Hands-On Tree-based Models for Classification

Paula Iborra, Marcel Pons

April 23, 2021

1 Introduction

In this section, different Tree Based Models for classification will be performed upon the **Heart Disease Dataset**, which contains several continuous and categorical variables that can explain whether or not a patient has a heart disease. The original dataset provided in the Machine Learning Repository contains 76 attributes, however here 14 attributes will be used, which are considered the most relevant predictors for the disease. Besides, missing data is already handled (either by dropping certain rows or by means of imputation techniques) and the response variable *hd* is binarized to 0-1 instead of having 4 levels as the original dataset has.

After all modifications, the used dataset contains 303 rows and 14 attributes, which are:

Column name	Description
age	patient's age
sex	male / female
cp	chest pain
restbp	resting blood pressure (in mmHg)
chol	serum cholesterol in mg/dl
fbs	fasting blood sugar
restecg	resting ECG results
thalach	maximum heart rate achieved
exang	exercise induced angina
oldpeak	ST depression induced by exercise relative to rest
slope	the slope of the peak exercise ST segment
ca	number of major vessels (0-3) colored by floroscopy
thal	thalium heart scan
hd	diagnosis of hear disease (response variable)

2 Exploratory Data Analysis

The first step before the model building is performing EDA techniques to understand and summarize the Heart Disease Dataset.

Our dataset contains more than twice males than females. By analysing the correlations between features, apparently, there is no significant linear correlation between continuous variables as it can be appreciated in Figure 1. However, the heatmap matrix suggests that there might be some positive correlation between the heart disease (*hd*) and the chest pain (*cp*) and the maximum heart rate achieved (*thalach*). Contrarily, the correlation heatmap matrix indicates a negative correlation between our target variable *hd* and the number of major vessels (*ca*), the exercise induced angina (*exang*) and the ST depression induced by exercise relative to rest (*oldpeak*). It is intuitive to first think that elder people might have higher chances of heart disease, but according to the distribution plot of age with respect to target variable, it is ambiguous that there exist a slight positive correlation. All these relations motioned can also be be noticed in the pair-plot (Figure 2) distributions of selected features against heart disease.

