AI 511 / Machine Learning Project Report

Title

Why So Harsh ?

Team Name :- **Mavericks**

Members:-
Gopal Gupta (MT2022046)
Harsh Tripathi (MT2022048)

ABOUT THE CHALLENGE

Given a comment, classify it into “harsh”, “extremely harsh”, “vulgar”, “threatening”, “disrespect”, “targeted hate”. For example, consider a comment, “You useless piece of trash! you are such an asshole that you deserve to rot in the gutter alongside sewage.” That is certainly harsh! So, the label for this would be [1 1 0 0 1 1]. Translating to the comment being harsh, extremely harsh, disrespectful, and targeted hate.

DATASET DESCRIPTION

Context:

The dataset contains the text associated with a particular comment and the corresponding labels for each of the classes. The classes are:

- harsh
- extremely_harsh
- vulgar
- threatening
- disrespect
- targeted_hate

Files:

- train.csv - training set
- test.csv - test set
- sample_submission.csv – a sample submission file in the correct format

Columns:

Following are the columns present in the dataset,

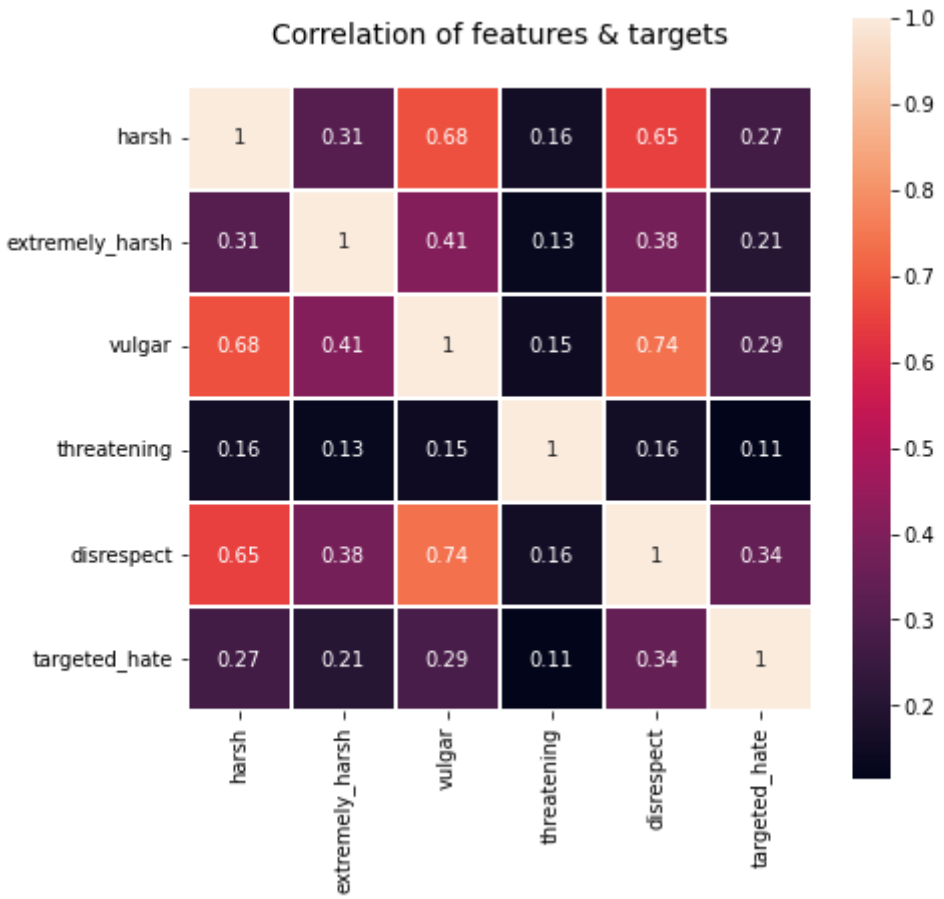
Id	text	harsh	extremely_harsh
vulgar	threating	disrespect	targeted_hate

ANALYSIS OF DATASET

1. Checking percentage of comments that are classified as harsh

	harsh	extremely _harsh	vulgar	threating	disrespect	targeted _hate
0	0.904218	0.989738	0.946933	0.997001	0.95085	0.991025
1	0.095782	0.010262	0.053067	0.002999	0.04915	0.008975

