אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

Twitter Sentiment Analysis

students:

Avihai Serfati 204520803

Haim Reyes 319510475

kaggle team name:

bibi

Intro

Natural Language Processing (NLP): The discipline of computer science, artificial intelligence and linguistics that is concerned with the creation of computational models that process and understand natural language. These include: making the computer understand the semantic grouping of words (e.g. cat and dog are semantically more similar than cat and spoon), text to speech, language translation and many more

Sentiment Analysis: It is the interpretation and classification of emotions (positive, negative and neutral) within text data using text analysis techniques. Sentiment analysis .allows organizations to identify public sentiment towards certain words or topics

In this project, we'll develop a **model** that can perform **Sentiment Analysis** to categorize a tweet as **Positive or Negative**

Dataset

The dataset being used is the given competition twitter dataset. It contains 80K tweets. The tweets have been annotated (0 = Negative, 1 = Positive) and they can be used to detect sentiment.

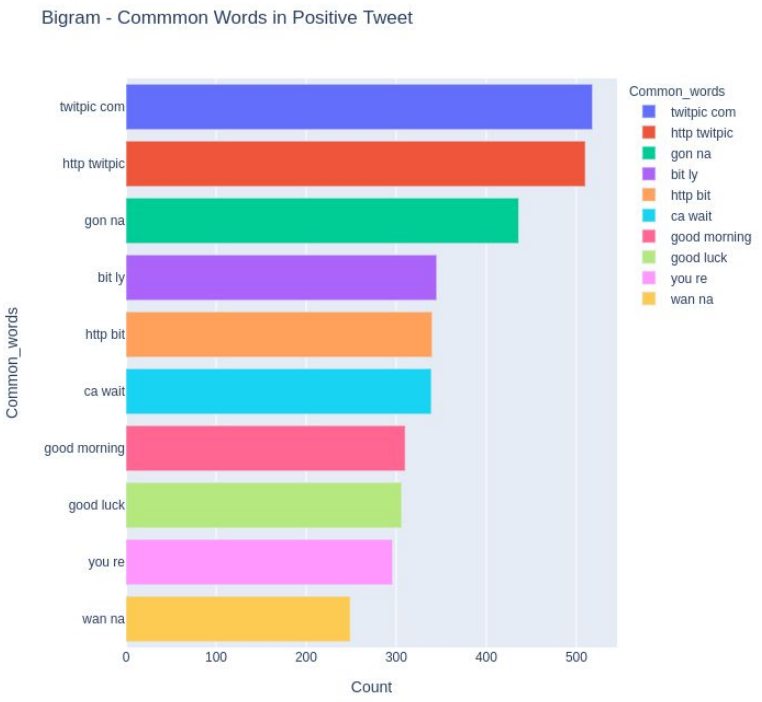
It contains the following 2 fields:

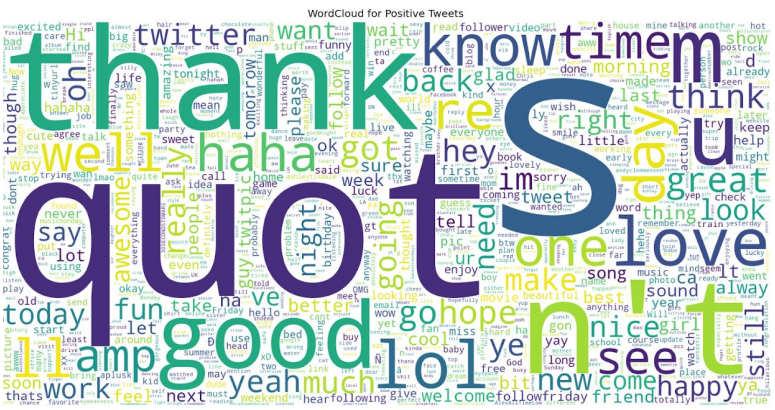
- **Sentiment**: the emotion of the tweet *(0 = negative, 1 = positive)*
- **SentimentText**: the text of the tweet

Project Sections

This document builds from 2 main sections. The **first** section focuses on deep learning model and describes its data preparation and model creation.
The second section describes another approach using classic machine learning algorithms such as Logistic Regression and Naive Bayes and their evaluation results.

