Yogesh Shelgaonkar 1005467

Kevin Richan 1005605

Hung Chia Yu 1005603

Wee Neville 1005035

Victoria Chong 1005528

[Mother Language]

Machine Learning Project Task 4

1. An introduction of your best performing model (how it works)

Data preparation:

Our group looked through the dataset provided and tried to capture the intuition behind how to classify the twitter post as hateful or non-hateful as a human. We realised certain trends in the dataset, such as hateful speech often contains offensive words like “f*ck”, “n*gger” or “black guys”. Essentially, this type of trend is what the model tries to learn and hence we made an assumption that TF-IDF features may not be very suitable for this task. The reason is because TF-IDF penalises words that occur frequently throughout the corpus and assign those words with a low TF-IDF score. This means that most of those offensive words that suggest that the post is hateful are being assigned a low TF-IDF score. The TF-IDF scores of these words might not appear in the top 5000 TF-IDF features given to us. To test this, we run a simple XGboost model with the 5000 TF-IDF features without hyperparameter tuning, and much to our expectation, the macro F1-score achieved is about 0.68 which is a relatively low score. Hence, we proposed to use TF features instead. We ran the XGboost model on TF features without hyperparameter tuning and we realised the macro F1-score is about 0.70. This shows that TF-features are actually a better representation of the dataset. To push it further, we include bi-grams in our TF-features, attempting to capture sarcasm or phrases that give strong indicators to the tweet being hateful. An example of such phrases is “black guys”.

```
## Create countVectorizer for creating TF features

count_vec = CountVectorizer(
    tokenizer=word_tokenize,
    strip_accents="ascii", lowercase=True,
    token_pattern=None,ngram_range=(1,2)) ## Create TF features instead of TF-IDF features with unigrams and bigram
count_vec.fit(df.stemmed_post)
```

In order to reduce the number of unique vocabulary in the dataset, we attempt to stem the post using the nltk PorterStemmer. This will allow the TF-features to have less unique vocabulary and lower the chance of naive bayes underfitting.

```
from nltk.stem import PorterStemmer

ps = PorterStemmer()
```

```
def stem_post(row):
    ## Stem all words in the post
    words = row.split()
    result = ""
    for word in words:
        result+=ps.stem(word)+' '
    return result[:-1]
```

Model selection:

1st model: Logistic regression: Maked use of the sigmoid function to output an estimated probability.

```
def logistic_regression(train_df):
    """ Train logistic regression model given the stemmed twitter post.
        Return: trained model and fitted CountVectorizer
    """
    count_vec = CountVectorizer(
    tokenizer=word_tokenize,
    strip_accents="ascii", lowercase=True,
    token_pattern=None,ngram_range=(1,2))
    count_vec.fit(train_df.stemmed_post)
    model = BaggingClassifier(base_estimator=linear_model.LogisticRegression(max_iter=200),
            n_estimators=10, random_state=0)
# fit the model on training data reviews and sentiment
    xtrain = count_vec.transform(train_df['stemmed_post'])

    model.fit(xtrain, train_df.label)

    return model,count_vec
```

We start with a simple model which is logistic regression on the TF-features of unigrams and bigrams and we achieved a macro F1-score of 0.71. (Surprisingly high)

2nd model: Naive Bayes: Classification Technique based on Bayes Theorem with an assumption of independence between the features.Easy to build and fast for large datasets, using calculation of posterior probability and good with categorical data.

```
## Naive bayes method
def naive_b(train_df):

    """ Train naive bayes model given the stemmed twitter post.
        Return: trained model and fitted CountVectorizer """
    count_vec = CountVectorizer(
    tokenizer=word_tokenize,
    strip_accents="ascii", lowercase=True,
    token_pattern=None)
    count_vec.fit(train_df.stemmed_post)
    clf = BaggingClassifier(base_estimator=naive_bayes.MultinomialNB(),
            n_estimators=10, random_state=0)
    x_train = count_vec.transform(train_df['stemmed_post'])

    clf.fit(x_train,train_df['label'])

    return clf,count_vec
```

We trained naive bayes on TF-features of unigrams. Using bigram here will cause the naive bayes model to underfit due to the small size of the dataset. The macro F1-score of using unigram+bigram is 0.60 and the macro F1-score of using unigram only is 0.71.

