TDT4173: Machine Learning and Case-Based Reasoning
Assignment 4
Stian Ismar

1 Theory:

1.1 Deep learning vs. shallow learning

Deep learning is characterized as representational learning. This means that the data samples supported to the learning system is an a raw format. This could be a matrix of image pixels.

These inputs are then passed through a multi-layered network which can pick up on many complex features. An example would be a convolutional network made for classifying animals.

Shallow learning, on the other side, is based on the user having some prior knowledge of the specific features on the input data. Feature selection and optimization is often emphasized when it comes to shallow learning.

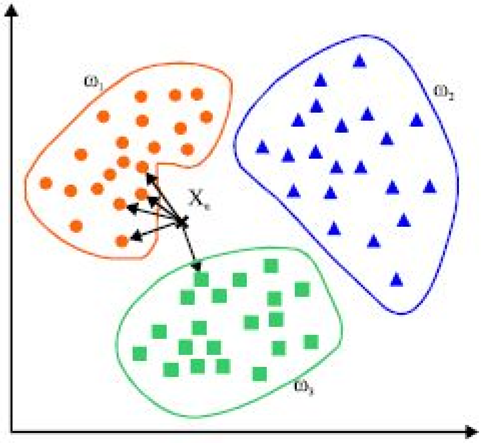
An example of shallow learning is a neural network which solves the XOR problem. Here, we only need one layer, and the input is not exactly in a "raw" format.

1.2 Machine learning techniques¶

In this section, I will look at some of the advantages and disadvantages of a selection of machine learning methods. The "No Free Lunch" theorem states that there is not one model that works best for every problem, as we will see.

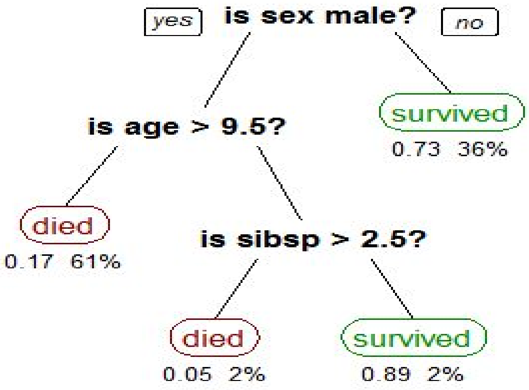
k-NN

k-nearest neighbours is a easy and simple machine learning model. It works by finding the k-closest values to the one your are trying to classify. The method can be used for classification or regression. By using the mean value of the k-nearest values, one can use it as a regression model. By using "votes" one can determine as to which class the unknown data sample belongs to This model works only if the data is graphable.



k-NN is computationally costly, meaning it is not suitable if the data set is large.

Decision tree



Decision trees support non-linearity, and is also used to solve classification or regression problems. The algorithm uses the best attribute of the dataset at the root of the tree. However, the technique is prone to outliers- There is a high risk of overfitting the tree.

One positive side of Decision trees is that it is a "white box" method. Compared to Deep Neural Networks, where the hidden layers are difficult to reason with, it is easier to determine why a data point is given a certain classification. The reasoning is visible to the domain expert.

Decision trees do not work that well on small data sets with many features. On the other side, its explainable nature makes it easy to explain the decisions made by the algorithm.

SVM

Support vector machines are algorithms that can be used for both classification and regression, but are mostly used in classification problems. SVMs are used to find a hyperplane that best divides a dataset, much like linear regression.

SVM supports both linear and non-linear data. If the data is not linearly separable, one can use kerneling to introduce another dimension.

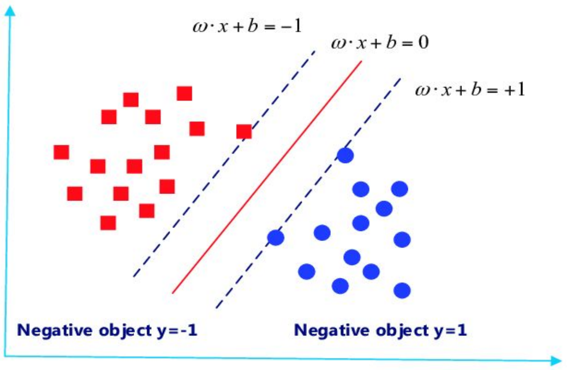


Figure 10. Support vector machine (SVM) classifier.

