Hate Speech Detection - Model Design

Kareem Ehab Kassab 900182771
Computer Science and Engineering
The American University in Cairo
kareemikassab@aucegypt.edu

Mahmoud Elshinawy 900183926
Computer Science and Engineering
The American University in Cairo
mahmoudelshenawy@aucegypt.edu

I. Introduction

Hate Speech is an important text classification problem that is being addressed in multiple paradigms including online communities, social media, and video games just to name a few. This has gained importance because people are increasingly engaging in online social activities, and just like normal social activities, interactions between people need to have boundaries and rules that are to maintain such interactions. Therefore, with the growing of such online communities, grew the need to contain, and followingly, auto-detect hate speech to limit its usage in these social networks.

Through several past phases, we preprocessed data, tested with different models and hyper parameters and reported their accuracy, and finally decided on Neural Networks to be the best. In this phase, we are implementing our Neural Network design in line with a client application to simplify the interaction with the model. In this report we discuss the implementations, pivotal corrections, and the results of the final model.

II. Model Implementation

A. Preprocessing and Text Embedding

As explained in the previous phases, the first stage of the preprocessing includes that the Dataset is fed into the preprocessing model: special characters, stop words, numbers are removed, and words are returned to their stem forms- this has not changed.

The word2vec part was changed in this phase as per the professor's comment. Now, The text embedding is the first layer of the neural network as it acts as. a true embedding layer that is optimized against the output (hate / not) not against the context. This first layer has a number of neurons n=4 that equals the 4th root of the number of the words m in our corpus/dictionary- which is actually the 4th root of 31011 which is around 13.

The following diagram (taken from the slides) explains the concept generally. In our project, specifically, we don't have the yellow inputs beside the words. The words in blue represent the input layer and these are around 31011 inputs representing the words in the corpus. The embedding layer is the 4th root of the number of words, and ths, we chose to have 13 neurons, and these have an activation function of tanh. They then pass to the hidden layer of 5 neurons with a relu activation function, which will be, furthermore, explained in the neural network binary classifier section.

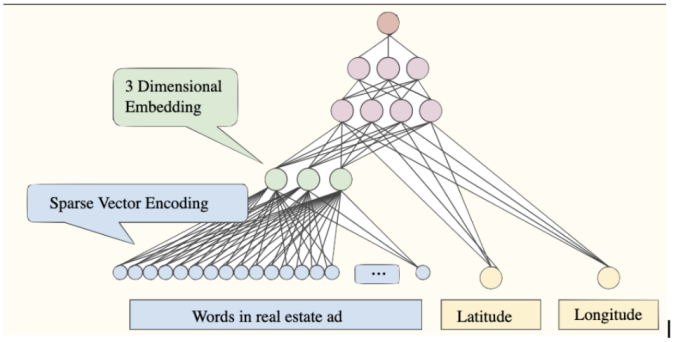

Fig. 1. Neural Network Design.

B. Neural Network Binary Classifier

The heart of this phase was the neural network binary classifier model. In this phase, we worked on the implementation of the model, in addition to fine tuning it, testing it, solving several problems associated with it, and experimenting with different

parameters, and reiterating over it to bring the best results.

The structure of our neural network consists of: the input layer: a layer with a number of neurons equal to the number of words in our corpus, the embedding layer: 13 neurons that perform the text embedding function. Hidden layer: 5 neurons that feed the final classifier in the output layer. The output layer: our final binary classifier with 2 outputs.

We experimented with a variety of activation functions, and decided on using the following for each of our model: for the embedding layer, we are using a tanh activation function, meanwhile, for the hidden layer, we are using relu functions (does not have any zero slopes and handles negative numbers to avoid any errors), and for our final output binary classifier we are using a sigmoid function.

For the loss function, we switched from using the binary cross entropy to using the focal loss in order to account for the imbalance in our data. Binary Cross entropy might not be the best choice in the presence of class imbalance, and the data was skewed towards one class over the other in a 60/40 ratio. The learning rate was also changed from 0.001 to 0.5 as the 0.001 never converged in our experiments, and this was done through trial and error. The optimizer that we decided to use was the ADAM optimizer; we used this instead of gradient descent as it did not converge. ADAM uses moving average concept for better convergence on weights. We decided to use batch training instead of updating the weights upon training each instance; we decided to use a batch size of 1000 by trial and error, this was the best so that the model converges. We used joblib to save the model instead of pickle as it yielded much less file size (13mb instead of 400mb). Also, we used joblib to reload that model in django for the web app that is to be explained in the next section.

$$FL(p) = \begin{cases} -\alpha(1-p)^{\gamma} log(p), & y = 1 \\ -(1-\alpha)p^{\gamma} log(1-p), & otherwise \end{cases}$$

Eq. 8: Final Focal Loss(Alpha Form)

Fig. 2. Focal Loss Formula.

For the optimizer that we decided to use was the ADAM optimizer; we used this instead of gradient descent as it did not converge. ADAM uses moving average concept for better convergence on weights. We decided to use batch training instead of updating the weights upon training each instance; we decided to use a batch size of 1000 by trial and error, this was the best so that the model converges. We used joblib to save the model instead of pickle as it yielded much less file size (13mb instead of 400mb). Also, we used joblib to reload that model in django for the web app that is to be explained in the next section.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Moving averages of gradient and squared gradient.

Fig. 3. ADAM Formula.

