50.007 Machine Learning Report - Numpy Warriors

Group Members:

- 1005501 Harikrishnan Chalapathy Anirudh
- 1005802 Swastik Majumdar
- 1005374 Nguyen Thai Huy
- 1005147 Hayden Ang Wei En
- 1005265 Win Tun Kyaw

Macro F1 scores for applying 2000, 1000, 500, and 100 components on the test set (Task 2 deliverable)

- **F1 score for 100: 0.52971**
- **F1 score for 500: 0.49234**
- **F1 score for 1000: 0.43323**
- **F1 score for 2000 : 0.33998**

Analysis of the reduced components

- **Time** - By using the time module, we were able to record the time taken for each prediction. We obtained 53.19764566421509 seconds for n_components = 2000, 1.2955138683319092 seconds for n_components = 1000, 0.733565092086792 seconds for n_components = 500 and 0.30568814277648926 seconds for n_components = 100. Thus, we can see that, as we reduce the dimension, we can also reduce the time required to train the machine learning model. This is mainly because the lower the dimension, the less computationally expensive it is. Thus, training takes less time while retaining our accuracies, which increases our efficiency.
- **F1 score** - We also noticed that the F1 score reduced as the dimensions increased. We find that for KNN, reduced dimensions work better for model accuracy, and this further supports our motive for using PCA to reduce the dimensions.

Introduction of our best performing model (Extra-trees Classifier)

Our best performing model implements the Extra-Trees Classifier or also known as Extremely Randomized Trees. The core idea behind Extra-Trees Classifier is to fit **n** number of **randomized** decision trees with several sub-samples of our dataset, which then employs averaging to improve the accuracy of our model while controlling over-fitting of the training dataset. Although a single decision tree might be a weak classifier, we introduce variations into the decision tree by randomly selecting a random subset of features when building each node of a tree. By putting **n** number of **randomized** decision

trees together, a single low accuracy classifier is then able to be turned into a high performing forest.

Other models used

The other model that we implemented are as follows:

- Logistic Regression
- Gradient Boosting
- Voting Classifier
- Bagging Classifier
- Random Forest Classifier

Logistic Regression

Logistic Regression predicts the probability of occurrence of a binary event by utilizing a logit function. Although, this model is traditionally a model used to classify binary classes, the sklearn library offers the option for multiclass classification. However, since our problem statement requires to perform binary classification, we do not need this multiclass option.

Gradient Boosting

The Gradient Boosting model is an additive model, allowing for optimization of an arbitrary differentiable loss function. It works by combining weak learning models together to create a strong predictive model, most often, using decision trees similar to Extra-Trees Classifier and Random Forest Classifier. What separates them from one another is that in Gradient Boosting classifiers, the trees are trained sequentially, where each tree is trained to correct the errors of the previous ones and when determining the output of each tree, Gradient Boosting classifiers have to check each tree in a fixed order.

Voting Classifier

A voting classifier is a ML model that trains an ensemble of numerous models, then predicts an output by aggregating the findings of each classifier passed into the Voting Classifier. It predicts the output class based on the highest majority of voting. It supports two types of voting, hard and soft, where hard voting is just the aggregation of the output of each class and selecting the class with the majority of votes. Whereas for soft voting, the summation of probabilities for each class from all models is taken into consideration, and the output class is determined by the class that had the highest probability.

Bagging Classifier

A bagging classifier is an ensemble meta-estimator that fits base classifiers each with a random subset of the original dataset then aggregating the individual predictions similar to the voting classifier to arrive at the final output.

Random Forest Classifier

Random Forest Classifier is very similar to the Extra-Trees Classifier, both composed of many decision trees, where the final output is determined by considering the prediction of all trees. However, where they differ is in the selection of subsample of the dataset when partitioning each node. Random Forest uses bootstrap replicas where it subsamples input data with replacement, this causes increase in variance as bootstrapping makes it more diversified. Another difference is the selection of cut points in order to split the nodes of the trees. Random Forest chooses the optimum split while Extra-Trees chooses it randomly.

How did we "tune" the model?

