Toxic Comment Classification

Natural Language Processing Project Report

Alberto Paparella

1 Introduction

Toxic comment classification is a problem in natural language processing (NLP) that involves identifying and categorizing toxic or abusive comments in online platforms such as social media, forums, or comment sections. The goal is to develop machine learning models that can automatically detect and flag such comments, helping to promote healthier and more respectful online conversations. Toxic comments can include a range of harmful behaviors, such as hate speech, offensive language, personal attacks, threats, or any form of harassment. These comments not only create a negative environment for users but also have the potential to cause emotional distress and perpetuate online bullying or discrimination. Machine learning models for toxic comment classification often utilize various NLP techniques and algorithms, such as recurrent neural networks (RNNs), convolutional neural networks (CNNs), or transformer models like BERT or GPT. These models can capture semantic and contextual information from the text, allowing them to identify toxic patterns, offensive language, and abusive behavior. To build effective models, it is crucial to have high-quality labeled datasets that accurately represent different types of toxic comments, and to adopt data preprocessing techniques to clean the text before training the models. The toxic comment classification problem has gained significant attention due to the increasing need for maintaining a safe and inclusive online environment, helping to automate the identification and moderation of toxic content and fostering healthier online interactions. This report is meant to discuss the build of a multi-headed model that's capable of detecting different types of toxicity. In order to do so, different technologies have been evaluated and compared on the mean column-wise ROC AUC score. The designed model predicts a probability of each type of toxicity for each comment in the test set.

2 Dataset

The dataset is comprised of a large number of comments from Wikipedia's talk page edits which have been labeled by human raters for toxic behavior. The types of toxicity are: *toxic, severe toxic, obscene, threat, insult,* and *identity hate*.

The dataset is given through the following files:

- train.csv - the training set, contains comments with their binary labels
- test.csv - the test set, the model must predict the toxicity probabilities for these comments
- test_labels.csv - labels for the test data; value of -1 indicates it was not used for scoring

The dataset is provided under CC0, with the underlying comment text being governed by Wikipedia's CC-SA-3.0

3 Data analysis

The comments are labelled as one or more of the following six categories: *toxic, severe toxic, obscene, threat, insult* and *identity hate*. The training data contains a row per comment, with an *id*, the *text of the comment*, and 6 different labels, which are the target of the model predictions. All rows in the training dataset don't contain *null* values; specifically, they all contain comments, removing the necessity to clean up null fields. Tab.1 serves as an example, while Tab.2 provides a summary of the train dataset. The mean values are very small (some way below 0.05), as 89.8321% of the comments are not labelled in any of the six categories and therefore not considered toxic. Specifically, the train dataset contains:

- 159571 total rows
- 143346 unlabelled (positive) comments
- 15294 comments labeled as toxic
- 1595 comments labeled as severe toxic
- 8449 comments labeled as obscene
- 478 comments labeled as threat
- 7877 comments labeled as insult
- 1405 comments labeled as identity hate

Regarding the character length for the rows in the training data, the length of the comments varies a lot, as shown by the histogram plot in Fig.1, with a mean of 394.07322 and a standard deviation of 590.72028. Most of the text length are within 500 characters, with some up to 5,000 characters long. Fig. 2 represents the distribution of the number of words in the sentences. Finally, Fig.3 represents the correlations among the target variables, showing how some of the labels are higher correlated, such as insult-obscene (0.74), toxic-obscene (0.68) and toxic-insult (0.65).

id	text	toxic	severe_toxic	obscene	threat	insult	identity_hate
b03...57f	...	1	0	1	0	1	0
50a...9b6	...	0	0	0	0	0	0
e2f...949	...	1	0	1	0	0	0
908...41c	...	1	1	0	0	0	0
368...c81	...	1	0	0	0	0	0

Table 1: Example of a subset of the training data; the text of the comments has been purposely left out.

	toxic	sev_toxic	obscene	threat	insult	idnt_hate	none
count	159571	159571	159571	159571	159571	159571	159571
mean	0.095844	0.009996	0.052948	0.002996	0.049364	0.008805	0.898321
std	0.294379	0.099477	0.223931	0.054650	0.216627	0.093420	0.302226
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Table 2: Summary of the train dataset; the *none* label has been used to keep track of how many comments have no labels.

