Capstone Project Report. Machine Learning Engineering Nanodegree

Text Classification: Toxic Comment Analysis

Stefano Paccagnella

6 October 2021

Contents

Project Overview

In today's word 2.5 quintillion bytes of data are generated daily. The importance of interpreting this continuous stream of information has never been so important, and the actual use of AI and ML algorithms is finding fertile ground for achieving concrete and extraordinary results. Among these data exchanged, text represents a huge proportion of the overall information transmitted worldwide.

As unstructured data, the analysis of text can be extremely hard to achieve, but thanks to Natural Language Processing (NLP) and ML algorithms this type of analysis is now getting easier and easier, which is also translating in an increasing value for business. Among the main application of NLP, we surely have Sentiment Analysis. The main scope of Sentiment Analysis is to extract the meaning behind a text, by classifying it as positive, negative or neutral.

The application of NLP and Text Classification which will be used in this project is referring to a Kaggle competition held in 2017-2018, the "Toxic Comment Classification Challenge". The idea behind the competition is to try to go behind the scope of a traditional sentiment analysis approach, i.e., to classify a text as positive/negative and to develop a model which will recognize the level of toxicity of a target text. This will mean for example to classify the target variable as an "insult", "threats", "obscenity", etc., based on a pre-defined set of labels.

Project Statement

The use of "hate speech" and harassment online is a treat not only from a social point of views, but also economically and politically speaking. Therefore, the ability to classify comments, post, messages, etc. in the proper way, it is becoming every day more impelling for all online platforms.

In this context, the Conversation AI team has developed several models to improve the monitoring of conversation online. The main issue of these models (despite the error in the prediction as such) is the fact that they don't allow to label the type of toxicity of the comments analyzed. This means that we cannot really generalize the use of these predictions, as not all online platforms may be willing to act on a text (e.g., with censorship) with the same level of intervention. It will mainly depend on the internal policy of the company. Therefore, the possibility to develop a model that will grant the possibility to define and label the type/tone of a text data type into a specific category, e.g., "insult", "threats", etc., it is quite interesting.

In order to do so, we will be using a set of NLP models, with increasing complexity, to identify the one that is better performing with the provided dataset and possibly scalable on other NLP text classification problems. The models we have identified for this project are: "Bag of Words, "Word2Vec Embeddings", "FastText Embeddings", "Convolutional Neural Networks (CNN)" and possibly other two NLP models like "Long Short-Term Memory (LSTM)" and "Bidirectional Encoder Representation from Transformers (BERT)".

The benchmark model that will be used is the simpler model at the base of Text Classification, i.e. "Bag of Words". In this model, a text is represented as a "bag" (multiset) of words, disregarding grammar and even word order but keeping multiplicity. We will be using a sklearn implementation

of “Bag of Words”, *CounterVectorizer*, to convert the provided text into a numerical matrix that can be than mapped on the pre-defined set of label provided (i.e. “toxic”, “severe toxic”, “obscene”, “threat”, “insult” and “identity hate”). We will then use a Naive Bayes and Logistic Regression algorithms on data created by the CountVectorizer and will keep as benchmark the one which is better performing. The prediction will be then realizes by using a Multi-Output Classifier from sklearn to predict all 6 categories.

We will than evaluate the performance of other models Vs. the benchmark one to identify the one that is better fitting with our dataset and scalable to other NLP projects.

Metrics

In order to evaluate the performance of our models, we will use the AUC-ROC Curve. We have decided to use this evaluation metrics as the best one in terms of multi-class classification problem. AUC stands for “Area under the Curve”, while ROC stands for “Receiver Operating Characteristics”. More specifically, ROC is a probability curve and AUC represent the degree of measure of separability, i.e., it tells how well the model is capable of distinguishing between labels/classes. Ideally the best model should score an AUC closer to 1, which means is has a very good separability. At the same time the worst model has an AUC closer to 0, which means that has a bad score in separability. A model with AUC at 0.5 means that the model has no class separation capacity at all (see understanding AUC-ROC curve).

What is a ROC:

The ROC curve is plotted with TPR against the FPR where TPR is on the y-axis and FPR is on the x-axis.

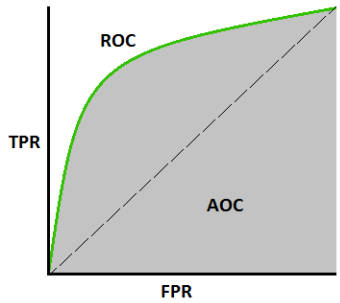


Figure 1 – AUC -ROC Curve

The terms used by the AUC-ROC curve can be explained as:

True Positive Rate (TPR) is a synonym for recall and is therefore defined as follows:

$$TPR=TPTP+FN$$

False Positive Rate (FPR) is defined as follows:

$$FPR=FPFP+TN$$

As the *AUC-ROC* curve depicts the rate of true positives with respect to the rate of false positives, we can also say that it displays the sensitivity of the classifier model.

$$Sensitivity = \frac{TP}{TP + FN}$$

ROC curve can be used to select a threshold for a classifier, which maximizes the true positives and in turn minimizes the false positives. ROC Curves help determine the exact trade-off between the true positive rate and false-positive rate for a model using different measures of probability thresholds.

What is a AUC:

Area Under Curve or AUC is one of the most widely used metrics for model evaluation. It is generally used for binary classification problems. AUC measures the entire two-dimensional area present underneath the entire ROC curve. AUC of a classifier is equal to the probability that the classifier will rank a randomly chosen positive example higher than that of a randomly chosen negative example. The Area Under the Curve provides the ability for a classifier to distinguish between classes and is used as a summary of the ROC curve. The higher the AUC, it is assumed that the better the performance of the model at distinguishing between the positive and negative classes.

To conclude, AUC-ROC is the valued metric used for evaluating the performance in classification models. The AUC-ROC metric clearly helps determine and tell us about the capability of a model in distinguishing the classes. The judging criteria being – Higher the AUC, better the model. AUC-ROC curves are frequently used to depict in a graphical way the connection and trade-off between sensitivity and specificity for every possible cut-off for a test being performed or a combination of tests being performed.

Analysis

The dataset, provided in the Kaggle "Toxic Comment Classification Challenge", it is composed by an extraction of Wikipedia comments, which have been manually pre-labelled. The level of toxicity defined/labelled is "toxic", "severe toxic", "obscene", "threat", "insult" and "identity hate".

