Machine Learning Assignment (4)

Decision Tree & Ensemble Algorithms

Instructed by Dr. Sattar Hashemi

Abbas Mehrbanian

abbas.mrbn@gmail.com, Stu ID: 40130935

Reza Tahmasebi

saeed77t@gmail.com, Stu ID: 40160957

Table of Contents

1 Introduction

This project splits into two parts, and in each, we have to learn about the concepts of decision trees, ensembles, AdaBoost, and imbalanced data.

2 Implementation

Since the details of implementation are a lot, we will focus on key points and main decisions made in implementation for each part.

2.1 Grid search

Grid search is a technique used for hyperparameter tuning in machine learning models. It is a systematic approach to find the optimal combination of hyperparameter values that yield the best performance for a given model.

Hyperparameters are parameters that are set before the learning process begins and control the behavior of the learning algorithm. Examples of hyperparameters include learning rate, regularization strength, number of hidden units in a neural network, etc.

Grid search involves defining a grid of hyperparameter values to explore. It exhaustively searches all possible combinations of these values and evaluates the model's performance using cross-validation or a separate validation set.

By evaluating the model's performance for each combination of hyperparameters, grid search helps to identify the hyperparameter values that result in the best performance. It allows you to compare and select the optimal set of hyperparameters for your model.

Grid search is often performed in conjunction with cross-validation, where the dataset is divided into multiple subsets (folds). The model is trained on different combinations of hyperparameters using one subset as the validation set and the remaining subsets as the training set. This process is repeated for each fold, and the performance metrics are averaged to obtain a more reliable estimate of the model's performance.

The main advantage of grid search is that it provides an automated and systematic way to search for the best hyperparameter values, saving time and effort compared to a manual

search. It helps to fine-tune models and improve their performance by optimizing hyperparameters.[1]

We implemented grid search as a method named *perform_grid_search* inside utils.py file. We will use this method to evaluate the performance of our implementation with different config sets. This function reports the metrics that were requested in project instructions such as Precision, Recall, F-measure, ROC curve, and AUC measures which are one-class measures and G-mean.

2.2 Part A

2.2.1 Pre-processing data

Using the original dataset, we faced problems like the decision tree not getting deeper in height. To fix this issue we first temporarily transformed our 3 class data into a binary class data by first considering the smallest class as minority data and the rest of data as majority. Then we compute either correlation between features using a method like Kendall or simply calculate the Hellinger distance for each feature.

In our implementation we used the Hellinger distance approach and removed the datasets with maximum possible value which is $\sqrt{2} \cong 1.41$ according to original paper provided in project files. To be sure we removed any feature which had Hellinger distance above "1". This helped the tree to grow deeper. We also used "stratify" argument in *train_test_split* method to split data in a stratified fashion.

2.2.2 HDDT

We implemented HDDT as a class named HDDT inside HDDT.py file. This class accepts two parameters max_depth and cut_off_size which were discussed in project instructions file.

We used a recursive approach as was shown in pseudo-code inside project instructions, to implement the training process of HDDT. You can use *fit* method to train the model given the data and labels. This function calls the *_build_tree* which is a recursive method to build the tree. There are three conditions defined inside this method to end recursion:

- If the node is pure, by other means: all of the data are from a single class.

- If the tree has reached the maximum depth
- If the node can't be split in smaller subsets anymore.

Our implementation also uses a custom data structure named Node defined as a separate class that save information about a node in tree such as: The feature this node representing, the label assigned to this node (for terminal nodes), the number of negative and positive samples of the node, the total samples of this node and the children which is saved as a python collection.

To find the best feature for the current node based on Hellinger distance we use *_find_best_split* function with help of *calc_hellinger_distance* and *_split* functions. The *_find_best_split* function finds the best feature and returns the dataset splits based on chosen feature [as node's children] and returns them.

The prediction is done through a recursive process in which we traverse the tree until we reach a leaf for a given sample. the prediction is done with *predict* method which uses *_predict_single* method to recursively traverse the tree.

We also calculate the probability of each node which will come handy later in 2.2.5. We simply calculate the probability for both classes by dividing the number of samples belonging to that class to the number of total samples. This is done through *predict_prob* method which uses another method named *_peredic_single_probs* to recursively traverse the tree.

2.2.3 Minority vs. rest

The first config we're going to work with is Minority vs Rest which the class with smallest amount of data is considered as minority and the rest is considered as majority. To do this separation we wrote a method named *select_minority_vs_rest* which accepts the data and split it into two-class data with the method we just discussed.

2.2.4 One vs. one

For this part we defined a class named as OVO_HDDT so we can pass it to *perform_grid_search* and implement its own predict method. We make 3 models in *fit* function of this class since we have 3-class dataset. We also save the original labels

alongside the model so we can use them in prediction. We use *select_OVO* function inside this class to made sub datasets.

The tricky part is the prediction method in which we predict samples using all 3 models and assign them their real label and then take a majority vote among all 3 predictions made for each test sample and choose that as the predicted label.

2.2.5 One vs. all

Again, we defined a class named OVA_HDDT for this part for the same reason as before. We use *select_OVA* function inside this class to made sub datasets. The fit method is used to train the model.

The prediction for this part is different. At first, we tried the same method we used for One vs. one approach but it didn't work well in case of One vs. all approach. To fix this issue we used the probabilities of minority class and chose the argmax of 3 predicted probabilities for each test sample.

2.3 Part B

The description for this part is almost given inside the project instructions file alongside the pseudo-codes.

2.3.1 Pre-processing data

Since the given dataset contains nan values, we used *IterativeImputer* class from Sickit learn to fill those values with mode approach.

2.3.2 Bagging with under-sampling

We impelemted this part as a class named BaggingWithUndersampling. To sample the data, we used NumPy functions such as *unique*, *where*, *random.choice* and *concatenate*.

We added two more parameters to this class named: samples_ceof and disable_under_sampling. samples_ceof is used as coefficient times the number of samples from minority class. disable_under_sampling is used to whether do under-sampling or not (This has been done due to instructions although this deemed to be useless since we use the bagging method to use its sampling properties.)

