Word2Vec with Simple Neural Networks For Text classification

Harsha Vardhan, Khurdula.

CS59000 Natural Language Processing

Professor: Jon, Rusert.

5th of April 2023

Introduction

The current task at hand involves using Word2Vec embeddings in combination with a simple neural network, in this scenario it will be a *Multi Layer Perceptron*. The objective is to explore how Word2Vec embeddings can be implemented in a task for Text Classification.

Embeddings in general, is a logical representation of text and their relationships captured and represented as a dense array. This is very useful for text classification.

Word2Vec is generally referred to as a module but in reality it is an algorithm which can be used to predict a word based on surrounding words! Word2Vec uses two main algorithms for training, *Continuous Bag of Words* and *Skip-Gram* algorithms. The result? Are embeddings that are a result of training upon a very large corpus of textual data.

Data Preprocessing

For this task, in order to generate word embeddings we need to perform some preprocessing to reduce noise within the data, and I did the following:

- Case Consistency: Transform the entire case into lower case, for case consistency.
- Capturing only Alphabetical text: Discard everything except textual words, such as punctuation, integers, etc. However Alnum words are still kept intact.
- Tokenization: Divide the entire sentence into a list of words.
- Token Selection: Select tokens from the list with length within specific range.

All of this can be simply performed using a simple method within gensim.util.simple_preprocess

Word2Vec Model Training

Before training a Word2Vec model in order to generate word embeddings, I had to decide on certain factors which would determine the dimensionality, and contents of the word embeddings. Here are my hyper parameters for training a Word2Vec Model:

- Embedding Size: I decided to set the embedding_size parameter to the model as 100, which basically results in each embedding being of size 100.

- Window: Now that I mentioned Word2Vec aims to look around for surrounding words to understand context, the window size determines how many words have to be considered for the current word. I decided to go with window = 5. Which would then mean that the model has to look 5 words before the word at consideration and 5 words after it, to grab the context.
- Minimum Frequency: There can be many tokens with their frequency within entire corpus being 1 (they only occur once.) which are not very useful while training and act as additional noisy data, by defining a threshold i.e. min_count (parameter to the Word2Vec) Model I intend to consider the tokens which meet the threshold and have a minimum frequency of 3.
- Total Workers: This is a self explanatory parameter that determines how many threads can be used in order to train the model. The larger the training data, the longer it would take for the model to train, having more than just one thread would fasten this process. I decided to have 4 threads for this task of simultaneous model training.

Using this Word2Vec model, I generated an embedding of size 100 for each comment within the dataset, although I've not experimented with the change of classification metrics of our Neural Network with change in value of embedding size, I will try and do the same in later stages of this paper. However, the resulting input to the NN would now be just an embedding and also the toxic label which is an integer and is 1 if the comment is toxic 0 otherwise.

Training a Multi Layer Perceptron

The Multi Layer Perceptron, or MLP has to be trained on the newly created vector of embeddings. One challenge I faced while trying to train any new instance of the MLP was to decide on the architecture of the Hidden layers if they were to be greater than one in count.

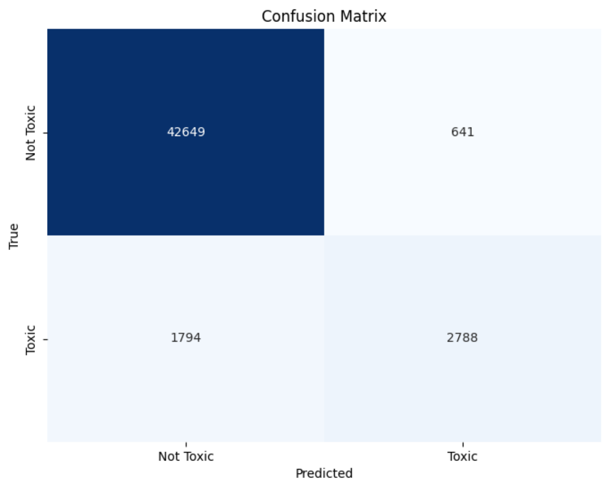
Traditionally it is assumed to be a good practice to decrease the number of neurons in each layer as we move forward within the neural network (each forward pass). This is done, because it tends to create a hierarchical feature extraction i.e. the initial layers within the NN tend to capture low level features very well, and later level features combine these to form high level layers.

However, I've found it to be the case that this approach is not very helpful for the current set. Hence I decided to go with a simple equal sized hidden layer architecture, within which each hidden layer has the exact number of 'n' neurons.

With that being said, I trained each of the MLP model variants i.e. A model with One hidden layer, a model with two and then three, respectively. I also tested it on the same training set to define a baseline of metrics. Here are my observations:
For Multi Layer Perceptron with One hidden layer:

Class	Precision	Recall	F1-Score
Not Toxic	**0.96**	**0.99**	**0.97**
Toxic	**0.82**	**0.61**	**0.70**
Weighted Avg	**0.95**	**0.95**	**0.95**
Accuracy			**0.95**

Classification Metrics for MLP with 1 hidden layer.



