Homework6_DS-GA1003

April 20, 2021

Name: Zhuoyuan Xu (Kallen)

NetID: zx1137

1 Decision Tree Implementation

Problem 1

[1]:
```
import matplotlib.pyplot as plt
from itertools import product
import numpy as np
from collections import Counter
from sklearn.base import BaseEstimator, RegressorMixin, ClassifierMixin
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor,␣
↪export_graphviz
from sklearn.ensemble import GradientBoostingClassifier,␣
↪GradientBoostingRegressor, RandomForestClassifier
import graphviz

from IPython.display import Image

%matplotlib inline
```

[2]:
```
data_train = np.loadtxt('svm-train.txt')
data_test = np.loadtxt('svm-test.txt')
x_train, y_train = data_train[:, 0: 2], data_train[:, 2].reshape(-1, 1)
x_test, y_test = data_test[:, 0: 2], data_test[:, 2].reshape(-1, 1)
```

[3]:
```
# Change target to 0-1 label
y_train_label = np.array(list(map(lambda x: 1 if x > 0 else 0, y_train))).
↪reshape(-1, 1)
```

[4]:
```
def compute_entropy(label_array):
    '''
    Calulate the entropy of given label list

    :param label_array: a numpy array of binary labels shape = (n, 1)
    :return entropy: entropy value
```

```
    '''
    N = len(label_array)
    if N == 0:
        return 0

    P = np.sum(label_array)/N
    if P == 0 or (1-P) == 0:
        return 0
    entropy = -P * np.log2(P) - (1-P) * np.log2(1-P)

    return entropy

def compute_gini(label_array):
    '''
    Calulate the gini index of label list

    :param label_array: a numpy array of labels shape = (n, 1)
    :return gini: gini index value
    '''
    N = len(label_array)
    P = np.sum(label_array)/N

    gini = 1 - (P**2 + (1-P)**2)

    return gini
```

Problem 2

[31]:
```
class Decision_Tree(BaseEstimator):

    def __init__(self, split_loss_function, leaf_value_estimator,
                 depth=0, min_sample=5, max_depth=10):
        '''
        Initialize the decision tree classifier

        :param split_loss_function: method with args y returning loss
        :param leaf_value_estimator: method for estimating leaf value from␣
 ↪array of ys
        :param depth: depth indicator, default value is 0, representing root␣
 ↪node
        :param min_sample: an internal node can be splitted only if it contains␣
 ↪points more than min_smaple
        :param max_depth: restriction of tree depth.
        '''
        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

    def fit(self, x, y):
        '''
        This should fit the tree classifier by setting the values self.is_leaf,
        self.split_id (the index of the feature we want ot split on, if we're splitting),
        self.split_value (the corresponding value of that feature where the split is),
        and self.value, which is the prediction value if the tree is a leaf node.  If we are
        splitting the node, we should also init self.left and self.right to be Decision_Tree
        objects corresponding to the left and right subtrees. These subtrees should be fit on
        the data that fall to the left and right,respectively, of self.split_value.
        This is a recurisive tree building procedure.

        :param X: a numpy array of training data, shape = (n, m)
        :param y: a numpy array of labels, shape = (n, 1)

        :return self
        '''
        if len(y) <= self.min_sample or self.depth == self.max_depth or len(np.unique(y)) <= 1:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self

        self.find_best_feature_split(x, y)
        x_left = x[x[:, self.split_id] <= self.split_value]
        y_left = y[x[:, self.split_id] <= self.split_value]
        x_right = x[x[:, self.split_id] > self.split_value]
        y_right = y[x[:, self.split_id] > self.split_value]
        N = len(y)

        loss = (self.split_loss_function(y_left)*len(y_left) + self.split_loss_function(y_right)*len(y_right))/N
        if loss < self.split_loss_function(y):
            self.left = Decision_Tree(self.split_loss_function, self.leaf_value_estimator, self.depth+1, self.min_sample, self.max_depth)
            self.right = Decision_Tree(self.split_loss_function, self.leaf_value_estimator, self.depth+1, self.min_sample, self.max_depth)
```

```
            self.left.fit(x_left, y_left)
            self.right.fit(x_right, y_right)
        else:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
        return self

    def find_best_split(self, x_node, y_node, feature_id):
        '''
        For feature number feature_id, returns the optimal splitting point
        for data X_node, y_node, and corresponding loss
        :param X: a numpy array of training data, shape = (num_samples,␣
↪num_features)
        :param y: a numpy array of labels, shape = (n_node, 1)
        '''

        N = len(y_node)
        sort_x = np.sort(x_node[:, feature_id])
        loss = []
        val = []

        for i in range(len(x_node)-1):
            split_point = 0.5*(sort_x[i]+sort_x[i+1])
            val.append(split_point)
            left = y_node[x_node[:, feature_id] <= split_point]
            right = y_node[x_node[:, feature_id] > split_point]
            l = (self.split_loss_function(left)*len(left) + self.
↪split_loss_function(right)*len(right))/N
            loss.append(l)

        best_loss = min(loss)
        split_value = val[np.argmin(loss)]

        return split_value, best_loss

    def find_best_feature_split(self, x_node, y_node):
        '''
        Returns the optimal feature to split and best splitting point
        for data X_node, y_node.
        :param X: a numpy array of training data, shape = (num_samples,␣
↪num_features)
        :param y: a numpy array of labels, shape = (n_node, 1)
        '''
        split = []
        for f_id, c in enumerate(x_node.T):
            split_value, best_loss = self.find_best_split(x_node, y_node, f_id)
            split.append([f_id, split_value, best_loss])
```

```
        min_split = min(split, key = lambda x:x[2])

        self.split_id = min_split[0]
        self.split_value = min_split[1]

    def predict_instance(self, instance):
        '''
        Predict label by decision tree

        :param instance: a numpy array with new data, shape (1, m)

        :return whatever is returned by leaf_value_estimator for leaf␣
↪containing 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)
```