Pros of SVMs are that they are accurate and work well on smaller, cleaner datasets. Like k-NN, the method is not as good for training on larger dataset, since it is computationally expensive.

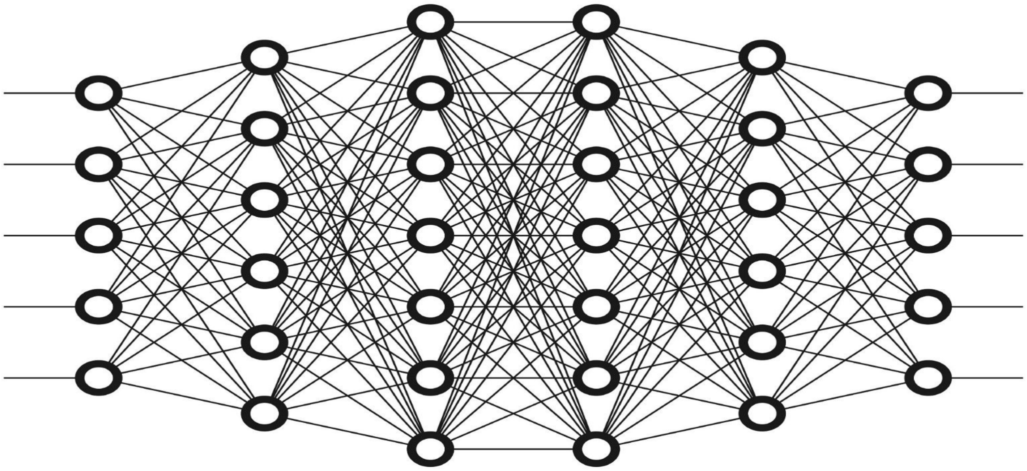
Deep learning

Amongst all the techniques looked at, deep learning is the best method for large data sets. More data means better performance.

For the other techniques, k-NN, SVM and DT, the feature extraction is done prior to running the algorithm. One or more dimensions are "removed" by not considering it when doing a classification for instance. With DNN however, the feature extraction process is done within the algorithm.

Deep neural networks use lower-level features to be able to for example classify images. This could be images of cats and dogs, where multiple features would be evaluated to get a good classification accuracy.

A commonly used deep neural network is called a convolutional neural network (CNN).



1.3 Ensemble methods

Ensemble methods are good to use when the training data is small. Many hypothesis will satisfy it, and ensambles reduces the risk of picking a bad classifier.

Ensemble methods is a machine learning method that combines several models. An unknown class label is predicted by aggregating predictions made by multiple classifiers. This means that random errors will be cancelled out, so that the correct decisions are reinforced. This way, ensemble classifiers are often more accurate than any of its individual classifiers.

There are three reasons as to why ensemble methods might do better. The first one is that they reduce the chance of picking a bad classifier. Secondly, ensembles reduce the chance of getting stuck in a bad minia. Thirdly, an ensemble method may result in a classifier that was not possible in the original set of hypothesis.

3 types of ensemble machine learning methods:

1. Boosting:

Involves using weak classifiers that are trained in sequence. A weak learner (classifier, predictor etc.) is a learner that performs just better than random. For each iteration in the boosting algorithm, the weak classifier focuses on the data points that were wrongly classified. In the end, the ensemble is strong. AdaBoost is a common implementation of a boosting algorithm.

2. Bootstrap aggregating (Bagging)

Bagging uses the general bootstrapping algorithm on a set of training data L. The general bootstrapping algorithm draws samples (with replacements) from the original set L and calculates the statistic of theta for each of the drawn samples. Then, the theatas for all samples are used to make further statistical inference such as estimating the confidence interval for theta.

The bagging algorithm also draws samples from the original data set. For each drawn subset, a predictor is trained. The trained predictors are then combined either by voting (classification) or averaging (estimation/regression)

2. Random forests

The third ensemble learning method I will be presenting is random forests, or random decision forests. Random forests are built of of many decision trees.

The different decision trees in the forest use a random subset of features from the data set. For each of the decision trees to be weak learners, we need to limit the depth of each tree to 1. Each of the trees can also access a random set of training data.

