AI511 - Machine Learning Project Report

Project Name: Why So Harsh?

Team Name: 45AA

Team Members:

Adrij Sharma (IMT2019004)

Agrim Jain (IMT2019005)

PRE-PROCESSING

1. Lemmatization of text

Lemmatization of Training & Test Comments

```python
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

lemmatizer = WordNetLemmatizer()
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
stopWords = stopwords.words('english')

def lemmatize_text(text):
    return [lemmatizer.lemmatize(w) for w in w_tokenizer.tokenize(text)]

train_text = train["text"]
test_text = test["text"]
train_text = train_text.apply(lemmatize_text)
test_text = test_text.apply(lemmatize_text)
train_text = train_text.apply(lambda x: ' '.join(([word for word in x]))) # Join back the lemmztized words to form complete sentences.
test_text = test_text.apply(lambda x: ' '.join(([word for word in x])))
train["text"] = train_text
test["text"] = test_text
del train_text, test_text

# train_text = train_text.apply(lambda x: ' '.join(([word for word in x if word not in (stopWords)])))
# Instead of removing the stop words here, we take care of them during tokenization, vectorization of words and sentences.
```

✓ 35.8s

- The NLTK python library contains predefined functions for performing lemmatization or stemming on any text data.
- Lemmatization is the process of grouping together the different inflected forms of a word so they can be analyzed as a single item.
- Used in comprehensive retrieval systems like search engines.
- Examples of lemmatization:
  - rocks, stone, rock - rock
  - corpora - corpus
  - better, goodness, goodwill – good
  - finally, final, finalized – final
  - going, goes, gone – go

Lemmatization of training and test text is done at the same time.

Removal of stop word was not done before or after this step because they have been taken care of while vectorization using TF-IDF.

Why we prefer lemmatization over stemming?

Stemming is simpler as compared to lemmatization. With stemming, words are reduced to their word stems. A word stem need not be the same root as a dictionary-based morphological root, it just is an equal to or smaller form of the word. Examples: history, historical – histor. Lemmatization will group history and historical as a single group and the common name for that group will be "history" which is a sensible word.

2. Manual Text Clean-up

- Replace the words, symbols and emojis which are not making much sense with words which are very meaningful so that better vectorization results are obtained after applying TF-IDF. Here is a list of the nonsense words and symbols and the corresponding meaningful words they have been replaced with.

```
"i'm": "i am",
"m": "am",
"i'll" : "i will",
"its" : "it is",
"it's" : "it is",
"'s" : " is",
"that's" : "that is",
"weren't" : "were not",
"doesn't": "does not",
"didn't": "did not",
"hasn't": "has not",
":d": " smile ",
":p": " smile ",
":dd": " smile ",
"8)": " smile ",
":-)": " smile ",
":)": " smile ",
";)": " smile ",
"(-:": " smile ",
"(:": " smile ",
r"\br\b": "are",
r"\bu\b": "you",
r"\bhaha\b": "ha",
r"\bhahaha\b": "ha",
r"\bdon't\b": "do not",
r"\bdoesn't\b": "does not",
r"\bdidn't\b": "did not",
r"\bhasn't\b": "has not",
r"\bhaven't\b": "have not",
r"\bhadn't\b": "had not",
r"\bwon't\b": "will not",
r"\bwouldn't\b": "would not",
r"\bcan't\b": "can not",
r"\bcannot\b": "can not",
```

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

- Remove words like "http" or "www" from the comments which contain URLs. Only the relevant part of the URL which will contribute towards training the model are retained in the comment.
- Remove all numerical values from the training and test comments because they carry no significance.
- Remove all the punctuation marks apart from '!' and '?' because these are the only punctuation marks which showcase some sort of emotions.
- Do NOT convert the uppercase words into lowercase because the training and test data contains uppercase words which need to be treated differently from the lowercase words while preprocessing to obtain a well-trained model.
- At the end of all these steps, we update the "text" columns of our original training and test data frame itself so that the extraction of the pre-processed text for further computation is easy.