Data Visualization





Preprocessing

Tweets usually contain a lot of information apart from the text, like mentions, hashtags, urls, emojis or symbols. Since normally, NLP models cannot parse those data, we need to clean up the tweet and replace tokens that actually contain meaningful information for the model.

The Preprocessing **steps** taken are:

1. Lower Casing: Each text is converted to lowercase.
2. Replacing URLs: Links starting with 'http' or 'https' or 'www' are replaced by '<url>'.
3. Replacing Usernames: Replace @Usernames with word '<user>'. [eg: '@Kaggle' to '<user>'].
4. Replacing Consecutive letters: 3 or more consecutive letters are replaced by 2 letters. [eg: 'Heyyyy' to 'Heyy']
5. Replacing Emojis: Replace emojis by using a regex expression.
6. Replacing Contractions: Replacing contractions with their meanings. [eg: "can't" to 'can not']
7. Removing Non-Alphabets: Replacing characters except Digits, Alphabets and pre-defined Symbols with a space.
8. Stop Words Removal: Removing all stop words from tweets. (only on section 2)
9. Porter Stemmer: Removing the commoner morphological and inflexional endings from words. (only on section 2)

As much as the preprocessing steps are important, the actual sequence is also important while cleaning up the text. For example, removing the punctuations before replacing the urls means the regex expression cannot find the urls. Same with mentions or hashtags. So make sure, the actual sequence of cleaning makes sense.

Feature Extraction section 1 shows on DNN.py

Word2Vec was developed by Google and is one of the most popular techniques to learn word embeddings using shallow neural networks. Word2Vec can create word embeddings using two methods (both involving Neural Networks): **Skip Gram** and **Common Bag Of Words (CBOW)**.

Word2Vec() function creates and trains the word embeddings using the data passed.

Tokenizing and Padding datasets

Tokenization is a way of separating a piece of text into smaller units called **tokens**. Here, tokens can be either words, characters, or subwords. Hence, tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization.

All the neural networks require inputs that have the same shape and size. However, when we pre-process and use the texts as inputs for our model e.g. LSTM, not all the sentences have the same length. We need to have the inputs with the same size, this is where the **padding** is necessary.

Padding is the process by which we can add padding tokens at the start or end of a sentence to increase it's length upto the required size. If required, we can also drop some words to reduce to the specified length.

- **Tokenizer:** Tokenizes the dataset into a list of tokens.
- **pad_sequences:** Pads the tokenized data to a certain length.

The **input_length** has been set to 60. This will be the length after the data is tokenized and padded.

Splitting the Data section 1 & 2

Machine Learning models are trained and tested on different sets of data. This is done so to reduce the chance of the model overfitting to the training data, i.e it fits well on the training dataset but a has poor fit with new ones.

sklearn.model_selection.cross_val_score shuffles the dataset and splits it into train and test dataset.

The Pre-processed Data is divided into 2 sets of data using 10 Cross Validation.

For the deep learning model we use a validation split of 5% from the training set as a model parameter. see below

Creating Embedding Matrix section 1 shows on DNN.py

Embedding Matrix is a matrix of all words and their corresponding embeddings. We use an embedding matrix in an Embedding layer in our model to embedded a token into it's vector representation, that contains information regarding that token or word.

We get the embedding vocabulary from the tokenizer and the corresponding vectors from the Embedding Model, which in this case is the Word2Vec model.

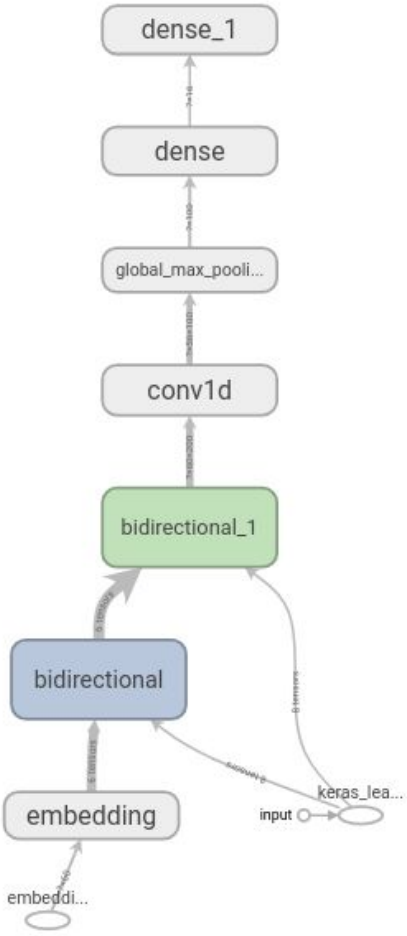
Shape of Embedding matrix is usually the Vocab Length * Embedding Dimension.