2. Plotting the heatmap



3. Checking the missing values

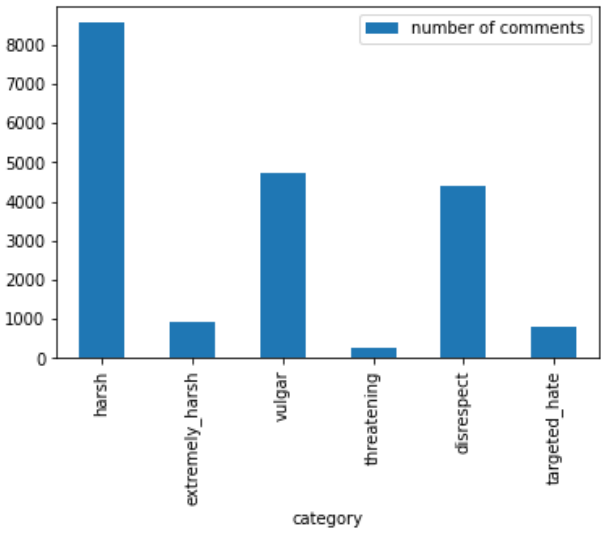
```
id                0
text              0
harsh             0
extremely_harsh   0
vulgar            0
threatening       0
disrespect        0
targeted_hate     0
dtype: int64
```

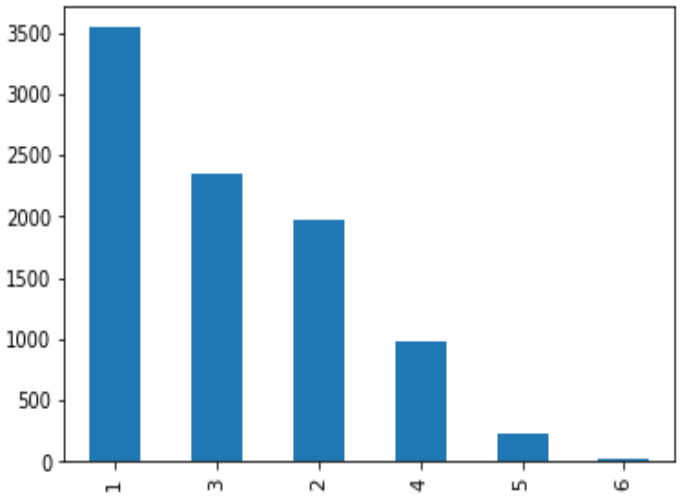
4. Getting count for each labels

	category	number of comments
0	harsh	8559
1	extremely_harsh	917
2	vulgar	4742
3	threatening	268
4	disrespect	4392
5	targeted_hate	802

5. Bar Plots

- 1st plots number of comments and 2nd shows lables per comment.





GETTING FEATURES (PREPROCESSING OF TEXT)

All pre-processing was done for both the train and test dataset

1. Lemmatization of text

- The NLTK python library contains predefined functions for performing lemmatization or stemming on any text data.
- Lemmatization is the process of grouping together the different inflected forms of a word so they can be analyzed as a single item
- Lemmatization is one of the most common text pre-processing techniques used in Natural Language Processing (NLP) and machine learning in general. If you've already read my post about stemming of words in NLP, you'll already know that lemmatization is not that much different. Both in stemming and in lemmatization, we try to reduce a given word to its root word. The root word is called a stem in the stemming process, and it is called a lemma in the lemmatization process. But there are a few more differences to the two than that. Let's see what those are.

```
lemmatizer = WordNetLemmatizer()
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
stopWords = stopwords.words('english')
extraword = [word for word in collection.keys()]
def clean_text(txt):
    txt= word_tokenize(txt) #tokenize (split into list and remove whitespace)
    clean_text=""
    #txt=[collection[word] if word in collection else word for word in txt]
    for w in txt:
        if w[:4] == 'http' or w[:3] == 'www': # If we encounter some links in the comments section,
            continue
        if w in extraword:
            w = collection[w]
        if w not in stop_words:
            stem=lemmatizer.lemmatize(w) #stem
            clean_text += stem +" "
    clean_text = re.sub('[^a-zA-Z ?!]+', '', clean_text)
    return clean_text
```

How is Lemmatization different from Stemming

In stemming, a part of the word is just chopped off at the tail end to arrive at the stem of the word. There are definitely different algorithms used to find out how many characters

have to be chopped off, but the algorithms don't actually know the meaning of the word in the language it belongs to.

