metrics we have used are:

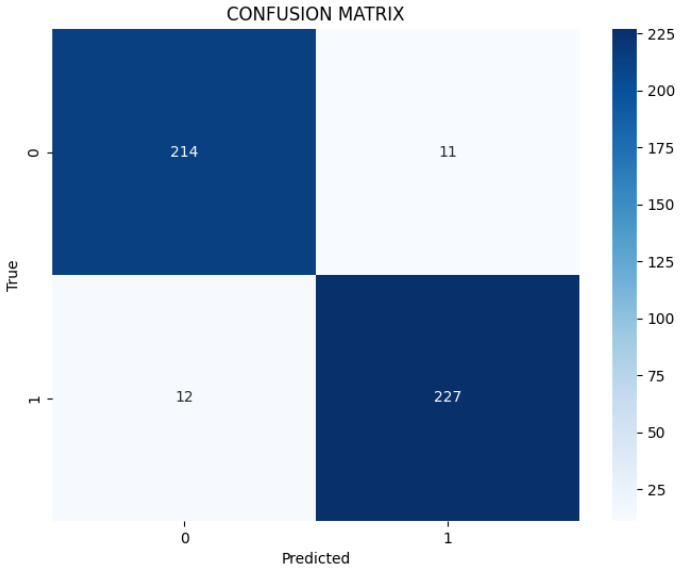
- Accuracy
- Precision
- Recall
- F1- score

Below are the results of our work, including evaluation metrics, the confusion matrix, and the loss history for some combination of models and a specific optimizer:

(all these results are obtained with the best possible parameters combination in order of performances).

RESULTS:

- **GAT Conv + Adam**

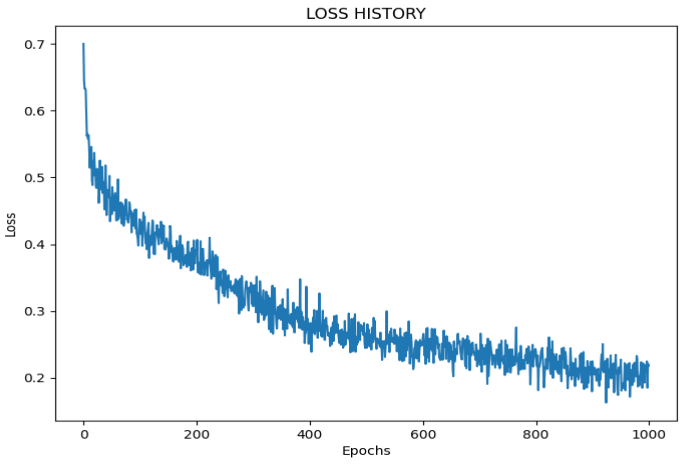


Accuracy: 95.04 %

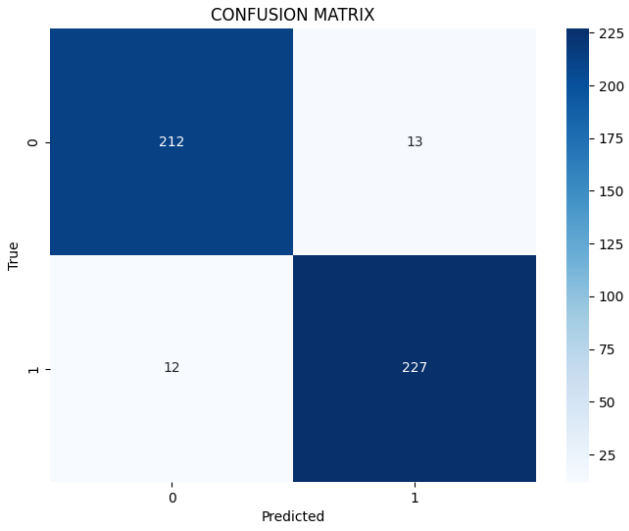
Precision: 95.38%

Recall: 94.98%

F1_Score: 95.18%



- **GCN Conv + RMSProp**

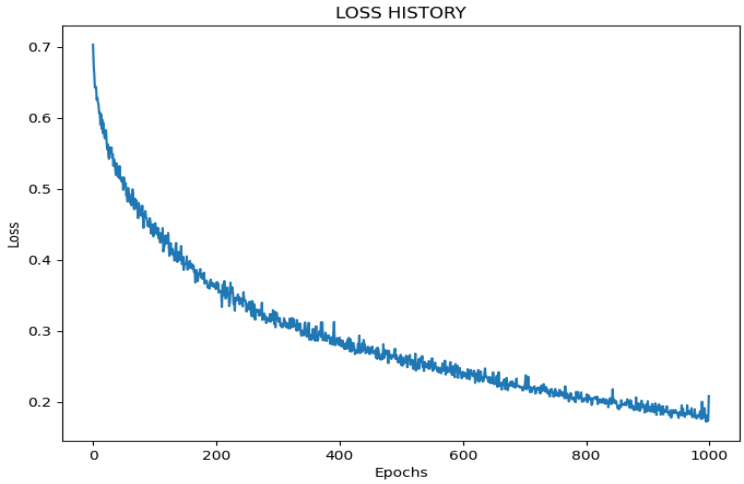


Accuracy: 94.61 %

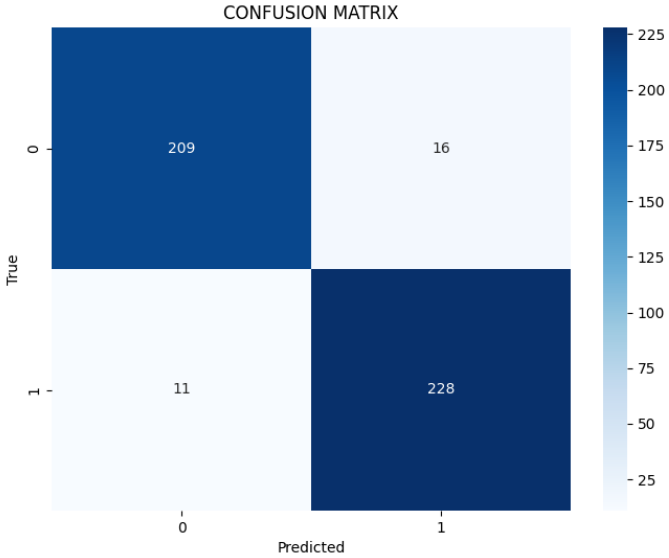
Precision: 94.58%

Recall: 94.98%

F1_Score: 94.78%



- **GAT Conv + RMSProp**

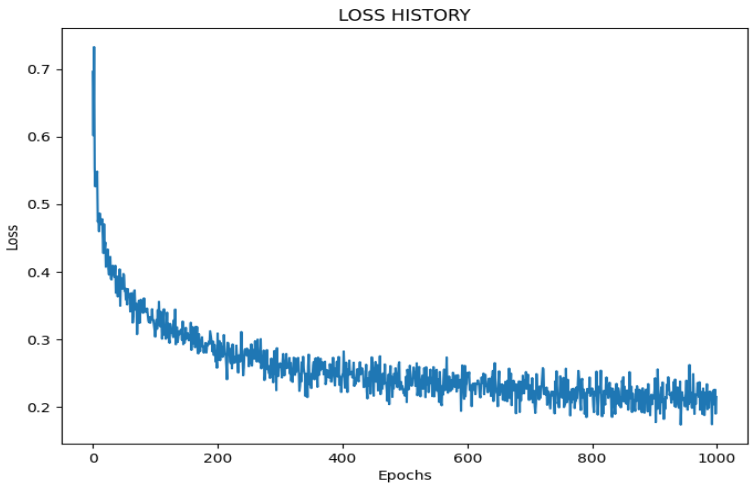


Accuracy: 94.40 %

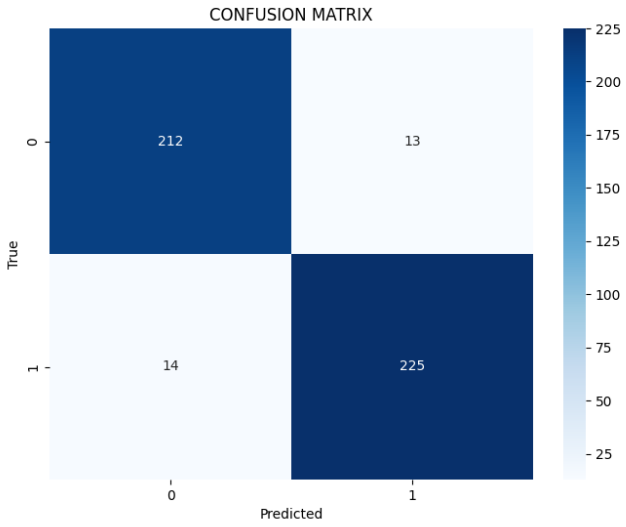
Precision: 94.56%

Recall: 94.56%

F1_Score: 94.56%



- **GCN Conv + Adam**

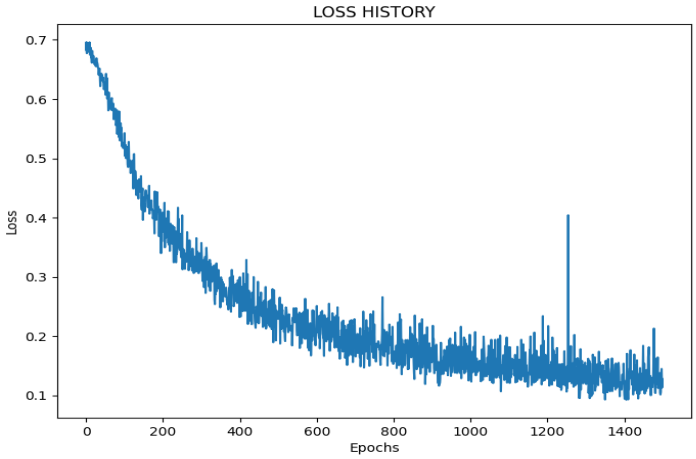


Accuracy: 94.18 %

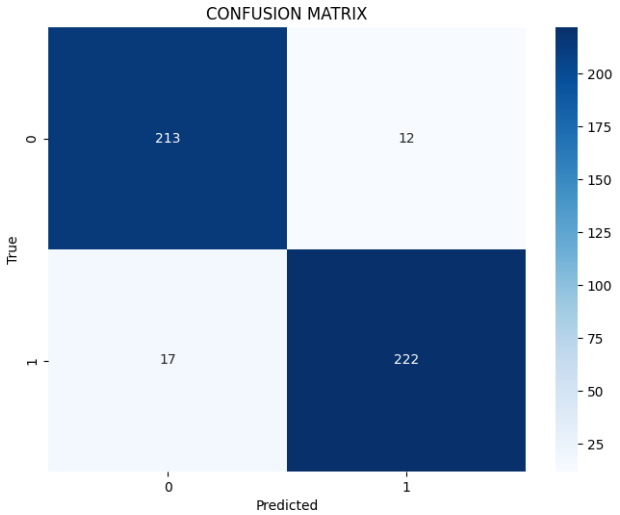
Precision: 94.54%

Recall: 94.14%

F1_Score: 94.34%



- **ChebConv + Adam**

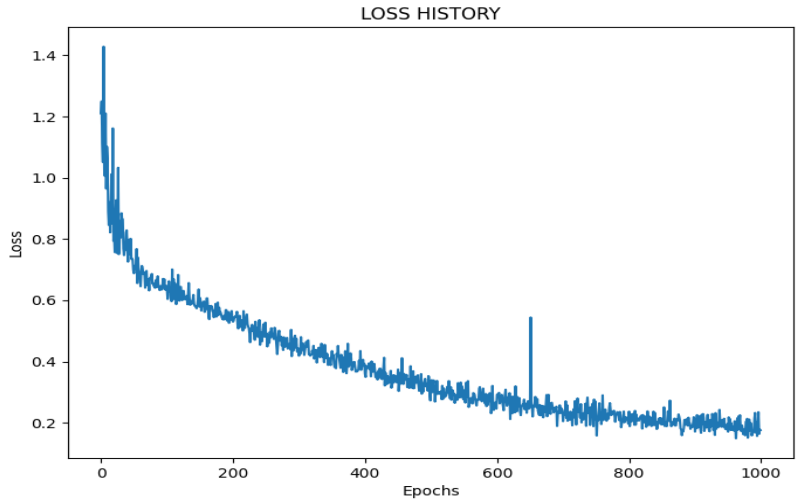


Accuracy: 93.75 %

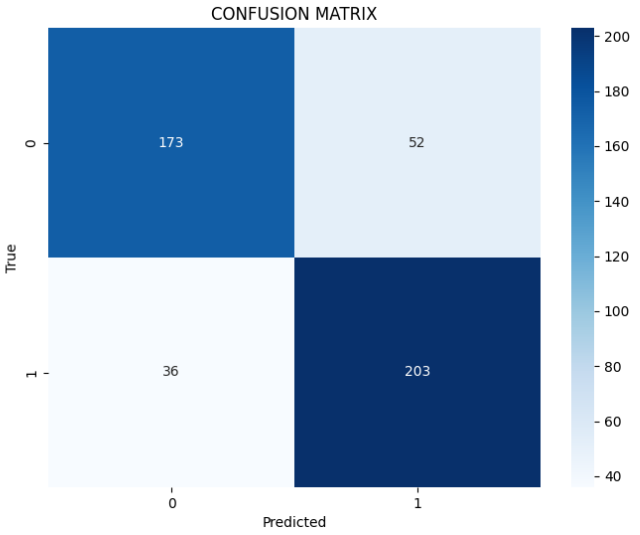
Precision: 94.87%

Recall: 92.89%

F1_Score: 93.87%



- **GCN Conv + Adagrad**

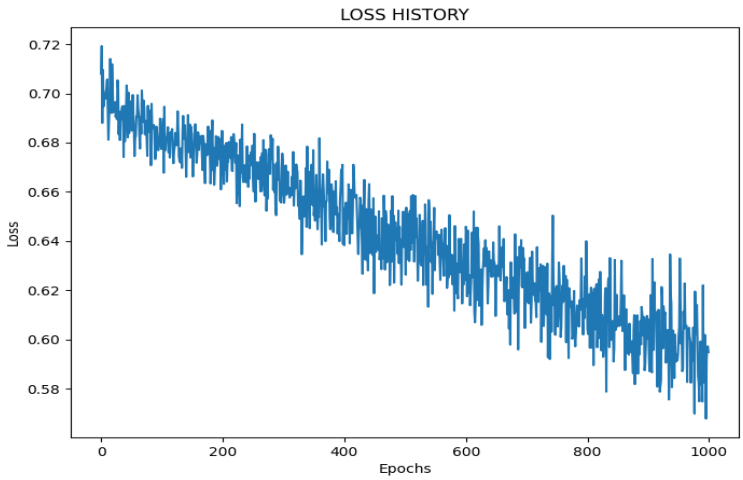


Accuracy: 81.03 %

Precision: 79.61%

Recall: 84.94%

F1_Score: 82.19%



RESULT ANALYSIS