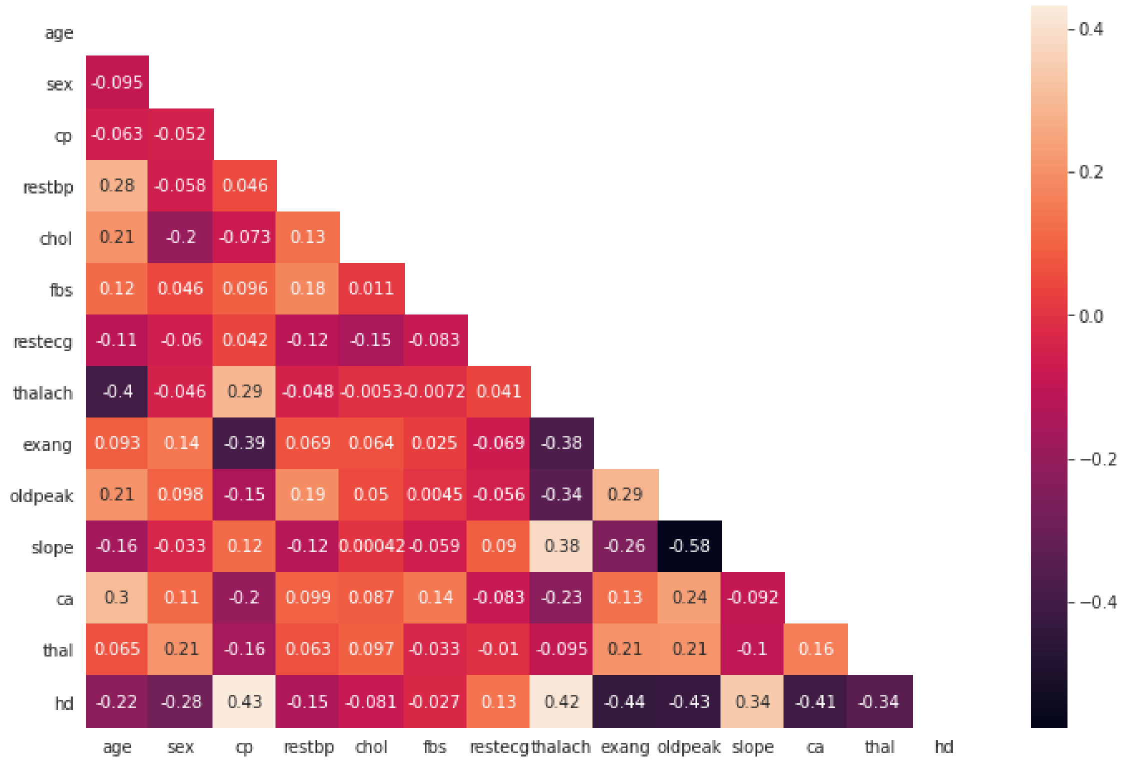


Figure 1

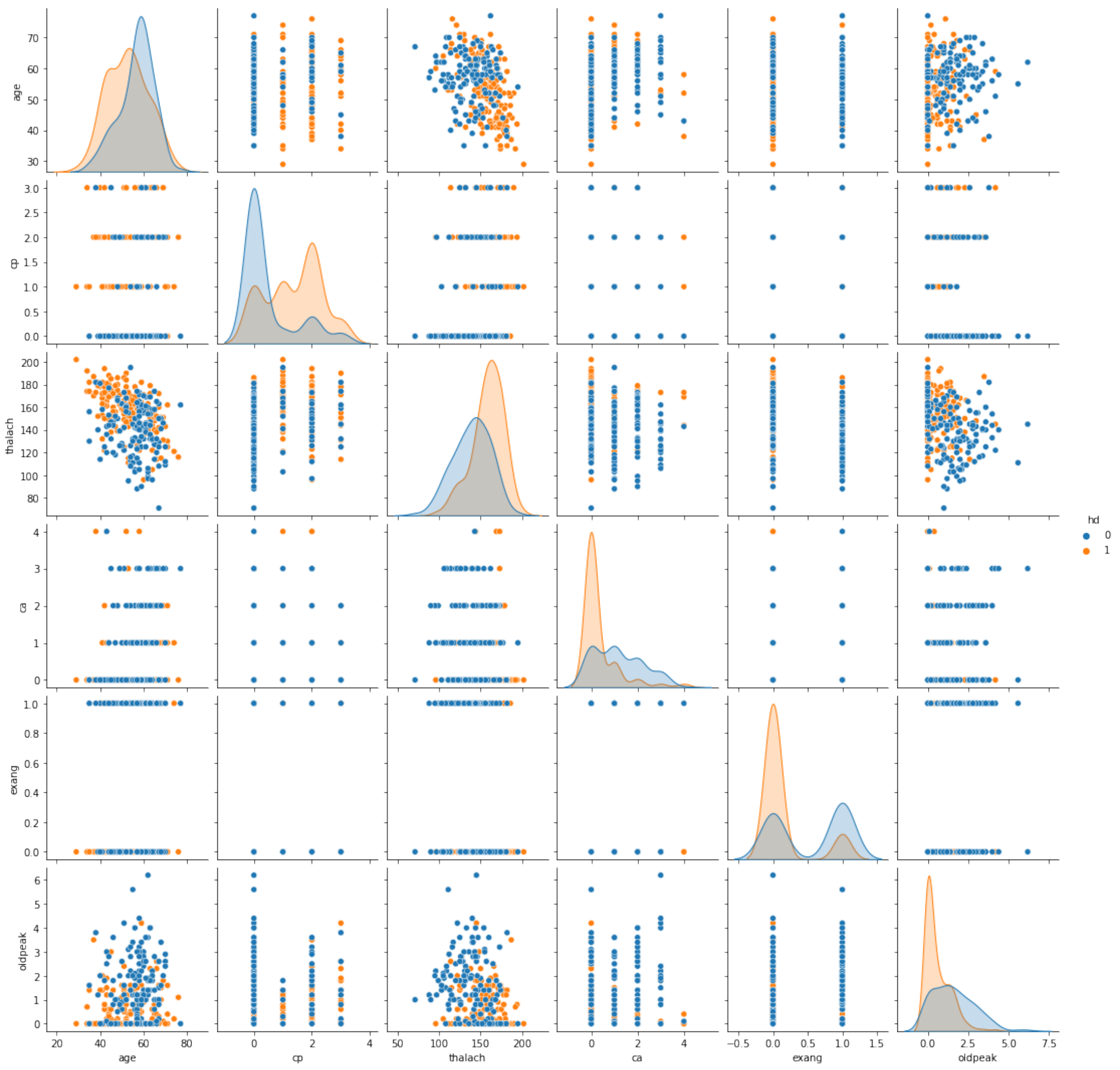


Figure 2: Pair-plot of selected features with respect to target variable hd.

By analysing the target variable *hd* it can be appreciated from Figure 3 that the dataset contains a similar number of observations for both target categories. Therefore, the target variable is balanced. This is an important fact to take into account when splitting the dataset into train and test sets, since if unbalance is present the split has to be stratified in order to distribute the observations evenly between sets, otherwise problems would arise when performing machine learning models.