Problem 3

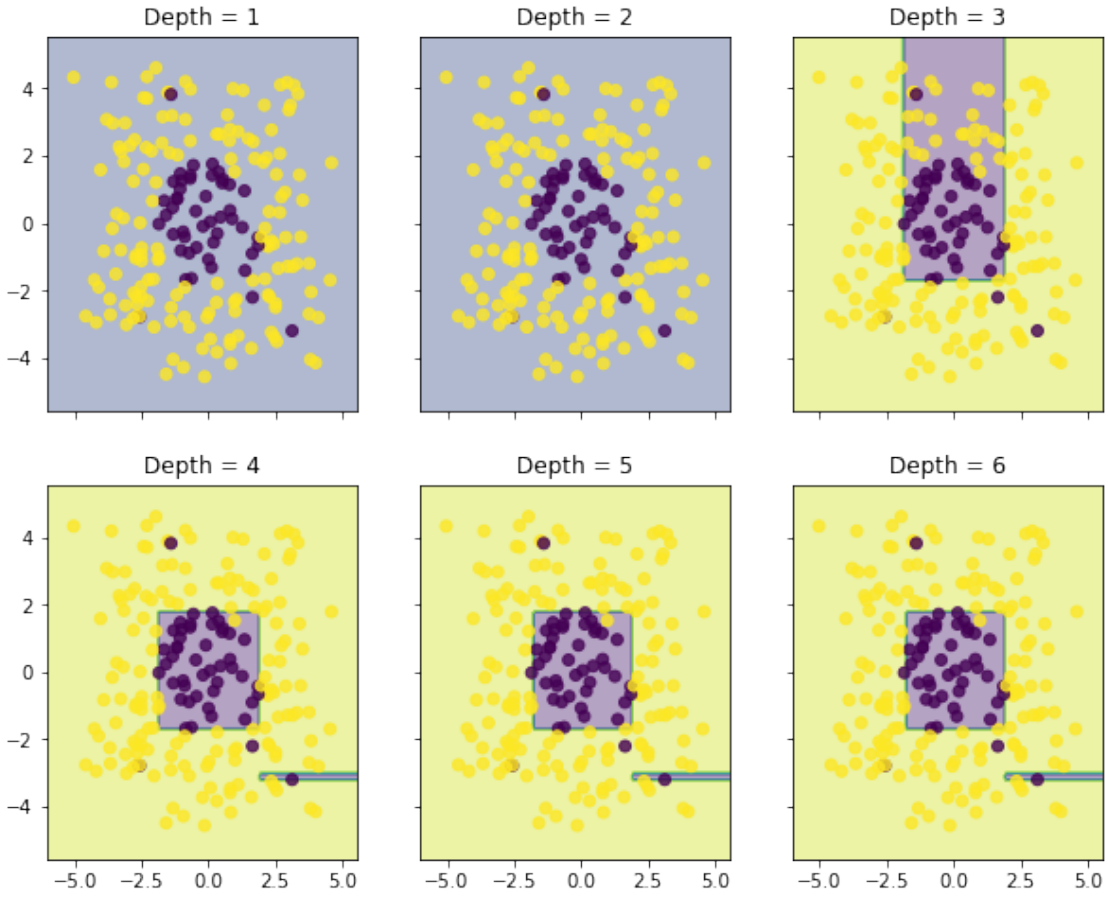
[32]:
```
def most_common_label(y):
    '''
    Find most common label
    '''
    label_cnt = Counter(y.reshape(len(y)))
    label = label_cnt.most_common(1)[0][0]
    return label
```

[33]:
```
class Classification_Tree(BaseEstimator, ClassifierMixin):

    loss_function_dict = {
        'entropy': compute_entropy,
        'gini': compute_gini
    }

    def __init__(self, loss_function='entropy', min_sample=5, max_depth=10):
        '''
        :param loss_function(str): loss function for splitting internal node
        '''

        self.tree = Decision_Tree(self.loss_function_dict[loss_function],
                                  most_common_label,
                                  0, min_sample, max_depth)
```

```
    def fit(self, X, y=None):
        self.tree.fit(X,y)
        return self

    def predict_instance(self, instance):
        value = self.tree.predict_instance(instance)
        return value
```

[34]:
```
# Training classifiers with different depth
clf1 = Classification_Tree(max_depth=1, min_sample=2)
clf1.fit(x_train, y_train_label)

clf2 = Classification_Tree(max_depth=2, min_sample=2)
clf2.fit(x_train, y_train_label)

clf3 = Classification_Tree(max_depth=3, min_sample=2)
clf3.fit(x_train, y_train_label)

clf4 = Classification_Tree(max_depth=4, min_sample=2)
clf4.fit(x_train, y_train_label)

clf5 = Classification_Tree(max_depth=5, min_sample=2)
clf5.fit(x_train, y_train_label)

clf6 = Classification_Tree(max_depth=6, min_sample=2)
clf6.fit(x_train, y_train_label)

# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(2, 3, sharex='col', sharey='row', figsize=(10, 8))

for idx, clf, tt in zip(product([0, 1], [0, 1, 2]),
                        [clf1, clf2, clf3, clf4, clf5, clf6],
                        ['Depth = {}'.format(n) for n in range(1, 7)]):

    Z = np.array([clf.predict_instance(x) for x in np.c_[xx.ravel(), yy.
 ↪ravel()]])
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.4)
    axarr[idx[0], idx[1]].scatter(x_train[:, 0], x_train[:, 1],␣
 ↪c=y_train_label[:,0], alpha=0.8)
    axarr[idx[0], idx[1]].set_title(tt)
```

```
plt.show()
```

