Why So Harsh Project Report

Objective:

Building a model to detect any new toxic comment and classifying them into 6 mutually independent classes of “harsh”, “extremely harsh”, “vulgar”, “threatening”, “disrespect”, “targeted hate”.

Dataset Overview:

Training Dataset: Contains 127656 rows and 8 columns

```
train.head()
```

	id	text	harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate
0	52e0f91a5d7b74552c55	New Main Picture \n\nHow about this for the ma...	0	0	0	0	0	0
1	e2c8e370a8e53ba26bae	Think of them like population charts. Just bec...	0	0	0	0	0	0
2	03c807f61149a13c8404	This page seems a little misleading. The reaso...	0	0	0	0	0	0
3	fc63a1ba3372899db19f	"\n\nActually, accounts are never deleted. "	0	0	0	0	0	0
4	0c2bfd9cde8974d9915f	"\n\nYeah yeah, OK. So did I. Still, what I ...	0	0	0	0	0	0

Test Dataset: Contains 31915 rows and 2 columns.

```
] test.head()
```

	id	text
0	25f48f649f60423c091b	, 19 May 2006 (UTC)\nThey debate, they don't v...
1	5c7ac6d7fb400bbadfc7	"\n\nI am completely nonplussed at this ""We'v...
2	d00a363d57952496854f	"\n\nUnblock request\n\nCategory:User block te...
3	b082c69afa60b378503d	Dave 1185 \n\nIf you have a moment, can you he...
4	1a585118ed7e1f29b38b	WarningPlease stop adding nonsense to Wikipedi...

Then we checked how many comments are clean and how many comments are tagged as toxic in the training set:

```
Total comments =  127656
Total clean comments =  114692
Total tags = 28117
```

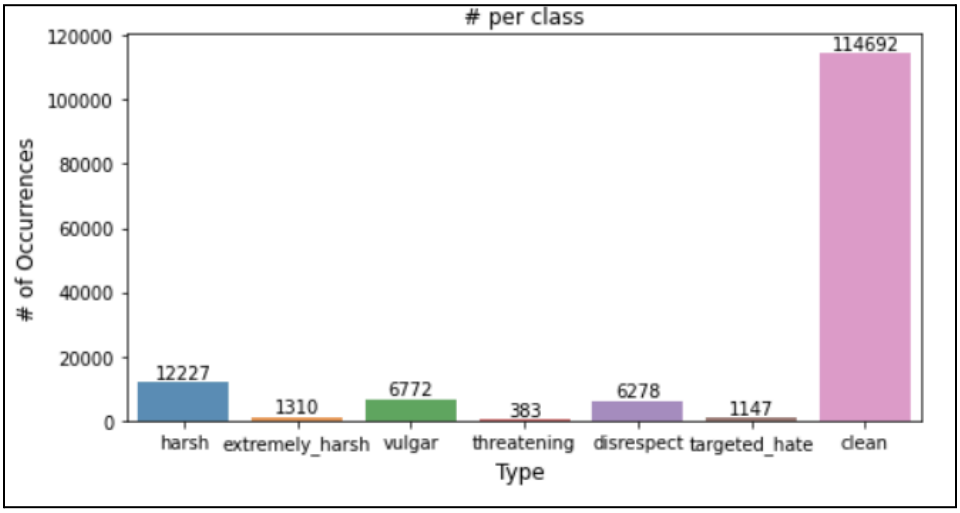
Missing Value Checking:

Checked for any missing values in the training set. Found there were no missing values in the training set

```
id                0
text              0
harsh             0
extremely_harsh   0
vulgar            0
threatening       0
disrespect        0
targeted_hate     0
clean             0
dtype: int64
filling NA with "unknown"
```

Visualization of the distribution of tags:

We wanted to check how many comments are tagged in each category, as in how many comments are tagged as ‘harsh’, how many comments are tagged as ‘vulgar’ etc.
From the chart it is evident most of the tags are not evenly spread out, most of the comments are tagged as ‘harsh’, ‘vulgar’ or ‘disrespect’ whereas threatening is very few. So we could get an idea that **multiple tags** are associated with each comment.

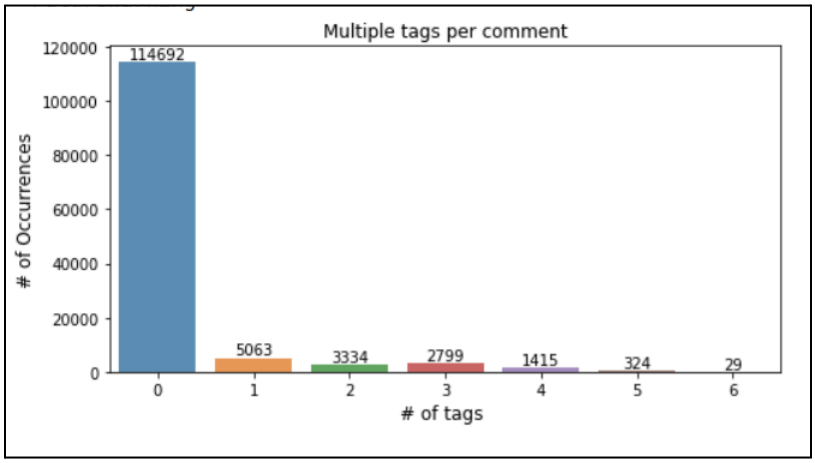


Checking Multi-tagging:

Then we checked how many comments are multi tagged. So there are 1415 comets which are tagged in 4 different categories. Similarly found there are 29 comments which are tagged in all 6 categories