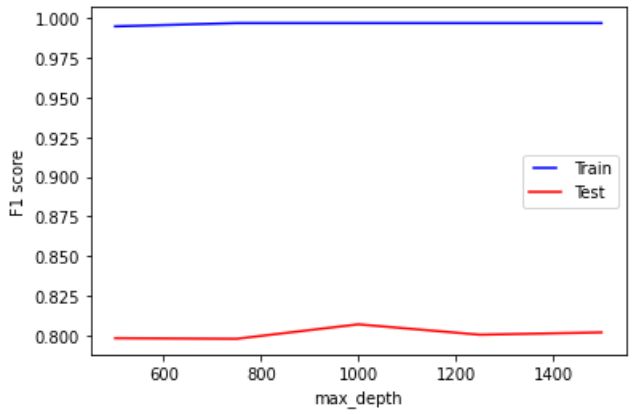
We will also discuss the parameters that we used and the different parameters that we tried before arriving at the best results.

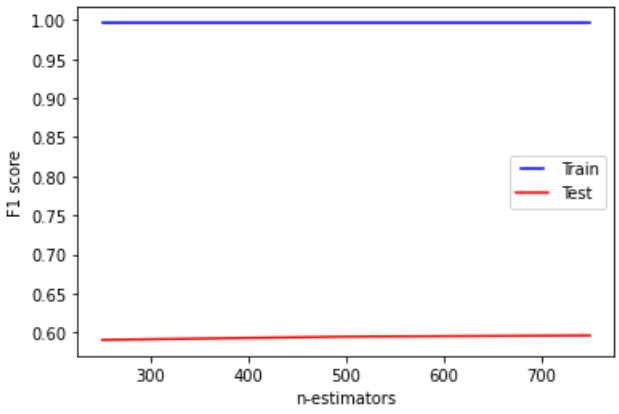
Parameter tuning for Extra-trees classifier (Best performing model)

After doing our research, we found that the most significant parameters that affects the Extra Trees Classifier were the **n_estimators** (number of trees in forest)**, max_depth** (depth of the tree), **max_features** (The number of features to consider when looking for the best split) and the **criterion** (measure quality of split) parameters. Hence, we started by fine tuning these parameters one by one.

We first ran the model with the default value of n_estimators = 100 and tried increasing the number of estimators to 110 and 120 to see its effect on the accuracy. However, manually keying in the parameters took a significant amount of time and thus, we decided to find the sweet spot, by performing a parameter search. We plotted a graph between the F1 score and n_estimators. Larger number of trees is usually better for learning data. It is, however, important to note that adding a lot of trees can significantly slow the training process. From the graph, we observed that the accuracy of the model generally remained constant as we increased the number of estimators. Thus, we finalized **n_estimators = 500** since the time required to train the model would be faster compared to n_estimators = 1000 while still giving similar results.

Moving on, we adjusted the max_features parameter, by manually trying the three options of {“sqrt”, “log2”, None}. We observed that sqrt gave us the best results which is the default parameter for the extra trees classifier.





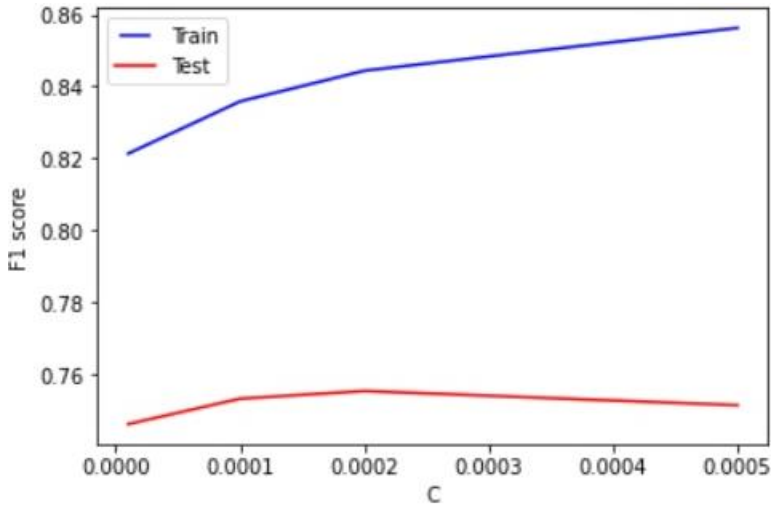
Beyond these parameters, we stared tuning the **max_depth** and the **criterion.** In general, the deeper your decision tree is, the more complex your model is, and if your decision tree is too deep, it might cause your model to overfit the training data. The converse is also true, where too shallow of a tree would result in underfitting of the dataset. Hence, we adjusted the max_depth of classifier so that we are able to achieve good results for both our training and test accuracy. Similar to n_estimators, we plotted a graph between the F1 score and max_depth. From the graph, we observe a slight peak at **max_depth = 1000.**