III. Web Application

As discussed before, both the backend and frontend of the web app were implemented in python Django. The web application was hosted on Amazon Web Services (AWS) Server using an EC2 instance and Apache server. We decided to use EC2 for several reasons: it allows us to host without buying a certain domain; In addition, AWS is reliable, fast, and can be accessed everywhere.app utility -> django, ec2 instance for hosting remote server.

A. Backend

In our MVC model, the backend/model is responsible for receiving the data input string (coming through the API from the frontend/View), preprocessing the input data, predicting the class, and then passing it to the controller to be passed to Django's View through the API again.

For the preprocessing part, the backend code cleans the input strings from stop words, numbers, special characters, html tags. Then a tokenizer tokenizes the words, and then we put them back into their stem/root forms. The dictionary is saved in the frequency_encoded_words.txt.

Embedding is now a layer in the model. Thus, in the training step, the weighs are assigned to the words by that layer. During the testing, when a new string is being input, the values in its vector dimensions corresponding to these words are set to 1 and the rest are zeroes. For the prediction step, the numerical representation vector is fed into the neural network binary classifier that at last outputs whether the fed text is hate speech or not.

Since we implemented the online learning bonus, the backend increments a counter in the counter.txt file so that it retrains after every 10 entries. Also note that there are checks to check if there are new words to the corpus in the entries, and these are appended at the end of the corpus if they exist.

We also tried the 'adaptive' learning rate option, but we did not get any better performance than we got with the constant 0.0001 learning rate. Fig. 7. Shows the performance of the model with adaptive learning rate initiated at 0.0001.

B. APIs

Between the Django implementations in the back end and front end, the APIs used in between were simply mainly GET. the user inputs an input string to be tested, and this is passed to the back end using a GET request. After the backend is done with preprocessing and prediction, the output class 0 or 1 is returned to the front end using a GET response.

C. Frontend

This simple web interface is built using Django. The Django server with the database (as per in our model diagram) is being communicated with through sub-URLs (endpoints) to navigate between URLs, and this is what connects the front end to backend. This was all possible because of using the Django framework.

The starting interface is simple with a main textbox and submit form to input the string. For the results interface,depending on the result a user may get a red screen with a "hate speech" message and an image showing that this is class 1 (hate/offensive speech) or a green screen with a " not hate speech" message and an image showing that this is class 0 (not hate/offensive speech)

D. Data Flow

The Modified Data Flow has no text embedding as a word2vec before the Neural Network model. However, we decided to add the embedding as a layer in the neural network model. First, the Dataset is fed into the preprocessing model: special characters, stop words, numbers are removed, and words are returned to their stem forms. Then, in the updated version, we added the embedding as a layer in the neural network model. Then, such vectors are fed to the Neural Network Classifier model to train on. Finally, the trained weights are stored in the database for testing and predicting user inputs.

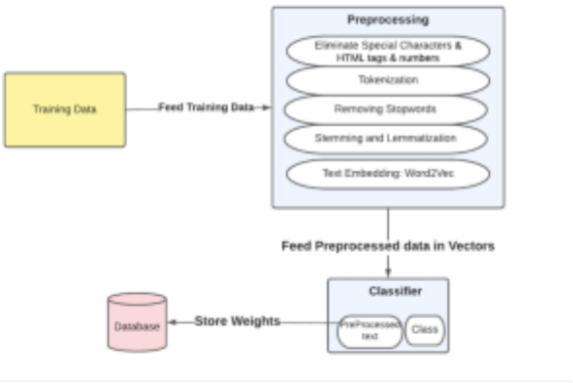

Fig. 4. Data Flow.

E. Bonus: Online Training

For the online training, we decided to retrain once every 10 entries. In the backend code, there is a txt file that maintains such a counter to keep track of the model usage. With each entry, we check for new words to be added at the end of the corpus; in addition, the new sentences are added as test samples with their predicted classes to be used in the next retraining session (which is held every 10 entries.)

IV. Model Performance

Our model turns to outperform the ready implementation model in terms of the recall and got a recall of 87 and 72 in class 0 and 1 respectively.)

```
In [308]: from sklearn.metrics import classification_report
          print(classification_report(y_test[0:], y_pred))
```

```
              precision    recall  f1-score   support

         0.0       0.81      0.87      0.84      7974
         1.0       0.79      0.71      0.75      5582

    accuracy                           0.80     13556
   macro avg       0.80      0.79      0.79     13556
weighted avg       0.80      0.80      0.80     13556
```

Fig. 5. Our Model Performance.

Neural Network Relu adam 0.0001 (Hidden Layer= 25)

```
27]: from sklearn.neural_network import MLPClassifier
     nn_model_reluadam0001 = MLPClassifier(hidden_layer_sizes=(25,),random_state=1,solver='adam',learning_rate_init=0.00
     nn_model_reluadam0001.fit(train_df.values,y_train.values)
     nn_results_reluadam0001 = nn_model_reluadam0001.predict(test_df.values)
```

```
28]: from sklearn.metrics import classification_report, accuracy_score
     print(classification_report(y_test.values,nn_results_reluadam0001))
```

```
              precision    recall  f1-score   support

         0.0       0.81      0.86      0.83     16018
         1.0       0.77      0.71      0.74     11094

    accuracy                           0.80     27112
   macro avg       0.79      0.78      0.79     27112
weighted avg       0.80      0.80      0.80     27112
```

Fig. 6. Ready Implementation Model Performance.