2.3.3 AdaBoost

We implemented the Adaboost algorithm as a class named AdaBoostUnderSampling which accepts number of estimators (T), base learner model object and rounds.

3 Results

3.1 Part A

The results for each implemented model are given in tables below.

Table 1: Average of Precision, Recall, F-measure, G-mean, and AUC using grid search method for 10 runs for HDDT model with minority vs rest as input binary dataset.

Max depth	Cut-off size	Precision	Recall	F-measure	G-mean	AUC
2	**10**	0.6282	0.6492	0.6385	0.7924	0.8081
2	**50**	0.6282	0.6492	0.6385	0.7924	0.8081
2	**100**	0.6282	0.6492	0.6385	0.7924	0.8081
3	**10**	0.6425	0.915	0.7549	0.9355	0.9357
3	**50**	0.6425	0.915	0.7549	0.9355	0.9357
3	**100**	0.6425	0.915	0.7549	0.9355	0.9357
4	**10**	0.748	0.8708	0.8048	0.9214	0.9229
4	**50**	0.748	0.8683	0.8037	0.9201	0.9216
4	**100**	0.7511	0.8675	0.8051	0.9199	0.9214
5	**10**	0.8333	0.8042	0.8185	0.8906	0.8952
5	**50**	0.8338	0.8025	0.8178	0.8897	0.8944
5	**100**	0.8118	0.8125	0.8122	0.8941	0.8982
None	**10**	0.8308	0.8142	0.8224	0.8959	0.9
None	**50**	0.832	0.8092	0.8204	0.8932	0.8976
None	**100**	0.8082	0.8217	0.8149	0.8989	0.9025

Table 2: Average of Precision, Recall, F-measure, G-mean, and AUC using grid search method for 10 runs for HDDT One vs. One model.

Max depth	Cut-off size	Precision	Recall	F-measure	G-mean	AUC
2	**10**	0.5786	1.0	0.733	0.9683	0.9688
2	**50**	0.5786	1.0	0.733	0.9683	0.9688
2	**100**	0.5786	1.0	0.733	0.9683	0.9688
3	**10**	0.7528	0.8475	0.7973	0.9096	0.9118
3	**50**	0.7528	0.8475	0.7973	0.9096	0.9118
3	**100**	0.7528	0.8475	0.7973	0.9096	0.9118
4	**10**	0.8076	0.7833	0.7953	0.878	0.8837
4	**50**	0.8079	0.7817	0.7946	0.8771	0.8829
4	**100**	0.7978	0.7892	0.7935	0.8807	0.886
5	**10**	0.8314	0.7683	0.7986	0.8707	0.8775
5	**50**	0.8294	0.77	0.7986	0.8715	0.8782
5	**100**	0.8126	0.7842	0.7981	0.8787	0.8843

Table 3: Average of Precision, Recall, F-measure, G-mean, and AUC using grid search method for 10 runs for HDDT One vs. All model.

Max depth	Cut-off size	Precision	Recall	F-measure	G-mean	AUC
2	**10**	0.6282	0.6492	0.6385	0.7924	0.8081
2	**50**	0.6282	0.6492	0.6385	0.7924	0.8081
2	**100**	0.6282	0.6492	0.6385	0.7924	0.8081
3	**10**	0.6425	0.915	0.7549	0.9355	0.9357
3	**50**	0.6425	0.915	0.7549	0.9355	0.9357
3	**100**	0.6425	0.915	0.7549	0.9355	0.9357
4	**10**	0.7491	0.8683	0.8043	0.9202	0.9217
4	**50**	0.7491	0.8658	0.8032	0.9189	0.9205
4	**100**	0.7511	0.865	0.804	0.9186	0.9202
5	**10**	0.8383	0.7992	0.8183	0.888	0.893

5	**50**	0.8387	0.7975	0.8176	0.8871	0.8922
5	**100**	0.815	0.8075	0.8112	0.8915	0.8959

3.2 Part B

3.2.1 Bagging

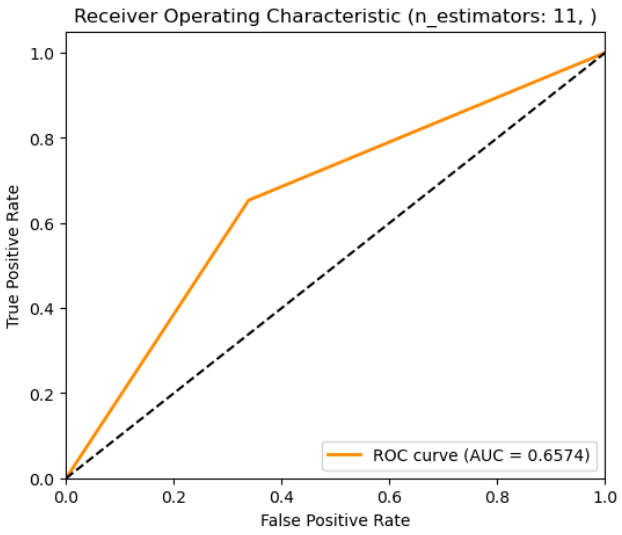
3.2.1.1 Using built-in decision tree with under-sampling

Table 4: Mean and STD of evaluation metrics achieved by using grid search method for 10 runs for Bagging with build-in decision tree.