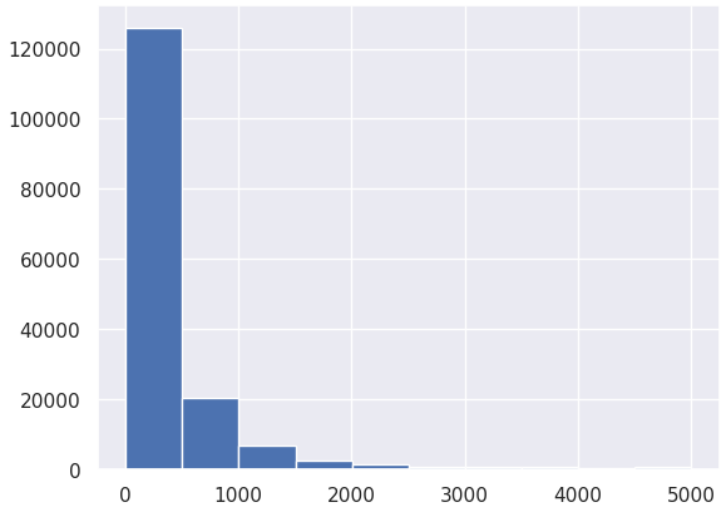

Figure 1: Histogram plot for text length of the comments in the training data.

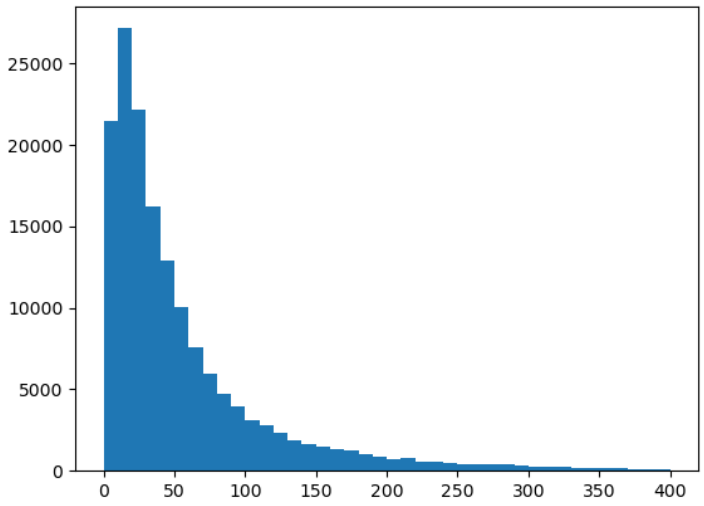

Figure 2: Distribution of the number of words in the sentences.