Creating the Model section 1 shows on DNN.py

There are different approaches which we can use to build our Sentiment analysis model. We're going to build a deep learning Sequence model.

Sequence model is very good at getting the context of a sentence, since it can understand the meaning rather than employ techniques like counting positive or negative words like in a Bag-of-Words model.

Model Architecture



Bidirectional: Bidirectional wrapper for RNNs. It means the context is carried from both left to right and right to left in the wrapped RNN layer.

LSTM: Long Short Term Memory, it's a variant of **RNN** which *has a memory* state cell to learn the context of words which are further along the text to carry contextual meaning rather than just neighbouring words as in case of RNN.

Model Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 60, 100)           4540300
_________________________________________________________________
bidirectional (Bidirectional (None, 60, 200)           160800
_________________________________________________________________
bidirectional_1 (Bidirection (None, 60, 200)           240800
_________________________________________________________________
conv1d (Conv1D)              (None, 56, 100)           100100
_________________________________________________________________
global_max_pooling1d (Global (None, 100)               0
_________________________________________________________________
dense (Dense)                (None, 16)                1616
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 5,043,633
Trainable params: 503,333
Non-trainable params: 4,540,300
```

Training the Model section 1 shows on DNN.py

Model Callbacks

```
callbacks = [ReduceLROnPlateau(monitor='val_loss', patience=5, cooldown=0),
             EarlyStopping(monitor='val_acc', min_delta=1e-6, patience=15),
             tensor_board
             ]
```

Callbacks are objects that can perform actions at various stages of training.

We can use callbacks to write **TensorBoard logs** after every batch of training, periodically save our model, stop training early or even to get a view on internal states and statistics during training.

ReduceLROnPlateau: Reduces Learning Rate whenever the gain in performance metric specified stops improving.

EarlyStopping: Stop training when a monitored metric has stopped improving.

Model Compile

The Model must be compiled to define the **loss, metrics and optimizer**. Defining the proper loss and metric is essential while training the model.

Loss: We're using **Binary Cross-Entropy**. It is used when we have binary output categories.

Metric: We've selected **some** of the common evaluation metrics in classification problems.