Furthermore, this information is also relevant when it comes to decide which score to be used for evaluating the performance of the models. If imbalance is present, the accuracy score is not a good option because a high score will be biased by simply predicting that all observations belong to the majority class.

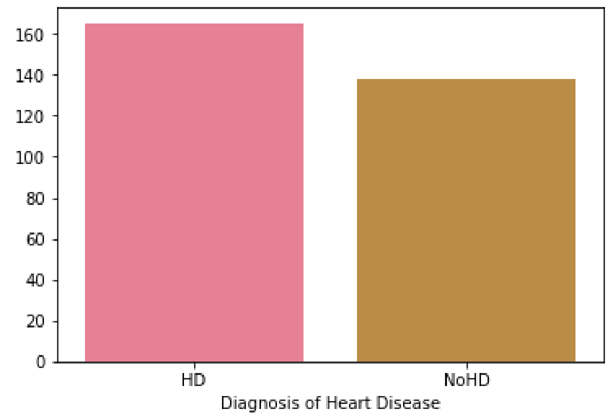


Figure 3: Distribution of the levels of the target variable

3 Data Preprocessing

When inspecting the data types of the dataframe, some columns, such as *cp* and *slope*, have a numeric data type instead of being considered categorical. These columns contain an *int* data type although they should not to be used as integers but as different levels of a category (e.g. in *slope*, 1, 2, 3 refer respectively to *unslopping*, *flat* and *downslopping*, not ordinal integers).

The aforementioned columns have to be handled because the sklearn models that are going to be implemented do not natively support categorical data. Therefore, the columns with more than 2 levels have to be converted into multiple columns of binary values, named One-Hot Encoding[1].

One Hot Encoding can be performed both with sklearn `OneHotEncoder` or pandas `get_dummies()`. Since further feature engineering with sklearn and pipelines are not going to be used, it is preferred to use the later method, which in some way is simpler and stores the output in a dataframe rather than in an array.

```
df_encoded = pd.get_dummies(df, columns = ['cp', 'restecg', 'slope', 'thal'])
```

In many machine learning models, further preprocessing typically takes place, such as feature selection and feature scaling. On this dataset, feature scaling is not considered since tree based models do not make assumptions on relationships and distributions, such as co-linearity, normality or homoscedasticity. Nonetheless, often data sets contains high amount of features including non-informative ones. Our goal is to create a model that only includes those features more important to predict our target value. This has a significant impact in reducing the variance of our model, and thus the overfitting, it also creates a more simple model to interpret, and it reduces the computational cost (run time) of training a model. The process of identifying only the most relevant features is called feature selection. Tree-based methods are often used for this propose

[1]Dummy encoding could be also considered, which instead of generating *k* variables from a column with *k* levels, it generates *k-1* variables, enclosing one level in the intercept, thus reducing the co-linearity that may arise between new variables. Since tree-based models follow non-parametric approaches, no assumptions on relationships are made.

due to the fact that the these methods naturally rank by how well they improve the purity of the node. This mean decrease in impurity over all trees (called *gini* index, which tells us what is the probability of misclassifying an observation). Nodes with the greatest decrease in *gini* index happen at the start of the trees, while nodes with the least decrease occur at the end of trees. Thus, by pruning trees below a particular node, we can create a subset of the most important features. We will use this technique when building our models in order to try to simplify them.

4 Model Building

Once all the necessary preprocessing is done, the dataset is split into train and test sets using `train_test_split` from sklearn. The conventional notation on machine leaning is used, in which `X` represents the explanatory variables and `y` the target variable. The data is split into 20% test set and 80% training set, resulting in 242 observations on the former and 61 on the later.

```
X_data = df_encoded.drop('hd', axis=1)
y_data = df_encoded[['hd']]

X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, test_size=0.2)
```

4.1 Decision Trees

The first model that is built corresponds to a classification tree using `DecisionTreeClassifier` from `sklearn.tree`. Classification trees are an exceptionally useful machine learning method when it is needed to know how the decisions that are being made.

Firstly, a classification tree is built without specifying any parameter.

```
clf_dt = DecisionTreeClassifier()
clf_dt.fit(X_train,y_train)
```

We obtain a tree of depth 10 that yields 100% training accuracy and 75.4% test accuracy, consequently meaning that the model has overfit the training data. For this reason, pruning needs to be applied to the model in order to make it perform better with unseen data.

