50.007 Machine Learning Project

Summer 2022

Report (Task 4)

Skynet

Lim Fuo En 1005125

Lee Wai Shun 1005115

Rohit Raghuram Murarishetti 1005398

Mohammed Fauzaan 1005404

Jyotit Kaushal 1005245

Table of Contents

Task 2

Principal Component Analysis was carried out in two distinct ways at first:

1. Without minmax scaling: The rationale behind this was that the features were already in the range [0,1], due to tf-idf conversion. So we did not find it necessary to further scale the models
2. With Minmax scaling: Doing this converts the features such that they are in the range (0,1), which we expected to give different results

Not surprisingly, both the variations produced the same results for the prescribed number of features, i.e., 100, 500, 1000 and 2000. The following were the best parameters:

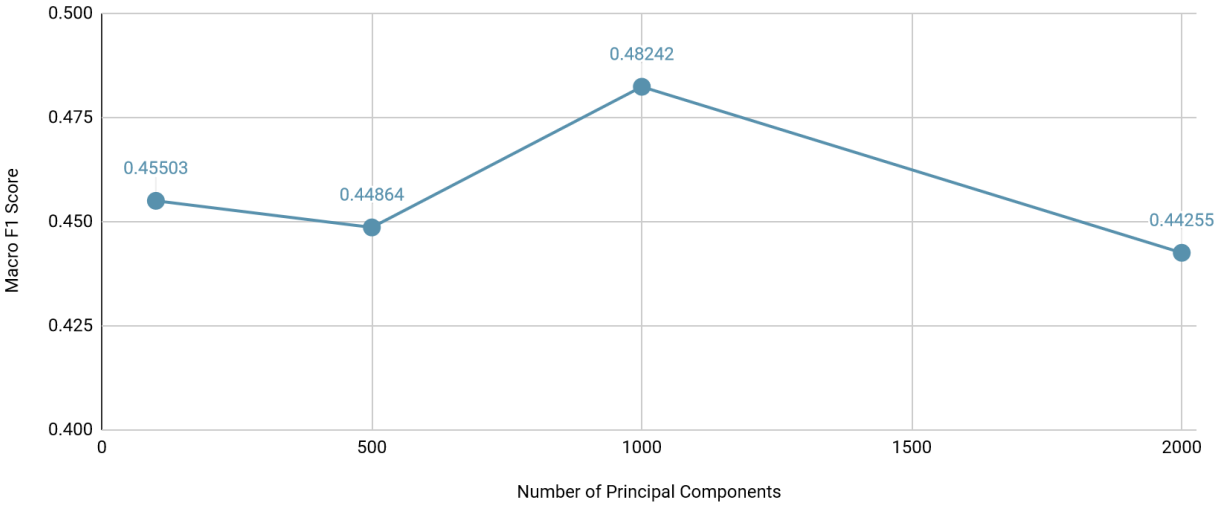
2000: `{'algorithm': 'kd_tree', 'weights': 'distance'}`
1000: `{'algorithm': 'ball_tree', 'weights': 'distance'}`
500: `{'algorithm': 'kd_tree', 'weights': 'distance'}`
100: `{'algorithm': 'auto', 'weights': 'distance'}`

We also ran gridsearch CV before fitting the KNN model as we wanted to determine the best possible algorithm. We

Using the results derived from the cross validation, we ran the KNN model and retrieved the following Macro F1 Scores by submitting predictions that were predicted by a K-Nearest Neighbors model on reduced datasets with a varied number of principal components through Principal Component Analysis (PCA).

The results are as follows:

Number of Principal Components	100	500	1000	2000
Macro F1 Score	0.45503	0.44864	0.48242	0.44255



Task 3

Models

The scores displayed below are the best scores for each model.

Best Scores for each model in the public leaderboard

- Extra Trees (Best Macro F1 Score: 0.72252)
- Support Vector Machine (Best Macro F1 Score: 0.71254)
- AdaBoost (Best Macro F1 Score: 0.67938)
- TPot Decision Tree (Best Macro F1 Score: 0.58004)
- Random Forest (Best Macro F1 Score: 0.48064)

Other Best Scores for each model in the private leaderboard

- Support Vector Machine (Best Macro F1 Score: 0.70554)
- Extra Trees Classifier (Best Macro F1 Score: 0.70299)
- AdaBoost (Best Macro F1 Score: 0.66695)
- TPot Decision Tree (Best Macro F1 Score: 0.58315)

Other models tried without retrieving F1 macro score

- Multinomial Naive Bayes
- KMeans
- Minka's MLE PCA reduction, then applied on all the models.

Support Vector Machine (Best Private Score)

Introduction of Model

In this project, the Support Vector Machine (SVM) model is used as a classification model whose decision boundary (the yellow line in **Figure 1.1**) ensures maximum distance between the boundary itself and the nearest training data point on either side of the decision boundary. This distance is known as the margin (γ) of the training dataset.

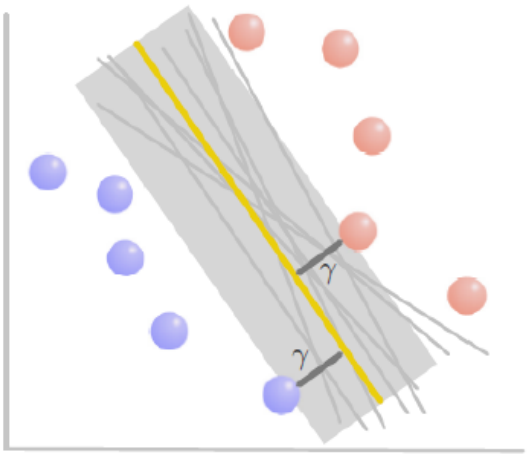


Figure 1.1 (taken from Week 4's Lecture Slides)

The motivation behind finding the decision boundary that maximizes the margin (of the training data set) is such that the model can generalize better, because in general, the larger the margin, the lower the generalization error of the model.