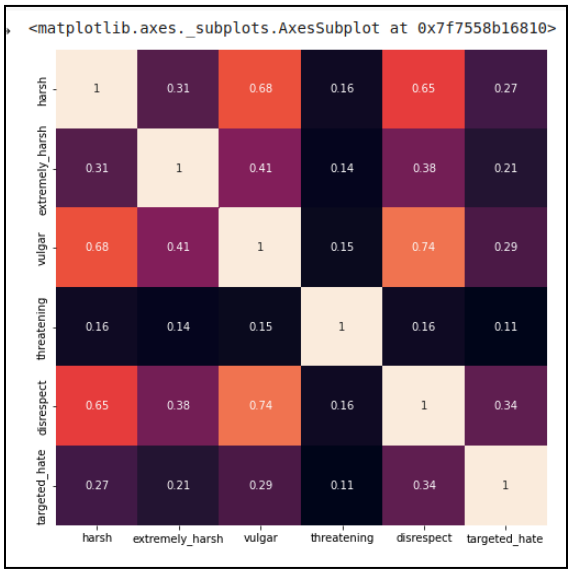
```
x=rowsums.value_counts()
x
```

```
0    114692
1      5063
3      3334
2      2799
4      1415
5       324
6        29
dtype: int64
```



Correlation plot to check which comments go together most often:

Found that ‘harsh’ comments are mostly tagged as ‘vulgar’ and/or ‘targeted hate’ as well, as these are showing good correlation. Also noticed that ‘vulgar’ comments are mostly tagged as ‘disrespect’ too.



But as these comments are ‘categorical’ so it is not a very good idea to try ‘Pearson Correlation’, so we tried to check the correlation with the confusion matrix also.
We did a crosstab analysis of all 6 classes with ‘Harsh Category’.
We can see how ‘harsh’ comments are also tagged in other categories.
We can see only 75 comments are tagged as ‘targteted_hate’ as well as ‘harsh’

	extremely_harsh		vulgar		threatening		disrespect		targeted_hate	
extremely_harsh	0	1	0	1	0	1	0	1	0	1
harsh										
0	115429	0	115010	419	115405	24	115000	429	115354	75
1	10917	1310	5874	6353	11868	359	6378	5849	11155	1072

Some Examples of Comment Types from the ‘vulgar’ class:

```
print("vulgar:")
print(train[train.vulgar==1].iloc[4,1])
```

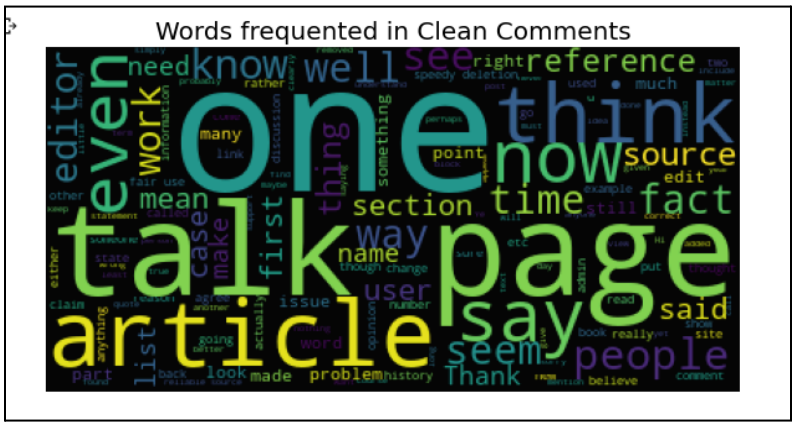
```
vulgar:
i hope they aren't breaking up for good because that would suck. there my favorite band ever!!!
```

Exploratory Data Analysis:

Word Cloud for most frequent words:

For each category of comments made a wordcloud to find which words are occurring most frequently.

Words occurring most frequently in ‘clean’ comments



Word Cloud for ‘Harsh’ comments:

Word cloud for ‘Extremely Harsh’:

Word Cloud for ‘Vulgar’ comments:

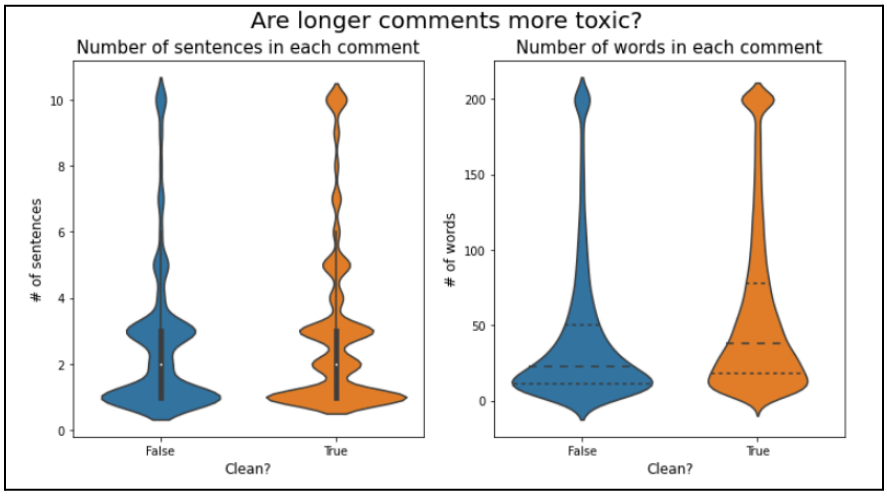
Word cloud for ‘Threatening’:

Word Cloud for ‘Vulgar’ comments:

Word cloud for ‘Threatening’:

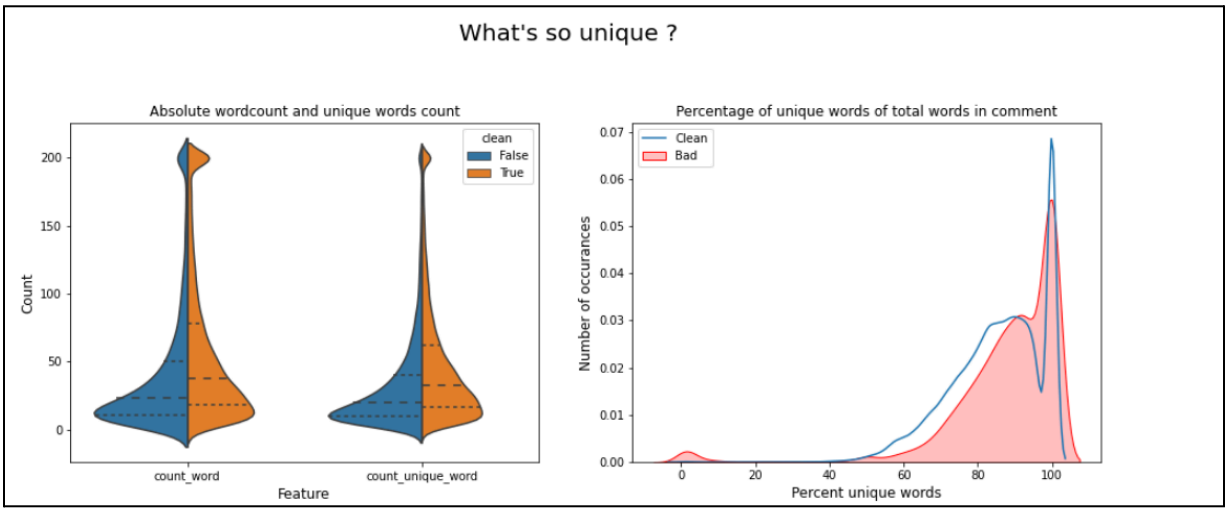
Violin plot for checking if ‘longer comments’ are more toxic:

Found there is no such relation between ‘longer sentences’ and toxicity. So we can’t really tell if longer sentences ought to be toxic comments.



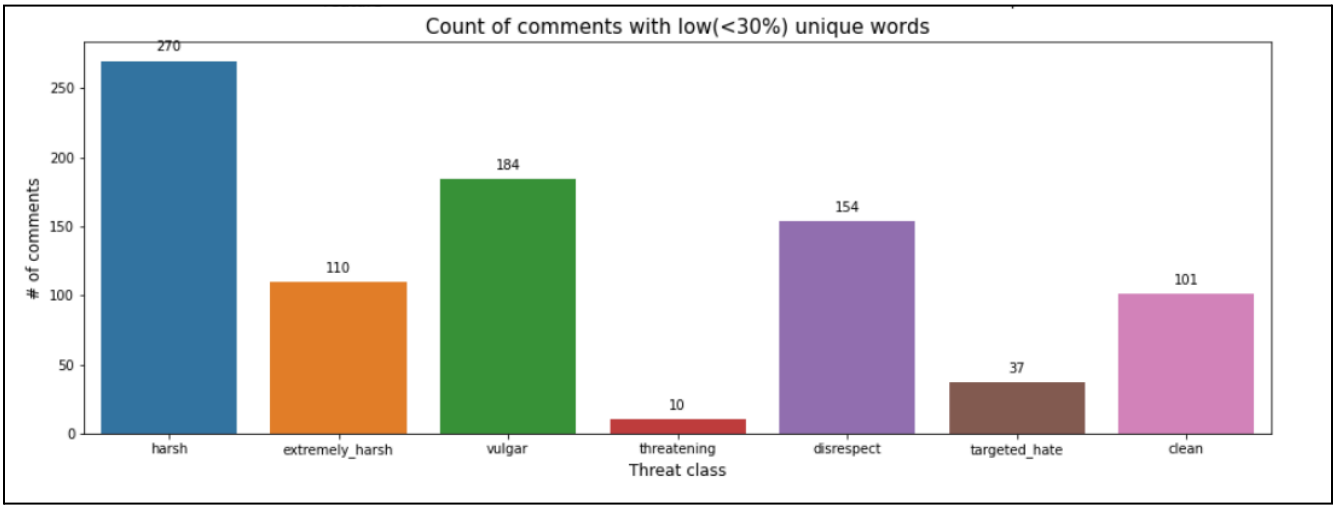
Checking if ‘unique words’ are more toxic or not:

We can see that from the violin plots, more unique words means more chance of being toxic.



Plotted a bar chart to see ‘how many comments’ are having less than 30% unique words in each category.

Found that mostly ‘harsh’ comments have a lot of repetitive words, more than 270 ‘Harsh’ comments contain less than 30% unique words which can be an important criteria for classifying. Similarly, ‘vulgar’ ad ‘disrespect’ categories all contain quite a few comments with less than 30% unique words.



Then checked for ‘spam comments’ where one toxic comment is repeated multiple times

Example of a spam comment from ‘Harsh Category’:

```
 UNBLOCK SHANNON!
Harsh Spam example:
WORST MOVIE EVER

FROZEN SUCKS
FROZEN SUCKSFROZEN SUCKSFROZEN SUCKS
FROZEN SUCKS
FROZEN SUCKSFROZEN SUCKSFROZEN SUCKSFROZEN SUCKS
FROZEN SUCKS
FROZEN SUCKSFROZEN SUCKS
FROZEN SUCKS
FROZEN SUCKSFROZEN SUCKSFROZEN SUCKSFROZEN SUCKSFROZEN SUCKSFROZEN SUCKSFROZEN SUCKSFROZEN SUCKSVVV
V
```

Preprocessing Steps:

Preprocessing steps include

1) **Removing special characters,digits**

Digits,Special characters doesn't add any information to word embeddings as embeddings are related to count of words and the spatial relation of word vectors.

```
text=re.sub(r"([^\w]*[!,:';\.\[\]\^\+\-\=\?][^\w]*)"," ",text)
```

```
text = re.sub(r"[0-9]", "", text)
```

2) **Replacing repeating characters by one character**

Word embedding vector generation will not miss out words because their letters are being repeated.

```
    text = re.sub(r'(.)\1+', r'\1', text)
```

3) Replacing abbreviations with full form

Abbreviations give different forms of the same words,which leads to non uniformity.

```
APPO = {
"aren't" : "are not",
"can't" : "cannot",
"couldn't" : "could not",
```

```
#aphost replacement
words=[APPO[word] if word in APPO else word for word in words]
```

4) Tokenization

Tokenization is in effect splitting the sentence to constituent words.

```
#Split the sentences into words
words=tokenizer.tokenize(comment)
```

5) Removing stop words

Stop Words such as a,about,the,is are not unique to the comment.Removing them helps to reduce the size of data.

```
words = [w for w in words if not w in eng_stopwords]
```

6) Lemmatization

It returns the base or dictionary form of a word, which is known as the lemma,and offers uniformity when dealing with a given word.

```
#lemmatize
words=[lem.lemmatize(word, "v") for word in words]
```

Feature Engineering avoids these preprocessing steps as no exclamation marks at end of comment can make the difference between harsh and extremely harsh.

