Semi-Automatic Discord Moderation

Austin John
UMBC
Department of Computer Science
ajohn8@umbc.edu

Jacob Enoch
UMBC
Department of Computer Science
jenoch1@umbc.edu

Abstract

A practical application of the deep learning comment moderation methods described by *Deep Learning for User Comment Moderation* Pavlopoulos et al. (2017) is proposed. An semi - automated moderation bot will be implemented to aid moderation teams in censoring inflammatory messages from text channels in Discord Servers.

1 Introduction

Moderation of any content platform (Twitter, Instagram, YouTube, etc.) is a time-consuming and expensive task. Moderators are tasked with manually reviewing comments on these platforms and determining if they are offensive, illegal, sexually suggestive or generally undesired content for public platforms. The use of a machine learning model that is able to accurately predict whether certain messages or comments are undesired could drastically reduce both the overhead generated by employing moderation teams but also the workload and expectations placed upon these moderation teams.

Inspiration for this project came from (Pavlopoulos et al., 2017). Pavlopoulos et al. designed, tested and compared various models (linear regression, multi-layer perceptron, convolutional neural network and recurrent neural network) with the end goal being to help moderate the comment sections of the Greek sports website Gazzetta. In comparison, the Discord model aims to instead moderate active text channels in real-time, providing near-instantaneous moderation feedback rather than post-hoc moderation. Similar moderation mechanisms were used for both models, including variations of a moderation threshold mechanism and model types.

Currently, many Discord servers opt to use an automated moderation system that uses a simple banned word list to remove comments that contain banned words or phrases. Any other moderation is typically left to teams of volunteers or employees of the companies running the Discord server. The implemented solution uses a combination of automatic and semi-automatic moderation techniques in order to bolster the robustness of the moderation decisions made by the bot. A semi-automatic method will be employed for any comment that the model says has a toxicity probability between 50% and 80%, resulting in the comment being flagged by the bot for manual moderation. Any comment that contains a word or phrase from a banned words/phrases list or has a probability greater than 80% likelihood of toxicity will be removed entirely without any manual moderation. In an optimal setting, the implemented solution would hopefully reduce the work that manual moderation teams need to do by intelligently and accurately flagging and/or removing comments as they appear in the chats.

2 Dataset

The dataset we used to train our model consisted of 120k+ Wikipedia[1] editor comments that were manually labelled as either "Toxic" (1) or "Not Toxic" (0) using a binary labelling schema.

This data was quite dirty despite some minimal preprocessing done by the team behind the data set. This is for two main reasons:

- data that originates from Wikipedia usually has quite a bit of markup language and strange symbology peppered throughout, and
- the textual data consisted of user-generated comments that did not have consistent punctuation, grammar, or structure throughout the data set

The data was also split between two separate files that had to be combined into one data set:

[1] Wikipedia Editor Comments Dataset

- Toxicity Annotated Comments consisted of columns rev_id (revision ID of the comment), comment, year and the split (train/development/test)

- Toxicity Annotations consisted of columns rev_id, a binary variable named toxicity (0 or 1) that describes, in general terms, if a comment is toxic or not and toxicity score, a categorical variable ranging from very toxic (-2), to neutral (0) to very healthy (2).

This classification task focused on the general toxicity score (0 or 1) in order to limit the scope of the modelling portion of the project because the data preprocessing and Discord bot implementation would need to be done alongside the experimentation/training/tuning for the three model architectures chosen.

2.1 Preprocessing

The data needed a considerable amount of additional preprocessing in order to both reduce the dimensionality and to make it compatible with the BERT (Devlin et al., 2019) base-uncased tokenizer and embedding models. This consisted of removing strange symbols, unnecessary characters or sequences, expanding contractions, and more in order to reduce dimensionality and increase the potency of the context each comment could provide to the model during training. The comments also had a highly variable length, with the smallest comments being less than 20 characters and the longest comments being over 6000 characters long. Many sentences needed to be trimmed down to a predefined length because of the limitations regarding the input dimensions of the BERT Base Uncased model used to generate sentence embeddings.