```
mymetrics=['acc',metrics.Precision(), metrics.Recall(), metrics.AUC(), metrics.RootMeanSquaredError()]
```

Optimizer: We're using **Adam**, optimization algorithm for Gradient Descent.

We'll now train our model using the **fit** method and store the output learning parameters in **history**, which can be used to plot out the learning curve.

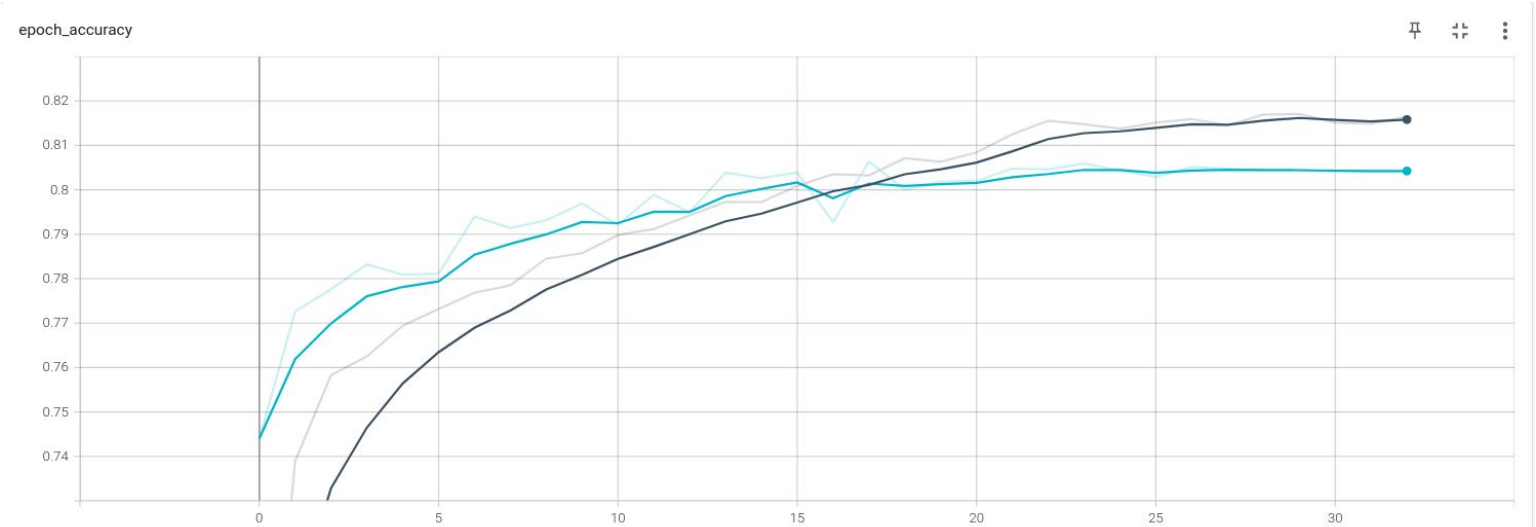
Model Parameters

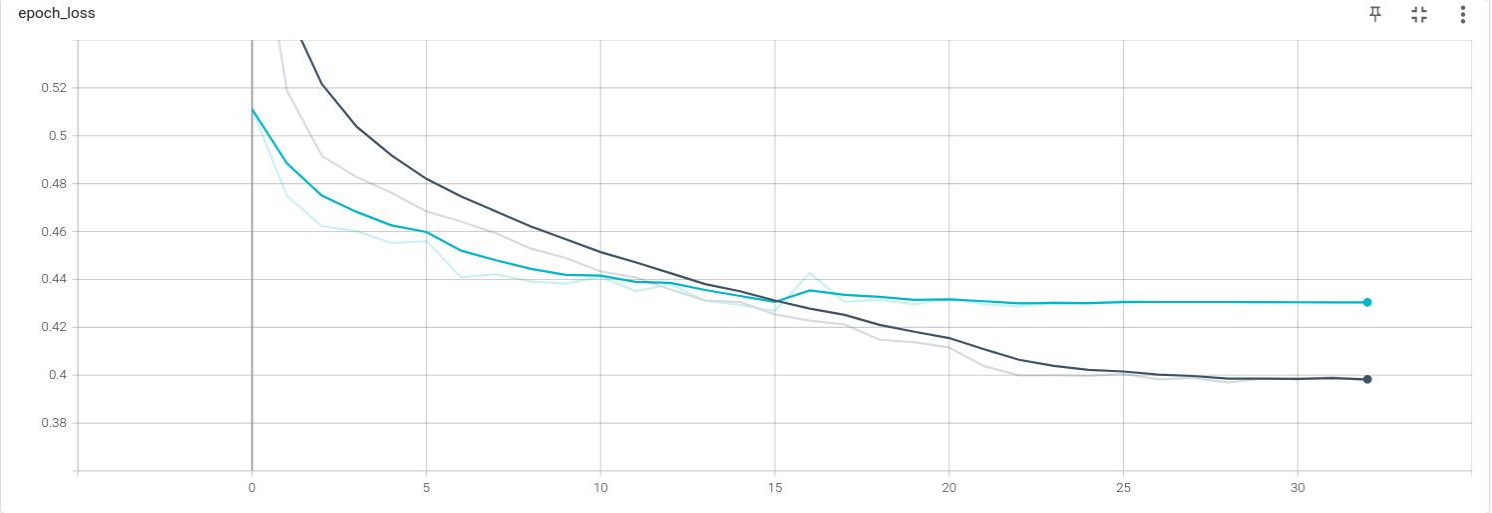
```
history = training_model.fit(
    X_data,
    y_data,
    batch_size=4096,
    epochs=50,
    validation_split=0.05,
    callbacks=callbacks,
    verbose=1
)
```

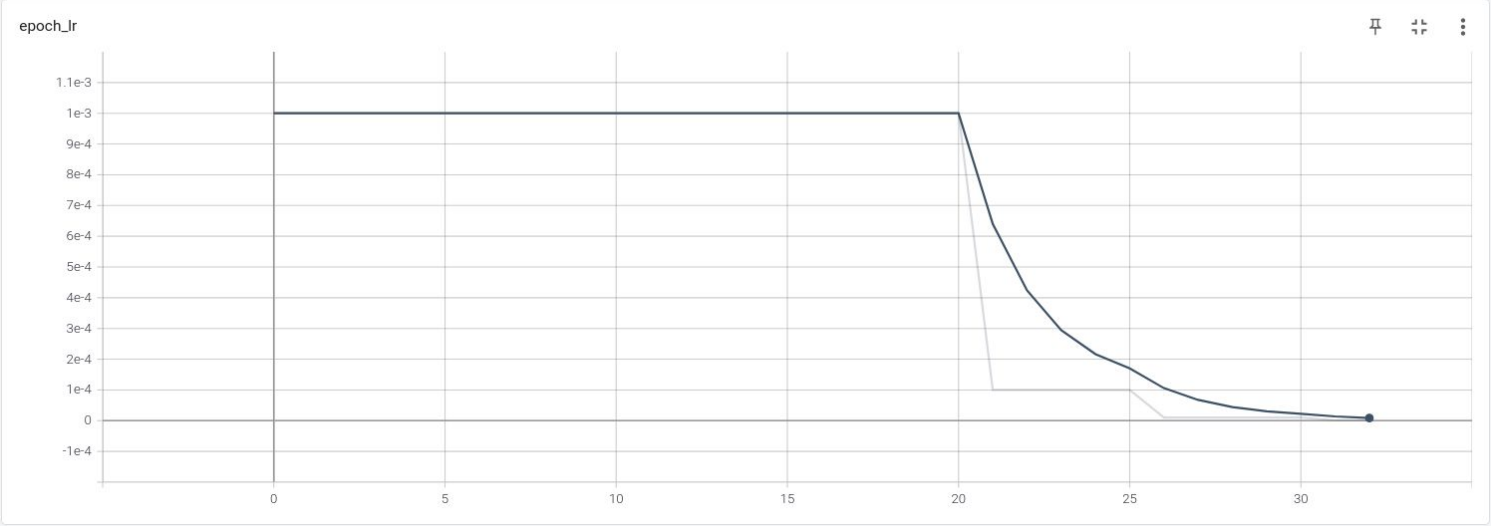
Evaluating Model section 1

We're choosing **Accuracy** as our main evaluation metric.

Furthermore, we're plotting the **Classification Report** to get an understanding of how our model is performing on both classification types. Printing out the Learning curve (**Tensorboard logging**)







Learning curves show the relationship between training set size and your chosen evaluation metric on your training and validation sets. They can be an extremely useful tool when diagnosing your model performance, as they can tell you whether your model is suffering from bias or variance.

