CSE 176: Evaluating Algorithms for Learning Decision Forests

Mashaallah Moradi, Jason Holmes, Jason Petersen, Apsara Fite

April 2023

Introduction

Ensemble methods have become increasingly popular in the field of machine learning due to their ability to improve the accuracy and robustness of models. Among these ensemble methods, decision forests - which are ensembles of decision trees - have shown promising results in real-world classification and regression tasks. However, the performance of decision forests depends on several factors, such as the choice of algorithm, hyperparameter tuning, and data preprocessing techniques.

In this report, we explore the HistGradientBoosting algorithm for learning decision forests and evaluate its performance on various real-world datasets. We also investigate the impact of hyperparameter tuning and data preprocessing on the accuracy and generalisation of the model. The findings of this study can help practitioners choose the best algorithm and optimization strategy for building decision forests in real-world scenarios, as well as learn more about datasets from models.

Algorithm

In the original gradient boosting algorithm, when we use decision trees to make predictions, we need to sort every feature and find the best splitting point based on a gain function. The algorithm we will be investigating is HistGradientBoosting, which is a variant of the gradient boosting algorithm. Instead of sorting features and evaluating each split point individually, it creates histograms. The first step is binning, where we assign values to samples based on the closest threshold. For example, if a sample is closest to 0, it goes into the bin labelled 0; if it's near 1, it goes into the bin labelled 1; and so on. By doing this, we convert our data from floating-point numbers to integers, which is faster to process.

After binning, we compute the gradients of our samples. These gradients represent the differences between the actual target values and the predicted values from the previous iteration. Once we have the gradients, we can sum them up to create our histograms.

Now, our split points will be the different thresholds we initially set for our bins. These split points will be located between each of our histograms. Building the histogram takes *O(n)* time complexity, where n is the number of bins. Similarly, for every split point, it takes *O(n)* time complexity. This is much faster compared to the traditional gradient boosting method, which requires *O(nlogn)* time complexity to sort the features and *O(n)* for every feature split.

In summary, HistGradientBoosting in scikit-learn speeds up the training process by using histograms instead of sorting features individually. This approach significantly reduces the computational time required for building the histograms and finding the optimal split points.

Parameter tuning

The HistGradientBoosting classifier has several hyperparameters, with the first one being 'max_bins.' This parameter determines the number of bins used to create thresholds for each feature. Increasing the number of bins can improve the accuracy of the sample features, but it may also slow down the process.

Next, there are different parameters specific to both the classifier and regressor. In the case of the HistGradientRegressor, we have the ‘loss’ parameter, which refers to the loss function we want to use. By default, it uses the least squares loss. Another hyperparameter is ‘least_absolute_deviation,’ which helps minimize absolute errors and provides a more accurate mode.

For the HistGradientClassifier, there is the ‘loss’ parameter, which determines whether we are dealing with a binary dataset or multiple classifiers. The ‘binary_crossentropy’ hyperparameter penalizes incorrectly labeled data, incentivizing the model to label correctly. We also have ‘categorical_crossentropy,’ which handles different classes by converting them into integers. Additionally, there is the ‘l2 regularization’ hyperparameter, shared by both the classifier and regressor. It forces the weights towards zero, increasing negative weights and decreasing positive weights, thereby assisting with overfitting.

Moving on to the boosting hyperparameters, we encounter the ‘learning_rate’ parameter. If the learning rate is too small, the model may try to fit to every single point and overfit the data. Conversely, if it's too large, it may skip over important points and underfit the data. The ‘max_iter’ parameter determines the number of trees that will be constructed during the boosting process. ‘Max_leaf_nodes’ specifies the maximum number of leaves each tree can have, while ‘min_samples_leaf’ determines the minimum number of samples required in a leaf node. Lastly, ‘max_depth’ determines the maximum depth that each tree can reach.

Summarily, the hyperparameters of the HistGradientBoosting classifier and regressor involve settings related to binning, loss functions, regularization, boosting, and tree construction. Careful adjustment of these parameters is essential to achieve the desired balance between accuracy, speed, and avoidance of overfitting or underfitting.