3rd model: XGboost: Gradient boosted decision tree machine learning library.Iteratively trains an ensemble of shallow decision trees, weighted sum of all the trees.

```
## train xgb given the train_df,test_df and hyperparameters
def xgbtrain(train_df,test_df,param,count_vec):


    xtrain = count_vec.transform(train_df['stemmed_post'])
    xtest = count_vec.transform(test_df['stemmed_post'])

    dtrain = xgb.DMatrix(xtrain,label=train_df['label'].to_numpy())
    dtest = xgb.DMatrix(xtest,label=test_df['label'].to_numpy())

    evallist = [(dtest, 'eval'), (dtrain, 'train')]
    param['eval_metric']= ['auc']
    num_round = param["n_round"]
    bst = xgb.train(param, dtrain, num_round, evallist,custom_metric=f1_eval_mac)

    ## Score is the f1 score of this model evaluated on the given test set
    score = f1_score(np.round(bst.predict(dtest)),test_df['label'],average='macro')


    return bst,score
```

```
##  K fold cross validation on xgb for hyperparameter tuning
def k_fold_xgb(df,param,n_fold=10):
    skf = StratifiedKFold(n_splits=n_fold, random_state=1, shuffle=True)
    f1_score_list = []
    Y = df['label']
    bst_list = []
    for train_index,test_index in skf.split(df,Y):
        train_df,test_df = df.loc[train_index],df.loc[test_index] ## Get the train test df on current split
        bst,score = xgbtrain(train_df,test_df,param,count_vec)## train it on xgboost
        bst_list.append(bst)
        f1_score_list.append(score)

    return bst_list,np.average(f1_score_list)
```

We trained XGboost on the TF-features of unigram and bigram and attempted to do a hyperparameter search through grid search. A problem we notice is that it is very easy to find a set of hyperparameters that will overfit the test set. To overcome this problem, we used K fold cross validation with 10 folds total. We trained 10 XGB models with a single set of parameters across the 10 fold and averaged the macro f1-score. The set of hyperparameters that gave the best average f1-score is the set of hyperparameters we chose in the end. The macro F1-score here is 0.72.

Final model: Ensemble model. Use log reg + xgboost + naive bayes.

In order to get an even better model, we decided to stack these 3 models together by simply taking the average of the predictions made by each model. This can be further improved by weighting the predictions of the models, however the weights would require tuning.This showed improvements on the public leaderboard, thus it ended up as our final model. However, it turns out that the xgboost model alone did the best on the private leaderboard.

2. How did you "tune" the model? Discuss the parameters that you have used and the different parameters that you have tried before arriving at the best results.

In order to tune the model, we tune the hyperparameter using grid search. We use wandb to help us to log the macro F1-score and select the hyperparameter that has the highest average f1 score across the 10 folds. Wandb is an experiment tracking tool for machine learning

3. Did you self-learned anything that is beyond the course? If yes, what are they, and do you think it should be taught in future Machine Learning courses.

Yes. We learnt that tree based methods are usually more suitable for tabular datasets such as TF-IDF features or TF features. We also learnt that what was given to us might not be the best or the most ideal for our solution and we should always ponder and explore other options. For instance, using our own TF instead of the TF-IDF provided for reasons that we provided in the final report. We also learnt that in hyperparameter tuning, it is very possible

to overfit the validation set, hence it is better to do k-fold cross validation when tuning the hyperparameter. The trade off here is that in k-fold cross validation, more models need to be trained which increase the train-test loop iteration speed and we might not be able to test out all of our ideas if we perform k-fold cross validation. We believe machine learning courses should cover basics of developing machine learning models such as starting with a simple model, creating a small train-test loop and slowly try out different ideas instead of starting to train a 300h machine learning model that ends up having the same performance as a model that takes 5 min to train. To test out a new model, it is always important to try and overfit on a small train sample to achieve 100% accuracy to prove that this problem is learnable by this machine learning model. Then we can explore the variation of this model on this dataset.We also believe in being efficient in your model training, testing and choice. Whilst extensive hyperparameter tuning or experimenting with complicated models can be effective at improving the model, it also takes a lot of time and computing power to run and may not be worth the time to attempt at length. Being students with many commitments and other projects as well, we made sure to keep our model relatively simple and be efficient with the models we tried. Biggest takeaway of this project: Do not start by training a 300 h model. XD.