ML AI511 Project

Why so Harsh?

Team name: PP Outliers

MT2022069 – Parul Ghotikar

MT2022071 – Parv Parikh

Data Analysis and Pre-Processing

- ## Lemmatization

  NLTK library is used for performing lemmatization or stemming on any text data for sentiment analysis.
  Lemmatization is the process of grouping together different inflected forms of a word so that they can be analysed as a single item.
  Lemmatization of train and text data is done at the same time.
  Removal of stop words was taken care of while vectorization(TFIDF)

- **Why lemmatization over stemming?**
  - With stemming words are reduced to their word stems. A word stem need not be the same root as dictionary based root. Example: history, historical – history
  - Lemmatization will group history and historical as a single group and common name would be history

- ## Manual Removal of words
  - Replacing words, symbols and emojis which are not making any sense with meaningful words for better vectorization results.
  - Remove words like http www which contain Url. Only relevant part is retained.
  - Removing all numeric values, and punctuation apart from ! And ? Because they showcase some sentiments.
  - Upper case to lowercase was done previously but did not give better results, on not converting it was analyzed that training and test data contained uppercase words which needs to be treat differently from lowercase.
  - At the end we update the text column of train and test data.
  - These are the words which need to be redefined to make them meaningful and understandable. For example: “I’m” was replaced by “I am” or “☺” was replaced with smile.
  - This extraction of pre-processed data makes computation easy.

Vectorization

TF-IDF Vectorization was used, it is an abbreviation for Term Frequency Inverse Document Frequency. It is used to transform text into a meaningful representation of numbers which is used to fit machine algorithm for prediction.

Word vectorization and character vectorization has been done separately for more accurate results. After the TF-IDF matrices of word vectorization and character vectorization are made, then we merge them column wise using hstack function.

TFIDF model fitting is done on the train and test data concatenated together to achieve a better fit.

- **Word Embedding**
  The resulting words are then subjected to the embedding process, where they are converted into vectors of numbers. These vectors capture the grammatical function (*syntax*) and the meaning (*semantics*) of the words.

```
In [ ]: 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, 2),
            min_df = 2,
            max_df = 0.5,
            norm = 'l2',
            max_features = 30000
        )
        word_vectorizer.fit(complete_text)
        train_word_features = word_vectorizer.transform(train_text)
        test_word_features = word_vectorizer.transform(test_text)
```

HyperParameters:

1. **sublinear_tf: bool, default=False**
   Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

2. **strip_accents: *{'ascii', 'unicode'} or callable, default=None***
   Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have a direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.
3. **analyzer*{'word', 'char', 'char_wb'} or callable, default='word'***
   Whether the feature should be made of word or character n-grams
4. **token_pattern: *str, default=r"(?u)\b\w\w+\b"***
   Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`.
5. **Lowercase: *bool, default=True***
   Convert all characters to lowercase before tokenizing.
6. **stop_words: *{'english'}, list, default=None***
   If a string, it is passed to _check_stop_list and the appropriate stop list is returned.
7. **Ngram:_range*tuple (min_n, max_n), default=(1, 1)***
   The lower and upper boundary of the range of n-values for different n-grams to be extracted.
8. **Min_df/Max_df:**
   When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words).
9. **Norm: *{'l1', 'l2'} or None, default='l2'***
   Each output row will have unit norm, either:
   a. 'l2': Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied.
   b. l1': Sum of absolute values of vector elements is 1
10. **max_features*int, default=None***

    If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

- ## Character Embedding

The output of the character embedding step is similar to the output of the word embedding step.

```
In [ ]: char_vectorizer = TfidfVectorizer (
            sublinear_tf = True,
            strip_accents = 'unicode',
            analyzer = 'char',
            ngram_range = (2, 6),
            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 better fit.
        train_char_features = char_vectorizer.transform(train_text)
        test_char_features = char_vectorizer.transform(test_text)
```

Classification models experimented

- **ONE VS REST**
  - **Logistic Regression**
  - **Multinomial Naive Bayes**
  - **Random Forest**
  - **SVC**

In [36]:
```
def cv_tf_train_test(df_done,label,ngram):

    X = df_done.text
    y = df_done[label]
    X_test = test['text']
    cv1 = TfidfVectorizer(ngram_range=(ngram), stop_words='english')
    X_train_cv1 = cv1.fit_transform(X)
    X_test_cv1  = cv1.transform(X_test)
    uk_test_fit = cv1.transform(test['text'])
    lr = LogisticRegression()
    lr.fit(X_train_cv1, y)
    ypredlr = lr.predict(X_test_cv1)
    uk_prob = lr.predict_proba(uk_test_fit)
    mnb = MultinomialNB()
    mnb.fit(X_train_cv1, y)
    ypredmnb = mnb.predict(X_test_cv1)
    rfc = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 42)
    rfc.fit(X_train_cv1, y)
    ypredrfc = rfc.predict(X_test_cv1)
    svm_model =SVC()
    svm_model.fit(X_train_cv1,y)
    ypredsvm = svm_model.predict(X_test_cv1)

    return ypredlr, ypredmnb, uk_prob,ypredrfc,ypredsvm,svm_model
```

