Flipkart Sentiment Analysis

Apeksha Shrirao, Anuja Merwade, Vasu Sharma

Abstract

This project is built with an aim of performing sentiment analysis of users' product reviews on an online e-commerce platform named Flipkart. The dataset used is of Flipkart customers. The project is developed with the intention of predicting sentiment of the users so that similar recommendations can be provided to the users in future. After obtaining the data, operations such as data cleaning, Exploratory Data Analysis, implementation of various Machine Learning algorithms and hyperparameter tuning have been performed in order to achieve best results for the respective ML algorithm. The project is implemented in python on Google Collab running on an IU supercomputer named 'burrow'. The future scope of the project is to take a larger dataset with less bias and more number of features with more data points.

Introduction

The main objective of the project is to analyze sentiment of the users who have purchased various products from an e-commerce website named Flipkart. This can be done based on various features present in the dataset such as reviews, summary etc.

We find this problem statement interesting because this analysis can be used to solve various problem statements such as improving recommendation systems, improving overall user experience of visiting and buying products on the website.

We believe that in this era, where technology is largely used and sometimes, people prefer to text rather than communicate verbally, text analysis using various Natural Language Processing techniques can provide major insights into people's behavior which can be used to predict the overall shopping pattern of the user.

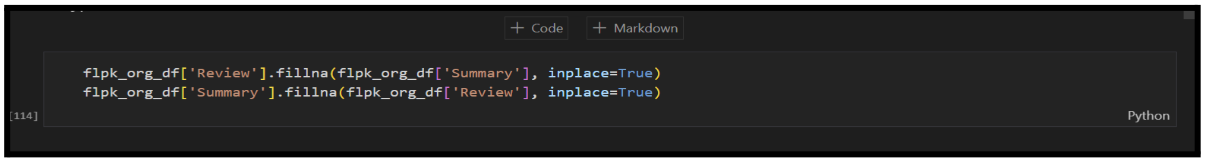
Methodology

3.1 Collecting Data:

The first step is to collect data. For this project, we have acquired the Flipkart dataset from Kaggle. This dataset has 205053 data points with 6 features namely product_name, product_price, Rate, Review, Summary, Sentiment.

3.2 Handling the missing values:

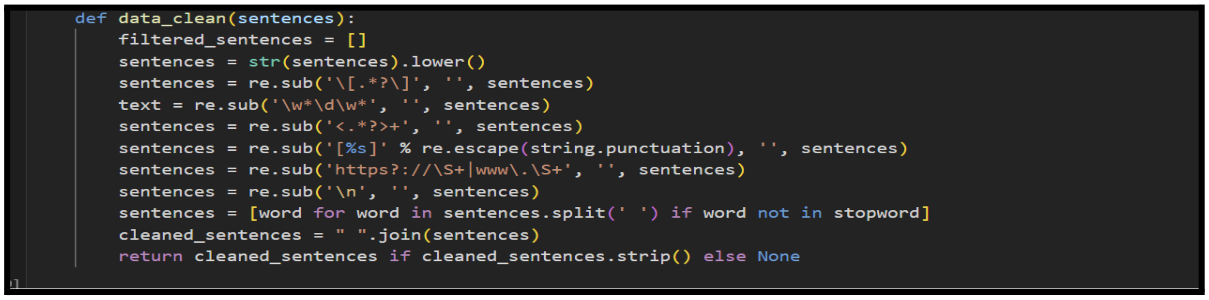
This dataset has around 24664 missing values for the Review column and 11 missing values for the Summary column. To preserve data, we replaced missing values in Review with values of Summary column and vice versa.



```
flpk_org_df['Review'].fillna(flpk_org_df['Summary'], inplace=True)
flpk_org_df['Summary'].fillna(flpk_org_df['Review'], inplace=True)
```

3.3 Data transformation:

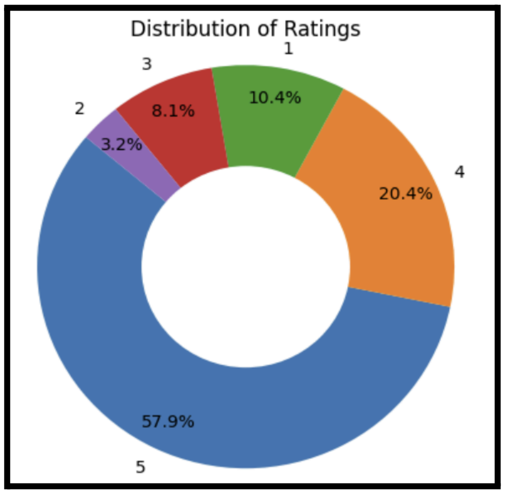
Once the missing values are handled, we begin the process of data cleaning. This was done by utilizing various methods such as changing text into lowercase, removing various special characters, stopwords, etc.

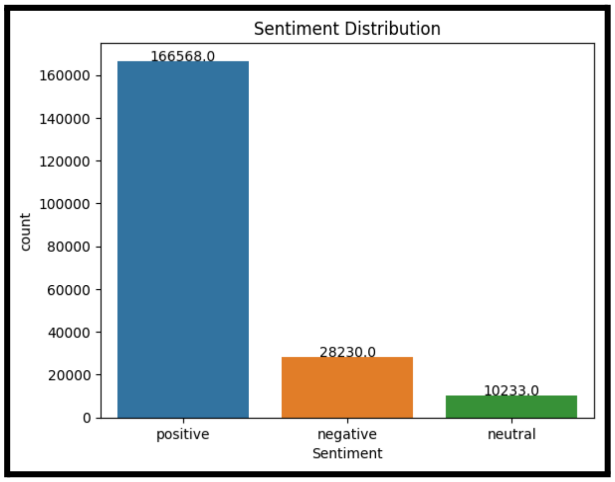


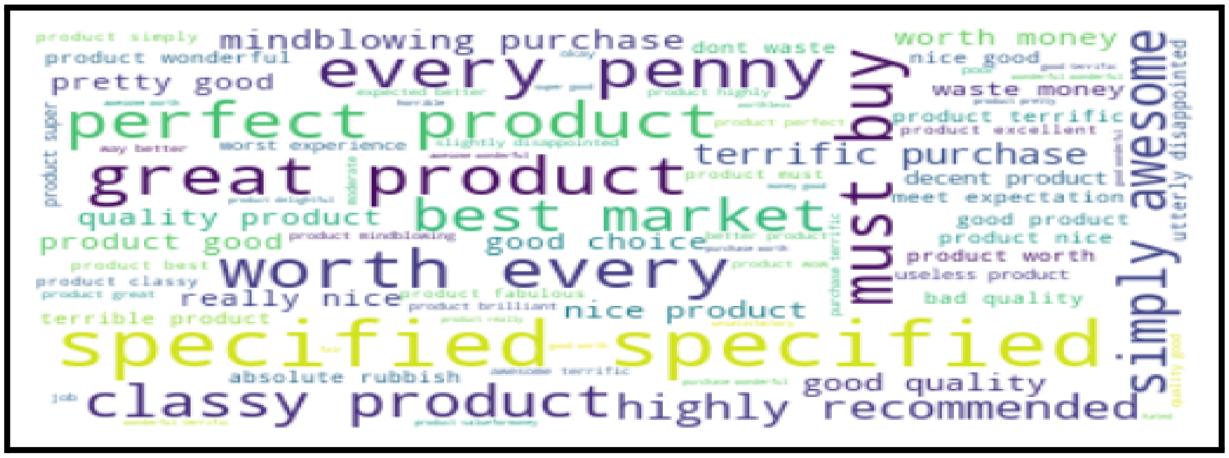
```
def data_clean(sentences):
    filtered_sentences = []
    sentences = str(sentences).lower()
    sentences = re.sub('\[.*?\]', '', sentences)
    text = re.sub('\w*\d\w*', '', sentences)
    sentences = re.sub('<.*?>+', '', sentences)
    sentences = re.sub('[%s]' % re.escape(string.punctuation), '', sentences)
    sentences = re.sub('https?://\S+|www\.\S+', '', sentences)
    sentences = re.sub('\n', '', sentences)
    sentences = [word for word in sentences.split(' ') if word not in stopword]
    cleaned_sentences = " ".join(sentences)
    return cleaned_sentences if cleaned_sentences.strip() else None
```

3.4 Exploratory Data Analysis

Once data cleaning and data transformation is done, we started working on performing Exploratory Data Analysis on the dataset. The various EDA that has been done are as follows:



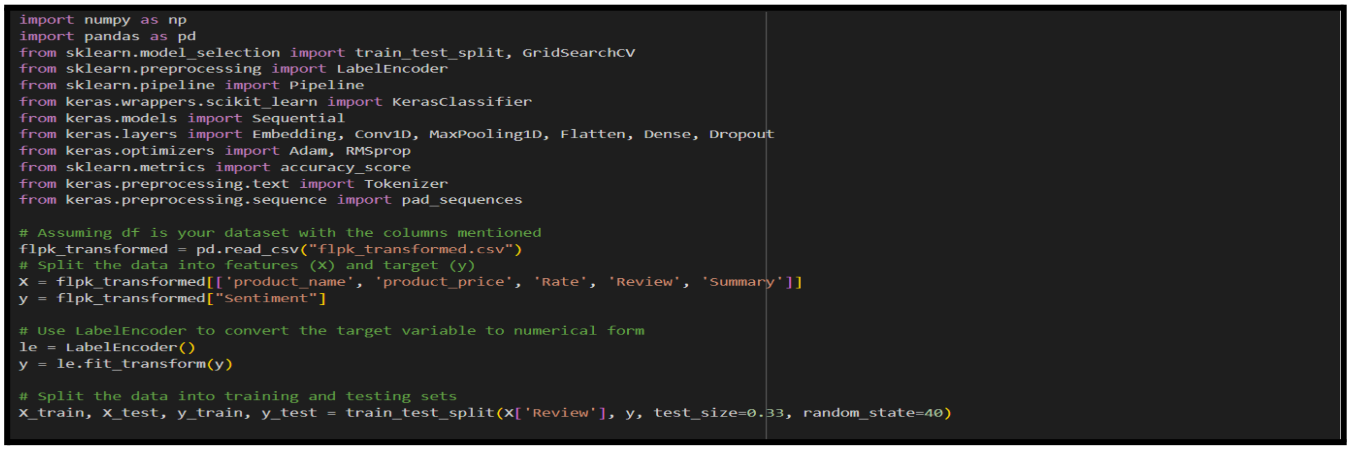




Algorithms and Experimental Setup Used:

Next, we implemented various ML algorithms on the dataset. Before applying ML algorithms, following steps are followed:

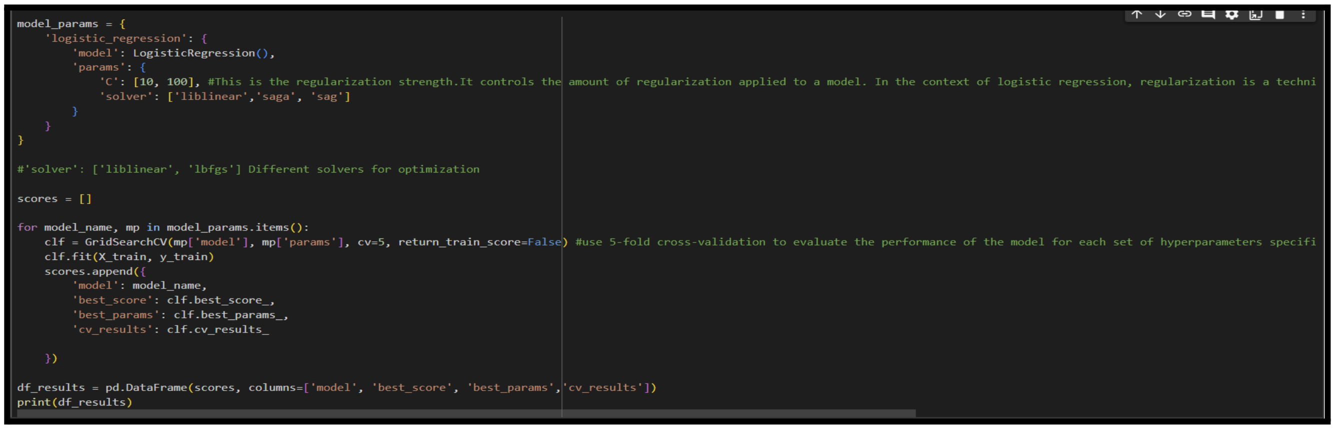
1. First the text data is converted into numerical data using the CountVectorizer method for columns such as Review, Summary and Product Name and Label encoder for the Sentiment Column.
2. The data is split into training and test data with test data size being 33% and remaining is training data.
3. We also performed Hyperparameter tuning on all the algorithms on Google Collab and using Indiana University resources such as burrow and sharks.



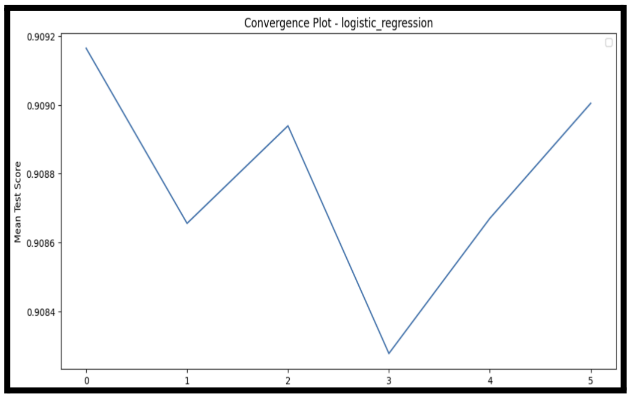
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from keras.wrappers.scikit_learn import KerasClassifier
from keras.models import Sequential
from keras.layers import Embedding, Conv1D, MaxPooling1D, Flatten, Dense, Dropout
from keras.optimizers import Adam, RMSprop
from sklearn.metrics import accuracy_score
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# Assuming df is your dataset with the columns mentioned
flpk_transformed = pd.read_csv("flpk_transformed.csv")
# Split the data into features (X) and target (y)
X = flpk_transformed[['product_name', 'product_price', 'Rate', 'Review', 'Summary']]
y = flpk_transformed["Sentiment"]

# Use LabelEncoder to convert the target variable to numerical form
le = LabelEncoder()
y = le.fit_transform(y)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X['Review'], y, test_size=0.33, random_state=40)
```

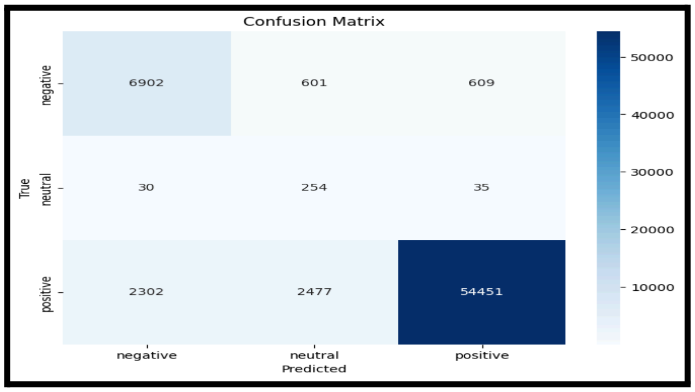
Logistic Regression:

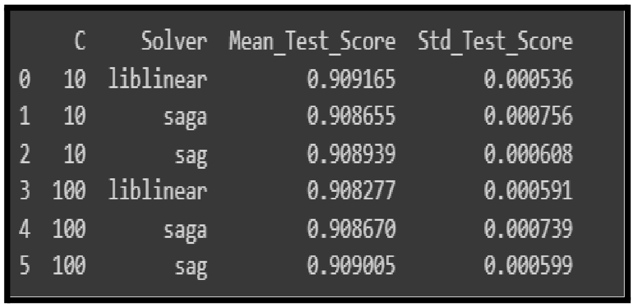
Logistic Regression is a binary classification algorithm that can be extended to multi-class classification, which makes it suitable for sentiment analysis where the goal is to classify text into positive, negative, or neutral sentiments. Scikit, by default, uses a one vs rest approach where a separate binary logistic regression classifier for each class is being trained.
During prediction, one obtains the scores from each classifier, and the class with the highest score is chosen as the final prediction. We have also implemented hyperparameter tuning on this algorithm by changing various parameters such as values of c and solver.
c: It is a regularization technique to prevent overfitting and improve the generalization of the model.
Liblinear: It is an iterative optimization method for finding the local minimum of a differentiable function.
SAGA: It is a variant of stochastic gradient descent (SGD) It incorporates variance reduction techniques.
SAG: It is another variant of stochastic gradient descent without the variance reduction techniques.



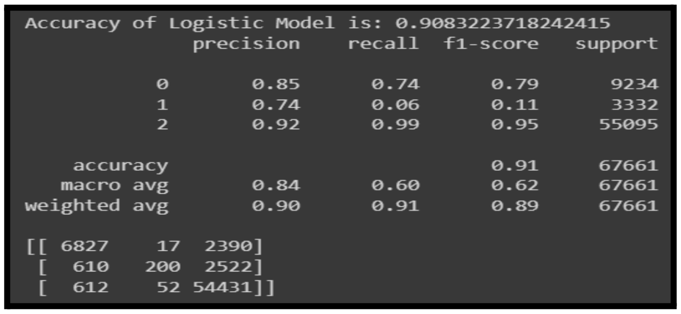
```
model_params = {
    'logistic_regression': {
        'model': LogisticRegression(),
        'params': {
            'C': [10, 100], #This is the regularization strength.It controls the amount of regularization applied to a model. In the context of logistic regression, regularization is a techni
            'solver': ['liblinear','saga', 'sag']
        }
    }
}