2. Manual Text Clean-up

We need to replace words which is not define clearly as they may constituent noise in the data for that we need to manually replace them

```
"aren't" : "are not",
"can't" : "cannot",
"couldn't" : "could not",
"didn't" : "did not",
"doesn't" : "does not",
"don't" : "do not",
"hadn't" : "had not",
"hasn't" : "has not",
"haven't" : "have not",
"he'd" : "he would",
"he'll" : "he will",
"he's" : "he is",
"i'd" : "I would",
"i'd" : "I had",
"i'll" : "I will",
"i'm" : "I am",
"isn't" : "is not",
"it's" : "it is",
"it'll":"it will",
"i've" : "I have",
"let's" : "let us",
"mightn't" : "might not",
"mustn't" : "must not",
"shan't" : "shall not",
"she'd" : "she would",
"she'll" : "she will",
"she's" : "she is",
"shouldn't" : "should not",
"that's" : "that is",
"there's" : "there is",
"they'd" : "they would",
"they'll" : "they will",
"they're" : "they are",
```

```
"they've" : "they have",
"we'd" : "we would",
"we're" : "we are",
"weren't" : "were not",
"we've" : "we have",
"what'll" : "what will",
"what're" : "what are",
"what's" : "what is",
"what've" : "what have",
"where's" : "where is",
"who'd" : "who would",
"who'll" : "who will",
"who're" : "who are",
"who's" : "who is",
"who've" : "who have",
"won't" : "will not",
"wouldn't" : "would not",
"you'd" : "you would",
"you'll" : "you will",
"you're" : "you are",
"you've" : "you have",
"'re": " are",
"wasn't": "was not",
"we'll":" will",
"didn't": "did not",
"tryin'":"trying",
":')": " sad ",
":-(": " sad ",
":(": " sad ",
":s": " sad ",
":-s": " sad ",
":-(": " frown ",
":(": " frown ",
```

```
":s": " frown ",
":-s": " frown ",
":/": " bad ",
":&gt;": " sad ",
":')": " sad ",
"&lt;3": " heart ",
":/": " worry ",
":&gt;": " angry ",
"yay!": " good ",
"yay": " good ",
"yaay": " good ",
"yaaay": " good ",
"yaaaay": " good ",
"yaaaaay": " good ",
"m": "am",
"r": "are",
"u": "you",
"haha": "ha",
"hahaha": "ha",
"don't": "do not",
"haven't": "have not",
"hadn't": "had not",
"won't": "will not",
"wouldn't": "would not",
"can't": "can not",
"cannot": "can not",
"i'm": "i am",
"m": "am",
"i'll" : "i will",
"its" : "it is",
"it's" : "it is",
"'s" : " is",
"that's" : "that is",
"weren't" : "were not",
```

VECTORIZATION

- Vectorization is jargon for a classic approach of converting input data from its raw format (i.e. text) into vectors of real numbers which is the format that ML models support. This approach has been there ever since computers were first built, it has worked wonderfully across various domains, and it's now used in NLP.
- In Machine Learning, vectorization is a step in feature extraction. The idea is to get some distinct features out of the text for the model to train on, by converting text to numerical vectors.

1. TF-IDF

TF-IDF is an abbreviation for Term Frequency Inverse Document Frequency. This is very common algorithm to transform text into a meaningful representation of numbers which is used to fit machine algorithm for prediction.

Code for the following:

```
word_vectorizer = TfidfVectorizer(
    sublinear_tf = True,
    strip_accents = 'unicode',
    analyzer = 'word',
    #token_pattern = '(?u)\\b\\w\\w+\\b\\w{,1}',
    lowercase = False,
    stop_words = 'english',
    ngram_range = (1, 1),
    min_df = 5,
    max_df = 0.25,
    norm = 'l2',
    max_features = 30000
) #lowercase = true : convert all characters into lower case before tokenzing
word_vectorizer.fit(complete_text) # Apply tfidf fitting on the whole preprocessed text data so th
trwfea = word_vectorizer.transform(train_text)
tewfea = word_vectorizer.transform(test_text)

char_vectorizer = TfidfVectorizer (
    sublinear_tf = True,
    strip_accents = 'unicode',
    analyzer = 'char',
    ngram_range = (2, 4),
    min_df = 2,
    max_df = 0.5,
    max_features = 20000
)
char_vectorizer.fit(complete_text) # We fit on complete training + test data so as to achieve a be
trcfea = char_vectorizer.transform(train_text)
tecfea = char_vectorizer.transform(test_text)
```

we vectorize both word and char in out dataset for better results.