EMBEDDING & VECTORIZATION

- 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 horizontally (column wise) using **'hstack'** function.
- The TF-IDF model fitting is done on the train and test data concatenated together to achieve a better fit.

1. Definition of TF-IFD

TF-IDF stands for **term frequency–inverse document frequency**. It is a numerical statistic that is intended to reflect how important a word is to a document in a collection. The mathematics involved in computing the TF-IDF matrix has been explained briefly below:

- tf(t,d) = count of t in d / number of words in d
- df(t) = occurrence of t in documents
- df(t) = N(t)
  - where df(t) = Document frequency of a term t
  - N(t) = Number of documents containing the term t
- idf(t) = log(N/ df(t))
- tf-idf(t, d) = tf(t, d) * idf(t)

For a more detailed explanation regarding the mathematics involved in how TF-IDF measures the importance of words in a document, please refer to [this article].

2. Word Embedding – Vectorization of words using TF-IDF

Word Embedding - Vectorization of Words using TF-IDF (Analyser = 'word')

```python
word_vectorizer = TfidfVectorizer(
    sublinear_tf = True,  # It seems unlikely that twenty occurrences of a term in a document truly carry twenty times the significance of a single occurrence.
    # Accordingly, there has been considerable research into variants of term frequency that go beyond counting the number of occurrences of a term.
    # A common modification is to use instead the logarithm of the term frequency, which assigns a weight. 1 + log(tf)
    strip_accents = 'unicode', # Remove accents and perform other character normalization during the preprocessing step.
                               # ‘ascii’ is a fast method that only works on characters that have an direct ASCII mapping.
                               # ‘unicode’ is a slightly slower method that works on any characters.
    analyzer = 'word',  # Whether the feature should be made of word or character.
    token_pattern = '(?u)\\b\\w\\w+\\b\\w{,1}',
    lowercase = False,  # Do not convert the uppercase letters into lowercase because they carry significance.
    stop_words = 'english',  # Remove all the stop words of english
    # ngram is the set of n words together.
    ngram_range = (1, 2),  # We consider set of 1 or 2 words together for tokenization.
    min_df = 2,
    max_df = 0.5,
    norm = 'l2',
    max_features = 30000
) #lowercase = true : convert all characters into lower case before tokenzing
word_vectorizer.fit(all_text) # Apply tfidf fitting on the whole preprocessed text data so that we schieve a better fitted model.
train_word_features = word_vectorizer.transform(train_text)
test_word_features = word_vectorizer.transform(test_text)
```

✓ 35.5s

Hyperparameters:

➢ sublinear_tf = true

- It is unlikely that twenty occurrences of a term in a document carry twenty times the significance of a single occurrence. A modification to deal with this issue is to use the logarithm of the term frequency, which assigns a weight.
- tf <=> 1 + log(tf)

➢ ngram_range = (1,2)

- ngram is the set of n words together. Range (1,2) signifies that we take single words while tokenization or we take asset of 2 words together.
- The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used. For example, an ngram_range of (1,1) means only unigrams, (1,2) means unigrams and bigrams, and (2,2)means only bigrams. Only applies if analyser is not callable.

➢ lowercase = False

- Do not convert the uppercase letters into lowercase because they uppercase words carry significance.

➢ stop_words = ‘english’

- We intend to remove all the stop words of the English language which have been pre-defined in the NLTK library.

- Remember that we did not do this during the text pre processing step because a better accuracy is obtained if stop words are ignored during the vectorization step.

➢ token_pattern = ‘(?u)\\b\\w\\w+\\b\\w{,1}’

- Regular expression denoting what constitutes a “token”, only used if analyser = ‘word’. The default regexp selects tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).
- If there is a capturing group in token_pattern then the captured group content, not the entire match, becomes the token. At most one capturing group is permitted.

➢ norm = l2

Each output row will have unit norm, i.e. the sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied.

➢ max_features = 30000

The number of vocabulary words in the tf-idf matrix ⇔ The number of columns of the tf-idf matrix ⇔ 30000.