Even if the training dataset is not linearly separable, this model allows for an option to either allow some misclassification of training data points (**Figure 1.2**), or to map the training data points to a higher dimensional feature space (**Figure 1.3**), in order to be able to fit a decision boundary that is a hyperplane of the higher dimensional feature space with the maximum margin.

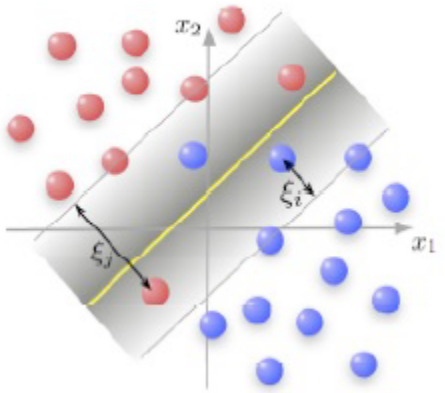


Figure 1.2 (taken from Week 4's Lecture Slides)

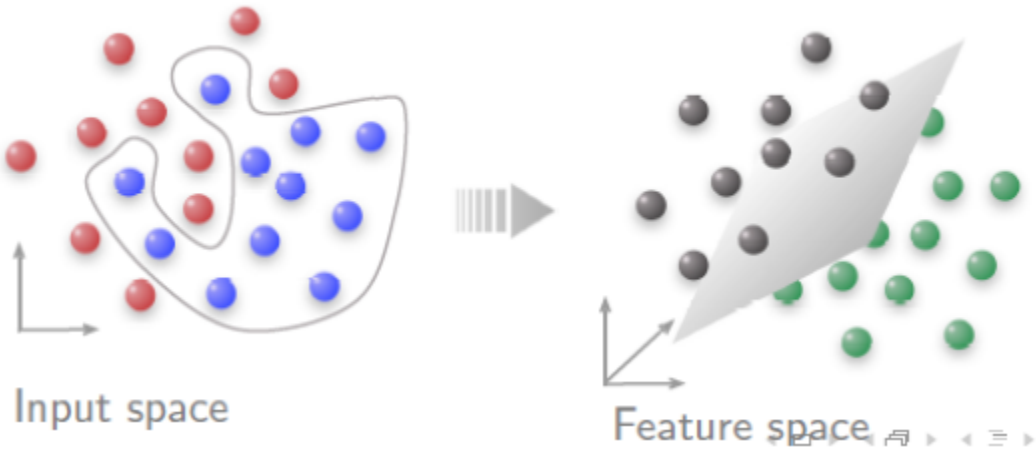


Figure 1.3 (taken from Week 4's Lecture Slides)

Hyper-parameter Tuning

Principal Component Analysis

We started off by performing PCA on the training dataset such that the reduced training dataset contains principal components that accounts for 95% variance of the original training dataset. 95% was initially chosen as it accounted for a high degree of variance. The goal was twofold, PCA could help us reduce the number of features that we use for training while also preventing overfitting to the training dataset by making the removing less significant and potentially extraneous variations in the dataset.

```
train_set_label = df_train.loc[:, ["label"]]
features_names = [str(i) for i in range(0, 5000)] # since there are 5000 features
train_set_features = df_train.loc[:, features_names] # train_set_features will contain just the feature columns of the original training set

# perform PCA
pca = PCA(n_components = 0.95) # 95% variance for PCA
train_set_reduced = pca.fit_transform(train_set_features) # fits the PCA to the train_set, then transforms train_set into a dimensionally reduced set
train_set_reduced = pd.DataFrame(data = train_set_reduced) # convert the train_set_reduced into a pandas dataframe
```

Figure 1.4

We then applied that same PCA that we did on the training dataset to the testing dataset. This is to ensure that the PCA performed is consistent between the training and testing datasets.

```
features_names = [str(i) for i in range(0, 5000)] # since there are 5000 features
test_set_features = df_test.loc[:, features_names] # test_set_features will contain just the feature columns of the original testing set
test_set_reduced = pca.transform(test_set_features) # uses PCA from the train_set to reduce the testing set
test_set_features = pd.DataFrame(data = test_set_reduced)
```

Figure 1.5

Model Parameters

Using sklearn's GridSearchCV, we ran a grid search to find the hyper-parameters for the SVM (*C* and *gamma*). Providing the SVM with a list of potential values for each hyperparameter, combinations of these parameters were tested to generate the best model. The values are as shown:

- *C* : [0.1, 1, 10, 1000]
- *gamma* : [1, 0.1, 0.01, 0.001]
- *Kernel* : Radial Basis Function ('rbf')

C is a hyperparameter that determines the amount of penalty/misclassification that is allowed for the SVM. A lower value of *C* means there are less misclassifications of the training data points allowed and a smaller margin hyperplane when fitting the SVM, while a higher value of *C* means there are more misclassifications of the training data points allowed and a larger margin hyperplane.

gamma is a hyperparameter that determines how much the SVM is to be fitted to the training data points. A higher value of *gamma* means that the SVM will fit the training dataset more, which may lead to overfitting. A lower value of *gamma* means that the SVM will more loosely fit the training dataset, which may either lead to underfitting, or better generalization.

We only ran GridSearchCV with the Radial Basis Function ('rbf') kernel as it is a popular kernel that works well in practice and is relatively easy to adjust, and to minimize computation time of the tuning of the hyper-parameters (i.e. other kernel functions) as GridSearchCV is a rather time-consuming process.