Overfitting in decision trees can be reduced by optimizing parameters that are designed to reduce it, such as `max_depth` and `min_samples` —which respectively control the maximum depth of a tree and minimum number of observations a leaf can have—, thereby avoiding trees to grow in the direction towards isolating unique observations and eventually overfitting the data. These parameters can be optimized by using Cross Validation and Grid Search. Nevertheless, cost complexity pruning is also an effective way to find the best tree and can simplify the process of finding a smaller tree.

Pruning a decision tree consists on finding the right value for the pruning parameter `alpha`[2] which controls the extend on how much pruning happens. In the Figure 4 it is appreciated that as the value of alpha increases, both the number of nodes and the depth of the tree decreases. That reduction stems from the fact that the tree gradually gets smaller until it reaches a point where the tree corresponds only to the root node.

```
# Determine values for alpha
path = clf_dt.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas = path.ccp_alphas
ccp_alphas, impurities = path.ccp_alphas, path.impurities
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)

clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1] # ommit maximum value

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
```

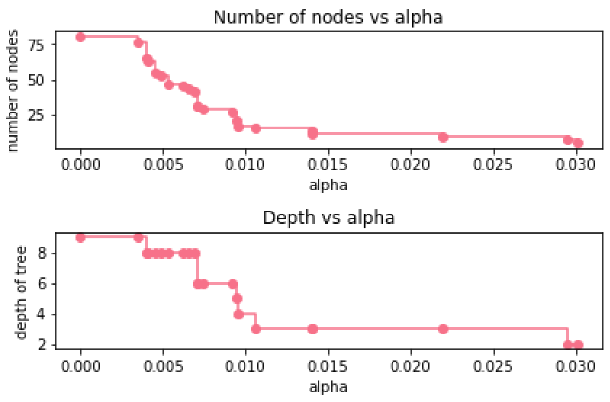


Figure 4: Effect of `alpha` on the number of nodes and depth of the tree.

By means of graphing the accuracy of the trees using the training set and the testing set as a function of `alpha` the best value for the parameter can be found. When `alpha` is set to zero, the tree overfits, leading to a 100% training accuracy and 64% testing accuracy. As `alpha` increases, more pruning takes place, thus creating a decision tree that generalizes better. Here, setting `alpha=0.023` maximizes the test accuracy.

[2] As a reminder `alpha` is a complexity parameter that is taken into consideration in order to penalize large trees. It influences the Cost Complexity Function $C_\alpha(T) = R(T) + \alpha|T|$, which quantifies the capacity of a tree to predict accurately the data in terms of missclassification rate.

```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]
```

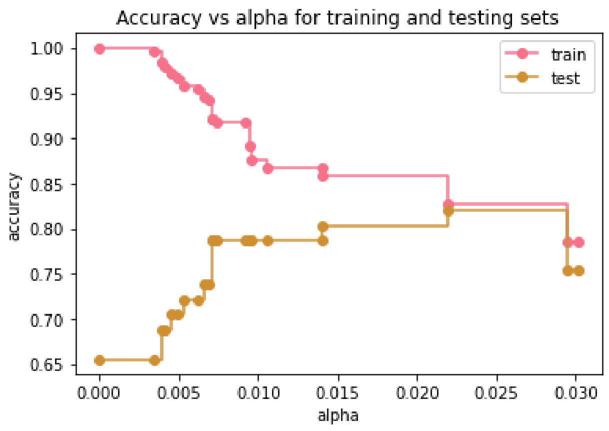


Figure 5: Effect of parameter `alpha` on training and test accuracy.

Figure 5 only has illustrative purposes since values on the parameters cannot be optimized from the test set. For this reason, cross validation is used to find the best value for `alpha` that generates trees flexible enough to classify accurately unseen data. To that end, 5-fold cross validation is performed where the average accuracy and average standard deviation are stored for each value of `alpha`.

```
alpha_values = []

for a in ccp_alphas:
    clf_dt = DecisionTreeClassifier(random_state=0, ccp_alpha=a)
    scores = cross_val_score(clf_dt, X_train, y_train, cv=5)
    alpha_values.append([a, np.mean(scores), np.std(scores)])
```

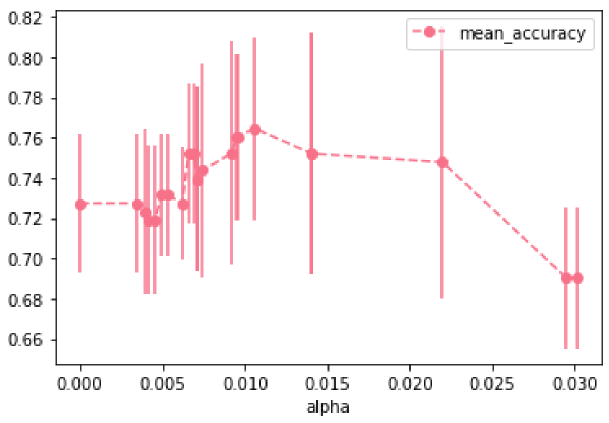


Figure 6: Result of 5-cv for the value of `alpha`

By using the `alpha` that generated the maximum mean accuracy, a decision tree classifier pruned with this value is built.

```
clf_dt_pruned =DecisionTreeClassifier(ccp_alpha=ideal_ccp_alpha)
clf_dt_pruned.fit(X_train, y_train)

y_pred = clf_dt_pruned.predict(X_test)
```