It is also important to note that the provided text data was not split into individual sentences - each comment could range from a single sentence to a few paragraphs in length. Initial efforts focused on using the Natural Language Tool Kit (NLTK) (Bird et al., 2009) library to split each comment into individual sentences and propagate the label of the unsplit comment to all sentences that belong to the unsplit comment and create a new, larger and hopefully more representative data set. However, this quickly became overwhelming because the training data set alone had over 415,000 sentences after being split into sentences. Generating sentence embeddings for all 415,000+ training sentences would have taken at least 12 hours of Google Colab GPU compute time and an untold amount of hard drive space. This was unfeasible because Colab does not provide unlimited processing time - it is likely that the embedding generation process would be interrupted before it could finish.

2.1.1 Data Cleaning

The first step was to remove unambiguously unnecessary characters from our comments like "NEWLINE_TOKEN", "TAB_TOKEN", "::::", etc. These were added by the creators of the data set to replace newlines and tabs.
Each comments' toxicity level was provided by 10 annotators who scored individual comments on a scale of -2 to 2. Final comment labels were generated by comment reviewer consensus: when 50% or more of the annotators agreed that a particular comment was toxic (1), the final label for the comment would be toxic (1). The comment would be labelled as non-toxic (0) otherwise.

The input comments' length had to further be reduced since the chosen BERT model only accepts input sequences up to 512 tokens in length. The input sequences' length was capped off at 500 to leave a buffer (12 characters) for the [CLS] and [SEP] tokens (plus two spaces) that the BERT tokenizer adds to each sequence.

2.1.2 Dimensionality Reduction

Reducing the dimensionality of the data set can provide great benefits to not only training speed but also overall model performance and robustness. Reducing the dimensionality for BERT involved using regex for simple housekeeping tasks like removing extra spaces, comment tags, leading spaces, leading colons, leading apostrophes, punctuation and converting to lowercase.

Another approach to further reduce dimensionality is to expand common contractions since there is no real difference in semantic meaning between "haven't" and "have not". Hence, our comments were passed through a function which expanded the most commonly known contractions in the English language[2].

2.1.3 Data Balancing

One of the major features of the data that the creators of the data set mentioned was the imbalance that was present between the toxic and non-toxic comments. As Figure 1 shows, the entire data set

[2]Common Contractions in English

had a highly unbalanced representation between toxic (1) and non-toxic (0) comments. A common

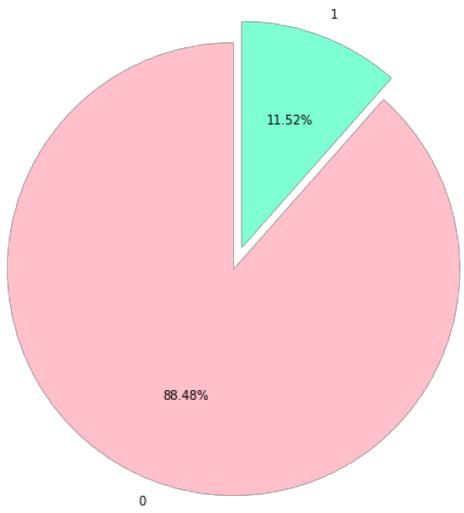

Figure 1: Toxic (1) vs Non-Toxic (0)

way to address imbalanced data sets such as these is to randomly and evenly sample data from both classes in the data, creating a smaller but much more balanced data set. Using an imbalanced data set to train machine learning models might result in the model disproportionately weighting non-toxic comments over toxic comments because they are so much more prevalent. Balancing the dataset will prevent the model from predicting "non-toxic" for nearly every label. The final training data set used to train the models consisted of 22,198 instances, with 11,099 toxic comments and 11,099 non-toxic comments (50/50) being the new balance.

3 Methodology

One of the first roadblocks the implementation ran into was generating relevant embeddings to use within a machine learning model. Initially, the plan was to use pre-trained word embeddings from Word2Vec or GLoVe. This was very quickly abandoned once a more thorough understanding of the quality of the textual data was realized. The data was user generated, meaning that it was highly likely that there could be a large number of unknown tokens with effectively no embedding representation with GloVe or Word2Vec due to misspellings or typos. Custom embeddings via a pre-trained BERT model seemed like a better option that would yield better results. A BERT model would also be able to facilitate using an RNN thanks to the specific output of the pre-trained model.