Grid search for best hyper parameters:

```
for x in categories:
    print("processing",x)
    pipeline = Pipeline([
        ('tfidf', TfidfVectorizer(stop_words=stop_words)),
        ('clf', RidgeClassifier(alpha=25,fit_intercept=True, solver='sag', max_iter=200,random_state=0)),
    ])
    parameters = {
        'tfidf__min_df' :(2,5),
        'tfidf__max_df': (0.25, 0.5, 0.75),
        'tfidf__ngram_range': [(1, 1), (1, 2), (1, 3)],
    }

    grid_search_tune = GridSearchCV(pipeline, parameters, cv=2, n_jobs=2, verbose=3)
    grid_search_tune.fit(X_train, y_train[x])

    print("Best parameters set:")
    print (grid_search_tune.best_estimator_.steps)
    print("Best: %f using %s" % (grid_search_tune.best_score_, grid_search_tune.best_params_))
```

Parameters used :

max_features	If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.
strip_accents	Remove accents and perform other character normalization during the preprocessing step. Unicode works on all.
analyzer	Whether the feature should be made of word or character n-grams.
ngram_range	he lower and upper boundary of the range of n-values for different n-grams to be extracted.eg (1,1) is unigram while (1,2) is bigram.
min_df	When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold.
max_df	When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words)

2. Using Word2Vec

- Word2Vec is a method to construct such an embedding. It can be obtained using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW)
- ***CBOW Model:*** This method takes the context of each word as the input and tries to predict the word corresponding to the context.

CODE:

Following are the code for word2vec:

```
num_features = 300    # Word vector dimensionality
min_word_count = 40   # Minimum word count
num_workers = 4       # Number of threads to run in parallel
context = 10          # Context window size
downsampling = 1e-3   # Downsample setting for frequent words
# Initialize and train the model (this will take some time)
w2v_model = Word2Vec(pd.concat([X_train_w, X_test_w]), workers=num_workers,
                     vector_size=num_features, min_count = min_word_count, window = context, sample = downsampling)
```

+ Code + Markdown

```
w2v_model.wv.most_similar('nigga')
```

```
[('twat', 0.814550518989563),
 ('cum', 0.8137764930725098),
 ('kick', 0.7954991459846497),
 ('pussy', 0.79521244764328),
 ('motherfucking', 0.7934619784355164),
 ('fuckin', 0.7922528982162476),
 ('delanoy', 0.7886987924575806),
 ('whore', 0.7860848903656006),
 ('wat', 0.7838656306266785),
 ('wanker', 0.781448245048523)]
```

Parameters Used:

num_features	Dimensionality of the word vectors. Eg, each rows contains 300 columns
min_word_count	Ignores all words with total frequency lower than this.
context	List of context words .Eg Synonyms
downsampling	Downsampimg for frequent words.

3)Using GloVe

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

CODE:

Following is the code:

```
# create embedding index
embedding_index = {}
with open('../input/glove6b100dtxt/glove.6B.100d.txt', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embedding_index[word] = coefs
```

+ Code + Markdown

```
embedding_index['good']
```

```
array([-0.030769 ,  0.11993  ,  0.53909  , -0.43696  , -0.73937  ,
       -0.15345  ,  0.081126 , -0.38559  , -0.68797  , -0.41632  ,
       -0.13183  , -0.24922  ,  0.441    ,  0.085919 ,  0.20871  ,
       -0.063582 ,  0.062228 , -0.051234 , -0.13398  ,  1.1418   ,
        0.036526 ,  0.49029  , -0.24567  , -0.412    ,  0.12349  ,
        0.41336  , -0.48397  , -0.54243  , -0.27787  , -0.26015  ,
       -0.38485  ,  0.78656  ,  0.1023   , -0.20712  ,  0.40751  ,
        0.32026  , -0.51052  ,  0.48362  , -0.0099498, -0.38685  ,
```

Since gloVe used pretrained data and gives weight according to that. Input file is downloaded from the gloVe repository.

