Introduction

Sentiment classification in music is a common problem with numerous applications. Many companies in many fields are interested in satisfying their audience with the right music. One central feature is the positiveness of a song. In this project, we try to create a model which predicts if a song's lyric is positive or not.

Valence: Describes the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

Steps:

1. Define the problem.
2. Get the data.
3. Explore and analyze the data.
4. Prepare data.
5. Explore the models:
    a. Baseline model.
    b. Advanced model.
6. Fine-tune the selected model.
7. Results and conclusions.

Problem definition

Prediction if a song's lyric is positive or not.

Get the data

We choose to use Kaggle data, which has the next attributes:

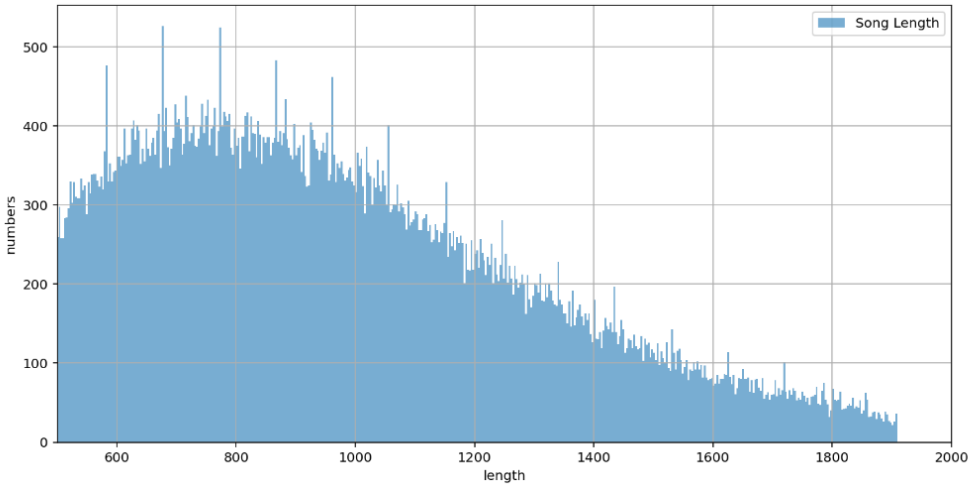
Number of instances: 158,353.

- **artist** - artist name (14,691 unique values).
- **seq** - song's lyrics (135,991 unique values).
- **song** - song title (99,031 unique values).
- **label** - Spotify valence feature attribute for this song (continuous variable).