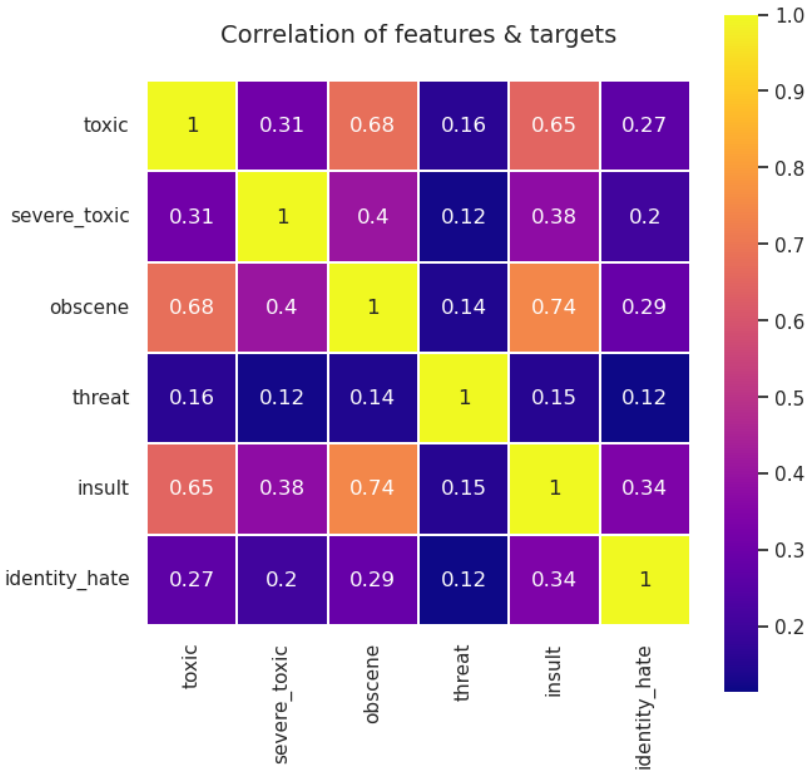

Figure 3: Correlations among the target variables.

4 Data pre-processing

Data pre-processing is carried out in a similar, yet slightly different way for the models, especially regarding vectorization. This is because each approach proved to work better with a specific technique but worse with the others, indicating that each model requires specific pre-processing decisions to work efficiently. Preprocessing required the following steps:

1. **Data cleaning** - First of all, all text has been converted to lower case, all short form verbs such as *can't, i'm, 've, 're, 'd* have been converted to their full form, such as *cannot, i am, are, would*, except for *'s* which could assume various meanings (genitive, to be verbe), and is therefore removed. Regex symbols, such as $\backslash s$ or $\backslash W$, have also been removed. As noted in the previous section, the datasets don't contain null values, so there's no need to take care of them.

2. **Tokenization** - The sentences are broken down into unique words (e.g., "We must turn right and then turn left" becomes ["We", "must", "turn", "right", "and", "then", "turn", "left"]).

3. **Indexing** - The words are in a dictionary-like structure and given them an index (e.g., 1: "We", 2: "must", 3: "turn", 4: "right", 5: "and", 6: "then", 7: "left").

4. **Index Representation** - The sequence of words in the comments are represented in the form of indexes (e.g., [1, 2, 3, 4, 5, 6, 3, 7].

5. **Padding** - The shorter sentences are made as long as the others by filling the shortfall by zeros and the longer ones are trimmed to the same length (*maxlen*) as the short ones; a smaller *maxlen* could result in the loss of some useful feature, while a greater *maxlen* requires the LSTM cells to be larger to store the possible values or states.

Regarding Machine Learning techniques, the comments have been vectorized creating a bag of words representation as a term document matrix. This has been done through the use of a **TfidVectorizer**, provided by the **sci-kit learn** library, which also includes a tokenizer, and which is used to learn the vocabulary in the training data and to create a document-term matrix accordingly for both datasets. This allows for the creation of a sparse matrix with only a small number of non-zero elements. For the **LSTM**, data is preprocessed using the **Tokenizer** provided by the **Keras** API, choosing a *maxlen* of 300 and 20000 unique words for the dictionary, while for **BERT** fine tuning the **BertTokenizer** provided by the **HuggingFace trandformers** library has been used.

5 Proposed models

5.1 Logistic regression

One way to approach a multi-label classification problem is to transform the problem into separate single-class classifier problems. This is refered to as **problem transformation**[1], and it can be done in multiple ways:

- **Binary Relevance[2].** This is probably the simplest approach, treating each label as a separate single classification problem. The key assumption is that there are no correlations among the various labels.
- **Classifier Chains[3].** In this method, the first classifier is trained on the input X. Then the subsequent classifiers are trained on the input X and all previous classifiers' predictions in the chain. This method attempts to draw the signals from the correlation among preceding target variables.
- **Label Powerset.** This method transforms the problem into a multi-class problem where the multi-class labels are essentially all the unique label combinations. In this case, where there are six labels, Label Powerset would turn this into a 2^6 or 64-class problem.

