Cyberbullying Detection on Twitter Data Using Machine Learning Classifiers

Pradip Dhakal
Statistics and Data Science Department
University of Central Florida

***Abstract*—This study compares some of the popular machine learning techniques like Logistic Regression, Multinomial Naive Bayes, K-Nearest Neighbor, and Extreme Gradient Boosting to classify the tweets into three different categories: cyberbullying based on religion, cyberbullying based on ethnicity, or no cyberbullying. First, various data-cleaning approaches are used to clean the tweet data. After the data is clean and ready, the word embedding techniques, such as a bag of words and term frequency-Inverse document frequency, are used to convert the words into mathematical vectors. Finally, the model will be fitted using the combination of the above-mentioned word embedding techniques and machine learning algorithms.**

***Keywords*—Logistic Regression, Multinomial Naive Bayes, K-Nearest Neighbor, Extreme Gradient Boosting, Bag of Words, Term Frequency-Inverse Document Frequency**

I. Introduction

Social media has been popular for quite a long time. People from all around the world are able to communicate with each other, share their knowledge and thoughts, and know what's happening on another side of the world. Some of the popular social platforms are Facebook, Instagram, Twitter, YouTube, and Snapchat. Alongside these advantages, people argue with each other, show aggressive behaviors, leave negative and racist comments, and bully other people on different social platforms. Many people have been victims of these kinds of activities, leading to an adverse psychological impact on the victim's emotions. Many research studies have been proposed to mitigate these kinds of activities. Most researchers formulated this problem as a classification problem. The study by Dinakar et al. [1] performed binary classification to see whether the comments on YouTube could be classified as sensitive or not; they also performed multi-label classification to see what comments belong to what classes. Another study from Chavan and Shylaja [2] performed the binary classification, where they classified the texts as bullying texts and non-bullying texts. Similarly, the study from Dadvar et al. [3] incorporated the user's age and gender to improve the accuracy of cyberbullying detection. These articles motivated me to work in the area of cyberbullying detection.

Our study consists of three different classes: ethnicity-based cyberbullying with the label '1', religion-based cyberbullying with the label '2', and no cyberbullying with the label '0'. Our primary goal is to come up with the best model that classifies the words very accurately. The best model that we obtain from this study can be used in the social application platform to detect bullying words and warn the users to use alternative words.

II. Design Overview

This section explains the methodology and framework used in this study. In fig 1, we can see the flowchart of the overall design. First, the data is cleaned and pre-processed. Second, the cleaned data is used to create the word clouds for each class. Also, the train-test split is performed after the data pre-processing. Next, taking the split data, two different word embedding techniques: Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF), are used to convert the tokens into mathematical vectors. Finally, four different classification algorithms: Logistic Regression (LR), Multinomial Naive Bayes (MNB), K-Nearest Neighbor (KNN), and Extreme Gradient Boosting (XgBoost) are used with each word embedding technique to perform the multi-label classification. The results are then compared using the classification accuracy and F1 score metrics.

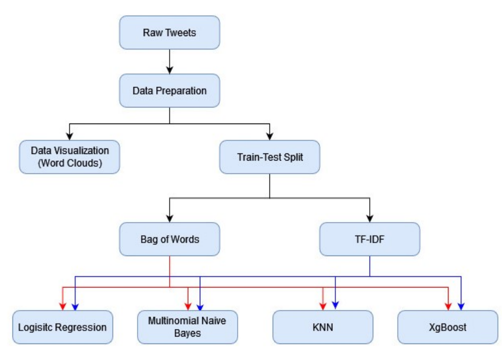

Fig. 1. Design Overview

III. Data

A. Data Information

The dataset used in this paper is available here to the public, and it was contributed by J. Wang et al. [4]. Among the available various classes, I am using the text data sets that belong to ethnicity, religion, and not cyberbullying classes.

The data is balanced for each class, and each class contains 8000 tweets. Since I am dealing with three classes, I have a total of 24,000 tweets to work with.

B. Data Preparation

One of the most important work for text analytics is to clean and prepare the data. If the data is not cleaned properly, it will lead to poor visualization of the data, and the model's accuracy will be impaired. The data cleaning process consists of various steps, which are described below.

1. Converting all the tweets to lowercase
This is required so that the same words with uppercase and lowercase are treated the same during the analysis.