From the training curve we can conclude that our model doesn't have bias nor is it overfitting. The accuracy curve has flattened, which means training got best results.

Metrics section 1 shows on DNN.py

Set/Metric	*Accuracy*	*Recall*	*Precision*	*RMSE*	*AUC*
Train	0.8204	0.8499	0.8345	0.3534	0.9027
Validation	0.8037	0.8481	0.8156	0.3691	0.8859

Using the classification report, we can see that the model achieves around **80% Accuracy** after training for just **30 epochs** (EarlyStopping callback stopped the model). This is really good and better than most other models achieve.

Saving the Model section 1 shows on DNN.py

We're saving the **tokenizer and Tensorflow model** for use later.

Kaggle Competition

We achieved **79.7%** accuracy using our DNN model on the test set which was provided by the competition and taking the 6th place. Our team name for this competition is **'bibi'.**

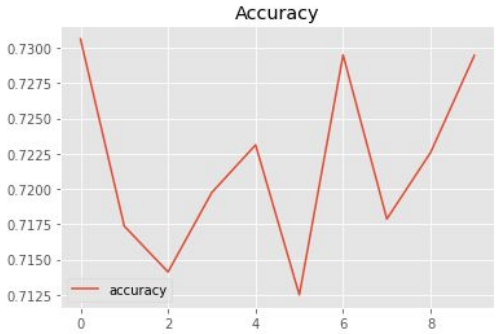
Section 2 - Classic Machine Learning Classifiers

These are more simple ML models (our non-competition models) meant for exploring additional methods for Sentiment Analysis and Machine Learning. We will put each model through **10fold cross validation** with **tfidf** vectorization and **count** vectorization.

