**A DEEP LEARNING REPORT**

**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, comprising 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 seminal paper 'Semi-Supervised Classification with Graph Convolutional Networks' by Kipf and Welling (2017), 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 paper begins by analyzing the loss function used to train the model, which is characterized by two components:

L = L₀ + λ \* Lreg

Where L₀ is the supervised loss with respect to the labeled part of the graph, Lreg is the regularization term, and λ is a weighting factor.

The key operation in GCN is graph convolution, which can be viewed as a message passing operation between adjacent nodes, like in the standard the 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:

Where:

* x'ᵢ⁽ˡ⁾: The aggregated feature representation for node i at layer l
* N(i): The set of neighbors of node i
* eⱼᵢ: The weight of the edge from node j to node i (in our case, either 1 or 2)
* dⱼ and dᵢ: The degrees of nodes j and i, respectively, including any self-loops
* hⱼ⁽ˡ⁻¹⁾: The feature vector of node j from the previous layer

The normalization factor √(dⱼdᵢ) ensures stability in the learning process by preventing the scale of features from growing or shrinking exponentially with network depth.

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

Where:

* hᵢ⁽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 for the activation layers, sigmoid for the final layer in binary classification tasks)

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

**ChebConv**:

The second model we have tested is the ChebConv model based on the use of Chebyschev polynomial:

Which is explain in great detail in the paper: ‘Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering’ by

The main idea of ChebConv 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.

)

where is computed recursively by:

Z (1) = X

Z (2) = L \* X

Z (k)​ = 2⋅L^⋅ Z(k−1) – Z^(k−2) per k≥3.​

**Graph Attention Networks (GAT):**

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

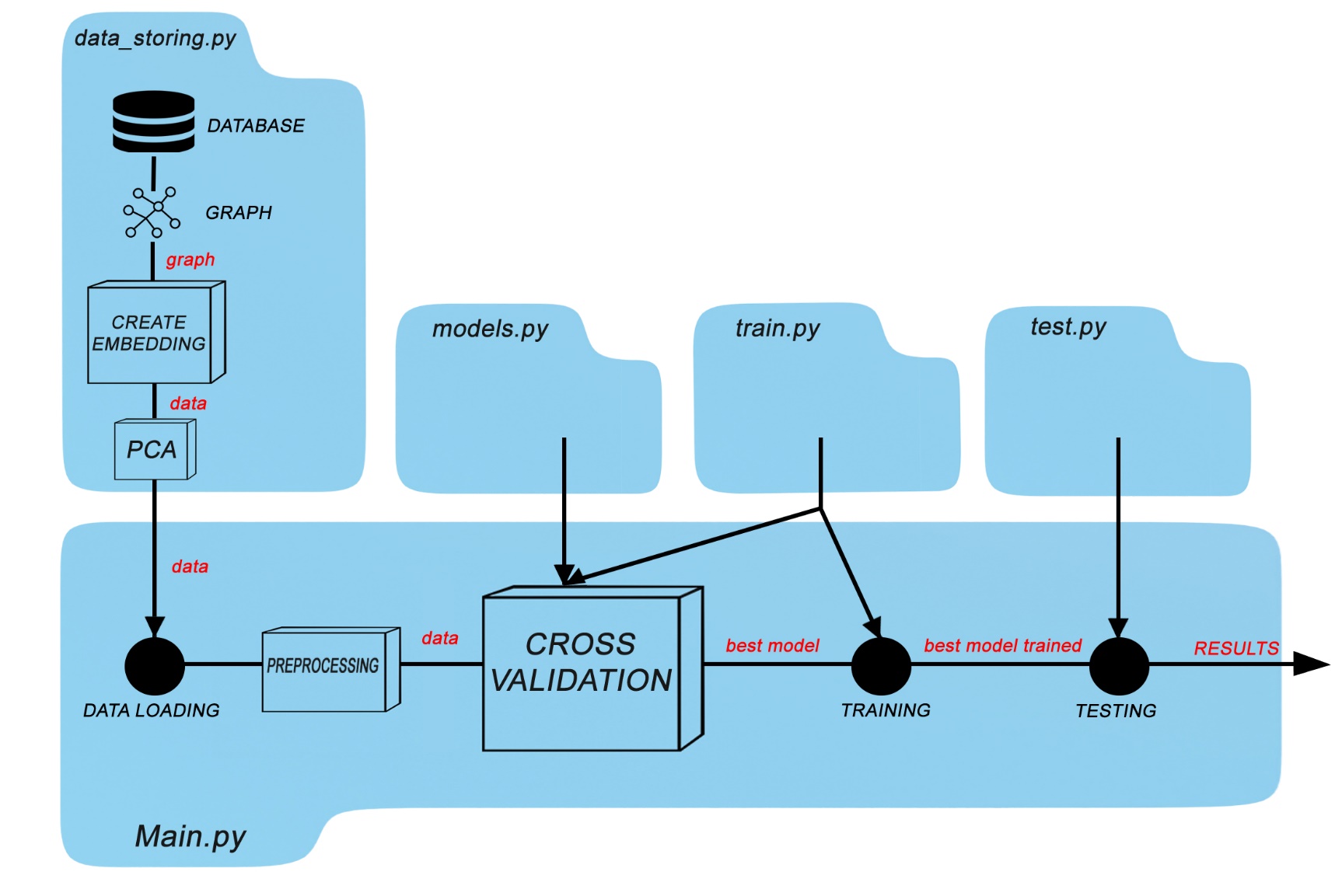
The fundamental concept of GATs is to employ an attention mechanism to assign weights to neighboring nodes, enabling the network to focus on the most relevant nodes during information aggregation. This approach addresses certain limitations of Graph Convolutional Networks (GCNs), where edge weights are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

The mathematical details of this model are thoroughly presented in the paper "Graph Attention Networks" by Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio.

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

**GRAPHICAL REPRESENTATION OF THE ENTIRE PIPELINE**

simple representation of each phase of our project in pipeline, starting from the database and arriving to the results:



**CAP 1 – DATA STORING**

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

Using network libraries in python, we 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 cardinality 2.

In the filling process, we checked nodes, deleting ones with unavailable urls starting from status codes higher than 300.

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:

Immagine che contiene testo, diagramma, schermata, Piano

Descrizione generata automaticamente

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 the paper titled: “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension.”

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 texts in input, 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.

For this task, we 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 principal component analysis to reduce the number of features, from 384 to 50 principal components, to improve the velocity in the training phase.

**PART 2 – ARCHITECTURES, CROSS-VALIDATION & TRAINING**

**PART 3 – TESTING & RESULTS EVALUATION**