#'solver': ['liblinear', 'lbfgs'] Different solvers for optimization

scores = []

for model_name, mp in model_params.items():
    clf = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False) #use 5-fold cross-validation to evaluate the performance of the model for each set of hyperparameters specifi
    clf.fit(X_train, y_train)
    scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_,
        'cv_results': clf.cv_results_

    })

df_results = pd.DataFrame(scores, columns=['model', 'best_score', 'best_params','cv_results'])
print(df_results)
```







```
      C     Solver  Mean_Test_Score  Std_Test_Score
0    10  liblinear         0.909165        0.000536
1    10       saga         0.908655        0.000756
2    10        sag         0.908939        0.000608
3   100  liblinear         0.908277        0.000591
4   100       saga         0.908670        0.000739
5   100        sag         0.909005        0.000599
```



```
Accuracy of Logistic Model is: 0.9083223718242415
              precision    recall  f1-score   support

           0       0.85      0.74      0.79      9234
           1       0.74      0.06      0.11      3332
           2       0.92      0.99      0.95     55095

    accuracy                           0.91     67661
   macro avg       0.84      0.60      0.62     67661
weighted avg       0.90      0.91      0.89     67661

[[ 6827    17  2390]
 [  610   200  2522]
 [  612    52 54431]]
```

Multinomial Naive Bayes:

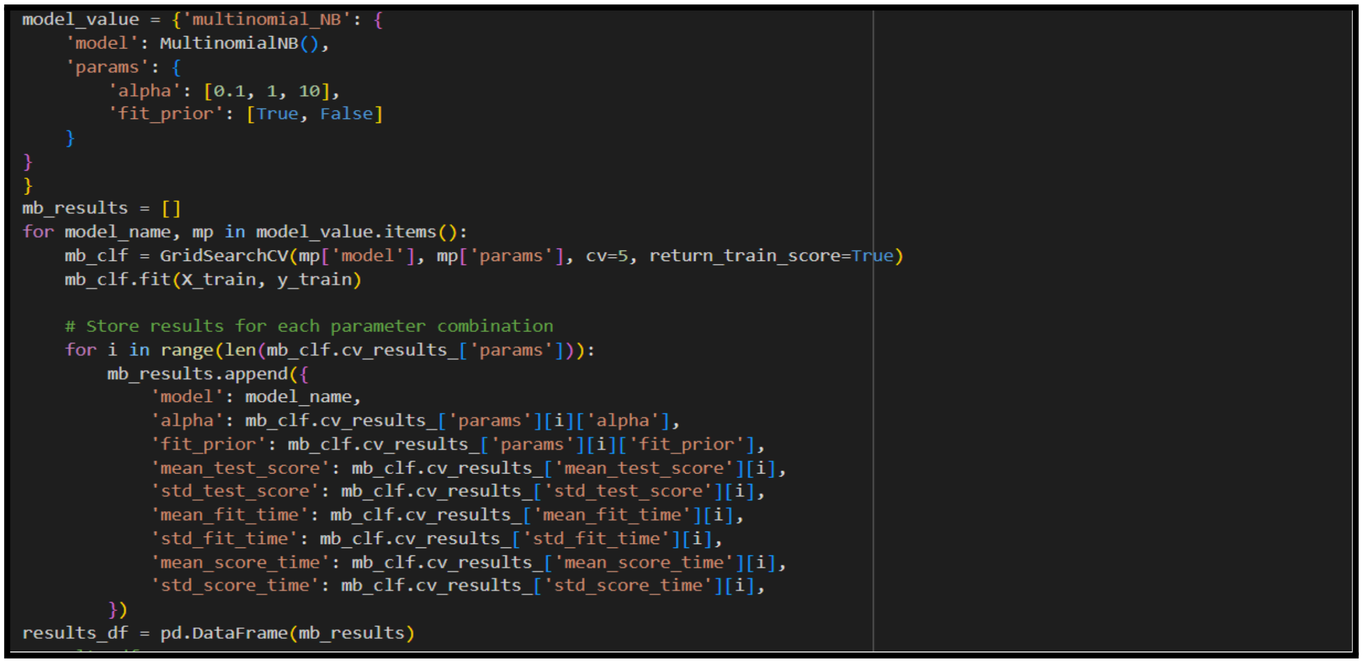
This works by computing the probability of each possible sentiment label (positive, negative, neutral) given a piece of text which would be Review and Summary in our dataset. We have performed hyperparameter tuning for this algorithms by changing following parameters:

Alpha:

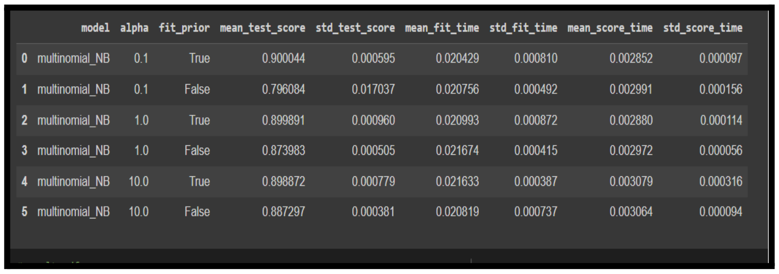
It is a smoothing parameter used to handle the issue of zero probabilities to prevent biased models. It essentially adds a small amount to the count of each feature, ensuring that no feature has a probability of zero.

Fit Prior:

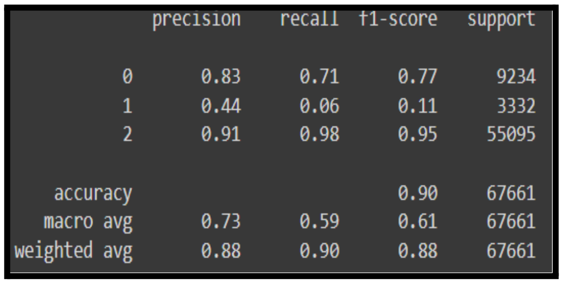
It is a parameter that determines whether to learn class prior probabilities or not. If fit prior is set to True (default), the classifier will estimate and use the prior probabilities based on the training data. If set to False, it assumes uniform prior probabilities for all classes.



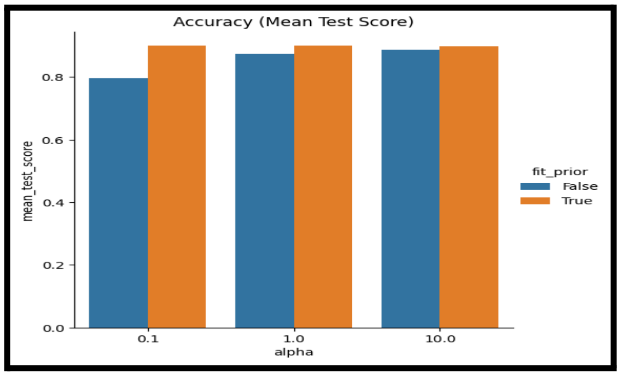
```python
model_value = {'multinomial_NB': {
    'model': MultinomialNB(),
    'params': {
        'alpha': [0.1, 1, 10],
        'fit_prior': [True, False]
    }
}
}
mb_results = []
for model_name, mp in model_value.items():
    mb_clf = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=True)
    mb_clf.fit(X_train, y_train)

