Natural Language Processing (NLP)

Course-End Project - Solution



**Help Twitter Combat Hate Speech Using NLP and Machine Learning**

**Objective:** Using NLP and ML, make a model identify hate speech (racist or sexist tweets) in Twitter.

**Problem Statement:**

Twitter is the biggest platform where anybody and everybody can have their views heard. Some of these voices spread hate and negativity. Twitter is wary of its platform being used as a medium to spread hate.

You are a Data Scientist at Twitter, and you will help Twitter in identifying the tweets with hate speech and removing them from the platform. You will use NLP techniques, perform specific cleanup for tweets data, and make a robust model.

**Domain:** Social Media

**Analysis to be done:** Clean up tweets and build a classification model by using NLP techniques, cleanup specific for tweets data, regularization and hyperparameter tuning using stratified k-fold and cross validation to get the best model.

**Content:**

id: identifier number of the tweet

Label: 0 (non-hate) /1 (hate)

Tweet: the text in the tweet

**Tasks:**

1. Load the tweets file using read\_csv function from Pandas package.

Importing the usual libraries.

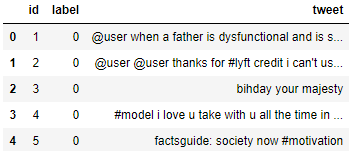
import pandas as pd, numpy as np

import os, re

Reading the csv file.

inp\_tweets0 = pd.read\_csv("TwitterHate.csv")

inp\_tweets0.head()



1. Check out the distribution of the label. Is there class imbalance?

inp\_tweets0.label.value\_counts(normalize=True)



Looks like there is high imbalance in the classes.

The modeling process will have to account for this.

1. Get the tweets into a list, for easy text cleanup and manipulation.

tweets0 = inp\_tweets0.tweet.values

1. Cleanup:
   1. Normalize the case.
   2. Using Regular expressions, remove user handles. These begin with '@.
   3. Using Regular expressions, remove URLs.
   4. Using TweetTokenizer from NLTK, tokenize the tweets into individual terms.
   5. Remove stop words.
   6. Remove redundant terms like ‘amp’, ‘rt’.
   7. Remove ‘#’ symbols from the tweet, while retaining the term.
   8. Extra cleanup by removing terms with a length of 1.

Normalizing case:

tweets\_lower = [twt.lower() for twt in tweets]

Removing user handles.

Testing on a test string:

import re

re.sub("@\w+","", "@Rahim this course rocks! http://rahimbaig.com/ai")



Applying on the data:

tweets\_nouser = [re.sub("@\w+","", twt) for twt in tweets\_lower]

Removing URLs.

Test string example:

re.sub("\w+://\S+","", "@Rahim this course rocks! http://rahimbaig.com/ai")



Applying on the data:

tweets\_nourl = [re.sub("\w+://\S+","", twt) for twt in tweets\_nouser]

Tokenizing using TweerTokenizer from NLTK:

from nltk.tokenize import TweetTokenizer

tkn = TweetTokenizer()

Applying the tokenizer on the data using list comprehension:

tweet\_token = [tkn.tokenize(sent) for sent in tweets\_nourl]

print(tweet\_token[0])

Removing punctuation, stop words, redundant terms like ‘rt’, ‘amp’, and removing terms with length of 1:

from nltk.corpus import stopwords

from string import punctuation

stop\_nltk = stopwords.words("english")

stop\_punct = list(punctuation)

Adding some specific punctuation from the data :

stop\_punct.extend(['...','``',"''",".."])

stop\_context = ['rt', 'amp']

Final stop word list including all of these:

stop\_final = stop\_nltk + stop\_punct + stop\_context

Define a function to:

**a**. Remove stop words from a single tokenized sentence.

**b**. Remove # tags.

**c.** Remove terms with length = 1.

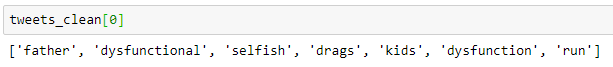
def del\_stop(sent):

return [re.sub("#","",term) for term in sent if ((term not in stop\_final) & (len(term)>1))]

Applying the function on the data:

tweets\_clean = [del\_stop(tweet) for tweet in tweet\_token]

Take a look at one of the tweets:



1. Check out the top terms in the tweets.
   1. First, get all the tokenized terms into one large list.
   2. Use counter and find the 10 most common terms.

Adding all terms to one huge list:

term\_list = []

for tweet in tweets\_clean:

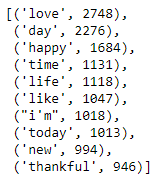
term\_list.extend(tweet)

Using counter to get top terms:

from collections import Counter

res = Counter(term\_list)

res.most\_common(10)



1. Data formatting for predictive modeling:
   1. Join the tokens back to form strings. This will be required for the vectorizers.

tweets\_clean = [" ".join(tweet) for tweet in tweets\_clean]

* 1. Assign x and y:

X = tweets\_clean

y = inp\_tweets0.label.values

* 1. Perform train\_test\_split using sklearn:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.30, random\_state=42)

1. You’ll use TF-IDF values for the terms as feature to get into a vector space model
   1. Import TF-IDF vectorizer from sklearn.

from sklearn.feature\_extraction.text import TfidfVectorizer

* 1. Instantiate with a maximum of 5000 terms in your vocabulary.

vectorizer = TfidfVectorizer(max\_features = 5000)

* 1. Fit and apply on the train set.

