Why So Harsh?
Harsh Comment Classification
(Project Report)

Team - Tech Knights

Group Members:

IMT2020079 - Surya Sastry

IMT2020094 - Riddhi Chatterjee

Pre Processing

1.1. Checking for missing values

```
[16]  train_df.isna().sum()

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

There are no missing values in the training dataset. Checking further...

```
[17]  (train_df == "?").sum()

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

Indeed there are no missing values in the training dataset

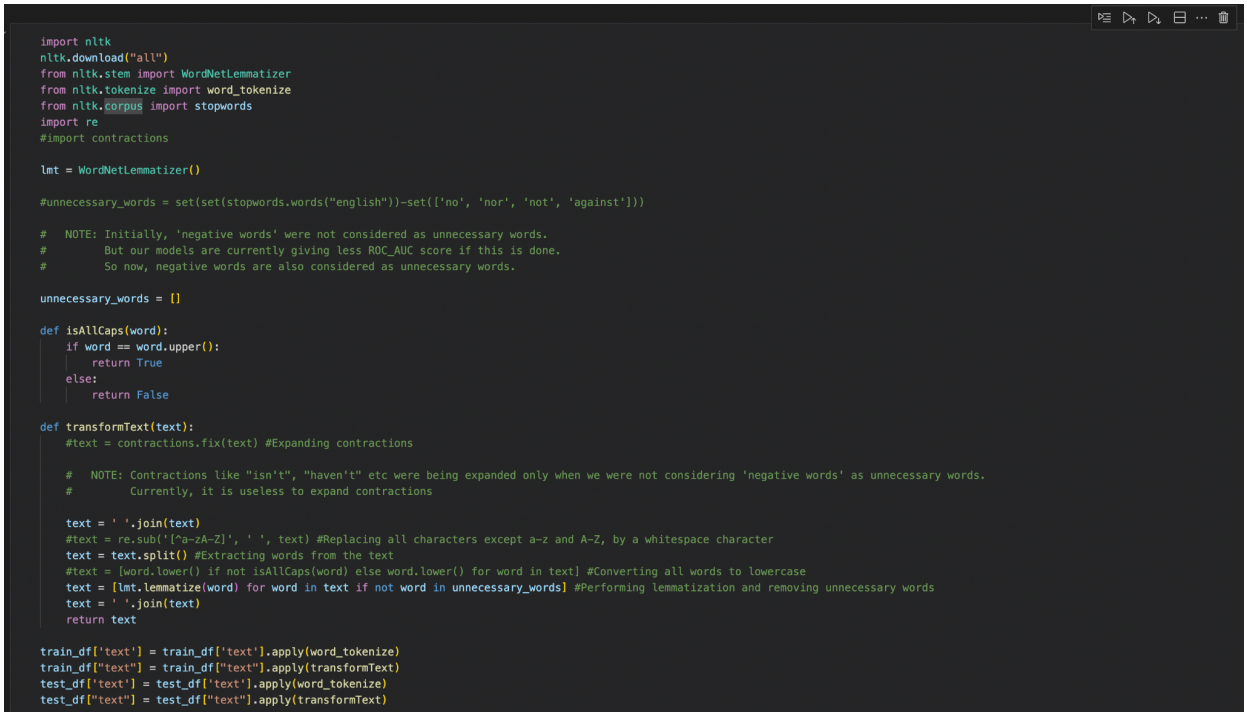
```
[18]  test_df.isna().sum()

...   id      0
      text    0
      dtype: int64
```

No missing values in the test dataset. Checking further...

```
[19]  (test_df == "?").sum()

...   id      0
      text    0
      dtype: int64
```

There are no missing values in both the training and the test datasets.

1.2. Lemmatization of the text and Manual Text Clean Up



```
import nltk
nltk.download("all")
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import re
#import contractions

lmt = WordNetLemmatizer()

#unnecessary_words = set(set(stopwords.words("english"))-set(['no', 'nor', 'not', 'against']))

#   NOTE: Initially, 'negative words' were not considered as unnecessary words.
#         But our models are currently giving less ROC_AUC score if this is done.
#         So now, negative words are also considered as unnecessary words.

unnecessary_words = []

def isAllCaps(word):
    if word == word.upper():
        return True
    else:
        return False

def transformText(text):
    #text = contractions.fix(text) #Expanding contractions

    #   NOTE: Contractions like "isn't", "haven't" etc were being expanded only when we were not considering 'negative words' as unnecessary words.
    #         Currently, it is useless to expand contractions

    text = ' '.join(text)
    #text = re.sub('[^a-zA-Z]', ' ', text) #Replacing all characters except a-z and A-Z, by a whitespace character
    text = text.split() #Extracting words from the text
    #text = [word.lower() if not isAllCaps(word) else word.lower() for word in text] #Converting all words to lowercase
    text = [lmt.lemmatize(word) for word in text if not word in unnecessary_words] #Performing lemmatization and removing unnecessary words
    text = ' '.join(text)
    return text

train_df['text'] = train_df['text'].apply(word_tokenize)
train_df["text"] = train_df["text"].apply(transformText)
test_df['text'] = test_df['text'].apply(word_tokenize)
test_df["text"] = test_df["text"].apply(transformText)
```

The aim of lemmatization, like stemming, is to reduce inflectional forms to a common base form. As opposed to stemming, lemmatization does not simply chop off inflections. Instead, it uses lexical knowledge bases to get the correct base forms of words.

The NLTK python library has been used to perform lemmatization.

Since we are analyzing emotions, the meaning of the word has an utmost importance. Lemmatization would be recommended when the meaning of the word is important for analysis because Lemmatization always gives the dictionary meaning while converting into root-form.

Note: Few operations such as “expansion of contractions”, “not converting all-capital lettered words into lowercase” and “removing duplicate rows in the training dataset” had been performed initially. But since they didn’t improve the roc_auc score, these operations are not being performed currently.

1.3. Oversampling the data

```python
from imblearn.over_sampling import RandomOverSampler

performOverSampling = False #Oversampling is performed if this flag is True

def over_sample(label_type_index):
    ros = RandomOverSampler(random_state=0)
    X_resampled = ''
    y_resampled = ''
    if performOverSampling:
        X_resampled, y_resampled = ros.fit_resample(train_df['text'].to_numpy().reshape(-1,1), y_train[label_type[label_type_index]].to_numpy().reshape(-1,1))
    else:
        X_resampled = train_df['text'].to_numpy().reshape(-1,1)
        y_resampled = y_train[label_type[label_type_index]].to_numpy().reshape(-1,1)
    return (X_resampled, y_resampled)
```

Python

```python
train_df_X_dict = {}
train_df_y_dict = {}
for i in range(6):
    XTrain, labels = over_sample(i)
    train_df_X_dict[label_type[i]] = pd.DataFrame(XTrain)[0]
    train_df_y_dict[label_type[i]] = pd.DataFrame(labels)[0]
```