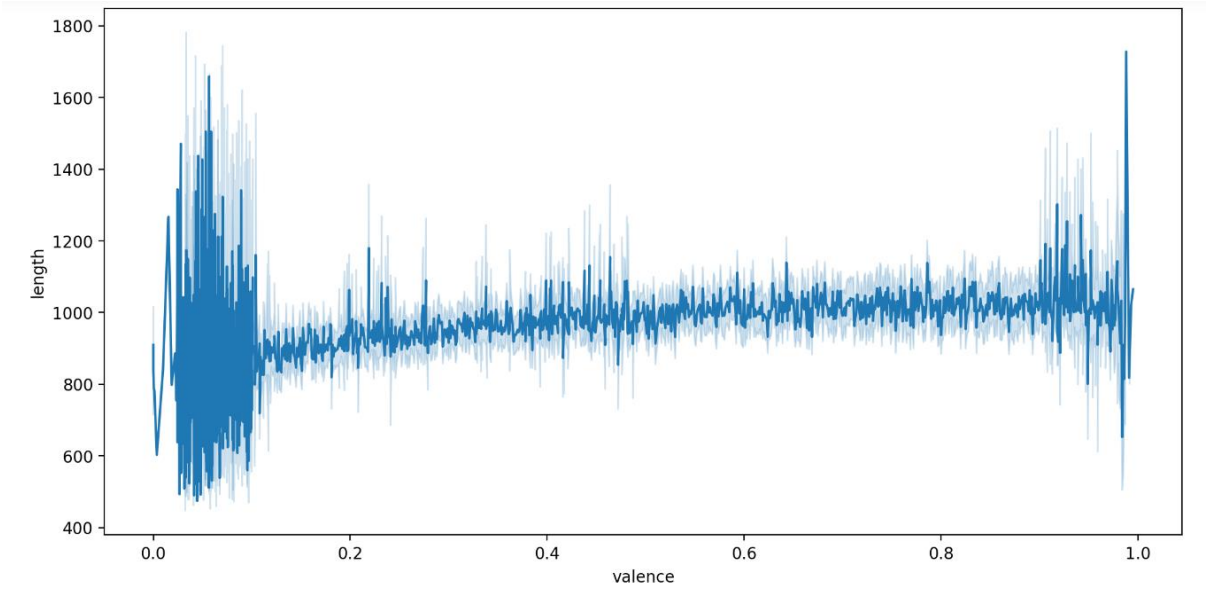
Source

Explore and analyze the data

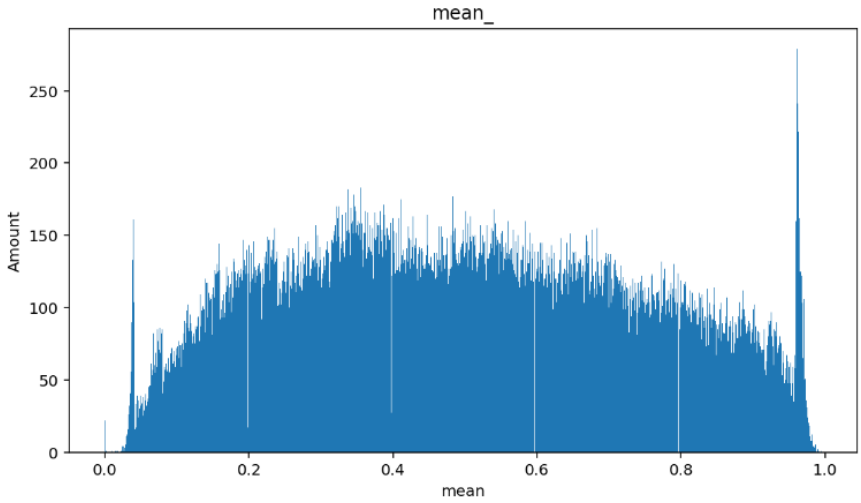
Distribution of song length (between 500 to 2000 words):



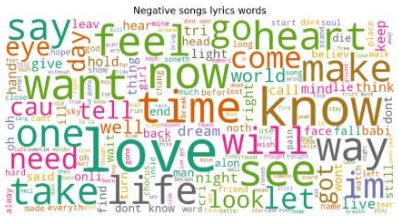
Valence – lyrics' length (words) correlation:



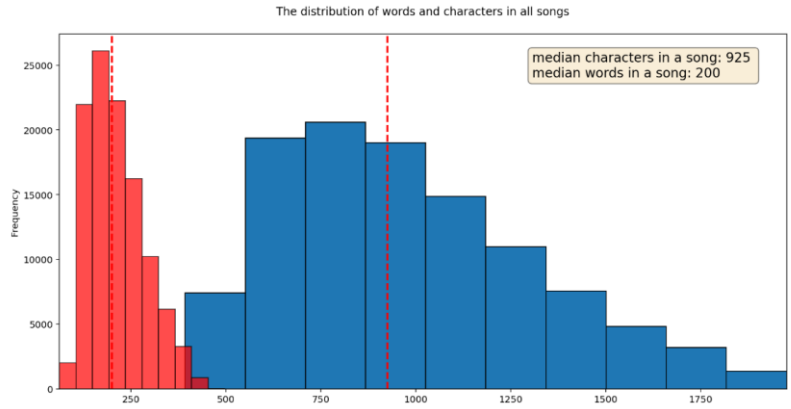
Distribution of valence:

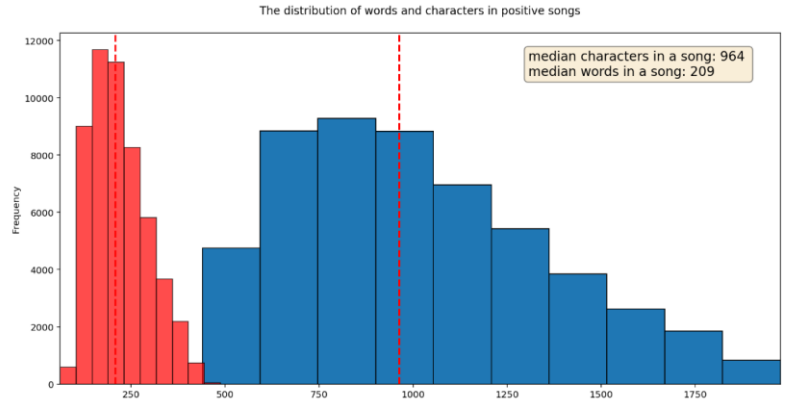


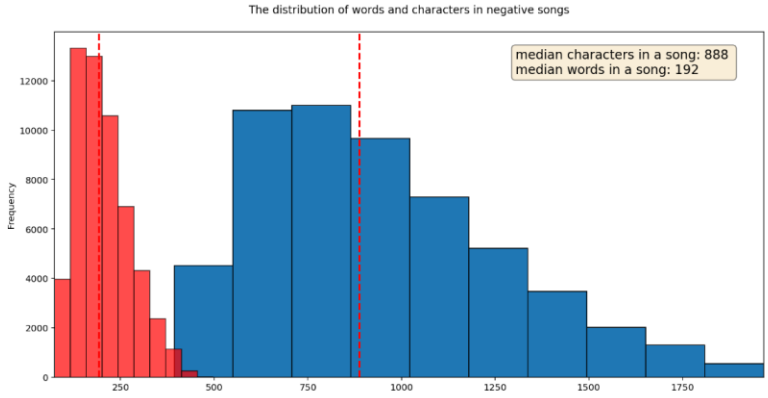
Frequent words (All data, Positive songs, Negative Songs):



Distribution of words and characters (All data, Positive songs, Negative Songs):







Prepare data

In this section, in order to build the best model, we manipulate data in difference ways:

- Drop Duplicates.
- Lowercase all words.
- Remove Punctuation.
- Remove Stopwords.
- Remove digits, and characters that noised our data.
- Applying Stemming.
- Create new columns:
    - Mood.
    - Song Length.
- Select our data for the model according to the range of song length and valence.
- Down sampling the data for equal sample for the target.

To improve our model, we came back to these steps often after moving on to the next steps.

Drop Duplicates:

Drop all duplicated instances according to the lyric column.

Lowercasing, remove punctuation:

Using 'String' library.

Remove digits, and characters that noised our data:

While explore our data (by code and by the CSV file), we found out some digits/words/patterns/characters that cause to our data to be noisy.

Remove Stopwords:

Using the default English stopwords list of 'nltk'.

- After trying different stopwords list, we got a conclusion that Using the default English stopwords list of 'nltk' is the best for our model.

Stemming:

Using the SnowballStemmer of 'nltk'.

Create new columns:

- Mood – Model's target column. We manipulate the continues 'valence' column to be a binary column.
  we decide to consider a positive song which is valence is above 0.85, and a negative song under 0.15.

  At the beginning, we set our threshold to 0.5 (and trying 0.6, 0.7 and different thresholds), but after evaluating our model we understand that it not the best for our model. In this situation we actually forcing our data to be something else, when it not in all cases true.

  Hence, we decide to get rid from the middle area and keep the tails for our model.

- Song Length – We created this attribute to see the distribution of our data and find outliers.
  We saw according to the distribution that there are outliers, and decided to keep only the lyrics whose have between 500 to 2000 words.

Down sampling:

We wanted a represents sample data for our model. We had the next requires:

1. The ratio between 1 and 0 in the 'Mood' columns should be the same.
2. The sample should be random.
3. The sample should not be large because of memory limitations.

After filter instances in the previous steps, we checked the one between 1/0 has the minimum instances. We choose our sample to be 16,000, when number of instances with '1' value in 'Mood' is 8000, and accordingly the number of '0'.