2. Removing stopwords
Stopwords are words that don't really add meaning to the sentence, so it is the best idea to remove these words.

3. Removing URLs
Links to the websites do not add any significant insight to the data, so it is best to remove the links from the data.

4. Removing mentions
Mentions are basically usernames, so there is no significance in keeping this in the data.

5. Removing numeric data
The numbers in the tweets do not add any value to the data, so the numbers are removed.

6. Removing punctuations
The punctuations do not have any significance in the text data, so they are also removed.

7. Removing emojis
Tweets contain a lot of emojis; however, they need to be removed in text analytics and natural language processing.

8. Tokenization
The process of breaking sentences or paragraphs into meaningful words is known as tokenization. I tokenized all the tweets into individual words. Since I will be analyzing words individually, these words will be treated as a unigram in the analysis section. A unigram means that each token is treated independently of the other tokens that came before it.

9. Lemmatization
Lemmatization is the process of grouping words with the same root or lemma but various meaning derivatives or inflections so they can be studied as a single unit. For instance, lemmatizing the word 'doing' returns 'do.'

C. Data Visualization

After preparing the data, I prepared the word clouds from the tweets of each class.

1. Ethnicity-Based Cyberbullying Class

Fig. 2. Word Clouds for Ethnicity-Based Cyberbullying Tweets

From fig 2, we can see bullying and racist words such as nigger, white, black, racism, black, fuck, and so on. Larger words in the word cloud mean these are repeated more frequently. For instance, the words such as nigger, dumb, fuck, and Obama are repeated multiple times.

2. Religion-Based Cyberbullying Class

Fig. 3. Word Clouds for Ethnicity-Based Cyberbullying Tweets

From fig 3, we can see words related to bullying and religion. For instance, the words such as Muslim, idiot, Christian, woman, terrorist, Islam, India, and Hindu are repeated multiple times.

3.Non Cyberbullying Class

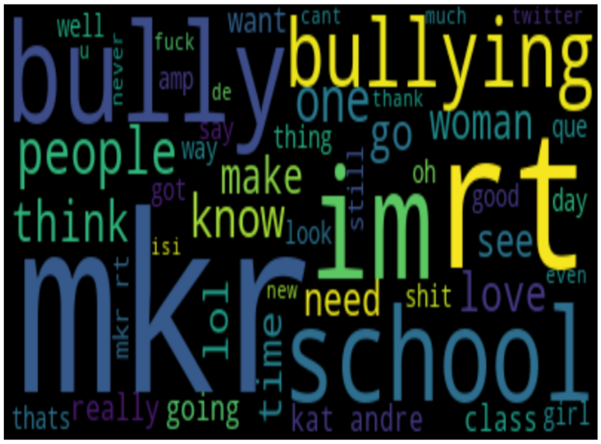

Fig. 4. Word Clouds for Non-Cyberbullying Tweets

From fig 4, we can see words that are mostly neutral in meaning. It looks like lemmatization is not able to catch 'bully' and 'bullying,' so we see both words in the world cloud. The most repeated words are bully, school, people, and know. Some of the words that are repeated multiple times but do not make sense to me are mkr, im, and rt. These words might be acronyms, or removing punctuation has changed their meaning.

D. Train-Test Split of Data

Since the data is cleaned and prepared, the next step is to divide the data into train data and test data. The data is split using a 70–30 train-test split ratio. The training data set contains 16800 tokens of tweets, and the testing data set contains 7200 tokens of tweets. The prime objective of this split is to evaluate the model's performance on data set that it has never seen beforehand.

IV. Analysis

A. Word Embedding Techniques

Raw texts cannot be used directly by machine learning algorithms; instead, the text must be transformed into numbers, more specifically, numerical vectors, so that we can perform various machine learning tasks on them. This process of converting text into numerical vectors is known as word embedding. This is also known as feature extraction or feature encoding since each token will be represented as a feature. After performing the word embedding, each row will be the individual tweets, and each column will be the feature. There are several word embedding techniques; however, I will use only two of them in this study.