Python

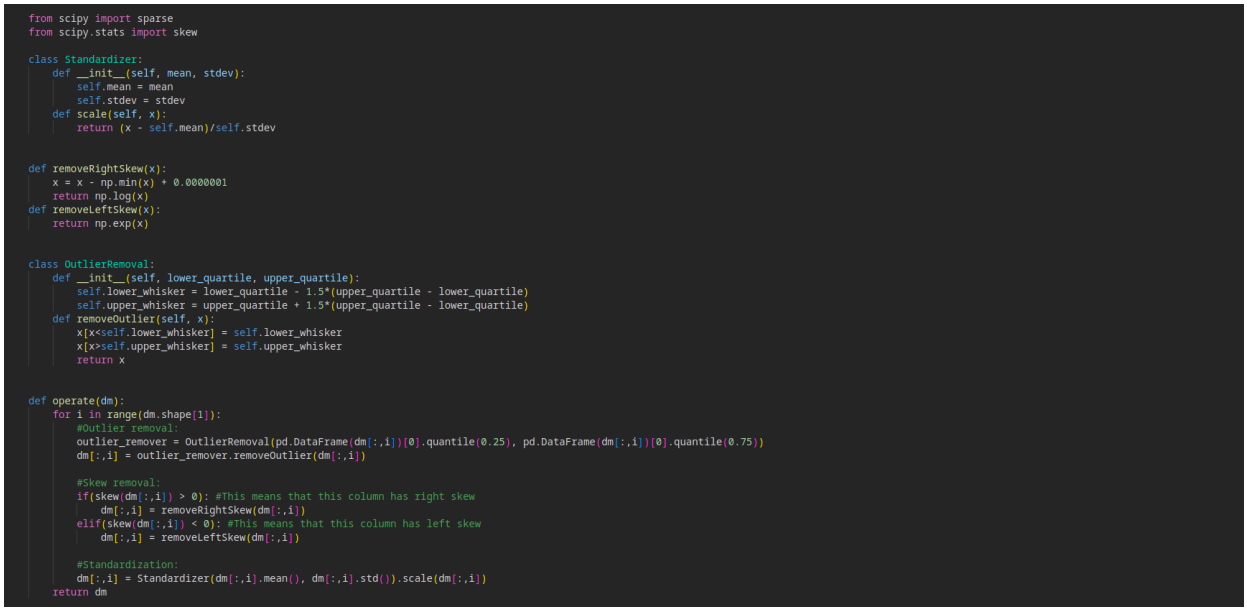
Imbalanced datasets are those where there is a severe skew in the class distribution. This bias in the training dataset can influence many machine learning algorithms, leading some to ignore the minority class entirely. One approach to addressing the problem of class imbalance is to duplicate examples from the minority class (oversampling).

The given dataset is highly imbalanced. Thus, oversampling was tried out individually for each of the classes using the imblearn library.

Note: Even though oversampling worked well for logistic regression, it didn't work well for the voting classifier (which is

currently our best classification model). Thus we were forced to ditch oversampling.

1.4. Standardization and skew, outlier removal



```
from scipy import sparse
from scipy.stats import skew

class Standardizer:
    def __init__(self, mean, stdev):
        self.mean = mean
        self.stdev = stdev
    def scale(self, x):
        return (x - self.mean)/self.stdev


def removeRightSkew(x):
    x = x - np.min(x) + 0.0000001
    return np.log(x)
def removeLeftSkew(x):
    return np.exp(x)


class OutlierRemoval:
    def __init__(self, lower_quartile, upper_quartile):
        self.lower_whisker = lower_quartile - 1.5*(upper_quartile - lower_quartile)
        self.upper_whisker = upper_quartile + 1.5*(upper_quartile - lower_quartile)
    def removeOutlier(self, x):
        x[x<self.lower_whisker] = self.lower_whisker
        x[x>self.upper_whisker] = self.upper_whisker
        return x


def operate(dm):
    for i in range(dm.shape[1]):
        #Outlier removal:
        outlier_remover = OutlierRemoval(pd.DataFrame(dm[:,i])[0].quantile(0.25), pd.DataFrame(dm[:,i])[0].quantile(0.75))
        dm[:,i] = outlier_remover.removeOutlier(dm[:,i])

        #Skew removal:
        if(skew(dm[:,i]) > 0): #This means that this column has right skew
            dm[:,i] = removeRightSkew(dm[:,i])
        elif(skew(dm[:,i]) < 0): #This means that this column has left skew
            dm[:,i] = removeLeftSkew(dm[:,i])

        #Standardization:
        dm[:,i] = Standardizer(dm[:,i].mean(), dm[:,i].std()).scale(dm[:,i])
    return dm
```

Classes and functions have been implemented for standardization (using mean and standard deviation), skew removal and outlier removal. All these operations have been performed in the operate().

The scipy library has been used to decide whether a particular column/attribute has left skew or right skew, so that we can remove the skew appropriately.

1.5. Data Matrices for models

i) Bag of words

```
from sklearn.feature_extraction.text import CountVectorizer
def getBagOfWords(train_text, test_text):
    cv = CountVectorizer()
    #column_list = [train_text, test_text]
    #combined_data = pd.concat(column_list, ignore_index=True)
    cv.fit(train_text) #Fitting on the training data
    Xtrainbow = cv.transform(train_text)
    Xtestbow = cv.transform(test_text)
    print(Xtrainbow.shape)
    print(Xtestbow.shape)
    return (Xtrainbow, Xtestbow)

_, X_test_bow = getBagOfWords(train_df['text'], test_df['text'])
X_train_bow = {}
for i in range(6):
    X_train_bow[label_type[i]], _ = getBagOfWords(train_df_X_dict[label_type[i]], test_df['text'])
```

It's a collection of words to represent a sentence with word count and mostly disregarding the order in which they appear. It creates a vocabulary of all the unique words occurring in all the documents. For this, CountVectorizer from sklearn has been used.

ii) TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer

def getTFIDF(train_text, test_text):
    #train_text = pd.DataFrame(train_text)
    #test_text = pd.DataFrame(test_text)

    train_text = train_text
    test_text = test_text

    total_text = pd.concat([train_text, test_text])


    vectorizer = TfidfVectorizer(strip_accents = 'unicode', analyzer = 'word', token_pattern = r"(?u)\b\w\w+\b", lowercase = True, stop_words = 'english', ngram_range = (1, 1), norm = 'l2', sublinear_tf = True)
    vectorizer.fit(total_text)
    train_word_features = vectorizer.transform(train_text)
    test_word_features = vectorizer.transform(test_text)

    char_vectorizer = TfidfVectorizer (strip_accents = 'unicode', analyzer = 'char', ngram_range = (3, 5), sublinear_tf = True)
    char_vectorizer.fit(total_text)
    train_char_features = char_vectorizer.transform(train_text)
    test_char_features = char_vectorizer.transform(test_text)

    train_features = np.hstack([train_char_features, train_word_features])
    test_features = np.hstack([test_char_features, test_word_features])

    Xtraintfidf = train_features[0]
    Xtesttfidf = test_features[0]
    print(Xtraintfidf.shape)
    print(Xtesttfidf.shape)
    return (Xtraintfidf, Xtesttfidf)