Feature Engineering Methods

Word2Vec

Word2Vec is generated using two models :

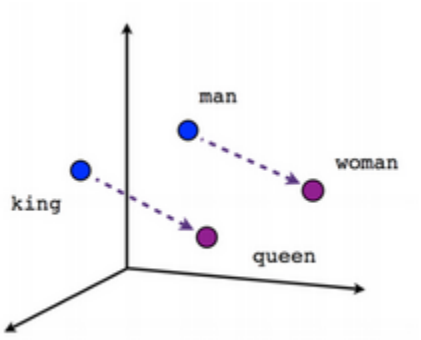
- **Skip gram**

  The model is given the fake task of predicting its neighboring words.Goal here is to learn the weights of the model,inputs and outputs of the model is not taken into consideration.

- **CBOW(Common Bag of Words)**

  Here the model predicts the word in the middle given the neighboring words.

  Word2Vec vectors of similar words come closer to each other in n-dimensional space



Sentence Vectorizer

Input is each sentence from the dataset after pre processing and cleaning.

Each word from the sentence is then taken and we find the word2vec vector of that word and append the vector values to a list.

After all words in the sentence are considered, we divide the final list of values by the total number of words in the sentence.

```
def sent_vectorizer(sent,model):
  sent_vec=[]
  num=0
  for w in sent:
    try:
      if num==0:
        sent_vec=model[w]
      else:
        sent_vec=np.add(sent_vec,model[w])
      num += 1
    except:
      pass
  #print(np.asarray(sent_vec)/num)
  l=np.asarray(sent_vec)/num
  #print(l)
  return l
```

CountVectorizer

CountVectorizer creates a matrix in which each unique word is represented by a column of the matrix, and each text sample from the document is a row in the matrix.Inside CountVectorizer, these words are not stored as strings. Rather, they are given a particular index value. In this case, ‘at’ would have index 0, ‘each’ would have index 1, ‘four’ would have index 2 and so on.

	hello	is	james	my	name	notebook	python	this
0	1	1	1	1	1	0	0	0
1	0	1	0	1	0	1	1	1

TF-IDF (Term Frequency Inverse Document Frequency):

This technique is used to quantify words in a set of documents where we compute the score for each word to signify the importance of that word in the document and the corpus. ML algorithms can’t work with textual data so we vectorize these textual data into numbers.

Stop Words: Stop words were removed from the document while calculating TF-IDF score. Stop words are the words that occur very frequently in the text but they carry very little information. Examples of stop words in English are “a”, “the”, “is”, “are” and etc.

Term Frequency: Measures the frequency of the word in the document.
If the term doesn’t exist in a particular document, that particular TF value will be 0 for that particular document. In an extreme case, if all the words in the document are the same, then TF will be 1. The final value of the normalized TF value will be in the range of [0 to 1]

$$\text{TF(word, text)} = \frac{\text{number of times the word occurs in the text}}{\text{number of words in the text}}$$

$$\text{IDF(word)} = \log\left[\frac{\text{number of texts}}{\text{number of texts where the word occurs}}\right]$$

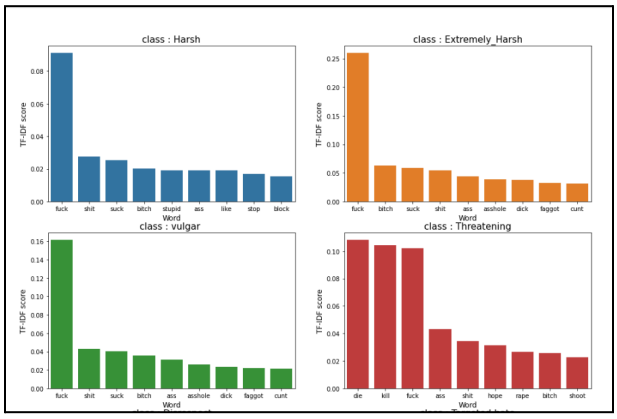
$$\text{TF-IDF(word, text)} = \text{TF(word, text)} \times \text{IDF(word)}$$

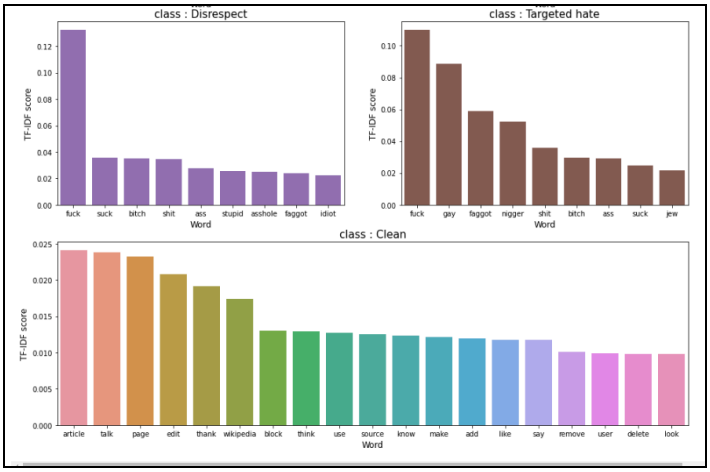
Document Frequency: DF is the count of **occurrences** of term t in the document set N. In other words, DF is the number of documents in which the word is present.

Inverse Document Frequency: IDF is the inverse of the document frequency which measures the informativeness of term t. It penalizes words/terms that are too frequent in the text such as stop words.
Then finally we find the TF-IDF value for each word by multiplying TF and IDF values.

