COMP4901K Course Project

Team rtx – IP, Thomas

Toxic comment classification

https://github.com/tommyip/toxic_comment_classification

The problem

Toxic comments are a huge problem in today's online platforms. It is thus paramount that a robust and accurate system is built to automatically filter such text before they cause harm to other users.

In this project we will use the dataset from Kaggle's Toxic Comment Classification Challenge, which consists of a large number of hand labelled Wikipedia comments. This is a multi-label classification problem, where each instance of a comment can be a classified by a combination of the labels: toxic, severe toxic, obscene, threat, insult and identity hate. The objective is to maximise the mean column-wise ROC AUC score based on the probability predictions from the model.

The dataset

Examples from the dataset, one that is not toxic at all and one that fits multiple category:

Text	Toxic	Severe toxic	Obscene	Threat	Insult	Identity hate
That's in the version that TRP demand son having, I have it in use to expand on the GK section. –	0	0	0	0	0	0
, oh shit, stop making warnings, motherf**ker, also, f**k you.	1	1	1	0	1	0

The training dataset consists of a total of 159571 text/labels instances. Since our computational resource is limited, we will take a sample of 5000 for training, and 1000 for validation and testing respectively. The dataset is also highly unbalanced, for example there are only ~500 instances of comments that are labelled threat, compared to ~15000 which has a toxic label. Unfortunately since the labels are correlated, there are no simple algorithms to balance the dataset, as a result we have to rebalance the data manually (see the first notebook for label imbalances before and after balancing).

For reference, the winning entry in the Kaggle competition achieves a mean columnwise ROC AUC score of 0.988. Though it should be noted that it would be inappropriate to compare it to our results since we are only training and testing on a tiny subset of the provided data.

Attempts

#1 Naïve Bayes with BOW representation (baseline)

In this attempt we built a simple Naïve Bayes classifier to obtain a baseline score. Preprocessing steps includes stopword/non-alpha removal and lemmatization. The tokens are then vectorised with one-hot encoding to a vector of 16648 dimensions. The Naïve Bayes classifier is trained on each label in a one-vs-rest strategy.

Test ROC AUC score: 0.905

#2 Naïve Bayes with BOW including n-gram features

In this attempt, we created a custom head/stopword removal filter by counting the frequencies of tokens in the training set. The most/least common tokens are then stripped before vectorization. Experiments showed that this halved the size of the token dictionary while retaining test scores.

Instead of just using individual words as features, we introduced 2-gram features to the BOW representation. Unfortunately, this seems to make no improvement to the model. My hypothesis for this is that a lot of words that carry toxic meaning are either made-up on the spot or are a concatenation of multiple words, n-grams obviously cannot capture these information.

Test ROC AUC score: 0.894

#3 MLP classifier with word embedding representation

Bag-of-word models are simple to understand, however, they are poor at capturing meaning. In this attempt, we instead turn to word embedding models. We use a pretrained model from spaCy to convert tokens to 300-dimensions dense vectors. Each comment instance are then represented by a document vector calculated from the mean of individual word vectors. The classifier is a shallow MLP with one hidden layer of size 50. This simple approach performs surprisingly well and training only took around a minute on CPU.

Test ROC AUC score: 0.967

#4 LSTM with word embedding representation

Instead of simply taking the mean of individual token vectors to represent each comment instance, we tried to train a RNN to learn the mapping between a sequence of token vectors and the document vector. We use a LSTM network with 128 hidden units and 2 dense layers with 32 hidden units, a dropout layer is sandwiched between the dense layers to prevent overfitting. These hyperparameters are tuned using the validation set, the network is quite small but generalized much better than larger and deeper networks. Unfortunately this network still seem to overfit the training set, the training score is around 0.993 while the test score is only marginally better than the MLP approach.

Test ROC AUC score: 0.969

#5 Pretrained transformers (xlm-roberta-base)

In the previous attempt we saw that LSTMs are only marginally better than simply averaging a sequence of tokens to obtain a document vector for classification, so we try deeper and more complex language models to see if they can learn better. We retrain a pretrained model from huggingface (xlm-roberta-base) for the task. We also use automatic hyperparameters optimization to further improve the classifier.

Test ROC AUC score: 0.980

Concluding thoughts

From my experiment's results, we can see that larger and deeper models generally performs better. However, it is clear that going in this direction will yield diminishing returns. For example, a simple MLP

model that only took a few minutes of CPU training time has a very compelling score, while large LSTM/transformers takes much longer even on Google Colab's Tesla K80s but only perform marginally better.

A technique with huge potential that we did not explore is subword vectors. The embedding that we used are from word2vec which treats each token as an atomic entity. Unfortunately for the dataset at hand, a lot of informative words are actually multiple words concatenated together that occurs very rarely. We might be able to exploit this feature by using Facebook's fastText model instead of word2vec/glove. The reason we did not use it is the retrainable binary model could not fit into Google's Colab RAM.

-

Please see the notebooks for implementation and comments.