_, X_test_tfidf = getTFIDF(train_df['text'], test_df['text'])
X_train_tfidf = {}
for i in range(6):
    X_train_tfidf[label_type[i]], _ = getTFIDF(train_df_X_dict[label_type[i]], test_df['text'])
```

TF-IDF or (Term Frequency(TF)- Inverse Dense Frequency(IDF))is a technique which is used to find meaning of sentences consisting of words and cancels out the incapabilities of Bag of Words technique which is good for text classification or for helping a machine read words in numbers.

IDF =Log[(# Number of documents) / (Number of documents containing the word)]
TF = (Number of repetitions of word in a document) / (# of words in a document)
Take the product of these two terms.

iii) Doc2Vec - PV-DM and PV-DBOW

```
def getDoc2Vec(train_text, test_text):
    train_df_tokenized = []
    test_df_tokenized = []

    for text in train_text:
        train_df_tokenized.append(text.split())

    for text in test_text:
        test_df_tokenized.append(text.split())

    my_texts = train_df_tokenized + test_df_tokenized #Consists of texts from both the training and test data

    #Using 'my_texts' to train the Doc2Vec models:
    documents = [TaggedDocument(doc, [i]) for i, doc in enumerate(my_texts)]
    model_PV_DM = Doc2Vec(documents, vector_size=2000, window=2, min_count=1, workers=4, dm=1) #Paragraph Vector - Distributed Memory
    model_PV_DBOW = Doc2Vec(documents, vector_size=2000, window=2, min_count=1, workers=4, dm=0) #Paragraph Vector - Distributed Bag of Words

    #Obtaining data matrices for PV-DM and PV-DBOW:
    lst = []
    for word_list in train_df_tokenized:
        lst.append(model_PV_DM.infer_vector(word_list))
    XtrainPVDM = np.array(lst)

    lst = []
    for word_list in test_df_tokenized:
        lst.append(model_PV_DM.infer_vector(word_list))
    XtestPVDM = np.array(lst)

    lst = []
    for word_list in train_df_tokenized:
        lst.append(model_PV_DBOW.infer_vector(word_list))
    XtrainPVDBOW = np.array(lst)

    lst = []
    for word_list in test_df_tokenized:
        lst.append(model_PV_DBOW.infer_vector(word_list))
    XtestPVDBOW = np.array(lst)
```

PVDM - randomly sample consecutive words from a paragraph and predict a center word from the randomly sampled set of words by taking as input, the context words and a paragraph id

DBOW - The DBOW model ignores the context words in the input, but forces the model to predict words randomly sampled from the paragraph in the output.

iv) Word2Vec

```python
from gensim.test.utils import common_texts
from gensim.models import Word2Vec

def getWord2VecMethod1(train_text, test_text):
    my_texts = [] #Consists of texts from both the training and test data
    for text in train_text:
        my_texts.append(text.split())
    for text in test_text:
        my_texts.append(text.split())

    #vectorSize = int(X_train_tfidf.shape[1]/2)
    vectorSize = 1000
    model = Word2Vec(sentences=my_texts, vector_size=vectorSize, window=5, min_count=1, workers=4) #Using 'my_texts' to train the Word2Vec model
    dummyFV = np.array([0]*2*vectorSize)

    featureLst = []
    for text in train_text:
        if text == "":
            featureLst.append(dummyFV)
            continue
        min = model.wv[text.split()[0]]
        max = min
        for word in text.split():
            min = np.minimum(model.wv[word], min) #Obtaining the coordinate-wise minimum of the vector representations of all the words in the text
            max = np.maximum(model.wv[word], max) #Obtaining the coordinate-wise maximum of the vector representations of all the words in the text
        featureVec = np.concatenate((min, max)) #Concatenating min and max to obtain a feature vector for each data point
        featureLst.append(featureVec)

    XtrainW2Vmethod1 = np.stack(featureLst) #Obtaining the data matrix for the training data
    print(XtrainW2Vmethod1.shape)
```

Word2Vec is a combination of models used to represent distributed representations of words in a corpus C. Word2Vec (W2V) is an algorithm that accepts text corpus as an input and outputs a vector representation for each word. The vectors we use to represent words are called neural word embeddings.

Classification Models

The execute() — a general function that handles training and testing of any model with any data matrix:

```python
def execute(clf, model_name, data_matrix_name, csv_filename, mode = "8020Split"):
    #General function to train and test models on different data matrices

    train_data = data_matrices[data_matrix_name][0]
    test_data = data_matrices[data_matrix_name][1]

    if not exists("TestClassProbs"):
        os.system("mkdir TestClassProbs")
    if not exists("ModelOutputs"):
        os.system("mkdir ModelOutputs")

    fname = "ModelOutputs/"+csv_filename
    fname = fname[:len(fname)-9] + ".out"
    with open(fname, "w") as f:
        f.write("Model: "+model_name+"\n")
        f.write("Data matrix: "+data_matrix_name+"\n\n")

        if mode == "GS": #Fits the model on the entire training data, after performing 5 fold grid search CV based on roc_auc score
            data = train_data
            CVScores, CVParams = clf.fit(data, train_df_y_dict, 'roc_auc', 5)

            f.write("Tuned parameters after 5 fold grid search CV:\n")
            for i in range(6):
                f.write("Parameters for "+label_type[i]+": "+str(CVParams[label_type[i]])+"\n")
            f.write("\n")
            f.write("Best roc_auc score after 5 fold grid search CV:\n")
            for i in range(6):
                f.write("roc_auc score for "+label_type[i]+": "+str(CVScores[label_type[i]])+"\n")
            f.write("\n")

        elif mode == "WGS": #Fits the model on the entire training data, without performing grid search CV
            data = train_data
            clf.fitWithoutGridSearch(data, train_df_y_dict)
```