We ran the GridSearchCV with the above-mentioned list of values of *C* and *gamma* as they provide a good range of values for the hyperparameters, and there are only 4 values each for *C*

and *gamma*, which will reduce computation time for the GridSearchCV. Due to the wide range covered by the list of *C* and *gamma* values, we are confident that the GridSearchCV will cover a wide range of *C* and *gamma* values sufficiently and find the best one.

'f1_macro' was chosen as the scoring parameter for GridSearch as the Kaggle competition's scoring is based on Macro F1 score. 'f1_macro' was also chosen for the refit parameter so that GridSearchCV refits the SVM using the hyper-parameters that resulted in the best Macro F1 score on the entire training dataset.

Once all these parameters are set, we simply run the GridSearchCV for it to find the best hyper-parameters to train the SVM using the hyper-parameters (*C* and *gamma*) that were found to result in the highest Macro F1 score.

Cross Validation

To prevent overfitting, we needed to use some sort of cross validation. A common technique is K-folds - splitting our dataset into random groups, holding one group out as the test, and training the model on the remaining groups. This process is repeated for each group being held as the test group, then the average of the models is used for the resulting model.

Since the number of words labeled hate speech as compared to non-hate speech is relatively low, K-folds might run into the case where the sampled dataset contains a disproportionate number of non-hate speech. Similar to the idea that using F1 score instead of accuracy, the use of K-folds might introduce a bias towards the non-hate speech classification Hence, stratified k-folds are used. Stratified K-folds deal with classification tasks with imbalanced class distributions by ensuring that each fold of the dataset has the same proportion of observations with a given label.

For example, given 9 data points and imbalanced class distribution. In the dataset, 6 of the data points belong to +1 and the rest (i.e. 3) belong to -1. The ratio of class +1 to class -1 is 1/3. So each sample (both training and test dataset) will contain the class +1 and -1 in this ratio. This preserves the class distribution in the splits by StratifiedKFold while KFold does not take this into consideration. Looking at the Stratified K-fold parameters, n_split parameter refers to the number of different validation (and training) sets you will create from the given dataset. Repeated Stratified K-folds method was also tested but took too long and did not produce significant improvements in performance.

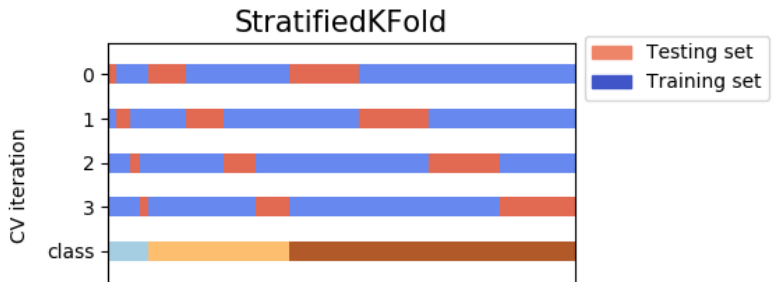


Figure 1.6

For the purposes of this exercise, we used Stratified K-folds using n-splits =3 and shuffle = true. With reference to **Figure 1.7** below, k = 3 is used as 3 is the minimum allowed value for k, and to minimize the number of fits of the SVM on the k - 1 subsets of the training datasets, to minimize computation time of GridSearchCV.

```
# finding the optimum hyper-parameters for the SVM
hyper_parameters = {'C': [0.1, 1, 10, 100], 'gamma': [1, 0.1, 0.01, 0.001], 'kernel': ['rbf']} # initialise the hyper-parameters
kfold = StratifiedKFold(n_splits = 3, shuffle = True, random_state = 0) # for 3-fold cross validation when finding the optimum hyper-parameters

# find the optimum hyper-parameters evaluated based on Macro F1 score
grid = GridSearchCV(SVC(), param_grid = hyper_parameters, scoring = 'f1_macro', refit = 'f1_macro', n_jobs = 1 , cv = kfold, verbose = 2)
grid.fit(X_train, np.ravel(y_train)) # training the model on the training set (X_train for features and y_train for labels) using the best hyper-parameters from the GridSearchCV
```

Figure 1.7

Results and Fine-tuning

For PCA with 95% variance, the best hyper-parameters were found to be ***C* = 10** and ***gamma* = 1**. Once this is done, the grid from the GridSearchCV is used to fit to the training dataset, which is then used to predict the test dataset.

We did a number of runs of the GridSearchCV with various factors, like various PCA variance percentages. For some of the runs, we also performed PCA on the combined dataset (train dataset combined with test dataset), and used Min-Max Scaling on the datasets, before feeding them to the GridSearchCV. We also varied other factors like using RandomizedSearchCV instead of GridSearchCV, and changing the 'scoring' and 'refit' parameters.

A summary of some of our best initial runs with SVM are shown in **Figure 1.8** below.

PCA % variance	Scaled?	Combined?	Kernel	Tuning	Scoring	Refit	Score
95	No	No	RBF	Grid Search CV	F1 Macro	F1 Macro	71.172
94	No	No	RBF	Grid Search CV	F1 Macro	F1 Macro	71.044
96	No	No	RBF	Grid Search CV	F1 Macro	F1 Macro	70.82
93	No	No	RBF	Grid Search CV	F1 Macro	F1 Macro	70.23
99	No	No	RBF	Grid Search CV	F1 Macro	F1 Macro	70.16

Figure 1.8

We plotted the results as points on a grid in an attempt to reveal a possible and predictable relationship between the PCA variance and the score on Kaggle, while keeping all other factors constant. The plot in **Figure 1.9** below illustrates that, where the *x* value of a point *(x, y)* is the PCA variance, while the *y* value is the corresponding score on Kaggle (multiplied by 100). We would also like to point out that the points (runs) below are on datasets that are unscaled (prior to the PCA) and uncombined (for the PCA).