The data provided are mainly:

- ***train.csv*** *- the training set, containing comments with their binary labels*
- ***test.csv*** *- the test set, on which prediction on toxicity probabilities must be determined. To deter hand labelling, the test set contains some comments which are not included in scoring.*

The goal is to develop a model which predicts a probability of each type of toxicity for each comment in the test file.

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0

In order to have a better understanding our dataset, let's proceed with some **Exploratory Data Analysis - EDA.**

From the dataset we can notice we have around 160k training texts and about 153k test data.

```
train_text.shape
```

```
(159571, 8)
```

```
test_text.shape
```

```
(153164, 2)
```

In order to apply any NLP model (or ML more in general), it is also very important to analyze if there are any Null values in our dataset. From the below table it is evident (from the mow mean) that there are no Null values that we must clean.

	toxic	severe_toxic	obscene	threat	insult	identity_hate	char_length
count	159,571.00	159,571.00	159,571.00	159,571.00	159,571.00	159,571.00	159,571.00
mean	0.10	0.01	0.05	0.00	0.05	0.01	394.07
std	0.29	0.10	0.22	0.05	0.22	0.09	590.72
min	0.00	0.00	0.00	0.00	0.00	0.00	6.00
25%	0.00	0.00	0.00	0.00	0.00	0.00	96.00
50%	0.00	0.00	0.00	0.00	0.00	0.00	205.00
75%	0.00	0.00	0.00	0.00	0.00	0.00	435.00
max	1.00	1.00	1.00	1.00	1.00	1.00	5,000.00

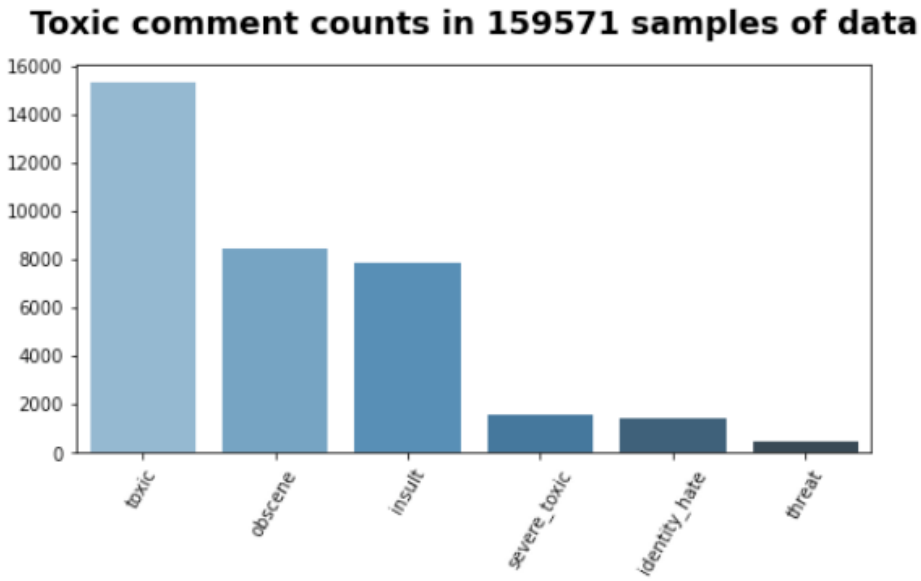
–

which is alternatively described also from the below image, both showing the absence of Null values in the training and test dataset.

```
train_text.isnull().any(),test_text.isnull().any()
```

```
(id               False
 comment_text     False
 toxic            False
 severe_toxic     False
 obscene          False
 threat           False
 insult           False
 identity_hate    False
 dtype: bool,
 id              False
 comment_text    False
 dtype: bool)
```

It is also interesting to analyze the distribution of our testing data. More specifically let’s analyze the labels distribution.



From the above graph it is evident that most classified comments are labelled “Toxic”. This shows an unbalance in the classification of the training text. It is also interesting to see that 89% of testing data are unlabeled.

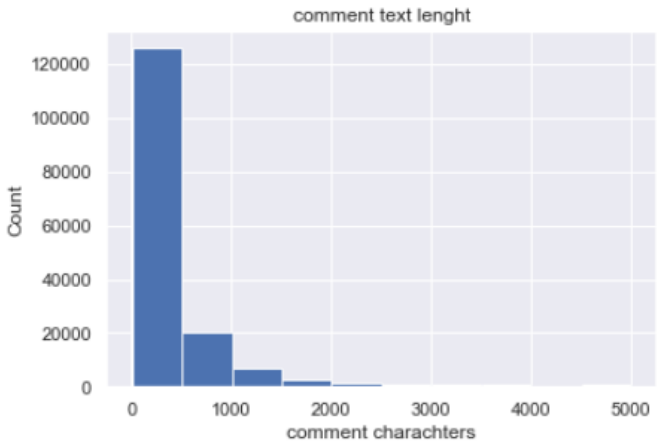
We can conclude that there is an unbalance between classified and classified-non classified testing data, which is recap by the below table:

	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0.90	0.99	0.95	1.00	0.95	0.99
1	0.10	0.01	0.05	0.00	0.05	0.01

This is very important as we will need to take into consideration this unbalance when we will apply any of our ML models. Nevertheless, we need also to be cautious on how we are going to consider this unbalance, especially if we consider the unbalance between different classes. Here, in fact, we are speaking mainly of a Multi-Label Classification problem, which is different from multi-class classification, i.e., trying to classify a text over multiple variables simultaneously, e.g., “positive”, “negative” and “neutral” (like in sentiment analysis). In our Multi-Label Classification problem, we must consider each category as a separate word. Therefore, in our dataset the unbalance is between what is classified as toxic (represented by the integer 1) and what is not toxic (represented by the integer 0) or more specifically not-labelled; or what is considered an insult (represented by the integer 1) and what is not labeled as an insult (represented by the integer 0), etc. Our ML model will most likely predict over each single category separately and we will need most likely to consider this “internal” distribution, rather than an external distribution between different classes. It is also important to consider that the “external” unbalance between classes tells the model how