3.1 Embeddings

Embeddings were generated in batches of training, development and test respectively. The embeddings were written to a CSV file to avoid overloading the RAM of the Colab session. For each CSV file there were 768 columns that directly corresponded to a single embedding - this greatly sped up the time it took to access the embeddings once they were generated. This method also ensured that the time-consuming process of making embeddings only had to occur once.

One mistake was made during embedding generation, however. While making embeddings, only the raw embedding values were written to a file. Unfortunately, this mistake was realized too late. PyTorch and Keras both require very specific inputs in order to use their recurrent neural network API - raw embeddings were not enough. Generating any type of usable embedding with this data set was a feat in of itself so the implementation pivoted to using other models instead.

3.2 Models

As mentioned previously, the three models that would be experimented with would be a Multi-Layer Perceptron (MLP), a 2D convolutional neural network, and a 1D convolutional network. The MLP was chosen to act as a baseline to compare the other convolutional models to, while convolutional models were chosen to experiment with how well convolutional neural networks could capture spatial and semantic meaning within the raw embeddings.

Convolutional neural networks are often used in image classification. Images are typically three UxV pixel value grids stacked on top of one another, with each one corresponding to a color channel (RGB being the most common). Convolutional neural networks use sliding windows called kernels that capture a MxM area of a pixel grid. Kernels attempt to capture fine-grain detail and relationships of pixel values by repeatedly shrinking the size of the area with which values appear via successive convolutions and pooling of values, as shown in Figure 2 The idea behind using a convolutional neural network for natural language processing is similar. Sentence embeddings attempt to capture semantic meaning of sentences, similar to how pixel values might correspond to specific items in an image. It seemed plausible that a convolutional neural network would be able to also pick up on the spatial relationships of the values in the embeddings too.

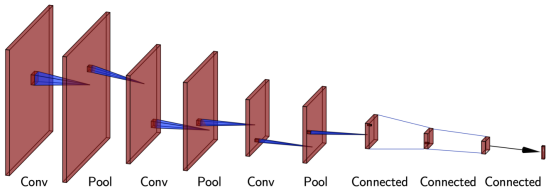

Figure 2: Convolutional Neural Network (Huang et al., 2018)

3.3 Training

The training process differed slightly depending on the model. Training for the 1D convolutional neural network and multi-layer perceptron was relatively straight forward: the raw 1x768 embeddings and their corresponding labels could be fed directly into Keras functional models. Training the 2D convolutional neural network required re-shaping the embeddings from a 1x768 dimensional vector to a UxV dimensional matrix - similar to a very small image. 768 does not divide evenly so the closest numbers were chosen to get a relatively square matrix of size 24x32 (very close to CIFAR-10 image sizes).

Training consisted of manually tweaking various hyper-parameters unique to each model. The learning rate, optimizer, number of neurons per layer, and activation functions were tuned for the MLP model. Kernel size, number of output filters, pooling type, optimizer, batch normalization, and regularization (L1 and L2) as well as the MLP hyper-parameters named above for the linear layers were all hyper-parameters that were tuned for the 2D CNN. The 1D CNN was very similar to the 2D CNN tuning, with the exception that the outputs were all 1-dimensional instead of 2-dimensional.

Hyper-parameters for each model were tuned between training sessions in an attempt to squeeze as much performance as possible out of them. When training completed, the validation and training accuracy were plotted in order to detect any issues regarding over-fitting or under-fitting. Hyper-parameters were tuned for each model until improvements ceased or were insignificant. Relatively well-performing models were saved and will be provided along with the code and assorted project files.