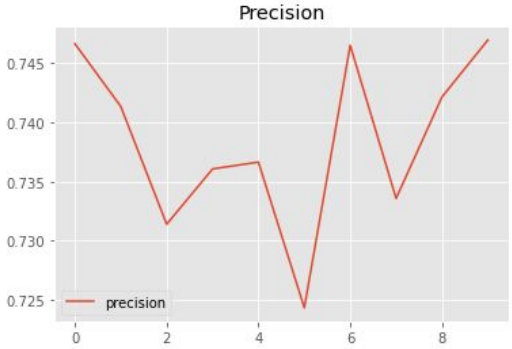
First Classifier: SGD classifier

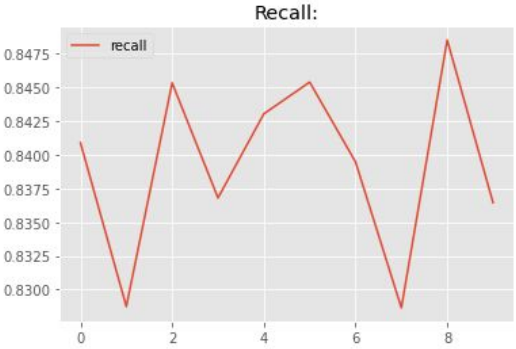
Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to fitting linear classifiers and regressors under **convex loss functions** such as (linear) SVM and Logistic Regression. We will try it because of his best advantage, his Efficiency.

10Fold cross validation with tfidf vectorization training data

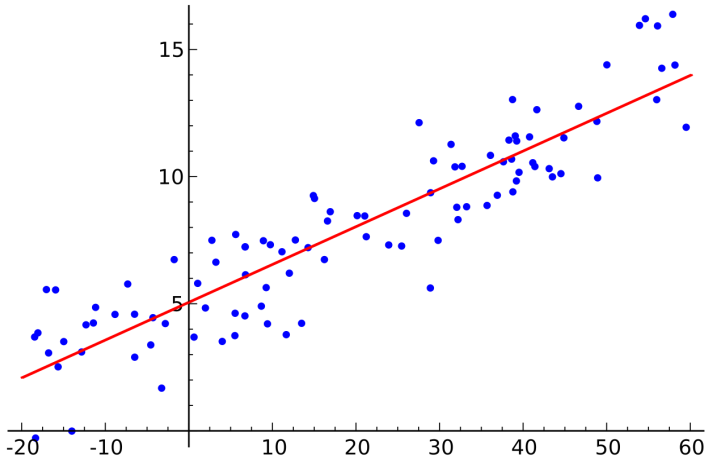


10Fold cross validation with count vectorization training data



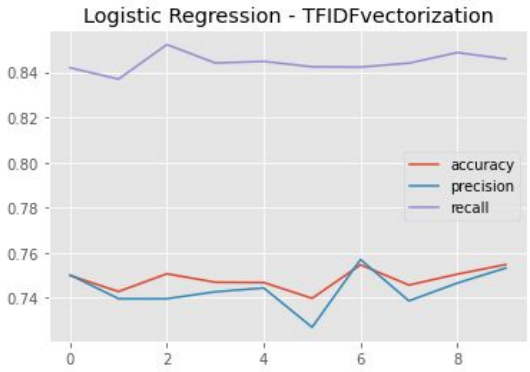


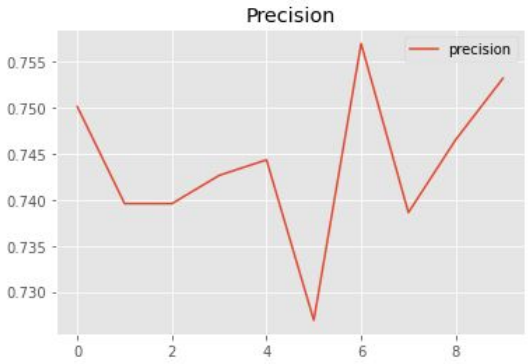
Second Classifier: Logistic Regression

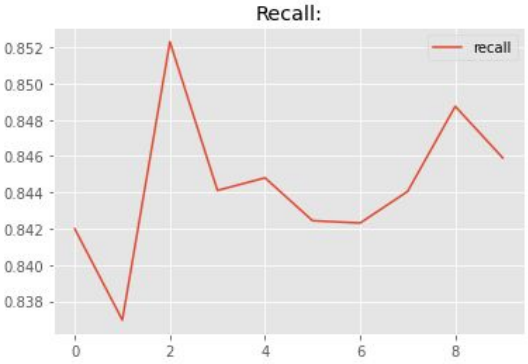


Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable, although many more complex extensions exist. We will insert our features to the model in order to classify our own binary results, positive or negative.

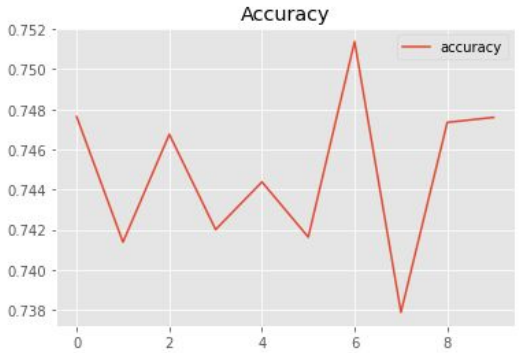
10Fold cross validation with tfidf vectorization training data

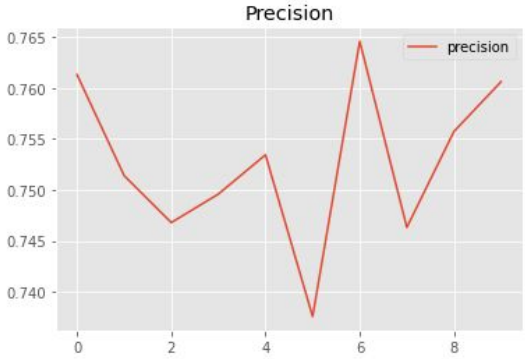






10Fold cross validation with count vectorization training data

