Mustererkennung/Machine Learning - Assignment 6

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
```

```
data = np.array(pd.read_csv('./spambase/spambase.data', header=None))

X = data[:,:-1] # features
y = data[:,-1] # Last column is label

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0,
shuffle=True, stratify=y)
```

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
print(X_train[1])
print(y_train[1])
```

```
(3450, 57)
(1151, 57)
(3450,)
(1151,)
[ 0.     0.     0.     0.     0.     0.     0.     0.     0.     0.
  0.     1.14   0.     0.     0.     0.     0.     0.     2.29   0.
  0.     0.     0.     0.     1.14   1.14   0.     0.     0.     0.
  1.14   0.     0.     0.     0.     0.     0.     0.     0.     0.
  0.     0.     0.     2.29   0.     0.     0.     0.     0.     0.
  0.     0.596  0.     0.198  2.133 14.    64.   ]
0.0
```

Task 1

```
def majority(y):
    counts = np.bincount(y)
    return np.argmax(counts)

def square(num):
    return pow(num, 2)

# reorder x and y according to j-dimension
def reorder_data(x_data, y_data, j):
```

```
    index = x_data[:,j-1].argsort()

    temp_x = x_data[index]
    temp_y = y_data[index]

    return temp_x, temp_y

def calculate_entropy(feature, y_data):
    right = (feature == True).sum() / feature.size
    left = 1 - right

    left_child = np.sum(y_data[feature]) / y_data[feature].size
    right_child = np.sum(y_data[np.invert(feature)]) /
y_data[np.invert(feature)].size

    Q_1 = right * loss_function(left_child)
    Q_2 = left * loss_function(right_child)
    Q_tot = Q_1 + Q_2

    return Q_tot, right_child, left_child

def loss_function(p):
        if p == 1 or p == 0:
            return 0

        return - (p * np.log(p) + (1-p) * np.log((1-p)))
```

```
class DecisionTree():

    def __init__(self, height):
        self.min_size = 3
        self.height = height

    def set_min_size(self, min_size):
        self.min_size = min_size

    def fit(self, X_data, y_data, sample_weight=None):
        self.tree_size = pow(2, self.height) - 1
        self.tmp_size = pow(2, self.height + 1) - 1
        self.features = X_data.shape[1]
        self.tree = np.full(self.tmp_size, -1)
        self.tree_tmp = np.full(self.tmp_size + 1, -1)

        self.split_tree(X_data, y_data, 0)

    # go through the decision tree
    def predict(self, X_data):
        predictions = []
        for x in X_data:
            i = 0
            leaf = self.tree[i]

            while self.tree[self.left_node(i)] != -1 or
self.tree[self.right_node(i)] != -1:

                if leaf >= self.tree_size:
                    return
```

```
                if x[leaf]:
                    i = self.right_node(i)
                else:
                    i = self.left_node(i)
                prediction = self.tree_tmp[i]
                leaf = self.tree[i]
            predictions.append(prediction)
        return predictions

    def split_data(self, index, value, X_data):
        left, right = [], []
        for x in X_data:
            if x[index] < value:
                left.append(x)
            else:
                right.append(x)
        return left, right

    def split_tree(self, X_data, y_data, leaf):
        if leaf >= self.tree_size:
            return

        entropies = np.full(self.features, np.inf)
        left = np.empty(self.features)
        right = np.empty(self.features)

        for i, feature in enumerate(X_data.T):
            temp = feature.astype(int)
            if np.sum(feature) == 0 or np.sum(np.invert(temp)) == 0:
                continue
            entropies[i], left[i], right[i] = calculate_entropy(feature, y_data)

        index = np.argmin(entropies)

        right = X_data[:,index]
        left = np.invert(right)

        self.tree[leaf] = index
        if index < len(self.tree_tmp):
            if (index < len(left)) and (index < len(right)):
                self.tree_tmp[self.left_node(leaf)] = left[index]
                self.tree_tmp[self.right_node(leaf)] = right[index]

        if len(y_data[right]) == 0 or len(y_data[left]) == 0:
            return

        if leaf >= self.min_size:
            return

        self.split_tree(X_data[left], y_data[left], self.left_node(leaf))
        self.split_tree(X_data[right], y_data[right], self.right_node(leaf))

    def left_node(self, node):
        return 2 * node + 1

    def right_node(self, node):
        return 2 * node + 2
```

```
means = (np.mean(X_train[y_train==1], axis=0) + np.mean(X_train[y_train==0])) /
2

X_train_means = X_train > means
X_test_means = X_test > means

tree = DecisionTree(20)
tree.fit(X_train_means, y_train)
predictions = tree.predict(X_test_means)
```

```
<ipython-input-35-91ecc1c01dec>:33: RuntimeWarning: invalid value encountered in
double_scalars
  right_child = np.sum(y_data[np.invert(feature)]) /
y_data[np.invert(feature)].size
```