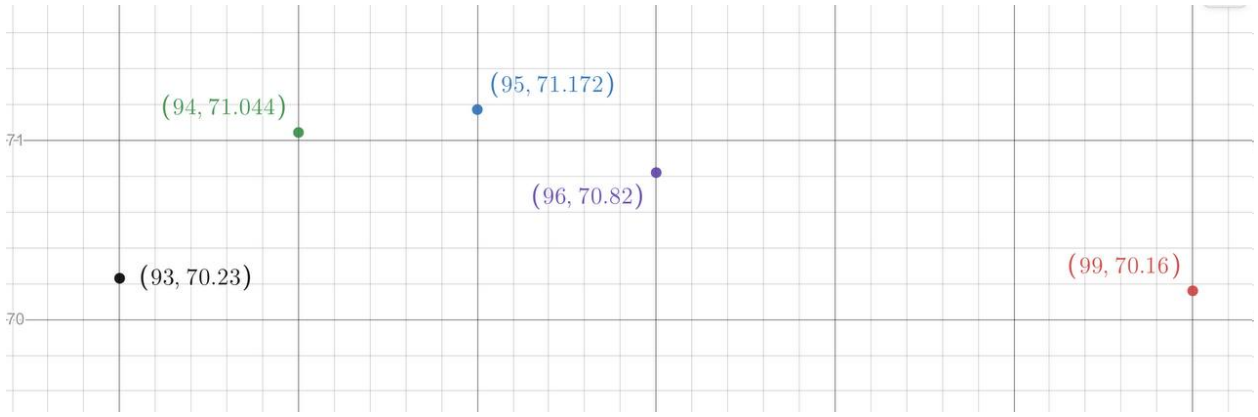


Figure 1.9

From **Figure 1.9**, we realized that a PCA variance of 95% may not result in the highest score on Kaggle, and hypothesized it to be around the 95% range. It turned out to be 94.5%, which has resulted in the best public leaderboard score on Kaggle for us (0.71254), and it was, for more than a week, before Extra Trees finally resulted in higher public leaderboard scores.

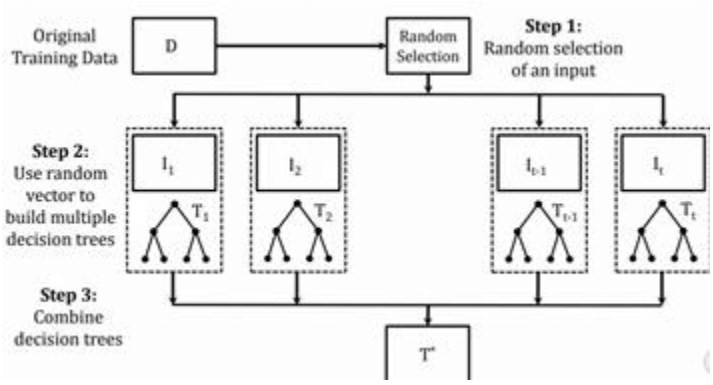
Extra Trees (Best Public Score)

Introduction of Model

Extra Trees (or Extremely Randomized Trees) classifier is an ensemble method just like the random forest classifier. However, the extra trees model tends to outperform the random trees classifier in cases where the data is noisy, as a result of the way that the extra trees model functions. Subjectively speaking, it would be reasonable to assume that the dataset given to us contains data that is noisy- i.e, The relative frequency of the words in each document is independent and erratic.

The Extra Trees model is very closely related to its counterpart i.e. the Random Forests Classifier. During the initial stage, it was very important for us to carefully note down the differences between the two classifiers.

One of the key standout features of the Extra Trees model compared to the above discussed Random Forests Classifier is the absence of Bootstrapping, which involves drawing samples with replacement for our dataset, primarily done to estimate a population parameter. Here, “with replacement” implies that while resampling our dataset several times- there is a possibility that we will get the same data point in more than one of those instances. Another key thing to note is that while other “tree” classifiers have methods to always make the best possible split, extra trees classifier goes for a random split each time. This might be something highly undesirable with datasets with a low entropy and a fewer number of features- however in our case with the highly noisy and sparse dataset available the random nature of the split could theoretically work in our favor. Collectively these two features worked in our favor and gave us our best public leaderboard score.



Hyper-parameter Tuning

Principal Component Analysis

The PCA was carried out by removing the labels on the training dataset and combining it with the testing dataset. By doing so we hoped to achieve better accuracy in terms of unseen models. The ideology behind this was that when exposing the model to the variance of a greater number of samples, overfitting will be avoided and the model will perform well on unseen datasets.

We played around with the number of features passed into the Extra Trees Classifier by carrying out permutations of the following attributes:

- Adjusted variance of the model
- Carried out Minka's Maximum Likelihood Estimator (a PCA technique defined in scikit-learn PCA documentation that helps find the optimum number of features.) The model ran for about 599 minutes and found that the optimum number of features is 4995.

Based on Minka's MLE, we decided that the model would perform best when PCA reduction was not carried out at all. As the number of optimum features was given to be 4995, including 5 extra dimensions would not increase the complexity by a great deal.

Subsequently, our best model in extra trees was produced when we didn't use the combined dataset, and instead fed the raw tf-idf data into Extra trees model with no scaling or PCA reduction carried out.

