Machine Learning WS 19 - Assignment 7

December 4, 2018

Adrian Gruszczynski / Yann Salimi

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
class DTNode:
    def __init__(self, feature, threshold):
        self.feature = feature
        self.threshold = threshold
        self.left = None
        self.right = None

    def predict(self, x):
        if not self._is_initialized:
            raise ValueError('node is not initialized')
        if x[self.feature] < self.threshold:
            return self.left.predict(x)
        else:
            return self.right.predict(x)

    @property
    def _is_initialized(self):
        return self.left and self.right


class DTLeaf:
    def __init__(self, y):
        self.y = y

    def predict(self, _):
        return self.y
```

In [3]:
```
class AdaBoost:
    def __init__(self):
        self.classifiers = None
        self.alpha = None

    def fit(self, X, y, classifiers, M):
```

```
        n_classifiers = len(classifiers)
        n_samples, _ = X.shape

        self.classifiers = []
        self.alpha = []

        # initialize weights
        weights = np.ones(n_samples)

        # initialize scouting matrix
        S = np.empty((n_samples, n_classifiers))
        for k, classifier in enumerate(classifiers):
            S[:, k] = [2 * (classifier.predict(X[i]) == y[i]) - 1
                       for i in range(n_samples)]

        for m in range(M):
            # select classifier that minimizes W_e
            best_W_e = np.Inf
            classifier_data = None
            for k, classifier in enumerate(classifiers):
                W_e = np.sum(weights[np.where(S[:, k] == -1)])

                if W_e < best_W_e:
                    best_W_e = W_e
                    classifier_data = k, classifier

            # calculate alpha of the selected classifier
            W = np.sum(weights)
            e_m = best_W_e / W
            alpha = np.log((1 - e_m) / e_m) / 2

            # update weights
            k, classifier = classifier_data
            weights = weights * np.exp(-S[:, k] * alpha)

            self.classifiers.append(classifier)
            self.alpha.append(alpha)

    def predict(self, x):
        if not self._is_fitted:
            raise ValueError('AdaBoost is not fitted')
        score = 0
        for classifier, alpha in zip(self.classifiers, self.alpha):
            y_pred = classifier.predict(x)
            score += (2 * y_pred - 1) * alpha
        y_pred = 0 if score <= 0 else 1
        return y_pred
```

```
    @property
    def _is_fitted(self):
        return self.classifiers and self.alpha
```

In [4]:
```
def entropy(X):
    probabilities = np.bincount(X) / len(X)
    probabilities = probabilities[probabilities > 0]
    return -np.sum(probabilities * np.log2(probabilities))


def buildDT(X, y, n_features_sampled=None, max_depth=None):
    best_information_gain, node_data = 0, None
    n_samples, n_features = X.shape
    H_before_split = entropy(y)

    if n_features_sampled:
        features = np.random.choice(n_features,
                                    min(n_features, n_features_sampled),
                                    replace=False)
    else:
        features = np.arange(n_features)

    for feature in features:
        X_feature = X[:, feature]
        threshold = np.mean(X_feature)

        left_idx = X_feature < threshold
        right_idx = X_feature >= threshold

        y_left = y[left_idx]
        y_right = y[right_idx]
        p_y_left = len(y_left) / n_samples
        p_y_right = len(y_right) / n_samples

        H_after_split = p_y_left * entropy(y_left) + p_y_right * entropy(y_right)
        information_gain = H_before_split - H_after_split

        if information_gain > best_information_gain:
            best_information_gain = information_gain
            node_data = feature, threshold, left_idx, y_left, right_idx, y_right

    if max_depth == 0 or not best_information_gain:
        most_frequent_y = np.argmax(np.bincount(y))
        return DTLeaf(most_frequent_y)
    else:
        feature, threshold, left_idx, y_left, right_idx, y_right = node_data
        depth = None if max_depth is None else max_depth - 1
        node = DTNode(feature, threshold)
```

```
            node.left = buildDT(X[left_idx], y_left, n_features_sampled, depth)
            node.right = buildDT(X[right_idx], y_right, n_features_sampled, depth)
            return node


    def unison_shuffle(a, b):
        if len(a) != len(b):
            raise ValueError('array lengths do not match')
        idx = np.random.permutation(len(a))
        return a[idx], b[idx]


    def accuracy_score(y_true, y_pred):
        if y_true.shape != y_pred.shape:
            raise ValueError('array shapes do not match')
        return np.sum(np.equal(y_true, y_pred)) / len(y_true)
```

```
In [10]: if __name__ == '__main__':
             np.random.seed(12345)
             df = np.array(pd.read_csv('spambase.data', header=None))

             X, y = df[:, :-1], df[:, -1].astype(np.bool_)
             X, y = unison_shuffle(X, y)

             split = len(X) // 2

             X_train, y_train = X[:split], y[:split]
             X_val, y_val = X[split:], y[split:]

             _, n_features = X.shape
             n_features_sampled = int(np.sqrt(n_features))
             forest_size = 50
             decision_trees = []

             print("For stumps with depth=1")

             for tree_idx in range(forest_size):
                 sampled_idx = np.random.randint(0, high=split, size=split)
                 X_bootstrap, y_bootstrap = X_train[sampled_idx], y_train[sampled_idx]
                 decision_tree = buildDT(X_bootstrap, y_bootstrap,
                                         n_features_sampled=n_features_sampled,
                                         max_depth=1)    # 1 split
                 decision_trees.append(decision_tree)

             M = 50
             decision_forest = AdaBoost()
             decision_forest.fit(X_train, y_train, decision_trees, M)
```

```python
        y_val_pred = np.empty(y_val.shape)
        for i in range(len(X_val)):
            y_val_pred[i] = decision_forest.predict(X_val[i])

        val_accuracy = 100 * accuracy_score(y_val, y_val_pred)
        print('val accuracy: %.2f%%' % val_accuracy)

        y_train_pred = np.empty(y_train.shape)
        for i in range(len(X_train)):
            y_train_pred[i] = decision_forest.predict(X_train[i])

        train_accuracy = 100 * accuracy_score(y_train, y_train_pred)
        print('train accuracy: %.2f%%' % train_accuracy)

        print("For stumps with depth=2")

        for tree_idx in range(forest_size):
            sampled_idx = np.random.randint(0, high=split, size=split)
            X_bootstrap, y_bootstrap = X_train[sampled_idx], y_train[sampled_idx]
            decision_tree = buildDT(X_bootstrap, y_bootstrap,
                                    n_features_sampled=n_features_sampled,
                                    max_depth=2) # 2 splits
            decision_trees.append(decision_tree)

        M = 50
        decision_forest = AdaBoost()
        decision_forest.fit(X_train, y_train, decision_trees, M)

        y_val_pred = np.empty(y_val.shape)
        for i in range(len(X_val)):
            y_val_pred[i] = decision_forest.predict(X_val[i])

        val_accuracy = 100 * accuracy_score(y_val, y_val_pred)
        print('val accuracy: %.2f%%' % val_accuracy)

        y_train_pred = np.empty(y_train.shape)
        for i in range(len(X_train)):
            y_train_pred[i] = decision_forest.predict(X_train[i])

        train_accuracy = 100 * accuracy_score(y_train, y_train_pred)
        print('train accuracy: %.2f%%' % train_accuracy)
```

```
For stumps with depth=1
val accuracy: 82.75%
train accuracy: 81.78%
For stumps with depth=2
val accuracy: 88.44%
train accuracy: 88.78%
```

0.0.1 Adaboost performs better when using stronger weak learners. The reason for that is, that a commitee consisting of stronger experts can predict even better than commitee of weaker experts.