Final Project Documentation

Topic:- **To analyze the given dataset and figure out if the tweet was real disaster related or not by the binary data in the Target column.**

Group Members:

1. Mohit Dale (md685@njit.edu)
2. Maulik Patel(mp766@njit.edu)
3. Myles Dsouza (mad35@njit.edu)

Tool used: Jupyter notebook
Language used: python (libraries: pandas, numpy, sklearn, seaborn, plotly, etc.)
Steps: Data-preprocessing, Data-cleaning, EDA and modeling and prediction.

Introduction:

This project asked us to predict the probability that an online tweet about a disaster is real or not as denoted by the binary column. We were given the data that contained ID, keywords in tweets, location of the tweets, text content in the tweets and the target column. Based on the target column and all the other columns provided we had to predict whether the tweet is '1' i.e real or '0' i.e fake.

Data pre-processing:

First and foremost, we imported several libraries in preparation for this assignment. We imported the libraries 'numpy' and 'pandas', 'matplotlib', 'seaborn', 'sklearn' as ways to access and manipulate our dataframe.

We were given several data descriptions and had to explore these data points further in order to successfully decipher if the tweet was legitimate or not. These were Location, which was the see where the tweet originated from, Keyword, which contained a particular keyword from the text and Text, which was the one we decided to use as our X as it contained the data contents of the tweet. Since this was a dataset with not many columns we did not need to waste time in using a multitude of combinations to fulfill our EDA and lead us to feature engineering and ultimately the machine learning model.

Data Cleaning:

We first started by figuring out how many null values we had in the dataset. The dataset contains 7613 rows and 5 columns. We found that there were 61 null values for the column “keyword” and 2533 for the column “location”. We just want to get an idea of how consistent the data we have is and were not planning on dropping too many columns as we did not have many to begin with. The “location” column has data such as Houston, TX and New york, NY even though there is a value such as USA and so we tried our best to include all these cities which are in the USA to fall under the USA value and the same of other countries where the Country and cities are different.

EDA

We begin our EDA by first comparing the tweets that are valid or not by creating a countplot shown as a bar graph, based on the “target” column. We then check the summary of the statistical functions for the dataset to see if there are any outliers or if it is useful to use the summary stat function in the first place. Now just to be sure if there are any duplicates in the dataset, we use the following code to remove any duplicates that may be present:

df.drop_duplicates(subset=['text', 'target'], keep='first')

We then start by creating variables for the dataset containing real or fake tweets using the “target” column. We then create plots to see the keywords appearing the most in the real and fake tweets for the top 20 keywords. We realized that the top 20 most frequently used words for real and fake tweets were vastly different. We then create plots to see the volume of real and fake tweets by the country they originated from and see that the USA was the leader for both.

Feature Engineering:

For feature engineering, we decided that “text” would be the column to use for our Machine learning algorithm and so we had to clean up the column as it consisted of blank spaces, https links, special characters, numbers and many more things. So in order to make the column consistent we created a function called new_text that would clean up the data in the “text” column and then we added the column containing the cleaned text into our original dataset. The function we created is as follows:

```
def clean_text(text):
    text= text.lower()
    text= re.sub('[0-9]', '', text)
    text  = "".join([char for char in text if char not in string.punctuation])
    tokens = word_tokenize(text)
    text = ' '.join(tokens)
    tokens=[word for word in tokens if word not in stopwords.words('english')]
```

return text

The code below shows how we added the cleaned column into the original dataset:

df["new_text"]=df["text"].apply(clean_text)

Modeling:

We imported **LogisticRegression, RandomForestClassifier, GradientBoostingClassifier, DecisionTreeClassifier, and BaggingClassifier** from the **sklearn** library. You might be wondering why we chose these models. The main reason for this was that each individual estimated the model differently. This was the finest answer to our inquiry on how we know this model is the best. We used the roc auc score and accuracy functions to provide a visual representation of how good the results were. For our improvement model, we imported train test split and **GridSearchCV** to evaluate which parameters are optimal for the tuned model. We needed StratifiedShuffleSplit because it was one of the project's most important requirements. To achieve the desired outcome of shuffling the data 10 times and separating it with a test size of 10%, we had to configure StratifiedShuffleSplit. Last but not least, there's the TfidfVectorizer import, which will be covered in more detail later.

Before we started modeling, we wanted to create a function that would allow us to quickly implement several models. That's where the evaluate model performance function was created by us. There will be a total of 6 parameters for this function. Two of them are X and Y, which are our independent and dependent features, followed by model_object and model_name, which are the model and its name, run_times, which is an int representing the number of times you want it to run, and finally the test size for the dataset.

```
def evaluate_model_performance(X,y,model_object,model_name, run_times = 10, test_size = 0.1):

    '''
    It's useful to understand how TF-IDF works so that you can gain a better understanding of how machine learning algorithm
    function. While machine learning algorithms traditionally work better with numbers, TF-IDF algorithms help them decipher
    by allocating them a numerical value or vector.This has been revolutionary for machine learning,
    especially in fields related to NLP such as text analysis.

    '''

    print(f"\t\t=====\{model_name} Results=====\n")
    scores = []
    auc = []
    sss = StratifiedShuffleSplit(n_splits=run_times, test_size=test_size, random_state=0)  # shuffling the data
    rf = model_object
    i = 0
    for train_index, test_index in sss.split(X, y):
        i+=1
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        # Fit and transform the training data to a document-term matrix using TfidfVectorizer
        tfidf = TfidfVectorizer()
        X_train_tfidf = tfidf.fit_transform(np.array(X_train.values))
        X_test_tfdif = tfidf.transform(X_test)

        rf.fit(X_train_tfidf, y_train)
        pred = rf.predict(X_test_tfdif)
        ROC_AUC = roc_auc_score(y_test,rf.predict_proba(X_test_tfdif)[:, 1]) # checking AUC
        accuracy = accuracy_score(y_test, pred) # checking accuracy
        scores.append(accuracy)
        auc.append(ROC_AUC)
        print(f"for {i} execution : Accuracy={round(accuracy,3)}, AUC={round(ROC_AUC,3)}")


    # get accurracy of each prediction
    print("Average Accuracy :",round(np.mean(scores),3))
    print("Average AUC :",round(np.mean(auc),3))
```