Classification Dataset

The dataset used in the classification portion of this project is a set of credit card transactions made by European cardholders in 2013. The transactions in this set took place over the course of two days. This dataset poses a binary classification problem as its data points are labelled as fraud and non-fraud. The vast majority of transactions in this data set are legitimate transactions, making it highly unbalanced. Out of 284,807 transactions, there are 492 cases of fraud. This leaves 99.828% of the data as negative (non-fraud) with only 0.172% of the positive (fraud) class.

The ‘Time’ and ‘Amount’ features are the only two original features left in the dataset. ‘Time’ is the amount of time elapsed since the first transaction had taken place. ‘Amount’ is simply the amount of money spent in the transaction. The rest of the original features are unavailable to us in this data set in order to protect the cardholders’ confidentiality. There are 28 of these features, V1, V2, … V28 for each transaction. They are numerical input variables which are the result of a PCA transformation for which we are given no background information. The only other feature is ‘Class’ which is binary and states whether the transaction was fraud or not. It has value 1 if the transaction is fraud and 0 if not.

Processing Data

The dataset is in the form of a .csv (comma separated values) file with a shape of 31 columns by 284,808 rows. This is 31 features for each of the 284,807 transactions plus a header row. The first row of the csv file is separated from the rest of the data as the header has no use for our training. The raw data without the headers is then split on the last column in order to have separate arrays for our x, inputs, and y, true classification values.

For training a model on the original data set with the class imbalance, sklearn's train_test_split is used to randomly split the raw x and y data into training x and y and testing x and y. The size of the test set is set to 25% of the original dataset. This leaves us with 4 arrays: training x, training y, testing x, and testing y where the ratio of training to testing is 3:1. There is no need to manually separate a validation set since k-folds cross validation is used during parameter tuning.

In order to deal with the class imbalance during the data processing portion, we use sklearn's resample to change the sizes of the different class data and make them equivalent. Three different configurations of resampled data are created with different amounts of each class. This is carried out after the test set is separated out from the data so as to not skew the results of the test. Each of these resamplings were carried out with replacement rather than random permutations. In order to resample the data, the positive data points needed to be separated out from the negative datasets so that there are two arrays, one for fraud data and one for non-fraud data.

The first resampling of the data was an oversampling of the minority data set. This resamples all of the transaction points marked as fraud to be a size of 213,243 which is the size of the majority class's training set. Resampling the data in this way preserves all transaction data while balancing the classes by using replacement to increase the number of fraud cases from 492 to 213,243. This nearly doubles the size of the dataset which will cause significant increases in run time for fitting our model. Since this oversampling creates copies of the minority class, it may also increase the possibility that overfitting will take place.

Inversely, we can resample the majority class to the size of the minority class. This resizes the entire data set to less than 0.4% of the original set. This change removes the point of using the histogram gradient boosting classifier over other gradient boosting classifier algorithms since the histogram gradient boosting classifier's advantage is that it provides results similar to other gradient boosting classifiers at a faster run time in large data sets. Undersampling this majority class also removes most of the negative class data, which may result in a less accurate model. Overfitting and poor generalisation are also more likely with the undersampled data set. If the results of the undersampled model are effective enough, this set could be used to very quickly search through parameter lists for the best combination.

A middle ground between oversampling or undersampling just one of the classes can be found by undersampling the majority class to half its original size and oversampling the minority class to the same size as the majority. This way only half of the majority class data is lost and the size of the set stays near its original size. This can still be less effective than oversampling since data is lost, but it won't take nearly as long to train a model on it. Below is a table showing the amounts of each class for each data set created.