CREATING WORD CLOUD

- Word Cloud is a data visualization technique used for representing text data in which the size of each word indicates its frequency or importance. Significant textual data points can be highlighted using a word cloud. Word clouds are widely used for analyzing data from social network websites.
- For generating word cloud in Python, modules needed are – matplotlib, pandas and wordcloud. To install these packages, run the following commands :

Words frequenced in harsh

Words frequenced in extremely_harsh

Words frequenced in vulgar

Words frequenced in threatening

Advantages Of Word Cloud

- Analyzing customer and employee feedback.
- Identifying new SEO keywords to target.

BALANCING DATASET

For balancing the dataset the number of rows where target=1 is equal to number of rows where target=0

CODE:

```
data_1 = tr_new[tr_new['harsh'] == 1 ]
data_harsh = tr_new[tr_new['harsh'] == 1 ].iloc[0:len(data_1),:]

data_1 = tr_new[tr_new['extremely_harsh'] == 1 ]
data_extremely_harsh = tr_new[tr_new['extremely_harsh'] == 1 ].iloc[0:len(data_1),:]

data_1 = tr_new[tr_new['vulgar'] == 1 ]
data_vulgar = tr_new[tr_new['vulgar'] == 1 ].iloc[0:len(data_1),:]

data_1 = tr_new[tr_new['threatening'] == 1 ]
data_threatening = tr_new[tr_new['threatening'] == 1 ].iloc[0:len(data_1),:]

data_1 = tr_new[tr_new['disrespect'] == 1 ]
data_disrespect = tr_new[tr_new['disrespect'] == 1 ].iloc[0:len(data_1),:]

data_1 = tr_new[tr_new['targeted_hate'] == 1 ]
data_targeted_hate = tr_new[tr_new['targeted_hate'] == 1 ].iloc[0:len(data_1),:]
```

Even After using this accuract remains moreover unchanged.

DATA MODELLING

1)LINEAR SCV CLASSIFIER

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

Code:

```
from sklearn.calibration import CalibratedClassifierCV
from sklearn.metrics import classification_report
from sklearn.metrics import roc_auc_score

SVC_pipeline=CalibratedClassifierCV(LinearSVC(class_weight='balanced'))
#data['id'] = ts_hate['id']
for category in categories:
    print('... Processing {}'.format(category))
    SVC_pipeline.fit(tr_fea, y_train[category])
    prediction=SVC_pipeline.predict(ts_fea)
    #print(roc_auc_score(y_test[category],prediction))
    print(classification_report(y_test[category], prediction))
    print(confusion_matrix(y_test[category], prediction))
```

Some Output of Classification report and confusion matrix:

```
... Processing threatening
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     29395
           1       0.52      0.28      0.36        94

    accuracy                           1.00     29489
   macro avg       0.76      0.64      0.68     29489
weighted avg       1.00      1.00      1.00     29489

[[29371    24]
 [   68    26]]
... Processing disrespect
              precision    recall  f1-score   support

           0       0.98      0.99      0.99     28031
           1       0.79      0.59      0.67      1458

    accuracy                           0.97     29489
   macro avg       0.88      0.79      0.83     29489
weighted avg       0.97      0.97      0.97     29489

[[27798   233]
 [  605   853]]
```

Parametes used in SVC are only **class_weight** for making sure that data model isn't too much biased towards one class. Use of **CalibratedCV** to give ouput in probability.

2. RIDGE & LOGISTIC CLASSIFIER

- In machine learning, ridge classification is a technique used to analyze linear discriminant models. It is a form of regularization that penalizes model coefficients to prevent overfitting.
- Overfitting is a common issue in machine learning that occurs when a model is too complex and captures noise in the data instead of the underlying signal. This can lead to poor generalization performance on new data.
- Ridge classification addresses this problem by adding a penalty term to the cost function that discourage complexity. This results in a model that is better able to generalize to new data.
- Ridge classification works by adding a penalty term to the cost function that discourages complexity. The penalty term is typically the sum of the squared coefficients of the features in the model. This forces the coefficients to remain small, which prevents overfitting.
- The amount of regularization can be controlled by changing the penalty term. A larger penalty results in more regularization and a smaller coefficient values. This can be beneficial when there is little training data available. However, if the penalty term is too large, it can result in underfitting.