    # Store results for each parameter combination
    for i in range(len(mb_clf.cv_results_['params'])):
        mb_results.append({
            'model': model_name,
            'alpha': mb_clf.cv_results_['params'][i]['alpha'],
            'fit_prior': mb_clf.cv_results_['params'][i]['fit_prior'],
            'mean_test_score': mb_clf.cv_results_['mean_test_score'][i],
            'std_test_score': mb_clf.cv_results_['std_test_score'][i],
            'mean_fit_time': mb_clf.cv_results_['mean_fit_time'][i],
            'std_fit_time': mb_clf.cv_results_['std_fit_time'][i],
            'mean_score_time': mb_clf.cv_results_['mean_score_time'][i],
            'std_score_time': mb_clf.cv_results_['std_score_time'][i],
        })
results_df = pd.DataFrame(mb_results)
```

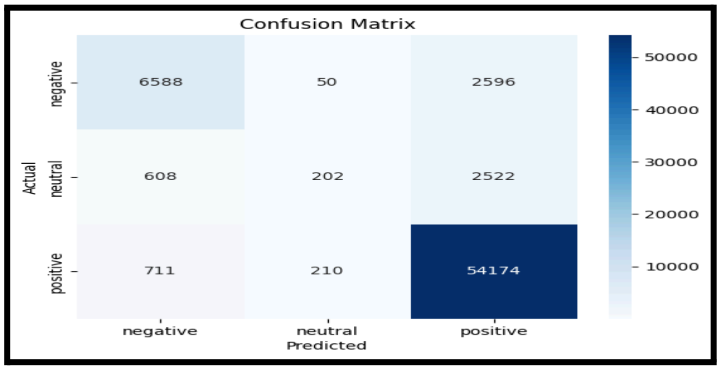


	model	alpha	fit_prior	mean_test_score	std_test_score	mean_fit_time	std_fit_time	mean_score_time	std_score_time
0	multinomial_NB	0.1	True	0.900044	0.000595	0.020429	0.000810	0.002852	0.000097
1	multinomial_NB	0.1	False	0.796084	0.017037	0.020756	0.000492	0.002991	0.000156
2	multinomial_NB	1.0	True	0.899891	0.000960	0.020993	0.000872	0.002880	0.000114
3	multinomial_NB	1.0	False	0.873983	0.000505	0.021674	0.000415	0.002972	0.000056
4	multinomial_NB	10.0	True	0.898872	0.000779	0.021633	0.000387	0.003079	0.000316
5	multinomial_NB	10.0	False	0.887297	0.000381	0.020819	0.000737	0.003064	0.000094



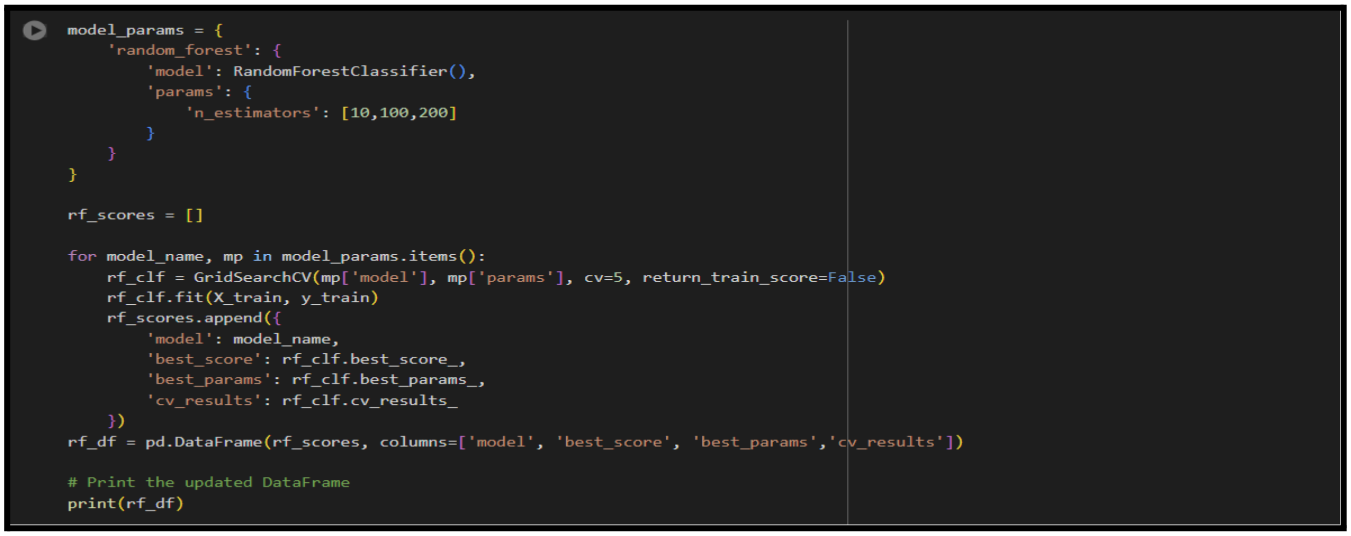
```
              precision    recall  f1-score   support

           0       0.83      0.71      0.77      9234
           1       0.44      0.06      0.11      3332
           2       0.91      0.98      0.95     55095

    accuracy                           0.90     67661
   macro avg       0.73      0.59      0.61     67661
weighted avg       0.88      0.90      0.88     67661
```



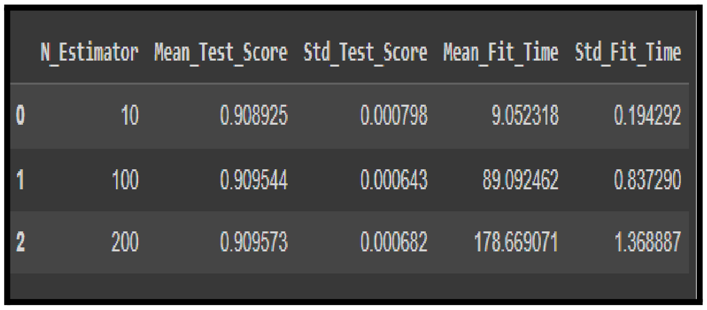


Random Forest:

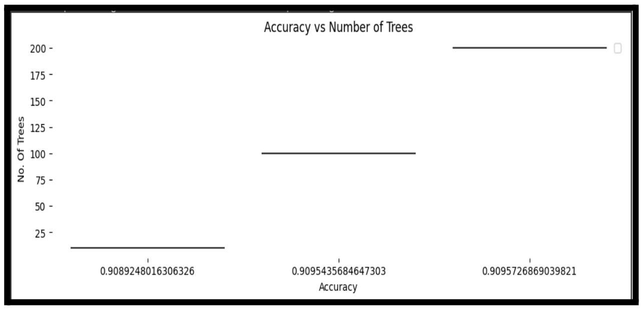
Random Forest works by building an ensemble of decision trees during training. Each tree is being trained on a random subset of the reviews (bootstrapped sample). This randomness helps reduce overfitting.
At each split of a tree, only a random subset of features is considered. When a new review is presented for prediction, each tree in the forest predicts the sentiment independently. The final sentiment prediction is determined by a majority vote among all the trees. We have performed hyperparameter tuning by changing n_estimator value which is the number of trees to be created while implementing Random Forest Algorithm.

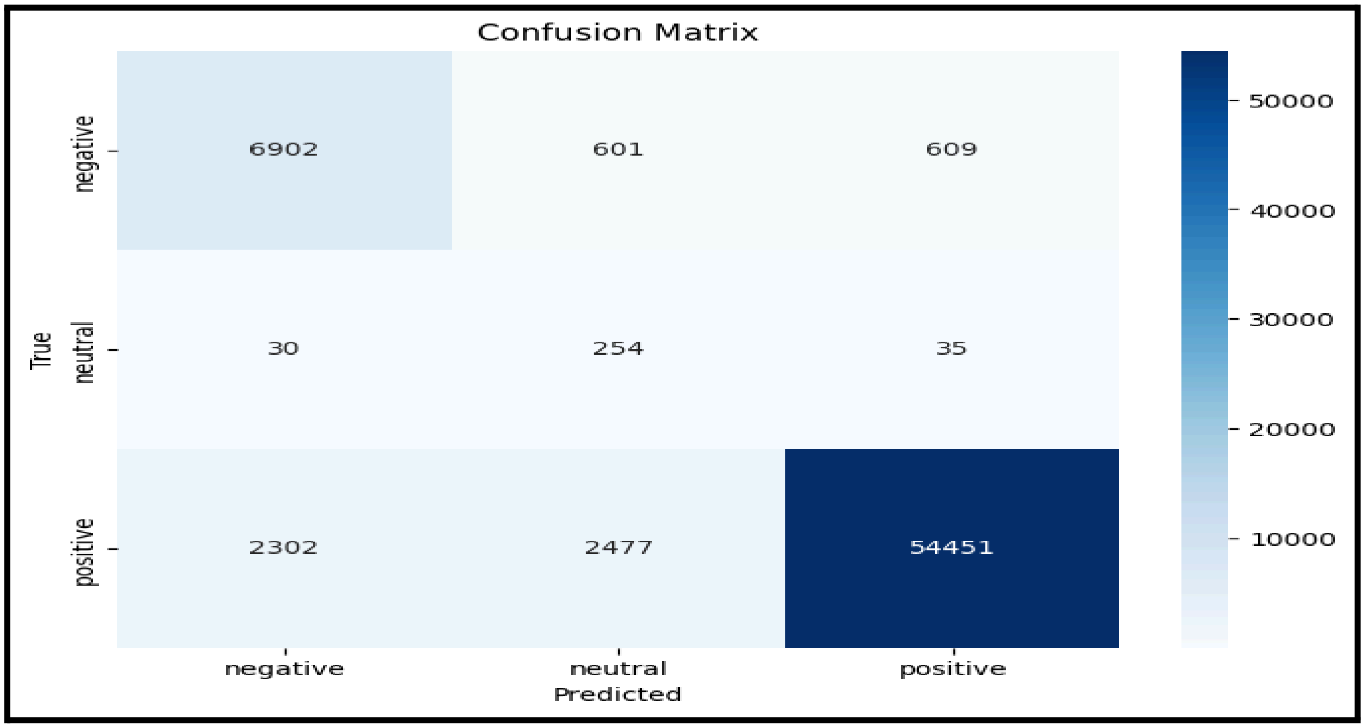


```
model_params = {
    'random_forest': {
        'model': RandomForestClassifier(),
        'params': {
            'n_estimators': [10,100,200]
        }
    }
}

rf_scores = []

for model_name, mp in model_params.items():
    rf_clf = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False)
    rf_clf.fit(X_train, y_train)
    rf_scores.append({
        'model': model_name,
        'best_score': rf_clf.best_score_,
        'best_params': rf_clf.best_params_,
        'cv_results': rf_clf.cv_results_
    })
rf_df = pd.DataFrame(rf_scores, columns=['model', 'best_score', 'best_params','cv_results'])

