Homework 6

Name: Zheyuan Hu

NetID: zh2095

Decision Tree Implementation

Load Data

In [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_graph
from sklearn.ensemble import GradientBoostingClassifier, GradientBoostingRegressor,
import graphviz

from IPython.display import Image

%matplotlib inline
```

In [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)
```

In [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(-
```

Q1

In [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
    '''
    # Your code goes here
    K = np.unique(label_array)
    N = label_array.shape[0]
    entropy = 0
    for k in K:
        p = np.count_nonzero(label_array == k)/N
        entropy -= p*np.log(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
    '''
    # Your code goes here
    K = np.unique(label_array)
    N = label_array.shape[0]
    gini = 0
    for k in K:
        p = np.count_nonzero(label_array == k)/N
        gini += p*(1-p)
    return gini
```

Q2

In [5]:

```
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 (X, 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
        :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 splittin
        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
        splitting the node, we should also init self.left and self.right to be Decisio
        objects corresponding to the left and right subtrees. These subtrees should be
        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
        '''
        # Your code goes here
        # if reach min sample or max depth, then end up spliting
        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

        split_id, split_value = self.find_best_feature_split(x, y)
        if split_id != None and split_value != None:
            # update depth after each spliting
            self.depth += 1
            # initialize the two trees to fit the left and right data
            self.left = Decision_Tree(self.split_loss_function, self.leaf_value_estim
```

```
            self.right = Decision_Tree(self.split_loss_function, self.leaf_value_esti
            # split with the optimal feature and the corresponding value
            idx_left = np.where(x[:,split_id] <= split_value)
            idx_right = np.where(x[:,split_id] > split_value)
            x_left = x[idx_left]
            x_right = x[idx_right]
            y_left = y[idx_left]
            y_right = y[idx_right]
            # fit the tree on the left and right nodes
            self.left.fit(x_left, y_left)
            self.right.fit(x_right, y_right)
            self.split_id = split_id
            self.split_value = split_value
        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 = (n_node)
        :param y: a numpy array of labels, shape = (n_node, 1)
        '''
        # Your code
        n_node = x_node.shape[0]
        x = x_node[:, feature_id]
        best_loss = self.split_loss_function(y_node)
        split_value = None
        for i in range(n_node):
            idx_left = np.where(x <= x[i])
            idx_right = np.where(x > x[i])
            x_left = x_node[idx_left]
            x_right = x_node[idx_right]
            y_left = y_node[idx_left]
            y_right = y_node[idx_right]
            left_loss = self.split_loss_function(y_left)
            right_loss = self.split_loss_function(y_right)
            # loss is the weighted average of left and right loss
            loss = (len(y_left)*left_loss + len(y_right)*right_loss)/len(y_node)

            if loss < best_loss:
                best_loss = loss
                split_value = x[i]
        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 = (n_node, 1)
        :param y: a numpy array of labels, shape = (n_node, 1)
        '''

        # Your code
        m = x_node.shape[1]
        best_loss = self.split_loss_function(y_node)
        split_id = None
        split_value = None
        for feature_id in range(m):
            value, loss = self.find_best_split(x_node, y_node, feature_id)
```

```
            if loss < best_loss:
                split_id = feature_id
                split_value = value
                best_loss = loss
        return split_id, split_value


    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 insta
        '''
        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

Decision Tree Classifier

In [6]:

```
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
```

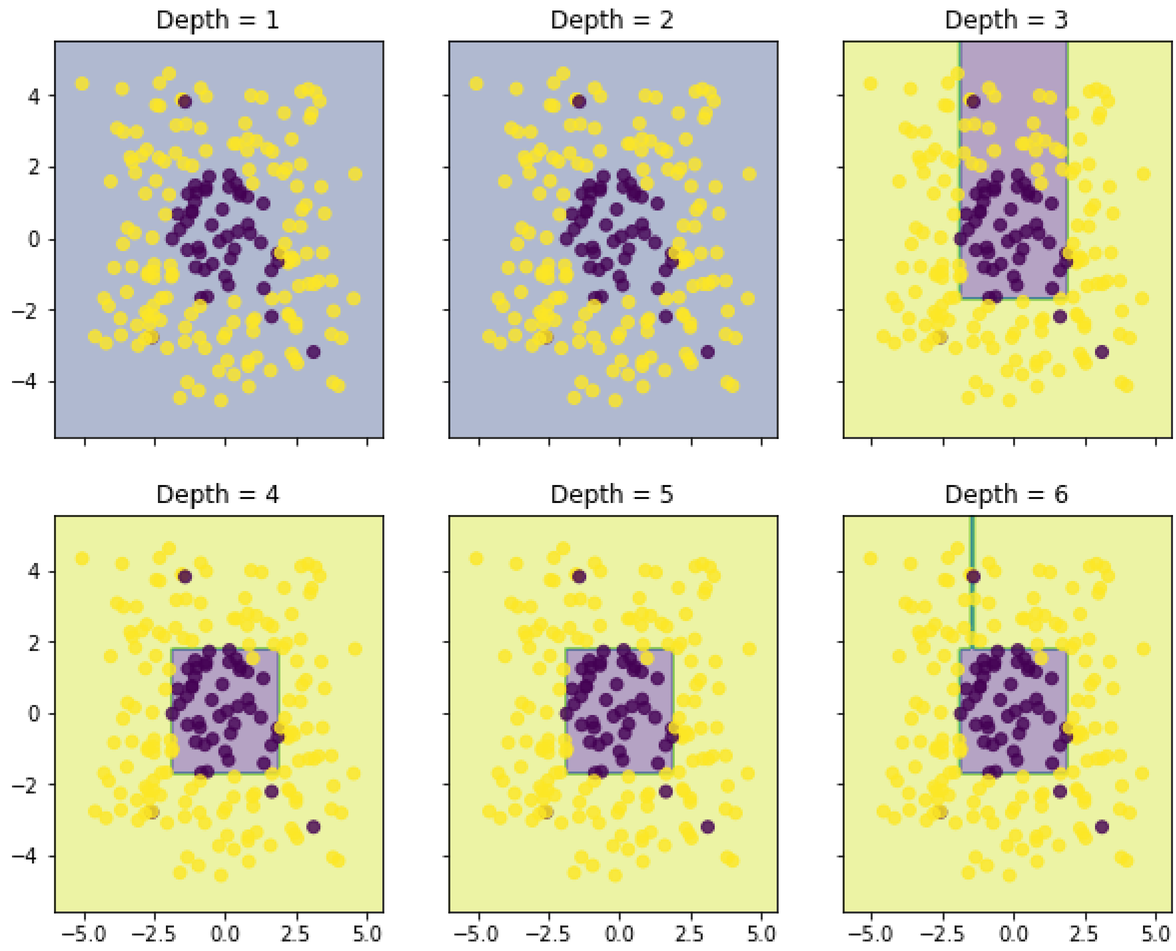
In [7]:

```
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
```

In [8]:

```
# Training classifiers with different depth
```

```
clf1 = Classification_Tree(max_depth=1)
clf1.fit(x_train, y_train_label)

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

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

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

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

clf6 = Classification_Tree(max_depth=20)
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, alpha
    axarr[idx[0], idx[1]].set_title(tt)

plt.show()
```



Compare decision tree with tree model in sklearn

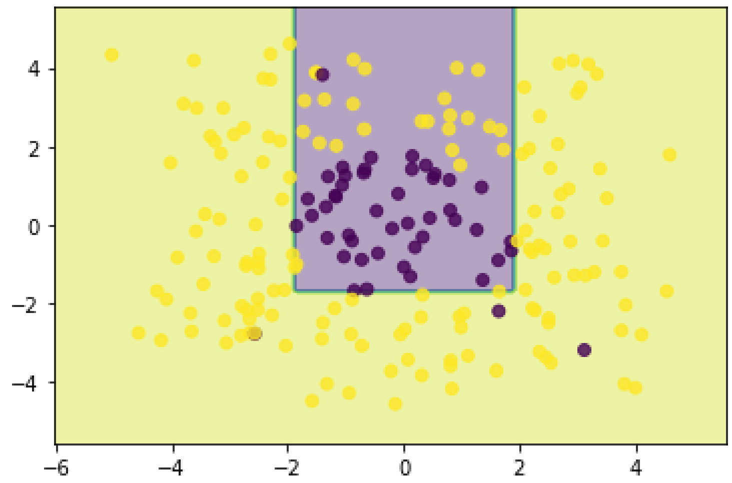
In [9]:

```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=3, min_samples_split=2)
clf.fit(x_train, y_train_label)
export_graphviz(clf, out_file='tree_classifier.dot')
```

In [10]:

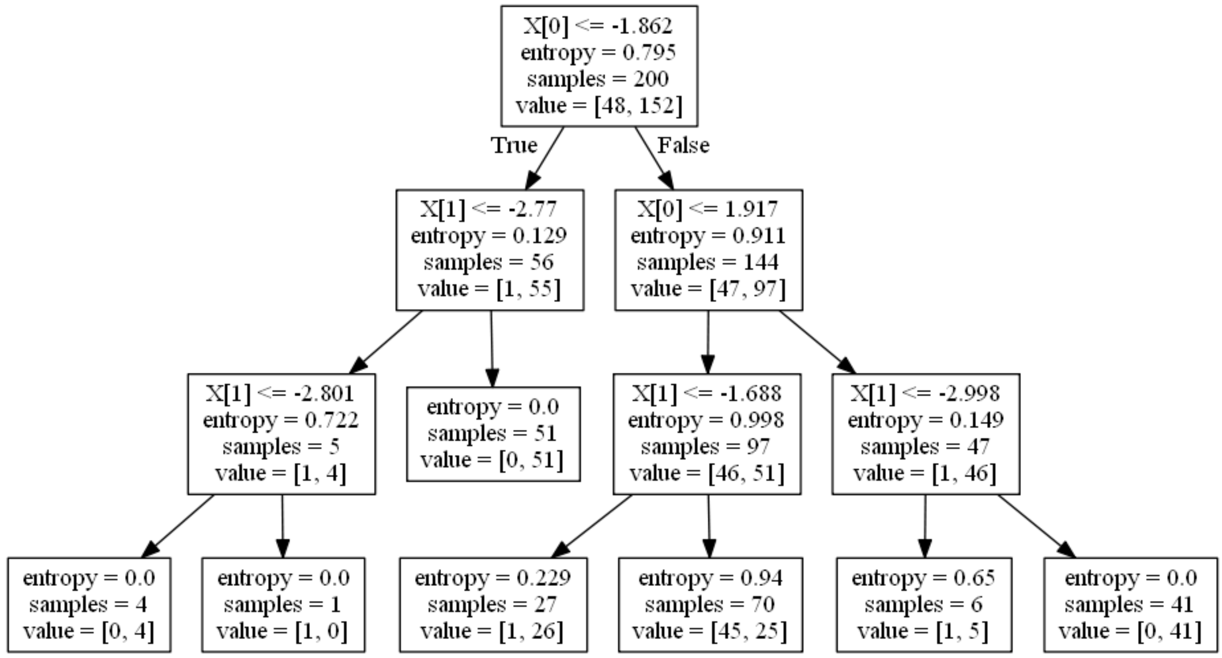
```
# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

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)
```

Out[10]: <matplotlib.collections.PathCollection at 0x1aa2b48d520>



In [11]:
```
# Visualize decision tree
!dot -Tpng tree_classifier.dot -o tree_classifier.png
Image(filename='tree_classifier.png')
```

Out[11]:



Q4

Decision Tree Regressor

In [12]:
```
# 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
    '''
    # Your code goes here
    y_med = np.median(y)
    mae = np.mean(np.abs(y-y_med))
    return mae
```

In [13]:
```
class Regression_Tree():
```

```
    '''
    :attribute loss_function_dict: dictionary containing the loss functions used for s
    :attribute estimator_dict: dictionary containing the estimation functions used in
    '''