After pruning, the test set accuracy has increased from 63.9% to 78.7%, which can be considered a significant improvement. By representing the classification of the test observations in a confusion matrix, it can be seen that of the 22 + 8 = 30 people that did not have HD, 22 (73%) were correctly classified. And of 5+26=31 people that have the disease, 26 (83%) were corretly classified.

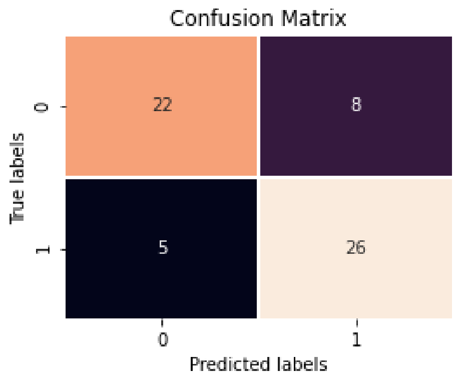


Figure 7: Confusion matrix for the improved Decision Tree Classifier.

The final representation of the optimized tree is illustrated in Figure 8. It corresponds to a tree of depth 4 —in contrast to depth 10 of the complete tree previously built— that considers the variables *thal*, *cp*, *oldpeack*, *ca* and *thalach* as the optimal features to split the data and obtain the maximum information gain. Among these variables, thalium heart scan (*thal*) is the variable that splits the 242 samples of the root into 131 and 111 samples in the new nodes, therefore it can be considered the most important variable to determine the presence of heart disease in the decision tree.

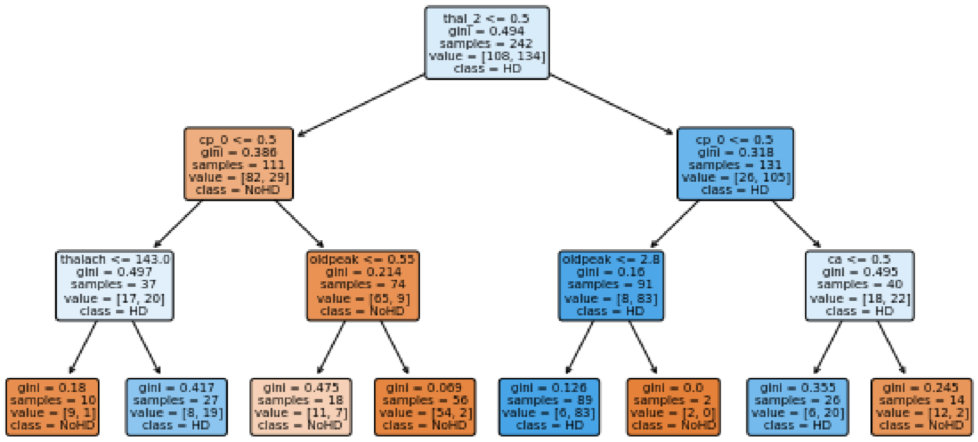


Figure 8: Tree representation

4.2 Random Forest

Random forest is a tree-based algorithm composed of many decision trees where their outputs are combined to enhance the performance of a given model. Here, a random forest classifier is going to be implemented using `RandomForestClassifier` from `sklearn.ensemble`. As done previously, we fit a random forest without specifying any parameter[3].

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
```

The resulting model has a 78.7% test accuracy. However, by optimizing the parameters of the model an improvement on the classification capacity can be achieved. Grid Search is used to find the optimal hyperparameters for the model, which through an exhaustive search through a manually specified subset of the hyperparameter space are found. The `f1` score is used as criteria to find the best hyperparameters and 5 folds of hold out[4] data are used in each iteration. The considered hyperparameters are:

Hyperparameter	Definition
`bootstrap`	Whether bootstrap samples are used when building trees.
`n_estimators`	Number of trees in the forest
`max_features`	The number of features to consider when looking for the best split
`min_samples_split`	The minimum number of samples required to split an internal node.
`min_samples_leaf`	The minimum number of samples required to be at a leaf node.