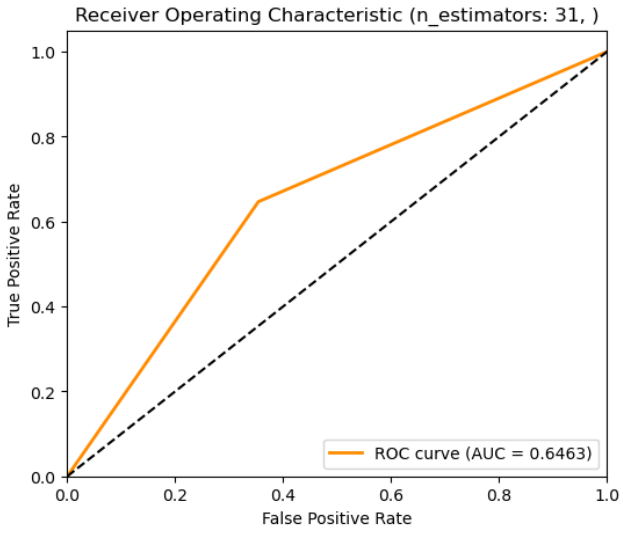
T	**Precision**		**Recall**		**F-measure**		**G-mean**		**AUC**	
	Mean	**STD**	**Mean**	**STD**	**Mean**	**STD**	**Mean**	**STD**	**Mean**	**STD**
11	0.1152	0.0189	0.6533	0.1137	0.1956	0.0317	0.6544	0.0557	0.6574	0.056
31	0.1096	0.0121	0.6467	0.0748	0.1873	0.0203	0.6449	0.0372	0.6463	0.0366
51	0.1157	0.0141	0.68	0.0872	0.1975	0.0235	0.6614	0.0437	0.6639	0.0433
101	0.1187	0.0196	0.6833	0.0872	0.2019	0.0314	0.6665	0.0515	0.6685	0.0511

Table 5: Accuracies achieved by grid search method for 10 runs for Bagging with build-in decision tree.

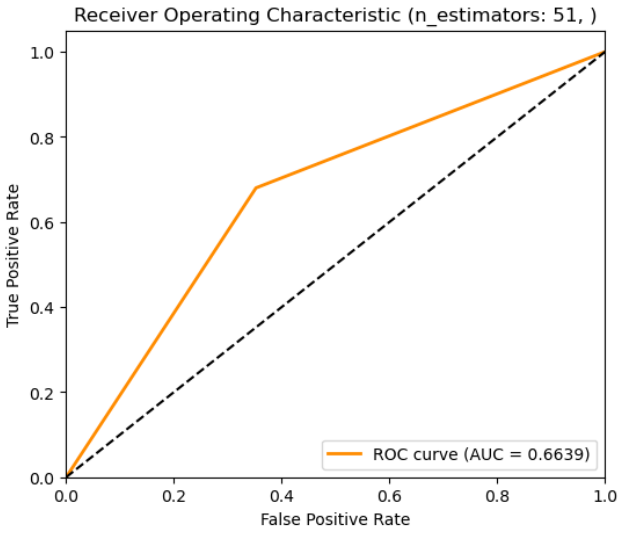
T	**Mean**	**STD**
11	0.6609	0.0454
31	0.646	0.0434
51	0.6498	0.0244
101	0.6555	0.0328



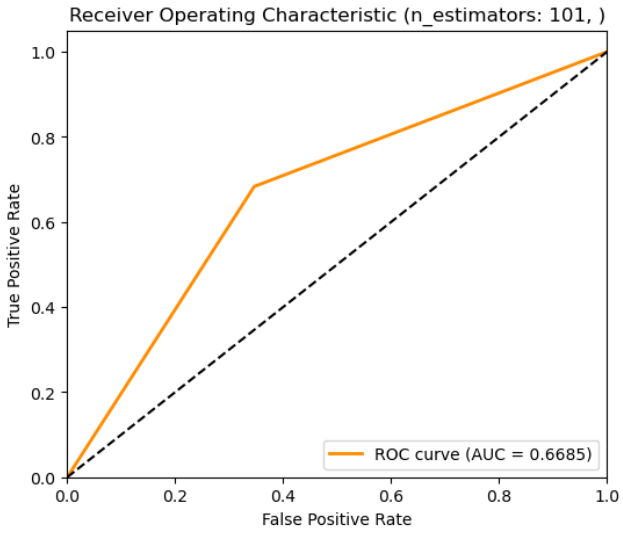
(a) T = 11



(b) T = 31



(c) T = 51



(d) T = 101

Figure 1: ROC curve plots for the minority class for given values (Bagging with built-in decision tree as its learner model)

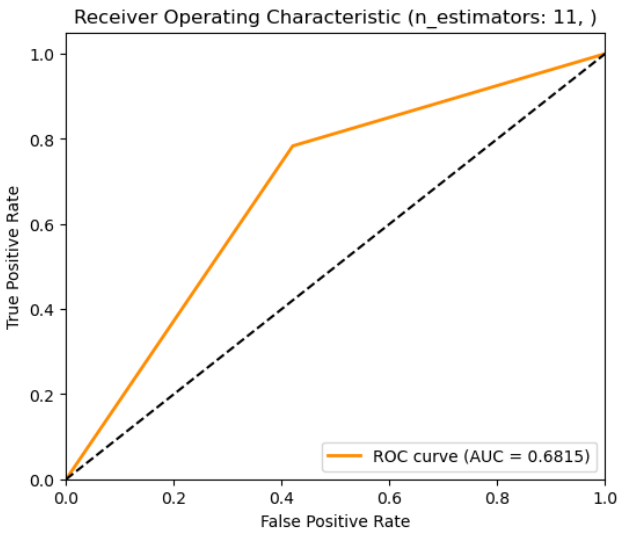
3.2.1.2 Using HDDT with under-sampling

Table 6: Mean and STD of evaluation metrics achieved by using grid search method for 10 runs for Bagging with build-in decision tree.

