1. Problem definition

The task can be formulated as the prediction of heart disease, based on some biological attributes. The dataset was collected from a clinic in Cleveland and contains 303 observations with 14 attributes:

- age : integer between 29 and 77 - continuous;
- sex : integer (1 for male, 0 for female) - discrete;
- chest pain : integer (1, 2, 3 or 4) - discrete;
- resting blood pressure : integer from 94 to 200 - continuous;
- cholesterol : integer between 126 and 564 - continuous;
- fasting blood sugar : integer (1 for true, 0 for false) - discrete;
- resting electrocardiographic results : integer (0, 1 or 2) - discrete;
- maximum heart rate achieved : integer between 71 to 202 - continuous;
- exercise induced angina : integer (1 for true, 0 for false) - discrete;
- ST depression induced by exercise relative to rest : double between 0 and 6.2 - continuous;
- the slope of the peak exercise ST segment : integer (0, 1 or 2) - discrete;
- number of major vessels colored by fluorosopy : integer (0, 1, 2 or 3) - discrete;
- thalassemia : integer (0, 1, 2 or 3) - discrete;
- diagnosis of heart disease : integer (1 for true, 0 for false) - discrete.

As the target value has only two values (1 for true - presence of heart disease, 0 for false - absence of heart disease), the task is a classification problem.

2. Random forest algorithm

In order to solve the problem, the model that was used is random forests. This model is an ensemble learning method that manages to manages to solve classification and regression problems. A random forest contains multiple decision trees that were built somehow randomly and that each cast a vote for the final result. When building the trees, at each node of a tree, the algorithm takes a random subset of variables and then applies a statistical test that will decide which variable to use for splitting the current node.

In random forests the j-th base learner is a tree denoted $h_j(X, \Theta_j)$, where Θ_j is a collection of random variables and the Θ_j's are independent for j=1,...J.

For a training data $\mathcal{D} = (x_1, y_1), ..., (x_N, y_N)$, where $x_i = (x_{i,1}, ...x_{i,p})^T$ the algorithm can be described in the following way:

For j=1 to J:

1. Draw a bootstrap sample $\mathcal{D}_j$ of size N from $\mathcal{D}$.
2. Grow a tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
    (a) Select m variables at random from the p variables.
    (b) Pick the best variable/split-point among m.
    (c) Split the node into two descendent nodes.

In order to predict the output for an input x the output prediction is the class that was obtained by the majority of the trees.

2.1 Implementation

```python
# Random Forest Algorithm
# @param train - array of floats - the train data
# @param test - array of floats - the test data
# @param max_depth - integer - the maximum depth of a tree
# @param min_size - integer - the minimum number of observations in a node
    for splitting
# @param sample_size - float between 0 and 1 - the dimension of the random
    subsample for building a tree
# @param n_trees - integer - the number of trees
# @param n_features - integer - the number of features that are selected
    randomly when building a tree
def random_forest(train, test, max_depth, min_size, sample_size, n_trees,
    n_features):
    trees = list()
    # define vector for variable importance
    variable_importance = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    # define variable for mean out of bag error
    oob_error_rate = 0
    for i in range(n_trees): # for each tree
        # select a sample of the dataset
        sample, oob_data = subsample(train, sample_size)
        # build the tree using the sample
        tree = decisiontree.build_tree(sample, max_depth, min_size,
            n_features, variable_importance)
        # update the out of bag error
        oob_error_rate += evaluation.oob_error_rate(tree, oob_data)
        # add tree to forest
        trees.append(tree)
    # after the forest has been created, make a prediction for each row in
        the test data
    predictions = [predict(trees, row) for row in test]
    # calculate variable importance
    for i in range(len(variable_importance)):
        variable_importance[i] /= n_trees
    # calculate mean out of bag error
    oob_error_rate /= n_trees
    return (predictions, variable_importance, oob_error_rate)
```

Listing 2.1: Random Forest Algorithm

```python
# Build a decision tree
# @param train_data - array of floats - the sample used to build the tree
# @param max_depth - integer - the maximum depth of a tree
# @param min_size - integer - the minimum number of observations in a node
    for splitting
# @param n_features - integer - the number of features that are selected
    randomly when building a tree
# @param variable_importance - array of floats - variable importance array
def build_tree(train_data, max_depth, min_size, n_features,
    variable_importance):
    # compute the current gini score
    class_values = list(set(row[-1] for row in train_data))
    current_gini = gini_index([train_data], class_values)
    # select feature for the root
    root = select_best_feature(train_data, n_features, current_gini,
        variable_importance)
    # split node and build tree recursively
    split_node(root, max_depth, min_size, n_features, 1, variable_importance)
    return root
```

