Toxic Comment Classification

Camille BRUNEAU, Roxane COHEN, Manon GEORGET

April 2022

In this project, we tackle a text classification task by trying to classify toxic comments in terms of their toxicity. We denote 7 classes : toxic, severe toxic, obscene, threat, insult, identity hate and non-toxic comments. These comments were extracted from a Kaggle dataset [1], itself created with Wikipedia comments. Based on our previous review on the article [3], we want in particular to study the effectiveness of Multi-Level Graph Neural Network (MLGNN), extended to a multi-class text classification problem. This is the core of this project : in particular, we would like to see if MLGNN scales well to multi-class classification and if its framework is well adapted to longer sequences.

In order to do so, we start by testing different baselines such as Bag-of-Words (BOW), Multi-layer perceptron (MLP) and RandomForest before diving into MLGNN. We run all our experiments using Google Colab and we especially use Scikit-Learn [5], Pytorch [4] with the Deep Graph Library (DGL) [7]. We use in particular Colab GPU for MLGNN. Then, we establish a fair comparison between all these methods, both in terms of time and space complexity, and we point out their advantages and disadvantages. Finally we conclude this work with some perspectives that remain.

Contents

1 Dataset

The Toxic Comment dataset, available on Kaggle [1], is made of comments from Wikipedia pages. Training and test sets are provided with respectively 159 571 and 153 164 comments. In this project, we only care about the available training set, from which we extract a customized dataset.

A large part of the comments are not toxic, whereas toxic comments are classified in terms of their toxicity. We have 7 classes : toxic, severe toxic, obscene, threat, insult and identity hate, plus the neutral class that denotes regular comments.

To begin with, we start by exploring the dataset. First, before any preprocessing, the class balance is the following : on 159,571 examples, 15,294 are toxic, 1,595 are severe toxic, 8,449 are obscene, 478 are threats, 7,877 are insults, 1,405 are identity hate, and then 124,473 are regular comments. This is obviously a very high unbalanced dataset, that should be taken into account for our own dataset.

Classes \ Statistics	Average length	Average number of !
Toxic	295.25 ± 617.38	3.63 ± 77.23
Severe toxic	453.64 ± 1090.99	16.28 ± 181.23
Obscene	286.78 ± 641.09	3.33 ± 59
Threat	307.74 ± 730.21	16.82 ± 223.45
Insult	277.28 ± 622.55	3.73 ± 62.80
Identity hate	308.54 ± 691.63	2.31 ± 15.35
Neutral	404.35 ± 586.50	0.34 ± 10.45

Table 1: Average length and number of ! in comments for each class.

We found comments that belong to several classes at the same time. Indeed, all the severe toxic comments also belong to the toxic class. A multi-label classification is a much more harder problem and that is why we decided here to avoid such multi-label classification. Then, we delete all the comments that belong to more than one class. These are around 10,000. Deleting these comments is not problematic, as the dataset is sufficiently large to still provide a satisfiable number of examples.

At the beginning of this project, we planned to use 100,000 comments from the Kaggle training dataset to build our own. However, after having implemented a MLP model, we realized that simply instanciating MLP's weights for 100,000 comments and their associated features was intractable as the RAM provided by Google Colab was not enough. We iteratively decrease our dataset size until all of our methods fits in memory. We finally end up with 10,000 comments, all of them having less than 100 words. This was the best dataset we could have, given our ressources. We are a bit disappointed to restrict ourselves that much. However, 100 words is still enough to see if MLGNN can be applied on texts longer than those presented in [3], with 28 words at most in average.

On these remaining 10,000 comments of at most 100 words, we notice that the comment classes are highly unbalanced with respectively (5666,0,317,22,301,54,3640) examples in these classes. The severe toxic class has no longer examples as we have removed multi-labels data points : each severe toxic comment is obviously considered as simply toxic.