Explore the models and Fine-Tuning:

- Baseline Models

  Setup:

  - X – lyrics columns after TFIDF.
  - Y – 'Mood' column (1/0).
  - TFIDF: max_features=1000.
  - train_test_split(x, y, test_size=0.3, random_state=42).

  Here, we tried to initialized different setup to improve our model, such as change the max features in TFIDF or set a different test size in the train_test_split function.

  Models:

We implemented various of models to find our best model.

- Logistic Regression.
- Linear SVC.
- Decision Tree.
- Random Forest.
- SVM.
- KNN.
- AdaBoost.
- Naïve Bayes.

The results are:

	LogReg	LinearSVC	DecisionTree	RandomForest	SVM	KNN	AdaBoost	Naive Bayes
accuracy	0.765416667	0.75625	0.661875	0.777916667	0.773958	0.5927083	0.7266667	0.76979167
precision	0.759643917	0.74842238	0.651540383	0.782417582	0.770976	0.554939	0.7265591	0.76110645
recall	0.762228839	0.75669928	0.665674181	0.757124628	0.766057	0.8507018	0.7086346	0.77243726
f1	0.765334167	0.75619514	0.661842579	0.777624451	0.773833	0.5676766	0.7263777	0.769752

Applying Cross-Validation and computing the mean and standard deviation of the performance measure:

The k-fold cross-validation procedure divides a limited dataset into k non-overlapping folds. Each of the k folds is given an opportunity to be used as a held back test set, whilst all other folds collectively are used as a training dataset. A total of k models are fit and evaluated on the k hold-out test sets and the mean performance is reported.

Repeated k-fold cross-validation has the benefit of improving the estimate of the mean model performance at the cost of fitting and evaluating many more models.

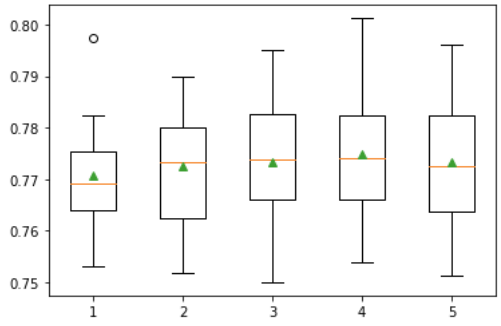
Measuring by comparing the distributions of mean performance scores under differing numbers of repeats.

There is a true unknown underlying mean performance of a model on a dataset and that repeated k-fold cross-validation runs estimate this mean. We can estimate the error in the mean performance from the true unknown underlying mean performance using a statistical tool called the standard error.

The standard error can provide an indication for a given sample size of the amount of error or the spread of error that may be expected from the sample mean to the underlying and unknown population mean.

As we can see, we succeed to coverage to better results of our data:

```
Cross Validation scores (Mean & Standard Deviation)
>1 mean=0.7707 se=0.004
>2 mean=0.7724 se=0.003
>3 mean=0.7733 se=0.002
>4 mean=0.7748 se=0.002
>5 mean=0.7732 se=0.002
```



Fine-Tuning:

For the reason Random Forest model gave us the best accuracy score, we chose to optimize it.

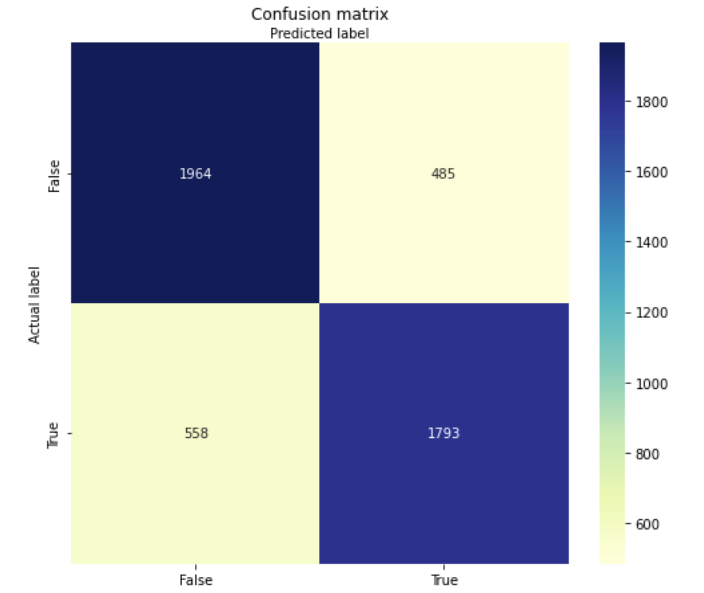
- Note: Because of memory and performance limitation we cannot run all permutation and even not all parameters. We select small sample of two parameters and run **Grid-Search** with them.