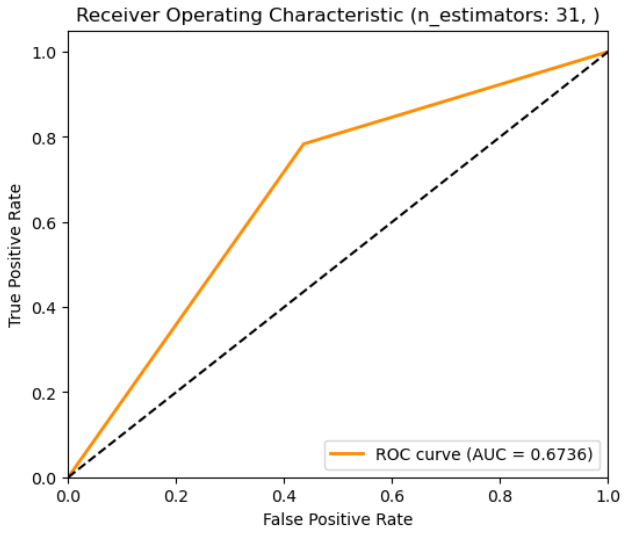
T	Precision		Recall		F-measure		G-mean		AUC	
	Mean	**STD**	**Mean**	**STD**	**Mean**	**STD**	**Mean**	**STD**	**Mean**	**STD**
11	0.112	0.0181	0.7833	0.1195	0.1958	0.0306	0.671	0.056	0.6815	0.0609
31	0.1081	0.0107	0.7833	0.0946	0.1898	0.0183	0.6617	0.0355	0.6736	0.0391
51	0.1127	0.0202	0.7733	0.1218	0.1965	0.0338	0.6706	0.064	0.6797	0.0674
101	0.1063	0.0186	0.7233	0.13	0.1853	0.032	0.6504	0.0608	0.6571	0.0644

Table 7: Accuracies achieved by grid search method for 10 runs for Bagging with build-in decision tree.

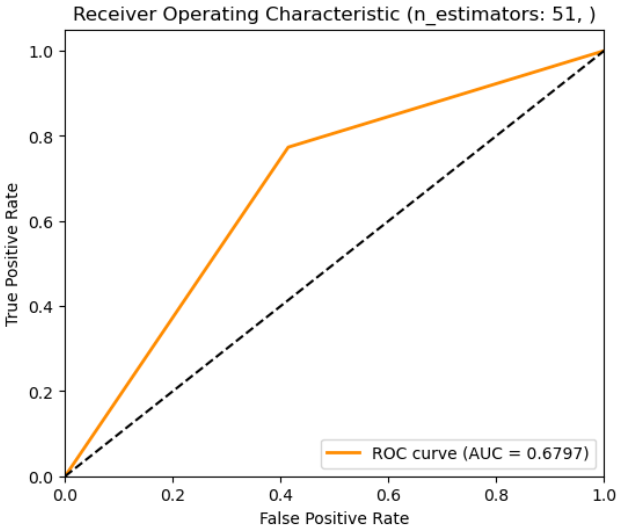
T	Mean	STD
11	0.5924	0.0409
31	0.5777	0.0436
51	0.5979	0.0453
101	0.5992	0.03



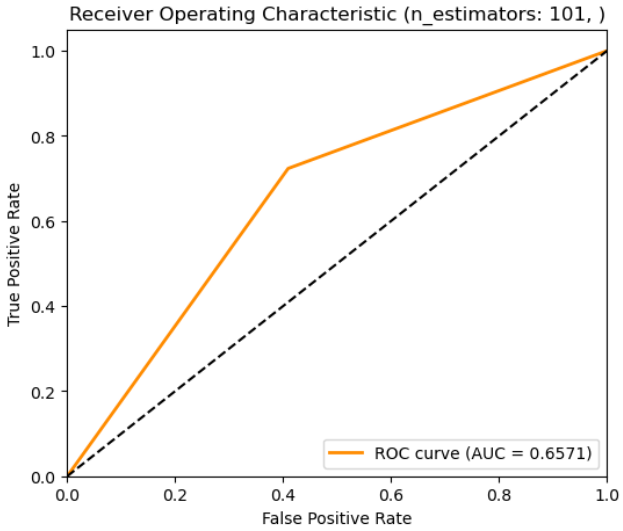
(a) T = 11



(b) T = 31



(c) T = 51



(d) T = 101

Figure 2: ROC curve plots for the minority class for given (Bagging with HDDT as learner)

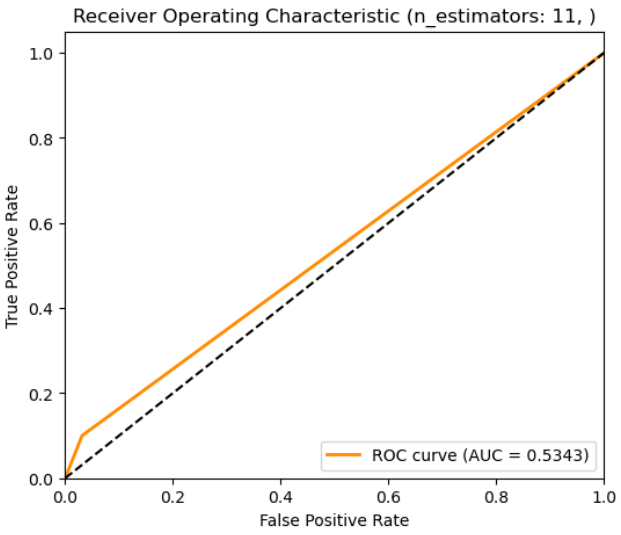
3.2.1.3 Using HDDT without under-sampling

Table 8: Mean and STD of evaluation metrics achieved by using grid search method for 10 runs for Bagging with HDDT without under-sampling

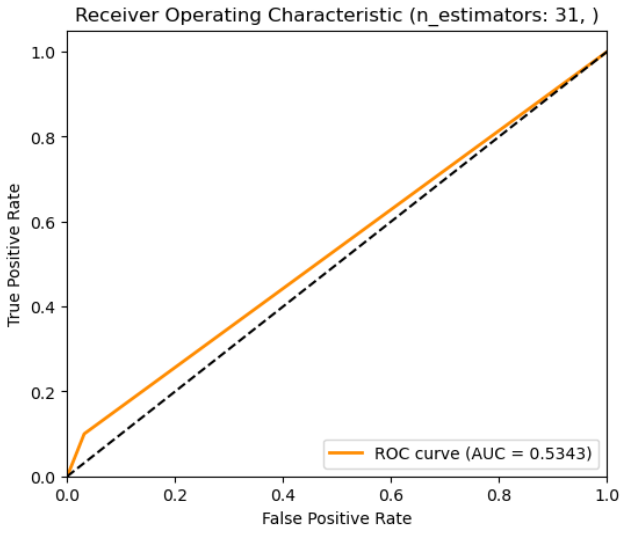
T	Precision		Recall		F-measure		G-mean		AUC	
	Mean	**STD**	**Mean**	**STD**	**Mean**	**STD**	**Mean**	**STD**	**Mean**	**STD**
11	0.1765	0	0.1	0	0.1277	0	0.3112	0	0.5343	0
31	0.1765	0	0.1	0	0.1277	0	0.3112	0	0.5343	0
51	0.1765	0	0.1	0	0.1277	0	0.3112	0	0.5343	0
101	0.1765	0	0.1	0	0.1277	0	0.3112	0	0.5343	0