Each of the n trees in the forest make a prediction based on the feature and training data. In the end, voting is done from all the predictions of the trees in the forest. This leaves us with a final prediction of the unknown sample.

2 Programming:

2.1: k-NN implementation from scratch

General implementation of k-NN:

```
# kNN implementation,
# Xin is the training data,
# Yin is the class labels (supervised)
# x is the unknown sample
def find_K_neighbours(Xin,Yin,x):
    k = 10 # from the assignment
    # The closes neighbours to the left in asc_neig:
    asc_neig = []
    for rownum, train_point in Xin.iterrows():
        # Calculate the distance to the sample x and insert it into a sorted list:
        bisect.insort(asc_neig, (findDistance(train_point, x),Yin[rownum],rownum))
    # Slicing the list from element 0 to element k:

    # Removing the closest one, which is the actual point from the training data:
    asc_neig = asc_neig[1:]

    return asc_neig[0:(k)]
```

Helper function for euclidean distance:

```
# Returns the euclidean distance sum in n-dimensional space:
# p2 is the test point!
def findDistance(p1,p2):
    euclid_dist_sum = 0
    for i in range(len(p1)):
        euclid_dist_sum += (p1[i] - p2[i])**2
    return (euclid_dist_sum)
```

In this particular exercise the “unknown” sample x was in the training data. I fixed this issue by removing the closest value from the list of closest values. See code above.

Classification problem, solved with voting:

```
df_class = load_csv("./dataset(3)/knn_classification.csv")
Y_class = list(df_class.y)
X_class = df_class.loc[:,'x1':'x4']

knn_classification_problem = find_K_neighbours(X_class,Y_class,[6.3, 2.7, 4.91, 1.8])
# Voting:
# Making a list with the different target values we have seen in the training data:
distinct_vals_Y = []
for y in Y_class:
    if y not in distinct_vals_Y:
        distinct_vals_Y.append(y)

# Count the values

def count_votes():
    votes = [0 for x in range(len(distinct_vals_Y))]
    for i in range(len(votes)):
        for n in knn_classification_problem:
            if (n[1] == i):
                votes[i]+=1
    return votes.index(max(votes))


classifier_result = count_votes()

def printClosestAsc(nearest_neighbours,input_x):
    i = 1
    for elem in nearest_neighbours:
```

```
    print("Number ",i, "is:")
    row_index = (elem[2])
    p = input_x.iloc[row_index,:]
    print(row_index)
    print(p)
    print("Distance to target:", round(elem[0],4))
    i+=1
    print(" ")
```

The results were:

```
The datapoint is in the class:  2
```

The ten closest neighbours for the classification problem were:

```
Number  1 is:
126
x1    6.2
x2    2.8
x3    4.8
x4    1.8
Name: 126, dtype: float64
Distance to target: 0.0321

Number  2 is:
146
x1    6.3
x2    2.5
x3    5.0
x4    1.9
Name: 146, dtype: float64
Distance to target: 0.0581

Number  3 is:
127
x1    6.1
x2    3.0
x3    4.9
x4    1.8
Name: 127, dtype: float64
Distance to target: 0.1301

Number  4 is:
72
x1    6.3
```

```
x2    2.5
x3    4.9
x4    1.5
Name: 72, dtype: float64
Distance to target: 0.1301

Number  5 is:
133
x1    6.3
x2    2.8
x3    5.1
x4    1.5
Name: 133, dtype: float64
Distance to target: 0.1361

Number  6 is:
83
x1    6.0
x2    2.7
x3    5.1
x4    1.6
Name: 83, dtype: float64
Distance to target: 0.1661

Number  7 is:
111
x1    6.4
x2    2.7
x3    5.3
x4    1.9
Name: 111, dtype: float64
Distance to target: 0.1721

Number  8 is:
138
x1    6.0
x2    3.0
x3    4.8
x4    1.8
Name: 138, dtype: float64
Distance to target: 0.1921

Number  9 is:
54
```

```
x1    6.5
x2    2.8
x3    4.6
x4    1.5
Name: 54, dtype: float64
Distance to target: 0.2361

Number  10 is:
147
x1    6.5
x2    3.0
x3    5.2
x4    2.0
Name: 147, dtype: float64
Distance to target: 0.2541
```