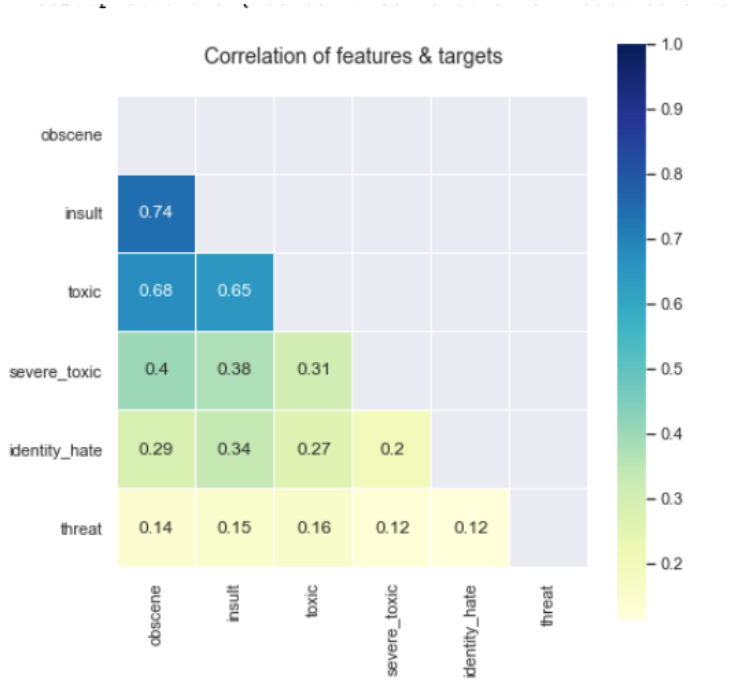
frequent a label is compared to another, e.g., “toxic” may be a most likely prediction over “threat”. Overall, we will need to consider this unbalance in each model implementation and partially we have already done that by considering the ROC-AUC metrics, instead of using simply the accuracy that can be tricked by class unbalance.

Another aspect that we can analyze is the lengths of characters in the comment text.



From the above histogram, we can conclude that most of the text length are within 500 characters, with some up to 5,000 characters long.

It could be also interesting to check the correlations among predictors.



From the above heat map, it looks like some of the labels are higher correlated, e.g., insult-obscene has the highest at 0.74, followed by toxic-obscene and toxic-insult.

ML models – Analysis, Methodology and Evaluation

As mentioned at the beginning of our project, the main ML NLP algorithms that we will be using are: "Bag of Words, "Word2Vec Embeddings", "FastText Embeddings", "Convolutional Neural Networks (CNN)", "Long Short-Term Memory (LSTM)" and "Bidirectional Encoder Representation from Transformers (BERT)".

In this section, we will analyze the functioning of each model and their implementation.

Bag of Words:

The bag-of-words model is a way of representing text data when modeling text with machine learning algorithms.

The bag-of-words model is simple to understand and implement and has seen great success in problems such as language modeling and document classification. This is the reason why "Bag of Words" will also to be considered in our project as the Benchmark model of our analysis.

The implementation of "Bag of Words" technique is quite straight forward. It basically the process of tokenizing a selected text and find out the frequency of each token. For example, in the sentence "It was the best of times" is deconstruct in its simpler token: 'It', 'was', 'the', 'best', 'of', 'times'. It than create a vector input that can be analyzed by ML algorithms. This means calculating the frequency of each token in a target text, assigning an integer to each token if present in that text. In our example, we could land in something like:

"It was the best of times" = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]

The process of converting NLP text into numbers is called vectorization in ML. Different ways to convert text into vectors are:

1. Counting the number of times each word appears in a document.
2. Calculating the frequency that each word appears in a document out of all the words in the document.

	1 This	2 movie	3 is	4 very	5 scary	6 and	7 long	8 not	9 slow	10 spooky	11 good	Length of the review(in words)
Review 1	1	1	1	1	1	1	1	0	0	0	0	7
Review 2	1	1	2	0	0	1	1	0	1	0	0	8
Review 3	1	1	1	0	0	0	1	0	0	1	1	6

The above implementation technique is what we will implement in our project. More specifically, we will:

1. Split our training data set into Train and Validation set;
2. Pre-process our dataset to clean the main data text issues;
3. Creating the Bag of Words, or frequency matrix;
4. Initialize and train a Multi Output Classifier;
5. Measuring the performance on the validation set.

Data pre-processing:

In order to eliminate all the issues, we may have in the comments text of our dataset, we have decided to:

- Remove all English stop words.
- Remove punctuation

This will allow us to create a more fluid dataset to be processed and analyzed by our ML algorithm. By keeping, in fact, these attributes will only add noise to our input dataset.

Bag of Words:

The second step has been to create a frequency table with a max feature od 5000 words. In our EDA analysis, we have in fact seen that the maximum number of characters for some text has been of up to 5000 words. More features will only increase the risk of having a sparse input, which will lead to misprediction by our model. Our final matrix will look like the one below:

	abc	abide	ability	able	abortion	about	absence	absolute	absolutely	absurd	...	yourselfgo	youth	youtube	youve	ytmndin	yugoslavia	zealand	zero
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0

Multi Output Classifier:

The next step is to initialize a Multi Output Classifier. Since we need to classify each sentence as toxic or not, severe_toxic or not, obscene or not, threat or not, insult or not and identity_hate or not, we need to classify the sentence against 6 output variables (This is called Multi-Label Classification which is different from multi-class classification where a target variable has more than 2 options e.g., a sentence can be positive, negative and neutral.

For the same, we will be using MultiOutputClassifier from sklearn which as mentioned earlier is a wrapper. This strategy consists of fitting one classifier per target. We will be trying using a Naïve Bayes model and a Logistic Regression one. We will than keep the model that is better performing (AUC-ROC) with our dataset.

As in the EDA phase, we have seen that our model is quite unbalanced, in the logistic regression model, we will apply a class weighting. When the class_weights = 'balanced', the model automatically assigns the class weights inversely proportional to their respective frequencies.