Unigrams : Used TF-IDF vectorizer to find the most occurring unigram and plotted the occurrence of words

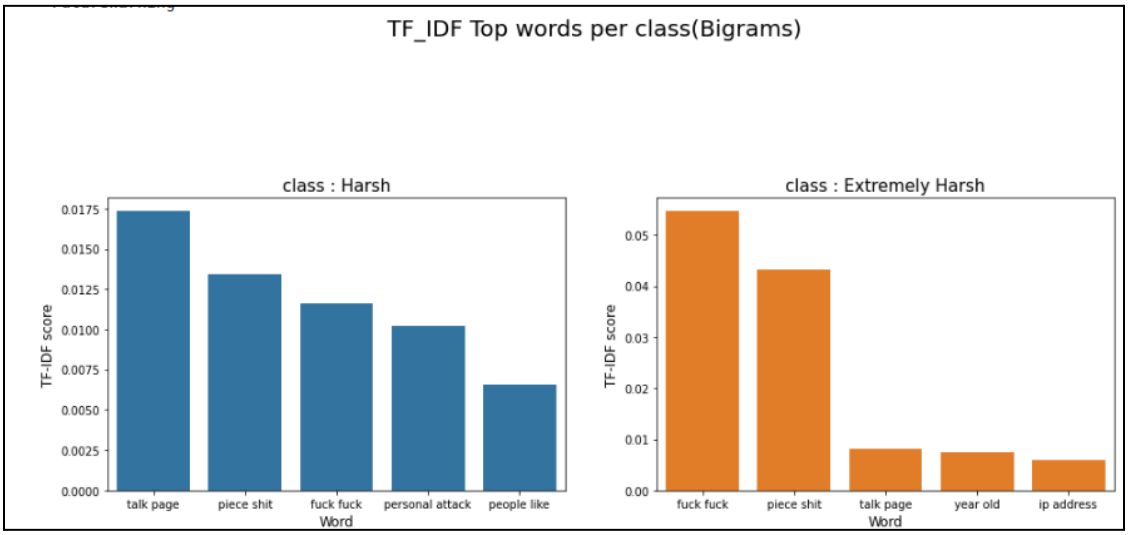
Plotted the top words per class (unigrams)

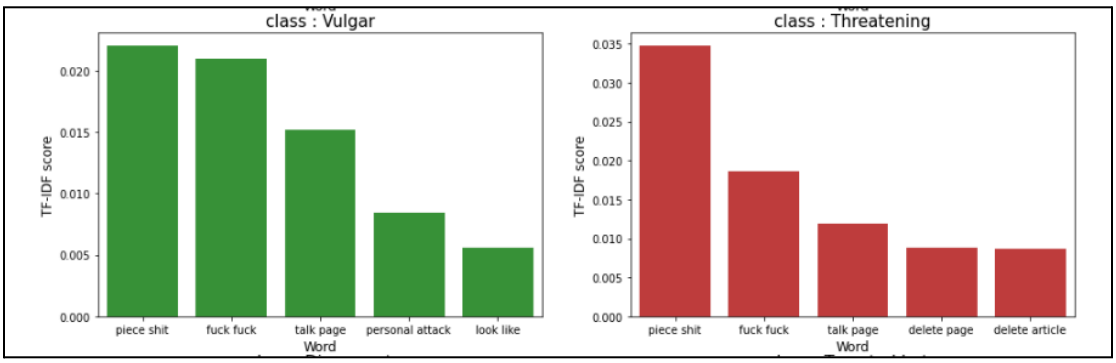


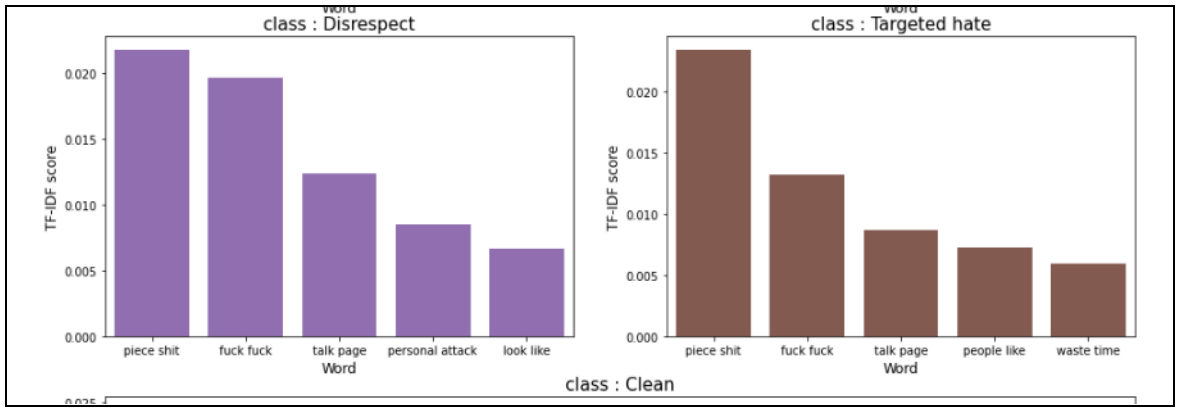


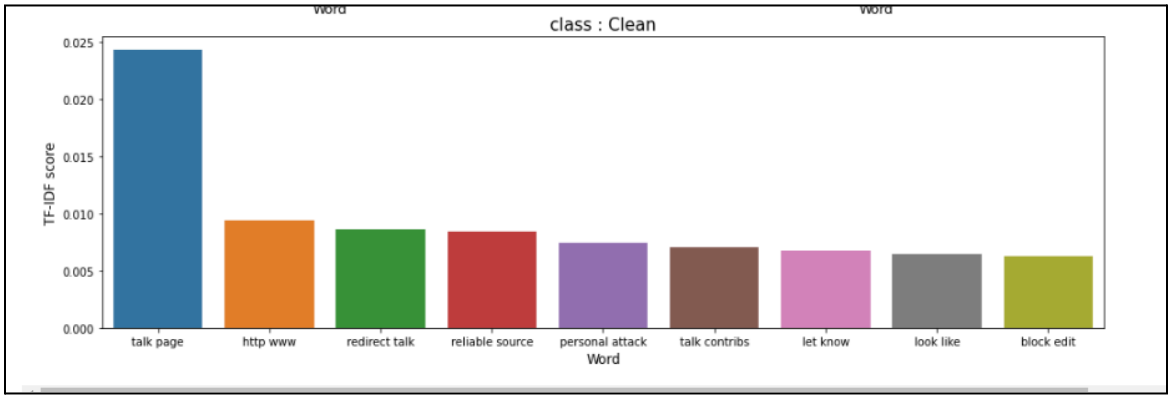
TF-IDF top words per class (bigrams)

Bigrams are two consecutive words in a sentence whereas unigrams are single words. Found which two words are occuring most frequently and plotted the frequency in each class.