```python
    elif mode == "8020Split":
        #Performs a 80-20 split of the training data
        #Fits the model on the train split, without performing grid search CV
        #Validates the model on the validation/test split
        data = train_data
        X_train_new = {}
        X_val = {}
        y_train_new = {}
        y_val = {}
        for i in range(6):
            X_train_new[label_type[i]], X_val[label_type[i]], y_train_new[label_type[i]], y_val[label_type[i]] = train_test_split(data[label_type[i]],
            train_df_y_dict[label_type[i]], test_size=0.2, shuffle=False)
        # clf.fit(X_train_new, y_train_new, 'roc_auc', 5)
        clf.fitWithoutGridSearch(X_train_new, y_train_new)
        probs = clf.predictProbabilities(X_val)
        f.write("ROC_AUC scores for the test split:\n")
        mean_ROC_AUC_score = 0
        for i in range(6):
            labels = y_val[label_type[i]]
            ROC_AUC_score = roc_auc_score(labels, probs[label_type[i]])
            f.write("ROC_AUC score for "+label_type[i]+": "+str(ROC_AUC_score)+"\n")
            mean_ROC_AUC_score += ROC_AUC_score
        mean_ROC_AUC_score = mean_ROC_AUC_score/6
        f.write("\nMean ROC_AUC_score: "+str(mean_ROC_AUC_score)+"\n")
        f.write("\n")
    #Calculating accuracy and f1 scores for the entire training data:
    data = train_data
    predictions = clf.predict(data)
    f.write("Accuracy and f1 scores for the entire training data:\n")
    for i in range(6):
        labels = train_df_y_dict[label_type[i]]
        f.write("Accuracy score for "+label_type[i]+": "+str(accuracy_score(labels, predictions[label_type[i]]))+"\n")
        f.write("F1 score for "+label_type[i]+": "+str(f1_score(labels, predictions[label_type[i]]))+"\n")
    f.write("\n")
    #Calculating final class probabilities for test data:
    data = test_data
    probs = clf.predictProbabilities(data)
    df=pd.DataFrame([])
    df['id'] = test_df["id"]
    column_values = ['harsh', 'extremely_harsh', 'vulgar', 'threatening', 'disrespect', 'targeted_hate']
    for column_type in column_values:
        df[column_type] = probs[column_type]
    print("Final class probabilities for first 20 samples of test data:")
    print(df.loc[0:20,:])
    df.to_csv("TestClassProbs/"+csv_filename, index = False) #Storing final class probabilities for the test data in a csv file
```

The execute() currently supports the following three modes of operation:

- *“GS” mode: Fits the model on the entire training data, after performing a 5-fold grid search CV based on roc_auc score.*
- *“WGS” mode: Fits the model on the entire training data, without performing grid search CV.*
- *“80-20 Split” mode:*
    - *Performs a 80-20 split of the training data*
    - *Fits the model on the train split, without performing grid search CV*
    - *Validates the model on the validation/test split*

  Though not as effective as the grid search CV method, this “80-20 Split” method has been used to test model performance for all our models. The main reason for this is that the “80-20 Split” method is much faster than the grid search CV method.

The classification models developed are as follows:

Note: For every model, 5-Fold grid search CV is also implemented for automatically fine tuning the hyperparameters. But we have resorted to manual fine tuning of hyperparameters for all models since the grid search CV approach takes a lot of time.

i) Logistic Regression

```python
from sklearn.linear_model import LogisticRegression
from imblearn.over_sampling import RandomOverSampler
from scipy import sparse

class LogisticRegClassifier(object): #Pack of 6 binary Logistic Regression classifiers
    def __init__(self, initial_params):
        self.param_grid = {
            'solver' : ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
            }
        self.models = []

        for i in range(6): #Appending 6 binary Logistic Regression classifiers (one for each class)
            self.models.append(LogisticRegression(max_iter=initial_params[i]['max_iter'], class_weight=initial_params[i]['class_weight'],
            penalty=initial_params[i]['penalty'], solver=initial_params[i]['solver'], C=initial_params[i]['C'], tol=1e-9))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = LogisticRegression(solver=bestParams['solver'],penalty='l2', max_iter=100000, class_weight='balanced')
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            self.models[i].fit(XTrain[label_type[i]], yTrain[label_type[i]])
            print("Fitted model "+str(i))
```

Logistic Regression is a Supervised statistical technique to find the probability of a dependent variable. Logistic regression uses functions called the logit functions,that help derive a relationship between the dependent variable and independent variables by predicting the probabilities or chances of occurrence.

Hyperparameters:

- max_iter: Maximum number of iterations taken for the solvers to converge
- class_weight: These are the weights associated with the classes
  The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples/(n_classes * np.bincount(y))
- penalty: l1, l2 or elastic net regularization/penalty
- solver: Denotes the algorithm to be used in the optimisation problem
- C: Inverse of regularization strength (smaller values specify stronger regularization)
- tol: Tolerance for stopping criteria

The following functionalities are similar for all the models:

```python
def predict(self, X): #Returns the binary classification results for each of the models
    isDict = False
    X_data = X
    if type(X) == dict:
        isDict = True
    predictions = {}
    for i, model in enumerate(self.models):
        if isDict:
            X_data = X[label_type[i]]
        predictions[label_type[i]] = model.predict(X_data)
    return predictions

def predictProbabilities(self, X): #Returns the predicted class probabilities for each of the classes
    isDict = False
    X_data = X
    if type(X) == dict:
        isDict = True
    probabilities = {}
    for i, model in enumerate(self.models):
        if isDict:
            X_data = X[label_type[i]]
        classes = model.classes_
        index = np.where(classes == 1)[0][0] #Getting the index of class 1
        probabilities[label_type[i]] = model.predict_proba(X_data)[:, index] #Extracting the probabilities of class 1
    return probabilities
```

Manual fine tuning of the hyperparameters:

```python
IP = {
            0 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'lbfgs', 'C':1.9},
            1 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':0.175},
            2 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.8},
            3 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.6},
            4 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.55},
            5 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':2}
        } #Initial parameters for logistic regression
```

Using the “80-20 Split” mode of the execute function we verified that the above set of hyperparameters is one of the best for the logistic regression model.

Trying out the logistic regression model with different data matrices:
(similarly we have tried out other models also with the various data matrices)

```
clf = LogisticRegClassifier(IP)
csv_filename = "LogisticReg_bow_Prob.csv"
model_name = "Logistic Regression Classifier"
data_matrix_name = "Bag of Words"
#execute(clf, model_name, data_matrix_name, csv_filename, "WGS")
execute(clf, model_name, data_matrix_name, csv_filename)
```

```
clf = LogisticRegClassifier(IP)
csv_filename = "LogisticReg_tfidf_Prob.csv"
model_name = "Logistic Regression Classifier"
data_matrix_name = "TF-IDF"
#execute(clf, model_name, data_matrix_name, csv_filename, "WGS")
execute(clf, model_name, data_matrix_name, csv_filename)
```

```
clf = LogisticRegClassifier(IP)
csv_filename = "LogisticReg_PV_DBOW_Prob.csv"
model_name = "Logistic Regression Classifier"
data_matrix_name = "Paragraph Vector - Distributed Bag Of Words"
execute(clf, model_name, data_matrix_name, csv_filename)
```

```
clf = LogisticRegClassifier(IP)
csv_filename = "LogisticReg_PV_DM_Prob.csv"
model_name = "Logistic Regression Classifier"
data_matrix_name = "Paragraph Vector - Distributed Memory"
execute(clf, model_name, data_matrix_name, csv_filename)
```

```
clf = LogisticRegClassifier(IP)
csv_filename = "LogisticReg_W2V_method_1_Prob.csv"
model_name = "Logistic Regression Classifier"
data_matrix_name = "Word2Vec Method_1"
execute(clf, model_name, data_matrix_name, csv_filename)
```

ii) Voting Classifier