In [37]:
```
cols=['harsh', 'extremely_harsh', 'vulgar', 'threatening','disrespect', 'targeted_hate']
dones=[data_harsh_set,data_extremely_harsh_set,data_vulgar_set,data_threatening_set,data_disrespect_set,data_targeted_hate_set]
cvs=[]
probs={}
ypredlr = {}
ypredmnb = {}
ypredrfc = {}
ypredsvm = {}
model = {}
for i in range(len(cols)):
    print("***",cols[i],"***")
    ans=cv_tf_train_test(dones[i],cols[i], (1,1))
    #cvs.append(ans[0])
    ypredlr[cols[i]] = ans[0]
    ypredmnb[cols[i]] = ans[1]
    probs[cols[i]]=ans[2]
    ypredrfc[cols[i]]=ans[3]
    ypredsvm[cols[i]]=ans[4]
    model[cols[i]]=ans[5]
    #cvs[i].rename(columns={'F1 Score': 'F1 Score'}, inplace=True)


#cvs
```

```
*** harsh ***
*** extremely_harsh ***
*** vulgar ***
*** threatening ***
*** disrespect ***
*** targeted_hate ***
```

- **Chaining Classifier**
    - **Logistic Regression**
    - **Multinomial Naive Bayes**
    - **SVC**

In [ ]:
```
lr = ClassifierChain(LogisticRegression())
lr.fit(X_train_cv1, y)

ypredlr = lr.predict(X_test_cv1)
```

In [ ]:
```
mnb = ClassifierChain(MultinomialNB(),order='random', random_state=0)
mnb.fit(X_train_cv1, y)

ypredmnb = mnb.predict(X_test_cv1)
```

In [ ]:
```
svm_model = ClassifierChain(SVC(kernel= 'linear', random_state=1, C=0.1))
#clf = CalibratedClassifierCV(svm_model)
svm_model.fit(X_train_cv1,y)

ypredsvm = svm_model.predict(X_test_cv1)
```

- **Label PowerSet**
    - **SVC**
- **Ridge Classifier**

In [50]:
```
cross_validation_scores = []
model_storage = open('model_storage.pckl', 'wb')

for category in categories:
    train_target = train[category]
    ridgeClassifier = Ridge(solver = 'sag', max_iter = 50, fit_intercept = True, tol = 0.001, alpha = 70, copy_X = True, random_s
    cv_score = np.mean(cross_val_score(ridgeClassifier, train_features, train_target, cv = 50, scoring = 'roc_auc'))
    cross_validation_scores.append(cv_score)
    ridgeClassifier.fit(train_features, train_target)
    pickle.dump(ridgeClassifier, model_storage)

model_storage.close()
print('Cross Validation score is {}'.format(np.mean(cross_validation_scores)))
```

- **Logistic Regression CV**

In [51]:
```
cross_validation_scores = []
model_storage = open('model_storage_lr.pckl', 'wb')
from sklearn.linear_model import LogisticRegressionCV
lr = LogisticRegressionCV(solver = 'liblinear', n_jobs = -1)
for category in tqdm(categories):
    train_target = train[category]
    train_target_values = train_target.values
    lr = LogisticRegression(max_iter = 150, dual = False, C = 2)
    eec_lr = EasyEnsembleClassifier(base_estimator = LogisticRegression(solver = 'sag', max_iter = 750,n_jobs=-1, C = 2))
    cv_score = np.mean(cross_val_score(eec_lr, train_features, train_target, cv = 10, scoring = 'roc_auc'))
    cross_validation_scores.append(cv_score)
    eec_lr.fit(train_features, train_target)
    pickle.dump(eec_lr, model_storage)
```

```
100%|██████████| 6/6 [22:03<00:00, 220.59s/it]
```

Chaining Classifier	
Logistic Regression	0.7046
Multinomial Naïve Bayes	0.8299
Random Forest	0.8667
Support Vector Classifier	0.8966

Label Power Set	
Support Vector Classifier	0.6830

Ridge Classifier	
Ridge Classifier	0.9837

Logistic Regression CV	
Logistic Regression CV	0.92886