# Print the updated DataFrame
print(rf_df)
```



	N_Estimator	Mean_Test_Score	Std_Test_Score	Mean_Fit_Time	Std_Fit_Time
0	10	0.908925	0.000798	9.052318	0.194292
1	100	0.909544	0.000643	89.092462	0.837290
2	200	0.909573	0.000682	178.669071	1.368887



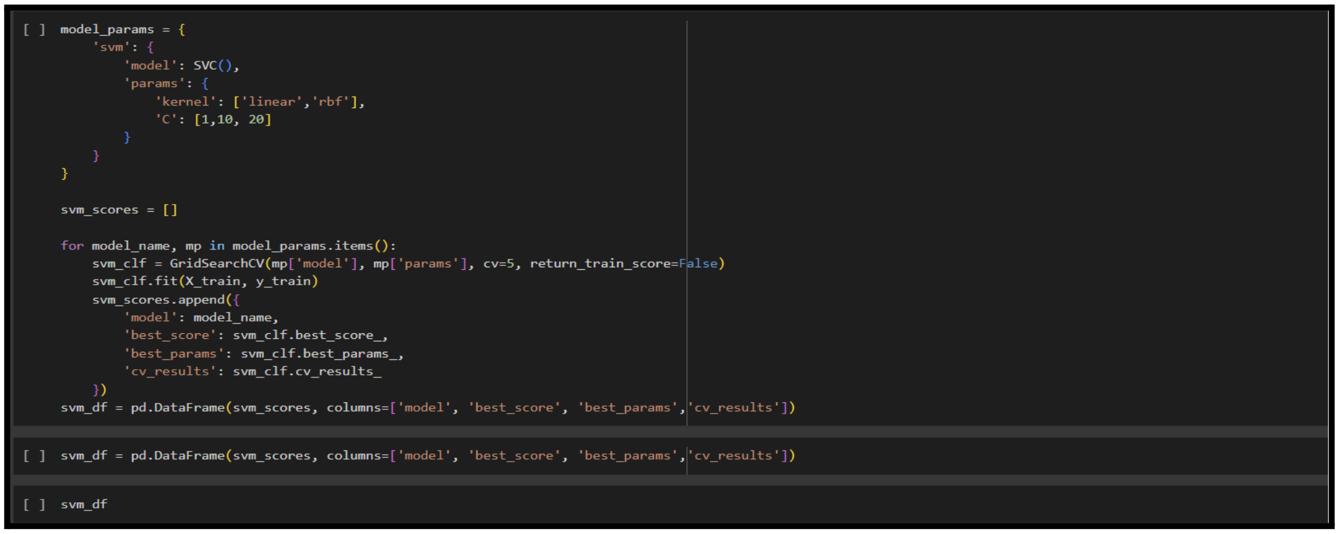


Support Vector Machine:

A supervised learning algorithm that can be used for classification or regression tasks. It finds a hyperplane that separates the data into different classes while maximizing the margin between the classes. Hyperparameter tuning with different kernels such as linear, radial basis function (RBF) for different values of C is done.

While doing hyperparameter tuning here, we have experimented with the different values of c and kernel. Some of the kernels used are as follows:

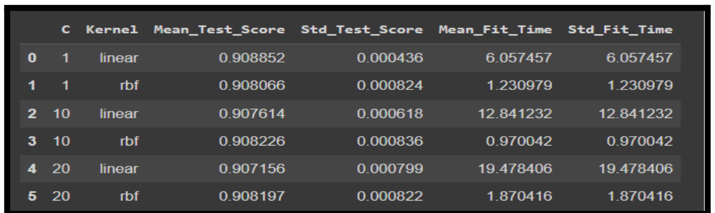
Linear: Used for linearly separable datasets where the classes can be separated by a straight line.
RBF : Used where the classes are not easily separable and may have complex, non-linear relationships.



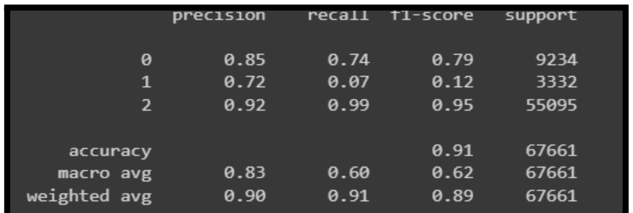
```
[ ] model_params = {
        'svm': {
            'model': SVC(),
            'params': {
                'kernel': ['linear','rbf'],
                'C': [1,10, 20]
            }
        }
    }

    svm_scores = []

    for model_name, mp in model_params.items():
        svm_clf = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False)
        svm_clf.fit(X_train, y_train)
        svm_scores.append({
            'model': model_name,
            'best_score': svm_clf.best_score_,
            'best_params': svm_clf.best_params_,
            'cv_results': svm_clf.cv_results_
        })
    svm_df = pd.DataFrame(svm_scores, columns=['model', 'best_score', 'best_params','cv_results'])
```

```
[ ] svm_df = pd.DataFrame(svm_scores, columns=['model', 'best_score', 'best_params','cv_results'])
```

```
[ ] svm_df
```



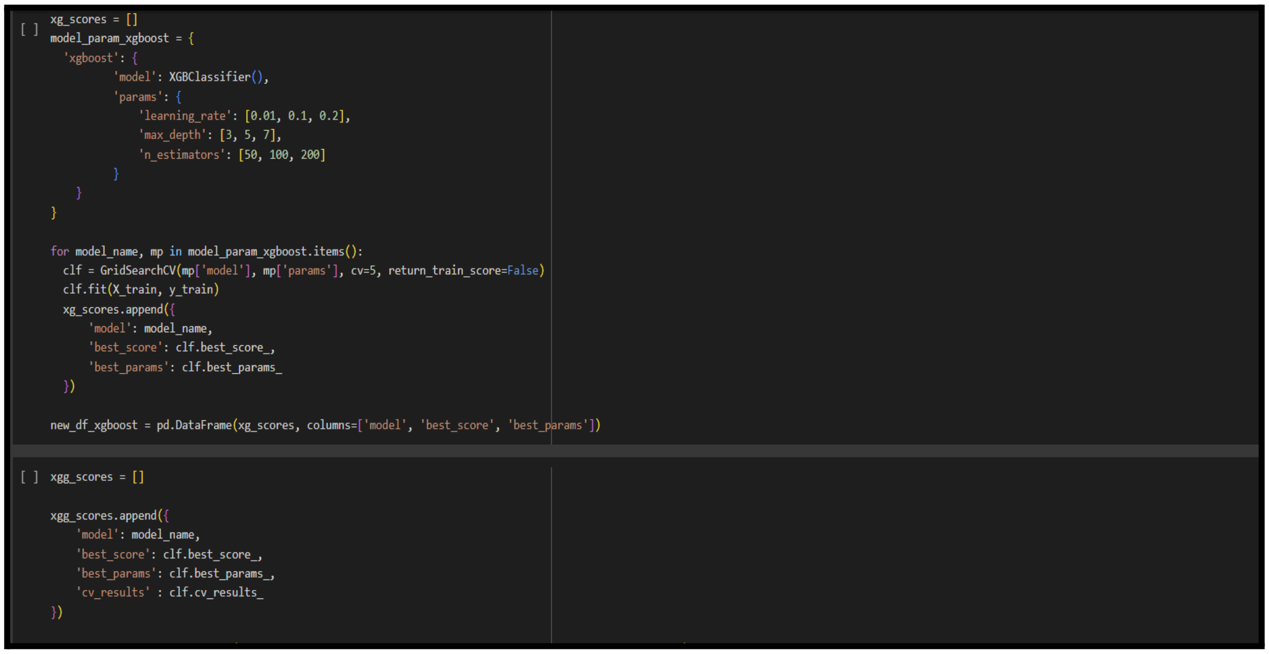
	C	Kernel	Mean_Test_Score	Std_Test_Score	Mean_Fit_Time	Std_Fit_Time
0	1	linear	0.908852	0.000436	6.057457	6.057457
1	1	rbf	0.908066	0.000824	1.230979	1.230979
2	10	linear	0.907614	0.000618	12.841232	12.841232
3	10	rbf	0.908226	0.000836	0.970042	0.970042
4	20	linear	0.907156	0.000799	19.478406	19.478406
5	20	rbf	0.908197	0.000822	1.870416	1.870416



