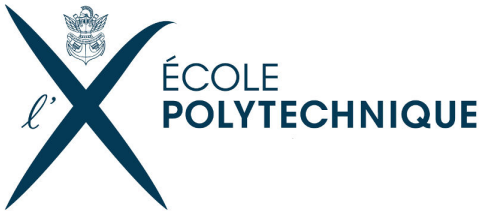


Final report for INF554 Data Challenge

Created by :
TEAM Xav, Alain & Eric :
Xavier Lekeuka
Alain Kuissu
Éric Delabrouille

Introduction

Before starting to work on binary predictors, we decided to search for other works on the subject, especially hoping to find new data sets to have a larger data bank on which we could train our model.
We found that the context of this Data Challenge was the RST-DT (Rhetorical Structure Theory - Discourse Treebank"). Even though there are many works on related topics, we didn't find any writings treating especially the task we were about to work on. Still, we searched for any exploitable additional data.
- We considered the UBC NLP Group MegaDT Data set, but its' format wasn't at all usable for our work : short extracts, which aren't a conversation, with a realised prediction incompatible with our task.
- The University of Pennsylvania's RST-DT was sadly not open-source.

We decided to split work - each of us working on a different techniques to create a predictor : if we came up with several different good predictors, we could use them together with aggregating. The different predictors we tried to create where SVM, Random Forest, Xgboost, feed forward neural network, LSTM, BERT, BERT+ LSTM and GNNs. We finally retain BERT+ LSTM, Xgboost and GNNs as they seemed more promising. Our best predictor was BERT+ LSTM which gave us the F_1-score= 0.61924 on the Kaggle Leaderbord.

TABLE .1 – Models Performance (F_1-score) Comparison on the Leaderboard

Model	BERT+LSTM	GNN	Xgboost	Random Forests
Best Performance	0.61924	0.30283	0.60498	0.58048

Feature Selection/Extraction

Bert + LSTM pipeline

The first step is a data augmentation process to handle the imbalanced dataset. In fact there is approximately a ratio of 1/4.4 between class 1 and class 0, as stated in the assignment. To mitigate this bias, we assume that by concatenating 2 texts from the class 1, we obtain a text from the class 1 again. Then we augmented the dataset using this process one time to double the size of class 1.

Then we use the BERT tokenizer for word embeddings in a 768 dimensions using a HuggingFace checkpoint from a pre-trained BERT named e5. We choose this model for its place in the MTEB Leaderboard (Massive Text Embedding Benchmark) and its size which was pretty good for our small Google Colab's GPUs (T4 of 15Go). We also tried simple BERT but a pre-trained model like e5 outperformed it for our main result. Transformer architecture suit very well for text embedding by leveraging the power of attention mechanism. That is why we choose this type of model for this task.
We use padding method to fill up all our vectors and truncation to cut if greater than 768. All embedding are normalized to avoid scaling issues. One other issue is possible spelling mistake on a dataset word. In fact, words embedding are very sensitive to spelling mistake. One method that we wanted to use was leveraging the power of big open source LLM (Large Language Model) like LLama, OPT, or Flan T5 to correct automatically the mistakes but we were stopped by memory limitations.

At the end we split our data in training, validation and test set to apply early stopping strategy and test the performance F_1-score on the test set.

GNNs Pipeline

Our first thought was that we could create a vertex binary classifier only relying on the properties of the discourse's graph. The intuition was that the 16 different possible classes for the edges and the geometry of the graph would be enough to deduce some information. Thus our first features were only the graph's adjacency matrix, and a class given for each edge. At first, we trained the model at each epoch successively on every of our $\sim$100 training graphs. After a while, fearing that this would introduce asymmetry in how our data is processed and be threatening for the model's convergence, we switched for an aggregation of the $\sim$100 training graphs into one unique large graph, with several connected component. This decision indeed gave both better convergence and computing time.

Frustrated by the low success of our model, we successively added several features : first, the speaker as a class for each vertex : perhaps that since each speaker have a different role in the meeting, the importance of their sentences

aren't the same. Then, we thought about the vertices' associated string length : even though the string's content is very significant, its' length could give a first idea of the sentence's purpose.
Then, in a desperate attempt, we tried to use for each vertex, as shown in the baseline, the transformed vector obtained using a BERT on its' string as a feature. We removed in this last attempt the feature "string length", since it would already be captured by the BERT.

Regrettably, after trying these features in this successive adding order, and some re-combinations, none of these combinations reached a F_1-score significantly higher than the naive baseline.

