Q1. **HTSPC- Hate Speech Classification**

Packages/lbraries needed to be installed :

- Natural Language Toolkit (NLTK) library
- tweet-preprocessor module
- wordninja

Input format :

We use command line arguments to take the input of the paths of the files.
We take the path of the training data csv file as our first command line argument and path of test data file as our second argument.

Data preprocessing :

1. First we use the tweet preprocessor library in Python to remove the URLs, emojis, numbers and mention tags present in the data.

The code snippet is:

In this section we are removing all URLs, emojis, numbers and mention tags from the data.

```
import preprocessor as p
p.set_options(p.OPT.URL, p.OPT.EMOJI, p.OPT.SMILEY, p.OPT.NUMBER, p.OPT.MENTION)
i = 0
for line in X:
    cleaned_line = p.clean(line)
    X[i] = cleaned_line
    i = i+1
```

2. We use the wordninja module to seperate the different words.

Using the wordninja module, we are splitting the data into seperate words.

```
[46] import wordninja
     i = 0
     for text in X:
         arr = wordninja.split(text)
         arr1 = ""
         for j in arr:
             arr1 = arr1+" "+j
         X[i] = arr1
         i = i+1
```

3. We remove the special characters and extra spaces from the data by writing regular expressions and matching them to the strings.

In this section we are removing the special characters and extra spaces from the data.

```
[ ] import re
    no_space = re.compile("(&(\w*))|(@(\w*))|(\;)|(\')|(#)|(\.)|(\;)|(\:)|(\!)|(\*)|(\')|(\?)|(\,)|(\")|(\()|(\))|(\[)|(\])")
    space = re.compile("(\-)|(\/)")
    single_digits = re.compile(r"\b[A-Za-z]\b")
    digits = re.compile("\d+")
    extra_spaces = re.compile(r'\s+')
    backslash = re.compile(r'\\')

    def preprocess_reviews(reviews):

        reviews = [no_space.sub("", line.lower()) for line in reviews]
        reviews = [space.sub(" ", line) for line in reviews]
        reviews = [single_digits.sub(" ", line) for line in reviews]
        reviews = [digits.sub(" ", line) for line in reviews]
        reviews = [backslash.sub(" ", line) for line in reviews]
        reviews = [extra_spaces.sub(" ", line) for line in reviews]

        return reviews
```

4. We perform lemmatization, a process of grouping together the different inflected forms of a word so they can be analyzed as a single item.

Performing lemmatization

```
[1319] from nltk.stem import WordNetLemmatizer
       def get_lemmatized_text(corpus):
           lemmatizer = WordNetLemmatizer()
           corpus = [' '.join([lemmatizer.lemmatize(word) for word in line.split()]) for line in corpus]
           return corpus
```

5. We vectorize using Tfidf vectorizer, to transform our documents into vector representations.

Vectorizer

```
vectorizer = TfidfVectorizer(stop_words = 'english', ngram_range=(1, 2))
X = vectorizer.fit_transform(X)
```

Here, we remove the stop words – words that occur most frequently in the English language, since they will not be much useful in classifying hate speech.

We perform many other tests, in order to check what performs better. These are as follows -

- We test by using the stopwords of the nltk library, and get comparable results.
  The code snippet is given below -

This section deals with removing of stop words.

```
from nltk.corpus import stopwords
stopwords = stopwords.words('english')
def remove_stop_words(corpus):
    preprocessed_data = list()
    for line in corpus:
        preprocessed_data.append(' '.join([word for word in line.split() if word not in stopwords]))
    return preprocessed_data
```

- We also test using CountVectorizer.
- We test using different ngram ranges.