3.4 Evaluation

Evaluation was done two-fold - once during training using the development data set and one final evaluation using the test data. Evaluation over the development data was done at the conclusion of each epoch - this was done to see in real-time how well the model performs on data similar to the test data as model weights are tuned. At the end of each training session, a plot that showed validation versus training performance. A ROC plot was also generated after evaluation on the test data to plot TPR versus FPR, as well as to see how close the AUC score was to 1. For classification, an AUC score closer to 1 indicates that the model is effective. An AUC score that is greater than 0.5 is also a strong indicator that a model has actually learned something and is not relying on blind guessing.

4 Observations

The implementation required a pivot to making a MLP model as a baseline to be compared against 1D and 2D CNNs. The goal would be to improve the weights/hyperparameters of one of the CNNs to achieve an accuracy and recall that is comparable to or better than the MLP implementation. One of the most obvious observations related to the training and validation set charts. The MLP had the smoothest accuracy curves by far, as seen in Figure 3.

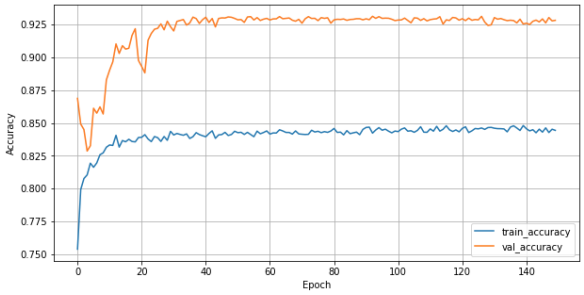

Figure 3: Train Acc. vs. Validation Acc. (MLP)

The training/validation accuracy scores for the convolutional neural networks were much more spiky in comparison. Despite this, the 2D CNN model was able to out-perform the MLP model, for reasons described below.

4.0.1 Multi-Layer Perceptron Architecture

The MLP was implemented using Keras with a Glorot Uniform seed. This Glorot Uniform seed aims to initialize weights such that the variance of the activation functions are same across each layer. This aids in preventing the gradients from vanishing or exploding during back propogation. Three hidden layers were used for the model, all using the same "ReLu" activation function with a 0.3 and 0.15 dropout between the hidden layers 1

and 3 respectively. Dropout is a technique where randomly selected neurons are "dropped out" from training, removing them from forward passes and weight updates on the backward pass. This leaves the other neurons to have to step in and make predictions for the neurons that are missing from the network. This in effect makes the network less sensitive to weights of specific neurons and be able to generalize better, thus making it less likely to over-fit on the training data. The final output layer uses the sigmoid activation function to "squish" all final values within a range between 0 and 1. The end result was a generally accurate and well trained model, as can be seen by the AUROC curve in Figure 4.

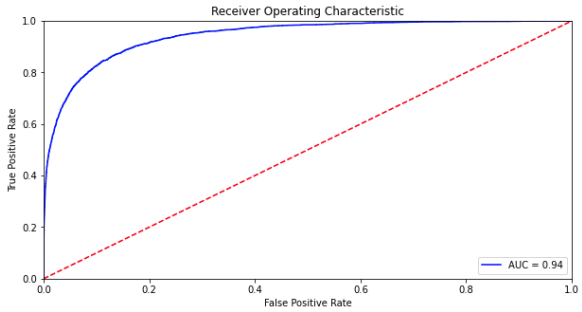

Figure 4: MLP AUROC

This model had the highest overall test accuracy (93%), lowest positive class recall (0.61) and second-highest AUROC score (0.942). This model was not chosen to be used for Discord moderation because of the low positive class recall. Positive class recall is being prioritized because missing a potentially toxic comment could be significantly more harmful when compared to accidentally flagging and/or removing a comment that was not actually toxic. The relatively low positive class recall might be indicative of the possibility that the MLP has inadvertently over-fit to the non-toxic comments, despite there being an equal number of toxic/non-toxic instances within the training set.

4.0.2 2D Convolutional Neural Network

Since CNN's generally work well with spatial data like images, the next logical choice of implementation was to implement both a 2D and 1D CNN and see how they performed relative to the baseline MLP. The data had to first be reshaped into what essentially were 24x32 1 channel images. They were then passed into a model with a single convolutional layer (32 filters, stride of 2, valid padding) and max pooling layer (2x2 pooling). The values were then flattened before being ran through the final output network containing 2 dense layers, the first with an ReLu activation function and the second with the sigmoid activation function. The 2D CNN showed a wide variation of accuracy between the validation and training data as shown in Figure 5.