Model building using Direct Features:

Checked the presence of null values in test data. Found there are some null values in test data. Filled the NA values with empty space.

	train	test
id	0	0.0
text	0	0.0
harsh	0	NaN
extremely_harsh	0	NaN
vulgar	0	NaN
threatening	0	NaN
disrespect	0	NaN
targeted_hate	0	NaN

```
test.fillna(' ',inplace=True)
```

TF-IDF Vectorization for word and characters:

Used the TfidfVectorizer function from sklearn and created a word vector of unigrams, bigrams and trigrams. Took into account 20000 max features, and made lowercase= True, so that there won't be any discrepancy between upper and lower case letters so this will convert all the words into lowercase. Make stop_words = 'english' so that very common stop words that are very frequent in the english language will be removed while calculating the TF-IDF score. Ngram_range is chosen as (1,3) so it will take into account all n grams in between 1 and 3. Analyzer = 'word' so that we are creating only word vectors for now.

Then if we put analyzer = 'char' then we will create character n gram vectors where also we chose the max_features as 40000 and ngram_range = (3,6) so we took tri gram to 6-gram character vectors.

```
#TF-IDF vectorization for word and characters

vect_word = TfidfVectorizer(max_features=20000, lowercase=True, analyzer='word',
                        stop_words= 'english',ngram_range=(1,3),dtype=np.float32)
vect_char = TfidfVectorizer(max_features=40000, lowercase=True, analyzer='char',
                        stop_words= 'english',ngram_range=(3,6),dtype=np.float32)

# Word ngram vector
tr_vect = vect_word.fit_transform(train['text'])
ts_vect = vect_word.transform(test['text'])

# Character n gram vector
tr_vect_char = vect_char.fit_transform(train['text'])
ts_vect_char = vect_char.transform(test['text'])
```

Then we used fit_transform to fit and transform the training text data according to the word and character n gram vector. Then we used these vectors to transform the test text data.

```
#stacking the training and test data

X = sparse.hstack([tr_vect, tr_vect_char])
x_test = sparse.hstack([ts_vect, ts_vect_char])

target_col = ['harsh', 'extremely_harsh', 'vulgar', 'threatening', 'disrespect', 'targeted_hate']
y = train[target_col]
del tr_vect, ts_vect, tr_vect_char, ts_vect_char
```

Stacked the word vector and character vectors of the training and test data. As these are mostly sparse matrices, so use the sparse.hstack() function.

Model Building :

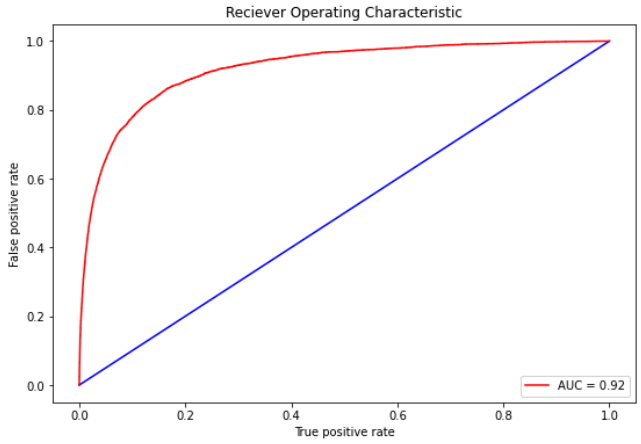
1) Logistic Regression Model

Logistic regression is a supervised learning classification algorithm **used to predict the probability of a target variable**. It gives a value between 0 and 1 and it uses the class_weight = "balanced" argument as we noticed our data was a bit imbalanced. We trained the logistic regression model on the training data and then used the predict_proba() function to generate probability for each class of the test dataset.

```
## logistic regression model

prd_1 = np.zeros((x_test.shape[0],y.shape[1]))

for i,col in enumerate(target_col):
    lr = LogisticRegression(C=2,random_state = i,class_weight = 'balanced')
    print('Building {} model for column:{''}'.format(i,col))
    lr.fit(X,y[col])

    prd_1[:,i] = lr.predict_proba(x_test)[:,1]
```

From the ROC-AUC curve found the area under the curve to be **0.92 for the training set** and **after submission on the test set, the accuracy was 0.98327**

```
Column: disrespect
Text(0.5, 1.0, 'Reciever Operating Characteristic')
```



Confusion matrix for the column 'disrespect'

```
Column: disrespect

Confusion matrix
 [[120622    756]
 [  4584   1694]]
              precision    recall  f1-score   support

           0       0.96      0.99      0.98    121378
           1       0.69      0.27      0.39      6278

    accuracy                           0.96    127656
   macro avg       0.83      0.63      0.68    127656
weighted avg       0.95      0.96      0.95    127656
```

The probabilities for each class after using logistic regression model on the test set :

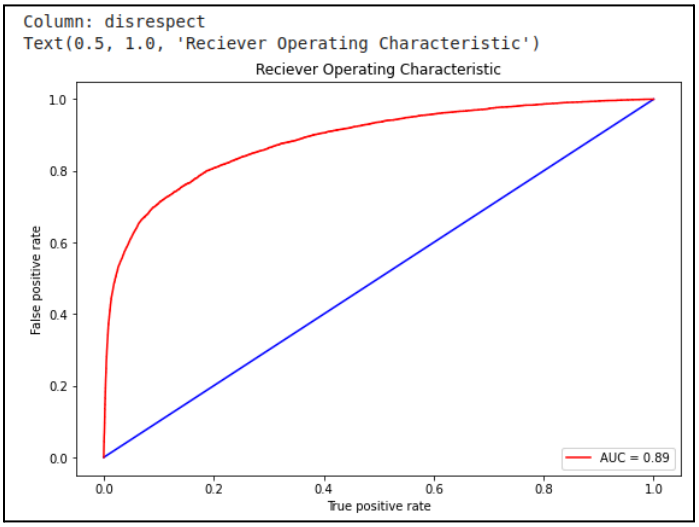
	id	harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate
0	25f48f649f60423c091b	0.013100	0.002681	0.010406	0.000672	0.008795	0.006095
1	5c7ac6d7fb400bbadfc7	0.000539	0.000332	0.000540	0.000038	0.000931	0.000579
2	d00a363d57952496854f	0.007102	0.017337	0.008603	0.000738	0.005941	0.000280
3	b082c69afa60b378503d	0.019841	0.000397	0.019797	0.000252	0.002633	0.000369
4	1a585118ed7e1f29b38b	0.014182	0.000306	0.023821	0.000097	0.018991	0.006193