```python
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier

class VotClassifier(object): #Pack of 6 binary Voting classifiers
    def __init__(self, initial_params):
        self.param_grid = {
            'voting' : ['soft']
            }
        self.models = []
        self.initial_params = initial_params
        max_depth=850
        max_features=5000
        max_samples=None
        min_samples_leaf=100

        for i in range(6): #Appending 6 binary Logistic Regression classifiers (one for each class)
            clf1 = MultinomialNB()
            clf2 = SGDClassifier(loss='modified_huber')
            clf3 = BaggingClassifier(base_estimator=LogisticRegression(max_iter=self.initial_params[i]['max_iter'], class_weight=self.initial_params[i]['class_weight'],
            penalty=self.initial_params[i]['penalty'], solver=self.initial_params[i]['solver'], C=self.initial_params[i]['C'], tol=1e-9), max_samples=1.0,
            n_estimators = 50, oob_score=True)
            clf4 = LogisticRegression(max_iter=self.initial_params[i]['max_iter'], class_weight=self.initial_params[i]['class_weight'],
            penalty=self.initial_params[i]['penalty'], solver=self.initial_params[i]['solver'], C=self.initial_params[i]['C'], tol=1e-9)
            clf6 = RandomForestClassifier(max_depth=max_depth, max_features=max_features, max_samples=max_samples, min_samples_leaf = min_samples_leaf)
            clf7 = MLPClassifier(random_state=1)
            clf10 = DecisionTreeClassifier()
            self.models.append(VotingClassifier(estimators = [('mnbc', clf1), ('sgdc', clf2), ('bgc', clf3), ('lr', clf4), ('rdf', clf6),
            ('mlp', clf7), ('dtc', clf10)], voting = 'soft',weights=[1,3,5,5,5,3,1]))
```

```python
    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore
            self.models[i] = VotingClassifier(estimators = [('lr1', LogisticRegression(max_iter=self.initial_params[i]['max_iter'],
            class_weight=self.initial_params[i]['class_weight'], penalty=self.initial_params[i]['penalty'], solver=self.initial_params[i]['solver'],
            C=self.initial_params[i]['C'], tol=1e-9)), ('lr2', LogisticRegression(max_iter=self.initial_params[i]['max_iter'],
            class_weight=self.initial_params[i]['class_weight'], penalty=self.initial_params[i]['penalty'], solver=self.initial_params[i]['solver'],
            C=self.initial_params[i]['C'], tol=1e-9)), ('lr3', LogisticRegression(max_iter=self.initial_params[i]['max_iter'],
            class_weight=self.initial_params[i]['class_weight'], penalty=self.initial_params[i]['penalty'], solver=self.initial_params[i]['solver'],
            C=self.initial_params[i]['C'], tol=1e-9))], voting = bestParams['voting'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
        return CVScores, CVParams
    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))

    def predict(self, X): #Returns the binary classification results for each of the models
        isDict = False
        X_data = X
        if type(X) == dict:
            isDict = True
        predictions = {}
        for i, model in enumerate(self.models):
            if isDict:
                X_data = X[label_type[i]]
            predictions[label_type[i]] = model.predict(X_data)
        return predictions
```

A collection of several models working together on a single set is called an ensemble. The method is called Ensemble Learning. It is much more useful to use all different models rather than any one. Voting is one of the simplest ways of combining the predictions from multiple machine learning algorithms. Voting classifier isn't an actual classifier but a wrapper for a set of different ones that are trained and evaluated in parallel in order to exploit the different peculiarities of each algorithm.

```
IP = {
            0 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'lbfgs', 'C':1.9},
            1 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':0.175},
            2 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.8},
            3 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.6},
            4 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.55},
            5 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':2}
        } #Initial parameters for logistic regression
```

Hyperparameters:

- estimators: Invoking the fitmethod on the Voting Classifier will fit clones of those original estimators that will be stored in the class attribute self_estimators..An estimator can be set to drop using set_params.
- voting: If ‘hard’, uses predicted class labels for majority rule voting. Else if ‘soft’, predicts the class label based on the argmax of the sums of the predicted probabilities, which is recommended for an ensemble of well-calibrated classifiers.
- weights: Sequence of weights to weight the occurrences of predicted class labels or class probabilities before averaging . Uses uniform weights if None.

iii) SVM Classifier

```
from sklearn import svm

class SVMClassifier(object): #Pack of 6 binary SVM classifiers
    def __init__(self):
        self.param_grid = {
            'C' : [1.0], #default=1.0
            'gamma' : ['auto'], #default='scale'
        }
        self.models = []

        for i in range(6): #Appending 6 binary SVM classifiers (one for each class)
            self.models.append(svm.SVC(kernel = 'poly', degree = 5, max_iter=1000, probability=True))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = svm.SVC(C = bestParams['C'], gamma = bestParams['gamma'], kernel = 'poly', degree = 5, max_iter=1000, probability=True)
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

The basic principle behind SVM is to draw a hyperplane that best separates the two classes, so you start off by drawing a random hyperplane and then you check the distance between the hyperplane and the closest data points from each class. The hyperplane which has the maximum distance from the support vectors is the most optimal hyperplane.

Hyperparameters:

- gamma: Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’.
- C: Inverse of regularization strength (smaller values specify stronger regularization)
- kernel: Specifies the kernel type to be used in the algorithm. If none is given, ‘rbf’ will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape
- degree: Degree of the polynomial kernel function (‘poly’). Must be non-negative. Ignored by all other kernels.
- max_iter: Maximum number of iterations taken for the solvers to converge
- probability: Whether to enable probability estimates. This must be enabled prior to calling fit, will slow down that method as it internally uses 5-fold cross-validation, and predict_proba may be inconsistent with predict.

iv) Gaussian Naive Bayes

```python
from sklearn.naive_bayes import GaussianNB

class GNBClassifier(object): #Pack of 6 binary Gaussian naive bayes classifiers
    def __init__(self):
        self.param_grid = {
            'var_smoothing' : [1e-9] #default=1e-9
        }
        self.models = []

        for i in range(6): #Appending 6 binary Gaussian naive bayes classifiers (one for each class)
            self.models.append(GaussianNB())

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = GaussianNB(var_smoothing=bestParams['var_smoothing'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

Naive Bayes models ultimately take any new, previously unseen data point, and calculate the probabilities of that data point separately as if it belonged to each different class, then choose the most probable class based on which probability is highest.

Gaussian Naive Bayes is a variant of Naive Bayes which supports continuous values and has an assumption that each class is normally distributed.

Hyperparameters:

- var_smoothing: Portion of the largest variance of all features that is added to variances for calculation stability.

v) Multinomial Naive Bayes

```python
from sklearn.naive_bayes import MultinomialNB

class MNBClassifier(object): #Pack of 6 binary Multinomial naive bayes classifiers
    def __init__(self):
        self.param_grid = {
            'alpha' : [1.0] #default=1.0
        }
        self.models = []

        for i in range(6): #Appending 6 binary Multinomial naive bayes classifiers (one for each class)
            self.models.append(MultinomialNB())

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = MultinomialNB(alpha=bestParams['alpha'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

Multinomial Naive Bayes is a variant which is an event-based model that has features as vectors where the sample represents frequencies with which certain events have occurred.