1. Bag of Words

This feature extraction method describes the occurrence of the words in the documents. Each document represents a row, and each word or token represents a feature. If the token is present in the document, it gets a value of 1, and if it is absent, it gets a value of 0. This method results in a sparse matrix which is later used to create machine-learning models. In our study, we are classifying each word into their class, so we treated each token as a unigram while fitting this model. The bag of words has some limitations, such as this method is not able to assign large weights for tokens that have more semantic meaning and small weights for tokens with less semantic meaning; each token gets a value of 1 if they are present in the document. To overcome this issue, we can use another word embedding technique called Term Frequency-Inverse Document Frequency (TF-IDF).

2. Term Frequency-Inverse Document Frequency

This is another feature extraction method that is calculated by taking the product of term frequency and inverse document frequency.

$$w_{x,y} = tf_{x,y} * log(\frac{N}{df_x}) \tag{1}$$

where x is a term, y is a document, $tf_{x,y}$ is the frequency of term (x) in the document (y), N is the total number of documents, and df_x is the total number of the document containing the term (x).
This method also considers each token as a feature and each document as a row similar to a bag of words. However, the values in the sparse matrix are calculated using the formula from equation 1, and this formula assigns a large weight for tokens with more semantic meaning and small weights for tokens with less semantic meaning. Hence, this method overcomes the limitations of the bag of words (BoW) method.

B. Modeling

This section talks about the four different models that we fitted in this study. Each of the four models was used with both BoW and TF-IDF.

1. Logistic Regression

Logistic regression is a very simple linear model that is used in classification. The multiple logistic regression equation can be written as follows:

$$ln\left(\frac{\hat{p}}{(1-\hat{p})}\right) = b_0 + b_1 * X1 + ... + b_p * X_p$$

$$\hat{p} = \frac{exp(b_0 + b_1 * X1 + ... + b_p * X_p)}{1 + exp(b_0 + b_1 * X1 + ... + b_p * X_p)} \tag{2}$$

Using the equation 2, we can calculate the probability for each class. Since we have three classes, the class with the highest probability will be returned as a predicted class.

The logistic model was fitted with both BoW and TF-IDF on the training dataset. On predicting the results on the test dataset, the following results were obtained.

Logistic Regression		
Word Embedding	Accuracy	F1 Score
BoW	0.96	0.96
TF-IDF	0.96	0.96

TABLE I
LOGISTIC REGRESSION RESULTS

From the table I, both BoW and TF-IDF gave the same classification accuracy and F1 score with the logistic regression model.

Confusion Matrices:

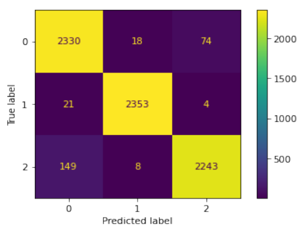

Fig. 5. Logistic Regression with Bag of Words

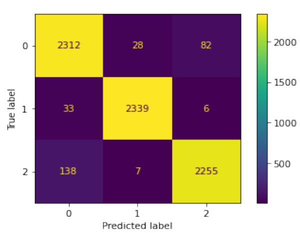

Fig. 6. Logistic Regression with TF-IDF

2. Multinomial Naive Bayes

The multinomial Naive Bayes algorithm is the variation of the traditional naive Bayes algorithm that is employed in text classification. The naive Bayes algorithm assumes conditional independence between each pair of features given the class variable. The classification rule for naive Bayes is given below.

$$P(y|x_1, x_2, ..., x_n) \propto P(y) \prod_{i=1}^{n} P(x_i|y)$$

$$\hat{y} = \underset{y}{argmax} P(y) \prod_{i=1}^{n} P(x_i|y)$$

The multinomial naive Bayes model was fitted with both BoW and TF-IDF on the training dataset. On predicting the results on the test dataset, the following results were obtained.

Multinomial Naive Bayes		
Word Embedding	Accuracy	F1 Score
BoW	0.88	0.87
TF-IDF	0.86	0.85

TABLE II
MULTINOMIAL NAIVE BAYES RESULTS

From the table II, the BoW performed slightly better than TF-IDF with the multinomial naive Bayes algorithm.