2) Multinomial Naive Bayes Model

It is very suitable for classification with discrete features such as word counts in text classification. After the TF-IDF vectorization process, the Multinomial Naive Bayes model was applied.

```
[6] from sklearn.naive_bayes import MultinomialNB

    ## prediction with Naive Bayes

    prd_nv = np.zeros((x_test.shape[0],y.shape[1]))

    for i,col in enumerate(target_col):
        naive = MultinomialNB()
        print('Building {} model for column:{''}'.format(i,col))
        naive.fit(X,y[col])

        prd_nv[:,i] = naive.predict_proba(x_test)[:,1]
```

From the ROC-AUC curve, AUC = 0.89. After submission, accuracy on the test set was 0.94058



Confusion matrix for the class 'disrespect' in case of multinomial naive bayes model:

```
Column: disrespect

Confusion matrix
 [[120837    541]
 [  4688   1590]]
              precision    recall  f1-score   support

           0       0.96      1.00      0.98    121378
           1       0.75      0.25      0.38      6278

    accuracy                           0.96    127656
   macro avg       0.85      0.62      0.68    127656
weighted avg       0.95      0.96      0.95    127656
```

Prediction for each class with Multinomial naive bayes model:

```
prd_2 = pd.DataFrame(prd_nv,columns=y.columns)
submit = pd.concat([test['id'],prd_2],axis=1)

submit.to_csv('harsh_comment_naive.csv',index=False)
submit.head()
```

	id	harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate
0	25f48f649f60423c091b	5.077869e-04	1.721751e-06	1.107164e-04	4.003773e-07	5.697876e-05	6.278325e-06
1	5c7ac6d7fb400bbadfc7	1.838275e-08	6.951518e-12	1.338497e-09	1.037683e-12	1.075796e-09	1.113559e-10
2	d00a363d57952496854f	9.253458e-05	6.683507e-06	7.025056e-05	6.991307e-07	2.818264e-05	2.299552e-06
3	b082c69afa60b378503d	1.823333e-05	4.850705e-12	7.295536e-07	1.033562e-13	2.265437e-07	1.498287e-11
4	1a585118ed7e1f29b38b	3.033608e-07	5.074235e-11	9.438233e-08	3.090859e-12	5.954327e-08	2.484393e-10

3) SGD Classifier Model:

Stochastic Gradient Descent Classifier is a very effective approach in case of large scale data and sparse ML problems which are often encountered in NLP and text classification. Used SGD Classifier with different loss functions such as 'hinge', 'squared hinge', 'perceptron'

- SGD classifier trained with 'hinge loss' is equivalent to linear SVM
- As penalty term l2 was selected which is the standard regularizer term in linear SVM models
- Used CalibratedClassifierCV on the SGD model to get the probability for each class

The SGDclassifier with 'squared hinge' loss was performing best as compared to other loss functions , showing an accuracy of 0.98366 on the test set.

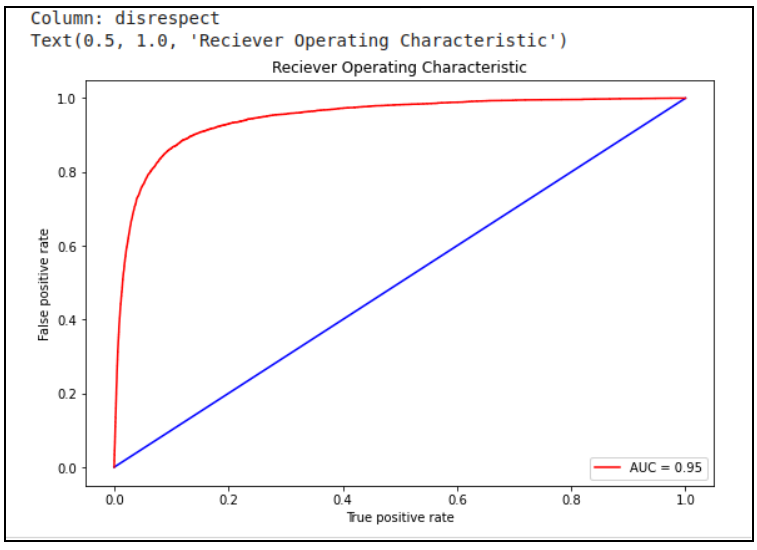
a) SGD Classifier with squared hinge loss function:

It implements the regularized linear models with stochastic gradient descent instead of gradient descent which make the process computationally faster. The gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The initial learning rate is chosen as 0.001 which is defined as eta0. Alpha is the constant that multiplies the regularization term. Max_iter was chosen as 1000; it defines the maximum number of passes over the training data.

```
sgdc = SGDClassifier(loss="squared_hinge", eta0=0.001,penalty='l2',alpha=1e-3,max_iter=1000, random_state=42)   #initially did wih 0.001
model_1 = CalibratedClassifierCV(sgdc)    #calibrated SVM classifier

#prediction
prd_sgdc = np.zeros((x_test.shape[0],y.shape[1]))

for i,col in enumerate(target_col):

    print('Building {} model for column:{''}'.format(i,col))
    model_1.fit(X,y[col])

    prd_sgdc[:,i] = model_1.predict_proba(x_test)[:,1]
```

This model performed best with an ROC-AUC score of 0.95 on training set and after submission on the test data, the accuracy was 0.98366



Confusion matrix for the class 'disrespect':

```
Column: disrespect

Confusion matrix
 [[121289     89]
 [  5961    317]]
              precision    recall  f1-score   support

           0       0.95      1.00      0.98    121378
           1       0.78      0.05      0.09      6278

    accuracy                           0.95    127656
   macro avg       0.87      0.52      0.54    127656
weighted avg       0.94      0.95      0.93    127656
```