Hyperparameters:

- alpha: Additive (Laplace/Lidstone) smoothing parameter (set alpha=0 and force_alpha=True, for no smoothing).

vi) Random Forest

```python
from sklearn.ensemble import RandomForestClassifier

class RFClassifier(object): #Pack of 6 binary Random forest classifiers
    def __init__(self, max_depth=850, max_features=5000, max_samples=None, min_samples_leaf=100):
        self.max_depth = max_depth
        self.max_features = max_features
        self.max_samples = max_samples
        self.min_samples_leaf = min_samples_leaf
        self.param_grid = {
            'n_estimators' : [100, 200, 300], #default=100
            'criterion' : ['gini', 'entropy'], #default="gini"
        }
        self.models = []
        for i in range(6): #Appending 6 binary Random forest classifiers (one for each class)
            self.models.append(RandomForestClassifier(max_depth=self.max_depth, max_features=self.max_features,
            max_samples=self.max_samples, min_samples_leaf = self.min_samples_leaf))
    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore
            self.models[i] = RandomForestClassifier(n_estimators = bestParams['n_estimators'], criterion = bestParams['criterion'],
            max_depth=self.max_depth, max_features=self.max_features, max_samples=self.max_samples, min_samples_leaf = self.min_samples_leaf)
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))
        return CVScores, CVParams
    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

In a Random Forest, each decision tree in the forest considers a random subset of features when forming questions and only has access to a random set of the training data points. Random Forests are an ensemble of k untrained Decision Trees (trees with only a root node) with M bootstrap samples (k and M do not have to be the same) trained using a variant of the ***random subspace method.***

Hyperparameters:

- n_estimators: The number of trees in the forest
- max_depth: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- max_features: The number of features to consider when looking for the best split
- max_samples: the number of samples to draw from X to train each base estimator
- min_samples_leaf: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression

- criterion: The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “log_loss” and “entropy” both for the Shannon information gain.
  This parameter is tree-specific.

vii) Bagging Classifier

```
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import LogisticRegression

class BGClassifier(object): #Pack of 6 binary Bagging classifiers
    def __init__(self, base_params):
        self.param_grid = {
            'n_estimators' : [10, 20, 30], #default=10
        }
        self.models = []
        self.base_params = base_params
        for i in range(6): #Appending 6 binary Bagging classifiers (one for each class)
            self.models.append(BaggingClassifier(base_estimator=LogisticRegression(max_iter=self.base_params[i]['max_iter'],
            class_weight=self.base_params[i]['class_weight'], penalty=self.base_params[i]['penalty'], solver=self.base_params[i]['solver'],
            C=self.base_params[i]['C'], tol=1e-9), max_samples=1.0, n_estimators = 50, oob_score=True))
    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the best CVScores and CVParams after performing grid search CV
        CVScores = {}
        CVParams = {}
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = BaggingClassifier(base_estimator=LogisticRegression(max_iter=self.base_params[i]['max_iter'],
            class_weight=self.base_params[i]['class_weight'], penalty=self.base_params[i]['penalty'], solver=self.base_params[i]['solver'],
            C=self.base_params[i]['C'], tol=1e-9), max_samples=1.0, n_estimators = bestParams['n_estimators'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

Bagging is used when our goal is to reduce the variance of a decision tree. Here the idea is to create several subsets of data from training samples chosen randomly with replacement. Now, each collection of subset data is used to train their decision trees. As a result, we end up with an ensemble of different models. Average of all the predictions from different trees are used which is more robust than a single decision tree.

```
BP = {
            0 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'lbfgs', 'C':1.9},
            1 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':0.175},
            2 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.8},
            3 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.6},
            4 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':1.55},
            5 : {'max_iter':100000, 'class_weight':'balanced', 'penalty':'l2', 'solver':'liblinear', 'C':2}
        } #Parameters for the base estimator
```

Hyperparameters:

- base_estimator: The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a DecisionTreeClassifier.
- oob_score: Whether to use out-of-bag samples to estimate the generalization error
- max_samples: The number of samples to draw from X to train each base estimator
- n_estimators: The number of base estimators in the ensemble

viii) Adaboost

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression

class ADBClassifier(object): #Pack of 6 binary AdaBoost classifiers
    def __init__(self):
        self.param_grid = {
            'n_estimators' : [5] #default=50
        }
        self.models = []

        for i in range(6):
            self.models.append(AdaBoostClassifier(base_estimator=LogisticRegression(max_iter=100000, class_weight='balanced', penalty='l2')))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the CVScores and CVParams after performing cross validation
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = AdaBoostClassifier(n_estimators = bestParams['n_estimators'],
            base_estimator=LogisticRegression(max_iter=100000, class_weight='balanced', penalty='l2'))
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

AdaBoost is used to boost the performance of decision trees on binary classification problems. Adaboost creates a forest of trees and is used to make classification better than other methods. It combines a lot of "weak learners" to make classifications. Most of them are referred to as stumps. Some of them get more weightage in classification than the others. New stumps are made taking the mistakes of previous stumps into account.

Hyperparameters:

- n_estimators: The number of trees in the forest
- base_estimator: The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper classes and n_classes attributes. If None, then the base estimator is DecisionTreeClassifier initialized with max_depth = 1

ix) Stochastic Gradient Descent Classifier

```python
from sklearn.linear_model import SGDClassifier
class StochasticGDClassifier(object): #Pack of 6 binary Stochastic Gradient Descent classifiers
    def __init__(self):
        self.param_grid = {
            'loss' : ['modified_huber']
        }
        self.models = []

        for i in range(6):
            self.models.append(SGDClassifier(loss='modified_huber'))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the CVScores and CVParams after performing cross validation
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = SGDClassifier(loss=bestParams['loss'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

Stochastic gradient descent performs a parameter update for each observation. So instead of looping over each observation, it just needs one to perform the parameter update. SGD is usually faster than batch gradient descent, but its frequent updates cause a higher variance in the error rate that can sometimes jump around instead of decreasing.

Hyperparameters:

- loss: This is the loss function to be used

x) Gradient Boosting Classifier

```python
from sklearn.ensemble import GradientBoostingClassifier
class GBClassifier(object): #Pack of 6 binary Gradient Boosting classifiers
    def __init__(self):
        self.param_grid = {
            'loss' : ['log_loss'] #default='log_loss'
        }
        self.models = []

        for i in range(6):
            self.models.append(GradientBoostingClassifier())

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the CVScores and CVParams after performing cross validation
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = GradientBoostingClassifier(loss=bestParams['loss'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

Given the gradient, calculate the change in the parameters with the learning rate. Recalculate the new gradient with the new value of the parameter. Repeat until convergence. Convergence is a name given to the situation where the loss function does not improve significantly, and we are stuck in a point near to the minima.
A gradient-boosted trees model is built in a stage-wise fashion as in other boosting methods, but it generalizes the other methods by allowing optimization of an arbitrary differentiable loss function.