Task 1 a)

If classifying a genuine E-Mail as spam is ten times worse than classifying spam as genuine, we can just exchange the prediction value from 1 to 0 and from 0 to 1.

```
from sklearn.metrics import confusion_matrix
estimates = (np.array(predictions) > 0.5)
print(confusion_matrix(predictions, estimates))
```

```
[[365   0]
 [  0 786]]
```

Task 1 b)

Top 5 features:

1. address
2. free
3. money
4. direct
5. re

```
indices = [1, 15, 23, 39, 44]
```

Task 2

```
class RandomForest:

    def __init__(self, height=7, n_trees = 100):
        self.n_trees = n_trees
        self.height = height
        self.trees = [DecisionTree(height = height) for _ in range(n_trees)]

    def fit(self, X, y, n_samples = 500):
        for tree in self.trees:
            random_samples = np.random.randint(0, high=len(X), size=n_samples)
```

```
            X_train = X[random_samples]
            y_train = y[random_samples]

            random_features = np.random.randint(0, high=len(X.T),
size=self.height*2)
            X_train = X_train[:,random_features]

            means = (np.mean(X_train[y_train==1], axis=0) +
np.mean(X_train[y_train==0])) / 2
            X_train_means = (X_train > means)
            tree.fit(X_train_means, y_train)

    def predict(self,X):
        forest_predictions = np.array(self.trees[0].predict(X))
        forest_predictions = forest_predictions[:, np.newaxis]

        for i in range(1, self.n_trees):
            prediction = np.array(self.trees[i].predict(X))
            forest_predictions = np.append(forest_predictions, prediction[:,
np.newaxis], axis=1)

        avg = np.array(np.mean(forest_predictions, axis=0))
        return avg
```

```
def test(tree):
    random_forest = RandomForest(height=7, n_trees=tree)
    random_forest.fit(X, y, n_samples = 1000)
    predictions_rf = random_forest.predict(X_test_means)

    estimates_rf = (np.array(predictions_rf) > 0.5)

    print("trees: ", tree)
    print(confusion_matrix(predictions_rf.round(), estimates_rf.round()))
    print("-----")
```

Task 2 a)

```
test(10)
test(30)
test(100)

# for tree in range(5, 300, 25):
#     test(tree)
```

```
<ipython-input-35-91ecc1c01dec>:33: RuntimeWarning: invalid value encountered in
double_scalars
  right_child = np.sum(y_data[np.invert(feature)]) /
y_data[np.invert(feature)].size
```

```
trees:  10
[[10]]
-----
trees:  30
[[ 7  0]
 [ 0 23]]
-----
trees:  100
[[ 9  0]
 [ 0 91]]
-----
```

Task 2 b)

In general, the more trees you use the better get the results. However, the improvement decreases as the number of trees increases, i.e. at a certain point the benefit in prediction performance from learning more trees will be lower than the cost in computation time for learning these additional trees.

Typical values for the number of trees is 10, 30 or 100.

```

```