For the realization of this project, only the Binary Relevance and the Classifier Chains approaches have been considered, solving each single-class classification problem using **logistic regression**[4]. Logistic regression is a supervised learning algorithm for binary classification which makes use of the logistic function (*sigmoid*) to map the input into a value between 0 and 1 representing the probability of the positive event to occur. The logistic function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Where x is a linear combination of the input variables weighted on the coefficients of the model. During the training phase, the logistic regressor uses the maximum likelihood method to estimate the coefficients which maximize the probability to obseve the relative label, minimizing a *log-loss* function. Once trained, the logistic regressor can be used to make predictions on new input data, giving as output a probability which can be mapped on a binary class given a threshold.

5.2 Naive Bayes - Logistic Regression

Naive Bayes is a classification algorithm based on the **Bayes Theorem**, asserting that *the probability of an hypothesis given the evidence can be computed using the probability of the evidence given the hypothesis, the probability of the hypothesis and the probability of the evidence.* It is refered to as *naive* as it assumes all input variables to be indipendent among each others, simplifying the conditional probabilities computation as the joint probability of the features

can be approximated by the product of the probabilities of the individual features. In the context of classification, the Naive Bayes algorithm computes the probability of an instance to belong to a specific class given the observation of its features as it follows:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}$$

Where $P(C|X)$ is the probability of an instance to belong to the class C given the set of features X, $P(X|C)$ is the probability to observe the set of features X given that the instance belog to the class C, $P(C)$ is the a priori probability of the class C and $P(X)$ is the probability of the set of features X. [5] proposes an interesting improvement of this method, named **Naive Bayes - Support Vector Machine**, combining it with a Support Vector Machine to solve a similar problem (Sentiment Analysis). Taking inspiration from this approach, a similar version has been implemented making use of logistic regression instead of a Support Vector Machine, which as stated in the paper gives similar results.

5.3 Long short-term memory (LSTM)

Long Short-Term Memory (**LSTM**)[6] is a type of **recurrent neural network** (**RNN**) architecture that is designed to handle sequential data, like text. The key idea behind LSTM is its ability to capture long-term dependencies and remember information over extended sequences. Traditional RNNs suffer from the *vanishing gradient problem*, where the gradients diminish exponentially as they propagate back through time, making it difficult for the network to learn and retain information from distant past time steps. LSTM overcomes this problem by incorporating a **memory cell** responsible for storing and propagating information across time steps. This cell consists of three main components:

- The **input gate** determines how much of the new input should be stored in the memory cell. It takes the current input and the previous hidden state as inputs, passes them through a *sigmoid* activation function, and produces an output between 0 (ignored) and 1 (fully stored).
- The **forget gate** decides what information should be discarded from the memory cell. It takes the current input and the previous hidden state as inputs, passes them through a *sigmoid* activation function, and generates a forget vector. This vector determines which parts of the previous memory cell state should be forgotten.
- The **output gate** controls how much of the memory cell state should be exposed as the output of the LSTM cell. It takes the current input and the previous hidden state as inputs, passes them through a sigmoid activation function, and also computes the cell state after passing it through a *tanh* activation function. The output gate then combines the cell state with the output of the *sigmoid* function to produce the final output of the cell.

The combination of these gates allows LSTM to learn when to remember or forget information, and when to produce an output. By using the gradient flow through the gates, LSTM can effectively propagate gradients across long sequences, allowing for the capture of long-term dependencies.
The proposed architecture has the following structure:

1. An **Input layer** which takes as input a sequence of *maxlen* words, with *maxlen* accordingly to data pre-processing ($maxlen = 300$).

2. An **Embedding layer** which projects the words to a defined vector space depending on the distance of the surrounding words in a sentence, allowing to reduce the model size. The output of the Embedding layer is just a list of the coordinates of the words in this vector space, e.g., (-81.012) for *right* and (-80.012) for *left*. The embedding layer outputs a 3D tensor of $(None, maxlen, embed\_size)$, representing an array of sentences, where for each word $(maxlen)$ in the sentence there is an array of $embed\_size$ coordinates in the vector space of the embedding.