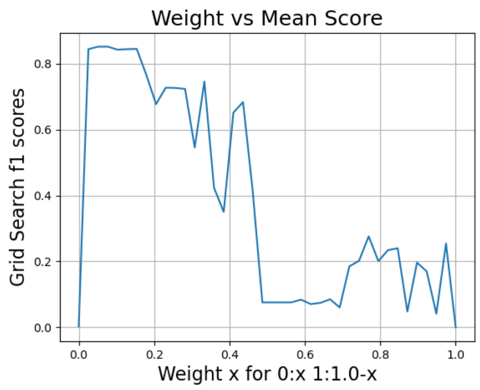
Data Sets	**Fraud Amount**	**Non-fraud Amount**	**Total**
original	362	213243	213605
oversampled minority	213243	213243	426486
undersampled majority	362	362	724
undersampled/oversampled	106620	106620	213240

Parameter Tuning

For the parameter tuning, we used sklearn's GridSearchCV to perform an exhaustive search over given parameter options. This function takes a list for each parameter and fits the model to every possible combination. Each model is scored using k-folds cross validation during the grid search. The best set of parameters is chosen based on the average scores of the cross validations. The amount of folds in the cross validation was set to three.

The scorer used to evaluate the models created is the F1 scorer. Because of the class imbalance, we need to make sure that the model is evaluated based equally on recall and precision. Otherwise the model could be evaluated as being highly accurate even though it misclassified most of the fraud cases. The F1 score equation is as follows: *F1 = 2 * (precision * recall) / (precision + recall)* where *precision = True Positive/(True Positive + False Positive)* and *recall = True Positive/(True Positive + False Negative)*.

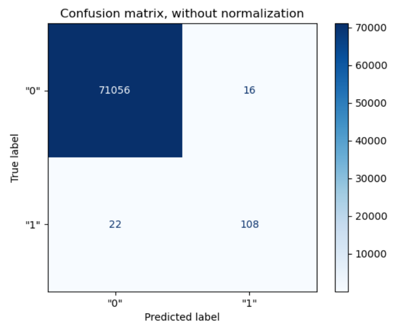
If we use the original data without any resampling, the class weights must be adjusted to compensate for the class imbalance. Since only 0.172% of the data points are fraud, the class weights must be set so that the ratio of fraud weight to non-fraud weight is very large. This will modify the algorithm to take into account this class imbalance without needing to modify the data beforehand. The parameters for a grid search focusing on the class weights are {0: x. 1: 1-x} for 20 x values in the range of 0 - 1. x corresponds to the weight for the negative class and 1 - x is the weight for the positive class. In the plot below, we can see how the model becomes much less accurate as the weights are adjusted from higher for the positive class to higher in the negative class.

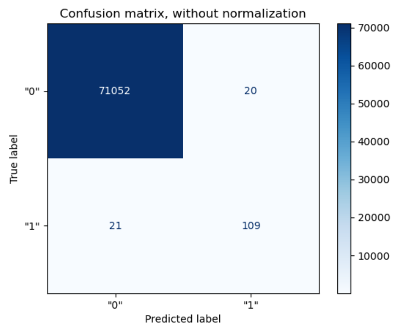


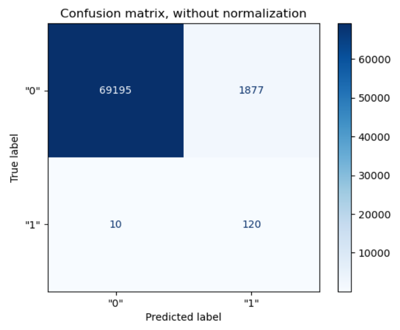
The best weight found is outside of the range we searched through in this grid search. The most effective weighting for the model was found to be calculated as (# samples)/(2 * #

of class samples). This is the weight formula used when setting weights parameter to 'balanced'.
Negative class weight = 0.5007
Positive class weight = 289.438

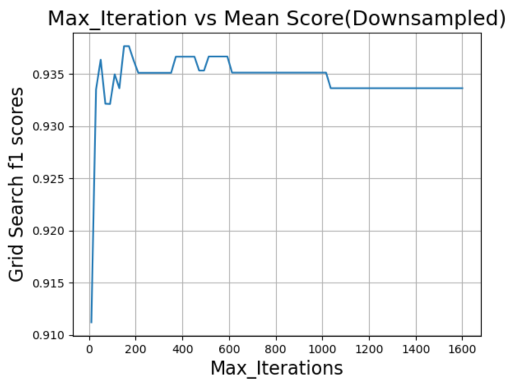
When resampling the data, the most accurate data set was the upsampled majority class. This sampling was only slightly more effective than doing the under/oversampling mix. The undersampled data set was clearly the worst of these options. The confusion matrices below show in order from left to right, upsampled model, mixed sampling model, and downsampled model.