Confusion Matrices:

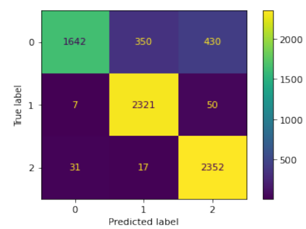

Fig. 7. Multinomial Naive Bayes with Bag of Words

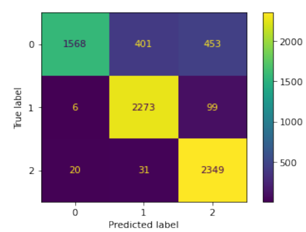

Fig. 8. Multinomial Naive Bayes with TF-IDF

3. K-Nearest Neighbor

This is another classification algorithm that finds the k nearest neighbors of a new point and labels the new point to the class which has a high number of points out of the k neighbors. The predicted value is given by the following equation.

$$\hat{y} = \frac{1}{K} \sum_{x_i \in N_k(x)} y_i \tag{3}$$

where $N_k(x)$ indicates k samples from the training data that are closest to x.

Using the equation 3, we can predict the class of each observation. The Minkowski distance is used to calculate the distance between the points to get the closest neighbors.

Before fitting the model, the optimal value for k needs to be calculated using both BoW and TF-IDF data. I used a 5-fold cross-validation method and obtain the optimal value for k to be 1 for both of them. The KNN model is now fitted with both BoW and TF-IDF on the training dataset. On predicting the results on the test dataset, the following results were obtained.

K-Nearest Neighbor			
Word Embedding	Best K	Accuracy	F1 Score
BoW	1	0.86	0.86
TF-IDF	1	0.46	0.40

TABLE III
K-NEAREST NEIGHBOR RESULTS

From the table III, the BoW performed way better than the TF-IDF with the K-Nearest Neighbor algorithm.

Confusion Matrices:

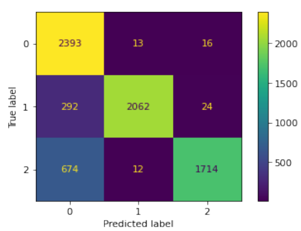

Fig. 9. K-Nearest Neighbor with Bag of Words

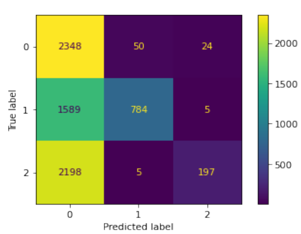

Fig. 10. K-Nearest Neighbor with TF-IDF

4. Extreme Gradient Boosting

Extreme gradient boosting is an improved version of Gradient Boosting Machines that creates more stable models by drastically lowering the likelihood of overfitting. Extreme gradient boosting is an ensemble learning technique that is basically built on the concept of a random forest algorithm. Ensemble learning is a method where multiple learners are employed to obtain better prediction accuracy; the model learns from previous mistakes at each iteration and reduces the loss.
Let $F = f_1, f_2, ..., f_n$ be the set of base learners, then the final prediction is given by

$$\hat{y_i} = \sum_{j=1}^{n} f_j(x_i)$$

A base learner is fitted to the negative gradient of the loss function with respect to the value from the previous iteration in order to get $f_j(xi)$ at each iteration.
The xgboost model was fitted with both BoW and TF-IDF on the training dataset using the default parameters. On predicting the results on the test dataset, the following results were obtained.

XgBoost		
Word Embedding	Accuracy	F1 Score
BoW	0.97	0.97
TF-IDF	0.97	0.97

TABLE IV
XGBOOST RESULTS

Confusion Matrices:

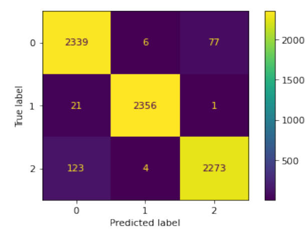

Fig. 11. Xgboost with Bag of Words

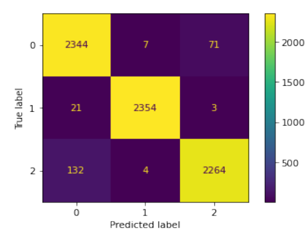

Fig. 12. Xgboost with TF-IDF

Next, a couple of hyperparameters: max depth and learning rate, were tuned to see if we could further improve the accuracy results.

Tuned Hyperparameters		
Word Embedding	max depth	learning rate
BoW	4	0.5
TF-IDF	6 (same as default)	0.3 (same as default)

TABLE V
XGBOOST HYPERPARAMETER TUNING

Table V lists the value of tuned max depth and learning rate. The tuned hyperparameter values were the same as the default values for TF-IDF; however, the tuned hyperparameter values for BoW differed. The xgboost model was again fitted with BoW using the tuned hyperparameter values, and the following result was obtained.