3. A **LSTM layer**, set to produce an output that has a dimension of 60 and to return the whole unrolled sequence of results. LSTM works by recursively feeding the output of a previous network into the input of the current network, taking the final output after X number of recursions; however, some cases, as this one, require to take the unrolled sequence (i.e., the outputs of each recursion) to pass as result to the next layer. LSTM takes in a tensor of $[BatchSize, TimeSteps, NumberofInputs]$, where $TimeSteps$ is the number of recursions it runs for each input. Specifically, it goes through the samples, recursively running the LSTM model for $maxlen$ times, passing in the coordinates of the words each time, and receiving a tensor with shape of $(None, maxlen, 60)$, where 60 is the output dimension previously defined.

4. A **Global Max Pooling layer**, which is traditionally used in CNN problems to reduce the dimensionality of image data. It goes through each patch of data, taking the maximum values of each patch. These collection of maximum values will be a new set of down-sized data to be used. This is because before passing the output to a normal layer, the 3D tensor has to be carefully reshaped into a 2D one, to avoid throwing away relevant data. Resulting data should be a good representative of the original data.

5. A **Dropout layer**, which indiscriminately *disable* some (10%) nodes so that the nodes in the next layer is forced to handle the representation of the missing data and the whole network could result in better generalization.

6. A **Dense layer** of 50 neurons with $RELU$ as activation function.

7. Another **Dropour layer** with a dropout rate of 0.1.

8. A **Sigmoid layer**, achieving a binary classification(1,0) for each of the 6 labels (the *sigmoid* function will squash the output between the bounds of 0 and 1).

5.4 BERT fine tuning

BERT[7], which stands for **Bidirectional Encoder Representations from Transformers**, is a transformer-based natural language processing model. Unlike traditional language models that operate in a left-to-right or right-to-left manner, it utilizes a bidirectional approach by considering both the left and right context of a word during training, enabling to better capture the meaning and dependencies of words in a sentence. BERT is initially pretrained on a large corpus of unlabeled text from sources like books, articles, and the internet. During pretraining, the model learns to generate word embeddings that capture rich semantic representations of words based on their context. This is achieved employing the *masked language modeling* technique, which involves randomly masking a certain percentage of words in each training sentence and training the model to predict the masked words, learning to understand the contextual relationships between words. It utilizes the emphtransformer architecture, which consists of stacked *self-attention* and *feed-forward layers*. Self-attention allows the model to weigh the importance of different words within a sentence based on their contextual relevance, helping BERT to capture long-range dependencies and understand the relationships between words more effectively. After pretraining, BERT can be fine-tuned on specific downstream tasks such as *text classification, named entity recognition, question-answering, sentiment analysis*, and more. During fine-tuning, BERT is further trained on labeled data specific to the target task. The pretrained BERT model acts as a feature extractor, and additional task-specific layers are added on top to make predictions. BERT produces contextual word embeddings, which means that the word representations vary based on the surrounding words in a sentence, allowing to better capture word meaning and disambiguate words with multiple interpretations. The contextual embeddings generated by BERT have been shown to improve performance on downstream NLP tasks significantly. BERT has been trained on data from various languages, enabling it to provide strong multilingual capabilities. This allows the model to transfer knowledge learned from one language to another, even when labeled data is scarce.
For this project, BERT has been fine-tuned in the following way: once locaded the pretrained BERT base-model, the first hidden-state from BERT output is fed into a **Dense layer** with 6 neurons and *sigmoid* activation (Classifier); the outputs of this layer can be interpreted as probabilities for each of the 6 classes.