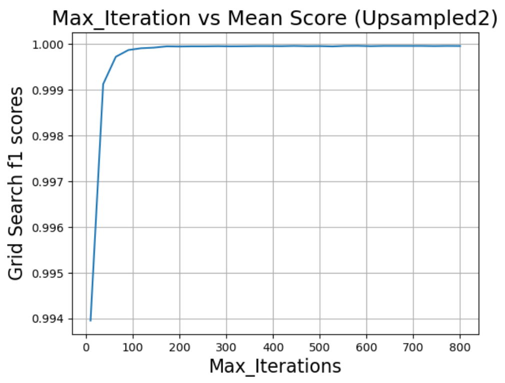




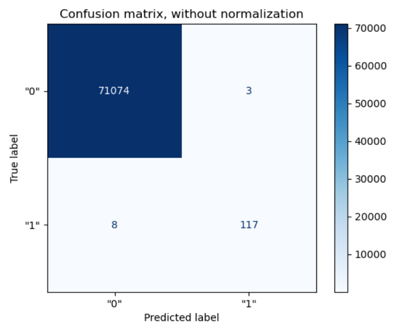


Also of note is the way the undersampled model appears to be more prone to overfitting than the other models. In the plots below, we can see the undersampled model seems to begin overfitting around 600 iterations while the oversampled model does not have any visible overfitting through 800 iterations.





The most accurate way to deal with the class imbalance in practice was the class weights set to 'balanced' using the original data set with no resample. This confusion matrix shows the results of the 'balanced' weights on a model which correctly classifies more of the test points than any of the resampled models. In addition to being the most accurate, it also does not take nearly as long to fit the model on original data when compared to data with oversampling.



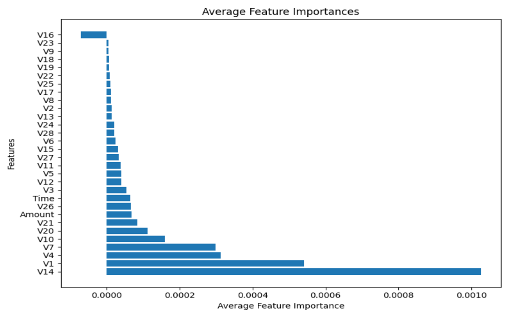
Because the 'balanced' class weights option was found to be more accurate than resampling in this situation, the rest of parameter tuning is carried out with these weights and the data set with no resampling.

The following parameter list was passed to grid search for parameter tuning over the balanced weight model: {'learning_rate': [0.01, 0.1, 0.5, 1], 'max_iter': [300, 400, 500, 600, 700, 800], ‘max_leaf_nodes’: [None, 15, 31, 50], ‘l2_regularization’: [0.5, 1]}. With the grid search set to 3 k-folds cv, it took a total of 30 minutes to search through each combination of parameters. The best F1 score was found to be 0.87606 with the parameters shown in the table below.

Learning Rate	max_iter	max_leaf_nodes	l2_regularization	F1 score
0.1	500	50	0.5	0.87606

Credit Card Fraud Feature Importance

Using permutation feature importance alongside our HistGradientBoosting classifier we were able to rank all the features based on how positively or negatively they affected our accuracy. The way permutation feature importance makes its calculations is by hiding a random feature or not taking it into account when the model is training and after doing so we compare our accuracy of when we trained without that feature to our accuracy when we were training without that particular feature being hidden or ‘shuffled’, if our accuracy increases it is likely due to that feature effecting our accuracy negatively and if the accuracy decreases then it is also likely that the feature we shuffled was positively affecting the accuracy.