XgBoost with Tuned Hyperparameters		
Word Embedding	Accuracy	F1 Score
BoW	0.97	0.97
TF-IDF	0.97	0.97

TABLE VI
XGBOOST WITH TUNED HYPERPARAMETERS RESULTS

The accuracy and F1 score in table IV and table VI are the same. The tuning of the hyperparameters did not increase the accuracy and F1 score of the xgboost model. Also, both BoW and TF-IDF performed the same with the xgboost algorithm.

Confusion Matrix:

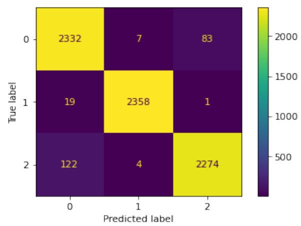

Fig. 13. Xgboost using Tuned Hyperparameters with Bag of Words

V. Conclusion

Comparing all four models, the highest accuracy and f1 score of 97% were obtained from the xgboost model with both BoW and TF-IDF. The second highest accuracy and f1 score of 96% were obtained from the logistic regression with both BoW and TF-IDF. The results are summarized in the table given below.

All Results					
Word Embedding	Metrics	LR	MNB	KNN	XgBoost
BoW	Accuracy	**0.96**	0.88	0.86	**0.97**
	F1 Score	**0.96**	0.87	0.86	**0.97**
TF-IDF	Accuracy	**0.96**	0.86	0.46	**0.97**
	F1 Score	**0.96**	0.85	0.40	**0.97**

TABLE VII
Final Comparison

The accuracy and f1 score is not much different for the xgboost and the logistic regression model. For this case study, accuracy matters more than interpretability, so even though the xgboost model is more complex than the logistic regression model, the xgboost model is preferred.

References

[1] K. Dinakar, R. Reichart, and H. Lieberman, "Modeling the detection of textual cyberbullying," In Proceedings of the Social Mobile Web. Citeseer, 2011.

[2] V. S. Chavan and S. Shylaja, "Machine learning approach for detection of cyber-aggressive comments by peers on social media network," in 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2015, pp. 2354–2358.

[3] M. Dadvar, F. d. Jong, R. Ordelman, and D. Trieschnigg, "Improved cyberbullying detection using gender information," in Proceedings of the Twelfth Dutch-Belgian Information Retrieval Workshop (DIR 2012). University of Ghent, 2012.

[4] J. Wang, K. Fu, C.T. Lu, "SOSNet: A Graph Convolutional Network Approach to Fine-Grained Cyberbullying Detection," Proceedings of the 2020 IEEE International Conference on Big Data (IEEE BigData 2020), pp. 1699-1708, December 10-13, 2020.