Model Parameters

Listed below are all the parameters that were adjusted in our extra trees model:

- n_estimators: number of trees to be used in the forest.
- criterion: specifies whether the criterion to use for classification is Gini/ log loss/ entropy

  Gini: $Gini(t) = 1 - \sum_j [p(j|t)]^2$ where j refers to class j and t refers to node t

  Entropy: $Entropy(t) = -\sum_j [p(j|t) logp(j|t)]$ where j refers to class j and t refers to node t

  Log Loss: $LogLoss(t) = -\frac{1}{N}\sum_j y_j \bullet logp(y_j) + (1 - y_j) \bullet log(1 - p(y_j))$

- max_depth
- max_features: {'sqrt', 'log', 'auto'}
- bootstrap and oob_score:
  The extra trees model is done without replacement by default, and this can be changed by setting default to True.

oob_score can be used when bootstrap is set to true,;it signals the algorithm that out-of-the-bag samples must be used to estimate the generalization score

- class_weights were also modified to indicate if the model should take in balanced samples or balanced_subclass samples.
  When set to ‘balanced’, the value of the labels are factored into the calculation of the weight
  When set to ‘balanced_subclass’, weights are computed based on the bootstrap sample given
- warm_start: indicates if the model should build upon the previous results or start anew

We carried out multiple manual iterations by combining the parameters given above in different permutations. For manual validation, we decided to compare the results with our previously best performing model (which was SVM model). We only retrieved the F1 macro scores if the similarity with our best model was above ~85%. Given below are some of the permutations we carried out and their similarity to our best model in percentage::

- max_features = None : 84.96275605214153 %
- bootstrap = True : 90.50279329608938 %
- bootstrap=True, class_weight='balanced', n_jobs=-2, warm_start=True : 90.29329608938548 %
- bootstrap=True, n_jobs=-2, oob_score=True : 91.10800744878958 %
- class_weight='balanced', n_jobs=-2 : 91.15456238361266 %
- class_weight='balanced_subsample', n_jobs=-2 : 91.36405959031657 %
- n_jobs=-2, criterion='log_loss', bootstrap=True, warm_start=True, oob_score=True : 91.4804469273743

Cross Validation

We made use of GridSearchCV along with RepeatedStratifiedKfold to determine the most suitable parameters for our model.
We ran over 1200 fits for 11 models with different numbers of features- each with variance ranging from 90% to 96% and a step size of 0.5%. The resulting gridsearch CV model ran for about 10 hours and gave us the following results:

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': 42}
Best run on 90.0 % variance
Score on this run: 0.5003147671219251
```

```
Parameters:  {'n_estimators': 3, 'n_jobs': -1, 'random_state': 42}
Best run on 90.5 % variance
Score on this run: 0.5070087285065668
```

```
Parameters:  {'n_estimators': 3, 'n_jobs': -1, 'random_state': 42}
Best run on 91.0 % variance
Score on this run: 0.5017061916946094
```

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': None}
Best run on 91.5 % variance
Score on this run: 0.5083469632150285
```

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': 1}
Best run on 92.0 % variance
Score on this run: 0.5021965124556748
```

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': 1}
Best run on 92.5 % variance
Score on this run: 0.5051237073703774
```

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': None}
Best run on 93.0 % variance
Score on this run: 0.5056819540533739
```

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': 1}
Best run on 93.5 % variance
Score on this run: 0.5056997460354934
```

```
Parameters:  {'n_estimators': 1, 'n_jobs': -1, 'random_state': 42}
Best run on 94.0 % variance
Score on this run: 0.5082079723985367
```

```
Parameters:  {'n_estimators': 3, 'n_jobs': -1, 'random_state': 1}
Best run on 94.5 % variance
Score on this run: 0.5043224952387992
```

```
Parameters:  {'n_estimators': 3, 'n_jobs': -1, 'random_state': 1}
Best run on 95.0 % variance
Score on this run: 0.5057429287131003
```

```
Parameters:  {'n_estimators': 3, 'n_jobs': -1, 'random_state': None}
Best run on 95.5 % variance
Score on this run: 0.5016454657198305
```

We ran the fits with the parameters given above and found that the similarity with our best model was very low (~60-70%). So we decided not to proceed with any of the above extra tree models.

Results and Fine-tuning

After trying out grid search CV on the uncombined dataset, we ended up with the best parameters as {n_estimators = 3000, random_state = 42, bootstrapped = true, oob_score = true,

class_weights=balanced, criterion='entropy'}. We arrived at these results after carrying out the model optimization with 1000 estimators and then fine tuned the model by playing around with the estimators while using oob score and changing the criteria to fit maximum information gain.

Our best parameters gave us a private leaderboard score of 0.70299, and our best public leaderboard score of 0.72252.

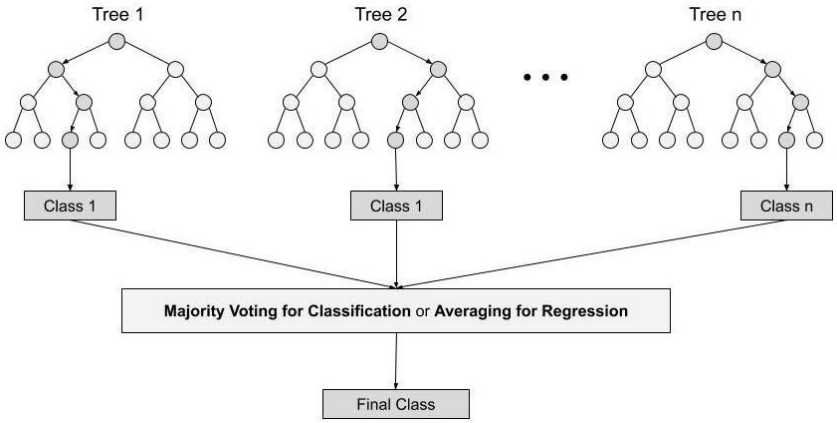
Random Forest

Introduction of Model

Random Forest is also an ensemble learning method. Each individual decision tree in the forest will be used to classify a particular dataset. In a polling manner, the classification that was agreed upon by a majority of the individual trees becomes the final model prediction for that dataset.