Alternative Approach

As stated above, coming up with a good way for pre-processing and embedding the relationships graph data with good results was challenging. To address this, we experimented with an intuitive technique for encoding the information. The idea stemmed from the observation that certain inferred relations in the graph, such as 'result' and 'elaboration,' exhibit closer proximity to important corresponding text, labeled as 1. Therefore, we constructed a dataframe that encompassed all relations as features and quantified, for each utterance, the frequency of such relations with its neighbors. This approach aimed to capture and leverage the inherent connections between specific relationships and the significance of the corresponding text, providing a meaningful representation for downstream analysis. We then aggregated that with the previously text embedding - by simple concatenation - and used the resulting training data to train several tree based models.

Model Choice, Tuning and Comparison

BERT + LSTM

Our model is a BERT with 2 last layers : 1 LSTM for its sequential processing capabilities to extract the last hidden state outputted by the BERT architecture and one linear layer as a classification head. This is the result of many architecture test and this is the more simple and efficient architecture we came up with. BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. We choose LSTM + linear layer as the classification head, to exploit both the last hidden state throw LSTM and the projection in 2 dimensions with a linear function. Then we train the model using this set of hyperparameters :

Table .2 – Hyperparameters

Hyperparameter	Values
batch_size	64
class_weight	[0.66802346, 1.98788745]
learning_rate	0.00001
weight decay	[1.2,0.0001]
epochs	10
Optimizer	AdamW
Loss function	CrossEntropy
Early stopping	Yes

We tuned the model with a wide range of hyperparameters sets and here is the best one (with weight decay = 0). There is no amelioration when touching the learning rate and by leveraging class_weight on our Cross Entropy loss we gain a higher score as the bias due to imbalance dataset is handled.

This model outperformed all the others. We can justify it by the transformer architecture and attention mechanism from bert which help on a better comprehension of text embedding in time.

Tree-based models : XGboost, Adaboost, Random Forests

The huge imbalance between the two class labels led us to opt first for several tree-based boosting and bagging algorithms including Random forests, Adaptive Boosting (AdaBoost), eXtreme Gradient Boosting X(XGboost) and also Neural Networks. The ability of boosting models to focus on the misclassified instances and learn from the

residuals makes them particularly suitable for addressing the challenges posed by imbalanced datasets. By iteratively emphasizing the errors made in previous rounds, boosting algorithms, such as XGBoost, strive to improve the classification performance on the minority class, effectively enhancing the model's ability to discern and correctly classify instances from the underrepresented class.Each of these models have several hyperparameters that may need to different results for different values. Here is how we proceeded for tuning hyperparameters :

For the tree-based algorithms we first defined ranges for each of the most important hyperparameters (learning rate, nb estimators, max depth,...). To prevent overfitting we added Lasso and Ridge regularization hyperparameters and the parameters controlling the fraction of samples used for growing trees and the fraction of features used for each tree in the model. We finally performed either GridSearch or Randomized Search by cross validation based on StratifiedKFold to assess the model's performance on different subsets of the data.

We performed the model tuning on the two different datasets : firstly, on the dataset comprising only the embedded text data, and secondly, on the dataset where the embedded text data was concatenated with the table of relationship counts. We compared the performance of the best models obtained from each approach. Remarkably, the model's performance (compared on cross validation F_1-scores) was significantly better on the dataset that combined both the embedded text and the relationship count dataframe.

GNNs

As stated before, the model used was to consider the discourse's graph, possibly adding new features, and to feed it to train a PyTorch GNN model that would then be used on the test graphs to predict their vertices' class. First, we tried using the classical PyTorch SAGEConv convolutional layers from Inductive Representation Learning on Large Graphs paper. Since it isn't able to take into account edge features, its' results weren't satisfying and we searched for a convolutional layer taking them into account.
We wanted a recent PyTorch convolutional layer, which wasn't designed in a specific purpose far from our goals. Considering the PyTorch GNN Operators, we settled for the GATv2Conv layer : first with 2 output channels representing the two classes and a Binary Cross Entropy loss. Though, since the relative gap between the outputs was very small, we switched for a unique output representing the probability of a vertex of being of class 1, with a Binary Cross Entropy loss. See the appendix for a graphical representation of the neural network in its' final state.

The first issue faced was the model's propensity to predict 0, caused by the imbalanced classes. A quick solve was weighting the classes. Testing with a weight ratio of 1/4.4, the ratio of 0s and 1s in the training dataset, made indeed our predictor predicting a ratio close of 1/4.4 in its' final predictions.