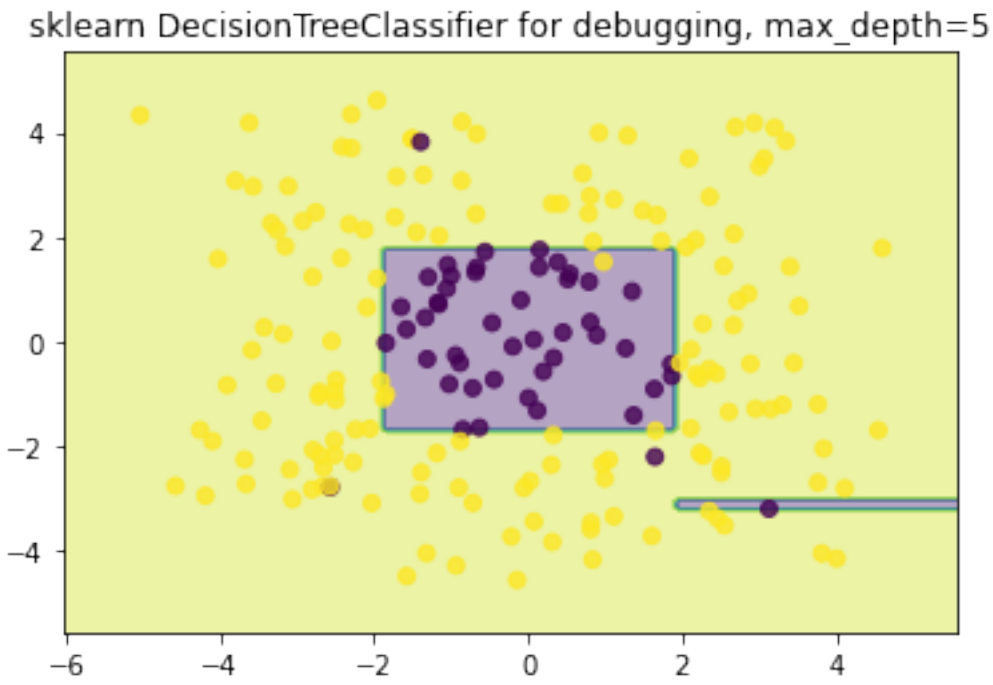


[21]:
```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=5,␣
↪min_samples_split=2)
clf.fit(x_train, y_train_label)
export_graphviz(clf, out_file='tree_classifier.dot')
```

[22]:
```
# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = np.array([clf.predict(x[np.newaxis,:]) for x in np.c_[xx.ravel(), yy.
↪ravel()]])
Z = Z.reshape(xx.shape)
plt.figure()
```

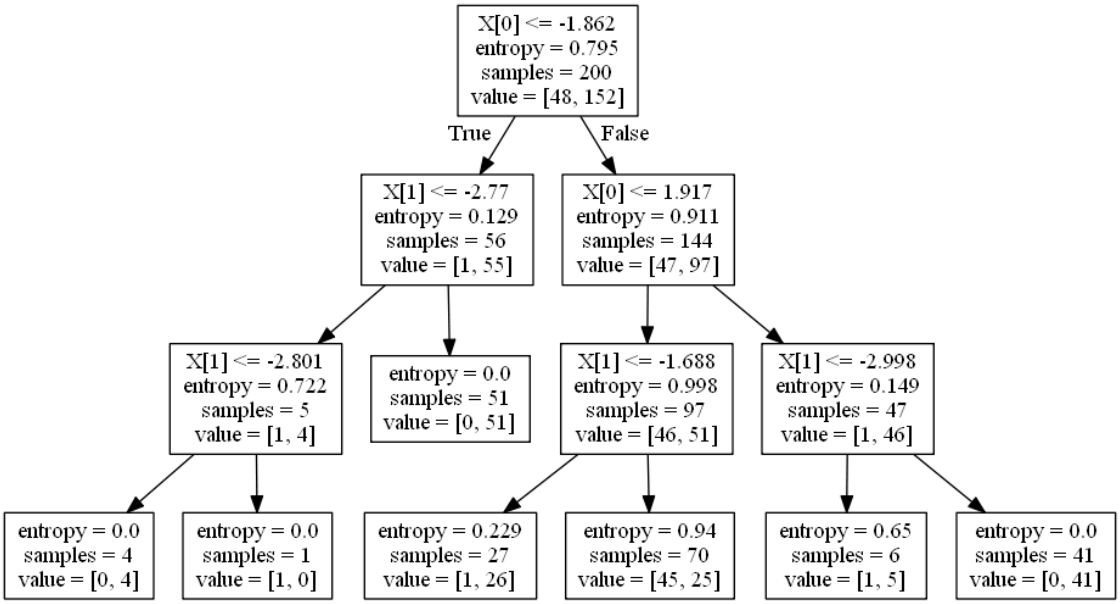
```
plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(x_train[:, 0], x_train[:, 1],
c=y_train_label[:,0], alpha=0.8)
plt.title('sklearn DecisionTreeClassifier for debugging, max_depth=5')
```

[22]: `Text(0.5, 1.0, 'sklearn DecisionTreeClassifier for debugging, max_depth=5')`



[155]:
```
# Visualize decision tree
!dot -Tpng tree_classifier.dot -o tree_classifier.png
Image(filename='tree_classifier.png')
```

[155]:



Problem 4

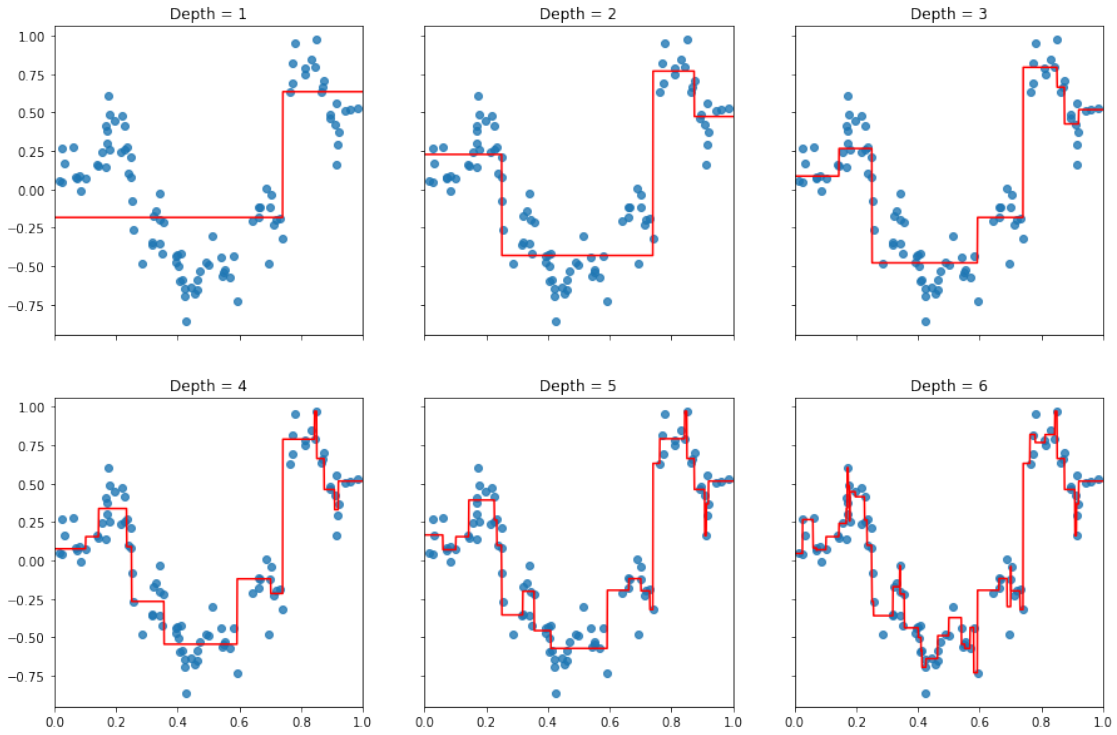
[156]:
```
# Regression Tree Specific Code
def mean_absolute_deviation_around_median(y):
    '''
    Calulate the mean absolute deviation around the median of a given target␣
 ↪list

    :param y: a numpy array of targets shape = (n, 1)
    :return mae
    '''
    if len(np.unique(y)) < 1:
        return 0

    median = np.median(y)
    N = len(y)

    mae = np.sum([np.abs(i - median) for i in y])/N
    return mae
```