Table 9: Accuracies achieved by grid search method for 10 runs for Bagging with HDDT algortihm without under-sampling

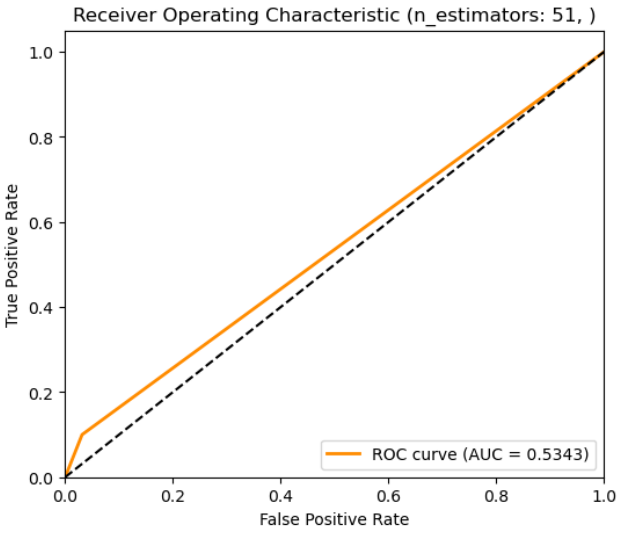
T	Mean	STD
11	0.9139	0
31	0.9139	0
51	0.9139	0
101	0.9139	0



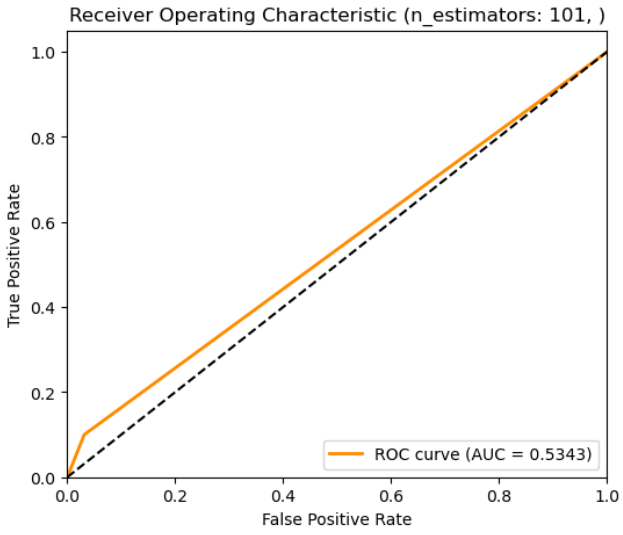
(a) T = 11



(b) T = 31



(c) T = 51



(d) T = 101

Figure 3: ROC curve plots for the minority class for given values (Bagging without under-sampling using HDDT as base learner)

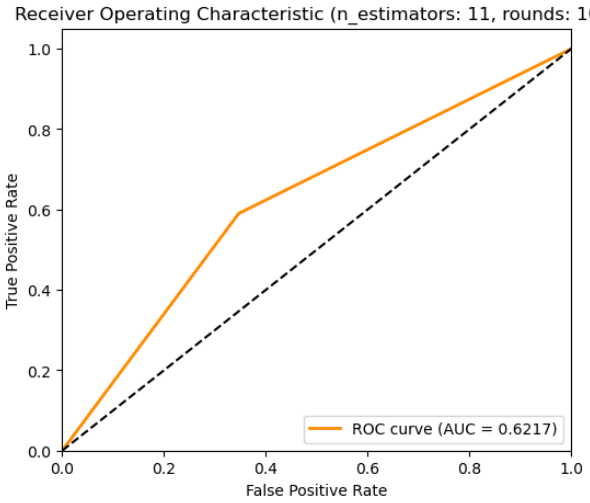
3.2.2 AdaBoost

Table 10: Mean and STD of evaluation metrics achieved by using grid search method for 10 runs with AdaBoost with Built-in decision tree as its base learner model.

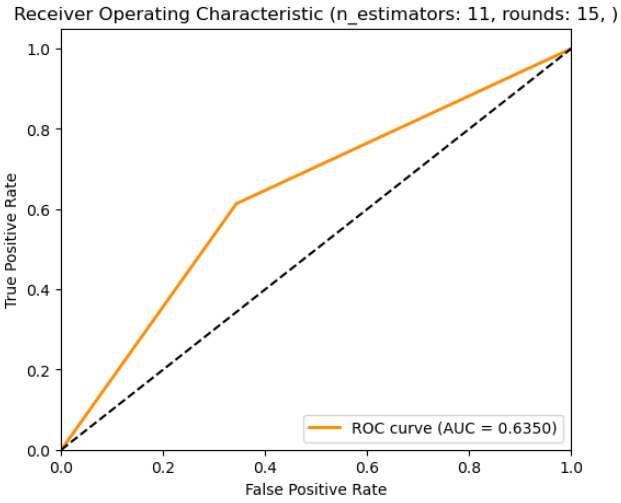
T	Rounds	Precision		Recall		F-measure		G-mean		AUC	
		Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD
11	**10**	0.1043	0.0147	0.59	0.0423	0.1769	0.0223	0.62	0.033	0.6217	0.0327
11	**15**	0.1071	0.0108	0.6133	0.1024	0.1821	0.0188	0.6309	0.039	0.635	0.038
31	**10**	0.1099	0.0138	0.7	0.0989	0.1898	0.0234	0.6548	0.0444	0.6589	0.0452
31	**15**	0.1084	0.0203	0.6233	0.0831	0.1844	0.0323	0.6347	0.0511	0.6364	0.0506
51	**10**	0.1073	0.0181	0.6233	0.1146	0.1827	0.0301	0.6325	0.0556	0.6363	0.0543
51	**15**	0.1127	0.0173	0.6233	0.0615	0.191	0.0269	0.6447	0.0406	0.646	0.0404
101	**10**	0.1037	0.012	0.62	0.0618	0.1774	0.0186	0.6266	0.0301	0.6282	0.0304
101	**15**	0.1105	0.0129	0.5967	0.0706	0.186	0.02	0.6319	0.0339	0.6351	0.0325

Table 11: Accuracies achieved by grid search method for 10 runs with AdaBoost with Built-in decision tree as its base learner model.