Code:

```
overall_score = []
for x in categories:
        print('... Processing {}'.format(x))
        class_column = y_train[x].values
        score = 0
        if x in ['targeted_hate', 'threatening']:
            model = LogisticRegression(n_jobs=-1, random_state=0, C=3)
            model.fit(tr_fea,y_train[x])
            score = roc_auc_score(y_test[x],model.predict_proba(ts_fea)[:,1])
        else:
            rid =  RidgeClassifier(alpha=17,fit_intercept=True, solver='sag', max_iter=250,random_state=0,tol = 0.0005)
            rid.fit(tr_fea,y_train[x])
            d = rid.decision_function(ts_fea)
            probs = np.exp(d) / np.sum(np.exp(d))
            score = roc_auc_score(y_test[x],probs)
        overall_score.append(score)
        print(x,score)
print("average score", sum(overall_score)/len(overall_score))
```

Since we have many lables and we used binary classification i.e. we treated every lable as seprate entity. We Implement all the lables on different model and pick models which best suits the label.

Output of the above code:

```
... Processing harsh
harsh 0.977939723095994
... Processing extremely_harsh
extremely_harsh 0.9845253352792922
... Processing vulgar
vulgar 0.9907925284798883
... Processing threatening
threatening 0.9842081262915607
... Processing disrespect
disrespect 0.9802833909292764
... Processing targeted_hate
targeted_hate 0.9795055400683048
average score 0.9828757740240528
```

predicted output:

	harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate	id
0	0.000022	0.000025	0.000021	0.001148	0.000022	0.001187	e0ae9d9474a5689a5791
1	0.000042	0.000026	0.000026	0.001526	0.000032	0.013923	b64a191301cad4f11287
2	0.000027	0.000025	0.000025	0.001081	0.000028	0.003730	5e1953d9ae04bdc66408
3	0.000017	0.000025	0.000021	0.000584	0.000021	0.000418	23128f98196c8e8f7b90
4	0.000021	0.000025	0.000022	0.000146	0.000023	0.000111	2d3f1254f71472bf2b78
...	...	...	...	...	...	...	...
38292	0.000017	0.000025	0.000021	0.000160	0.000021	0.000249	64ebe2494b078bc1ec18
38293	0.000019	0.000026	0.000021	0.000225	0.000022	0.000361	16259bc32bd803e6acf5
38294	0.000020	0.000025	0.000021	0.000372	0.000021	0.001115	1fe631c9625d88a4d492
38295	0.000123	0.000056	0.000161	0.003300	0.000131	0.034981	085ab9387dce9d4e0b68
38296	0.000018	0.000025	0.000020	0.000228	0.000021	0.000344	4fb0f98b22a4f4469fcf

38297 rows × 7 columns

Hyperparameters:

1. **alpha**: float , default=1.0

   Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to $1 / (2C)$ in other linear models such as **LogisticRegression** or **LinearSVC**.

   For our model we pick **alpha=17** as it give the best result.

2. **fit_intercept**: bool, default=True

   Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

3. **max_iter**: int, default=None

   Maximum number of iterations for conjugate gradient solver. The default value is determined by scipy.sparse.linalg.

4. **Tol**: float, default=1e-4

   Precision of the solution. Note that tol has no effect for solvers 'svd' and 'cholesky'.

5. **Solver**: {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga', 'lbfgs'}, default='auto'

   Solver to use in the computational routines:

   •'auto' chooses the solver automatically based on the type of data.

   •'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. It is the most stable solver, in particular more stable for singular matrices than 'cholesky' at the cost of being slower.

   •'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.

   •'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set tol and max_iter).

   •'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.

   •'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its unbiased and more flexible version named SAGA. Both methods use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

Since Ridge and Logistic are very fast models and produces result mostly in minutes so for selecting the best hyper parameter we manually check. Which is again quite fasted than gridSearchCV.

3. ENSEMBLING USING VOTING CLASSIFIER

- A Voting Classifier is a machine learning model that trains on an ensemble of numerous models and predicts an output (class) based on their highest probability of chosen class as the output.
- It simply aggregates the findings of each classifier passed into Voting Classifier and predicts the output class based on the highest majority of voting
- The idea is instead of creating separate dedicated models and finding the accuracy for each them, we create a single model which trains by these models and predicts output based on their combined majority of voting for each output class.

Voting Classifier supports two types of votings.

1. **Hard Voting**: In hard voting, the predicted output class is a class with the highest majority of votes i.e the class which had the highest probability of being predicted by each of the classifiers. Suppose three classifiers predicted the output class(A, A, B), so here the majority predicted A as output. Hence A will be the final prediction.
2. **Soft Voting**: In soft voting, the output class is the prediction based on the average of probability given to that class. Suppose given some input to three models, the prediction probability for class A = (0.30, 0.47, 0.53) and B = (0.20, 0.32, 0.40). So the average for class A is 0.4333 and B is 0.3067, the winner is clearly class A because it had the highest probability averaged by each classifier.