```
params_rf = {
    'bootstrap': [True, False],
    'n_estimators': [25, 50, 100, 350],
    'max_features': ['log2', 'auto', 'sqrt'],
    'min_samples_split': [2, 3, 4],
    'min_samples_leaf': [2, 10, 30]
}

grid_rf = GridSearchCV(estimator=rf,
                      param_grid=params_rf,
                      scoring='f1',
                      cv=5,
                      n_jobs=-1)

grid_rf.fit(X_train, y_train)

best_rf = grid_rf.best_estimator_
```

[3]By default, the model will consider 100 estimators, use the *gini* index as split criterion, minimum samples for split and for leaves at its minimum, and the other parameters to *None*.

[4]Out-of-Bag (OOB) data can also be used for validation, which consists on part of the training data that does not end up in the bootstraped dataset, therefore not used when building the trees. Ultimately, it can be used to measure how accurate the model is given the Out of Bag Error (OOE). For the sake of consistency between the different models applied, cross validation is used.

The best random forest classifier consists on an ensemble of 100 trees, with 2 minimum samples required for split and a minimum of 30 observations on each leaf a minimum sample for leaf of 30. The number of features considered for the best split, `max_features` equals $log_2(\text{n\_features})$.

With this configuration of hyperparameters, the random forest classifier accomplishes a 85.2% accuracy score. By looking at the confusion matrix, it can be appreciated that the classifier only returns 1 false negative and 8 false positives, with a 96.7% sensitivity and 73.3% specificity.

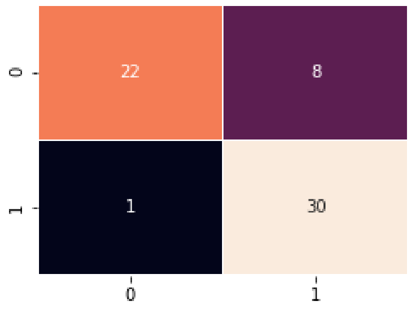


Figure 9: Confusion matrix for the best Random Forest Classifier

The scores showed in Figure 10 are the importance scores for each of the variables considered in our model that add up to 1. As it can be noted from the result values, some features do not contribute with any importance to the model, or its importance is extremely low. Therefore, we have decided to simplify our model by performing feature selection and avoid including in the model those variables that do not contribute to it.

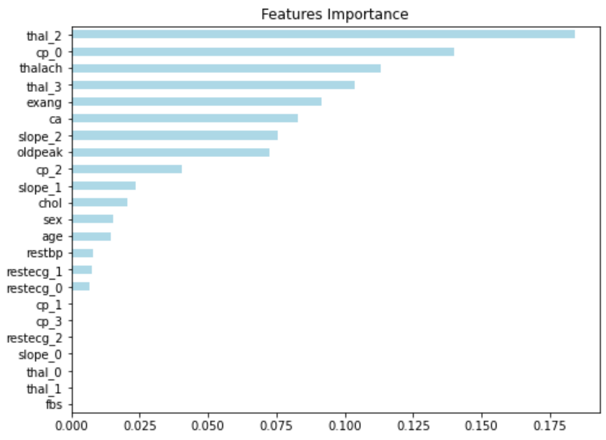


Figure 10: Variable Importance

We have used the `SelectFromModel` built-in function, as shown in the code below, that uses the best random forest classifier to identify features that have a significant importance to the model building. With a cutoff of 0.08 we retain the 6 most important features. Overall, the reduced feature model has no relative accuracy increment compared with the full feature model, however the amount of data used for the prediction has been reduced significantly. In this case, run time is inconsequential because of the small size of the data set, but in bigger datasets, this trade-off likely would be worth it.

```
# Selector that will use the random forest classifier to identify features that have␣
↪an importance of more than the threshold
# (0.08 keeps the accuracy while reducing the number of variables required in the␣
↪model to 6)
sfm = SelectFromModel(best_rf, threshold=0.08)
# Train the selector
sfm.fit(X_train, y_train)
# Transform the data to create a new dataset containing only the most important␣
↪features. We apply the transform to both the training X and test X data.
X_important_train = sfm.transform(X_train)
X_important_test = sfm.transform(X_test)
# Create a new random forest classifier for the most important features
best_rf_im = RandomForestClassifier(bootstrap=False, max_features='log2',
                       min_samples_leaf=10, n_estimators=50, random_state=177)
# Train the new classifier on the new dataset containing the most important features
best_rf_im.fit(X_important_train, y_train)
# Apply The Featured Selected Classifier To The Test Data
y_important_pred = best_rf_im.predict(X_important_test)
```

4.3 Boosting

In this section we will build the model based on two of the most popular boosting algorithms, Gradient Boosting and AdaBoost. The technique of Boosting uses various loss functions. In case of AdaBoost, it minimises the exponential loss function which can make the algorithm sensitive to the outliers. With Gradient Boosting, any differentiable loss function can be utilised.

Although there are some differences between the two boosting methods, both the algorithms follow a similar procedure. Both algorithms improve the performance of simple weak learners by repeatedly shifting their focus to problematic observations that are difficult to predict.

4.3.1 AdaBoost

Firstly, an AdaBoost classification model is built without imputing any hyperparameter.