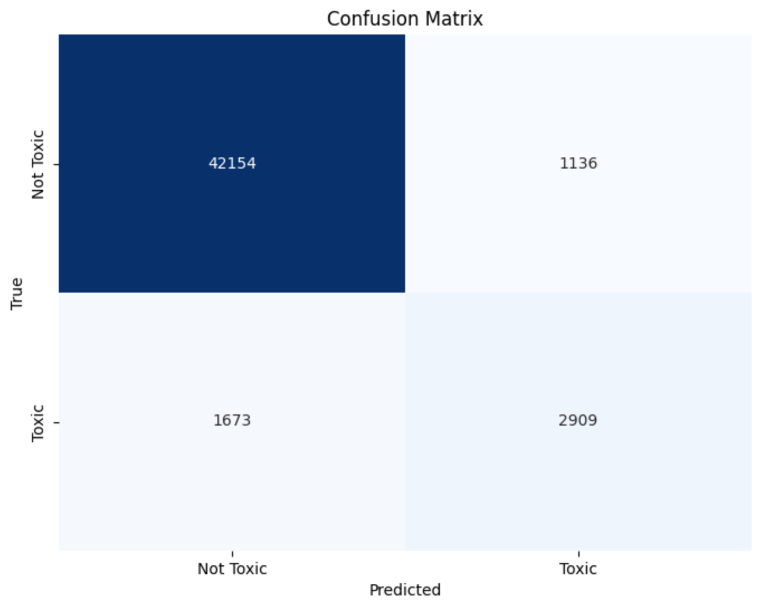
Confusion matrix of MLP with one Hidden Layer

For Multi Layer Perceptron with Two hidden layers:

Class	Precision	Recall	F1-Score

Not Toxic	**0.96**	**0.97**	**0.97**
Toxic	**0.72**	**0.65**	**0.68**
Weighted Avg	**0.94**	**0.94**	**0.94**
Accuracy			**0.94**

Classification Metrics for MLP with 2 hidden layer.

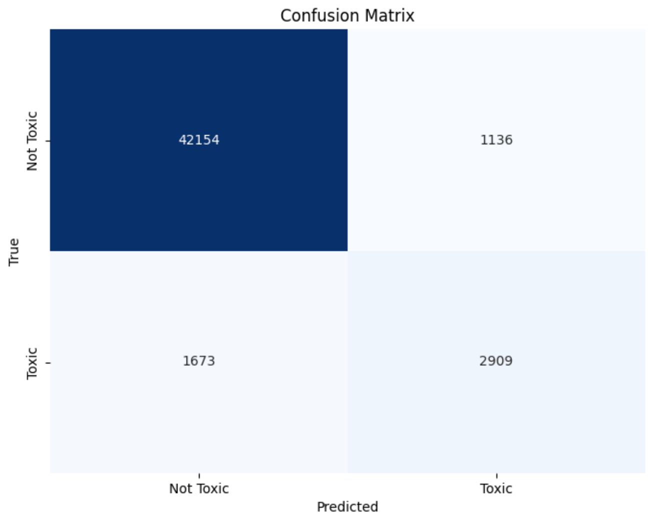


Confusion Matrix for MLP with 2 Hidden Layers.

For Multi Layer Perceptron with Three hidden layers:

Class	**Precision**	**Recall**	**F1-Score**
Not Toxic	**0.96**	**0.97**	**0.97**
Toxic	**0.70**	**0.64**	**0.67**
Weighted Avg	**0.94**	**0.94**	**0.94**
Accuracy			**0.95**

Classification Metrics for MLP with 3 hidden layer.



Confusion Matrix for MLP with Three Hidden Layers.

As we can see the difference in performance is very minute, however if we were to pick our highly performing model based on these metrics, then it would clearly be a MLP with one hidden layer.

Surprisingly I got the exact same classification for both layers two and three!! I believe the minute loss over time as the number of layers increase could be due to loss of information due to each forward pass.

Testing

To test this MLP we shall use an entirely different dataset, specifically separated for model testing. The goal is to test the model on this file and create an output file that will store the results.

The test file that I used was provided with the original dataset, that we used for all these hws however their labels were different the target variable had three distinct values for the testset -1, 0, 1 I assume these represent, not_toxic, neutral, and toxic. However our model does not have the ability to capture this. Hence I manually manipulated these labels. To be only 1s and 0s.

The output is produced and stored within a csv file named output.csv file within the same directory as one might be in. On the whole I still believe that Logistic Regression is the best model suited for this task, if we are to consider its performance against the current Simple Neural

Network at hand it clearly outperforms our MLP. This is because LR had a better balanced score for recall than MLP.

The produced output.csv has various features, comment_text which is the original comment made by the user, preprocessed_text: which is the tokenized list created after cleaning the original comment. Embedding: the vector of word embedding numerical values. Non_toxic_probability: a probability score for the comment to be non toxic. Toxic_probability: probability score for the comment to be toxic. Toxic_label: determines if a comment is toxic or not, 1 if yes 0 otherwise.

NOTES:

In order to run this scripts please use the following commands:

```
>python
>>>  from mlp import MLP_Classifier
>>> obj = MLP_Classifier()
>>> model = obj.train_MLP_model(<path to training file>, <total_number_of_hidden_layers)
>>> obj.test_MLP_model(<path to test file>, model)
```