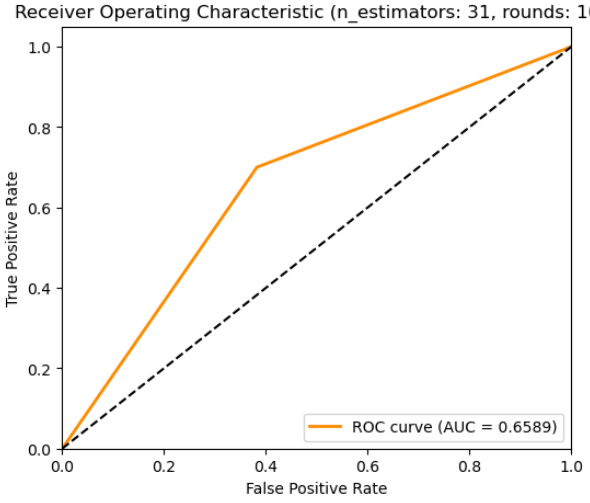
T	Rounds	Mean	STD
11	**10**	0.6494	0.0468
11	**15**	0.654	0.0338
31	**10**	0.6229	0.0359
31	**15**	0.6479	0.0442
51	**10**	0.6477	0.0379
51	**15**	0.6571	0.0475
101	**10**	0.6353	0.0387
101	**15**	0.6687	0.0408