Random forest assumes that each classifier (decision tree) created is independent, and hence combining the results of multiple such classifiers would outperform any individual constituent model. This type of ensemble model is called Bagging (as opposed to Boosting used by AdaBoost)

N number of random records are taken from k number of records to be used to construct a decision tree. These individual trees are then used to generate an output given a set of data points. In the final model, majority voting is used for classification (the class that has the most votes from the individual decision tree classifiers)



Hyper-parameter Tuning

Principal Component Analysis

As Random Forest Classifier is faster and parallelizable, the training is done on the entire training dataset (tfidf) first instead of using a PCA reduced training set. This is to get a good gauge of the rough accuracy of the Random Forest Classifier model.

Model Parameters

- n_estimators: [10, 100, 200]
- max_depth: [5, 10, 100, 1000]
- criterion: [“gini”, “entropy”,”log_loss”]
- bootstrap: [True, False]
- class_weight=[“balanced”, “balanced_subsample”]

The parameter n_estimators refers to the number of trees the algorithm builds before averaging the predictions. The n_estimators used in AdaBoost are used as a reference. The max_depth parameter refers to the maximum depth of the decision tree. Using the decision tree classifier run previously as a reference, the max_depth was chosen accordingly. Criterion refers to the splitting criterion for each node to determine the quality of a said split.

Gini: $Gini(t) = 1 - \sum_j [p(j|t)]^2$ where j refers to class j and t refers to node t

Entropy: $Entropy(t) = -\sum_j [p(j|t) logp(j|t)]$ where j refers to class j and t refers to node t

Log Loss: $LogLoss(t) = -\frac{1}{N}\sum_j y_j \bullet logp(y_j) + (1 - y_j) \bullet log(1 - p(y_j))$

Bootstrap is the process of sampling data from the entire dataset to build each tree in the forest. If set to false, the entire dataset will be used to build the tree. Since the dataset is rather large, there are enough datasets to do a non-bootstrapped (albeit unlikely to produce good results). Class weight is the weight associated with each class. The higher class weight would mean greater penalisation when the class is classified wrongly. We hypothesize that because the number of non-hate speech is much more that there is a bias toward classifying all data as non-hate speech. Hence, using a balanced mode where the weights are inversely proportional to class frequencies might help correct that balance.

Random forest produced a result of 0.696 for 'bootstrap': True, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 1000, 'n_estimators': 200. This was further refined in fine-tuning after running PCA at 95% variance on it.

Cross Validation

Once again, we used Stratified K-folds using n-splits =3 and shuffle = true. This is to minimize the computation time of GridSearchCV.

Results and Fine-tuning

We further investigated if the use of a 95% variance would improve the results as compared to a 90% variance. The updated parameter grid used for grid search was also updated as follows:

- n_estimators: [100, 200, 300]
- max_depth: [100, 1000, 1500]

This produced a score of 0.696351 using 'bootstrap': True, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 100, 'n_estimators': 300. However, this was still much lower than the benchmark and thus, other models were prioritized.

AdaBoost

Introduction of Model

AdaBoost is an ensemble learning method that employs many weak binary classifiers to improve the accuracy of the final classification. AdaBoost uses an iterative approach to learn from the mistakes of weak classifiers, and turn them into strong ones by combining the results of weak classifiers and updating the weights of these classifiers according to the error rate produced by the model.

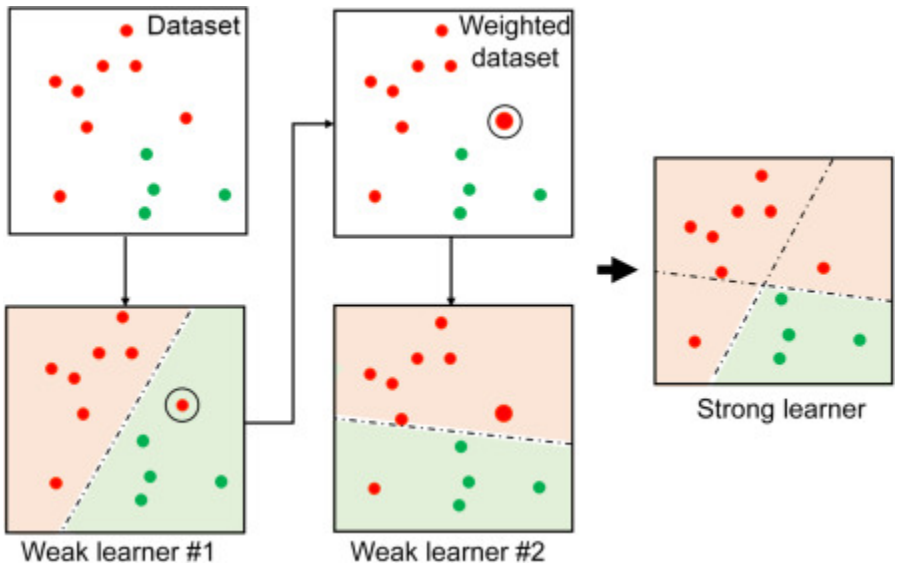


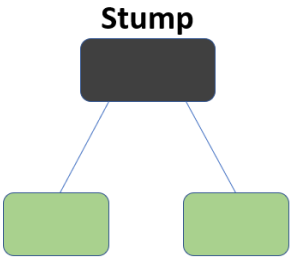
Figure 2.1

As shown in **Figure 2.1**, AdaBoost creates decision stumps which are trees with typically at most one node and two leaves. Similar to Random Forest, they use these decision stumps (as opposed to trees) to do classifications.

$$w_i = 1/N \in [0, 1]$$

Starting with equal weights, a sample from the training dataset is used to create a weak classifier.