Once we have created our dataset, we have to preprocess it. Some of the comments were written into arabic or chinese characters and we decided to remove them. We then limit ourself to only ascii-written comments. We delete some special caracters such as $\backslash n$, ", +, = and so on. In good old-classic NLP preprocessing, punctuation is often deleted and upper letters transformed in lower ones. However, in our case, we decided to keep these, as punctuation signs, such as ! or ? tend to be more present into toxic comments, for example in *Hey listen don't you ever!!!! Delete my edits ever again I'm annoyed because the WWE 2K15 a few of the roster have been confirmed and your stupid ass deletes what I write. just stop!!!! Please STOP!!!! [...] God your stupid.* The same hold for upper letters, for example : *COCKSUCKER BEFORE YOU PISS AROUND ON MY WORK.*

Average length and number of ! in comments per class, along with their standard deviations, are shown in Table 1. If the average length cannot, to human eyes, help to discriminate between classes, the number of ! per comment is more significant.

Finally, we end the preprocessing task by one-hot encoding the comment classes. We respectively encode the classes toxic, obscene, threat, insult, identity hate, neutral by 1, 6, 2, 3, 4, 5.

2 Methods : baselines and new approaches

In this section, we present the different methods we implemented. We would like to study the MLGNN framework, compared to more classic NLP baselines that have been successful.

We consider a BOW classifier as a first baseline. In BOW, inputs are represented as bags of their words, by vectors which do not take the word order into account. It only focuses on the multiplicity of each word of the input text dictionary. The number of occurences is used as a feature for the classifier. Given these features, we finally use a Logistic Regression classifier trained on this BOW features. BOW can be seen as a classical machine learning model with the use of features and of a logistic classifier, that is why it has a high computational efficiency. However this model is too simple, it looses many informations by ignoring the word order and it does not take into account non-linear relationships.

Second, we use a very simple MLP model. MLP is one of the simplest neural network architecture, characterised by multiple linear layers and by non-linear activation functions. Each comment should be represented by a feature vector of a given size. In this work, we used GPT2-embedding-based features.

Indeed, each word should be represented by a feature vector of a given size. In order to do so, we especially use GPT2 [6]. GP2 is a self-supervised learning algorithm. Using GPT-2 allows then to have pre-trained embeddings for each word in a sequence. However, GPT-2 uses a tokenizer which separates a sentence either in words or either in sub-words. Indeed, if a word does not belong to the pre-trained dictionnary, it is splitted into different parts, namely the sub-words. As the toxic comments are often made of unusual vocabulary, they contain many words which are divided into an unknown number of subwords and GPT-2 does not return an unique embedding for these words. To solve this problem, we decide to recover for each divided word the embeddings of its sub-words created by the tokenizer. Then we operate a sort of max pooling operation of the subwords embeddings. This maximum is considered as the unique word embedding, of size 768.

Because all sentences do not have the same number of words, we may end up with a ragged embedding matrix. We then pad it when necessary, in the MLP case.

Third, we test a RandomForest classifier. The RandomForest classifier is characterized by the use of many decision trees, where each decision tree represents the class labels thanks to its leaves. RandomForest combines the results of the decision trees classifiers on different samples of the dataset to obtain the classification results. We re-used the GPT2-based feature matrix used for MLP. However the creation of a large number of trees and the combination of their results may require a lot of computational power and resources.

Furthermore, we wanted to implement LSTM. LSTM is a a variant of recurrent neural network (RNN) which keeps more long-term global informations in memory and which mainly focuses on consecutive word sequences. They were during a long time state-of-the art methods for NLP tasks. We implemented a LSTM but had to renounce to it due to RAM limitations on Colab.

Finally, we test the main part of the project, the MLGNN framework, presented in [3]. As explained in our previous article review, MLGNN is a Graph Neural Network (GNN) method which differs from other GNNs by using three edge types, one for each sentence level. These three different analysis levels allow to fuse local features with global ones in order to obtain better text representations. MLGNN also requires embeddings : we represent each word by a single GPT2-embeddings, by merging the subwords embeddings, as before.

Notice we did not exactly implement the MLGNN architecture : indeed, motivated by simplicity and usual GNN definitions, we operated several changes in our MLGNN model, by replacing the bottom convolution by a GIN convolution, which is nearly the same and where the parameter β can be learn instead of fixed, and by using a max pooling, instead of a concatenation pooling, which is rarely used due to padding issues when dealing with graphs with different number of nodes.