CODE:

```
lg_clf = LogisticRegression(C = 10, penalty='l2', solver = 'liblinear', random_state=100)
svc_clf = CalibratedClassifierCV(LinearSVC(class_weight='balanced'))
lgb_clf = LGBMClassifier(scale_pos_weight=9)


voting_clf = VotingClassifier(estimators=[('lgb',lgb_clf),('svc',svc_clf),('lg',lg_clf)],voting='soft')


for category in categories:
    print('... Processing {}'.format(category))
    voting_clf.fit(train_features, y_train[category])
    # compute the testing accuracy
    prediction = voting_clf.predict(test_features)
    print('Test accuracy is {}'.format(accuracy_score(y_test[category], prediction)))
    print(classification_report(y_test[category], prediction))
    print(confusion_matrix(y_test[category],prediction))
```

Parameter used above is same as previously used. In LGBM we have used scale_pos_weight=9 such it will much priority to a class which has less values.

4. LSTM (for playing)

- LSTM stands for Long-Short Term Memory. LSTM is a type of recurrent neural network but is better than traditional recurrent neural networks in terms of memory. Having a good hold over memorizing certain patterns LSTMs perform fairly better. As with every other NN, LSTM can have multiple hidden layers and as it passes through every layer, the relevant information is kept and all the irrelevant information gets discarded in every single cell.

How does LSTM work?

LSTM has 4 main gate

1) FORGET Gate

This gate is responsible for deciding which information is kept for calculating the cell state and which is not relevant and can be discarded. The ht-1 is the information from the previous hidden state (previous cell) and xt is the information from the current cell. These are the 2 inputs given to the Forget gate. They are passed through a sigmoid function and the ones tending towards 0 are discarded, and others are passed further to calculate the cell state.

2) INPUT Gate

Input Gate updates the cell state and decides which information is important and which is not. As forget gate helps to discard the information, the input gate helps to find out important information and store certain data in the memory that relevant. ht-1 and xt are the inputs that are both passed through sigmoid and tanh functions respectively. tanh function regulates the network and reduces bias.

3) CELL Gate

All the information gained is then used to calculate the new cell state. The cell state is first multiplied with the output of the forget gate. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then a pointwise addition with the output from the input gate updates the cell state to new values that the neural network finds relevant.

4)OUTPUT Gate

The last gate which is the Output gate decides what the next hidden state should be. ht-1 and xt are passed to a sigmoid function. Then the newly modified cell state is passed through the tanh function and is multiplied with the sigmoid output to decide what information the hidden state should carry.

CODE:

Tokenize the data

```
tok = Tokenizer(num_words=1000, oov_token='UNK')
tok.fit_on_texts(complete_text)
# Extract binary BoW features
x_train = tok.texts_to_sequences(tr_new.text_new)
x_test = tok.texts_to_sequences(ts_hate.text_new)

vocab_size = len(tok.word_index) + 1
vocab_size
```

created a padded sequence for train/test split

```
training_padded = pad_sequences(x_train,
                                maxlen=50,
                                truncating='post',
                                padding='post'
                               )
test_padded = pad_sequences(x_test,
                            maxlen=50,
                            truncating='post',
                            padding='post'
                               )
```

Main model with 2 hidden layers using relu and sigmod, with 128 and 16 are the nodes in the hidden layers. Since we are using multilabel and binary classification so we used binary cross entropy loss and optimizer is Adamax.

```
model = models.Sequential()
model.add(layers.Embedding(vocab_size, 128, input_length=50))
model.add(layers.LSTM(512, dropout=0.2, recurrent_dropout=0.2, return_sequences=True))
model.add(layers.LSTM(128, dropout=0.2,recurrent_dropout=0.2))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(6, activation='sigmoid'))



model.compile(
    loss='binary_crossentropy',
    optimizer='Adamax',
    metrics=['accuracy'])

model.summary()
```

```
history = model.fit(training_padded,
                    labels,
                    epochs=3,
                    batch_size=512,
                    validation_split=0.2)
```

```
Epoch 1/3
2022-12-10 12:28:49.959460: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enable
stered 2)
140/140 [==============================] - 693s 5s/step - loss: 0.1744 - accuracy: 0.9545 - val_loss: 0.1452 - val_accuracy: 0.9947
Epoch 2/3
140/140 [==============================] - 682s 5s/step - loss: 0.1424 - accuracy: 0.9941 - val_loss: 0.1429 - val_accuracy: 0.9947
Epoch 3/3
140/140 [==============================] - 681s 5s/step - loss: 0.1414 - accuracy: 0.9941 - val loss: 0.1425 - val accuracy: 0.9947
```