Model selection :

Now, we perform train validation data split .
After that, we run different models to test which one perform better.
The outcomes are shown below :

1. Using logistic regression model with different values of C

Logistic Regression model

```
[113] from sklearn.linear_model import LogisticRegression

      c = [0.001, 0.01, 0.05, 0.5, 0.1, 1.0]
      for i in c:
          model_lr = LogisticRegression(C=i)
          model_lr.fit(X_train, y_train)
          y_pred = model_lr.predict(X_test)
          accuracy_lr = accuracy_score(y_test, y_pred)
          f1_lr = f1_score(y_test, y_pred)
          print ("C = ", i, " Accuracy: ", accuracy_lr)
          print ("C = ", i, " F1 score: ", f1_lr)
```

```
C =  0.001  Accuracy:  0.635673624288425
C =  0.001  F1 score:  0.7735849056603773
C =  0.01  Accuracy:  0.6603415559772297
C =  0.01  F1 score:  0.7716836734693877
C =  0.05  Accuracy:  0.6802656546489564
C =  0.05  F1 score:  0.7715254237288135
C =  0.5  Accuracy:  0.6622390891840607
C =  0.5  F1 score:  0.751048951048951
C =  0.1  Accuracy:  0.6802656546489564
C =  0.1  F1 score:  0.7696514012303487
C =  1.0  Accuracy:  0.6574952561669829
C =  1.0  F1 score:  0.7448763250883393
```

2. Using Linear SVC model with different values of C

Linear SVC model

```
from sklearn.svm import LinearSVC

c = [0.001, 0.01, 0.05, 0.5, 0.1, 1.0]
for i in c:
    model_svm = LinearSVC(C=i)
    model_svm.fit(X_train, y_train)
    y_pred = model_svm.predict(X_test)
    accuracy_svm = accuracy_score(y_test, y_pred)
    f1_svm = f1_score(y_test, y_pred)
    print ("C = ", i, " Accuracy: ", accuracy_svm)
    print ("C = ", i, " F1 score: ", f1_svm)
```

```
C =  0.001  Accuracy:  0.6660341555977229
C =  0.001  F1 score:  0.7740693196405648
C =  0.01  Accuracy:  0.6755218216318786
C =  0.01  F1 score:  0.7663934426229508
C =  0.05  Accuracy:  0.6574952561669829
C =  0.05  F1 score:  0.7477288609364081
C =  0.5  Accuracy:  0.6404174573055028
C =  0.5  F1 score:  0.7294789436117058
C =  0.1  Accuracy:  0.6584440227703985
C =  0.1  F1 score:  0.7457627118644068
C =  1.0  Accuracy:  0.6299810246679317
C =  1.0  F1 score:  0.7202295552367288
```

3. Using SVM with ‘linear’ kernel

SVM with 'linear' kernel

```
svm = SVC(kernel = 'linear')
svm.fit(X_train, y_train)
y_pred = svm.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred)
f1_svm = f1_score(y_test, y_pred)
print ("Accuracy: ", accuracy_svm)
print (" F1 score: ", f1_svm)
```

```
Accuracy:  0.6328273244781784
 F1 score:  0.7205776173285199
```

4. Using SVM with ‘poly’ kernel

SVM with 'poly' kernel

```
[134] svm = SVC(kernel = 'poly')
      svm.fit(X_train, y_train)
      y_pred = svm.predict(X_test)
      accuracy_svm = accuracy_score(y_test, y_pred)
      f1_svm = f1_score(y_test, y_pred)
      print ("Accuracy: ", accuracy_svm)
      print (" F1 score: ", f1_svm)
```

```
Accuracy:  0.6385199240986718
 F1 score:  0.7700663850331926
```

5. Using SVM with ‘rbf’ kernel

SVM with 'rbf' kernel

```
[132] svm = SVC(kernel = 'rbf')
      svm.fit(X_train, y_train)
      y_pred = svm.predict(X_test)
      accuracy_svm = accuracy_score(y_test, y_pred)
      f1_svm = f1_score(y_test, y_pred)
      print ("Accuracy: ", accuracy_svm)
      print (" F1 score: ", f1_svm)
```

```
Accuracy:  0.6717267552182163
 F1 score:  0.7747395833333333
```

From the above outcomes we choose SVM with ‘rbf’ kernel as our final model, as it performs pretty well.

We load the test data, and apply the same preprocessing on it, as we applied on the train data.

Finally, we save our predictions in a csv file.

We get an accuracy of around 80% on the final test data.