Listing 2.2: Decision tree

```python
# Select the best split point for a dataset
# @param dataset - array of floats - the sample used to build the tree
# @param n_features - integer - the number of features that are selected
    randomly when building a tree
# @param current_gini - float - current gini score to be taken into account
    for variable importance
# @param variable_importance - array of floats - variable importance array
def select_best_feature(dataset, n_features, current_gini,
    variable_importance):
    class_values = list(set(row[-1] for row in dataset))
    feature_index, feature_split_value, feature_gini_score, feature_clusters
        = 999, 999, 999, None # best feature
    features = list()
    # pick n random features to test in order to select the one for the node
    while len(features) < n_features:
        index = randrange(len(dataset[0]) - 1)
        if index not in features:
            features.append(index)
    # find the (feature, feature_split_value) pair that result in the lowest
        gini index
    for index in features: # for each feature
        for row in dataset: # for each value of that feature
```

```python
            # compute the clusters that result from the split
            clusters = split_dataset_by_feature(index, row[index],dataset)
            # compute the gini index for the clusters
            gini = gini_index(clusters, class_values)
            variable_importance[index] += current_gini - gini
            # replace the selected feature if a better one was found
            if gini < feature_gini_score:
                feature_index, feature_split_value, feature_gini_score, feature_clusters = index, row[index], gini, clusters
    return {'index': feature_index, 'value': feature_split_value, 'gini': feature_gini_score, 'clusters': feature_clusters}
```

Listing 2.3: Feature selection for node

```python
# Recursive function that splits the dataset while building the tree, until getting to terminal nodes
# @param node - object - the node to split
# @param max_depth - integer - the maximum depth of a tree
# @param min_size - integer - the minimum number of observations in a node for splitting
# @param n_features - integer - the number of features that are selected randomly when building a tree
# @param depth - integer - the current depth
# @param variable_importance - array of floats - variable importance array
def split_node(node, max_depth, min_size, n_features, depth, variable_importance):
    left, right = node['clusters']
    del (node['clusters'])
    # check if any of the clusters is empty
    if not left or not right:
        node['left'] = node['right'] = create_terminal_node(left + right)
        return
    # check if the max_depth of the tree has been reached
    if depth >= max_depth:
        node['left'], node['right'] = create_terminal_node(left), create_terminal_node(right)
        return
    # check if the min_size of a node has been reached (for left child)
    if len(left) <= min_size:
        node['left'] = create_terminal_node(left)
    else:  # compute the next split
        node['left'] = select_best_feature(left, n_features, node['gini'], variable_importance)
        split_node(node['left'], max_depth, min_size, n_features, depth + 1, variable_importance)
    # check if the min_size of a node has been reached (for right child)
    if len(right) <= min_size:
        node['right'] = create_terminal_node(right)
    else: # compute the next split
        node['right'] = select_best_feature(right, n_features, node['gini'],
            variable_importance)
        split_node(node['right'], max_depth, min_size, n_features, depth + 1,
             variable_importance)
```

Listing 2.4: Node splitting

2.2 Parameter tuning

The main parameters of the random forest algorithm are:

- m - the number of randomly selected predictor variables chosen for each node
- J - the number of trees that build the random forest
- n_{min} the minimum node size

The parametrization was realized as recommended by Breiman, for m $[\sqrt{p}]$ (where p is the total number of features) and for n_{min} 1. Various values were experimented for J, the number of trees.

2.3 Evaluation

In order to evaluate the algorithm, a k-fold cross validation was performed, k=10. The implementation is presented in Listing 2.5 and the results in the next section.

```
# Evaluate an algorithm using k-fold cross validation
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    mean_err = 0
    mean_acc = 0
    mean_prec = 0
    mean_sens = 0
    mean_spec = 0
    mean_f1 = 0
    mean_oob_err = 0
    mean_variable_importance = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted, variable_importance_fold, oob_error = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        cm = confusion_matrix(actual, predicted)
        mean_err += error_rate(cm)
        mean_acc += accuracy(cm)
        mean_prec += precision(cm)
        mean_sens += sensitivity(cm)
        mean_spec += specificity(cm)
        mean_f1 += f1_score(cm)
        mean_oob_err += oob_error
        for i in range(len(mean_variable_importance)):
            mean_variable_importance[i] += variable_importance_fold[i]
    for i in range(len(mean_variable_importance)):
        mean_variable_importance[i] /= n_folds
    return {
        'error_rate': mean_err / float(len(folds)),
        'oob_error_rate': mean_oob_err / float(len(folds)),
        'accuracy': mean_acc / float(len(folds)),
        'precision': mean_prec / float(len(folds)),
        'sensitivity': mean_sens / float(len(folds)),
        'specificity': mean_spec / float(len(folds)),
        'f1_score': mean_f1 / float(len(folds)),
        'variable_importance': mean_variable_importance
    }
```

Listing 2.5: k-fold cross validation

3. Results

To compare the results obtained by the implemented algorithm, the dataset was also entered in a machine learning tool with a random forest implementation. The performance metrics, confusion matrices and variable importance arrays can be observed in the tables below.