[157]:
```
class Regression_Tree():
    '''
    :attribute loss_function_dict: dictionary containing the loss functions␣
 ↪used for splitting
    :attribute estimator_dict: dictionary containing the estimation functions␣
 ↪used in leaf nodes
    '''
```

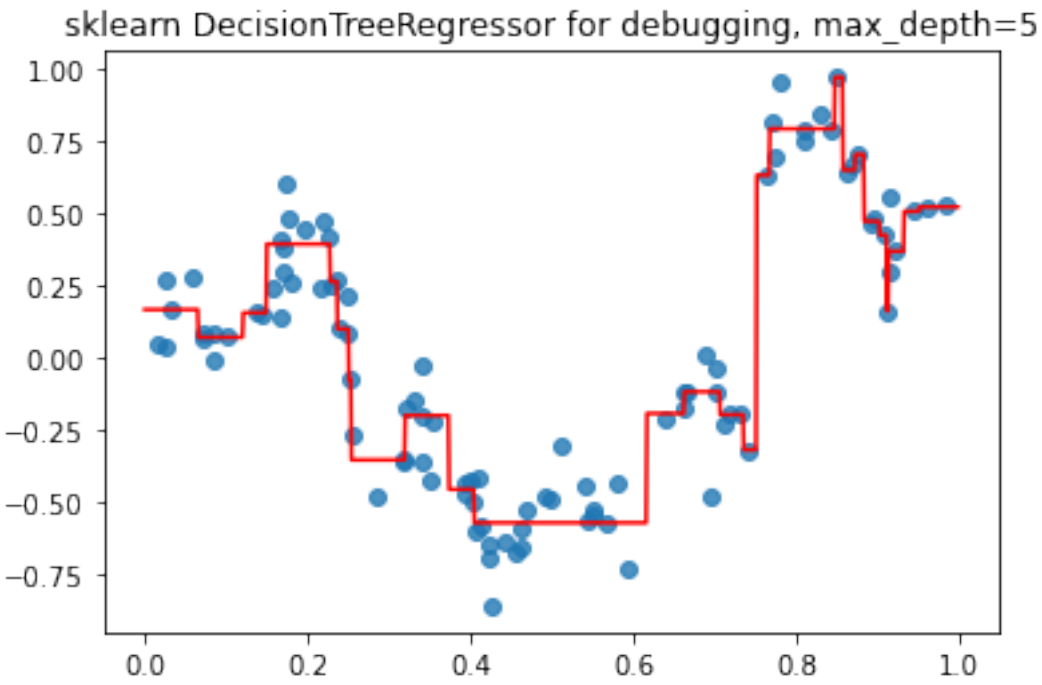
```
    loss_function_dict = {
        'mse': np.var,
        'mae': mean_absolute_deviation_around_median
    }

    estimator_dict = {
        'mean': np.mean,
        'median': np.median
    }

    def __init__(self, loss_function='mse', estimator='mean', min_sample=5,␣
↪max_depth=10):
        '''
        Initialize Regression_Tree
        :param loss_function(str): loss function used for splitting internal␣
↪nodes
        :param estimator(str): value estimator of internal node
        '''

        self.tree = Decision_Tree(self.loss_function_dict[loss_function],
                                  self.estimator_dict[estimator],
                                  0, min_sample, max_depth)

    def fit(self, X, y=None):
        self.tree.fit(X,y)
        return self

    def predict_instance(self, instance):
        value = self.tree.predict_instance(instance)
        return value
```

[136]:
```
data_krr_train = np.loadtxt('krr-train.txt')
data_krr_test = np.loadtxt('krr-test.txt')
x_krr_train, y_krr_train = data_krr_train[:,0].reshape(-1,1),data_krr_train[:
↪,1].reshape(-1,1)
x_krr_test, y_krr_test = data_krr_test[:,0].reshape(-1,1),data_krr_test[:,1].
↪reshape(-1,1)

# Training regression trees with different depth
clf1 = Regression_Tree(max_depth=1,  min_sample=3, loss_function='mae',␣
↪estimator='median')
clf1.fit(x_krr_train, y_krr_train)

clf2 = Regression_Tree(max_depth=2,  min_sample=3, loss_function='mae',␣
↪estimator='median')
clf2.fit(x_krr_train, y_krr_train)
```

```
clf3 = Regression_Tree(max_depth=3,  min_sample=3, loss_function='mae',␣
↪estimator='median')
clf3.fit(x_krr_train, y_krr_train)

clf4 = Regression_Tree(max_depth=4,  min_sample=3, loss_function='mae',␣
↪estimator='median')
clf4.fit(x_krr_train, y_krr_train)

clf5 = Regression_Tree(max_depth=5,  min_sample=3, loss_function='mae',␣
↪estimator='median')
clf5.fit(x_krr_train, y_krr_train)

clf6 = Regression_Tree(max_depth=10,  min_sample=3, loss_function='mae',␣
↪estimator='median')
clf6.fit(x_krr_train, y_krr_train)

plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)

f2, axarr2 = plt.subplots(2, 3, sharex='col', sharey='row', figsize=(15, 10))

for idx, clf, tt in zip(product([0, 1], [0, 1, 2]),
                        [clf1, clf2, clf3, clf4, clf5, clf6],
                        ['Depth = {}'.format(n) for n in range(1, 7)]):

    y_range_predict = np.array([clf.predict_instance(x) for x in x_range]).
↪reshape(-1, 1)

    axarr2[idx[0], idx[1]].plot(x_range, y_range_predict, color='r')
    axarr2[idx[0], idx[1]].scatter(x_krr_train, y_krr_train, alpha=0.8)
    axarr2[idx[0], idx[1]].set_title(tt)
    axarr2[idx[0], idx[1]].set_xlim(0, 1)
plt.show()
```