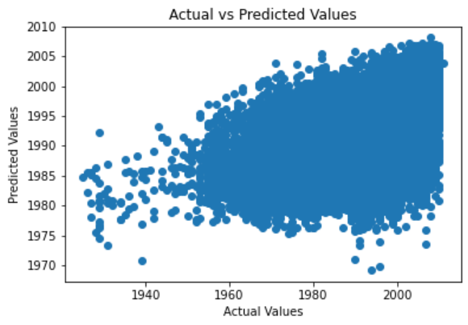


Here is a graph of our feature importances after 100 iterations or trees. It is important to note that as we increased the number of trees, support for the V14 feature increased significantly. As we are not given the actual names of each feature due to security concerns, we will not know what exactly needs to be changed, however we can look at the difference between the fraudulent vs non-fraudulent V14 values and compare.

When we extracted all V14 fraudulent values and compared them with the same number of randomly selected non-fraudulent V14 values, we noticed that our fraudulent values were 6.748 less than the non-fraudulent transactions. While this is an interesting find, we are unsure of the significance since we don't know what V14 stands for or the meaning behind their values.

Regression Dataset

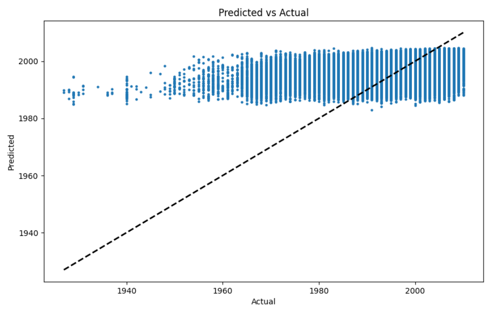
For our regression dataset we were assigned the YearPredictionMSD dataset, which is a subset of the million songs dataset. The attribute information for the dataset are as follows, the first row is the year each song was released which ranges from 1922 to 2011 followed by a 12 timbre average and 78 timbre covariance features which were extracted using the echo Nest API. We extract ‘segments’ from every song which are 12 dimensional timbre vectors, next we take the average and covariance of those vectors. This particular dataset was heavily skewed towards the early 2000s and does not have many samples from the 1960s and earlier. So our model was able to correctly identify songs for later years, however would not be able to do so for earlier years. Our training split for the 515,345 samples in the given dataset was 80% for training, 10% for validation/hyperparameter tuning, and 10% for testing. It is also important to mention that no same song appears on both the training and test sets to avoid the producer effect.

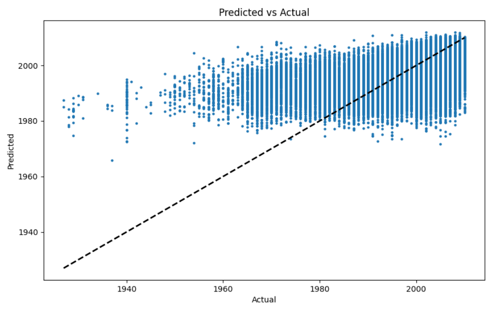


As we can see here from our testing samples the model predicts mostly in the range of 1980-2010 regardless of the datapoint.

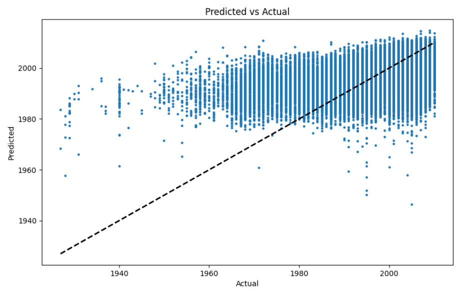
Parameter Tuning

For our parameters, our most significant finding was that with a low learning rate of 0.001, most of the songs would be classified as a more recent year than their true release year. On average, songs before 1960 were predicted as 30 years more recent than they truly were. Conversely, when we use larger learning rates such as 0.1, we get correct predictions for songs released in earlier years at the cost of prediction accuracy for later years.