```
              precision    recall  f1-score   support

           0       0.85      0.74      0.79      9234
           1       0.72      0.07      0.12      3332
           2       0.92      0.99      0.95     55095

    accuracy                           0.91     67661
   macro avg       0.83      0.60      0.62     67661
weighted avg       0.90      0.91      0.89     67661
```

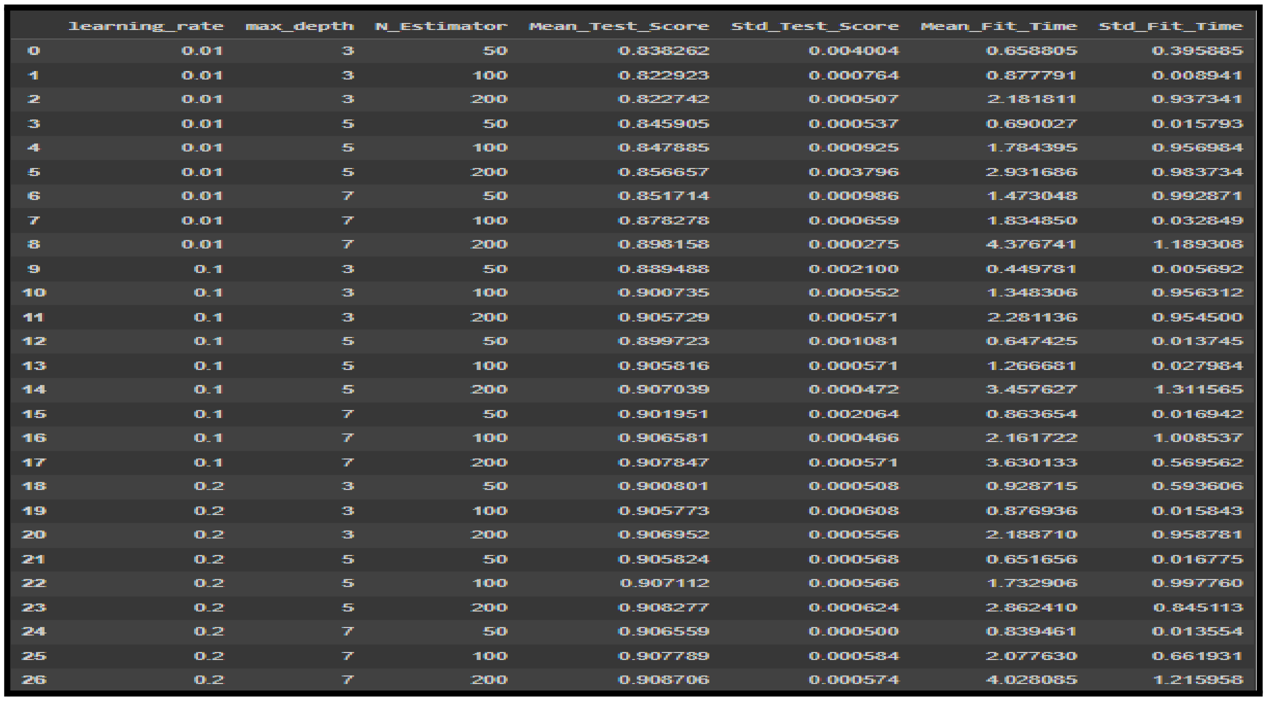
XGBoost:

An ensemble learning algorithm that combines the predictions of multiple weak learners (typically decision trees) to create a stronger and more robust model. It builds trees sequentially, with each tree attempting to correct the errors of the previous ones



```
[ ] xg_scores = []
    model_param_xgboost = {
      'xgboost': {
              'model': XGBClassifier(),
              'params': {
                  'learning_rate': [0.01, 0.1, 0.2],
                  'max_depth': [3, 5, 7],
                  'n_estimators': [50, 100, 200]
              }
          }
    }

    for model_name, mp in model_param_xgboost.items():
      clf = GridSearchCV(mp['model'], mp['params'], cv=5, return_train_score=False)
      clf.fit(X_train, y_train)
      xg_scores.append({
          'model': model_name,
          'best_score': clf.best_score_,
          'best_params': clf.best_params_
      })

    new_df_xgboost = pd.DataFrame(xg_scores, columns=['model', 'best_score', 'best_params'])
```

```
[ ] xgg_scores = []

    xgg_scores.append({
        'model': model_name,
        'best_score': clf.best_score_,
        'best_params': clf.best_params_,
        'cv_results' : clf.cv_results_
    })
```



	learning_rate	max_depth	N_Estimator	Mean_Test_Score	Std_Test_Score	Mean_Fit_Time	Std_Fit_Time
0	0.01	3	50	0.838262	0.004004	0.658805	0.395885
1	0.01	3	100	0.822923	0.000764	0.877791	0.008941
2	0.01	3	200	0.822742	0.000507	2.181811	0.937341
3	0.01	5	50	0.845905	0.000537	0.690027	0.015793
4	0.01	5	100	0.847885	0.000925	1.784395	0.956984
5	0.01	5	200	0.856657	0.003796	2.931686	0.983734
6	0.01	7	50	0.851714	0.000986	1.473048	0.992871
7	0.01	7	100	0.878278	0.000659	1.834850	0.032849
8	0.01	7	200	0.898158	0.000275	4.376741	1.189308
9	0.1	3	50	0.889488	0.002100	0.449781	0.005692
10	0.1	3	100	0.900735	0.000552	1.348306	0.956312
11	0.1	3	200	0.905729	0.000571	2.281136	0.954500
12	0.1	5	50	0.899723	0.001081	0.647425	0.013745
13	0.1	5	100	0.905816	0.000571	1.266681	0.027984
14	0.1	5	200	0.907039	0.000472	3.457627	1.311565
15	0.1	7	50	0.901951	0.002064	0.863654	0.016942
16	0.1	7	100	0.906581	0.000466	2.161722	1.008537
17	0.1	7	200	0.907847	0.000571	3.630133	0.569562
18	0.2	3	50	0.900801	0.000508	0.928715	0.593606
19	0.2	3	100	0.905773	0.000608	0.876936	0.015843
20	0.2	3	200	0.906952	0.000556	2.188710	0.958781
21	0.2	5	50	0.905824	0.000568	0.651656	0.016775
22	0.2	5	100	0.907112	0.000566	1.732906	0.997760
23	0.2	5	200	0.908277	0.000624	2.862410	0.845113
24	0.2	7	50	0.906559	0.000500	0.839461	0.013554
25	0.2	7	100	0.907789	0.000584	2.077630	0.661931
26	0.2	7	200	0.908706	0.000574	4.028085	1.215958

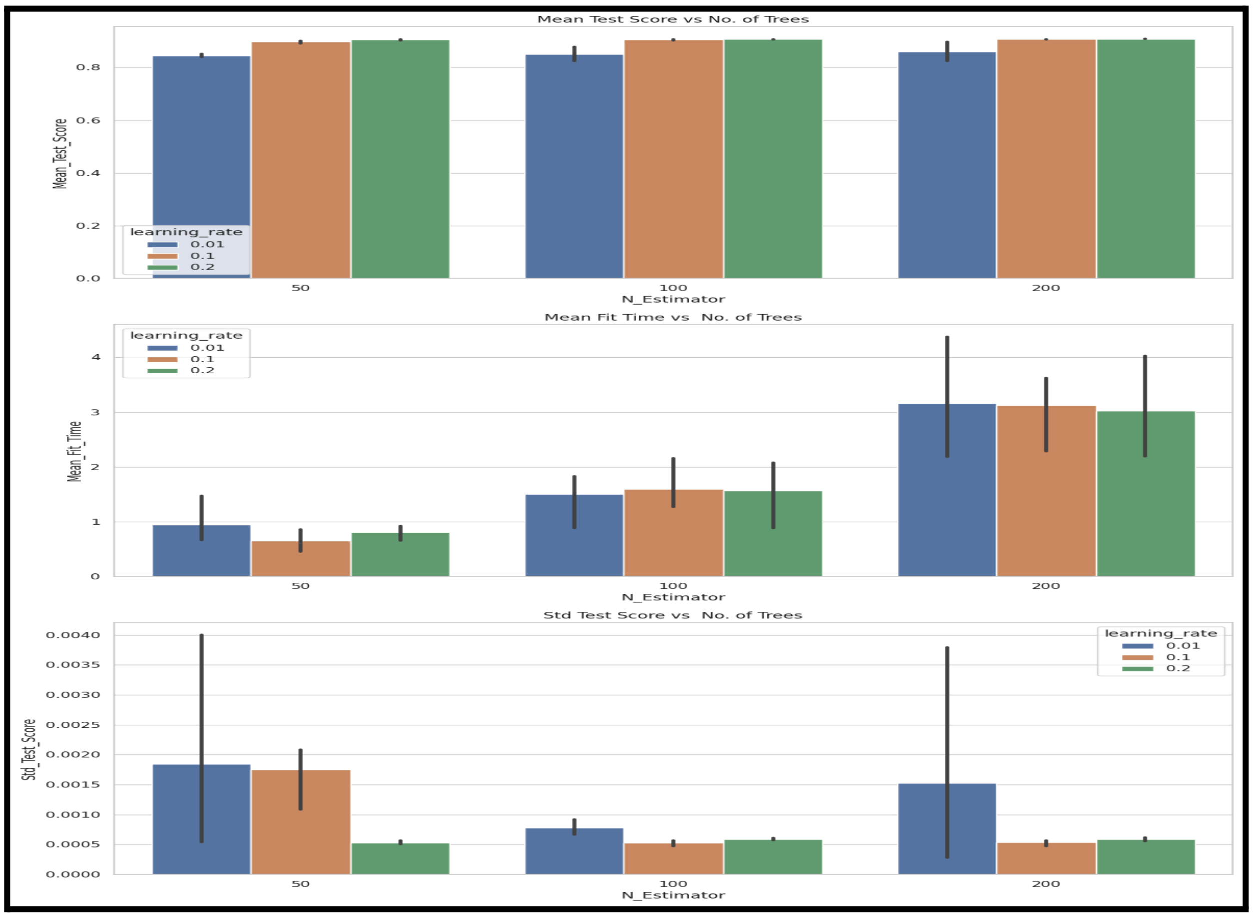
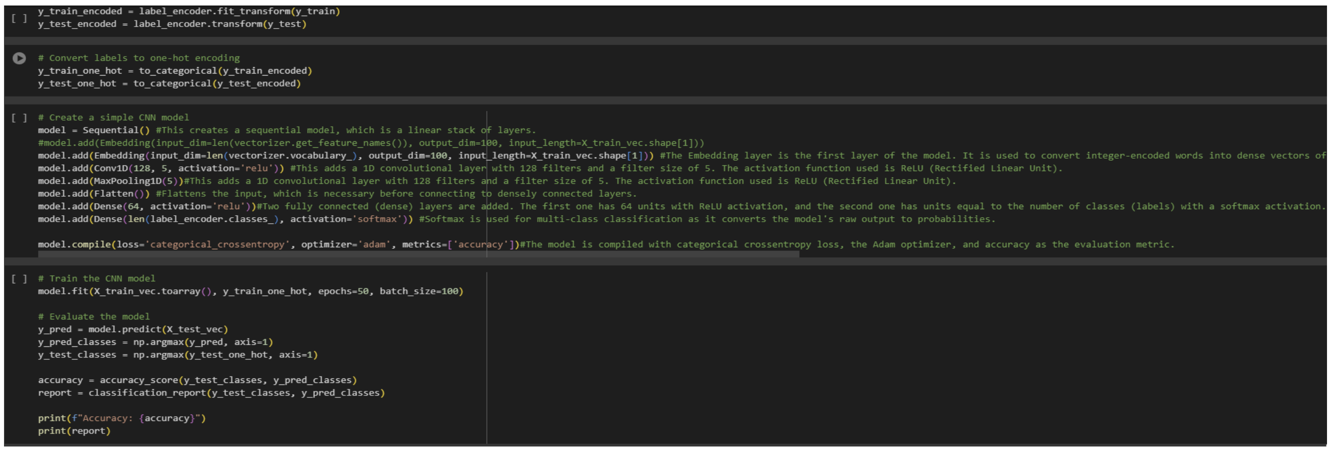


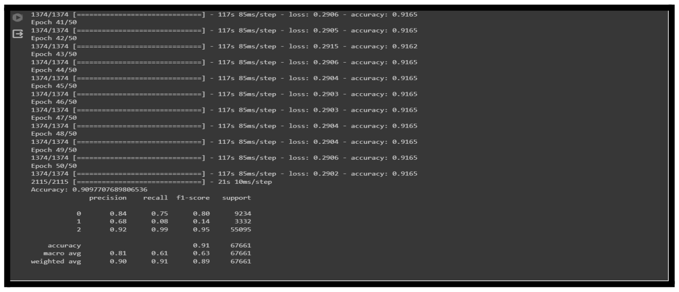
Fig. 10

Convolution Neural Network:

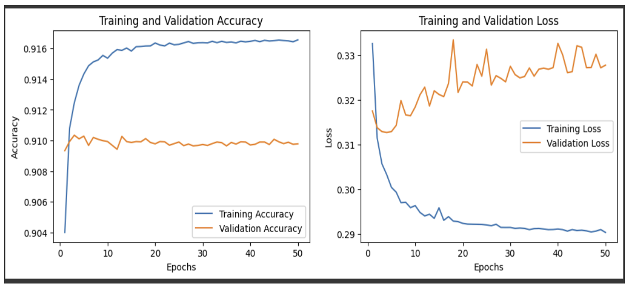
CNNs are deep learning models specifically designed for analyzing visual data like images or videos. They use filters to detect patterns hierarchically. Here CNNs can capture local patterns in phrases or combinations of words, providing a deep learning approach to sentiment analysis. We have used Sequential Model. We have added 1D convolutional layer with 128 filters and a filter size of 5. This code is executed for 50 epochs. Details about code are explained in the comments below:



```
[ ] y_train_encoded = label_encoder.fit_transform(y_train)
    y_test_encoded = label_encoder.transform(y_test)

    # Convert labels to one-hot encoding
    y_train_one_hot = to_categorical(y_train_encoded)
    y_test_one_hot = to_categorical(y_test_encoded)

[ ] # Create a simple CNN model
    model = Sequential() #This creates a sequential model, which is a linear stack of layers.
    #model.add(Embedding(input_dim=len(vectorizer.get_feature_names()), output_dim=100, input_length=X_train_vec.shape[1]))
    model.add(Embedding(input_dim=len(vectorizer.vocabulary_), output_dim=100, input_length=X_train_vec.shape[1])) #The Embedding layer is the first layer of the model. It is used to convert integer-encoded words into dense vectors of
    model.add(Conv1D(128, 5, activation='relu')) #This adds a 1D convolutional layer with 128 filters and a filter size of 5. The activation function used is ReLU (Rectified Linear Unit).
    model.add(MaxPooling1D(5))#This adds a 1D convolutional layer with 128 filters and a filter size of 5. The activation function used is ReLU (Rectified Linear Unit).
    model.add(Flatten()) #Flattens the input, which is necessary before connecting to densely connected layers.
    model.add(Dense(64, activation='relu'))#Two fully connected (dense) layers are added. The first one has 64 units with ReLU activation, and the second one has units equal to the number of classes (labels) with a softmax activation.
    model.add(Dense(len(label_encoder.classes_), activation='softmax')) #Softmax is used for multi-class classification as it converts the model's raw output to probabilities.

    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])#The model is compiled with categorical crossentropy loss, the Adam optimizer, and accuracy as the evaluation metric.

[ ] # Train the CNN model
    model.fit(X_train_vec.toarray(), y_train_one_hot, epochs=50, batch_size=100)

    # Evaluate the model
    y_pred = model.predict(X_test_vec)
    y_pred_classes = np.argmax(y_pred, axis=1)
    y_test_classes = np.argmax(y_test_one_hot, axis=1)

    accuracy = accuracy_score(y_test_classes, y_pred_classes)
    report = classification_report(y_test_classes, y_pred_classes)

    print(f"Accuracy: {accuracy}")
    print(report)
```

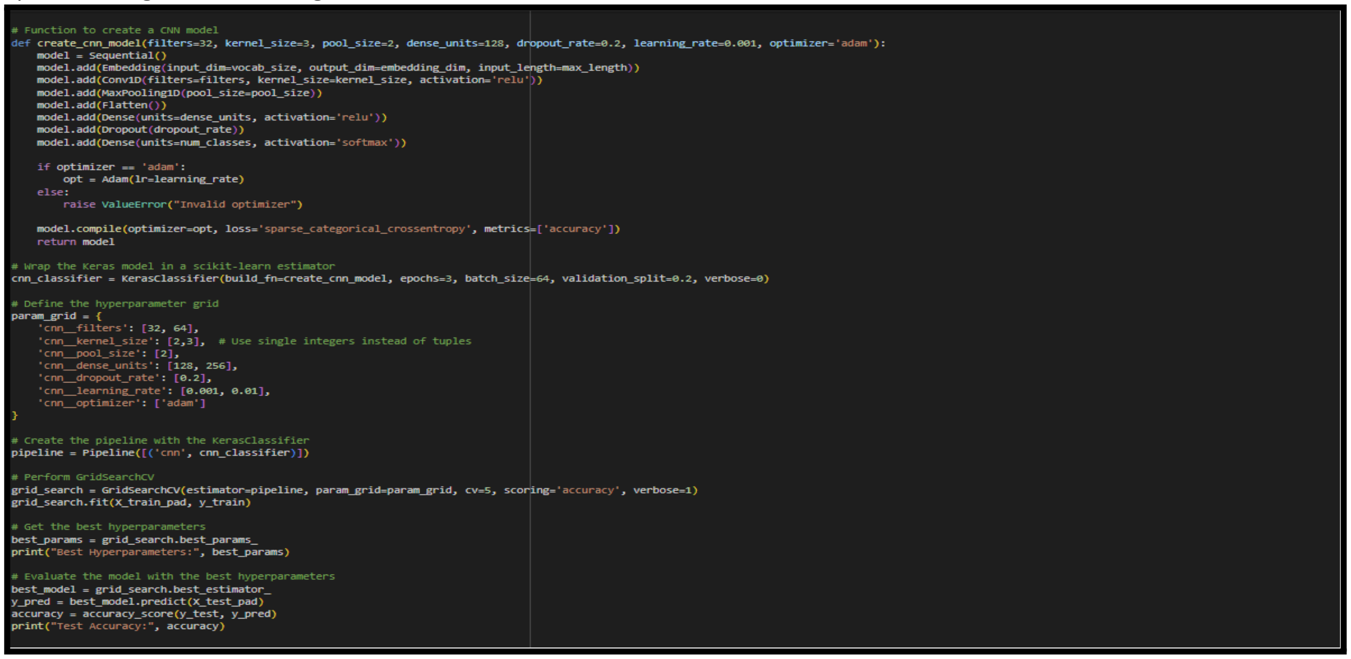