This decision stump will classify samples to their target classes using each possible variable and determine the best variable.



Using this decision stump, we look at how many samples are correctly or wrongly classified. For incorrectly classified samples, more weight is assigned to them in the next decision stump. This is shown below by the updating of $w_i^{(j+1)}$, the weight of each sample as shown below. If the classification is correct, $-\ \alpha$ is used, or else $+\ \alpha$ is used. :

$$w_i = \frac{w_{i-1}}{Z_j} \times e^{\pm\alpha} \text{ where}$$
$$\alpha_t = \frac{1}{2} ln \frac{(1-Total\ Error)}{Total\ Error}$$

This above process is repeated until all data points have been correctly classified or until the maximum number of iterations has been reached.

Hyper-parameter Tuning

Principal Component Analysis

Since the weightings of the examples in the next iteration of the algorithm depends on the performances of the previous iteration, the training of an AdaBoost ensemble in its purest form cannot be parallelized as future iterations depend on past iterations. This resulted in a significantly slower training speed as opposed to other training models that can run parallel.

With this knowledge, we started with a PCA accounted variance of 90% instead of 95% (best bet based on previous models). This resulted in a decrease in the number of features to train and hence a decrease in speed while also providing a good gauge of the performance of AdaBoost relative to its peers. In further iterations, a variance of 92.5% and 95% was used for the PCA.

Model Parameters

Using sklearn's GridSearchCV, we ran a grid search to find the hyper-parameters for the AdaBoost. The hyperparameter values are as shown:

- base_estimator: DecisionTreeClassifier (default)

- n_estimators: [10, 100, 200]
- learning_rate: [0.01, 0.1, 1.0]

The base_estimator refers to the type of weak classifier used for the AdaBoost model. This Decision Tree Classifier is also initialized to max_depth = 1. Since the base_estimator is weak to avoid overfitting and to reduce model complexity, we used the default classifier. N_estimators is the maximum number of these decision stumps created (provided there is no perfect fit during training). Initially, it was initialized as [10,100] but was changed to [10, 100, 500, 1000] as the best model returned in the previous iteration was 100. The learning_rate refers to the weight applied to each new decision stump in the subsequent iteration. Since the default is 1.0, the learning rates [0.01, 0.1, 1.0] were chosen to prevent the model from converging too quickly to a suboptimal solution.

Using the above results as the preliminary tuning, the best parameters were determined to be using a learning rate of 1.0 and 200 estimators to produce an accuracy of 0.656. As this result is much lower than the SVM model, further fine-tuning was needed.

Cross Validation

Once again, we used Stratified K-folds using n-splits =3 and shuffle = true. This is to minimize the computation time of GridSearchCV.

Results and Fine-tuning

Using a PCA value of 90%, a mean test score of 0.656164 is obtained when using a learning_rate of 1.0 and n_estimators of 200. This result is significantly lower than the SVM model. However, we proceeded with training the model after a PCA at 95% variance reduction is done. The updated parameters are as follows:

- n_estimators: [100, 100, 300]
- learning_rate: [0.5, 1.0, 1.5]

Based on the first grid search, we further narrowed down the range of optimal classifiers. Using the training dataset with a larger variance, we found that the best model had a learning rate of 1.0 and 300 estimators. This produced a score of 0.664 on the training set and 0.679 on Kaggle. Since this score is extremely far from the current optimal model, we decided not to further optimize this model.

Multinomial Naive Bayes

Introduction of Model

The multinomial naive bayes model is a classifier model that is considered suitable for the classification of data with discrete features, like an integer count for word occurrence. However, it can be used with fractional counts such as tf-idf.

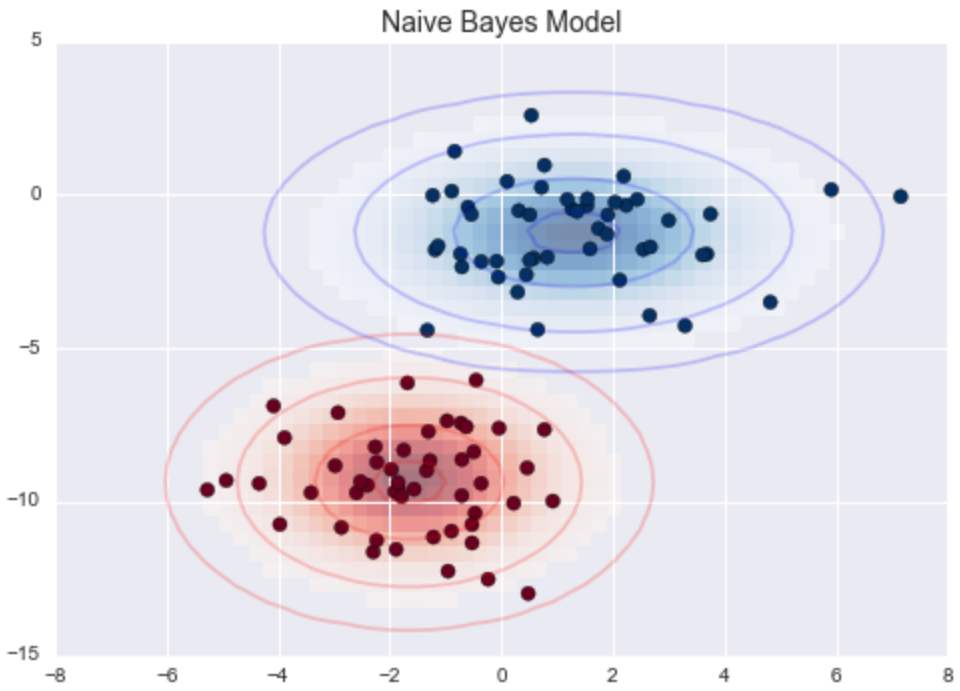
The distribution is parametrized by vectors $\Theta^{y} = (\Theta^{y1}, \Theta^{y2}, ..., \Theta^{yi}, ..., \Theta^{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and Θ^{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y.