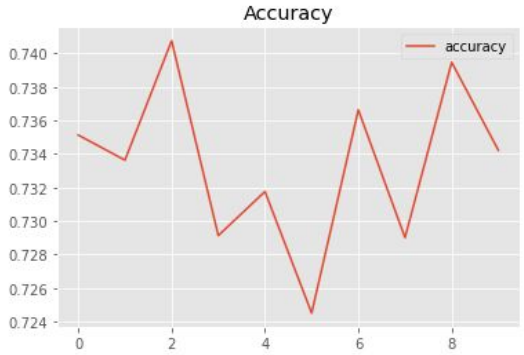


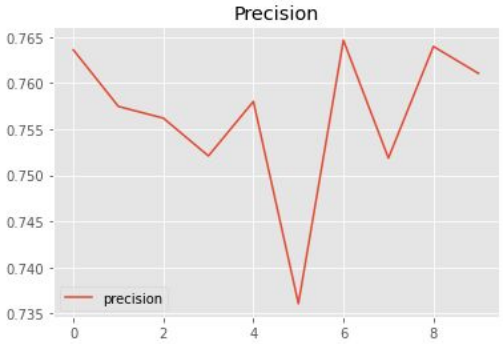


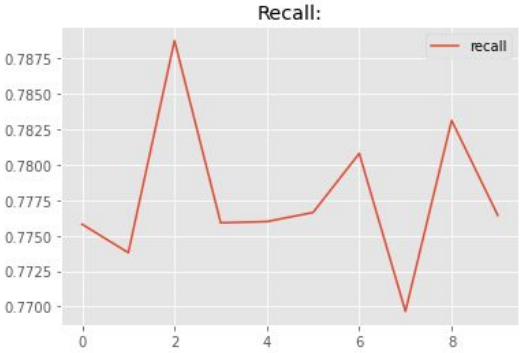
Third Classifier Multinomial Naive Bayes

Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features. In our second classifier, the **Logistic Regression**, the features are assumed to be codependent to some extent, we choose a naive model to see if the opposite approach will give us better results. Meaning we will try to treat the **features** as **fully independent**.

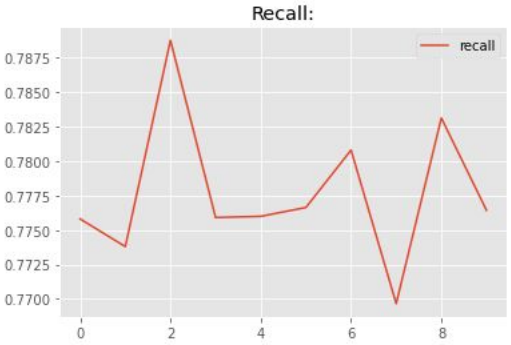
10Fold cross validation with tfidf vectorization training data







10Fold cross validation with count vectorization training data



Feature Extraction - In scikit-learn, the TF-IDF algorithm is implemented using **TfidfTransformer**. This transformer needs the count matrix which it will transform later. Hence, we use **CountVectorizer** first. Alternatively, one can use **TfidfVectorizer**, which is the equivalent of CountVectorizer followed by TfidfTransformer

Training Set Accuracy by Feature Extraction

Model / FE	*TfIdfVectorizer*	*CountVectorizer*
MultinomialNB	90.493%	93.437%
SGD	76.160%	91.450%
LogisticRegression	86.326%	95.944%

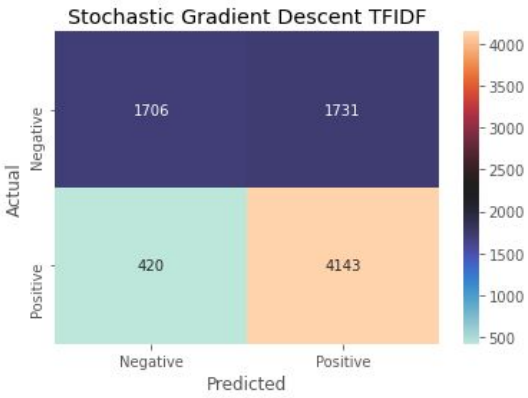