3 Experimental setup

In the previous section, we have presented the methods we are going to test. We would like to fairly compare them, with a relevant experimental setup.

For each method, we follow the same process, repeated 3 times with a fixed seed :

1. We split our training dataset into three parts : the training set where the classifier is trained, the validation set which is used for the hyperparameters tuning and finally the testing set in which we evaluate the classification results. These sets are equal for all the methods, even if the preprocessing is different.

2. We use balanced class weights so that our algorithm can deal with training examples in small quantities.

3. We perform a hyperparameter gridsearch. In the validation set, we select the ones with the best validation score.

4. We assess the model's performances in the testing test using different metrics : the accuracy, the f1-score and the ROC-AUC metric. Indeed, because our customized training set is highly unbalanced, the accuracy only is not a trustable metric and a good estimate of our model performances. That is why we are also interested in the multi-class f1-score and AUC.

F1-score finds an equal balance between precision and recall so it is often used to evaluate the performance of multi-class classification problem with unbalanced data.

$$\textit{f1-score} = \frac{TP}{TP + \frac{FP+FN}{2}}$$

where TP denotes True Positives, FP False Positives and FN False Negatives.

AUC measures the classification performances using all possible classification thresholds. It uses the ROC curve to compute the probability that a random positive example is ranked more highly than a random negative example. As f1-score, it is sensitive to unbalanced classes.

All the hyperparameters that were tested, details about train-validation-test splits, models and so on, are available in the corresponding notebooks.

4 Results

Results, obtained following the process explained in the previous section, are shown in Table 2.

Methods \ Metrics	Test accuracy	Test f1-score	Test AUC
BOW	0.761 ± 0.007	$\mathbf{0.753 \pm 0.007}$	$\mathbf{0.765 \pm 0.0157}$
MLP	0.77 ± 0.015	0.755 ± 0.017	0.703 ± 0.012
RandomForest	0.712 ± 0.0014	0.690 ± 0.0028	0.684 ± 0.0148
MLGNN	$\mathbf{0.801 \pm 0.007}$	$\mathbf{0.779 \pm 0.005}$	0.717 ± 0.018

Table 2: Best test accuracy, f1-score and AUC for different models, on 3 runs and with hyperparameter search.

To begin with, all the tested methods perform well on this problem, as the smallest test accuracy score is around 0.712. Recall this is a 6-class text classification so a random classifier would lead to 0.167 in accuracy approximately. Clearly, whether it is the embeddings from GPT2 or the frequency features, they well capture semantic and structural informations from the comments. BOW achieves very good performances given it only relies on frequencies so this tends to show this toxic comment classification is actually not so difficult. This may be explained by the fact we only keep short comments, with less than 150 words.

For all the tested methods, the accuracy is higher than the f1-score, which is itself higher than the AUC. This behavior was expected due to the highly unbalanced dataset.

The RandomForest classifier is the least effective classifier, by a large margin, with 0.71 in accuracy, f1-score of 0.69 and AUC of 0.684. RandomForest usually are known to be more interpretable than deep-learning models by their ability to tell what features were the most important in the final decision. However, because features are extracted from GPT2 embeddings, it is difficult to tell what these features where or why, or to investigate why a RandomForest gives lower result than a MLP, on the same feature matrix.

The MLP, despite its simplicity (one hidden layer but with a lot of neurons, plus dropout and batch normalisation), achieves 0.77 in accuracy, f1-score of 0.755 and AUC score of 0.703. Surprisingly, it is only in third place in the method ranking, as it was beaten by BOW, with 0.78 accuracy, 0.78 f1-score and 0.775 AUC. The margin is not so large but is worth to be mentionned. In fact, BOW tends to be a sort of naive method, with a frequency feature vector for each comment, while our MLP uses features extracted from GPT2 embeddings. This means the Logistic Regression classifier combined with frequency vectors better represents comments than a MLP with GPT2 embeddings. To the best of our knowledge, this should be due to the way we extract features from GPT2 embeddings : recall we merge the embeddings subwords of a word with a max pooling in order to obtain a unique embedding. By doing so, we must loose key informations in the process.