```
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2906 - accuracy: 0.9165
Epoch 41/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2905 - accuracy: 0.9165
Epoch 42/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2915 - accuracy: 0.9162
Epoch 43/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2906 - accuracy: 0.9165
Epoch 44/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2904 - accuracy: 0.9165
Epoch 45/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2903 - accuracy: 0.9165
Epoch 46/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2903 - accuracy: 0.9165
Epoch 47/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2904 - accuracy: 0.9165
Epoch 48/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2904 - accuracy: 0.9165
Epoch 49/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2906 - accuracy: 0.9165
Epoch 50/50
1374/1374 [==============================] - 117s 85ms/step - loss: 0.2902 - accuracy: 0.9165
2115/2115 [==============================] - 21s 10ms/step
Accuracy: 0.9097707689806536
              precision    recall  f1-score   support

           0       0.84      0.75      0.80      9234
           1       0.68      0.08      0.14      3332
           2       0.92      0.99      0.95     55095

    accuracy                           0.91     67661
   macro avg       0.81      0.61      0.63     67661
weighted avg       0.90      0.91      0.89     67661
```

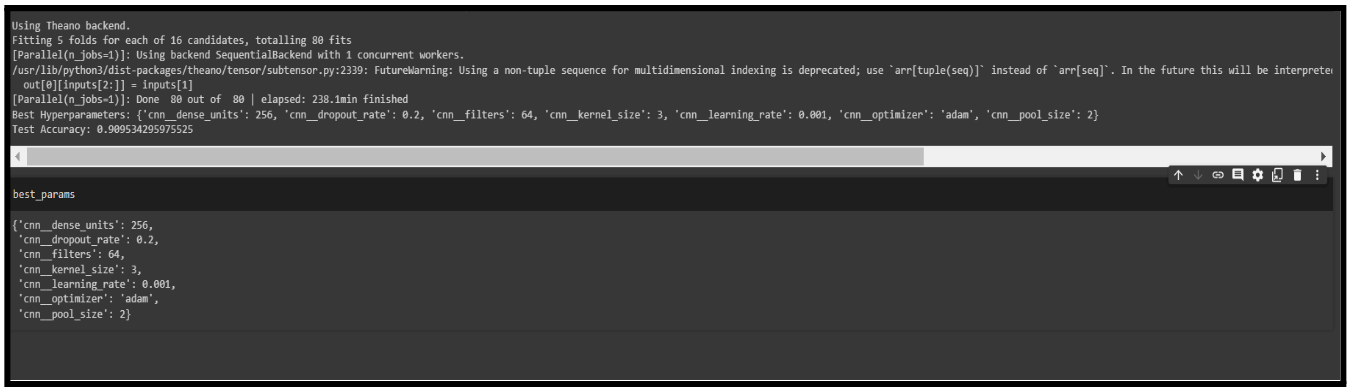


Hyperparameter Tuning:

We also tried to do hyperparameter tuning by changing various parameters such as filters, kernel size, learning rate etc. As GridSearchCV algorithm is being performed with this number of parameters changes we were able to execute code for 3 epochs . We got the following result.



```python
# Function to create a CNN model
def create_cnn_model(filters=32, kernel_size=3, pool_size=2, dense_units=128, dropout_rate=0.2, learning_rate=0.001, optimizer='adam'):
    model = Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length))
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='relu'))
    model.add(MaxPooling1D(pool_size=pool_size))
    model.add(Flatten())
    model.add(Dense(units=dense_units, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Dense(units=num_classes, activation='softmax'))

    if optimizer == 'adam':
        opt = Adam(lr=learning_rate)
    else:
        raise ValueError("Invalid optimizer")

    model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# Wrap the Keras model in a scikit-learn estimator
cnn_classifier = KerasClassifier(build_fn=create_cnn_model, epochs=3, batch_size=64, validation_split=0.2, verbose=0)

# Define the hyperparameter grid
param_grid = {
    'cnn__filters': [32, 64],
    'cnn__kernel_size': [2,3],  # Use single integers instead of tuples
    'cnn__pool_size': [2],
    'cnn__dense_units': [128, 256],
    'cnn__dropout_rate': [0.2],
    'cnn__learning_rate': [0.001, 0.01],
    'cnn__optimizer': ['adam']
}

# Create the pipeline with the KerasClassifier
pipeline = Pipeline([('cnn', cnn_classifier)])

# Perform GridSearchCV
grid_search = GridSearchCV(estimator=pipeline, param_grid=param_grid, cv=5, scoring='accuracy', verbose=1)
grid_search.fit(X_train_pad, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

# Evaluate the model with the best hyperparameters
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test_pad)
accuracy = accuracy_score(y_test, y_pred)
print("Test Accuracy:", accuracy)
```



```
Using Theano backend.
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
/usr/lib/python3/dist-packages/theano/tensor/subtensor.py:2339: FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interprete
  out[0][inputs[2:]] = inputs[1]
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed: 238.1min finished
Best Hyperparameters: {'cnn__dense_units': 256, 'cnn__dropout_rate': 0.2, 'cnn__filters': 64, 'cnn__kernel_size': 3, 'cnn__learning_rate': 0.001, 'cnn__optimizer': 'adam', 'cnn__pool_size': 2}
Test Accuracy: 0.909534295975525
```

```
best_params
```

```
{'cnn__dense_units': 256,
 'cnn__dropout_rate': 0.2,
 'cnn__filters': 64,
 'cnn__kernel_size': 3,
 'cnn__learning_rate': 0.001,
 'cnn__optimizer': 'adam',
 'cnn__pool_size': 2}
```

Conclusions:

Out of all the six algorithms that we tested for our dataset, the CNN algorithm proved to be the best suited for our use case yielding the highest accuracy of 90.98% that also for only 50 epochs without hyperparameter tuning. If better computational resources were present, one could get much better results for CNN with hyperparameter tuning for more number of epochs.

One of the limitation we faced was even though we incorporated hyperparameter tuning for all the models, the results did not vary substantially. This could be due to the limited amount of data we acquired and the way it was categorized.

As part of future scope, we would consider a larger amount of data with more features, where the labels are segregated proportionately so that the algorithms can produce better results along with significant differences in each algorithm. This would help us to better gauge the usability of that model for our data.
Also we plan to use better computational resources, in order to perform hyperparameter tuning with more parameter changes and more number of epochs in cases such as CNN.

Team Member Contribution:

Apeksha Shrirao:

- Research topic for development and its corresponding dataset.
- Performed data cleaning, EDA and extracting important insights from dataset
- Implemented Logistic Regression on the dataset after performing research if this algorithm can be used on Multi-class classification or not.
- Researched about hyperparameter tuning and helped team members to understand its concept and code.

Anuja Merwade:

- Performed EDA along with Apeksha to draw insights from the cleaned data.
- Trained the Multinomial Naive Bayes and Random Forest Classifier algorithms on the dataset to study the performance of these algorithms.
- Collectively brainstormed on performing hyperparameter tuning for all the algorithms.
- Researched and helped in finding and configuring systems that can be used to perform hyperparameter tuning,

Vasu Sharma:

- Implemented Support Vector Machine, XGBoost and Convolution Neural Network algorithms on the dataset
- Implemented Hyperparameter tuning for respective Machine Learning algorithms of the project.
- Researched on various parameters that can be changed and used during hyperparameter tuning for the respective ML algorithms.
- Created visualizations for the algorithms before and after implementing hyperparameter tuning on the dataset.

References:

1. Twitter as a corpus for Sentiment Analysis and Opinion Mining by Alexander Pak, Patrick Paroubek published in 2010. Published in Proceedings of the Seventh International Conference on Language Resources and Evaluation.
2. Sentiment Analysis of Twitter data by Sahar A. El_Rahman,Feddah Alhumaidi AlOtaibi,Wejdan Abdullah AlShehri. Published in 2019 at the International Conference on Computer and Information Science(ICCIS).
3. Twitter sentiment analysis using deep learning methods by Adyan Marendra Ramadhani. Published in 2017 at International Annual Engineering Seminar(InAES)
4. Twitter Sentiment Analysis with Deep Convolutional Neural Networks by Aliaksei Severyn, Alesandro Moschitti. It was published on Aug 9 2015 in Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval.
5. Sentimental analysis on Twitter by EUGENIO MARTÍNEZ-CÁMARA,M. TERESA MARTÍN-VALDIVIA,L. ALFONSO UREÑA-LÓPEZ and A RTURO MONTEJO-RÁEZ. It was published on 27 November 2012. It was published online in Cambridge University Press