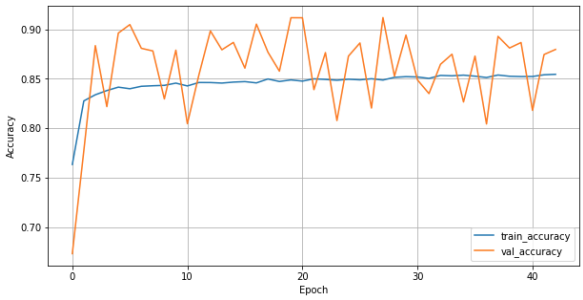

Figure 5: Train vs. Validation Accuracy (CNN)

Many model architectures were experimented with before this relatively simple but highly effective model was found. Some of the failed models involved stacking multiple convolutional/pooling/dropout layers, a flattening layer, and finally multiple dense linear layers. The more complex models tended to severely over-fit the training data so the model was iteratively simplified until high performance was achieved. The final 2D CNN model had the second highest test accuracy (91%), highest positive class recall (0.79) and highest AUROC score (0.944). This model was ultimately chosen to be used as the final moderation model because of the relatively high accuracy and because it had both the highest AUROC and positive class recall. The overall performance for this model was quite surprising. namely because of the inconsistent training charts seen in Figure 6.

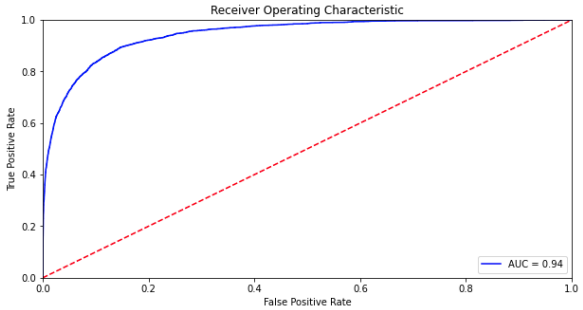

Figure 6: 2D CNN AUROC

4.0.3 1D Convolutional Neural Network

Since the data is primarily word embeddings in the 1d space, the next logical implementation was a 1D CNN - raw embeddings could be passed directly into the model. The implementation of the 1D was

largely similar to the 2D CNN where the implementation had one convolutioal layer (32 filters, stride of 1) and one MaxPooling (1x2 pool size) layer. The outputs were then passed through a flatten and Dense layer with ReLu before passing it through the final Dense layer with a sigmoid function.

Many architectures were also explored for the 1D CNN. The first architectures consisted of multiple stacked convolutional, pooling and dense layers. Again, the more complex models seemed to severely overfit the training data so layers were iteratvely removed until reasonable performance was achieved. Hyperparameter tuning commenced once reasonable baseline performance was achieved. The final 1D CNN model had the lowest overall test accuracy (90.3%), second-highest positive class recall score (0.77) and lowest AUROC score (0.938). This model, while still performing quite well, still performed the worst when compared to the other models. This was not surprising as this was the most exotic and experimental model chosen. Figures 7 and 8 show the model's respective training performance and AUROC scores.

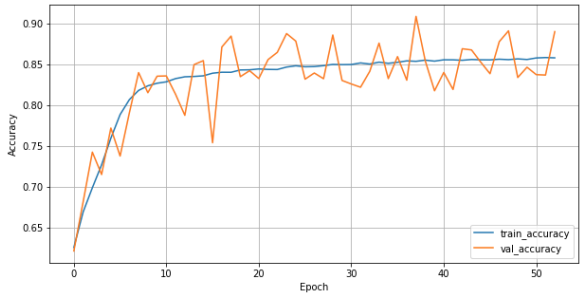

Figure 7: Train Acc. vs. Validation Acc. (1D CNN)