3. Character Embedding – Vectorization of characters using TF-IDF

Character Embedding - Vectorization of Individual Characters using TF-IDF (Analyser = 'char')

```
char_vectorizer = TfidfVectorizer (
    sublinear_tf = True,
    strip_accents = 'unicode', # Remove accents and perform other character normalization during the preprocessing step.
                               # ‘ascii’ is a fast method that only works on characters that have an direct ASCII mapping.
                               # ‘unicode’ is a slightly slower method that works on any characters.
    analyzer = 'char',
    ngram_range = (2, 6),  # ngram is the set of n words together.
    min_df = 2,
    max_df = 0.5,
    max_features = 20000
)
char_vectorizer.fit(all_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)
```

✓ 5m 57.2s MagicPython

Hyperparameters:

➢ If the analyser = ‘char’, then stop_words hyperparameter holds no significance and this is pretty logical.

➢ sublinear_tf = True

➢ ngram_range = (2, 6)

➢ strip_accents = 'unicode’

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 but better accuracy is observed.

CLASSIFICATION MODELS

Role of Pickle File

- A pickle file is used to store a trained model.
- The file format for a pickle file is **".pckl"**.
- The trained model can be loaded from this file to make predictions without re-training.
- The use of pickle file saves training time.
- If we use a pickle file, our code becomes more efficient and easier to debug.
- The application of pickle file is to serialize your machine learning algorithms and save the serialized format.

Classification Models Tried

Here is a list of all the classification models which we tried to get the best possible accuracy. They have been ranked according to the accuracy of the results which they produced.

1. Voting Classifier (trains on 4 different classification models) – Slow
2. Ridge Classifier – Fast
3. Random Forest Classifier – Slow
4. Logistic Regression CV – Fast
5. Gaussian Naïve Bayes Classifier – Fast