Main differences between tested architectures (in terms of performances) are due to optimizers, indeed the models that perform worse are those using Adagrad, as it requires much higher learning rates than those available in our parameter grid due to its rapid decay. On the other hand, the models that perform better use

Adagrad or RMSProp; In terms of convolutional layers, those with GAT achieve better results with the same optimizer, but only by a few percentage points.

Not all possible combinations have been tested, but these results help to understand, in general terms, the performance of the different architectures.

CONCLUSIONS

During this work, we faced some challenges: firstly, the provided dataset contained many insignificant nodes due to associated non-functional links.

Specifically, some contained unavailable content while others redirected us: consequently, we dropped these nodes.

Next, in summarizing the chunks, we had to handle ones containing unsupported characters by the BART transformer using a try-catch mechanism.

Then, we conducted a thorough search for the right transformer for the summarization task, looking for a balance between speed and summary quality by inspecting the various output chunks.

Finally, we performed careful fine-tuning of the parameters.

Some future improvements for our work could be using much more parameters in params grid or using more epochs; we could also try other convolutional layers. activation functions and optimizers as a comparison.

Lastly, our results could be improved by using different loss functions or changing convolutional layers' number used in our architectures.

BIBLIOGRAPHY

[1] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," arXiv preprint arXiv:1609.02907, 2017.

[2] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering," in Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS), 2016, pp. 3844-3852.

[3] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," International Conference on Learning Representations (ICLR), 2018.

[4] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," arXiv preprint arXiv:1910.13461, 2019.