To be more precise, the formula to calculate this is:

wj=n_samples / (n_classes * n_samplesj)

For example:

- wj is the weight for each class (j signifies the class)
- n_samplesis the total number of samples or rows in the dataset
- n_classesis the total number of unique classes in the target
- n_samplesjis the total number of rows of the respective class

n_samples= 43400, n_classes= 2(0&1), n_sample0= 42617, n_samples1= 783

- Weights for class 0:

    - w0= 43400/(2*42617) = 0.509
- Weights for class 1:
    - w1= 43400/(2*783) = 27.713

Evaluate the Model

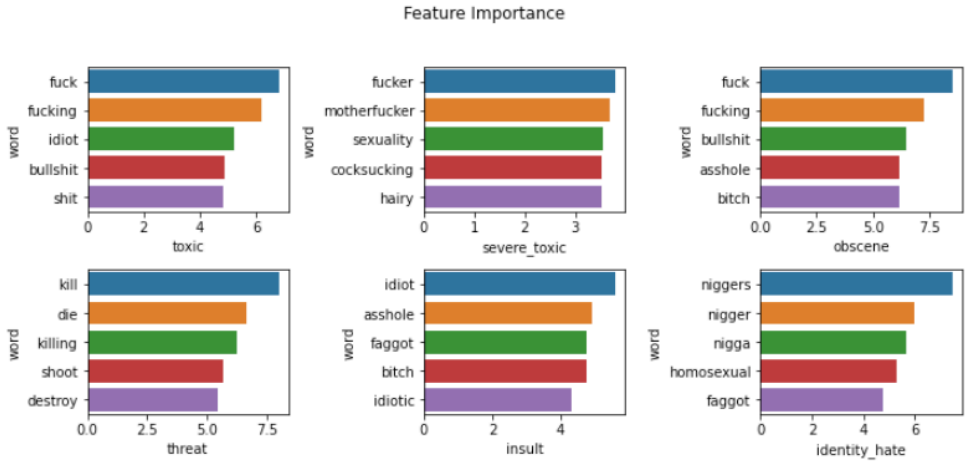
The final AUC-ROC score for each model is:

	Model	Mean AUC
0	MultinomialNB	0.94
1	LogisticRegression	0.94

By using both classifiers the score looks performing well, with almost 95%. We have anyway decided to use the Logistic regression one, as we consider that the problem of the unbalance classes is better treated.

This 95% AUC-ROC score will be used as the benchmark for all other models used in this project.

To conclude, we have also decided to analyze the Top 5 features importance of each predictor per each label. The results are shown in the graph below:



We can see that the models are quite rightly selecting the most important features and it makes complete sense. E.g., for threats — words like kill, shoot, destroy, etc. are most

important. For identity hate — words like nigger, nigga, homosexual, faggot. Most important words for toxic are less extreme than most important words for severe toxic.

As for any model used, there is always the margin for improvement. I think in this case, the main steps we should be using are:

1. Try TF-IDF instead of CountVectorizer. TF-IDF tend to perform better than CountVectorizer in some cases
2. Try ensemble models instead of Vanilla ML models. Bagging and Boosting models give better results than classic ML techniques in most cases
3. Better Text Preprocessing (Typo correction, Lemmatization, etc. can be done to further improve the model).

Word2Vec:

The next step in our project is to use another NLP model, which is called Word2Vec. Instead of using a CountVectorizer, as for the Bag of Words, we will try to create an embedding.

Word embeddings is a technique where individual words are transformed into a numerical representation of the word (a vector). Where each word is mapped to one vector, this vector is then learned in a way which resembles a neural network. The vectors try to capture various characteristics of that word about the overall text. These characteristics can include the semantic relationship of the word, definitions, context, etc. With these numerical representations, you can do many things like identify similarity or dissimilarity between words.

A machine cannot process text in their raw form, thus converting the text into an embedding will allow users to feed the embedding to classic machine learning models. The simplest embedding would be a one hot encoding of text data where each vector would be mapped to a category.

For example:

```
have = [1, 0, 0, 0, 0, 0, ... 0]
a    = [0, 1, 0, 0, 0, 0, ... 0]
good = [0, 0, 1, 0, 0, 0, ... 0]
day  = [0, 0, 0, 1, 0, 0, ... 0] ...
```

The effectiveness of Word2Vec comes from its ability to group together vectors of similar words. Given a large enough dataset, Word2Vec can make strong estimates about a word meaning based on their occurrences in the text. These estimates yield word associations with other words in the corpus. For example, words like “King” and “Queen” would be very similar with one another. When conducting algebraic operations on word embeddings you can find a close approximation of word similarities.

For example:

```
King    -    Man    +    Woman    =    Queen
[5,3]   -    [2,1]  +    [3, 2]   =    [5,4]
```

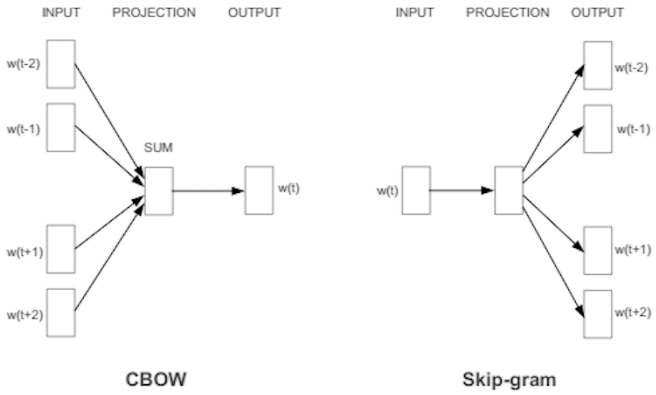
There are two main architectures which yield the success of word2vec. The skip-gram and CBOW architectures.

CBOW (Continuous Bag of Words)

This architecture is very similar to a feed forward neural network. This model architecture essentially tries to predict a target word from a list of context words.

Continuous Skip-Gram Model

The skip-gram model is the literal opposite of the CBOW model. In this architecture, it takes the current word as an input and tries to accurately predict the words before and after this current word.



In this project we will be using Gensim library. Gensim library enables us to develop word embeddings. Gensim gives you an option to choose either CBOW or Skip-gram while training your own embeddings. (Default is CBOW). Along with it, Gensim also has a directory of pre-trained embeddings which are trained on several documents like wiki pages, google news, Twitter tweets, etc. In this example, we will be using a pre-trained embedding based on the Google News corpus (3 billion running words) word vector model (3 million 300-dimension English word vectors).

The main steps we will be using to build our model are:

1. Pre-process our dataset to clean the main data text issues.
2. Load a pre-trained embedding model
3. Convert our target text into embeddings by using the pretrained model
4. Train and Validate the Multi-Output Classifier.
5. Measuring the performance of our model.