Model Explanations

1. Voting Classifier

```
cross_validation_scores = []
ratio_values = []
model_storage = open('model_storage.pckl', 'wb') # Pickle file for storing the trained models.

# A Voting Classifier is a machine learning model that trains on an ensemble of numerous models:
# 1. Logistic Regression.
# 2. Random Forest Classifier.
# 3. Easy Ensemble Classifier with logistic regression as the base eliminator.
# 4. Easy Ensemble Classifier with SGDC Classifier as the base eliminator.
# The voting classifier predicts an output (class) based on the highest probability of the varioes models of chosen class as the output.

for category in categories:
    train_target = train[category]
    train_target_values = train_target.values
    r = np.log(pr(1, train_target_values) / pr(0, train_target_values))
    x_nb = train_features.multiply(r)
    rfc = RandomForestClassifier(max_features = 999, max_depth = 100, min_samples_split = 10, criterion = 'gini', n_estimators = 120, min_weight_fraction_leaf = 0.0, max_leaf_nodes = None)
    lr = LogisticRegression(max_iter = 499, dual = False, C = 2)
    eec_lr = EasyEnsembleClassifier(base_estimator = LogisticRegression(solver = 'sag', max_iter = 450, C = 2)) # sag solver proved to be faster and more accurate than liblinear solver and the default solver.
    eec_sgdc = EasyEnsembleClassifier(base_estimator = SGDClassifier(max_iter = 165, alpha = 0.0002, penalty = "l2", loss = 'modified_huber'))
    vote = VotingClassifier(voting = 'soft', estimators = [('lr', eec_lr), ('sgd', eec_sgdc), ('lr1', lr), ('rdf', rfc)], weights = [0.9, 1.35, 0.65, 0.8])
    vote.fit(x_nb, train_target_values)
    ratio_values.append(r)
    pickle.dump(vote, model_storage)  # Dump the model into the pickle file after fitting it on the training data.
    cv_score = np.mean(cross_val_score(vote, x_nb, train_target, cv = 5, scoring = 'roc_auc'))
    cross_validation_scores.append(cv_score)
    #print('Cross Validation score for class {} is {}'.format(category, cv_score))

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

- A Voting Classifier that trains on an ensemble of numerous models and predicts an output (class) based on their highest probability of chosen class as the output.

- It aggregates the findings of each classifier passed into Voting Classifier and predicts the output class based on the highest majority of voting.
- The idea is instead of creating separate dedicated models and finding the accuracy for each them, we create a single model which trains by these models and predicts output based on their combined majority of voting for each output class.
- The model which stood as contender in our plan are:
  - Logistic Regression
    Hyperparameters: default solver, C = 2, dual = false, max iterations = 450
  - Random Forest Classifier
    Hyperparameters: maximum features = 1000, maximum depth of any decision tree = 100, minimum samples split = 10, criterion = gini.
  - Easy Ensemble Classifier (with Logistic regression as the base eliminator)
    The Logistic Regression model used as the base eliminator operates on 'sag' solver. 'sag' gave more accurate results than 'liblinear' or 'lbfgs'.
  - Easy Ensemble Classifier (with SGDC classifier as the base eliminator).
- Difference between 'hard' voting and 'soft' voting:
  - Hard – The predicted output class is a class with the highest majority of votes i.e. the class which had the highest probability of being predicted by each of the classifiers. Suppose three classifiers predicted the output class *(A, A, B)*, so here the majority predicted *A* as output. Hence *A* will be the final prediction.
  - Soft - In soft voting, the output class is the prediction based on the average of probability given to that class. Suppose given some input to three models, the prediction probability for class *A* = *(0.30, 0.47, 0.53)* and *B* = *(0.20, 0.32, 0.40)*. So, the average for class *A is 0.4333* and *B is 0.3067*, the winner is clearly class *A* because it had the highest probability averaged by each classifier.
- Once the model has been trained and fitted, just dump the model into the pickle file so that you can use the trained model later on.
- One of the models which the voting classifier uses for training is the Random Forest Classification which takes quite some time (about 250 minutes) for getting fit on the training data.
- For a more detailed explanation about the working of the Voting Classifier, along with other examples, please refer to [this article].

2. Ridge Classifier

2. Ridge Classification Model

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

for category in categories:
    train_target = train[category]
    ridgeClassifier = Ridge(solver = 'sag', max_iter = 150, fit_intercept = True, tol = 0.0025, alpha = 29, copy_X = True, random_state = 0)
    cv_score = np.mean(cross_val_score(ridgeClassifier, train_features, train_target, cv = 3, scoring = 'roc_auc'))
    cross_validation_scores.append(cv_score)
    #print('Cross Validation score for class {} is {}'.format(category, 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)))
```

```
Cross Validation score is 0.9834960337914218
```

Hyperparameters:

- solver = 'sag'
  'sag' stands for Stochastic Average Gradient descent. It is an iterative procedure and is faster than other solvers when both n_samples and n_features are large. 'sag' solver's fast convergence is only guaranteed on features with approximately the same scale.
- alpha = 29 or 27
  This denotes the Regularization strength. This must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization.
- copy_X = True
  If true, X will be copied, otherwise, it may be overwritten.
- fit_intercept = True (default)
  Whether to calculate the intercept for this model. If fit_intercept is set to false, no intercept will be used in the computations. In that case, the data is expected to be pre-centered.
- max_iter = 150
  Maximum number of iterations for the gradient solver.
- random_state = 0
  Used to shuffle the data
- tol = 0.0025
  A measure of the precision of the final output.

A noticeable difference between the Ridge Classifier and all the other classifiers that have been taken into use during the course of this project is that the predictions on the test data are made using the **.predict(test features)** function, whereas, in all the other classifiers the predictions are made using **.predict_proba(test features)** function.