Probabilities for each class on the test set after applying 'SGDClassifier' with 'Squared Hinge Loss'

	id	harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate
0	25f48f649f60423c091b	0.010885	0.001949	0.005373	0.000659	0.008780	0.003222
1	5c7ac6d7fb400bbadfc7	0.000643	0.002019	0.001434	0.000309	0.002632	0.001504
2	d00a363d57952496854f	0.006001	0.002800	0.006544	0.000993	0.008468	0.002576
3	b082c69afa60b378503d	0.002987	0.002335	0.003695	0.000471	0.004040	0.001856
4	1a585118ed7e1f29b38b	0.009449	0.002084	0.005060	0.000388	0.008069	0.002845

b) SGD Classifier with hinge loss function:

This is actually an SVM model only with stochastic gradient loss which makes the process of model building faster for each class.

```
]
#using SGDClassifier with Hinge loss, using CalibratedClassifierCV function to get the probabilities for each class

sgdc = SGDClassifier(loss="hinge", eta0=0.001,penalty='l2',alpha=1e-3,max_iter=1000, random_state=42)   #initially did wih 0.001
model = CalibratedClassifierCV(sgdc)    #calibrated SVM classifier
```

```
#prediction using SGDC classifier: with hinge loss

prd_sgdc = np.zeros((x_test.shape[0],y.shape[1]))

for i,col in enumerate(target_col):

    print('Building {} model for column:{''}'.format(i,col))
    model.fit(X,y[col])

    prd_sgdc[:,i] = model.predict_proba(x_test)[:,1]
```

Confusion Matrix for each class (applied SGDClassifier model)

```
Column: harsh

Confusion matrix
 [[115402     27]
 [ 11848    379]]
              precision    recall  f1-score   support

           0       0.91      1.00      0.95    115429
           1       0.93      0.03      0.06     12227

    accuracy                           0.91    127656
```

```
Column: extremely_harsh

Confusion matrix
 [[126045    301]
 [  1205    105]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99    126346
           1       0.26      0.08      0.12      1310

    accuracy                           0.99    127656
```

```
Column: vulgar

Confusion matrix
 [[120772    112]
 [  6478    294]]
              precision    recall  f1-score   support

           0       0.95      1.00      0.97    120884
           1       0.72      0.04      0.08      6772

    accuracy                           0.95    127656
   macro avg       0.84      0.52      0.53    127656
```

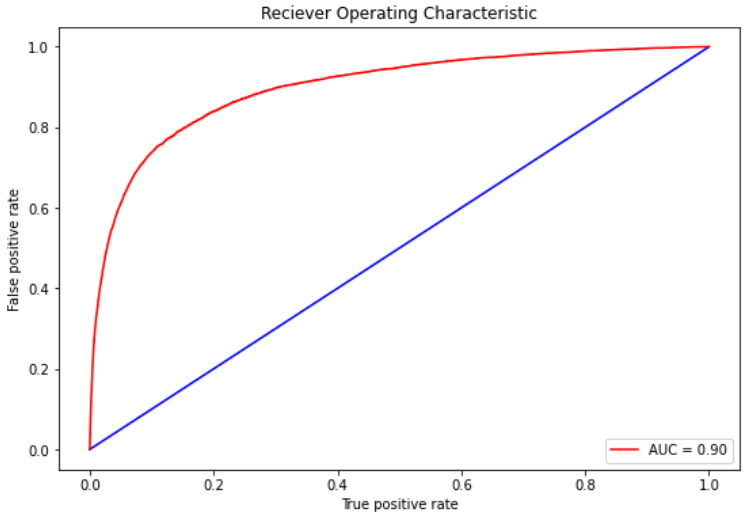
```
Column: threatening

Confusion matrix
 [[126886    387]
 [   364     19]]
              precision    recall  f1-score   suppor

           0       1.00      1.00      1.00    1272
           1       0.05      0.05      0.05      3

    accuracy                           0.99    1276
   macro avg       0.52      0.52      0.52    1276
weighted avg       0.99      0.99      0.99    1276
```

```
Column: disrespect

Confusion matrix
 [[121289     89]
 [  5961    317]]
              precision    recall  f1-score   support

           0       0.95      1.00      0.98    121378
           1       0.78      0.05      0.09      6278

    accuracy                           0.95    127656
   macro avg       0.87      0.52      0.54    127656
weighted avg       0.94      0.95      0.93    127656
```

```
olumn: targeted_hate

onfusion matrix
[[126426     83]
[   824    323]]
              precision    recall  f1-score   support

           0       0.99      1.00      1.00    126509
           1       0.80      0.28      0.42      1147

    accuracy                           0.99    127656
```

From the ROC-AUC curve the value was 0.90, after submitting the accuracy in the test set was 0.97958

```
Column: disrespect
Text(0.5, 1.0, 'Reciever Operating Characteristic')
```



Probabilities for each class using this model:

	id	harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate
0	25f48f649f60423c091b	0.035748	0.001531	0.014419	0.000257	0.020094	0.008438
1	5c7ac6d7fb400bbadfc7	0.011412	0.003191	0.009024	0.000154	0.012136	0.001576
2	d00a363d57952496854f	0.034150	0.001266	0.014423	0.000909	0.020045	0.002414
3	b082c69afa60b378503d	0.018738	0.002370	0.012241	0.000506	0.013085	0.003207
4	1a585118ed7e1f29b38b	0.033144	0.001263	0.015677	0.000195	0.018294	0.007864

Results and Discussion:

We found that **Logistic Regression model along with TF-IDF** performed best according to the **private scoreboard in Kaggle , having a score of 0.98276**

Then **TF-IDF vectorised SGDClassifier along with 'Squared Hinge Loss'** performed better with a **Kaggle score of 0.98218** in the private scoreboard.

Third best was **Countvectorizer with logistic Regression prediction probabilities with features 'count of hash','count of threat words','count of harsh words','count of targeted hate words','capital word count'** gave **Kaggle score of 0.96024**