Appendix

All the python code for this work is given below:

```
# Importing all the required packages

import pandas as pd
import numpy as np
import re
import matplotlib.pyplot as plt
import nltk
from nltk.stem import WordNetLemmatizer
from sklearn.preprocessing import LabelEncoder
import string
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
import xgboost as xgb
from wordcloud import WordCloud
from sklearn.model_selection import GridSearchCV

nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')
# Loading Data

def openTextasList(filename):
```

```
  with open(filename, encoding="utf8") as file_in:
    lines = []
    for line in file_in:
      # remove whitespace characters like `\n` at the end of each line
      line=line.strip()
      lines.append(line)
  return(lines)

ethnicity=openTextasList("..\DataMiningIFinalProject/8000ethnicity.txt")
religion=openTextasList("..\DataMiningIFinalProject/8000religion.txt")
notcb=openTextasList("..\DataMiningIFinalProject/8000notcb.txt")
alldoc=notcb + ethnicity + religion
print("There are %d ethnicity tweets:\n %s " % (len(ethnicity),ethnicity[0:5]))
print("There are %d religion tweets:\n %s " % (len(religion),religion[0:5]))
print("There are %d notcb tweets:\n %s " % (len(notcb),notcb[0:5]))

alldoc[0:2]

# define training labels
class_label = np.array(["notcb"for _ in range(8000)] +
                       ["ethencity"for _ in range(8000)] +
                       ["religion"for _ in range(8000)])
class_label.shape

# Construct a dataframe
lst = alldoc
# list of int
lst2 = class_label

# zipping both lists with columns specified
df = pd.DataFrame(list(zip(lst, lst2)),
            columns =['Tweets', 'Labels'])
```

```
df

df['Labels'].value_counts()

# Data Cleaning

np.sum(df.isnull())

# storing both Tweets and Labels in lists
tweets, labels = list(df['Tweets']), list(df['Labels'])

# labels
labelencoder = LabelEncoder()
df['LabelsEncoded'] = labelencoder.fit_transform(df['Labels'])

#Changing notcb to 0
df.LabelsEncoded = df.LabelsEncoded.replace([0,1,2], [1,0,2])
df[['Labels', 'LabelsEncoded']].value_counts()

# converting tweets to lower case
df['Tweets'] = df['Tweets'].str.lower()
df.head()

# removing stopwords
def RemoveStopWords(input_text):
    StopWordsList = stopwords.words('english')
    # Words that might indicate some sentiments are assigned to
    # WhiteList and are not removed
    WhiteList = ["n't", "not", "no"]
    words = input_text.split()
    CleanWords = [word for word in words if (word not in StopWordsList or word in WhiteList) and len(word) > 1]
    return " ".join(CleanWords)
```

```
df.Tweets = df["Tweets"].apply(RemoveStopWords)
df.Tweets.head()

# removing URLs
def RemoveURLs(text):
    return re.sub(r"((www.[^s]+)|(http\S+))","",text)

df['Tweets'] = df['Tweets'].apply(lambda x : RemoveURLs(x))
df.Tweets.head()

# removing mentions
def MentionsRemover(input_text):
    return re.sub(r'@\w+', '', input_text)

df.Tweets = df["Tweets"].apply(MentionsRemover)
df.Tweets.head()

# removing numeric data
def RemoveNumeric(text):
    return re.sub('[0-9]+', '', text)

df['Tweets'] = df['Tweets'].apply(lambda x: RemoveNumeric(x))
df.Tweets.head()

# removing punctuations
Punctuations = string.punctuation
print(Punctuations)

def RemovePunctuations(text):
    translator = str.maketrans('','', Punctuations)
    return text.translate(translator)
```

```
df['Tweets'] = df['Tweets'].apply(lambda x : RemovePunctuations(x))
df.Tweets.head()

# removing emojis
def RemoveEmoji(text):
    EmojiPattern = re.compile(pattern = "["
        u"\U0001F600-\U0001F64F"  # emoticons
        u"\U0001F300-\U0001F5FF"  # symbols & pictographs
        u"\U0001F680-\U0001F6FF"  # transport & map symbols
        u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                          "]+", flags = re.UNICODE)
    return EmojiPattern.sub(r'',text)

df['Tweets'] = df['Tweets'].apply(lambda x : RemoveEmoji(x))
df.Tweets.head()

# Tokenization of tweets
df.Tweets = df.Tweets.tolist()

TokenizeText = [word_tokenize(i) for i in df.Tweets]
# for i in TokenizeText:
#     print(i)
df.Tweets = TokenizeText
print(df.Tweets.head())