- max_depth = [None, 32, 64]
- n_estimators = [100, 128, 256]
- bootstrap= [True, False]

Best Parameters we got: {"bootstrap": false, "max_depth": null, "n_estimators": 256}

- For future you can try those parameters:
  - max_depth= [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None]
  - max_features= ['auto', 'sqrt']
  - min_samples_leaf= [1, 2, 4]
  - min_samples_split= [2, 5, 10]
  - n_estimators= [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]

<u>Random Forest with its best parameters:</u>



	Random forest
Roc-Auc	0.859613964
accuracy	0.782708333
precision	0.787093942
recall	0.76265419
f1	0.782432209

We can see improvement in all evaluation metrics.

- **Advanced Models**

Explore the models:
System limitations - lack of GPU, each model runs very slowly.
Advanced baseline – Neural network model:
Based on the article:
https://arxiv.org/ftp/arxiv/papers/1506/1506.05012.pdf

Before we present the models first, we will explain the methods we used and the motivation behind building the models.
For both models we first:

1. Loaded the data.
2. Separated the features("lyrics" column) and labels("Mood" column).
3. Pre-processed and Tokenized the data.
4. Used word embedding
5. Built each model
6. Evaluated the model and saved all that we have been required.

In the course we learned about **Word Embedding** A word embedding is a class of approaches for representing words and documents using a dense vector representation.
It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.
Instead, in an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space.
The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used.
The position of a word in the learned vector space is referred to as its embedding.
There are popular examples of methods of learning word embeddings from text such as Word2Vec, Glove.

LSTM is a special type of recurrent neural network. Specifically, this architecture is introduced to solve the problem of vanishing and exploding gradients. In addition, this type of network is better for maintaining long-range connections, recognizing the relationship between values at the beginning and end of a sequence.

Bidirectional LSTM (BiLSTM) is a recurrent neural network Unlike standard LSTM, BiLSTM adds one more LSTM layer, which reverses the direction of information flow. Briefly, it means that the input sequence flows backward in the additional LSTM layer. Then we combine the outputs from both LSTM layers in several ways, such as average, sum, multiplication, or concatenation.
BiLSTM is beneficial in some NLP tasks, sentence classification is between them. That way the meaning of each sentence will be clearer, and we might can predict what emotion came out of the song.

Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel.
During training, some number of layer outputs are randomly ignored or "*dropped out.*" This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "*view*" of the configured layer.

It is important to mention regarding the disadvantages of BiLSTM compared to LSTM, it's worth mentioning that BiLSTM is a much slower model and requires more time for training.
For both NN models we used we used The **Tokenizer class of Keras** which is used for vectorizing a text corpus. For this either, each text input is converted into integer sequence or a vector that has a coefficient for each token in the form of binary values.

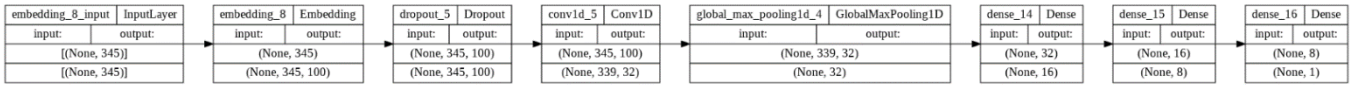
First prototype:

Word Embedding :

We download Glove's **pre-trained word embeddings in order to help solving our classification problems**.
We Loaded the Glove embeddings, and appended them to a dictionary. Then, used this dictionary to create an embedding matrix for each word in the training set. We got embedding vector for each word using "embedding_index". Words that isn't found, will be represented by zero.