6 Experiments

6.1 Experimental setup

For the logistic regression, the **sci-kit learn** python library has been used, which makes use of the **lbfgs (Large-scale Bound-constrained Optimization)**[8] algorithm, using 500 maximum iterations. Regarding the Long Short-Term Memory recurrent network, **Tensorflow** and its relative **Keras** API have been used, while the fine-tuning of BERT made use of the **HuggingFace** library.

LSTM has been trained with a *batch size* of 32 for 20 *epochs* using *BinaryCrossentropy* as loss function, calculated for each of the output 6 output neurons, and *Adam* as optimizer, also providing a callback function which halves the learning rate if the loss didn't decrease for 2 epochs, strarting from a *learning rate* of $1e^{-3}$. BERT has been trained only for 1 *epoch* also using *BinaryCrossentropy* as loss function calculated for each of the output 6 output neurons, while *AdamW* is used as optimizer with 1-cycle-policy from the Transformers library. For evaluation metrics, both **sci-kit learn** and **PyTorch** have been used for a more reliable comparison. Experiments on machine learning algorithms (i.e., logistic regression and naive bayes) have been executed on a Intel Core i5-8250U Processor, 6M Cache, up to 3.40 GHz, 4 cores, 8 threads, while experiments on deep learning techniques (i.e., recurrent networks and transformers) have been carried out making use of Kaggle's online notebook service, using 2 NVIDIA Tesla T4 GPUs with 16 GB of VRAM each. For the evaluation, the mean column-wise ROC AUC score has been used. ALL rows with -1 values in **test_labels** file have to be removed from the test dataset, as they are not used for evaluation.

6.2 Evaluation metrics

Models are evaluated on the mean column-wise **Receiver Operating Characteristic Area Under the Curve** (**ROC AUC**)[9], i.e., the average of the individual AUCs of each predicted column, a widely used performance metric for binary classification models. It is calculated by computing the area under the **ROC curve**, a graphical representation of a classifier's performance at various classification thresholds plotting the **recall** (i.e., the proportion of actual positive samples that are correctly classified as positive), against the **specificity** (the proportion of actual negative samples that are incorrectly classified as positive), and therefore providing a visual representation of a classifier's trade-off between sensitivity and specificity, where the best scenario is the one which maximizes the first and minimizes the latter. It ranges between 0 and 1, where a score of 1 represents a perfect classifier, while a score of 0.5 indicates a random classifier that performs no better than chance. A higher ROC AUC score implies better discrimination ability of the model. ROC AUC is particularly useful when dealing with imbalanced datasets, where the number of samples in the positive class differs significantly from the negative class, as in the toxic classification task. In such cases, **accuracy** alone might not be a reliable metric since a classifier could achieve high accuracy by simply predicting the majority class, while ROC AUC provides a more comprehensive evaluation of a model's performance, considering both false positives and false negatives. It's important to note that ROC AUC is not affected by the classification threshold, unlike other metrics such as accuracy, precision, and recall, providing a more holistic view of the model's performance across various threshold settings.

	toxic	sev_toxic	obscene	threat	insult	idnt_hate	AVG
LR - BR	0.94274	0.97374	0.95560	0.97340	0.94475	0.95762	0.95798
LR - CC	0.94274	0.97438	0.94707	0.97152	0.93268	0.94787	0.95271
NB - LR	0.94635	0.97892	0.96130	0.97502	0.95168	0.96340	0.96278
LSTM	0.95015	0.97942	0.96703	0.97412	0.95913	0.96391	0.96563
BERT	0.97230	0.98966	0.98089	0.98089	0.99434	0.97898	0.98960

Table 3: ROC AUC score for Linear Regression - Binary Relevance (LR - BR), Linear Regression - Classifier Chains (LR - CC), Naive Bayes - Logistic Regression (NB - LR), Long Short-Term Memory (LSTM), BERT fine tuning.