Data pre-processing:

As for the beg of words model, the main pre-processing of our data will be to:

- Remove English stop word.
- Remove punctuation
- All characters to lowercase
- Transform all sentences into tokens.

Load a pre-trained embedding model:

The model we will be using is a model offered by Google. The Goggle News model embedding vector has about 3 M words. By using a pre-trained model will allow us to better perform without having a large dataset. An example of how a Google news vector looks like is below:

```
[-0.22753906 -0.07617188 -0.06787109 -0.1015625   0.20214844  0.12890625
 0.1796875  -0.11035156  0.01123047  0.01794434  0.12402344  0.11132812
-0.3359375  -0.01104736 -0.16015625 -0.16113281 -0.13769531  0.4296875
-0.03979492  0.05297852]
time: 17.9 ms (started: 2021-10-10 18:28:57 +02:00)
```

Train and Validate the Multi-Output Classifier

Here we take the input tokenized texts from earlier and get the embeddings for each word in texts from the pre-trained embedding vector. This will give us the final input dataset in form of an embedding per sentence which can be used to train along with the output variables. More specifically, we will:

1. load the embedding vector from the training dataset.
2. split the embedding vector into train and validation set.
3. Apply a Logistic Regression on training embedding vector and output variables
4. Make prediction and evaluate the model

Evaluate the Model

Even though word2Vec has performed well in many NLP use cases, the AUR-ROC score is around 57% in our use case. Compared to the previous benchmarking model it is obvious that the model has scored quite purely.

There are of course margin of improvements and future development that could be implemented, like:

- Train a Word2Vec model from scratch
- Try ensemble models instead, like Vanilla ML models, Bagging and Boosting models which usually gives better results compare to classic ML techniques
- Better Text Preprocessing Typo correction etc. can be done to further improve the model

FastText:

If in word2vec, we have used Gensim library for getting pre-trained Word2Vec models/embedding vectors for the words used in the sentences, mapped them against the output variables toxic, severe_toxic, obscene, threat, insult, identity_hate, and used Multi-Output Logistic Regression Classifier wrapper from sklearn to create Logistic Regression models for all the 6 output variables.

In this one, we will be using the FastText library for both generating embeddings for the sentences as well as text classification. In fact, it gives us an option of doing both in one go.

In 2016, Facebook AI Research (FAIR) open-sourced fastText, a library designed to help build scalable solutions for text representation and classification. fastText take the idea of word embeddings in Word2Vec a step ahead and learns representations for character n-grams, and to represent words as the sum of the n-gram vectors. Taking the word "where" and n = 3 as an example, it will be represented by the character n-grams: <wh, whe, her, ere, re>.

FastText has 2 main benefits over regular word2vec embeddings:

- Word2Vec faces the problem of Out of vocabulary (OOV): Let's say we are training a Word2Vec model from scratch, we set up a vocabulary that contains all the words in the training data. Now if we have a new word in the test data for which we might be needing embedding, the new missing word will be OOV.
- By using a distinct vector representation for each word, the Word2Vec model ignores the internal structure of words

The above algorithm structure is what we will apply to our project. More specifically, we will:

1. Split train dataset into train and validation data
2. Pre-process our dataset to clean the main data text issues.
3. Train and Validate the Classifier.
4. Measuring the performance of our model.

Data pre-processing:

As before we will be applying the same data preprocessing function. We will also transform each label into __label 0__ and __label 1__ as required by FastText algorithm.

Train and Validate the Multi-Output Classifier:

Since the FastText classifier takes input a CSV file with the text data and the class label, we can't use the Multi-Output Classifier wrapper we were using in earlier notebooks. Moreover, as there is no API to date that can take a validation set and give out probabilities for the positive case, we can only get probabilities for one sentence at a time. So, we run a for loop around each of the validation sentence and store the probabilities in a list. We need probabilities as the performance metric is ROC-AUC.

Evaluate the Model:

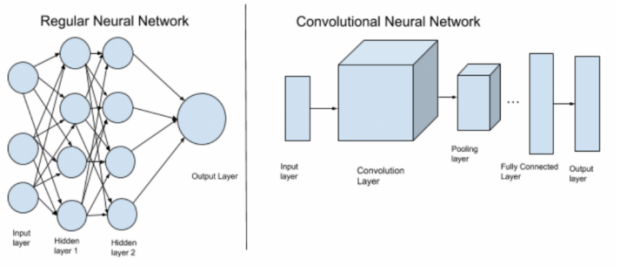
The total AUR-ROC score for the FastText implementation is of around 77%, which is much better than the Word2Vec model. Also, these are early results just on 2 epochs and not-tuned parameters.

In this optic there is margin for future development, like:

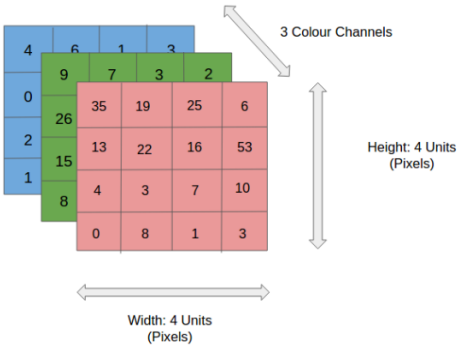
1. Better Text Preprocessing Typo correction etc. can be done to further improve the model
2. Try to better treat the unbalance between classes
3. Try tuning the hyperparameters to get better results

Convolutional Neural Networks (CNNs)

In this part of the project we will implement a Convolutional Neural Network (CNN) which is a deep learning algorithm. It is usually applied to image prediction, by assigning importance (weights and biases) to various aspects/objects in the image. The algorithms try to replicate the structure of a human brain to predict differences among inputs. The good results obtained through the application of CNN have of course extended the use of these algorithms to other fields, like NLP.



When we refer to CNN image classification, we can have a matrix representation of an image in the picture below:



In the figure, we have an RGB image which has been separated by its three color planes — Red, Green, and Blue. There are a number of such color spaces in which images exist — Grayscale, RGB, HSV, CMYK, etc. You can imagine how computationally intensive things would get once the images reach dimensions, say 8K (7680×4320).

The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction.