Layers:

embedding_layer -
we created the embedding layer. We didn't want this layer to re-trained so we set "trainable" to false.



The weights are set to the embedding matrix as, "vocab_num_words"
is the size of the vocabulary with one added because zero is reserved for padding and `input_length` is the length of "max_num_tokens", the longest length of the tokens.
Dropout – Used for regularization.

Conv1D – Common in neural networks for text classification, used 32 filters and kernel size of 5.

GlobalMaxPooling1D-

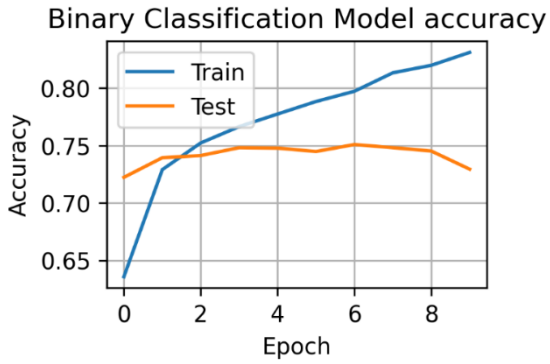
We Used global max pooling blocks as an alternative to the Flattening block after the last pooling block of the CNN.
And then we used few dense layers to make the network deeper. Deeper network can help to learn more features and might get higher accuracy.

Dense – output layer, made by 1 neuron because this is a binary classification. Used "sigmoid" in order to return a result between '0' to '1'.

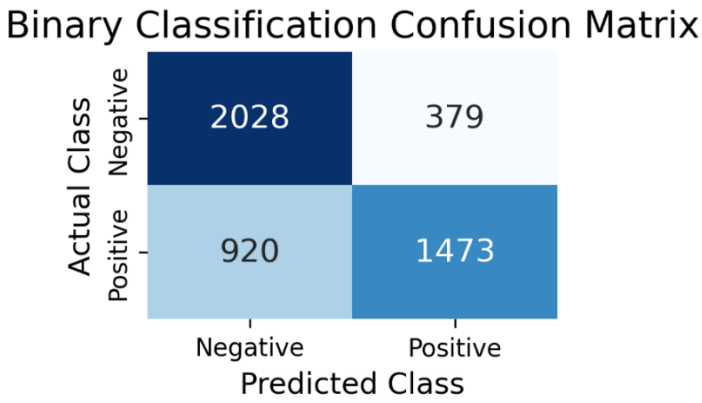
Batch size and epoch – we chose 128 size because smaller batch has better convergence. We could not use smaller batch because of the long run time. We took 10 epochs again, because the long run time.
Optimizer - The results of the Adam optimizer are generally better than every other optimization algorithms, have faster computation time, and require fewer parameters for tuning.
loss function - used in binary classification tasks and because "sigmoid" is the only activation function compatible with the binary cross entropy loss function which we used as the activation function in the output layer.

We created confusion matrix function in order to plot results,
plotted accuracy as a function of training epoch.

Results:



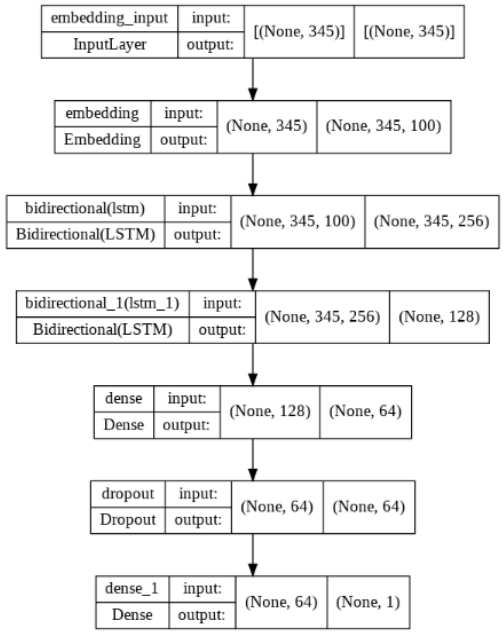
As we can see, there is over fitting, the accuracy of the train and test are very far from each other. Furthermore, the result is very low.