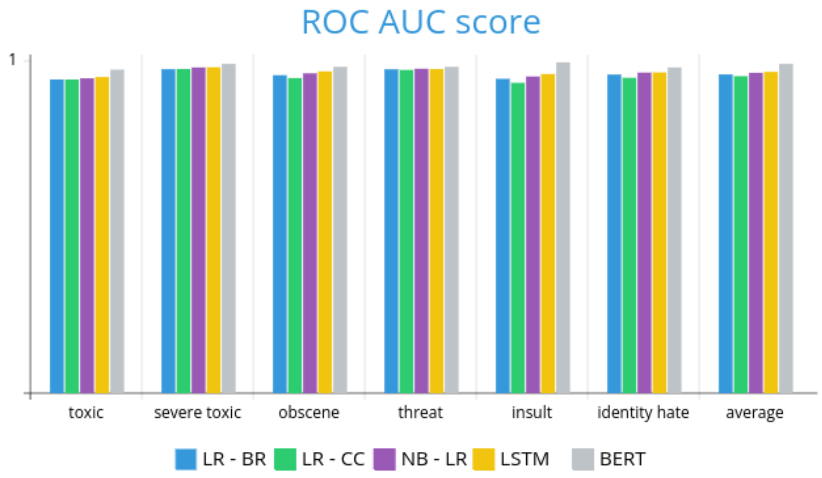

Figure 4: Bar chart representing ROC AUC score for Linear Regression - Binary Relevance (LR - BR), Linear Regression - Classifier Chains (LR - CC), Naive Bayes - Logistic Regression (NB - LR), Long Short-Term Memory (LSTM), BERT fine tuning..

6.3 Results

Table 3 shows the results of the experiments, reporting the ROC AUC score for each category as well as the mean column-wise ROC AUC score on all categories for each trained model, truncated at the fifth decimal digit. Both sci-kit learn metrics and torchmetrics gave the same score for this order of magnitude, therefore only one value is shown for each case. Each model has been trained 10 times, and each value corresponds to the average of the evaluation score over the experiments. As one should expect, the fine tuning of BERT provided the best scores, despite being trained for just one epoch, remarking once again the unchallenged superiority of transformers in the natural language processing tasks. This is clearly visibile in Fig. 4.

7 Conclusions and future work

Aim of this report and its relative project was to compare various machine and deep learning techniques addressed during the natural language processing course applied to a real everyday problem, such as the toxic comment classification task. After a brief introduction to the problem at hand, the report described the datasets used to train and evaluate the model as well as a concide analysis of the data they contains. Then it described the pre-processing steps, emphatizing the small differences between the various approaches when present. Follows a section about the abstract of the models in exam, i.e., logistic regression, naive bayes, recurrent neural networks and transformers, describing the chosen specific approach to the multi-layer problem, i.e., binary relevance and chain classifiers, NB-SVM, LSTM and the BERT fine-tuning. Finally, the last section described the experimental environment and the evaluation result, proving the great improvement derived from the introduction of transformers to the solution of natural language processing problems.
Talking about improvements, various are the possible ways to enhance each of the models, as for each of them it has been preferred to adopt a straight-forward approach, both regarding the architecture and the training process, especially regarding the deep learning solutions. For example, the use of GloVe[10] word vectors or the training of a bidirectional LSTM[11] could have a positive impact on the LSTM solution. Regarding the BERT fine-tuning, despite already being the best amongst the proposed models, it has only been trained for 1 epoch, and therefore it could probably benefit from a longer training.

References

[1] Everton Alvares Cherman, Maria Carolina Monard, and Jean Metz. Multi-label problem transformation methods: a case study. *CLEI Electronic Journal*, 14(1):4–4, 2011.

[2] Min-Ling Zhang, Yu-Kun Li, Xu-Ying Liu, and Xin Geng. Binary relevance for multi-label learning: an overview. *Frontiers of Computer Science*, 12:191–202, 2018.

[3] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine learning*, 85:333–359, 2011.

[4] Jan Salomon Cramer. The origins of logistic regression. 2002.

[5] Sida I Wang and Christopher D Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 90–94, 2012.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, dec 1997.

[9] Jin Huang and Charles X Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17(3):299–310, 2005.

[10] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[11] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.