X\_train\_bow = vectorizer.fit\_transform(X\_train)

* 1. Apply on the test set.

X\_test\_bow = vectorizer.transform(X\_test)

X\_train\_bow.shape, X\_test\_bow.shape



1. Model building: Ordinary Logistic Regression
   1. Instantiate LogisticRegression from sklearn with default parameters.
   2. Fit on the train data.
   3. Make predictions for the train and the test set.

from sklearn.linear\_model import LogisticRegression

logreg = LogisticRegression()

Fitting on the train data:

logreg.fit(X\_train\_bow, y\_train)

Making predictions:

y\_train\_pred = logreg.predict(X\_train\_bow)

y\_test\_pred = logreg.predict(X\_test\_bow)

1. Model evaluation: Accuracy, recall, and f1\_score
   1. Report the accuracy on the train set.
   2. Report the recall on the train set:decent, high, or low?
   3. Get the f1\_score on the train set.

from sklearn.metrics import accuracy\_score, classification\_report

Checking the accuracy score:

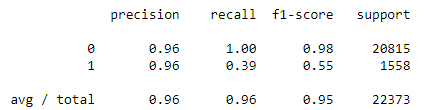
accuracy\_score(y\_train, y\_train\_pred)



95.6%. Is it impressive? Well, we do have a heavy class imbalance. So this accuracy may not be good if we are not capturing the 1s well at all.

So let’s look at the classification report:

print(classification\_report(y\_train, y\_train\_pred))



Oops! Recall is just 39% for 1 class which is not so great at all.

1. Looks like we need to adjust the class imbalance, as the model seems to focus on the 0s.
   1. Adjust the appropriate class in the LogisticRegression model.

Instantiating with the right parameter:

logreg = LogisticRegression(class\_weight="balanced")

1. Train again with the adjustment and evaluate.
   1. Train the model on the train set.
   2. Evaluate the predictions on the train set: accuracy, recall, and f1\_score.

logreg.fit(X\_train\_bow, y\_train)

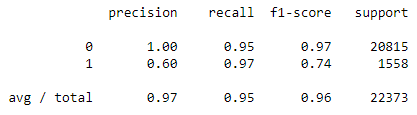
Evaluating the train set:

y\_train\_pred = logreg.predict(X\_train\_bow)

accuracy\_score(y\_train, y\_train\_pred)

Accuracy score is 95.35%. It is a little lower than the previous model, but the classification report will give better details.

print(classification\_report(y\_train, y\_train\_pred))



So much better on the train set! A recall of 97%! Looks like the parameter improved a lot.

The f1\_score is also better at 0.74.

Note that this is still the training data. The performance could be lower on the test set.

1. Regularization and Hyperparameter tuning:
   1. Import GridSearch and StratifiedKFold because of class imbalance.
   2. Provide the parameter grid to choose for ‘C’ and ‘penalty’ parameters.
   3. Use a balanced class weight while instantiating the logistic regression.

from sklearn.model\_selection import GridSearchCV, StratifiedKFold

Create the parameter grid based on the results of random search.

param\_grid = {

'C': [0.01,0.1,1,10,100],

'penalty': ["l1","l2"]

}

Instantiating the logistic regression model with a balanced class weight:

classifier\_lr = LogisticRegression(class\_weight="balanced")

1. Find the parameters with the best recall in cross validation.
   1. Choose ‘recall’ as the metric for scoring.
   2. Choose stratified 4 fold cross validation scheme.
   3. Fit it on the train set.

You’ll supply stratified k-fold as out cv strategy to GridSearchCV. You need to stratify as there is heavy class imbalance in the dataset.

grid\_search = GridSearchCV(estimator = classifier\_lr, param\_grid = param\_grid,

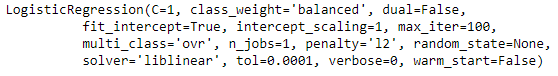
cv = StratifiedKFold(4), n\_jobs = -1, verbose = 1, scoring = "recall" )

Fitting on the train data to get the best parameters:

grid\_search.fit(X\_train\_bow, y\_train)

1. What are the best parameters?

grid\_search.best\_estimator\_

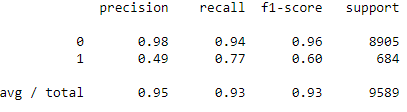


From cross validation, the best parameters are: C = 1, penalty = “l2”.

1. Predict and evaluate using the best estimator.
   1. Use the best estimator from the grid search to make predictions on the test set.
   2. What is the recall on the test set for the toxic comments?
   3. What is the f1\_score?

y\_test\_pred = grid\_search.best\_estimator\_.predict(X\_test\_bow)

print(classification\_report(y\_test, y\_test\_pred))



Looks like you did a good job on the test set. The f1\_score for 1 class is 0.60 and the recall is 0.77. Great!