Another hyperparameter was the numbers of epochs on which the GNN was trained. This was settled by plotting the loss as a function of the epoch : seeing the elbow of the plot gave us a rough idea of the order of epochs needed, and settled for a number of epochs between 100 and 500. The learning rate was chose among the classical's 0.01, 0.001 and 0.0001 according to which one gave the best convergence.

We also realised that the GNN's predictions changed a lot from one training to another. This indicated that our model was too sensitive to some randomness introduced in the convolutional layers. As a solution, we trained more models, and had them aggregate their predictions by voting. We thus had to compromise between good convergence and computation abilities : we let our models train less (closer of "the angle of the elbow"), and settled for 75 models trained for 75 epochs - which was taking approximately 10 minutes.

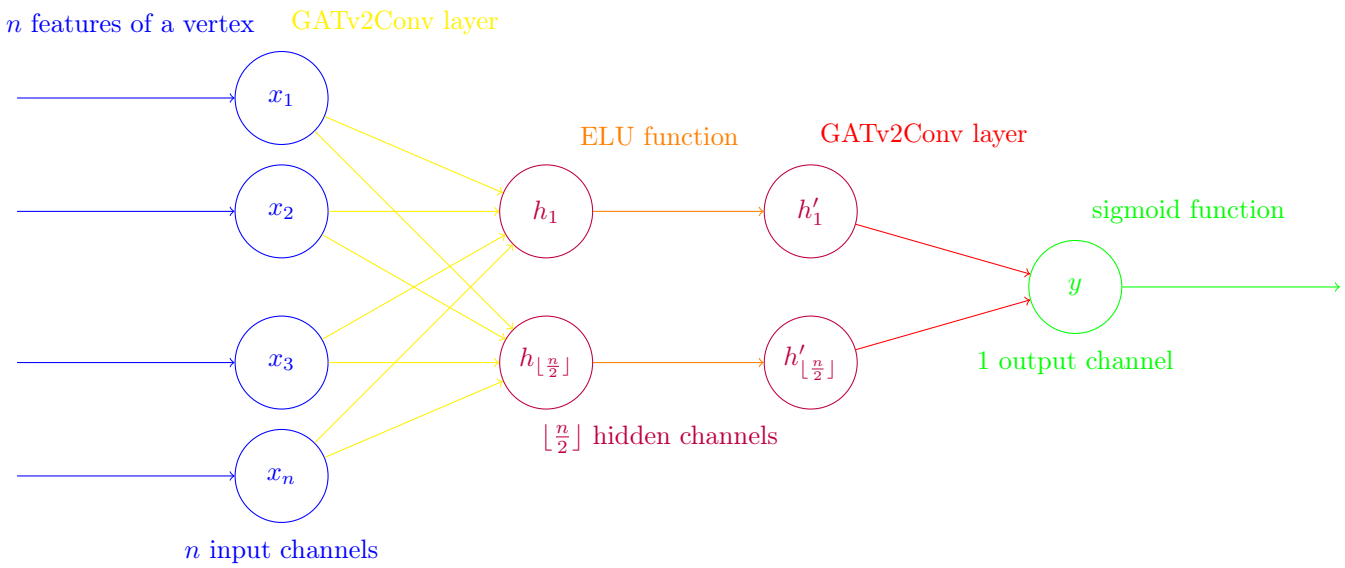
In our first attempts, overfitting wasn't an issue, because of our reasonable number of features and the size of our data (several thousands vertices). In the last attempt, using the transformed vector obtained using a BERT on its' string as a feature, introducing a random dropout at the hidden layer was the solution we selected to address overfitting risks.

Appendix

Abbreviations Summary :

BERT : Bidirectional Encoder Representations from Transformers
DT : Discourse Treebank
GAT : Graph ATTention network
GNN : Graph Neural Network
LLM : Large Language Model
LSTM : Long Short-Term Memory
MTEB : Massive Text Embedding Benchmark
NLP : Natural Language Processing
RST : Rhetorical Structure Theory
GraphSAGE : Graph Sample and AGGregation
SVM : Support Vector Machine
UBC : University of British Columbia

Simplified drawing of the last GNN implemented :



BERT+ LSTM Model Structure

```
(
  (bert_encoder): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (lstm): LSTM(768, 256, batch_first=True)
  (fc): Linear(in_features=256, out_features=2, bias=True)
)
```