<u>Second prototype:</u>

Word Embedding :

We download Glove's pre-trained word embeddings in order to help solving our classification problems. We Loaded the Glove embeddings, and appended them to a dictionary. Then, used this dictionary to create an embedding matrix for each word in the training set. We got embedding vector for each word using "embedding_index". Words that aren't found, will be represented by zero.



Layers:
embedding_layer- Same as first model.

Bidirectional layers –
We created the bidirectional layers one returns full sequences of successive outputs for each timestep and the second returns only the last output for each input sequence. This hopefully will make the model more accurate.

Dense layer –
We used dense layer in order to make the output layer learn less parameters.
Used "relu" activation layer because it is the most common activation function which gives better results then "sigmoid" and "tanh".

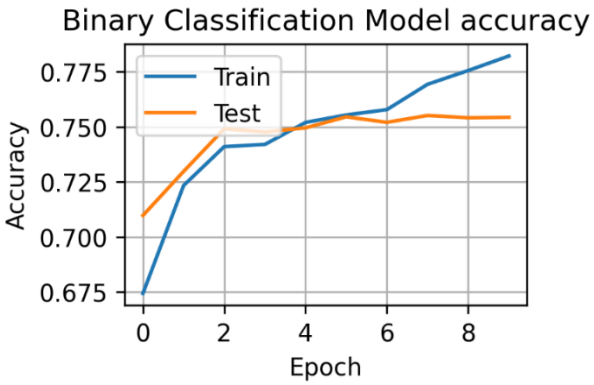
Dropout – Used for regularization.

Dense – output layer, made by 1 neuron because this is a binary classification. Used "sigmoid" in order to return a result between '0' to '1'.

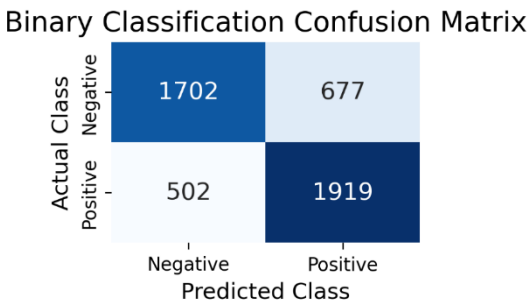
Batch size and epoch – we chose 128 size because smaller batch has better convergence. We could not use smaller batch because of the long run time. We took 10 epochs again, because the long run time.
Optimizer - The results of the Adam optimizer are generally better than every other optimization algorithms, have faster computation time, and require fewer parameters for tuning.
loss function - used in binary classification tasks and because "sigmoid" is the only activation function compatible with the binary crossentropy loss function which we used as the activation function in the output layer.

We created confusion matrix function in order to plot results,
plotted accuracy as a function of training epoch.

Results:



As we can see, there is less over fitting, the accuracy of the train and test are less far from each other. This model performed better but, still the result is very low. It might be because the variety of dataset.



As we can see, there are better results of the prediction. 2,596 True negative and True positive in contrast of 1852 True negative and True positive on the first model.

As we can see the first most significant variables are the batch, epoch, optimizer, and loss function as we mentioned above. Without a good optimizer the model will not be able to improve in the training process. The loss function is important in any statistical model - they define an objective which the performance of the model is evaluated against and the parameters learned by the model are determined by minimizing a chosen loss function.
The embedding layer plays an important role because it is already been trained before and Glove is known as a way of learning word embeddings from text.
Also, we discussed about the benefits of Bidirectional layers which improved the accuracy of our model.
Errors –

Resources:

Repeated K-fold and cross validation:
https://machinelearningmastery.com/repeated-k-fold-cross-validation-with-python/

Word embedding:
https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/

LSTM- BLSTM
https://www.baeldung.com/cs/bidirectional-vs-unidirectional-lstm
Adam optimizer –
https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/#:~:text=The%20results%20of%20the%20Adam,for%20most%20of%20the%20applications.

binary crossentropy –
https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy

GlobalMaxPooling1D-
https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/global-max-pooling-1d