It is difficult to tell which method should arrive in first place. Indeed, if the MLGNN has a better accuracy, its AUC score is significantly worse than the BOW score. Furthermore, the f1-score difference is not significant. One may note the MLGNN model that achieves such result was applied without class weights. Indeed, surprisingly, using class weight as done for all the other methods, highly decreases MLGNN performances. This is unexpected and despite a significant debugging time, no explanation could be found. That means MLGNN could, in theory, largely beat BOW, with proper class weights.

Last but no least, we finally showed what we wanted : given word embeddings, MLGNN can outperform other baselines that use the same embeddings, such as MLP. Notice that, for computational reasons, we only train our MLGNN models for 20 epochs, which is relatively low. A longer training and hyperparameter search, with more longer examples, would make MLGNN even better.

All of this tend to show MLGNN is the best of our methods.

Time and space complexity measures are displayed in Table 3. Brielfy, we observe that the MLP is the quickest method, with only 2 minutes approximately to run a small hyperparameter search. RandomForest and BOW are longer due to the scikit-learn pipeline, with operates a stratified k-folds plus a hyperparameter search. The MLGNN is computationally intensive, due to the high algorithm hyperparameter number and the length of the dataset. Because we use sparse features for BOW, it has the lowest RAM consumption. MLP and RandomForest use the same feature matrix, made of GPT2-embeddings, where a word embedding is of size 768 and padded to the maximum number of words.

Methods \ Complexity	Time (minutes)	Space (GB)
BOW	17.4	2.25
MLP	2.04	11.42
RandomForest	268.44	10.15
MLGNN	367.07	4.48 (9.45 GPU)

Table 3: Time and space complexity for the different models. Time complexity includes the hyperparameter search displayed in the notebooks.

Dataset comments have 100 words at most. Consequently, the feature matrix is heavy to load for both RandomForest and MLP and explains their RAM consumption. On the contrary, MLGNN takes advantage of its graph-based representation and does not pad embeddings.

5 Perspectives and conclusion

This project rises several perspectives that are left for future work.

First, we were considerably limited by the computational ressources at our disposal. Indeed, larger RAM and GPU would have been useful to extend our small-scale experiments to a larger setup. In particulary, it would have helped us to use LSTMs for our text classification task, what we did not do, due to this RAM limitation. We also wanted to test the impact of the p and q parameters on the MLGNN performances, depending on different datasets, with larger words per comment.

Second, because we need an unique embedding per word, we decided to operate a max pooling operation of subword embeddings. However, this first approach seems flawed and more clever representations should be obtained.

Third, Transformers have achieved state-of-the-art performances on NLP tasks and inspired other domains. The first Graph Transformer has been proposed in [2]. This could be interesting to use semantic informations from transformer-based GPT2 embeddings with structural informations, extracted with such Graph Transformers. Moreover, one could think to leverage and fine-tune the MLGNN framework, by adding skip connections in order to prevent the oversmoothing effect for example.

Finally, in the first stage of this project, we wanted to apply LSTM along with the GPT2 embeddings. Then, we questionned the idea's relevance : indeed, GPT2 used Transformers, which often outperform LSTM. In that case, does it still make sense to use a LSTM on these embeddings ? As we did not have the computational ressources to check this hypothesis, it would be interesting to test it in practice, with a larger setup.

To conclude, we experienced our first NLP project in this course. We faced a text classification problem, with a toxic comment dataset. We tested several methods, ranging from a good old-classic BOW trained with a Logistic Classifier to a MLGNN model. In particular, we extend the MLGNN framework to a multiclass text classification and implemented it from scratch, as no code were provided. This project was demanding and time-consuming but helped us to really dive into NLP. This encouraged us to apply for NLP-based internships.

References

[1] Kaggle toxic comment dataset. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge. Accessed: 2010-09-30.

[2] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs, 2020.

[3] Wenxiong Liao, Bi Zeng, Jianqi Liu, Pengfei Wei, Xiaochun Cheng, and Weiwen Zhang. Multi-level graph neural network for text sentiment analysis. *Computers & Electrical Engineering*, 92:107096, 2021.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[5] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[6] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[7] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. 2019.