1003 HW6

Long Chen
lc3424@nyu.edu

April 18th, 2021

Q1

```
def compute_entropy(label_array):
    entropy = 0
    for k in np.unique(label_array):
        pmk = (sum(np.equal(k, label_array).astype(int)) /
    label_array.shape[0])[0]
        entropy += pmk * np.log(pmk)

    return -entropy

def compute_gini(label_array):
    gini = 0
    for k in np.unique(label_array):
        pmk = (sum(np.equal(k, label_array).astype(int)) /
    label_array.shape[0])[0]
        gini += pmk * (1 - pmk)

    return gini
```

Q2

```
class Decision_Tree(BaseEstimator):
    def __init__(self, split_loss_function, leaf_value_estimator,
                 depth=0, min_sample=5, max_depth=10, force_split=
    True):
        self.split_loss_function = split_loss_function
        self.leaf_value_estimator = leaf_value_estimator
        self.depth = depth
        self.min_sample = min_sample
        self.max_depth = max_depth
        self.is_leaf = False
        self.force_split = force_split

    '''
    Author note: in my algorithm, I forces the node to split (
    except depth or min_sample criteria is met) even if the split
    will result in higher loss. A hyperparameter is added to the
    constructor.
```

```
    '''
    def fit(self, x, y):
        # Corner case: reaches max_depth OR does not meet min_sample to split.
        if self.depth == self.max_depth or len(y) <= self.min_sample:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        self.split_id, self.split_value, split_pos, best_loss = self.find_best_feature_split(x, y)

        if self.force_split or best_loss < self.split_loss_function(y): # split
            # find left and right x, y
            lx, ly, rx, ry = None, None, None, None
            D = np.concatenate([x, y],1) # concatenated dataset for easier sorting
            D = np.array(sorted(D, key=lambda x: x[self.split_id]))
            lx, ly = D[:split_pos+1, :-1], D[:split_pos+1, -1].reshape(-1, 1)
            rx, ry = D[split_pos+1:, :-1], D[split_pos+1:, -1].reshape(-1, 1)

            # initialize left and right node
            self.left = Decision_Tree(self.split_loss_function, self.leaf_value_estimator,
                    depth=self.depth+1, min_sample=self.min_sample, max_depth=self.max_depth)
            self.right = Decision_Tree(self.split_loss_function, self.leaf_value_estimator,
                    depth=self.depth+1, min_sample=self.min_sample, max_depth=self.max_depth)
            self.left.fit(lx, ly)
            self.right.fit(rx, ry)
        else:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        return self

    def find_best_split(self, x_node, y_node, feature_id):
        split_value, best_loss, best_pos = None, float('inf'), -1
        D = np.concatenate([x_node[:, feature_id].reshape(-1,  1), y_node],1) # concatenated dataset for easier sorting
        D = np.array(sorted(D, key=lambda x: x[0]))
        # iterate through all datapoint intervals
        for pos in range(len(D)-1):
            lx, ly = D[:pos+1, 0], D[:pos+1, 1].reshape(-1, 1)
            rx, ry = D[pos+1:, 0], D[pos+1:, 1].reshape(-1, 1)

            ll = len(ly)*self.split_loss_function(ly)/len(y_node)
            rl = len(ry)*self.split_loss_function(ry)/len(y_node)

            # split condition
```

```
            if ll + rl < best_loss:
                split_value = (lx[-1] + rx[0]) / 2
                best_loss = ll + rl
                best_pos = pos

        return split_value, best_loss, best_pos

    def find_best_feature_split(self, x_node, y_node):
        split_id, split_value, best_loss, best_pos = None, None,
    float('inf'), -1

        for feature_id in range(x_node.shape[1]):
            sv, best_l, pos = self.find_best_split(x_node, y_node,
    feature_id)
            if best_l < best_loss:
                split_id, split_value, best_pos = feature_id, sv,
    pos
                best_loss = best_l

        return split_id, split_value, best_pos, best_loss

    def predict_instance(self, instance):
        if self.is_leaf:
            return self.value
        if instance[self.split_id] <= self.split_value:
            return self.left.predict_instance(instance)
        else:
            return self.right.predict_instance(instance)
```