Validation Set using CountVectorizer Feature Extraction

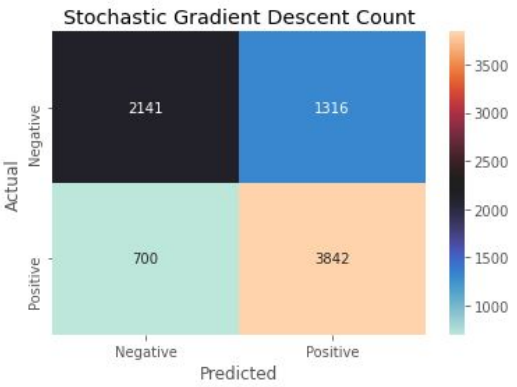
Model/Metric	*Accuracy*	*Recall*	*Precision*
SGDClassifier	74.075%	83.680%	73.866%
Logistic Regression	74.587%	81.443%	75.477%
MultinomialNB	73.637%	81.842%	74.142%

Validation Set using TfIdfVectorizer Feature Extraction

Model/Metric	*Accuracy*	*Recall*	*Precision*
SGDClassifier	73.187%	87.621%	71.387%
Logistic Regression	75.237% best	82.152%	75.946%
MultinomialNB	72.65%	90.124%	70.027%

Classic Model Confusion Matrix (best accuracy fold)





Appendix

Files and Folders:

- *DNN.py* - our main deep neural network model that is described in **section 1**.
- *classic_ml.ipynb* - Jupyter notebook which contains all of the classic machine learning algorithms that are described in **section 2**.
- *requirements.txt* - all the needed python packages can be found in this file.
- *plots folder* - contains all the plots that displayed on this document.
- *assets folder* - contains all assets of this project includes datasets and output files.
    - visualization.py - this file produces all the plots dynamically.
    - submission.py - this file creates a submission.csv file for the competition.