Sentiment Analysis on Tweets
Model and Utility Application Implementation

Amer Elsheikh
The American University in Cairo
Cairo, Egypt
amer.elsheikh@aucegypt.edu
900196010

Abdallah Abdelaziz
The American University in Cairo
Cairo, Egypt
Abdallah_taha@aucegypt.edu
900196083

In this report, we examine go through implementation choices, APIs, preprocessing components, and retraining method of our final model.

1. Model Implementation Choices

As suggested in the previous phase, we decided to implement a **neural network with an embedding layer**. The embedding layer takes the one-hot-encoded tweets and embeds them into a vector of size 10. We discuss the details of the architecture of the model below.

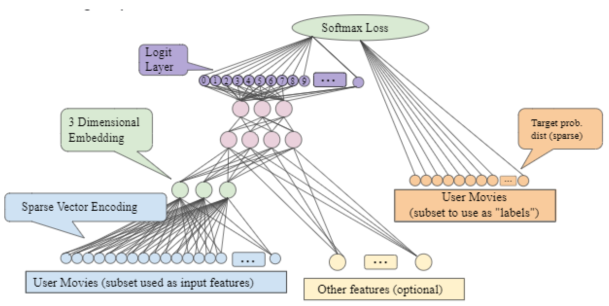
General Architecture

As mentioned, the model is a neural network with an embedding layer. The embedding layer has 6839 input nodes followed by a hidden layer that has 8 nodes. The hidden layer is then follwed by the output layer for the word embedding which contains 10 nodes. This means that a tweet is now represented by a vector with 10 dimensions. The size was decided based on Google Machine Learning Crash course and following the empirical rule

$$dimensions = \sqrt[4]{possible\ values} = \sqrt[4]{6839} = 9.09$$

After experimenting with many different structures of the embedding layer, the mentioned architecture yeilded good resutls and hence was the one implemented. The sigmoid function was used as the activation function for all embedding layers and it showed good resutls.

The output of the embedding layer is then joined with the time input represented by 7 one-hot-encoded features for the week-day and one feature indicating the hour at which the tweet was posted. The hour feature is normalized by dividing it by 23. These nodes forms the 18 nodes layer which has sigmoid as its actionaion function. This layer is then followed by one hidden layer of size 4 with sigmoid as its activation function. The final layer is the output layer with 2 nodes and softmax as the activation function. The final architecture of the model follows the same concept as the architecture in the figure below.

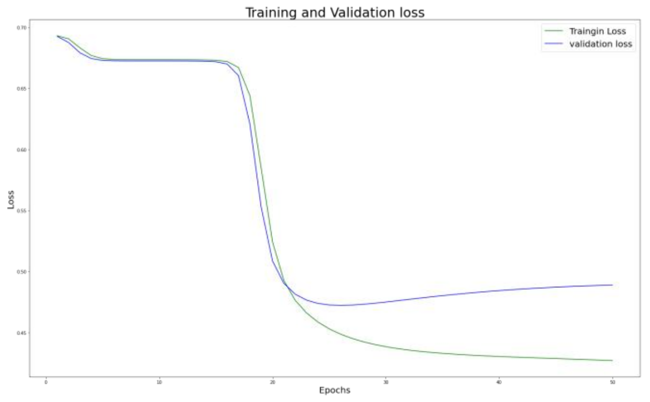

Softmax Loss
Logit Layer
3 Dimensional Embedding
Target prob. dist (sparse)
User Movies (subset to use as "labels")
Sparse Vector Encoding
User Movies (subset used as input features)
Other features (optional)

By experimentation, using two output nodes with a softmax activation function associated with the famous **softmax log likelihood as a loss function** yielded better results than using one output node with the sigmoid activation function. This choice does not change a lot in principle as the sigmoid function is a special case of the softmax activation function. This decision was made, however, based on experimentation and the performance of the models. In this case, the first node represents the probability of the sentiment to be negative, and the second node represents its probability to be positive.

Training the Model

Given that the dataset has approximately 1.5M tweets and that the input for the network is the one-hot-encoded tweet, training the model was quite challenging. Transforming the whole dataset before training was not possible due to memory space limitations. Hence, the training was divided into batches and each batch was transformed to the one-hot-encoded form just before fitting the model to the batch. Then the next batch is transformed, and the previous batch is discarded. This allows us to train the model without exceeding the memory limitations.

Validation Set
To avoid overfitting, a validation set was created and used to evaluate the loss function after each epoch. This was done to decide the point at which the model starts overfitting and hence the epoch at which the training should stop. The data was split such that 1,000,000 instances are used for training; 486,000 instances are used for testing and 10,338 instance were used for validating the model. The size of the validation set is relatively small due to memory and computational limitations. The split is around 66.8% for training 32.5% for testing and around 0.7%. You can find below the graph of both validation and training loss. We stopped training when the validation loss started to increase.



Optimizer
We decided to use an optimizer to make the learning faster and to make sure the model has good chances to converge to a minimum. To do so, we implemented the Adam optimizer. The optimizer combines the advantage of both the Adaptive gradient descent and the Root Mean square propagation. In short, the optimizer adapts a suitable learning rate for each weight parameter based on its gradient. The optimizer proved to be effective in our problem as it is suitable for problem with noisy data.