Convolution Layer — The Kernel

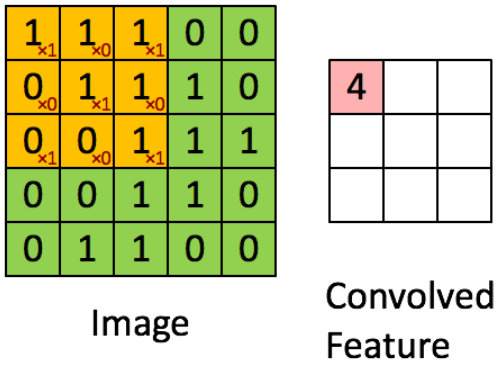


Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB)

In the above demonstration, the green section resembles our 5x5x1 input image, I. The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the Kernel/Filter, K, represented in the color yellow. We have selected K as a 3x3x1 matrix.

The magic is that we don't need to specify the kernels. We just need to mention the number of kernels and the model will learn the kernels on itself, just like weights in a normal ANN. The general idea is that, as we keep on increasing the number of Conv and Pool layers, the more complex features the model will be able to detect. The 1st layers recognize simple things like lines/colors and subsequent layers recognize more complex patterns.

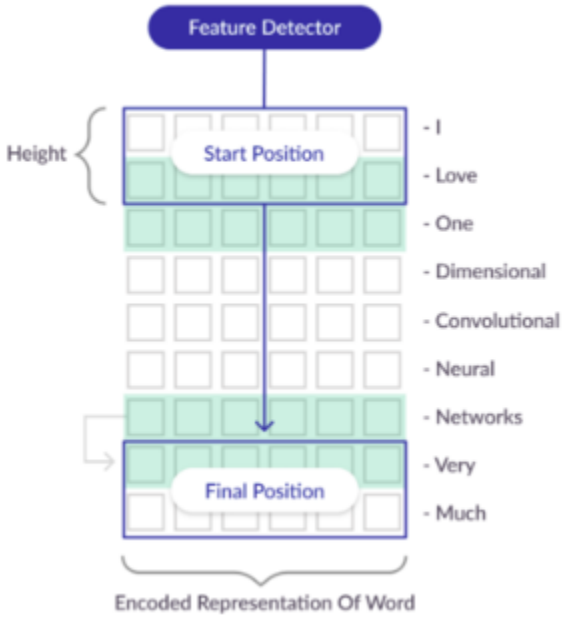
Pooling

Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the

network. The pooling layer summarizes the features present in a region of the feature map generated by a convolution layer.

In our project, we will be applying a 1D CNN. In Keras, the easiest way to define a model is to initiate a Sequential model class and keep adding required layers. A Sequential model is a plain stack of layers where each layer has exactly one input tensor and one output tensor.

A standard model for document classification is to use an Embedding layer as input, followed by a 1D convolutional neural network, pooling layer, and then a prediction output layer. We used 1 embedding layer, 3 sets of Convolution and Pooling layer, and 2 sets of Dense layer. We can either use a pre-trained embedding (like Word2Vec) to generate an embedding matrix of size Vocabulary Dimension of embedding or train a fresh embedding putting it as an input layer along with other weights.



Conv1D: CNNs were developed for image classification, in which the model accepts a 2-D input representing an image's pixels and color channels. This same process can be applied to 1D sequences of data. The model extracts feature from sequence data and maps the internal features of the sequence. CNNs consider the proximity of words to create trainable patterns. The kernel size/height in the convolutional layer defines the number of words to consider as the convolution is passed across the input text document, providing a grouping parameter. In our case, it will consider 5 words at a time, and in the image, it will consider 2 words at a time.

Max Pooling layer will consolidate the output from the convolutional layer.

We use sigmoid activation in the output layer. The sigmoid function gives us a probability score between 0 and 1 from each out of the output node. If we would have used softmax it gives a probability distribution across the output nodes that adds to 1.

Overall:

- For binary classification, we can have 1 output units, use sigmoid activation in the output layer and use binary cross-entropy loss.
- For multi-class classification, we can have N output units, use SoftMax activation in the output layer and use categorical cross-entropy loss.
- For multi-label classification, we can have N output units, use sigmoid activation in the output layer and use binary cross-entropy loss.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 128)         2560000
_________________________________________________________________
conv1d (Conv1D)              (None, None, 128)         82048
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, None, 128)         0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, None, 128)         82048
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, None, 128)         0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, None, 128)         82048
_________________________________________________________________
global_max_pooling1d (Global (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 128)               16512
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 774
=================================================================
Total params: 2,823,430
Trainable params: 2,823,430
Non-trainable params: 0
_________________________________________________________________
None
```

As discussed, the above 1D CNN is what we will implement in our project. More specifically, we will:

1. Pre-process our dataset to clean the main data text issues.
2. Implement the 1D CNN
3. Compile the model and Predict the Test data
4. Evaluate the model

Data pre-processing:

In order to process an input in a CNNs model, we need to transform this input into a number. Keras provides some basic tool to process data and prepare them for CNNs. We will use Tokenizer class, that will transform the text into a vector, by transforming each

text into a sequence of integers or into a vector where the coefficient of each token will be binary.

To do this we will have to perform 3 main steps:

- *Initializing the Tokenizer class*: All punctuations is removed, then sentences are splatted into sequences of words to be than split into tokens. We will than index this token. 0 is the reserved parameter to be assigned to any words. We set num_words to MAX_NUM_WORDS (20000) which is the maximum number of words to keep, based on word frequency. Only the most common num_words-1 words will be kept.

- *Calling the fit_on_texts function* - Updates internal vocabulary based on a list of texts. This method creates the vocabulary index based on word frequency. So, if you give it something like, "The cat sat on the mat", it will create a dictionary s.t. word_index["the"] = 1; word_index["cat"] = 2 it is word -> index dictionary so every word gets a unique integer value. 0 is reserved for padding. So lower integer means more frequent word (often the first few are stop words because they appear a lot).

- *Calling the texts_to_sequences function* - Transforms each text in texts to a sequence of integers. So, it basically takes each word in the text and replaces it with its corresponding integer value from the word_index dictionary.

Implement the 1D CNN

Before starting to train the model, we need to configure it. We need to mention the loss function which will be used to calculate the error at each iteration, optimizer which will specify how the weights will be updated, and the metrics which is to be evaluated by the model during training and testing

While fitting/ training the model, along with the training set we also pass the following parameters:

- batch_size = Number of samples that go through the network at a time.
- epochs = Number of times the whole set of training samples goes through the network
- validation_data = the dataset that will be used to evaluate the loss and any model metrics at the end of each epoch. This set will not be used for training.

Evaluate the model

After running our AUC-ROC function the score of our model is:

```
(119678, 1000) (119678, 6) (39893, 1000) (39893, 6)
935/935 [==============================] - 544s 561ms/step - los
s: 0.1265 - auc: 0.8648 - val_loss: 0.0543 - val_auc: 0.9765
```

It is therefore evident that the score of 0.9765 is better compared to the one obtained with the FastText model and with the Word2vec one.

It will be probably possible to further develop our model by

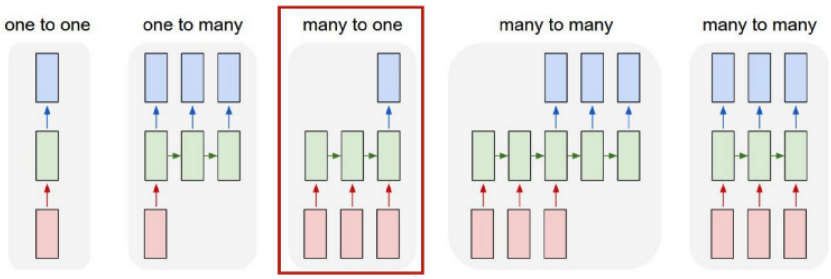
- using a multichannel CNN which would combine looking at different length (e.g. kernel size of 3, 5, and 7) of sentences at a time.
- Tuning the model layers and hyperparameters to improve the performance

Long Short-Term Memory (LSTM)

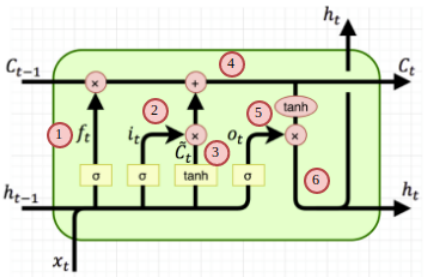
Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNNs) capable of learning order dependence in sequence prediction problems. This is a behavior required in complex problem domains like machine translation, speech recognition, and more. LSTMs are a complex area of deep learning. It can be hard to get your hands around what LSTMs are, and how terms like bidirectional and sequence-to-sequence relate to the field.

Let's first understand how RNNs works. In a traditional NN, we assume that all inputs (and outputs) are independent of each other. They don't share features learned across different positions of text. This might be an issue for sequential information such as text data or time-series data where each instance is also dependent on the previous ones. RNNs are called recurrent because they perform the same task for every element of a sequence.

There are mainly 5 structures that can represent an RNNs. In our case, we will be using a "Many to one" relationship, as we are speaking about a classification problem. Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state.



Long Short-Term Memory networks — usually just called "LSTMs" — are a special kind of RNN, capable of learning long-term dependencies. All RNNs have the form of a chain of repeating modules of neural networks. LSTMs also have this chain-like structure, but instead of the hidden layer we have something called LSTM cell and we have another connection that runs through all the time steps along with the hidden state. This structure can be summarized by the image below:



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Let's look at the 6 steps:

1. This is the forget gate which is responsible for how much to forget and since it passes through a sigmoid function, it will give a value of 0 to 1 which is the amount of memory to be retained from the previous cell state.

2. This is the input gate which is responsible for how much new information is to be added to the cell state. Like forget gate this will also give a value of 0 to 1 which is the amount of new memory to be added.

3. This is the creation of a new candidate vector/ cell state.

4. This is where the cell state is updated which is a combination of the previous cell state and the current cell state, the contribution of each is controlled using the forget gate and input gate respectively.

5. This is the output gate which is responsible for what part of the updated cell state is to be remembered in the hidden state having a value between 0 and 1.

6. This is the updated hidden state which will be the input for the next cell and is based on the current cell state controlled by the output gate

We will try to use the LSTM model to make our classification and identify the level of toxicity of our target text. In order to do so, we will:

1. Pre-process our dataset to clean the main data text issues.
2. Create the LSTM model
3. Train the model
4. Evaluate the model
5. Predict our train test

Data Pre-processing:

The preprocessing for the LSTM model is pretty much the same as the CNN. In this optic, we will:

- Initialize a class;
- Updates internal vocabulary based on a list of texts.
- Tokenize our vocabulary, by transforming each words of the vocabulary in a token

The output of this will be:

```
array([  101, 10930,  7743, 14855,  3627,  2003,  2062, 10514,  9468,
        2229,  3993,  2059,  2017,  1005,  2222,  2412,  2022,  2054,
        2015,  2039,  2007,  2017,  1998, 22650,  2017,  6517,  9587,
       11263,  3600, 21369,  2323,  7743, 14308, 24471,  9004,  9072,
        2594,  2317,  5344,  1998,  2131,  2017,  2000,  3610,  2026,
        4632,  2017,  4364,  5305,  2368,  2033, 14855,  3627,  2003,
        2055,  6620,  1999,  4830,  2189,  2158,  2123,  2102,  4487,
        4757,  2008,  4485,  2006,  2032,  1998, 24218,  2003,  3308,
       21388,  2078,  2066, 10722, 19498,  2002,  2001,  1037,  2567,
        2205, 11263, 18009,  2078,  2317,  3337,  2131,  2477,  2157,
        2279,  2051,  1010,   102,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0]))
```

Model Initialization:

In Keras, the easiest way to define a model is to initiate a Sequential model class and keep adding required layers. In this NN model, a new parameter called dropout is being used:

- Dropout: Dropout is a technique for addressing the problem of overfitting. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. This hyperparameter is introduced to specifies the probability at which outputs of the layer are dropped out.

In general, for binary classification, we can have 1 output units, use sigmoid activation in the output layer and use binary cross-entropy loss.

For multi-class classification, we can have N output units, use SoftMax activation in the output layer and use categorical cross-entropy loss.

For multi-label classification, we can have N output units, use sigmoid activation in the output layer and use binary cross-entropy loss.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 128)         2560000
_________________________________________________________________
lstm (LSTM)                  (None, 128)               131584
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 774
=================================================================
Total params: 2,692,358
Trainable params: 2,692,358
Non-trainable params: 0
_________________________________________________________________
None
```

Train the model:

The compiling and training/fitting code is also pretty much the same as the CNN model. Before starting to train the model, we need to configure it. We need to mention the loss function which will be used to calculate the error at each iteration, the optimizer which will specify how the weights will be updated, and the metrics which is to be evaluated by the model during training and testing. The parameters we will be using are the same as the one used in the CNN.