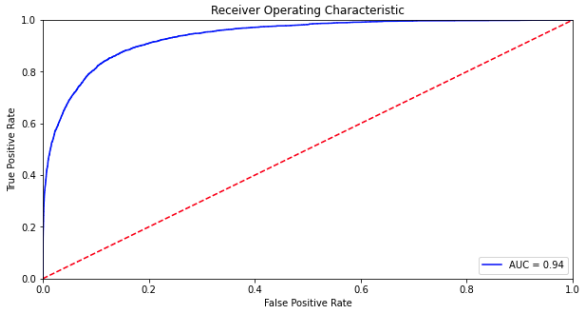

Figure 8: 1D CNN AUROC

5 Limitations

One of the most apparent limitations is the sensitivity of the bot. Currently, the bot will flag any comment that has a toxicity probability between 50% and 80% and remove any comment with a toxicity probability greater than 80%. The bot is generally good at noticing the difference between obviously toxic comments (i.e. noticing the difference between 'Hello there!' and 'F**k off idiot') but tends to struggle with some seemingly random sentences. For example, the phrase 'oh our model caught that' was flagged (but not removed) during testing despite having no toxic content present. It also tends to flag or remove messages that only mention potentially toxic topics. For example, the phrase 'Sexism is not cool' was deleted entirely despite not having any toxic content. This might be a limitation caused by the training data having to be trimmed before being tokenized and converted into embeddings. It is entirely possible that there might not have been enough context for the model to pick up on the differences between nuanced conversations and toxic comments. A potential work-around for this kind of sensitivity could involve having a separate "Discussions" or "Debates" channel that will only flag comments instead of potentially remove them.

6 Future Steps

The implementation mandated pivoting to models that took in raw embeddings from BERT as opposed to implementing an RNN function available in PyTorch or Keras that accepted hidden layers as its input. Since generating said embeddings are an insanely laborious and resource intensive task, it did not make sense to pass our inputs through BERT just to get the hidden layers needed for RNN training. Given more time and resources, this is definitely something that could be done in the future.

A substantial amount of our data contained more non toxic comments than it contained toxic comments. 11.52% of the wikipedia dataset contained toxic comments as can be seen in Figure 1. This imbalance would have wildely skewed our model's accuracy and recall. One fix is to randomly and equally sample the data which was the approach our implementation took. Another fix to resolve this imbalance and dataset size issue is to also use the twitter dataset, since tweets are capped at 280 characters we would never run into the input length issue with BERT. We could randomly and equally sample twitter's cleaned tweets and append the toxic and non toxic comments/tweets to our resultant dataset hence increasing the size of our training dataset for our model.

Attempts were also made in the early stages of the project to split the comments into separate sentences, tokenize these and then generate embeddings in order to circumvent the amount of information loss that could occur if all of the comments were trimmed to be 500 characters or less. This ended up being unfeasible due to time constraints. The training set consisted of over 400,000 sentences after being split up. Each of these 400,000 sentences would need to be tokenized and have embeddings generated. The training set alone would have taken over 16 hours to tokenize and generate embeddings for. In the future, we would like to be able to see the performance differences between the model we have now versus the model that was trained on sentences that have much more context in them to base predictions off. It is likely that some sentences would need to still be trimmed but most sentences would likely be less than 500 characters.

References

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Derek Huang, Arianna Serafini, and Eli Pugh. 2018. *Music Genre Classification*.

Son Thanh Nguyen. Better-profanity (blazingly fast cleaning swear words (and their leetspeak) in strings).

John Pavlopoulos, Prodromos Malakasiotis, and Ion Androutsopoulos. 2017. Deep learning for user comment moderation.

Dennis M. Riehle, Marco Niemann, Jens Brunk, Dennis Assenmacher, Heike Trautmann, and Jörg Becker. 2020. Building an integrated comment moderation system – towards a semi-automatic moderation tool. In *Social Computing and Social Media. Participation, User Experience, Consumer Experience, and Applications of Social Computing*, pages 71–86, Cham. Springer International Publishing.

Elizaveta Zinovyeva, Wolfgang Karl Härdle, and Stefan Lessmann. 2020. Antisocial online behavior detection using deep learning. *Decision Support Systems*, 138:113362.