[143]:
```
clf = DecisionTreeRegressor(criterion="mae", max_depth=5, min_samples_split=3)
clf.fit(x_krr_train, y_krr_train)
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)
y_range_predict = clf.predict(x_range)
plt.plot(x_range, y_range_predict, color='r')
plt.scatter(x_krr_train, y_krr_train, alpha=0.8)
plt.title('sklearn DecisionTreeRegressor for debugging, max_depth=5')
export_graphviz(dtr, out_file='tree_regressor.dot')
```



2 Ensembling

Problem 5

[102]:
```
#Pseudo-residual function.

def pseudo_residual_L2(train_target, train_predict):
    '''
    Compute the pseudo-residual based on current predicted value.
    '''
    return train_target - train_predict
```

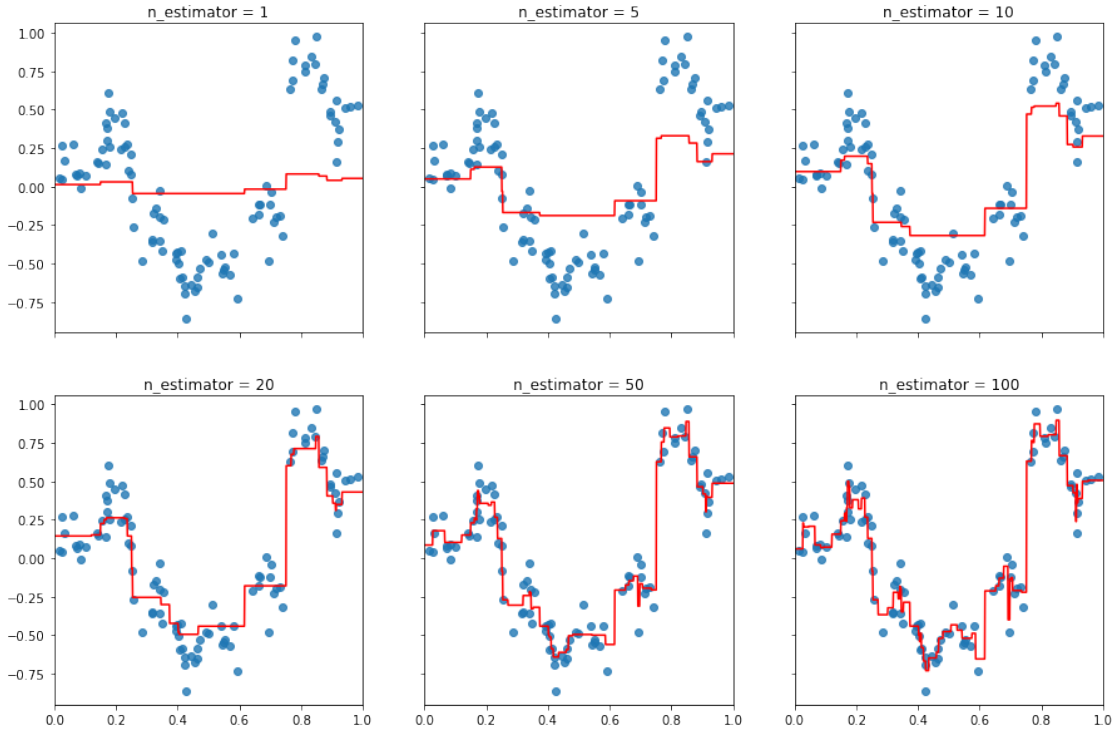
[103]:
```
class gradient_boosting():
    '''
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func, learning_rate=0.01,
                 min_sample=5, max_depth=5):
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounds of␣
↪gradient boosting)
```

```
        :pseudo_residual_func: function used for computing pseudo-residual␣
↪between training labels and predicted labels at each iteration
        :param learning_rate: step size of gradient descent
        '''
        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):
        '''
        Fit gradient boosting model
        :train_data array of inputs of size (n_samples, m_features)
        :train_target array of outputs of size (n_samples,)
        '''
        train_predict = np.zeros(len(train_target))
        for i in range(self.n_estimator):
            regressor = DecisionTreeRegressor(max_depth=self.max_depth,␣
↪min_samples_split=self.min_sample, criterion="mse")
            residuals = self.pseudo_residual_func(train_target, train_predict)
            regressor.fit(train_data, residuals)
            train_predict = train_predict + self.learning_rate * regressor.
↪predict(train_data)

            self.estimators.append(regressor)

    def predict(self, test_data):
        '''
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''
        test_predict = np.zeros(len(test_data))
        for regressor in self.estimators:
            test_predict = test_predict + self.learning_rate * regressor.
↪predict(test_data)

        return test_predict
```

Problem 6

[104]:
```
plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)

f2, axarr2 = plt.subplots(2, 3, sharex='col', sharey='row', figsize=(15, 10))
```