The parameters Θ^{y} is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\widehat{\Theta}_{yi} = \frac{N_{yi} + \alpha}{N_{y} + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_{y} = \sum_{i=1}^{n} N_{yi}$ is the total count of all features for class y.

The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.



Hyper-parameter Tuning

Principal Component Analysis

Before we can apply PCA on the Training TF-IDF set, we apply Minmax scaling on the training and testing datasets to scale all data values between 0 and 1. Later, we apply PCA on the combined and scaled testing and training dataset to obtain a reduced dataset. We chose 95% variance for our dataset to eliminate features that have a low contribution to the overall variance of the PCA.

Model Parameters

The model parameters include α, the laplacian/lidstone smoothing parameter used in the Multinomial Naive Bayes model.

The parameters grid that was used for Hyper-Parameter tuning was:

- {'alpha': [0.001, 0.01, 0.1, 0.3, 0.5, 0.7, 1]}

The best parameters from our GridSearchCV were:

- {'alpha': 0.5}

Halving and Randomized GridSearchCV was run using these parameters, with refit and scoring being based on the f1 macro score. This model's predictions were, however, not submitted to Kaggle because it had failed to meet our internal metrics for qualifying a prediction as submission worthy. It has shown a poor similarity of around 62.6% with our then best performing model's prediction (public leaderboard score), which was SVM with RBF kernel trained on 94.5% variance reduced PCA.

Cross Validation

We used the StratifiedKFold cross validation technique to reduce the computation time of the GridSearchCV on the Training Dataset.

The n_splits was set to 3, with shuffling set to true and random state being 1.

Results and Fine-tuning

The best f1-macro for the best parameter from the grid search was:

- best score: 0.382248

After repeated trials, we had deemed this model unworthy for further probing. The poor results might have emerged from the reason that the tf-idf data values were having decimals between 0 and 1 rather than larger integer values which are known to perform well with Multinomial Naive Bayes models. We didn't use this model further and switched to ensemble methods such as extra trees.

What did we self-learn?

Feature Engineering Introduction

Although the Term Frequency - Inverse Document Frequency (tf-idf) feature dataset was provided, there was motivation to explore if there were better ways to potentially represent the raw tweet dataset to allow for better performance. In particular, the use of the n-gram model was of interest to us. N-gram uses a contiguous sequence of n items from a given sample of texts to define a value. Since the original tf-idf was generated using a unigram (single word) as mentioned, we thought that a bigram model (which looks at two words as each unit) might give us a better model. The hypothesis is that since words often come in an order, bigrams (any n-grams where n>1) would better encode this inter-word feature and hence, help us better classify hate speech.

To test this theory, we ran the TfidfVectorizer with max_features=5000 and ngram = 2 on the raw dataset (without the tf-idf features). This generated a dataset with 500 features. We then ran this through PCA at a variance of 90% and conducted GridSearchCV using the AdaBoost machine learning as mentioned before. Since the only difference between this iteration and the one mentioned in the AdaBoost is the use of unigrams versus bigrams, the Kaggle result of 0.571 based on this prediction indicated that this n-gram might not be worth exploring further.

Importance of generalization

Through this project, we realized the paramount importance of minimizing generalization error. Due to the fact that the public score is based on only 20% of the testing dataset, it was more important than ever to ensure that the generalization error of the model is as low as possible.

Furthermore, we realized that our best predictions (based on our best public scores on Kaggle) actually had lower private scores than some of our other predictions that had lower public scores, and a plausible reason could be that there might be a little too much overfitting of the model onto the testing dataset, which may be coincidentally similar to the 20% of the testing dataset used for the public score. This goes to show that ensuring that the model is able to generalize well is often more important than minimizing training error, etc., which may result in overfitting.

Tools that find optimum hyper-parameters

It is no secret that the choice of hyper-parameters is often as important as the choice of the machine learning model. In order to automate and simplify the process of finding optimum

hyper-parameters for training our chosen model, we learnt about the various tools to do so, such as GridSearchCV.

Therefore, instead of us manually entering random hyper-parameters and then submitting our predictions on Kaggle to evaluate the choice of the hyper-parameters based on the public score on Kaggle, we could just run GridSearchCV, which will help us find the best hyper-parameters from a set of different hyper-parameters. This does not require our manual intervention, and also saves us numerous submission attempts onto Kaggle, especially when it is limited.

Applying machine learning

This project has taught us that while it is important that we understand how machine learning models work and how to implement them, it is even more important, in practice, that we know how to evaluate the available results to determine the best hyper-parameters and models. We also learnt that it is also very important to reduce the size of the training dataset (reduce the number of dimensions, etc.), not only to reduce computation time, but to reduce the risk of overfitting.

Should these lessons be taught in future Machine Learning courses?

Absolutely. We think that more focus should be placed on training us on practical things like determining optimum hyper-parameters, choosing the best model, reducing computation time, etc., instead of just teaching us how exactly the models work and how to implement them.

While we understand that all this theory of models is important, we think that they are often not the most important when it comes to applying machine learning to the real-world. After all, there will very likely be free packages with machine learning models already implemented and ready for us to easily use them.