Mustererkennung/Machine Learning - Assignment 7

Load the spam dataset:

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import random
from tqdm import tqdm
```

In [2]:

```
data = np.array(pd.read_csv('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=T
```

The Decision Tree implementation, which will be used in AdaBoost

In [3]:

```
def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

def gini(y_true, c):
    """
    For simplicity reasons this assumes that there are only 2 classes
    """
    y_true = np.array(y_true)
    p_mk = np.mean(y_true == c)

    return 2 * p_mk * (1 - p_mk)

class LeafNode():
    def fit(self, c):
        self.c = c

    def predict(self, x):
        return self.c

class InternalNode():
    """
    Node of decision tree, which accepts tabular data
    """
    def fit(self, x, y, depth, max_depth):
        m, n = x.shape
        # columns are j, split_index, loss_total
        split_infos = []

        for j in range(n):
            # sort rows by feature j in ascending order
            sorted_indices = x[:,j].argsort()
            x, y = x[sorted_indices], y[sorted_indices]
            for split_index in self.get_unique_indices(x[:, j])[:-1]:
                y_top_split = y[:split_index + 1]
                y_bottom_split = y[split_index + 1:]

                if y_top_split.shape[0] == 0:
                    raise Exception("Error 1")

                if y_bottom_split.shape[0] == 0:
                    raise Exception("Error 2")

                c1 = self.find_c(y_top_split)
                c2 = self.find_c(y_bottom_split)

                loss_1 = gini(y_top_split, c1)
                loss_2 = gini(y_bottom_split, c2)

                # use weighted average which had better results
                loss_total = (y_top_split.shape[0] / m) * loss_1  + (y_bottom_split

                row = np.array([j, split_index, loss_total])
                split_infos.append(row)

        split_infos = np.array(split_infos)
        best_split_idx = np.argmin(split_infos[:,-1], axis=0)
        best_split = split_infos[best_split_idx]
        self.j = int(best_split[0])
        split_index = int(best_split[1])
```

```
        sorted_indices = x[:,j].argsort()
        x, y = x[sorted_indices], y[sorted_indices]

        x_top_split, y_top_split = x[:split_index + 1], y[:split_index + 1]
        x_bottom_split, y_bottom_split = x[split_index + 1:], y[split_index + 1:]

        self.z = (x_top_split[-1, self.j] + x_bottom_split[0, self.j]) / 2

        if self.is_pure(y_top_split) or x_top_split.shape[0] <= 2 or depth >= max_d
            self.left_child = LeafNode()
            c = self.find_c(y_top_split)
            self.left_child.fit(c)
        else:
            self.left_child = InternalNode()
            self.left_child.fit(x_top_split, y_top_split, depth + 1, max_depth)

        if self.is_pure(y_bottom_split) or x_bottom_split.shape[0] <= 2 or depth >=
            self.right_child = LeafNode()
            c = self.find_c(y_bottom_split)
            self.right_child.fit(c)
        else:
            self.right_child = InternalNode()
            self.right_child.fit(x_bottom_split, y_bottom_split, depth + 1, max_dep

    def predict(self, x):
        if x[self.j] <= self.z:
            return self.left_child.predict(x)
        return self.right_child.predict(x)

    def get_unique_indices(self, arr):
        idx = []
        arr_len = len(arr)
        for i in range(len(arr)):
            if i == arr_len - 1:
                idx.append(i)
            elif arr[i] != arr[i + 1]:
                idx.append(i)
        return idx

    def find_c(self, y):
        """
        For simplicity reasons this assumes that there are only 2 classes
        """
        y = np.array(y)
        zeros = np.sum(y == 0)
        ones = np.sum(y == 1)
        if ones > zeros:
            return 1
        return 0

    def is_pure(self, y):
        y = np.array(y)
        if np.sum(y == 0) == y.shape[0] or np.sum(y == 1) == y.shape[0]:
            return True
        return False

class DecisionTreeClassifier():
    """
    Basically just holds the root node of the tree which starts the recursion
    """
```

```
    def __init__(self, max_depth):
        self.max_depth = max_depth

    def fit(self, x, y, features):
        x = np.copy(x)
        y = np.copy(y)
        self.root = InternalNode()
        self.root.fit(x, y, 1, self.max_depth)

    def predict(self, x):
        y_preds = []
        for sample in x:
            y_pred = self.root.predict(sample)
            y_preds.append(y_pred)
        return np.array(y_preds)
```

Excercise 1. AdaBoost

Implement AdaBoost using Python (incl. Numpy etc.) and use it on the SPAM-Dataset. The weak classifiers should be decision stumps (i.e. decision trees with one node).

In [4]:

```
class AdaBoost():
    def __init__(self, num_trees, max_depth):
        self.num_trees = num_trees
        self.max_depth = max_depth

    def fit(self, x, y, features=None):
        x = np.copy(x)
        y = np.copy(y)
        self.trees = []
        self.says = []
        for i in tqdm(range(self.num_trees)):
            # sample weights will always add up to one
            sample_weights = np.ones((y.shape[0], )) / y.shape[0]
            tree = DecisionTreeClassifier(max_depth=self.max_depth)
            tree.fit(x, y, features)
            y_pred = tree.predict(x)
            error = self.calculate_error(y, y_pred, sample_weights)
            say = self.error_to_say(error)
            sample_weights = self.update_sample_weights(y, y_pred, say, sample_weig
            x, y = self.weighted_dataset(x, y, sample_weights)
            self.trees.append(tree)
            self.says.append(say)

        self.says = np.array(self.says)

    def weighted_dataset(self, x, y, sample_weights):
        x_new, y_new = [], []
        sample_weights_cum = np.cumsum(sample_weights)
        rand = np.random.uniform(low=0.0, high=1.0, size=(x.shape[0], ))
        for rand_el in rand:
            for i, cum_weight in enumerate(sample_weights_cum):
                if cum_weight >= rand_el:
                    x_new.append(x[i])
                    y_new.append(y[i])
                    break
        return np.array(x_new), np.array(y_new)



    def calculate_error(self, y_true, y_pred, sample_weights):
        """
        How much say a stump has is calculated by it's error, which is just the
        sum of the sample_weights for the missclassified samples.
        The error is always between 0 and 1 because the sample weights add up to on
        0 is the lowest possible error and 1 is the highest.
        """
        error_idx = y_true != y_pred
        return np.sum(sample_weights[error_idx])

    def error_to_say(self, error):
        """
        Transforms the error a stump has into it's say which will be used
        to weight the importance of one stumps prediction in the final prediction.
        The say is ~ between 3.5 and -3.5 which means a stumps prediction can actua
        be weighted negaively in the final prediction if it error is high.
        If error is 0 we will have division by 0, if error is 1, we will have log(0
        which is also not possible. So a small eps is added / subtracted from the
        error to keep calculations stable.
        """
```

```
        eps = 10 ** -10
        if error == 0:
            error = error + eps
        elif error == 1:
            error = error - eps
        return 0.5 * np.log((1 - error) / error)

    def update_sample_weights(self, y_true, y_pred, say, sample_weights):
        """
        Updates the sample weights by scaling them based on the amount of say the s
        has and wether it properly classified the sample.
        If say is high and the sample was missclassified, the sample weight will go
        If say is high and the sample was propely classified, the sample weight wil
        If say is low and the sample was missclassified, the sample weight will go
        If say is low and the sample was properly classified, the sample weight wil
        After updating the sample weights will still sum up to one.
        """
        sample_weights = np.where(y_true == y_pred,
                                  sample_weights * np.exp(-say),
                                  sample_weights * np.exp(say))
        # normalization so sample_weights add up to 1 again
        sample_weights = sample_weights / np.sum(sample_weights)
        return sample_weights

    def predict(self, x, use_cascade=False):
        y_preds = []
        if use_cascade:
            for sample in x:
                votes = np.array([])
                made_prediction = False
                for tree in self.trees:
                    # decision tree expects matrix as input
                    sample = sample.reshape((1, -1))
                    prediction = tree.predict(sample)
                    votes = np.concatenate((votes, prediction))
                    yes_say = np.sum(self.says[:len(votes)][votes == 1])
                    no_say = np.sum(self.says[:len(votes)][votes == 0])
                    if no_say >= yes_say:
                        y_preds.append(0)
                        made_prediction = True
                        break
                if not made_prediction:
                    y_preds.append(1)
        else:
            for sample in x:
                votes = np.array([])
                for tree in self.trees:
                    # decision tree expects matrix as input
                    sample = sample.reshape((1, -1))
                    prediction = tree.predict(sample)
                    votes = np.concatenate((votes, prediction))
                yes_say = np.sum(self.says[votes == 1])
                no_say = np.sum(self.says[votes == 0])
                if yes_say > no_say:
                    y_preds.append(1)
                else:
                    y_preds.append(0)

        return np.array(y_preds)
```

In [5]:

```
%%time
clf = AdaBoost(num_trees=20, max_depth=1)
clf.fit(X_train, y_train)
```

```
100%|██████████| 20/20 [00:24<00:00,  1.21s/it]

CPU times: user 24.1 s, sys: 8.4 ms, total: 24.1 s
Wall time: 24.1 s
```

Using AdaBoost to predict on the SPAM dataset

In [6]:

```
%%time
y_pred = clf.predict(X_test)
acc = accuracy(y_test, y_pred)
print(f"Accuracy of {round(100 * acc, 4)}%")
```

```
Accuracy of 91.4857%
CPU times: user 139 ms, sys: 8.05 ms, total: 147 ms
Wall time: 138 ms
```

1.a) Print a confusion matrix

In [7]:

```
print(confusion_matrix(y_test, y_pred))
```

```
[[656  41]
 [ 57 397]]
```

1.b) Is AdaBoost better when using stronger weak learners? Why or why not? Compare your results to using depth-2 decision trees.

In [8]:

```
%%time
clf = AdaBoost(num_trees=20, max_depth=2)
clf.fit(X_train, y_train)
```

```
100%|██████████| 20/20 [00:34<00:00,  1.74s/it]

CPU times: user 34.6 s, sys: 132 ms, total: 34.7 s
Wall time: 34.9 s
```

In [9]:

```
%%time
y_pred = clf.predict(X_test)
acc = accuracy(y_test, y_pred)
print(f"Accuracy of {round(100 * acc, 4)}%")
```

```
Accuracy of 90.3562%
CPU times: user 156 ms, sys: 1e+03 ns, total: 156 ms
Wall time: 155 ms
```