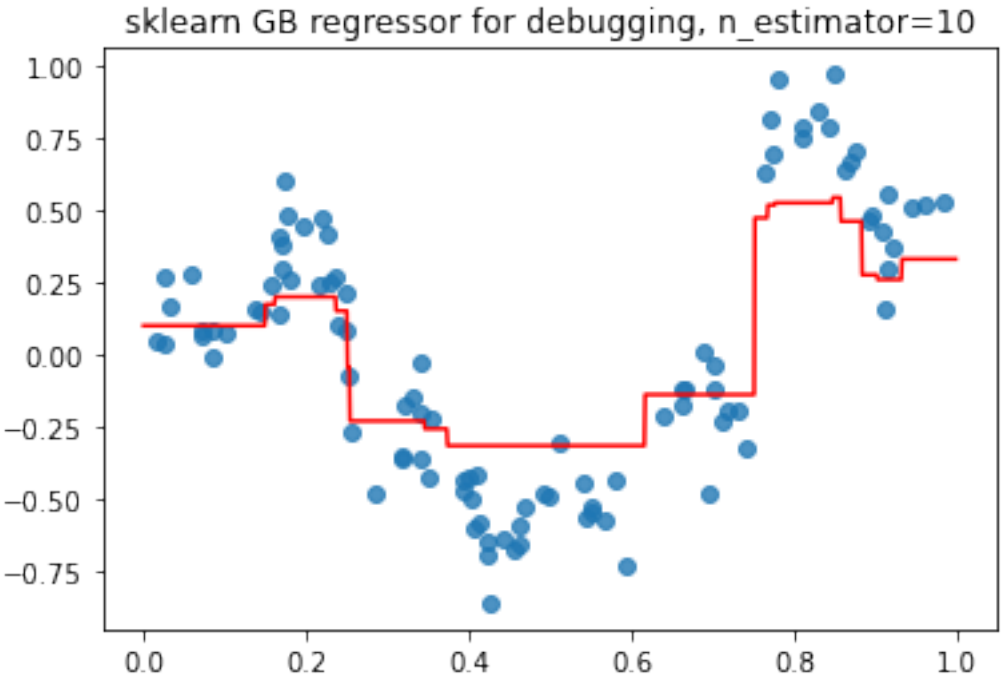
```
for idx, i, tt in zip(product([0, 1], [0, 1, 2]),
                      [1, 5, 10, 20, 50, 100],
                      ['n_estimator = {}'.format(n) for n in [1, 5, 10, 20,␣
↪50, 100]]):

    gbm_1d = gradient_boosting(n_estimator=i,␣
↪pseudo_residual_func=pseudo_residual_L2,
                               max_depth=3, learning_rate=0.1)
    gbm_1d.fit(x_krr_train, y_krr_train[:,0])

    y_range_predict = gbm_1d.predict(x_range)

    axarr2[idx[0], idx[1]].plot(x_range, y_range_predict, color='r')
    axarr2[idx[0], idx[1]].scatter(x_krr_train, y_krr_train, alpha=0.8)
    axarr2[idx[0], idx[1]].set_title(tt)
    axarr2[idx[0], idx[1]].set_xlim(0, 1)
```



[106]:
```
regressor = GradientBoostingRegressor(learning_rate=0.1, criterion="mse",␣
↪min_samples_split=5, max_depth=3, n_estimators=10)
regressor.fit(x_krr_train, y_krr_train[:,0])
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)
y_pred = regressor.predict(x_range)
```

```
plt.plot(x_range, y_pred, color='r')
plt.scatter(x_krr_train, y_krr_train, alpha=0.8)
plt.title('sklearn GB regressor for debugging, n_estimator=10')
```

[106]: Text(0.5, 1.0, 'sklearn GB regressor for debugging, n_estimator=10')



Problem 7

Given $Y = \{-1, 1\}$, $m = yf(x)$ and the logistic loss function $l(m) = ln(1 + e^{-m})$,

$$\begin{aligned} -g_m &= -(\frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} l(y_i, f_{m-1}(x_i)))_{j=1}^{n} \\ &= (\frac{y_j}{1 + e^{y_j f_{m-1}(x_j)}})_{j=1}^{n} \\ &= (\frac{y_1}{1 + e^{y_1 f_{m-1}(x_1)}}, ..., \frac{y_n}{1 + e^{y_n f_{m-1}(x_n)}}) \end{aligned}$$

The dimension of g_m is n.

Problem 8

$$h_m = arg \min_{h \in H} \sum_{i=1}^{n} (\frac{y_i}{1 + e^{y_i f_{m-1}(x_i)}} - h(x_i))^2$$

Problem 9

[35]:
```
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state
```

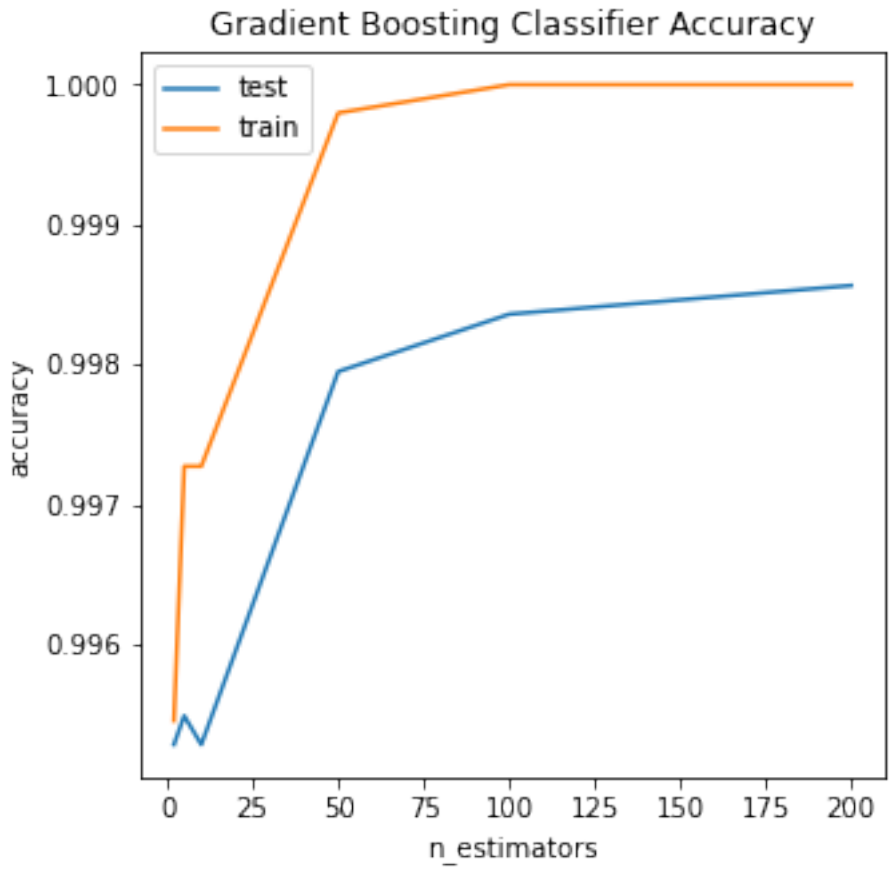
[36]:
```
def pre_process_mnist_01():
    """
    Load the mnist datasets, selects the classes 0 and 1
    and normalize the data.
    Args: none
    Outputs:
        X_train: np.array of size (n_training_samples, n_features)
        X_test: np.array of size (n_test_samples, n_features)
        y_train: np.array of size (n_training_samples)
        y_test: np.array of size (n_test_samples)
    """
    X_mnist, y_mnist = fetch_openml('mnist_784', version=1,
                                    return_X_y=True, as_frame=False)
    indicator_01 = (y_mnist == '0') + (y_mnist == '1')
    X_mnist_01 = X_mnist[indicator_01]
    y_mnist_01 = y_mnist[indicator_01]
    X_train, X_test, y_train, y_test = train_test_split(X_mnist_01, y_mnist_01,
                                                        test_size=0.33,
                                                        shuffle=False)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    y_test = 2 * np.array([int(y) for y in y_test]) - 1
    y_train = 2 * np.array([int(y) for y in y_train]) - 1
    return X_train, X_test, y_train, y_test
```

