Final Project Report

Extractive summarization with discourse graphs

December 2023

—

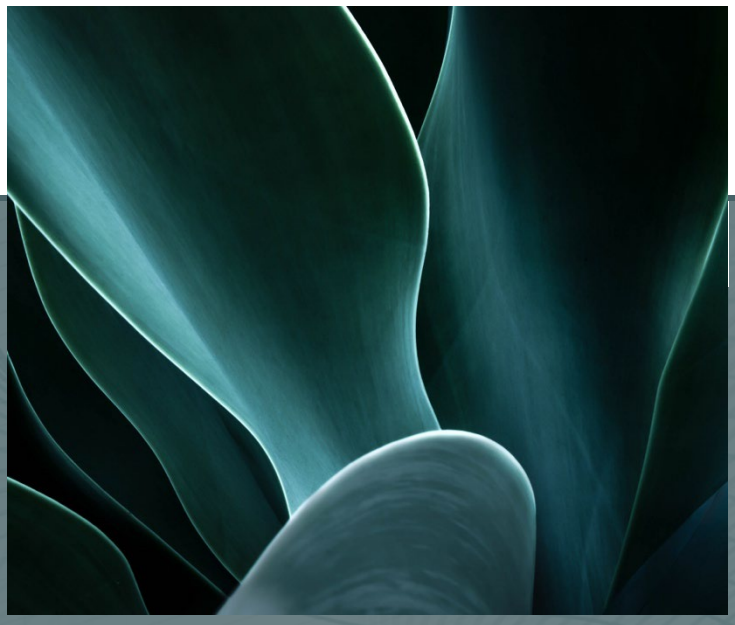
INF554 - Machine & Deep Learning

—

Kaggle name: AL YSU

Mathieu Poirée
Timothée Vinçon
Lélia Choukroun--Vînh

CONTENTS



Introduction

Summarization of textual data has become increasingly important in recent years due to the vast amount of information available in various domains. Extractive summarization, which involves the identification of key sentences or phrases from a large text corpus, has emerged as a popular technique for generating summaries that capture the essence of the original text. However, the challenge lies in producing accurate, coherent, and well-organized summaries.

Our project aims to develop an extractive summarization method based on discourse graphs constructed with data extracted from .json and .txt files, containing both the utterance's details and the relation between these utterances.

1. Feature Selection / Extraction

We approached the project through three main steps. First, we trained models without using the graph structure and by creating arrays with the selected features. Then, we implemented models trained on pure and raw graph structures built with the .json and .txt files without adding supplementary steps to the data processing. Finally, we tried to combine both of these primary approaches by creating graph structures to train the models on arrays containing the "text", "speaker", "previous sentence", and corresponding graph node features.

At the outset of our project, we utilized a basic "text feature" approach where we extracted the text from each utterance and used it to train our models. We used different encoding methods varying from embedding the whole sentence directly with pre-trained models to using Word2Vec and LSTM (Long short-term memory) models. For the first method, we used the transformer BERT available in the *sentence_transformers* package as it was made in the given file *baseline.py*. For the second method, we kept only the text and converted every word to a vector using Word2Vec and dealing with specific words like "*<vocalsound>*" or "*um*". We used the package Gensim and a pre-trained model called "glove-twitter-100" to do so.

To enhance the functionality of our system, we decided to incorporate a "speaker" feature along with the existing "text" feature. We first tried to use the embedding of the speaker labels in full (e.g., "Project manager" for "PM") using *sentence_transformers*. However, the input data size is essential, especially for machine learning models such as SVM. We had a relatively large dataset, and the computational time was quite long, even without treating the speaker separately. To solve this issue, we noticed that encoding the speaker using *sentence_transformers* was not optimal since it was then encoded in a vector of size 384, while there were only 4 possibilities for the speaker. Therefore, we decided to encode the speaker feature otherwise. The first possibility was to assign a different value to every speaker and to add this value as one extra feature of the vector previously computed using the text content. Another possibility, which is the one we decided to implement, was to use an encoder such as a OneHotEncoder to embed the speaker with a one-hot vector of size 4 containing only zeros and ones.

To further enhance the data, we also tried to add a feature corresponding to the length of the sentence to give explicitly this information to our model as we thought it might be relevant information. We also cleaned the sentences by removing useless words such as "*<vocalsound>*" or "*um*". Regarding the impact of feature selection on model performance, the previously mentioned modification resulted in marginal gains in terms of F1-score.

The second foremost step of our preprocessing phase optimization was to find a way to add some information from the relations graph. Firstly, we added the discourse relation of the edge targeting the node for every sentence, as there is always only one such relation. As for

the embedding, we used two different methods: first, by embedding the discourse relation type as a string using sentence_transformers and, secondly, using a OneHotEncoder. In the second method, the edge type is encoded to a one-hot vector of size 16 since there are 16 edge types. Our motivation here was to capture some easily accessible and valuable information from the graph structure to improve the performance of machine learning models that use this data. Concerning the impact on performance, those improvements didn't result in a significant gain in terms of F1-score. The score was even lower for some models, such as the SVM.

Our team attempted to enhance the feature extraction of the graph by introducing a "previous sentence" feature. We accomplished this by recovering the sentence index directly linked to the currently studied sentence using the discourse graph. Then, we embedded the feature using *sentence_transformers* and concatenated it with the previous embedded features. This allowed us to train the models on the resulting new vectors.