Q3

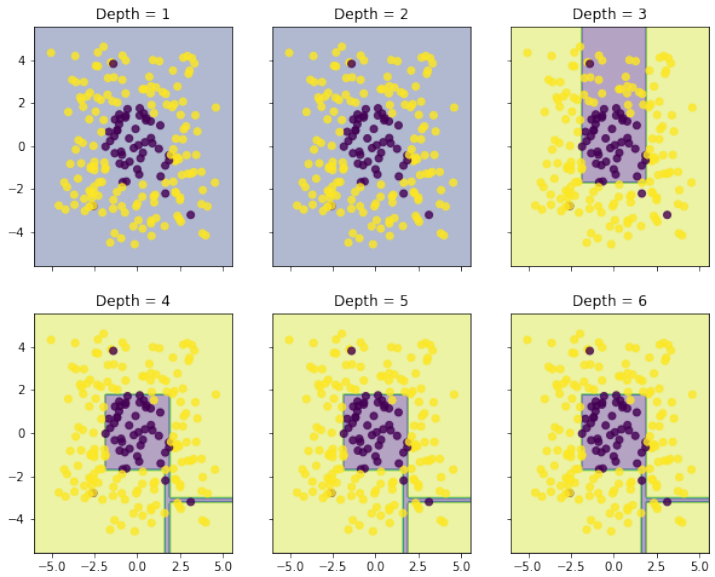

Figure 1: Q3 result.

Q4

```python
def mean_absolute_deviation_around_median(y):
    mae, m = 0, np.mean(y)
    for yi in y:
        mae += np.abs(yi - m)[0]

    return mae / len(y)
```

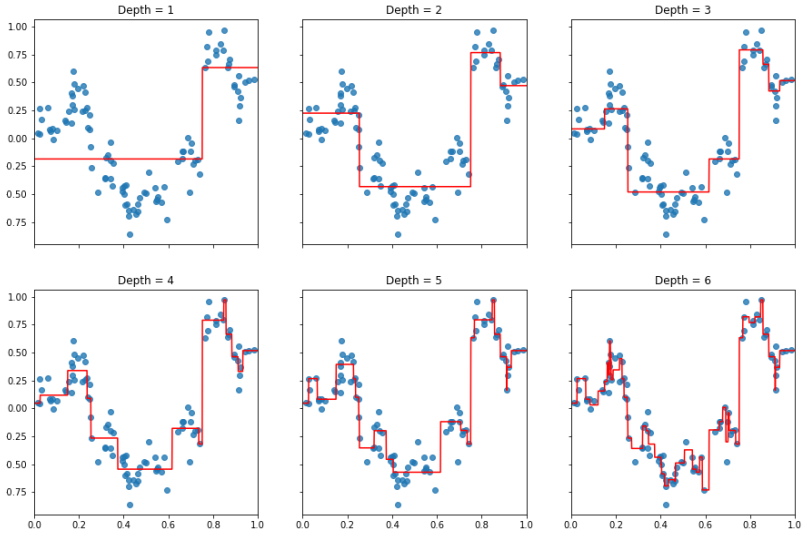

Figure 2: Q4 result.

Q5

```python
class gradient_boosting():
    def __init__(self, n_estimator, pseudo_residual_func,
    learning_rate=0.01,
                 min_sample=5, max_depth=5):

        self.n_estimator = n_estimator
        self.pseudo_residual_func = pseudo_residual_func
        self.learning_rate = learning_rate
        self.min_sample = min_sample
        self.max_depth = max_depth
        self.estimators = [] #will collect the n_estimator models

    def fit(self, train_data, train_target):
        # do f0
        res = self.pseudo_residual_func(train_target.reshape(-1),
    np.zeros(train_target.shape).reshape(-1))
        h0 = DecisionTreeRegressor(max_depth=self.max_depth,
    min_samples_leaf=self.min_sample, criterion='mse')
```

```
        h0.fit(train_data, res)
        self.estimators.append(h0)

        # do f[1, M]
        for m in range(self.n_estimator):
            step = 0
            for i in range(len(self.estimators)):
                step += self.learning_rate * self.estimators[i].
    predict(train_data)

            res = self.pseudo_residual_func(train_target.reshape
    (-1), step)
            hm = DecisionTreeRegressor(max_depth=self.max_depth,
    min_samples_leaf=self.min_sample, criterion='mse')
            hm.fit(train_data, res)
            self.estimators.append(hm)

    def predict(self, test_data):
        test_predict = self.learning_rate * sum([estimator.predict(
    test_data) for estimator in self.estimators])

        return test_predict
```