Hyperparameters:

- loss: the loss function to be used

xi) XGBoost Classifier

```python
from xgboost.sklearn import XGBClassifier
class XGBoostClassifier(object): #Pack of 6 binary xgboost classifiers
    def __init__(self):
        self.param_grid = {
            'n_estimators' : [100, 200]
        }
        self.models = []

        for i in range(6):
            self.models.append(XGBClassifier(n_estimators = 200))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the CVScores and CVParams after performing cross validation
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = XGBClassifier(n_estimators=bestParams['n_estimators'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

As opposed to the bagging where trees are built parallelly, in boosting, the trees are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each tree learns from its predecessors and updates the residual errors.
In XGBoost, we fit a model on the gradient of loss generated from the previous step. Modification of the gradient boosting algorithm takes place so that it works with any differentiable loss function.

Hyperparameters:

- n_estimators: The number of trees in the forest

xii) Gaussian Process Classifier

```python
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF

class GProcessClassifier(object): #Pack of 6 binary Gaussian Process classifiers
    def __init__(self):
        self.param_grid = {
            'n_restarts_optimizer' : [0]
        }
        self.models = []

        for i in range(6):
            self.models.append(GaussianProcessClassifier(1.0 * RBF(1.0)))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the CVScores and CVParams after performing cross validation
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = GaussianProcessClassifier(1.0 * RBF(1.0), n_restarts_optimizer = bestParams['n_restarts_optimizer'])
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

The prediction interpolates the observations (at least for regular kernels).The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and decide based on those if one should refit (online fitting, adaptive fitting) the prediction in some region of interest. Versatile: different kernels can be specified. Common kernels are provided, but it is also possible to specify custom kernels.

Hyperparameters:

- n_restarts_optimizer: The number of restarts of the optimizer for finding the kernel's parameters which maximize the log-marginal likelihood.
- kernel: The kernel specifying the covariance function of the GP. If None is passed, the kernel "1.0 * RBF(1.0)" is used as default. Note that the kernel's hyperparameters are optimized during fitting. Also the kernel cannot be a CompoundKernel.

xiii) KNN Classifier

```python
from sklearn.neighbors import KNeighborsClassifier
class KNNClassifier(object): #Pack of 6 binary K-Nearest Neighbors classifiers
    def __init__(self, n_neighbors=5, algorithm='auto'):
        self.param_grid = {
            'n_neighbors' : [5] #default=5
        }
        self.models = []
        self.n_neighbors = n_neighbors
        self.algorithm = algorithm

        for i in range(6):
            self.models.append(KNeighborsClassifier(n_neighbors=self.n_neighbors, algorithm=self.algorithm))

    def fit(self, XTrain, yTrain, CVScoreType, numFolds): #Fits the models and returns the CVScores and CVParams after performing cross validation
        CVScores = {}
        CVParams = {}

        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]

            bestParams, bestScore = performGridSearchCV(model, self.param_grid, XTrain[label_type[i]], labels, numFolds, CVScoreType)
            CVParams[label_type[i]] = bestParams
            CVScores[label_type[i]] = bestScore

            self.models[i] = KNeighborsClassifier(n_neighbors=bestParams['n_neighbors'], algorithm=self.algorithm)
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted Model "+str(i))

        return CVScores, CVParams

    def fitWithoutGridSearch(self, XTrain, yTrain): #Fits the models without performing grid search CV
        for i, model in enumerate(self.models):
            labels = yTrain[label_type[i]]
            self.models[i].fit(XTrain[label_type[i]], labels)
            print("Fitted model "+str(i))
```

KNN tries to predict the correct class for the test data by calculating the distance between the test data and all the training points. Then select the K number of points which is closest to the test data. The KNN algorithm calculates the probability of the test data belonging to the classes of 'K' training data and which class holds the highest probability will be selected.

Hyperparameters:

- n_neighbours: Number of neighbors to use by default for kneighbors queries
- algorithm: Algorithm used to compute the nearest neighbors

xiv) LSTM (Long Short Term Memory)

Our implementation of the LSTM model:

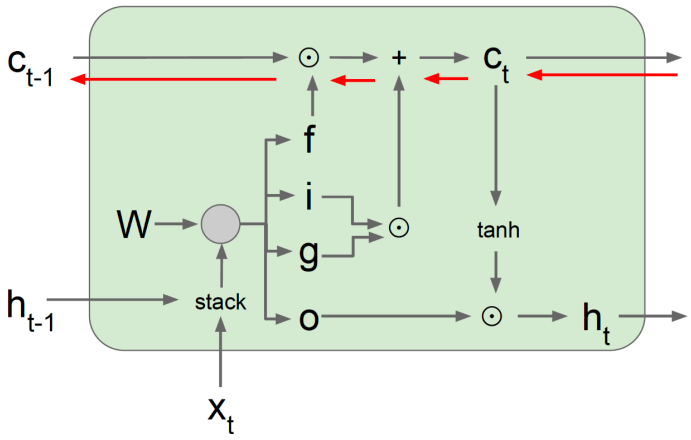
```python
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.nn import functional as F
import signal
import sys

def signal_handler(sig, frame):
    print('\nExiting...')
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)

class LSTM(nn.Module):
    def __init__(self, nb_layers, nb_lstm_units, inputSize):
        super(LSTM, self).__init__()
        self.device = 'cpu'
        if torch.cuda.is_available():
            self.device = 'cuda'
        self.nb_layers = nb_layers
        self.nb_lstm_units = nb_lstm_units
        self.input_size = inputSize

        # build actual NN
        self.__build_model()

    def __build_model(self):
        # design LSTM
        self.lstm = nn.LSTM(
            input_size = self.input_size,
            hidden_size=self.nb_lstm_units,
            num_layers=self.nb_layers,
            batch_first=True,
        ).to(self.device)

        # linear layer:
        self.fully_connected = nn.Linear(self.nb_lstm_units, 2).to(self.device)

        #Sigmoid layer:
        self.sigmoid = nn.Sigmoid().to(self.device)

        #Softmax layer:
        self.softmax = nn.Softmax().to(self.device)
```

```
def init_hidden(self):
    # the weights are of the form (nb_layers, batch_size, nb_lstm_units)
    hidden_a = torch.rand(self.nb_layers, self.batch_size, self.nb_lstm_units).to(self.device)
    hidden_b = torch.rand(self.nb_layers, self.batch_size, self.nb_lstm_units).to(self.device)

    hidden_a = Variable(hidden_a)
    hidden_b = Variable(hidden_b)

    return (hidden_a, hidden_b)

def getInputs(self, X, X_lengths):
    '''
    max_len = 0
    X_lengths = []
    num_features = 0
    for seq in X_list:
        X_lengths.append(len(seq))
        if len(seq) != 0:
            num_features = len(seq[0])
        if len(seq) > max_len:
            max_len = len(seq)

    padList = [0]*num_features

    for i in range(len(X_list)):
        iter = max_len - len(X_list[i])
        for j in range(iter):
            X_list[i].append(padList)

    X = torch.tensor(X_list, dtype = torch.float32)
    '''

    X = X.to(self.device)
    t = torch.tensor(X_lengths).to(self.device)
    sorted, indices = torch.sort(t, descending=True)
    X_lengths = [int(x) for x in sorted]
    X_copy = torch.clone(X)
    for ind1, ind2 in enumerate(indices):
        X[int(ind1)] = X_copy[int(ind2)]

    #print(X)
    #print(X_lengths)
    return [X, X_lengths, indices]
```

```python
def forward(self, X_padded, X_lengths):
    X, X_lengths, indices = self.getInputs(X_padded, X_lengths)
    self.batch_size, seq_len, _ = X.size()

    # reset the LSTM hidden state. Must be done before you run a new batch. Otherwise the LSTM will treat
    # a new batch as a continuation of a sequence

    self.hidden = self.init_hidden()

    # Dim transformation: (batch_size, seq_len, embedding_dim) -> (batch_size, seq_len, nb_lstm_units)

    # pack_padded_sequence so that padded items in the sequence won't be shown to the LSTM
    out = torch.nn.utils.rnn.pack_padded_sequence(X, X_lengths, batch_first=True)

    # now run through LSTM
    out, self.hidden = self.lstm(out, self.hidden)

    # undo the packing operation
    out, _ = torch.nn.utils.rnn.pad_packed_sequence(out, batch_first=True)

    out = self.fully_connected(out[:, out.shape[1]-1,:])
    #out = out.view(out.shape[0])

    #out = (self.sigmoid(out) * 40) + 10
    #out = (torch.tanh(out) * 20) + 30
    out = self.softmax(out) * 1


    out_copy = torch.clone(out)
    for ind1, ind2 in enumerate(indices):
        out[int(ind2)] = torch.clone(out_copy[int(ind1)])

    return out
```