```
dt = DecisionTreeClassifier(max_depth=3)
ada = AdaBoostClassifier(base_estimator=dt)
ada.fit(X_train, y_train)
```

This model performs with a 78.7% test accuracy. Similarly, as in previous section, the model is optimized by performing Grid Search. With the tuning of the hyperparameters illustrated in the code below, the AdaBoost classifier accomplishes an increased accuracy of 82%, which gets close to the best performance achieved so far by the improved random forest classifier.

```python
params_ada = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.5, 1]
}

grid_ada = GridSearchCV(estimator=ada,
                      param_grid=params_ada,
                      scoring='f1',
                      cv=5,
                      n_jobs=-1)
grid_ada.fit(X_train, y_train)
best_ada = grid_ada.best_estimator_
y_pred = best_ada.predict(X_test)
```

As in previous models, we will simplify our model by feature selection with the same procedure explained in 4.2 section. With AdaBoost classifier the reduction on the features for building the model has a significant impact on the algorithm performance, improving the accuracy up 85.2%, retaining top 16 features.

4.3.2 Gradient Boosting

For Gradient Boosting classification a model without any hyperparameter tuning is built, which accomplishes a test accuracy of 80.3%. By means of Grid Search optimization no improvement in the accuracy has been achieved. No increment in the performance has been obtained by feature selection procedure, however, we have build a simpler model mantianing the same accuracy while reducing the features to 11.

```python
gb = GradientBoostingClassifier()
gb.fit(X_train, y_train)
```

4.4 Extreme Gradient Boosting

Extreme Gradient Boosting models can be implemented by means of `xgboost` API, which has a similar method and attribute implementations as `sklearn`. An XGBoost classification model is firstly built without tuning any hyperparameter. Here, `early_stopping_rounds` is set to 10 to avoid building tress when they no longer improve the predictions. Besides, the `eval_metric` parameter is set to `'logloss'` to use the *negative log likelihood* as loss function, from which the gradients will be computed.

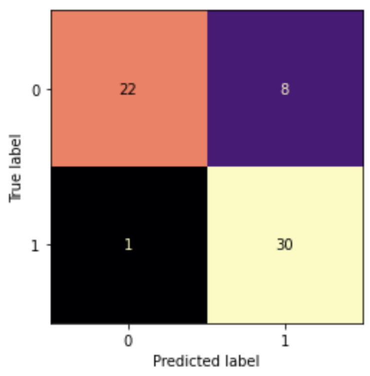
```
clf_xgb = xgb.XGBClassifier(objective='binary:logistic', missing=None)
clf_xgb.fit(X_train,
            y_train,
            verbose=0,
            early_stopping_rounds=10,
            eval_metric='logloss',
            eval_set=[(X_test, y_test)])
```

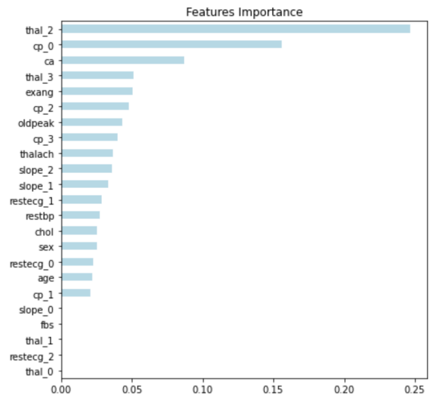
This model performs with a 83.6% test accuracy, which approximates to the maximum accuracy achieved so far, which corresponded to the optimized random forest. Optimization of this model can also be carried out by performing Grid Search. The hyperparameters considered to optimize are:

Hyperparameter	Definition
`max_depth`	Maximum depth of a tree.
`learning_rate (eta)`	Step size shrinkage used in update to prevent overfitting
`gamma`	Minimum loss reduction required to make a further partition on a leaf node of the tree.
`reg_lambda`	L2 regularization term on weights.
`scale_pos_weight`	Control the balance of positive and negative weights

The final hyperparameters are depicted in the code below, in which we also specify `subsample = 0.9` to use a random subset of data (90%) used for every tree and `colsample_bytree=0.7` to use a random subset o columns. These parameters, besides speeding up the process of training, also prevent overfitting.