[37]:
```
X_train, X_test, y_train, y_test = pre_process_mnist_01()
```

[38]:
```
estimators = [2, 5, 10, 50, 100, 200]
gb_accuracy_test = []
gb_accuracy_train = []
for e in estimators:
    classifier = GradientBoostingClassifier(loss="deviance", n_estimators=e,␣
 ↪max_depth=3)
    classifier.fit(X_train, y_train)
    gb_accuracy_test.append(classifier.score(X_test, y_test))
    gb_accuracy_train.append(classifier.score(X_train, y_train))
```

[39]:
```
fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(estimators, gb_accuracy_test, label='test')
ax.plot(estimators, gb_accuracy_train, label='train')
ax.set_title('Gradient Boosting Classifier Accuracy')
ax.set_xlabel('n_estimators')
ax.set_ylabel('accuracy')
ax.legend()
```

[39]:
```
<matplotlib.legend.Legend at 0x1de67bc6bb0>
```



3 Classification of Images with Random Forests

Problem 10

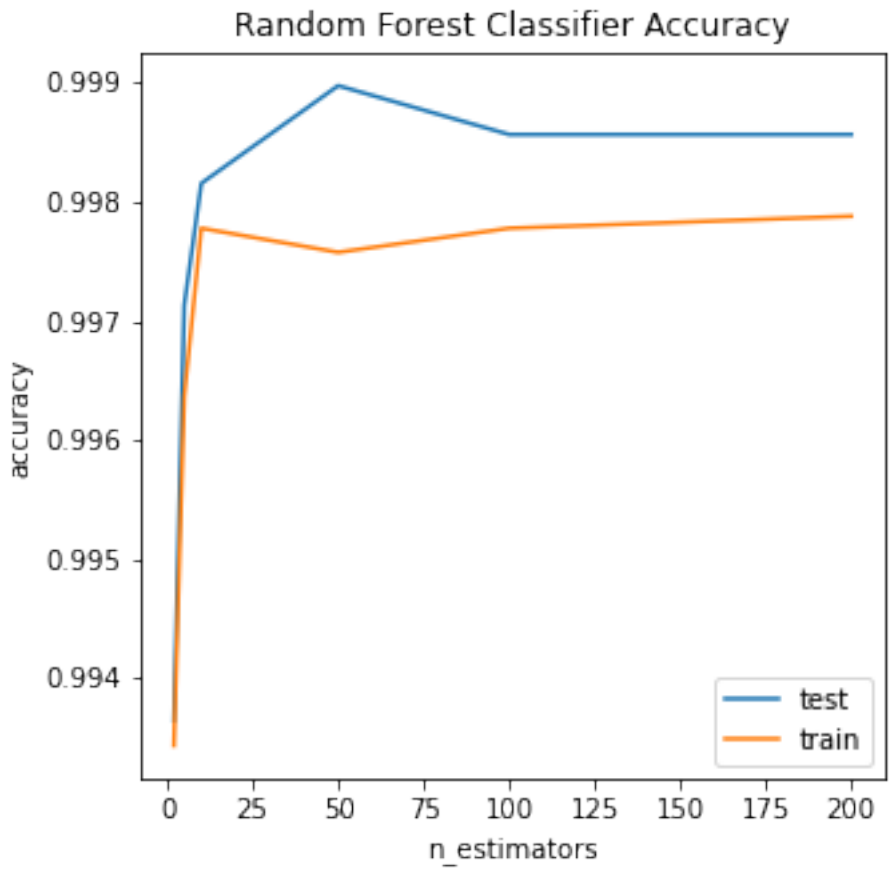
Random forests randomly select subsets of data to independently train decision trees that are used as parallel estimators. For classification problems, the result of the random forests would be based on majority votes of all decision trees. The class which wins most votes would become the result of the algorithm. For regression problems, the prediction of the model is the average of preditions from individual decision trees.

Problem 11

[46]:
```
estimators = [2, 5, 10, 50, 100, 200]
rf_accuracy_test = []
rf_accuracy_train = []
estimator_result = []
for e in estimators:
    classifier = RandomForestClassifier(criterion="entropy", n_estimators=e,␣
 ↪max_depth=3)
    classifier.fit(X_train, y_train)
    estimator_result.append(classifier)

    rf_accuracy_test.append(classifier.score(X_test, y_test))
    rf_accuracy_train.append(classifier.score(X_train, y_train))
```

[47]:
```
fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(estimators, rf_accuracy_test, label='test')
ax.plot(estimators, rf_accuracy_train, label='train')
ax.set_title('Random Forest Classifier Accuracy')
ax.set_xlabel('n_estimators')
ax.set_ylabel('accuracy')
ax.legend()
```