Following are the ouput of the previous model:

```
predict_dic = model.predict(test_padded, verbose=0)
predict_dic
```

```
array([[0.09208524, 0.01049462, 0.05227268, 0.0054906 , 0.04889032,
        0.00803027],
       [0.0920856 , 0.01049465, 0.05227277, 0.0054906 , 0.04889032,
        0.00803027],
       [0.09208518, 0.01049465, 0.05227271, 0.0054906 , 0.04889032,
        0.00803027],
       ...,
       [0.09208533, 0.01049465, 0.05227271, 0.00549051, 0.04889032,
        0.00803027],
       [0.09208521, 0.01049465, 0.0522728 , 0.0054906 , 0.04889038,
        0.00803033],
       [0.09208602, 0.01049465, 0.05227277, 0.00549054, 0.04889029,
        0.00803038]], dtype=float32)
```

CONCLUSION

The best accuracy(ROC-AUC) was predicted in ridge and logistic classifier using TF-IdF vectorization which is 0.98551

harsh	extremely_harsh	vulgar	threatening	disrespect	targeted_hate	id
2.15621735827378E-05	2.54864368053687E-05	2.07122184208824E-05	0.001148087184646	2.15038759891062E-05	0.00118656456754 2	e0ae9d9474a5689a5791
4.19315620075559E-05	2.5553435624 2195E-05	2.55805533344795E-05	0.001526417306072	3.23340483139635E-05	0.013922757854647	b64a191301cad4f11287
2.70488895059246E-05	2.45891923932246E-05	2.45833101768243E-05	0.001080814246968	2.77717918328131E-05	0.003730217377339	5e1953d9ae04bdc66408
1.73108114276494E-05	2.5493060899084E-05	2.08863438341268E-05	0.000584062837554	2.12899200315594E-05	0.00041822934598	23128f98196c8e8f7b90
2.10107500210457E-05	2.50381129877676E-05	2.18394989892728E-05	0.00014554047455	2.30648302077894E-05	0.000111462163816	2d3f1254f71472bf2b78
3.392392350204 07E-05	2.49979608368354E-05	2.54618582860204E-05	0.000158242904549	2.85099951100403E-05	0.00044772561900 5	21f4f0f4812a08ea6c28
2.16984198298096E-05	2.53544672794698E-05	2.1279720350983E-05	0.000320785791587	2.37456301933482E-05	0.001229342832625	733b43d534c67c1be948
2.12599192065859E-05	2.58646503338994E-05	2.22892409308397E-05	0.000439527553557	2.42235816247507E-05	0.001515142375938	aad47a397f7ddc629d5d
2.29876905868944E-05	2.54600820085775E-05	2.22253351499412E-05	0.000426147472648	2.30076222973278E-05	0.000426658856375	d19fcde8a3af2e472d74

REFERENCES

- https://towardsdatascience.com/lemmatization-in-natural-language-processing-nlp-and-machine-learning-a4416f69a7b6#:~:text=Lemmatization%20in%20Natural%20Language%20Processing%20(NLP)%20and%20Machine%20Learning,-Source%3A%20Unsplash&text=Lemmatization%20is%20one%20of%20the,and%20machine%20learning%20in%20general.
- https://www.geeksforgeeks.org/understanding-tf-idf-term-frequency-inverse-document-frequency/
- https://neptune.ai/blog/vectorization-techniques-in-nlp-guide
- https://medium.com/@cmukesh8688/tf-idf-vectorizer-scikit-learn-dbc0244a911a
- https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa
- https://nlp.stanford.edu/projects/glove/
- https://www.geeksforgeeks.org/generating-word-cloud-python/
- https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html
- https://pythonprogramming.net/linear-svc-example-scikit-learn-svm-python/
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html
- https://vitalflux.com/ridge-classification-concepts-python-examples/
- https://www.geeksforgeeks.org/ml-voting-classifier-using-sklearn/

Kind Regards,
Mavericks