```
clf_xgb = xgb.XGBClassifier(objective='binary:logistic',
                            gamma=1,
                            learning_rate=0.05,
                            max_depth=5,
                            reg_lambda=10,
                            scale_pos_weight=1,
                            n_estimators = 100,
                            subsample = 0.9,
                            colsample_bytree = 0.7,
                            seed=1717)
clf_xgb.fit(X_train,
            y_train,
            verbose=0,
            early_stopping_rounds=10,
            eval_metric='logloss',
            eval_set=[(X_test, y_test)])
```





With this improved configuration of hyperparameters, the XGB classifier accomplishes a 85.2% accuracy score. As shown in the confusion matrix, it can be appreciated that the classifier only returns 1 false negative and 8 false positives. Regarding the feature importance plot illustrated above, it can be appreciated that 5 variables are not considered when building the tree, and *thal_2*, which corresponds to *fixed defect* on thalium heart scan, it the most important variable to split the nodes. This importance is also observed in the previous models. A simpler model is built performing feature selection, which shows that by only retaining the top 5 features XGB classifier maintains its accuracy of 85.2%.

5 Conclusions

Ensemble algorithms, especially those that use decision trees as weak learners, have multiple advantages over other machine learning methods. These algorithms are non-parametric and thus they do not assume or require the data to follow a specific distribution, saving time of transforming the original dataset by an exhaustive preprocessing of scaling and normalizing data. They are also robust against overfitting. As a result of using many weak learners that underfit (high bias) and combine those predictions into a stronger learner, the variance of the model is reduced.

Another advantage of tree-based methods is that feature multicollinearity does not affect the accuracy and predictive performance of the models. Therefore, there is no need to delete features or perform feature engineering to reduce the correlation and interaction between them.

Robustness against outliers and noisy data is relatively well achieved by means of these methods. Usually, they will deal with noisy data (e.g. features that have no effect on the target) or outliers (e.g. extreme values), with little impact on overall performance.

Regarding the results of the accuracy obtained by all of our built models shown in Table 1 , Random Forest and Boosting algorithms perform much better than their weak learners.

Decision trees are the least accurate, performing with a 78.7% accuracy, due to their high variance/overfitting compared with the other classifier algorithms used to built the models.

Followed by **Gradient Boost**, which uses boosting method to combine individual decision trees, the accuracy is increased up to 80.3%.

Hyperparemetes are key parts of learning algorithms which effect the performance and accuracy of a model. Two critical hyperparameters for gradient boosting are the *learning rate* and *n_estimators*. Each added tree modifies the overall model, and by means of the learning rate parameter the magnitude of the modification is controlled. Models that learns slow tend to have a better performance, however, this has a computational cost on time and on the increment of trees needed in the model *(n_estimators)*. This also has a risk of overfitting if too many trees are used. In our model no increment in accuracy has been obtained by means of optimization when tuning the hyperparameters of this classifier.

The results for the three remaining classifiers are tied with an accuracy of 85.2%.

AdaBoost results show that it performs worse when irrelevant features are included in the model, improving its accuracy when only relevant important features are retained. This algorithm has only few hyperparameters that need to be tuned to improve model performance. AdaBoost is not optimized for speed, therefore being significantly slower than other classifiers.

Random Forest have some advantages over AdaBoost, since it is less affected by noise and it generalizes better reducing variance, and its speed is enhanced thanks to the parallel ensembling. Nevertheless, this algorithm requires more hyperparameter tuning because of a higher number of relevant parameters.

For the last classifier **XGBoost**, which is a relatively new algorithm introduced in 2016 that uses the concept of gradient tree boosting, its main advantage is the lightning speed[5] compared to the other algorithms, and its regularization parameter that successfully reduces variance. In addition to the regularization parameter, the algorithm also uses the *learning rate* and *subsamples* of features, thereby further improving its generalization ability. Nonetheless, compared with AdaBoost, Gradient Boost, and Random Forest, XGBoost is more difficult to understand and adjust. There are many hyperparameters that can be tuned to increase performance. This classifier is particularly interesting when both speed and high accuracy are of the essence. However, since model adjustment requires more time and user expertise to obtain meaningful results, more resources are needed to train the model. Therefore, the decision of when to use which tree-based algorithm depends on the dataset along with the resources available and the user expertise.

Decision Tree	Random Forest	AdaBoost	Gradient Boost	XGBoost
78.7%	85.2%	85.2%	80.3%	85.2%

Table 1: Comparison of results

The complete codes used for this report are available at GitHub repository.

[5]Since the dataset on which we implemented our models in a relatively small, differences in speed performance could not been appreciated between models.