[47]:
```
<matplotlib.legend.Legend at 0x1de00160ee0>
```

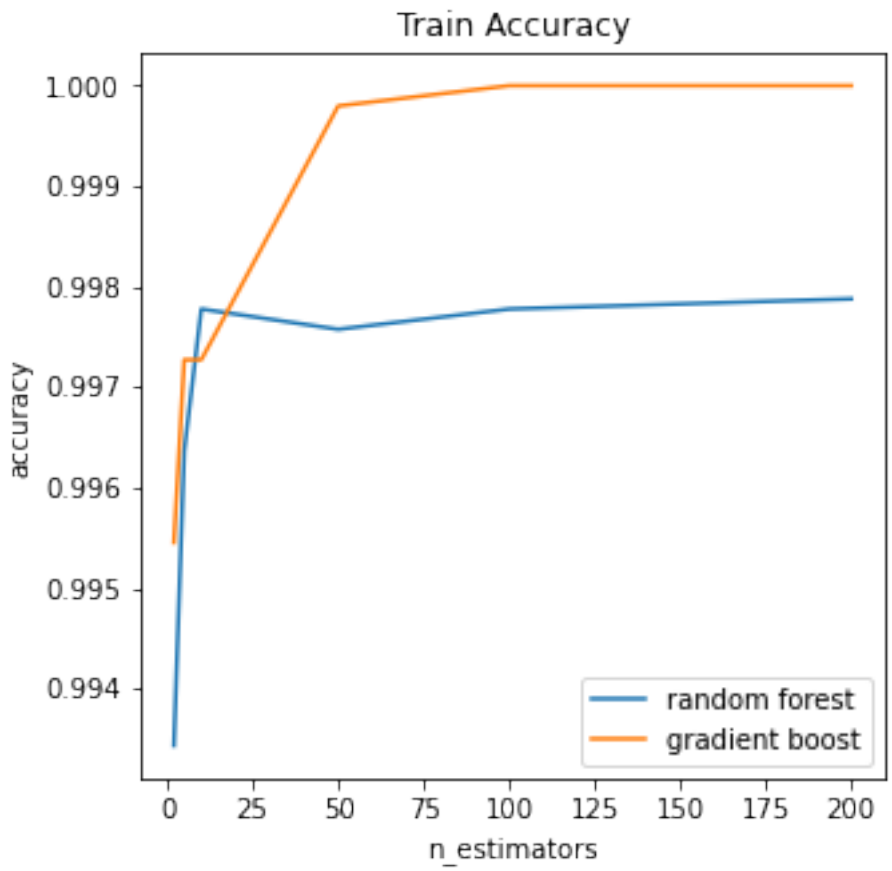


Problem 12

Generally, gradient boosted trees are more sensitive to overfitting given noisy data than random forests. Gradient boosted trees can also overfit if n_estimators is large. The train accuracies of the 2 methods using the same dataset are plotted below:

[48]:
```
fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(estimators, rf_accuracy_train, label='random forest')
ax.plot(estimators, gb_accuracy_train, label='gradient boost')
ax.set_title('Train Accuracy')
ax.set_xlabel('n_estimators')
ax.set_ylabel('accuracy')
ax.legend()
```

[48]: <matplotlib.legend.Legend at 0x1de01195940>

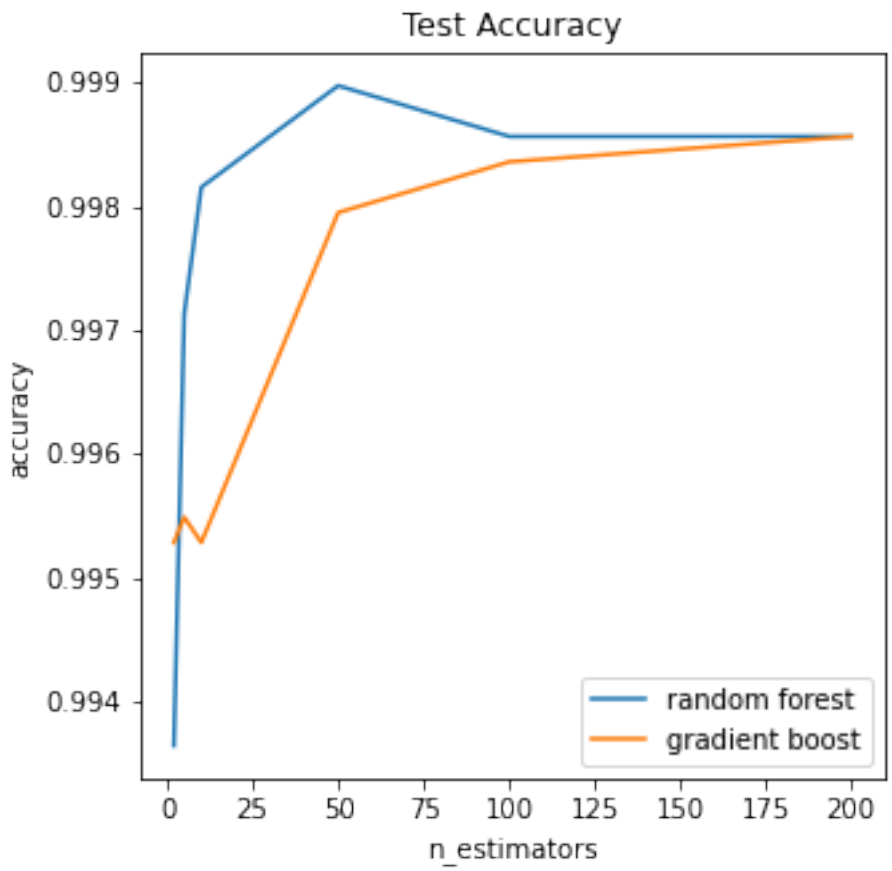


Gradient boosted trees gives the best train accuracies overall, and this result is expected because higher train accuracies close to 1 can be a sign of overfitting. This is consistent with our general remark above. In real world, we often have noisy data, and thus when we use gradient boosted trees we need to be careful for overfitting. Also, gradient boosted trees need careful tuning of hyperparameters, and the tuning can be harder than that of random forests. GB trees often take longer time to train as well.

The test accuracies of the 2 methods using the same dataset are plotted below:

[49]:
```
fig, ax = plt.subplots(figsize=(5, 5))
ax.plot(estimators, rf_accuracy_test, label='random forest')
ax.plot(estimators, gb_accuracy_test, label='gradient boost')
ax.set_title('Test Accuracy')
ax.set_xlabel('n_estimators')
ax.set_ylabel('accuracy')
ax.legend()
```

[49]:
```
<matplotlib.legend.Legend at 0x1de6801d520>
```



Overall, random forests have higher test accuracies than gradient boosted trees for our specific dataset.