The **StratifiedShuffleSplit** is used in this function, with the parameters n times being the number of run times, test size being the number of test size, and random state being 0. The randomized splits will be returned to us ten times, with 90 % training and 10% on testing. This is then put into a loop. After that, the model is trained for each number until it reaches the run times number. More specifically, we have the **TfidfVectorizer** function, which tells us how relevant a word is in a document. We need TfidfVectorizer because we are using the 'text' column as a feature, and Tfidf assigns a ranking to each word. Last but not least, all ten iterations are saved in a list, and the user is given an average of the entire run.
The RandomForestClassifier was the first model we used. Then there's LogisticRegression, then GradientBoostingClassifier, BaggingClassifier, SVM, and finally DecisionTreeClassifier. There were two runs in particular that stood out from the rest.

```
lr = LogisticRegression()
evaluate_model_performance(X,y,lr,"LogisticRegression",run_times = 10, test_size = 0.1)
```

```
               =====LogisticRegression Results=====

for 1 execution : Accuracy=0.827, AUC=0.88
for 2 execution : Accuracy=0.791, AUC=0.856
for 3 execution : Accuracy=0.79, AUC=0.847
for 4 execution : Accuracy=0.77, AUC=0.835
for 5 execution : Accuracy=0.769, AUC=0.836
for 6 execution : Accuracy=0.79, AUC=0.861
for 7 execution : Accuracy=0.808, AUC=0.87
for 8 execution : Accuracy=0.811, AUC=0.872
for 9 execution : Accuracy=0.795, AUC=0.847
for 10 execution : Accuracy=0.793, AUC=0.841
Average Accuracy : 0.794
Average AUC : 0.854
```

```
svc = SVC(probability=True)
evaluate_model_performance(X,y,svc,"SVM",run_times = 10, test_size = 0.1)
```

```
               =====SVM Results=====

for 1 execution : Accuracy=0.816, AUC=0.874
for 2 execution : Accuracy=0.791, AUC=0.847
for 3 execution : Accuracy=0.791, AUC=0.85
for 4 execution : Accuracy=0.778, AUC=0.836
for 5 execution : Accuracy=0.778, AUC=0.825
for 6 execution : Accuracy=0.794, AUC=0.865
for 7 execution : Accuracy=0.802, AUC=0.867
for 8 execution : Accuracy=0.823, AUC=0.869
for 9 execution : Accuracy=0.798, AUC=0.844
for 10 execution : Accuracy=0.785, AUC=0.841
Average Accuracy : 0.796
Average AUC : 0.852
```

SVM and LogisticRegression are the two methods. We decided to hypertune the SVM and LogisticRegression model to see if any improvements could be made.

GridSearchCV is a tool that can be used to test parameters of models to see the best parameters for the model for the dataset that was provided. We provided the valid parameters to the GridSearchCV and model to find out the best parameters for Support Vector Machine were {'C': 10, 'gamma': 1, 'kernel': 'rbf'} and for Linear Regression were {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}.

```
svc = SVC(probability=True, C=10.0, gamma=1, kernel='rbf')
evaluate_model_performance(X,y,svc,"SVM",run_times = 10, test_size = 0.1)
```

```
=====SVM Results=====

for 1 execution : Accuracy=0.807, AUC=0.865
for 2 execution : Accuracy=0.787, AUC=0.841
for 3 execution : Accuracy=0.782, AUC=0.84
for 4 execution : Accuracy=0.78, AUC=0.829
for 5 execution : Accuracy=0.78, AUC=0.815
for 6 execution : Accuracy=0.785, AUC=0.856
for 7 execution : Accuracy=0.806, AUC=0.851
for 8 execution : Accuracy=0.812, AUC=0.867
for 9 execution : Accuracy=0.785, AUC=0.83
for 10 execution : Accuracy=0.787, AUC=0.835
Average Accuracy : 0.791
Average AUC : 0.843
```

```
lr = LogisticRegression(C=1.0, penalty='l2', solver='lbfgs')
evaluate_model_performance(X,y,lr,"LogisticRegression",run_times = 10, test_size = 0.1)
```

```
=====LogisticRegression Results=====

for 1 execution : Accuracy=0.827, AUC=0.88
for 2 execution : Accuracy=0.791, AUC=0.856
for 3 execution : Accuracy=0.79, AUC=0.847
for 4 execution : Accuracy=0.77, AUC=0.835
for 5 execution : Accuracy=0.769, AUC=0.836
for 6 execution : Accuracy=0.79, AUC=0.861
for 7 execution : Accuracy=0.808, AUC=0.87
for 8 execution : Accuracy=0.811, AUC=0.872
for 9 execution : Accuracy=0.795, AUC=0.847
for 10 execution : Accuracy=0.793, AUC=0.841
Average Accuracy : 0.794
Average AUC : 0.854
```

Conclusion:

After setting the models to be tuned using GridSearchCV, we realized that there were no improvements to be made to the models. Our best results were the Average AUC of 10 runs

being 0.854 on the Linear Regression. Improvements could only be “possibly seen” if the StratifiedShuffleSplit method parameter random_state was changed to None instead of 0.