3. <u>Random Forest Classifier</u>

3. Random Forest Classification Model

```
cross_validation_scores = []
result = pd.DataFrame.from_dict({'id': test['id']})
model_storage = open('model_storage.pckl', 'wb')

for category in categories:
    train_target = train[category]
    rfc = RandomForestClassifier(max_features = 999, max_depth = 100, min_samples_split = 10, criterion = 'gini', n_estimators = 120, min_weight_fraction_leaf = 0.0, max_leaf_nodes = None)
    cv_score = np.mean(cross_val_score(rfc, train_features, train_target, cv=3, scoring='roc_auc'))
    cross_validation_scores.append(cv_score)
    #print('CV score for class {} is {}'.format(category, cv_score))
    rfc.fit(train_features, train_target)
    pickle.dump(rfc, model_storage)

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

Hyperparameters:

- Criterion = 'gini'
  The function to measure the quality of a split. This parameter is a tree-specific parameter.

- max_depth = 100

  The maximum depth of the decision tree. If this value is not specified, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. We tried leaving this none in our code initially. In that case we did not get an output even after 1000 minutes of code execution.
- max_features = 1000

  The function to measure the quality of a split.
- max_leaf_nodes = None

  Grow trees with max_leaf_nodes in best-first fashion.
- min_samples_split = 10

  The minimum number of samples required to split an interval node of the decision trees.
- min_weight_fraction_leaf = 0.0

  The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node.
- n_estimators = 120

  The number of trees in the forest. The default value is 100, but, for better accuracy we have increased the count by 20 trees.

RFC takes a lot of time for training (about 200 minutes). For a more detailed explanation about the Random Forest classifier, please refer to [this article].

4. Logistic Regression CV (along with trials of under sampling & over sampling)

4. Logistic Regression CV along with Sampling

```python
from imblearn.under_sampling import NearMiss  # Under Sampling
nm = NearMiss(random_state = 100)

# from imblearn.over_sampling import SMOTE  # Over Sampling
# smk = SMOTE(random_state = 12)

from sklearn.linear_model import LogisticRegressionCV
lr = LogisticRegressionCV(n_jobs = -1) # liblinear solver proved to be the best

for category in categories:
    Y_train = train_df[category].to_numpy().astype(np.float64)
    X_train_balanced, Y_train_balanced = nm.fit_resample(X_train, Y_train)
    lr.fit(X_train_balanced, Y_train_balanced)
    pickle.dump(lr, model_storage)

model_storage.close()
```

- Handling the imbalanced classes in the training data set using **under sampling**
  - imblearn module was used for performing under sampling.
  - Near - Miss Under Sampling: This algorithm looks at the class distribution and randomly eliminates samples from the larger class.

Hyperparameters of Logistic Regression CV model:

- solver = 'liblinear'

  This represents the algorithm for optimization. For larger datasets, 'sag' and 'saga' solvers are better.
- Tol = 0.0001

  Tolerance for stopping criteria.
- max_iter = 150

  Maximum number of iterations of the optimization algorithm.
- cv = cross-validation generator

  The default cross-validation generator used is Stratified K-Folds. If an integer is provided, then it is the number of folds used. The best hyperparameter is selected by the cross-validator StratifiedKFold, but it can be changed using the cv parameter.
- Stratified K Fold:
  - The number of instances of each class for each experiment in the train and test data are taken in a methodical way.
  - The folds are selected so that the mean response value is approximately equal in all the folds. In the case of a binary classification, each fold contains roughly the same proportions of the two types of class labels i.e. 1/0 or YES/NO.

PYTHON LIBRARIES

- Numpy
- Pandas
- NLTK
- Time
- Regex
- Pickle
- Sklearn
- Spipy
- Imblearn

RESOURCES

Documentations

- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html

Articles

- https://www.geeksforgeeks.org/ml-voting-classifier-using-sklearn/
- https://www.geeksforgeeks.org/random-forest-regression-in-python/
- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- https://towardsdatascience.com/besides-word-embedding-why-you-need-to-know-character-embedding-6096a34a3b10
- https://towardsdatascience.com/stemming-vs-lemmatization-2daddabcb221#:~:text=Stemming%20and%20Lemmatization%20both%20generate,words%20which%20makes%20it%20faster.

Video lectures

- https://www.youtube.com/playlist?list=PLQVvvaa0QuDf2JswnfiGkliBInZnIC4HL
- https://www.youtube.com/watch?v=fM4qTMfCoak
- https://www.youtube.com/watch?v=VOpETRQGXy0