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

LSTM (Long Short Term Memory) is an advanced recurrent neural network which takes care of the vanishing gradients problem in typical RNNs.
An LSTM model was implemented using PyTorch along with appropriate training & testing scripts, dataset classes etc. (All files related to our LSTM implementation are present in the 'LSTM_Implementation' folder).

***Note:** LSTMs require a lot of careful fine tuning and a lot of time to train properly. Thus, though immensely powerful, the LSTMs could only give us a maximum Kaggle ROC_AUC score of 0.95283 (using a 3-layer LSTM, learning_rate=0.0005, batchSize=1 and training only upto 30 epochs).*

Hyperparameters:

- Number of layers in the model
- Learning rate during gradient descent
- Batch size used while training
- Number of epochs up to which the model is trained

Selecting best class probabilities from all the combinations

```
csv_file = {
    'harsh': "VOTC_tfidf_Prob.csv",
    'extremely_harsh' : "VOTC_tfidf_Prob.csv",
    'vulgar': "VOTC_tfidf_Prob.csv",
    'threatening': "VOTC_tfidf_Prob.csv",
    'disrespect': "VOTC_tfidf_Prob.csv",
    'targeted_hate' : "VOTC_tfidf_Prob.csv"
    }

submission_df = pd.DataFrame([])
submission_df['id'] = test_df['id']
for i in range(6):
    submission_df[label_type[i]] = pd.read_csv("TestClassProbs/"+csv_file[label_type[i]])[label_type[i]]
submission_df.to_csv("submission.csv",index=False)
```

Final results obtained

The following table contains the mean ROC_AUC scores for the various combinations of classification models and data matrices:

Data matrix type / Classification model	TF-IDF	Bag Of Words	Doc2Vec (Paragraph Vector - Distributed Bag Of Words)	Doc2Vec (Paragraph Vector - Distributed Memory)	Word2Vec
AdaBoost Classifier	0.9380884020460277	0.9393081104618224	0.9578820434634792	0.9068647443165099	0.944908794368378
Bagging Classifier	0.9815074936053981	NA	NA	NA	NA
Gradient Boosting Classifier	0.8985247423424721	0.8971740974349486	0.9660024911725135	0.9253236078010296	0.9480535821841102
Gaussian Naive Bayes Classifier	NA	NA	0.7573152114734976	NA	0.7357535624835708
K-Nearest Neighbors Classifier	0.7802246383599435	NA	NA	NA	NA
Logistic Regression Classifier	0.962840497616202	NA	0.920913388912318	0.8694948578612708	NA
Multinomial Naive Bayes Classifier	NA	NA	0.9233882330549498	0.7930696527883915	0.8590691785786717
Random Forest Classifier	0.9561036932981178	0.9564984424623103	NA	NA	NA
Stochastic Gradient Descent Classifier	0.970547817200199	0.7142653055505183	0.7460921554026984	0.6844604152105376	0.7266562169324087
SVM Classifier	0.753555219698749	0.7091825292083515	0.7150996845021589	0.660900243040093	0.6211164945846107
Voting Classifier	0.9853 (Kaggle score)	NA	NA	NA	NA
XGBoost Classifier	0.9700436748671525	0.9687267802541962	NA	NA	0.962130737808827
LSTM (Long Short Term Memory)	NA	NA	NA	NA	0.95283 (Best kaggle score after some fine tuning)

Note:

- The mean ROC_AUC scores shown above have been mostly obtained by using the “80-20 Split” mode of the execute().
- For a few combinations we do not currently have the ROC_AUC score values (marked as “NA”) due to various reasons such as too much execution time, incompatible combination of model and data matrix, workload constraints etc.
- For the Voting Classifier (our best classification model), currently we only have the kaggle score because it has been executed using the “WGS” mode of the execute() (i.e. by using the entire training data for training purposes without a grid search). The “WGS” mode doesn’t provide

any estimate of mean ROC_AUC score. Instead it gives us the following output:

```
Model: Voting Classifier
Data matrix: TF-IDF

Accuracy and f1 scores for the entire training data:
Accuracy score for harsh: 0.990297563759666
F1 score for harsh: 0.949008998412045
Accuracy score for extremely_harsh: 0.994393401895724
F1 score for extremely_harsh: 0.7750336775931747
Accuracy score for vulgar: 0.9946060273727325
F1 score for vulgar: 0.9505843756407627
Accuracy score for threatening: 0.9998545194104679
F1 score for threatening: 0.9759704251386322
Accuracy score for disrespect: 0.9901073199118163
F1 score for disrespect: 0.9050687285223368
Accuracy score for targeted_hate: 0.9988697277274813
F1 score for targeted_hate: 0.9400593471810089
```

- The above scores were logged automatically as we kept on testing and further improving our models. Thus some of these scores may be outdated if they were logged during the initial tests that we conducted.

Resources used

- https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
- https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessRegressor.html
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html
- https://medium.com/analytics-vidhya/understanding-logistic-regression-b3c672deac04
- https://medium.com/analytics-vidhya/tf-idf-term-frequency-technique-easiest-explanation-for-text-classification-in-nlp-with-code-8ca3912e58c3
- https://medium.com/@amarbudhiraja/understanding-document-embeddings-of-doc2vec-bfe7237a26da
- https://medium.com/@zafaralibagh6/a-simple-word2vec-tutorial-61e64e38a6a1
- https://www.sciencedirect.com/science/article/pii/S1877050921000752?ref=pdf_download&fr=RR-2&rr=7702b57d1c266efb
- https://www.youtube.com/watch?v=6niqTuYFZLQ&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=10
- Representation learning for very short texts using weighted word embedding aggregation: https://arxiv.org/pdf/1607.00570.pdf