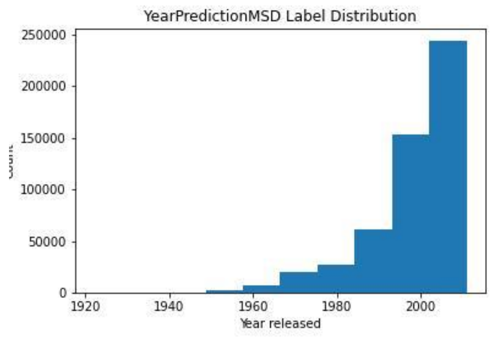




As we can see on the left we have a prediction vs actual graph with 0.001 learning rate and on the right we have 0.1 learning rate. A learning rate of 0.1 gives us our best r-squared value of 0.32; if we go any higher than this, we get too many misclassifications for the later years. See the following graph for predictions with a 0.5 learning rate.



The reason why our predicted year has such a tight range (most predicted points are from 1980s-200s) is due to our training data being over-saturated with songs from the 1990s-2000s. See below for the distribution of labels in the dataset.



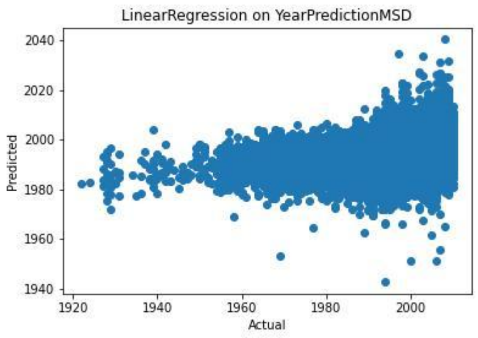
There are a disproportionate amount of songs labelled with those years, and thus our model isn't being trained fairly. It will be more likely to predict that a given song was released in those later years. The following is a table with our best hyperparameters.

learning_rate	max_depth	max_iter	min_samples_leaf	max_leaf_nodes	R-squared	MSE
0.1	6	1000	20	100	0.32	98.773

As we can see, our r-squared score of 0.32 suggests our trained model was unable to predict the year that a certain song came out by using the timbre average and covariance features. This is further reinforced by our mean-squared error of 98.773 years squared, which is very high and suggests that the regression model rarely predicts the year a song was released correctly.

Comparison to Linear Regression

We decided to compare the performance of HistGradientBoosting regressor with the basic Linear Regression model. First though, the YearPredictionMSD had to be preprocessed for convergence. The data was preprocessed using StandardScaler standardisation.



As expected, the linear regressor was computationally faster than the HistGradientBoosting regressor, at the cost of accuracy. The r-squared score for the linear regressor was 0.2424 and the mean squared error was 104.323. With both a higher R2 score and a lower error than the linear regression model, the HistGradientBoosting regressor does find a marginally better fit for the data.

Conclusion

From tuning the parameters of the HistGradientBoosting classifier on the fraudulent charge dataset, we found that the F1 score was the most useful because it equally takes into account misclassification of positive and negative classes despite the class imbalance. Our findings from the classifier model indicate that feature V14 of the dataset may be indicative of a fraudulent charge, though we are unsure since we don't know what that feature means. Additionally, it is unlikely that this model will be accurate for predicting fraudulent card charges in the long term. Firstly because the training dataset only spans credit card charges over a 2-day period, but also because fraudulent charge behaviour will change as time goes on. Given that credit card companies will adjust their system to prevent fraudulent charges based on the findings of the model, fraudsters will likely adjust their behaviour to continue evading the system. Thus, it is within companies' best interest to repeatedly fit data to match real-time behaviour.

As for the YearPredictionMSD, not much correlation could be found between the year a song was released and the predicted year. This may be because songs sound quite different even within one year; comparing rock from the 1980s to 2010s pop doesn't make much logical sense. It may be a better idea to fit data from individual genres of songs across the years, like comparing rock from the 1980s through 2010s. Separating the dataset by genre may result in a stronger correlation being found, and the individualised regression models will likely be more accurate.