We experimented different **learning rates** to give to the optimizer, and we found that **0.5** gives the best metrics as it helps the optimizer converges quickly and accurately.

2. Utility Application Choice

As mentioned in the previous phase, we will be using Django to implement our utility application. The web framework allowed us to achieve the client/server separation through its compatibility with REST API. It also supports multiple features that are very useful for our purpose. Django also provided us an easy solution to manage the dataset and easily create the user flow for our project. The default database for Django, and the one we used, is SQLite 3. This allows us to store the data we receive on the database and use them to retrain the model whenever needed.

3. APIs

The model is deployed using a Client/Server architecture where the model is only deployed in a central server, and all users (clients) communicate with that server through API requests communicating the data of the tweet and receiving the sentiment as a response.

In our case, the browser represents the client, when the user types a tweet in the textbox, the API posts the input to the server. The server then predicts the sentiment and sends the results back to be rendered on the html file shown in the browser. In general, this API handles the POST and GET requests.

Another API is the REST API which is used to communicate with the database. When the user enters data, the API stores the data in the database. The API also retrieves data from the database when data is needed to retrain the model.

4. Preprocessing Components

As explained in the previous phase. The preprocessing is done as follows.

The user will be asked to fill four required fields: the tweet, its posing day of the week, its posting hour of the day, and his suggested sentiment for it. The suggested sentiment will only be used for the retraining and online learning, and it will not affect the predicted sentiment in anyway.

Now, we will convert the day of the week to its one hot encoding as a vector of 7 values (6 are zeros and one is one). With respect to the hour, we will take the hour (between 0 to 23) and divide it by 23 to normalize it.

Lastly, for the tweet, we will do exactly the same preprocessing of phase 2 using help of nltk, re, contractions, and other libraries. As a quick recap, we will

1- Remove all tabs, mentions, newlines, punctuation marks.
2- Adjust contractions and lowercase all letters.
3- Tokenize the words
4- Lemmatize the words
5- Remove stop words
6- Handle negation using bigrams

After that, we will examine the remaining words along our **dictionary of words** which we got from the model. If the word exists in our vocabulary, we will add it a dictionary representing the tweet; if it does not, we will simply drop it which is one **limitation** of our model.

The day, the time, and the dictionary of the tweet will be then passed to the model. The model predicts the sentiment by converting the dictionary to a one-hot encoding representation. At this point, the data will be ready to be input to the model and the model predicts its sentiment by running it through the neural network.

All this preprocessing is done in the **backend** for purposes of scalability and maintainability, and it is more convenient for the preprocessing to be near the model in the backend

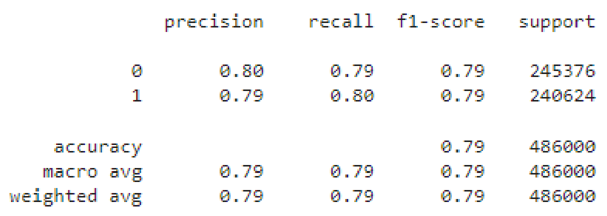
5. Retraining Methods

As mentioned, Django applications come with an SQLite database which we can use to store and retrieve data which can be used to retrain the model. Retraining the model is simple. All we need to do is the run "fit" the member function of our loaded model. The model will use the new sample to update the weights.

The retraining happens whenever the amount of new data reaches a prespecified threshold. When this threshold is met, the new portion of the data is retrieved from the database and the same preprocessing described is applied. The model is then trained on the new data and the weights are updated accordingly.

6. Results

The model proved to be highly accurate as it achieved an accuracy of 79% with a precision of 80% and 79% for negative and positive classes respectively. Below is the report for the test on the testing data, 486,000 that were not used in the training. The good performance of the model is clear.

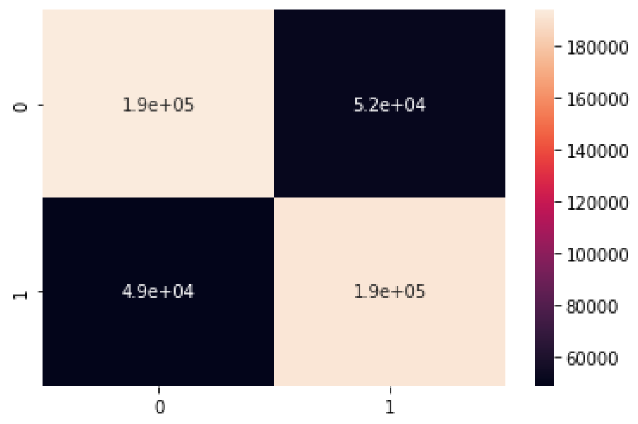


```
              precision    recall  f1-score   support

           0       0.80      0.79      0.79    245376
           1       0.79      0.80      0.79    240624

    accuracy                           0.79    486000
   macro avg       0.79      0.79      0.79    486000
weighted avg       0.79      0.79      0.79    486000
```

For a more visual result, we can find below the heat confusion matrix of our test data.



7. Resources

Google Machine Learning Crash Course: Embeddings, https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture

Front End Resources:

https://www.w3schools.com/w3css/w3css_cards.asp

https://simpleisbetterthancomplex.com/article/2017/08/19/how-to-render-django-form-manually.html