Regression problem, solved with mean:

```
# Loading the data for the regression problem:
df_reg = load_csv("./dataset(3)/knn_regression.csv")
Y_reg = list(df_reg.y)
X_reg = df_reg.loc[:,'x1':'x3']

knn_regression_problem = find_K_neighbours(X_reg,Y_reg,[6.3, 2.7, 4.91])

def sample_mean_value(ten_closest_neighbours):
    sum = 0
    for elem in ten_closest_neighbours:
        sum += elem[1]
    return sum/len(ten_closest_neighbours)

res_reg = sample_mean_value(knn_regression_problem)
```

The result for the regression problem was the following print:

```
The mean value for the regression problem is the following: 1.6099999999999999
The ten closest neighbours for the regression problem are:
Number  1 is:
126
x1    6.2
x2    2.8
x3    4.8
```

```
Name: 126, dtype: float64
Distance to target: 0.0321

Number  2 is:
72
x1    6.3
x2    2.5
x3    4.9
Name: 72, dtype: float64
Distance to target: 0.0401

Number  3 is:
133
x1    6.3
x2    2.8
x3    5.1
Name: 133, dtype: float64
Distance to target: 0.0461

Number  4 is:
146
x1    6.3
x2    2.5
x3    5.0
Name: 146, dtype: float64
Distance to target: 0.0481

Number  5 is:
73
x1    6.1
x2    2.8
x3    4.7
Name: 73, dtype: float64
Distance to target: 0.0941

Number  6 is:
63
x1    6.1
x2    2.9
x3    4.7
Name: 63, dtype: float64
Distance to target: 0.1241

Number  7 is:
83
x1    6.0
x2    2.7
```

```
x3    5.1
Name: 83, dtype: float64
Distance to target: 0.1261

Number  8 is:
127
x1    6.1
x2    3.0
x3    4.9
Name: 127, dtype: float64
Distance to target: 0.1301

Number  9 is:
54
x1    6.5
x2    2.8
x3    4.6
Name: 54, dtype: float64
Distance to target: 0.1461

Number  10 is:
111
x1    6.4
x2    2.7
x3    5.3
Name: 111, dtype: float64
Distance to target: 0.1621
```

2.2 AdaBoost implementation from scratch

This AdaBoost implementation for binary classification was done using sklearn decision trees with depth = 1 as the weak classifiers.

The error results plot for the training and test data show that the test data was classified a bit worse than the training data. This is expected. This data set had 10 features, and I experienced that I had to run the algorithm at least 100 times to get good results, and still run at a reasonable time.

The final error for the test data is within limits. The error was calculated using this function:

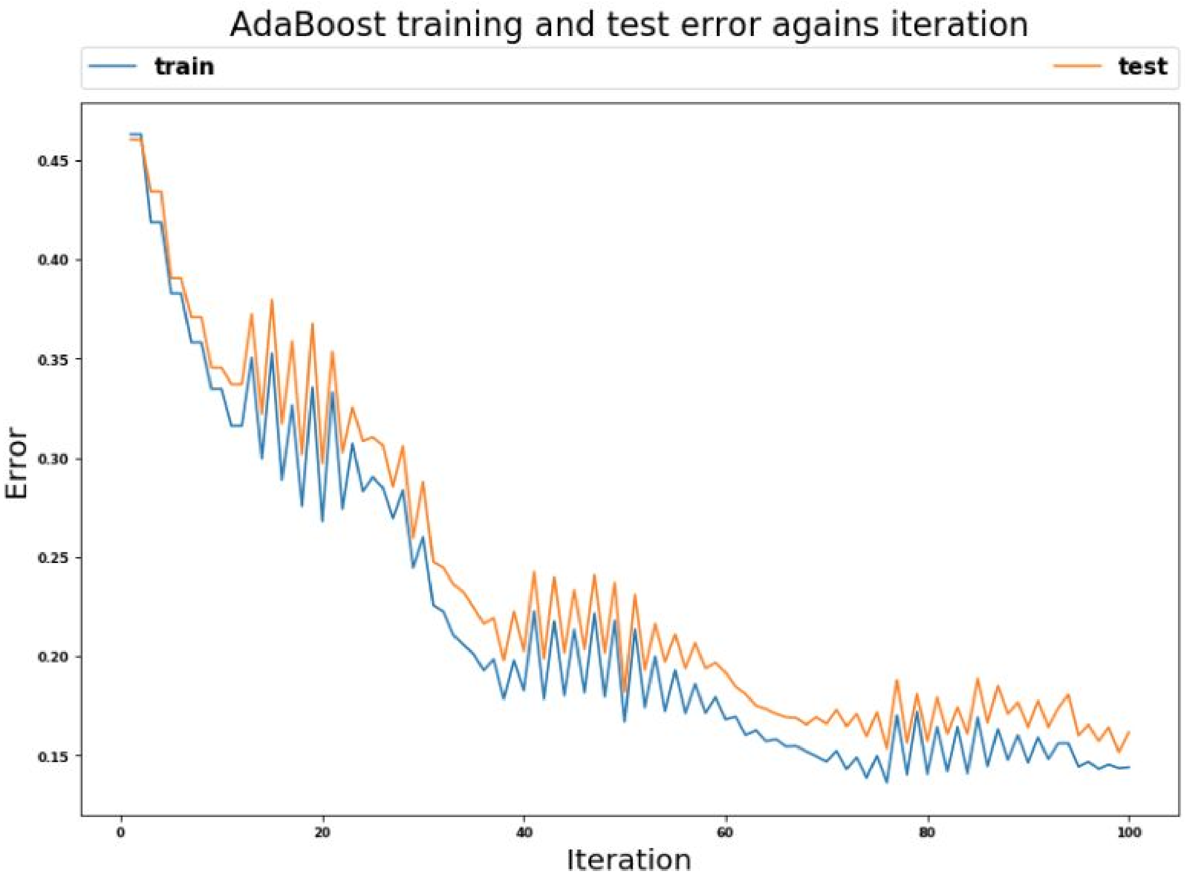
```python
def calculate_error_clf(y_pred, y):
    sum_error = 0
    y_pred = np.array(y_pred)
    y= (np.array(y['y']))
```

```
sum_error = len(y) - np.count_nonzero(y_pred==y)
sum_error = sum_error/y.size

return sum_error
```

The whole implementation is in the appendix. I made sure the implementation was general enough so that it took in an arbitrary weak classifier (and was not restricted to using the Decision Tree from sklearn).

The error results for the test and training data are in the following plot. I could see that the error for the training set was a little lower than for the test set. This is expected, since the weight in the model are trained in the training data.



2.3 k-NN vs. SVM vs. Random Forest on sklearn digit data set

For this task we were asked to use library implementations of k-NN, SVM and RandomForest. I chose the implementations from sklearn, and trained them on their “digits” dataset.

The three different classifiers were loaded:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm
from sklearn.metrics import accuracy_score
```

The digit data set was loaded like so:

```
from sklearn.datasets import load_digits
```

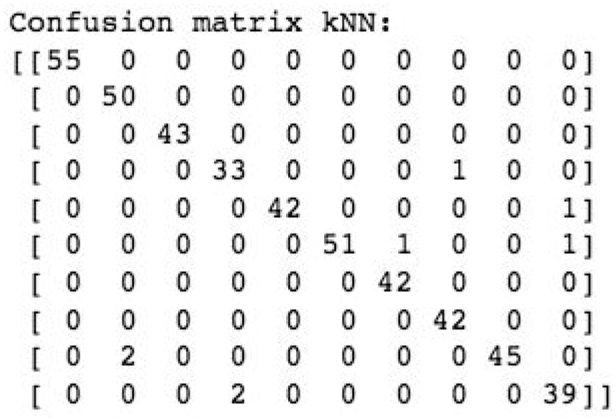
The data was split using:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

The confusion matrices were plotted with the predictions along the x-axis and the actual targets along the y-axis. The accuracy was calculated using sklearns function: “accuracy_score(y,ypred)”. This is done by *accuracy = number of correct predictions/number of samples.*

k-NN

```
neigh = KNeighborsClassifier(n_neighbors=3)
clf_knn = neigh.fit(X_train, y_train)
y_pred_knn = []
for sample in X_test:
   y_pred_knn.append(clf_knn.predict([sample])[0])
```