The criterion measures the quality of the split at each node. Information gain uses the entropy measure as the impurity measure and splits a node such that it gives the most amount of information gain. Whereas Gini Impurity measures the divergences between the probability distributions of the target attribute's values and splits a node such that it gives the least amount of impurity. Log-loss is indicative of how close the prediction probability is to the corresponding actual/true value. After testing the three parameters manually, we found that the resulting accuracies were quite similar, but **criterion = 'log_loss'** gave us the best accuracy.

Parameter tuning for Logistic Regression classifier

We implemented this using the sklearn logistic regression package. Upon researching more on the parameters of this package, we came across the regularization parameter C. The lower value this has, higher the regularization. Since we have most of our predictions to be 0, i.e. not hateful, we set this C to a very low value, 2e-4. We observed that if we increased C, then the model was performing poorly, as it was outputting more 1s than zeroes. We also set our **class_weight=balanced.** This is so that sklearn automatically gives a

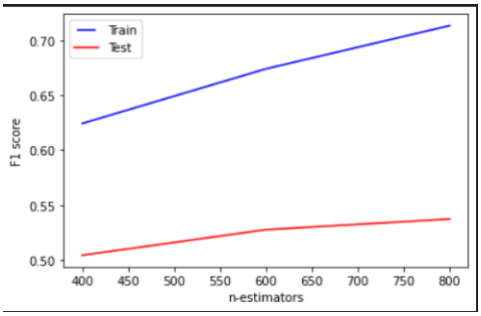
certain class its weightage depending on the frequency of the class occurring, and there is no bias to a certain class.

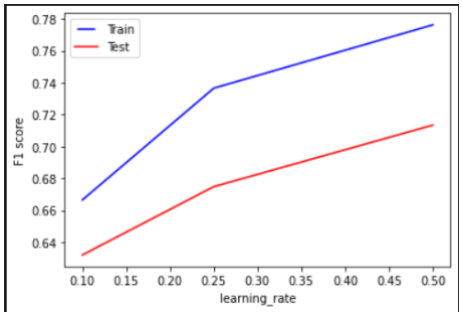


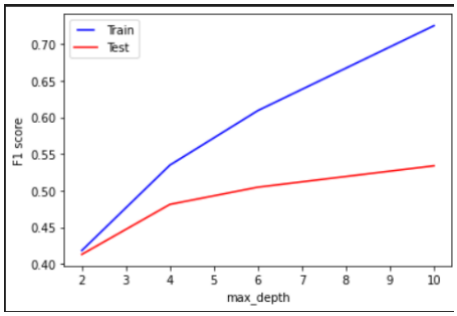
Parameter tuning for Gradient Boosting classifier

For this model, we found that the most significant parameters were **n_estimators** (number of trees in forest), **learning_rate** (shrinks the contribution of each tree by the learning_rate specified), and the **max_depth** (limits the number of nodes in the tree) parameters. Hence, we first started fine tuning these parameters one by one.

We first ran the model with the default value of n_estimators = 100. Due to Gradient Boost's robustness to over-fitting, it typically performs better with a larger number. Thus, to find the sweet spot, we plotted a graph between the F1 score and n_estimators. From the graph, we see that there is a slight peak at **n_estimators = 600.**