# Lemmatization
lemmatizer = WordNetLemmatizer()

def Lemmatization(text):
    text = [lemmatizer.lemmatize(word) for word in text]
    return text
```

```
df['Tweets'] = df['Tweets'].apply(lambda x: Lemmatization(x))
print(df['Tweets'].head())

df

# Joining all words with spaces
df['Tweets'] = df['Tweets'].apply(lambda x : " ".join(x))
df

# Word Clouds

# NOT cyberbullying tweets
NotCbDf = df.loc[df['LabelsEncoded'] == 0]
# Converting all tweets into a single list and then to single string
NotCbDfTweets = NotCbDf.Tweets.tolist()
NotCbDfTweets = " ".join(NotCbDfTweets)

#pip install wordcloud

# Word Cloud for NOT cyberbullying tweets
wordcloud = WordCloud(max_words=50).generate(NotCbDfTweets)
plt.figure(figsize=(12, 8))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()

# Etnicity tweets
EthnicityDf = df.loc[df['LabelsEncoded'] == 1]
# Converting all tweets into a single list and then to single string
EthnicityDfTweets = EthnicityDf.Tweets.tolist()
```

```
EthnicityDfTweets = " ".join(EthnicityDfTweets)

# Word Cloud for etnicity tweets
wordcloud = WordCloud(max_words=50).generate(EthnicityDfTweets)
plt.figure(figsize=(12, 8))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()

# Religion tweets
ReligionDf = df.loc[df['LabelsEncoded'] == 2]
# Converting all tweets into a single list and then to single string
ReligionDfTweets = ReligionDf.Tweets.tolist()
ReligionDfTweets = " ".join(ReligionDfTweets)

# Word Cloud for religion tweets
wordcloud = WordCloud(max_words=50).generate(ReligionDfTweets)
plt.figure(figsize=(12, 8))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()

# Splitting the data into train-test

# Splitting the data
X, y = df['Tweets'], df['LabelsEncoded']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.3, random_state= 111)

print(X_train.shape)
print(X_test.shape)

# Bag of Words and TF-IDF
```

```
# Bag of Words
BoW = CountVectorizer(ngram_range= (1,1))
# Train data
BoW_X_train = BoW.fit_transform(X_train)
print(BoW_X_train.toarray())
print(BoW_X_train.toarray().shape)
# Test data
BoW_X_test = BoW.transform(X_test)
print(BoW_X_test.toarray())
print(BoW_X_test.toarray().shape)

#Check
BoW_X_train.toarray()[100][21900:21950]

# TF-IDF
TF_IDF = TfidfVectorizer(ngram_range=(1,1), max_features= 200000)
#Train Data
TF_IDF_X_train = TF_IDF.fit_transform(X_train)
print(TF_IDF_X_train.toarray())
print(TF_IDF_X_train.toarray().shape)
# Test Data
TF_IDF_X_test = TF_IDF.transform(X_test)
print(TF_IDF_X_test.toarray())
print(TF_IDF_X_test.toarray().shape)

#Check
TF_IDF_X_train.toarray()[100][21900:21950]

# Modeling

#  Logistic Regression
```

```
# Function for logistic regression to compare Bag of Words and TF-IDF
def LogisticRegressionFunction(X_train, X_test, y_train, y_test, description):
    LogitClassifier = LogisticRegression(solver='lbfgs', multi_class='multinomial', random_state=111, n_jobs=-1)
    LogitClassifier.fit(X_train, y_train)
    y_prediction = LogitClassifier.predict(X_test)
    ConfMat = confusion_matrix(y_test, y_prediction)
    display = ConfusionMatrixDisplay(confusion_matrix= ConfMat)
    display.plot()
    plt.show()
    Accuracy = metrics.accuracy_score(y_test, y_prediction)
    F1 = metrics.f1_score(y_test, y_prediction, average='weighted')
    print('Accuracy for ', description,' is: {:.2f}'.format(Accuracy))
    print('F1 score for ', description,' is: {:.2f}'.format(F1))

# Logistic regression with Bag of Words
LogisticRegressionFunction(BoW_X_train, BoW_X_test, y_train, y_test, 'Logistic Regression with Bag of Words')

# Logistic regression with TF-IDF
LogisticRegressionFunction(TF_IDF_X_train, TF_IDF_X_test, y_train, y_test, 'Logistic regression with TF-IDF')

# Multinomial Naive Bayes

# Function for multinomial naive bayes to compare Bag of Words and TF-IDF
def MultinomialNaiveBayes(X_train, X_test, y_train, y_test, description):
    MultiNaiveBayesClassifier = MultinomialNB()
    MultiNaiveBayesClassifier.fit(X_train, y_train)
    y_prediction = MultiNaiveBayesClassifier.predict(X_test)
    ConfMat = confusion_matrix(y_test, y_prediction)
    display = ConfusionMatrixDisplay(confusion_matrix= ConfMat)
    display.plot()
    plt.show()
```