```
Confusion matrix kNN:
[[55  0  0  0  0  0  0  0  0  0]
 [ 0 50  0  0  0  0  0  0  0  0]
 [ 0  0 43  0  0  0  0  0  0  0]
 [ 0  0  0 33  0  0  0  1  0  0]
 [ 0  0  0  0 42  0  0  0  0  1]
 [ 0  0  0  0  0 51  1  0  0  1]
 [ 0  0  0  0  0  0 42  0  0  0]
 [ 0  0  0  0  0  0  0 42  0  0]
 [ 0  2  0  0  0  0  0  0 45  0]
 [ 0  0  0  2  0  0  0  0  0 39]]
```

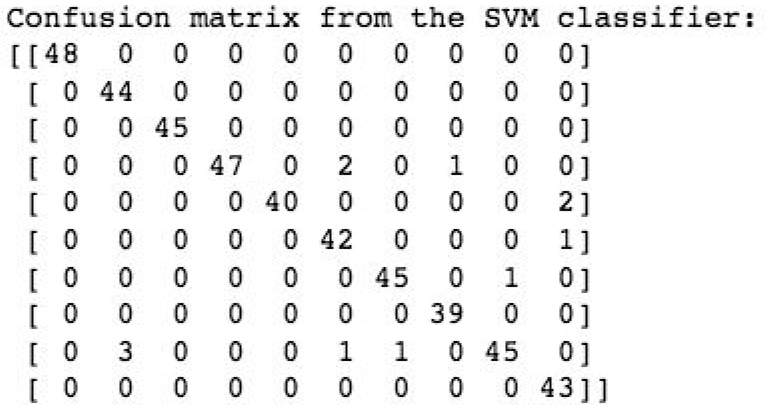
```
accuracy_score_knn: 0.9822222222222222
```

From the confusion matrix one can for example see that the k-NN algorithm did well. Only a few handwritten digits were misplaced. One example was that a “four” was mistakenly predicted to be a “nine”. In total, 4 digits were wrongly classified. k-NN had the best results of SVM and random forest!

SVM

```
clf_svm = svm.SVC(gamma='scale')
clf_svm = clf_svm.fit(X_train, y_train)
```

```
y_pred_svm = []
for sample in X_test:
    y_pred_svm.append(clf_svm.predict([sample])[0])
```



```
Confusion matrix from the SVM classifier:
[[48  0  0  0  0  0  0  0  0  0]
 [ 0 44  0  0  0  0  0  0  0  0]
 [ 0  0 45  0  0  0  0  0  0  0]
 [ 0  0  0 47  0  2  0  1  0  0]
 [ 0  0  0  0 40  0  0  0  0  2]
 [ 0  0  0  0  0 42  0  0  0  1]
 [ 0  0  0  0  0  0 45  0  1  0]
 [ 0  0  0  0  0  0  0 39  0  0]
 [ 0  3  0  0  0  1  1  0 45  0]
 [ 0  0  0  0  0  0  0  0  0 43]]
```

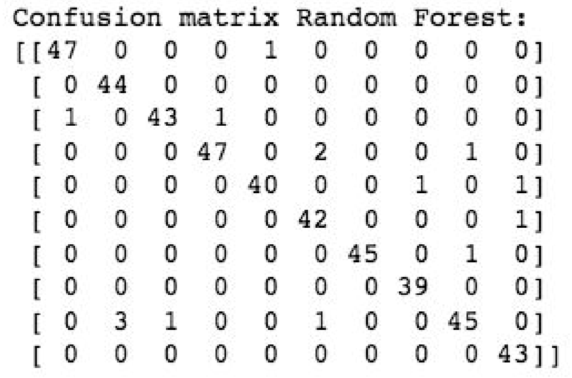
```
Accuracy score: 0.9733333333333334
```

The confusion matrix shows that 12 digits were misclassified. For instance, 3 “eight” digits were classified as 1’s. However, the accuracy score was reasonably high.

Random forest

```
clf_rf = RandomForestClassifier(n_estimators=100,
                                random_state=0)
clf_rf.fit(X_train, y_train)
```

```
y_pred_rf = []
for sample in X_test:
    y_pred_rf.append(clf_rf.predict([sample])[0])
```