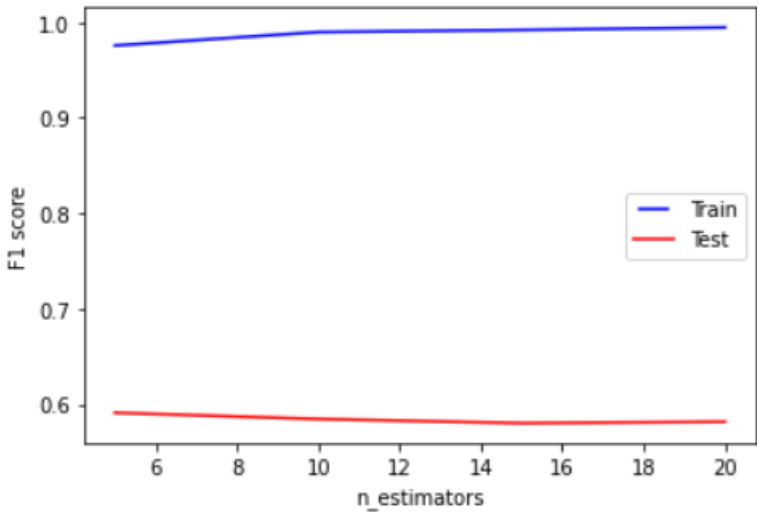
Moving on, we also plotted two others graphs between the F1 score and learning rate, and F1 score with max_depth. From these graphs, we selected **learning_rate = 0.25** and **max_depth = 6.**

Parameter tuning for Voting classifier

The idea behind voting classifier was to “democratize” our model. We wondered what would happen if we chose all our best performing models and took their individual responses to predict an output. We thus chose extra trees classifier, gradient boosting, and logistic regression to train a voting classifier and get the final output.

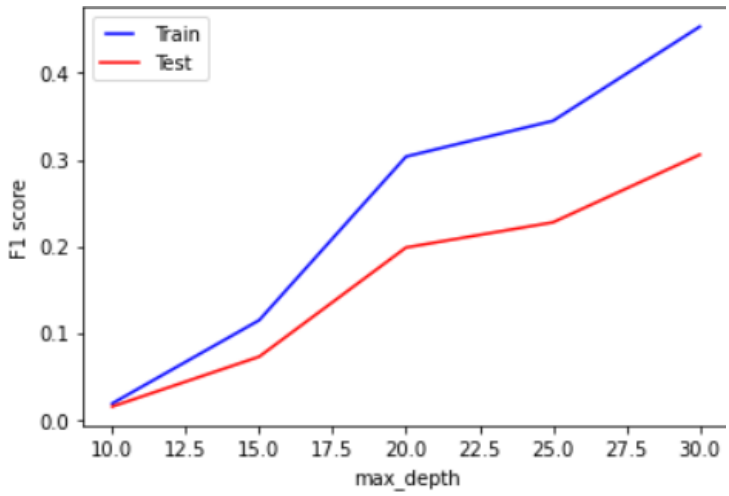
Parameter tuning for Bagging classifier

We also tried the bagging classifier model , we felt that extra trees classifier was very rigid in training the data .We used extra trees with **n_estimators = 10** after plotting a graph between F1 score and n_estimators. We see a slight decrease in the F1 score as we apply n_estimators of more than 10. However, bagging classifier also underperformed compared to extra tree classifier.



Parameter tuning for Random Forest classifier

Additionally, we also tried the Random Forest classifier. By plotting a graph between F1 score and max_depth, we found that the F1 score was increasing as the max_depth increased. Hence, we chose **max_depth = 25** considering the time required to train the model.



Did we self-learn anything that is beyond the course?

We noticed that the training dataset was imbalanced, with significantly more examples of class '0' compared to class '1'. Additionally, many initial models predicted a lot of the provided test features to be class '0'; clearly, the skewed dataset was causing them to perform poorly. Hence, we researched possible solutions and decided to use a data pre-processing method from imblearn called the Synthetic Minority Oversampling Technique (SMOTE).

This technique synthetically generates new data entries in the training set, from the minority class (in this case, class '1'), that are slightly different from existing entries until the two classes are at a 1:1 ratio. The advantages of using SMOTE are that firstly, no information is lost during data pre-processing. This is because all original data entries in the training set are kept. Secondly, variance in the dataset is preserved as the technique adds entries that are only similar to the original, and not duplicates. Finally, overfitting due to imbalanced dataset is reduced.

Such data pre-processing techniques should be taught in future Machine Learning courses. This is because datasets are likely to be imbalanced in real life. Knowing the proper data pre-processing techniques, and when to use the appropriate ones, would enhance one's effectiveness as a data scientist and improve the models that they create. This would also improve students' understanding of not only the limitations of using machine learning models and how to overcome them, but also the concepts of underfitting and overfitting. Additionally, introduction to concepts such as hyperparameter tuning will surely help Machine Learning engineers in the future.