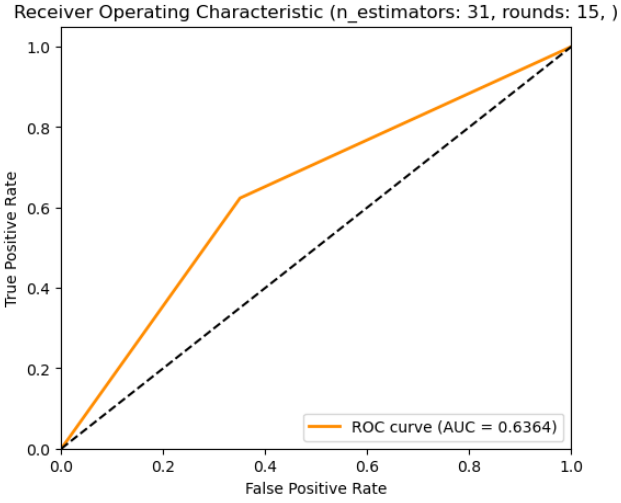
(a) T = 11, rounds = 10



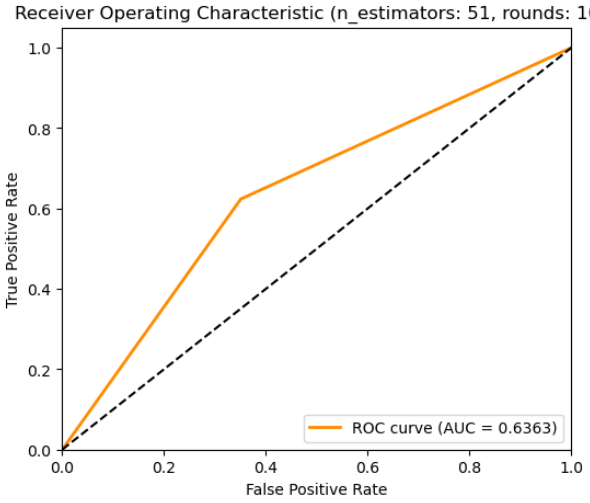
(b) T = 31, rounds = 15



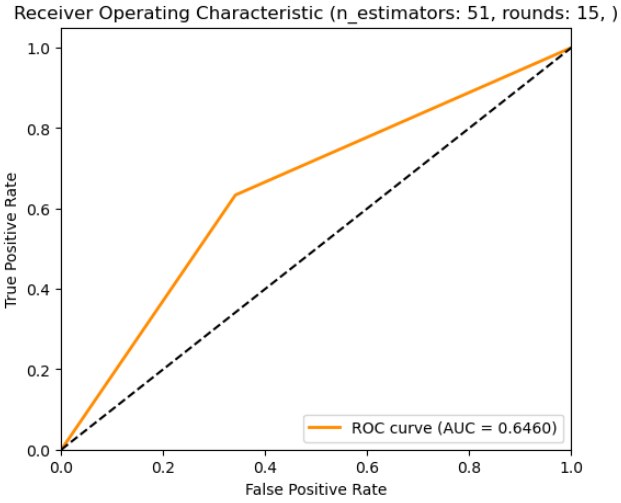
(c) T = 31, rounds = 10



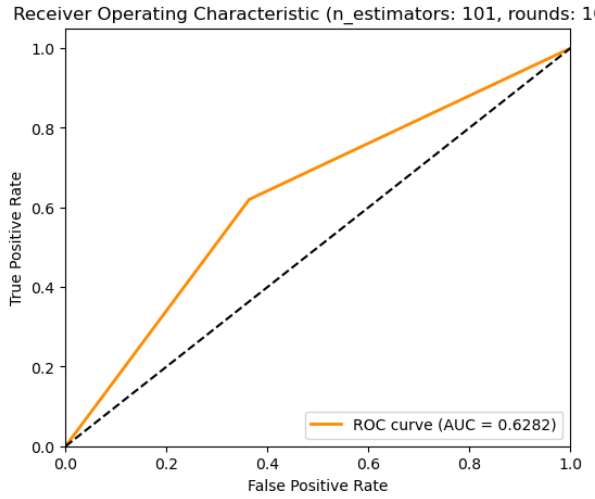
(d) T = 31, rounds = 15



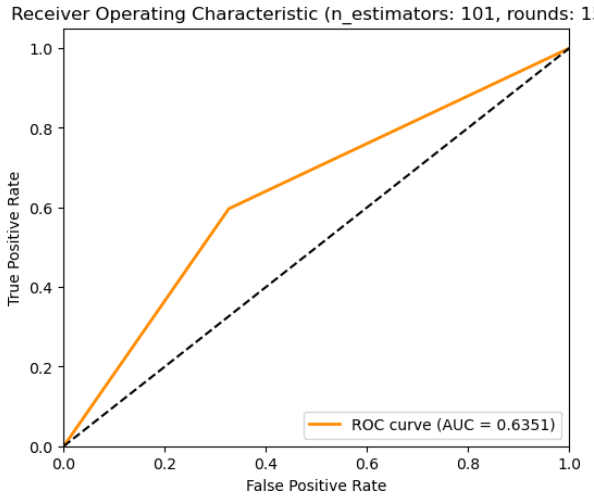
(c) T = 51, rounds = 10



(d) T = 51, rounds = 15



(c) T = 101, rounds = 10



(d) T = 101, rounds = 15

Figure 4: ROC curve plots for the minority class for given values (AdaBoost with built-in decision tree as its learner model)

3.3 Conclusions

When dealing with imbalanced data, metrics such as G-mean, ROC curve, and AUC (Area Under the Curve) offer several benefits over traditional metrics like accuracy, precision, recall, and F-measure. Here are the advantages of using G-mean, ROC curve, and AUC:

G-mean[1]:

- G-mean is particularly effective in imbalanced datasets as it considers both sensitivity (recall) and specificity.
- It provides a balanced measure by taking into account the performance on both positive and negative instances.

[1] Geometric Mean

- G-mean is especially useful when the cost of misclassifying either class is equally important.
- It helps to evaluate the overall performance of the model across all classes and provides a fair assessment.

ROC Curve[1]:

- The ROC curve is a graphical representation of the model's performance across various classification thresholds.
- It illustrates the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity) at different decision boundaries.
- The ROC curve provides a comprehensive evaluation of the model's ability to rank instances and is less sensitive to class imbalance.
- It helps to visualize the model's discrimination ability across different thresholds.

AUC[2]:

- AUC is a summary metric derived from the ROC curve, representing the overall performance of the model.
- It quantifies the model's ability to distinguish between positive and negative instances across various threshold settings.
- AUC provides a single value that captures the ranking and discrimination ability of the model.
- It is robust to class imbalance and provides a reliable measure of performance in imbalanced datasets.
- AUC is useful for comparing models and selecting the best-performing one.

[1] Receiver Operating Characteristic Curve
[2] Area Under the Curve

Compared to traditional metrics like accuracy, precision, recall, and F-measure, G-mean, ROC curve, and AUC offer a more comprehensive evaluation of the model's performance in imbalanced datasets. They consider both positive and negative instances, provide a balanced measure, and are less affected by the class distribution. These metrics help to assess the model's overall discrimination ability, rank instances correctly, and make informed decisions when dealing with imbalanced data.[2]

For the part A we can clearly see for all three configuration sets that the HDDT performed very well with even doing any under-sampling according to metrics like g-mean and AUC (higher is better). We won't rely on accuracy because it can be deceiving for such cases that we are dealing with imbalanced data.

For part B we tried the general results weren't very satisfying but since the dataset provided is a real world dataset it is logical. However by looking the ROC curve plots for the parts that we used either bagging or AdaBoost algorithm we can see the method worked very well. The more closer the orange line is to the center dotted line the closer the results are to a random classifier.

We tried the HDDT algorithm without under-sampling to see how HDDT does on its own since it is said to be robust against the imbalanced datasets. The results are equal to just run this model for 1 time since as you can see the STD value is 0 which means the results we are getting are not changing through iterations. We can see although the accuracy seems to be great, the other metrics are not, which describes the deceivability of this metric.

While Hellinger distance decision trees are generally considered robust against imbalanced datasets, the combination of bagging and under-sampling can further enhance the performance by reducing the dominance of the majority class and providing a more balanced training data for each decision tree in the ensemble. In this case can see that bagging with under sampling using HDDT as its base learner helped the model performance a bit.

3.4 Questions

3.4.1 Part A

3.4.1.1 Properties of the HDDT

The HDDT algorithm is specifically designed to handle imbalanced datasets and has several properties that make it suitable for such data:

Utilizes the Hellinger distance: HDDT employs the Hellinger distance as a splitting criterion. The Hellinger distance is a symmetric measure of the similarity between probability distributions and is well-suited for capturing differences between class distributions in imbalanced data.

Focuses on minority class: HDDT aims to improve the identification of patterns in the minority class by giving it higher importance during tree construction. It takes into account the distribution of instances in both classes, rather than relying solely on class frequencies.

Handles overlapping class distributions: HDDT considers overlapping class distributions and aims to find decision boundaries that can effectively separate the classes, even when they are not well-separated.

Decision boundary: HDDT builds a binary decision tree where each internal node represents a feature test, and each leaf node represents a class label. It partitions the feature space based on Hellinger distance.

Robust against imbalanced data: HDDT is designed to address the challenges posed by imbalanced datasets. It can mitigate the bias towards the majority class and improve the classification performance for the minority class. HDDT is suitable for unbalanced data because Hellinger distance is less sensitive to class imbalance compared to other measures like Gini index or information gain. It focuses on the shape of the distribution rather than the absolute counts.

3.4.1.2 Differences between Hellinger distances, Gini index, and information gain

Hellinger Distance: The Hellinger distance is a statistical measure used to quantify the similarity or dissimilarity between probability distributions. It measures the distance between two probability density functions by considering the square root of the sum of the squared differences between their square root values. In the context of decision trees, the

Hellinger distance can be used as a splitting criterion to determine the optimal feature and threshold for splitting a node.