Q6

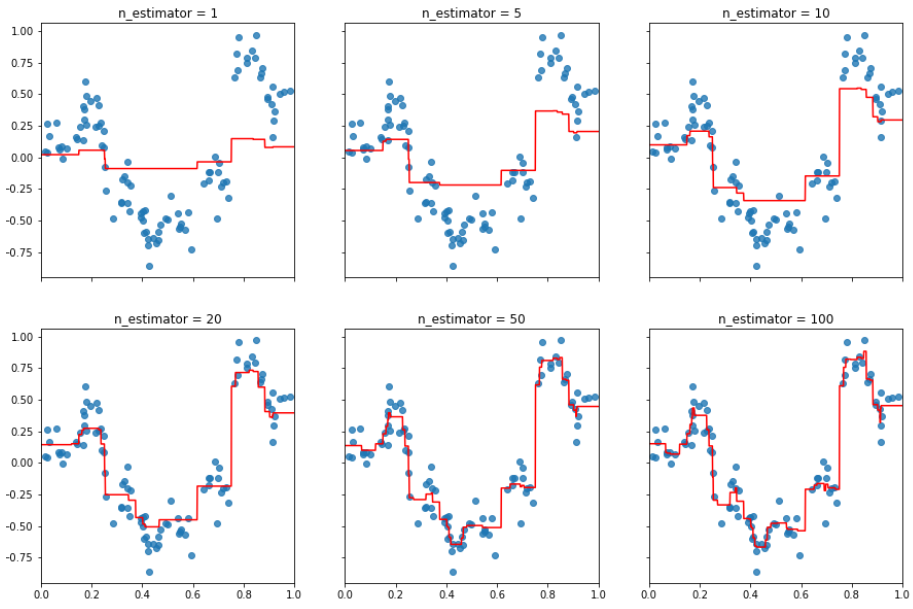

Figure 3: Q6 result.

Q7

$$
\begin{aligned}
-g_m &= -\left(\frac{\delta}{\delta f_{m-1}(x_j)} \sum_{i=1}^{m} \ell\left(y_i, f_{m-1}(x_i)\right) \right)_{j=1}^{n} \\
&= -\left(\frac{\delta}{\delta f_{m-1}(x_j)} \sum_{i=1}^{m} \ln\left(1 + \exp(-y_i f_{m-1}(x_i))\right) \right)_{j=1}^{n}
\end{aligned}
$$

Dimension of g_m is n.

Q8

$$
h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left((-g_m)_i - h(x_i)\right)^2
$$

Q9

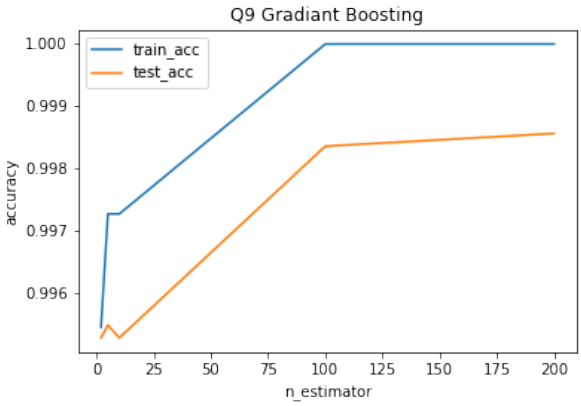

Figure 4: Q9 result.

Q10 (Optional)

Random forest is an ensemble method which uses bagged decision trees with modified tree-growing procedure to reduce the dependence between trees (a randomly chosen subset of features for splitting criteria). This will avoid dominance by strong features across the trees and will thus reduces variance (which is often high for a single decision tree).

Q11 (Optional)

```
for n_estimator in [2, 5, 10, 100, 200]:
    clf = RandomForestClassifier(criterion='entropy', max_depth=3,
    n_estimators=n_estimator)
    clf.fit(X_train, y_train)
    print('n_estimator: {:3}, Train accuracy: {:15}, test accuracy:
     {:15}'.format(n_estimator, accuracy_score(clf.predict(X_train)
    , y_train), accuracy_score(clf.predict(X_test), y_test)))
```

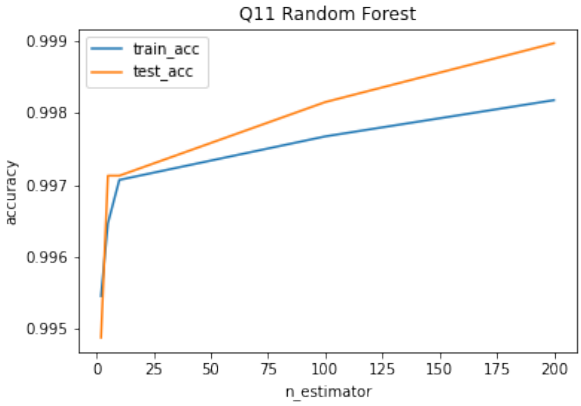

Figure 5: Q11 result.

Q12 (Optional)

We do observe less overfitting on the random forest results in comparison with gradiant boosted trees. The best training accuracy is achieved by gradient boosted tree (acc=1.0). This is as expected, since random forest tends to reduce variance and thus leading to less overfitting.

If I am understanding the question correctly, the "best performing method" refers to the model with best train_accuracy. A practical disadvantage is that if we have a new data point that is very different to any training samples, the prediction may be extremely bad.

In terms of test accuracy, random forest is a slightly ahead (0.9990 vs. 0.9986).