No. of trees	Max depth	Error rate	Accuracy	Precision	Sensitivity	Specificity	F1 score
20	5	14	86	67.73	100	79.37	80.43
40	5	12.33	87.66	73.07	100	81.46	84.18
60	5	13	87	71.64	100	80.68	82.98
80	5	12.66	87.33	73.04	100	80.84	84.16
100	5	13	87	70.76	100	81.14	81.79
120	5	13.33	86.66	69.8	100	80.38	81.67
140	5	12.66	87.33	72.37	100	81.15	83.13
160	5	11.33	88.66	76.98	100	81.98	86.52
180	5	12	88	74.79	100	81.86	84.99
200	5	11.66	88.33	73.75	100	83.04	83.78
20	6	9	91	83.36	100	83.28	90.73
40	6	8.33	91.66	85.08	100	84.05	91.71
60	6	6	94	88.89	100	88.66	93.84
80	6	8	92	85.5	100	84.35	91.89
100	6	6.33	93.66	88.74	100	87.86	93.82
120	6	7	93	86.66	100	86.56	92.66
140	6	6.66	93.33	88.6	100	87.05	93.64
160	6	7	93	87.2	100	86.81	92.98
180	6	7	93	87.23	100	86.88	92.98
200	6	7	93	87.24	100	85.93	93.03
20	7	14.33	85.66	67.79	100	78.9	80.52
40	7	11.33	88.66	75.94	100	83.05	85.61
60	7	12.66	87.33	72.24	100	80.82	83.6
80	7	11.33	88.66	75.67	100	82.72	85.63
100	7	12.33	87.66	72.89	100	81.4	83.92
120	7	12	88	73.16	100	81.99	83.81
140	7	10.33	89.66	77.77	100	83.64	87.29
160	7	12.33	87.66	72.93	100	82.03	83.63
180	7	11	89	76.46	100	82.97	86.27
200	7	12	88	72.64	100	82.31	83.5

Table 3.1: Results obtained running the algorithm on the dataset with different values for the number of trees and for the maximum depth of a tree; The best results were obtained for 60 trees and maximum depth of 6

Criterion	Value
Accuracy	81.4%
Error rate	18.6%
Precision	88.4%
Recall	76.7%
F measure	81.6%
Sensitivity	76.7%
Specificity	87.5%

Table 3.2: The best results obtained by the random forest tool on the same dataset; The values were obtained with 140 trees and maximum depth of 7

	actual positive	actual negative
predicted positive	12 (true positive)	4 (false positive)
predicted negative	0 (false negative)	14 (true negative)

Table 3.3: Confusion matrix obtained by running the algorithm with 60 trees and maximum depth of 6

	actual positive	actual negative
predicted positive	34 (true positive)	11 (false positive)
predicted negative	5 (false negative)	36 (true negative)

Table 3.4: Confusion matrix obtained using a tool of random forest on the same dataset

Variable	Variable importance score
age	3.63
sex	4.06
chest pain	4.21
resting blood pressure	2.65
cholesterol	2.41
fasting blood sugar	0.49
resting electrocardiographic results	1.9
maximum heart rate achieved	5.62
exercise induced angina	3.08
ST depression induced by exercise relative to rest	4.67
the slope of the peak exercise ST segment	4.34
number of major vessels colored by fluorosopy	3.56
thalassemia	6.19

Table 3.5: Variable importance score computed with gini index; Most important variables are: thalassemia, maximum heart rate achieved and ST depression induced by exercise relative to rest.

Variable	Weight
age	0.038
sex	0.037
chest pain	0.057
resting blood pressure	0.072
cholesterol	0.065
fasting blood sugar	0.038
resting electrocardiographic results	0.02
maximum heart rate achieved	0.106
exercise induced angina	0.038
ST depression induced by exercise relative to rest	0.08
the slope of the peak exercise ST segment	0.036
number of major vessels colored by fluorosopy	0.072
thalassemia	0.136

Table 3.6: Weights of the variables computed by the random forest tool; The variables that weight the most are: thalassemia, maximum heart rate achieved and ST depression induced by exercise relative to rest.