With a slightly new approach and to improve performance, we tried to capture even more information from the discourse graph. We attempted to train our models using the embeddings learned via DeepWalk on the raw graph structures, in which nodes contained both "text" and "speaker" features and edges the "discourse relations" feature between two nodes. This was done using the NetworkX library graphs. We also attempted to use the DGL library but have been unsuccessful.

Furthermore, no increase was observed when adding embeddings obtained through DeepWalk or including a feature representing sentence length.

Additionally, it was observed that removing words from sentences that were deemed unnecessary did not lead to a significant improvement in the F1 score. Despite the effort to simplify and streamline the text, the impact on the overall F1 score performance of the model was minimal.

2. Model choice, tuning & comparison

We compared various models on the same set of features. The models include Support Vector Regression, MLPs, XGBoost, Random Forest classifiers, LSTM with Word2Vec embedding, and KNN.

We approached the model selection issue through three main steps. First, we tried basic models such as a naive Bayesian and a linear regression classifier. We then tried more complex machine learning models such as SVMs and Tree classifiers. Finally, we tried deep learning models such as MLP, LSTM, and Transformers.

To prevent overfitting, we thought about several well-known techniques, such as dropout layers, data augmentation, and regularization. To accomplish this, we incorporated dropout layers, randomly dropping out some of the nodes during training to reduce reliance on any particular node. Additionally, we applied regularization, penalizing large weights in the model

to avoid over-reliance on any feature. We chose not to implement data augmentation techniques because we considered our dataset large enough. We even sometimes decided not to take the entire train set to reduce computational time. However, we could have increased the size of the dataset using data augmentation techniques such as generating similar sentences by replacing some words with synonyms, for instance.

During the tuning process, the primary objective was to determine the ideal values for the final threshold, scaling factor, learning rate, and other parameters, depending on the model we wanted to tune. The final threshold refers to the point at which the model considers an output significant. The scaling factor determines the minimum and maximum value for the input data, while the learning rate controls the speed at which the model learns. We noticed that the model converged to a local minimum for some attempts, putting every label to zero. To avoid that, we decided to increase the learning rate so the model could escape this local minimum. However, we couldn't have a significant learning rate for the entire training, so we implemented a decreasing learning rate. We also improved this feature by using a learning rate scheduler.

Let's focus on the SVM. We first tried default parameters and compared the results when changing some parameters. The first parameter that we tuned was the C parameter. Later, we noticed that the unbalanced dataset impacted the model's performance (see annex 2). We looked for a way to tackle this issue and found that a parameter called *class_weights* was precisely made for this, and we put this parameter to *class_weights = 'balanced'.*

Let's now focus on the MLP. We first tried adding dropout layers and regularizing the loss to penalize large weights. Later on, we tried to change the size of the layers, the number of layers, and the activation functions.

Concerning the LSTM, we first used a basic implementation and added dropout layers afterward. Later, we tried to keep the same input data (with every word of the sentence embedded using Word2Vec) and to change the model with a model usually used for image processing. We had a 2-dimensional array as input, so we wanted to try image processing techniques on this data. We first tried the ResNet50 model available in the package *torch.vision.* Finally, we tried to run a vision transformer (ViT) on this input data. Unfortunately, none of these models improved the previously achieved F1 scores.

Conclusion

In conclusion, our team explored and experimented with different feature extraction and model selection techniques to improve the performance of the extractive summarization with discourse graphs classifier. We trained models on graph structures and arrays with selected features and combined these approaches. We added features such as "speaker", "previous sentence", and "discourse relations" to enhance the information available to the models. However, the impact of these features on model performance varied, and some did not result in significant gains in terms of F1-score. Our findings suggest that while incorporating graph structures can help improve performance, careful feature selection and model tuning are crucial in achieving optimal results.

This project enabled us to apply and experiment with machine learning techniques covered in the course on a real-world project. We had the opportunity to implement numerous tools previously used during the lab sessions. We were also able to confront the challenges of implementing them in practice, both in choosing the model and selecting features. We may have overlooked some aspects of capturing essential information in the data and efficiently leveraging the graph, which could have further improved the F1 score. We feel that we have tried all the obvious ways to improve our performance and a lot of less obvious ones, but unfortunately, none of them proved significantly helpful.

Annexes

Model	Features	Graph implemented	F1 score
Support vector regression	Speaker, text, edge	No	**0.59**
Support vector machine	Speaker, text, edge	No	**0.59**
XGBoost Classifier	Speaker, text, edge	No	**0.56**
Neural network	Speaker, text, edge	No	**0.54**
LSTM (Word2Vec)	Speaker, text, edge	No	**0.54**
Naïve Bayes Classifier	Speaker, text, edge	No	**0.52**
Logistic regression	Speaker, text, edge	No	**0.46**
ViT	Speaker, text, edge	No	**0.46**
KNN	Speaker, text, edge	No	**0.41**
RandomForestClassifier	Speaker, text, edge	No	**0.29**
DecisionTreeClassifier	Raw graphs	Yes	**0.20**

Table 1: Recapitulative table of the models implemented and corresponding results

Table 2: Distribution of the labels in the training set