```
    Accuracy = metrics.accuracy_score(y_test, y_prediction)

    F1 = metrics.f1_score(y_test, y_prediction, average='weighted')

    print('Accuracy for ', description,' is: {:.2f}'.format(Accuracy))

    print('F1 score for ', description,' is: {:.2f}'.format(F1))


# Multinomial Naive Bayes with Bag of Words

MultinomialNaiveBayes(BoW_X_train, BoW_X_test,y_train, y_test, 'Multinomial Naive Bayes with Bag of Words')


# Multinomial Naive Bayes with TF-IDF

MultinomialNaiveBayes(TF_IDF_X_train, TF_IDF_X_test, y_train, y_test, 'Multinomial Naive Bayes with TF-IDF')


# K-Nearest Neighbor


# First, lets find the best value of k.


# define grid parameters

grid_params = { 'n_neighbors' : list(range(1,26))}

# grid search

GS = GridSearchCV(KNeighborsClassifier(), grid_params, verbose = 1, cv=5, n_jobs = -1)


# fit the model for Bag of Words

GridResult = GS.fit(BoW_X_train, y_train)

# hyperparameters with the best score

GridResult.best_params_


# fit the model for Bag of Words

GridResult = GS.fit(TF_IDF_X_train, y_train)

# hyperparameters with the best score

GridResult.best_params_
```

```
# Function for knn to compare Bag of Words and TF-IDF
def KnnFunction(X_train, X_test, y_train, y_test, n, description):
    KnnClassifier = KNeighborsClassifier(n_neighbors= n)
    KnnClassifier.fit(X_train, y_train)
    y_prediction = KnnClassifier.predict(X_test)
    ConfMat = confusion_matrix(y_test, y_prediction)
    display = ConfusionMatrixDisplay(confusion_matrix= ConfMat)
    display.plot()
    plt.show()
    Accuracy = metrics.accuracy_score(y_test, y_prediction)
    F1 = metrics.f1_score(y_test, y_prediction, average='weighted')
    print('Accuracy for ', description,' is: {:.2f}'.format(Accuracy))
    print('F1 score for ', description,' is: {:.2f}'.format(F1))

# KNN with Bag of Words
KnnFunction(BoW_X_train, BoW_X_test,y_train, y_test, 1, 'KNN with Bag of Words')

# KNN with TF-IDF
KnnFunction(TF_IDF_X_train, TF_IDF_X_test, y_train, y_test, 1, 'KNN with TF-IDF')

# Extreme Gradient Boosting

# Function for xgboost to compare Bag of Words and TF-IDF
def XgBoostFunction(X_train, X_test, y_train, y_test, learning_rate, max_depth, description):
    XgBoostClassifier = xgb.XGBClassifier(objective = 'multi:softmax',
                                          learning_rate = learning_rate, max_depth = max_depth, seed = 111)
    XgBoostClassifier.fit(X_train, y_train)
    y_prediction = XgBoostClassifier.predict(X_test)
    ConfMat = confusion_matrix(y_test, y_prediction)
    display = ConfusionMatrixDisplay(confusion_matrix= ConfMat)
    display.plot()
    plt.show()
```

```
    Accuracy = metrics.accuracy_score(y_test, y_prediction)
    F1 = metrics.f1_score(y_test, y_prediction, average='weighted')
    print('Accuracy for ', description,' is: {:.2f}'.format(Accuracy))
    print('F1 score for ', description,' is: {:.2f}'.format(F1))

# The default value for learning_rate is 0.3 and max_depth is 6.

# Xgboost with Bag of Words (using default parameters)
XgBoostFunction(BoW_X_train, BoW_X_test,y_train, y_test, 0.3, 6, 'Xgboost with Bag of Words')

# Xgboost with TF-IDF (Using default parameters)
XgBoostFunction(TF_IDF_X_train, TF_IDF_X_test, y_train, y_test, 0.3, 6, 'Xgboost with TF-IDF')

# Let's tune a couple of parameters: learning_rate and max_depth.

# define parameters
params = { 'max_depth': [3, 4, 5, 6, 7],
          'learning_rate': [0.01, 0.1, 0.3, 0.5, 0.7] }
# grid search
GrdSrch = GridSearchCV(estimator= xgb.XGBClassifier(objective = 'multi:softmax', seed = 111),
                    param_grid= params,
                    scoring='accuracy',
                    verbose=1)

# fit the model for Bag of Words
GrdSrchResult = GrdSrch.fit(BoW_X_train, y_train)
# hyperparameters with the best score
GrdSrchResult.best_params_

# fit the model for TF-IDF
GrdSrchResult = GrdSrch.fit(TF_IDF_X_train, y_train)
# hyperparameters with the best score
```

```
GrdSrchResult.best_params_


# Xgboost with Bag of Words (using tuned parameters)
XgBoostFunction(BoW_X_train, BoW_X_test,y_train, y_test, 0.5, 4, 'Xgboost with Bag of Words')


# For TF-IDF, we got default values as the best ones.
```