```
Confusion matrix Random Forest:
[[47  0  0  0  1  0  0  0  0  0]
 [ 0 44  0  0  0  0  0  0  0  0]
 [ 1  0 43  1  0  0  0  0  0  0]
 [ 0  0  0 47  0  2  0  0  1  0]
 [ 0  0  0  0 40  0  0  1  0  1]
 [ 0  0  0  0  0 42  0  0  0  1]
 [ 0  0  0  0  0  0 45  0  1  0]
 [ 0  0  0  0  0  0  0 39  0  0]
 [ 0  3  1  0  0  1  0  0 45  0]
 [ 0  0  0  0  0  0  0  0  0 43]]
```

```
Accuracy score random forest: 0.9666666666666667
```

The results shows that this classifier had 14 misclassifications, which was the poorest classification of the three methods in this task. This might have been because the function signature was not optimal. I tried increasing the amount of trees in the forest, but I only got a lower accuracy.

Appendix:

The AdaBoost implementation:

```
def adaBoost(X_train,Y_train, clf,iterations,x_test):
    classifiers = []
    classifiers_test = []

    # initializing the weights:
    N = len(Y_train)
    w_i = [1 / N] * N
    T = iterations
    clf_errors = []

    for t in range(T):
        clf.fit(X_train, Y_train, sample_weight = w_i)

        print(math.floor((t/T)*100),"%"," ", end='')

        #Predict all the values:
        y_pred = clf.predict(X_train)
        y_pred_test = clf.predict(x_test)
        error_internal = calc_error(w_i,Y_train,y_pred)
        alpha = np.log((1-error_internal)/ error_internal)

        # Add the predictions and alpha for later use for every iteration
        classifiers.append((y_pred, alpha))
        classifiers_test.append((y_pred_test,alpha))
        w_i = update_weights(w_i,y_pred,Y_train,alpha,clf)

    # Output the final prediction:
    G, prev_preds = output(classifiers, X_train)
    G_test, prev_preds_test = output(classifiers_test, x_test)
    return G, prev_preds, G_test, prev_preds_test

def output(clfs, X_train):
    res_pred_it = []
    s = np.zeros(len(X_train))
    i = 0
    for (y_pred, alpha) in clfs:
        s += alpha*y_pred
        res_pred_it.append(np.sign(s))
        i+=1
    return np.sign(s), res_pred_it

def calc_error(weights,Y_train,y_pred):
    err = 0
    for i in range(len(weights)):
        if y_pred[i] != Y_train['y'].iloc[i]:
            err= err + weights[i]

    # Normalizing the error:
    err = err/np.sum(weights)
    return err

# If the prediction is true, return 0. If it is not true, return 1.
def check_pred(y_p, y_t):
    if y_p == y_t:
        return 0
    else:
        return 1

def update_weights(w,y_pred,Y_train,alpha,clf):
    for j in range(len(w)):
        if y_pred[j] != Y_train['y'].iloc[j]:
            w[j] = w[j]* (np.exp( alpha * 1))
    return w

def calculate_error_clf(y_pred, y):
    sum_error = 0
    y_pred = np.array(y_pred)
    y= (np.array(y['y']))
    sum_error = len(y) - np.count_nonzero(y_pred==y)
    sum_error = sum_error/y.size

    return sum_error

def find_iteration_errors(it, X,Y_train,y_test,x_test):
    weak_clf = tree.DecisionTreeClassifier(max_depth = 1)
    errors = np.zeros(it)
    errors_test = np.zeros(it)
    final_pred, prev_preds, final_pred_test, prev_preds_test = adaBoost(X_train, Y_train,weak_clf,it,x_test)
    i=0

    # Get the x and y values for plotting the error for the training:
    for j in range(len(prev_preds)):
    # for pred in prev_preds:
        errors_test[i] = calculate_error_clf(prev_preds_test[j],y_test)
        errors[i] = calculate_error_clf(prev_preds[j], Y_train)
        i+=1
    print("Completed", end='')
    x_vals = list(range(1,it+1))

    # plt.show()
    return x_vals,errors,errors_test
```

```
x_vals, error_res, errors_test = find_iteration_errors(100,X_train,Y_train,y_test, x_test)
```