Advantages:

- The Hellinger distance is sensitive to differences in the shape and overlap of class distributions, making it useful for handling imbalanced datasets.
- It can capture the similarity between class distributions even when they overlap or are not well-separated.

Limitations:

- The Hellinger distance does not directly provide a measure of impurity or information gain. It primarily focuses on capturing differences between probability distributions.

Gini Index: The Gini index is a measure of impurity or diversity within a node in a decision tree. It quantifies the probability of misclassifying a randomly chosen element in a node. The Gini index is calculated by subtracting the sum of the squared probabilities of each class label from one.

Advantages:

- The Gini index is computationally efficient and easy to calculate.
- It is a widely used impurity measure and is commonly employed in decision tree algorithms such as CART.

Limitations:

- The Gini index does not directly consider the distributional differences between class labels or the information gain achieved by splitting on a feature. It only focuses on impurity reduction within individual nodes.

Information Gain: Information gain is a concept derived from information theory and is used as a criterion to measure the reduction in entropy (or uncertainty) achieved by splitting a node based on a specific feature. It quantifies the amount of information gained about the class labels after the split.

Advantages:

- Information gain takes into account the entropy of the class labels, which reflects the uncertainty of the data.
- It captures the reduction in entropy and helps to identify the most informative features for splitting.

Limitations:

- Information gain may exhibit a bias towards features with a large number of distinct values, potentially favoring complex or irrelevant attributes.
- It can be sensitive to the number of categories or levels in a feature, potentially leading to a bias towards features with a higher number of distinct values.

It's important to note that the choice of splitting criterion depends on the specific problem, dataset characteristics, and the goals of the analysis. Different measures may perform differently under various scenarios. Therefore, it is recommended to experiment with multiple criteria and assess their performance on your specific dataset.

3.4.1.3 Differences between Hellinger distances, Gini index, and information gain

Pruning can lead to better results in decision tree algorithms under certain conditions. Pruning is a technique used to reduce overfitting by removing branches or nodes from the decision tree. It helps to improve the generalization ability of the tree and avoid capturing noise or outliers in the training data. Pruning can be done in two main ways:

Pre-pruning: Pre-pruning involves stopping the growth of the tree early based on certain conditions, such as a maximum depth limit, a minimum number of samples per leaf, or a maximum impurity threshold. It helps prevent the tree from becoming too complex and overfitting the training data.

Post-pruning: post-pruning, also known as tree trimming or cost-complexity pruning, involves growing the full tree and then removing or collapsing nodes in a bottom-up manner. The decision to remove a node is based on a pruning criterion, such as the error rate or impurity increase after removing the subtree rooted at that node. This process helps

simplify the tree by removing unnecessary branches and improving its ability to generalize to unseen data.

Whether pruning leads to better results depends on the specific dataset and tree construction process. However, in some cases, pruning may lead to a slight decrease in performance if the tree is not overfitting or if the pruning process removes important discriminatory features. The effectiveness of pruning depends on factors such as the quality of the pruning criterion, the size and representativeness of the training data, and the complexity of the decision tree.

3.4.2 Part B

3.4.2.1 Why should we set max_depth parameter in Bagging with HDDT so that the base classifiers become a little better than random guess?

Setting the *max_depth* parameter in Bagging with Hellinger Distance-based Decision Trees (HDDT) to be shallow (e.g., 3 or 4) ensures that the base classifiers are slightly better than random guessing. This limitation on the depth of the decision trees helps prevent overfitting and restricts their complexity. By constraining the depth, the base classifiers become simpler and less prone to capturing noise or outliers in the training data. This way, Bagging can leverage the diversity of these slightly better-than-random classifiers and combine their predictions to create a more accurate and robust ensemble model.

3.4.2.2 Stable, Unstable, and Weak classifiers

In the context of ensemble learning:

A **stable classifier** refers to a base classifier whose predictions remain relatively consistent even with small changes or perturbations in the training data. Stable classifiers are less affected by noise or minor variations in the input and tend to produce consistent predictions. Examples of stable classifiers include linear models and shallow decision trees.

An **unstable classifier** is the opposite of a stable classifier. It is highly sensitive to small changes in the training data and can produce significantly different predictions with slight variations in the input. Unstable classifiers are more prone to overfitting and may capture noise or outliers in the data. Examples of unstable classifiers include deep decision trees and complex models such as neural networks.

A **weak classifier** refers to a base classifier that performs only slightly better than random guessing. Weak classifiers have limited predictive power on their own and are often simple or have low complexity. In ensemble methods like AdaBoost.M1, weak classifiers are combined to form a strong ensemble model. Despite their individual weakness, the combination of weak classifiers with appropriate weighting and aggregation techniques can lead to improved predictive performance.

3.4.2.3 Classifiers used in Bagging. Why AdaBoost.M1?

Bagging is a general ensemble method that can be used with a wide range of classifiers. It is particularly effective when combined with high-variance and unstable classifiers, such as deep decision trees or complex models like random forests. Bagging helps to reduce overfitting and increase the stability of these base classifiers by training them on bootstrapped samples of the training data and aggregating their predictions.

On the other hand, AdaBoost.M1 (Adaptive Boosting) is designed to work well with weak classifiers. Weak classifiers are typically simple and computationally efficient models, such as decision stumps (decision trees with a depth of one). AdaBoost.M1 focuses on boosting the performance of these weak classifiers by iteratively assigning higher weights to misclassified samples. This adaptive learning process allows AdaBoost.M1 to create a strong ensemble model that combines the strengths of the weak classifiers. Weak classifiers are preferred in AdaBoost.M1 as they provide a good balance between performance and simplicity, making the algorithm more robust and efficient.

4 References

[1] J. Brownlee, "An Introduction to Grid Search," 2023, [Online]. Available: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/.

[2] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intell. data Anal.*, vol. 6, no. 5, pp. 429–449, 2002.