Evaluate the model:

The score of our AUC-ROC curve is 0.9759, which is just slightly below the one of the CNN model, but anyway performing well compared to our Benchmark model.

```
(119678, 1000) (119678, 6) (39893, 1000) (39893, 6)
935/935 [==============================] - 2910s 3s/step - loss: 0.1444 - auc: 0.8239 - val_loss: 0.0524 - val_auc: 0.9759
```

Are of further improvements could be:

- Stack more than 1 LSTM layers
- Hyperparameter Tuning for epochs, learning rate, batch_size, early_stopping

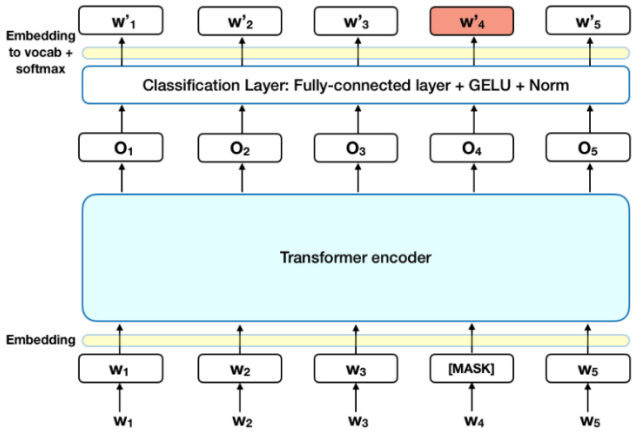
Bidirectional Encoder Representations from Transformers (BERT)

BERT (Bidirectional Encoder Representations from Transformers) is a recent paper published by researchers at Google AI Language. It has caused a stir in the Machine Learning community by presenting state-of-the-art results in a wide variety of NLP tasks, including Question Answering (SQuAD v1.1), Natural Language Inference (MNLI), and others. BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to language modelling.

BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. In its vanilla form, Transformer includes two separate mechanisms — an encoder that reads the text input and a decoder that produces a prediction for the task. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary. The detailed workings of Transformer are described in a paper by Google.

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires:

- Adding a classification layer on top of the encoder output.
- Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
- Calculating the probability of each word in the vocabulary with softmax.



The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. As a consequence, the model converges slower than directional models, a characteristic which is offset by its increased context awareness (see Takeaways #3).

We will overall implement this structure on our toxicity problem. The main steps we will be applying are:

1. Pre-process our dataset to clean the main data text issues.
2. Creating Train and Validation Data loaders
3. Load pre-trained BERT and set fine-tuning parameters
4. Tuning the BERT model
5. Evaluation

Pre-process dataset

First, we have cleaned the data:

- Cleaning the HTML tags using BeautifulSoup and
- Removing non-alphanumeric data

Then, we have prepared the data for BERT model as an input:

1. The first token of every sequence is always a special classification token ([CLS]) and we separate the sentences with a special token ([SEP])

2. Input must be split into tokens, and then these tokens must be mapped to their index in the tokenizer vocabulary.
3. Padding the input tokenized index sequence
4. Creating an attention mask

Creating Train and Validation Data loaders

First, we split the pre-trained dataset into train and validation. Then, we convert all the datasets to torch types so that it can be used by PyTorch. Finally, we will create the train and validation data loaders which can be used to generate batches of data while training/prediction.

Loading pre-trained BERT and setting fine-tuning parameters

First, we load the BERT model for sequence classification. We set the output neurons to 6 as we have 6 toxic types to be predicted for as yes or no. Next, we get weights for various layers and put them into a single list. Once that is done, we separate weight parameters (which needs to be updated) from bias, gamma, and beta parameters (which don't need to be updated).

Tuning the BERT model

This is the phase where we fine-tune the BERT model on our dataset for as many epochs as required and validate the model performance on Validation data. We will be using the BCEWithLogitLoss function for calculating multi-label loss between predicted and actual values. This is similar to adding a sigmoid function at the end of the network and calculating binary-cross entropy loss like in Keras we did for LSTM and CNN.

As we are using a macOS with only the possibility to use CPU memory to train our model, even with only 2 epochs, the training time is to consistent to be able to train the model like that. This is the reason why we have decided to use Google Colab and GPU technology. Although the training time has been still consistent, it has reduced drastically the performance of training our model.

Evaluation:

The results of our model are:

```
Train Loss: 0.0015603492090074198
Valid Loss: 0.0013186882466317675
```

Eventually we can think of improving our model by:

- Try different versions of BERT - RoBERTa, DistilBERT and ALBERT
- Hyperparameter Tuning for epochs, learning rate, batch_size, early_stoppinng.

Conclusion

In this project we have applied the most common NLP algorithms to our dataset to analyze the level of toxicity of the text comments extracted from Wikipedia.

We have applied the simplest NLP model, Bag of Words, that is at the basis of any text classification problem. Surprisingly, the algorithm used has performed extremely well with a 95%, compared to other algorithms like FastText (where the AUC-ROC curve score 77%).

From this we can conclude that complex algorithms are not necessary the best for any type of problems. The importance of analyzing the dataset provided is really, really important before deciding the strategy to be used.

This lead to the importance of data pre-processing, which is in NLP one of the main ingredient for the success of a model and for its replicability, sustainability and efficiency in predicting target classes.

Another important aspect to be considered is also the importance of model tunning and hyperparameters selection. By only applying algorithms without trying to optimizing it to the dataset we have and to the goal we are trying to achieve, this may lead to poor performance. For this reason, in each of the model used, we have actually identified margin for improvements.

Overall, the best model among the one used is probably the CNN one. This not only for the AUC-ROC score obtained, 0.9765, but also on a time efficieny proscpective. Model like the BERT one required a computational power that may be relevant for more complex problems, e.g. Question Answering, but not necessary for our multi-class classification problem.