    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
        '''
        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
```

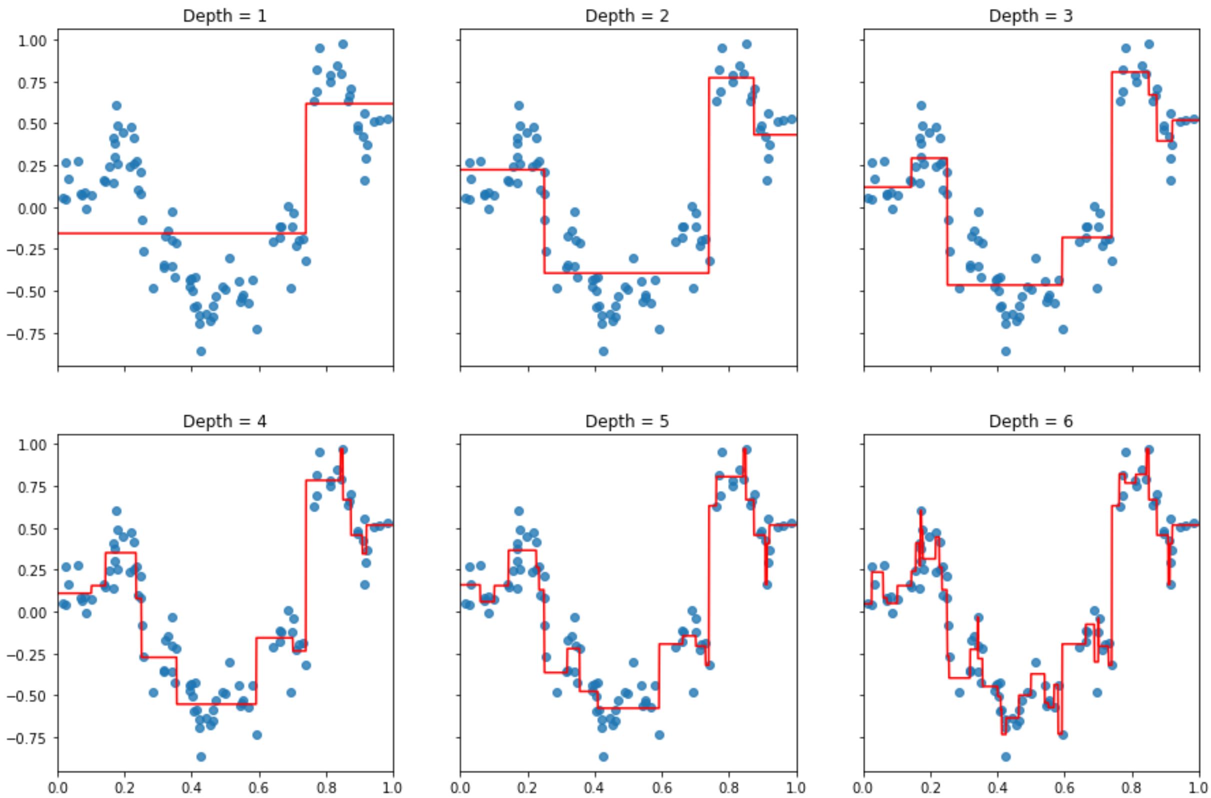
Fit regression tree to one-dimensional regression data

In [14]:

```
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].resh
x_krr_test, y_krr_test = data_krr_test[:,0].reshape(-1,1),data_krr_test[:,1].reshape(

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

clf2 = Regression_Tree(max_depth=2,  min_sample=3, loss_function='mae', estimator='me
clf2.fit(x_krr_train, y_krr_train)

clf3 = Regression_Tree(max_depth=3,  min_sample=3, loss_function='mae', estimator='me
clf3.fit(x_krr_train, y_krr_train)

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

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

clf6 = Regression_Tree(max_depth=10,  min_sample=3, loss_function='mae', estimator='m
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(-

    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()
```

```
C:\Users\52673\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:3372: RuntimeWarn
ing: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\52673\anaconda3\lib\site-packages\numpy\core\_methods.py:170: RuntimeWarning:
invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
```



Ensembling

Q5

In [15]:

```
#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
```

In [16]:

```
class gradient_boosting():
    '''
```

```
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func, learning_rate=0.1, min_samp
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounds of gradient bo
        :pseudo_residual_func: function used for computing pseudo-residual
        :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 = []


    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        '''
        # Your code goes here
        train_predict = 0
        residuals = self.pseudo_residual_func(train_target.reshape(-1), train_predict
        for i in range(self.n_estimator):
            hm = DecisionTreeRegressor(max_depth=self.max_depth, min_samples_leaf=sel
            self.estimators.append(hm)
            self.estimators[i].fit(train_data,residuals)
            train_predict += self.learning_rate * self.estimators[i].predict(train_da
            residuals = self.pseudo_residual_func(train_target.reshape(-1), train_pre
        return self

    def predict(self, test_data):
        '''
        Predict value
        '''
        # Your code goes here
        test_predict = 0
        for i in range(len(self.estimators)):
            test_predict += self.learning_rate * self.estimators[i].predict(test_data
        return test_predict
```

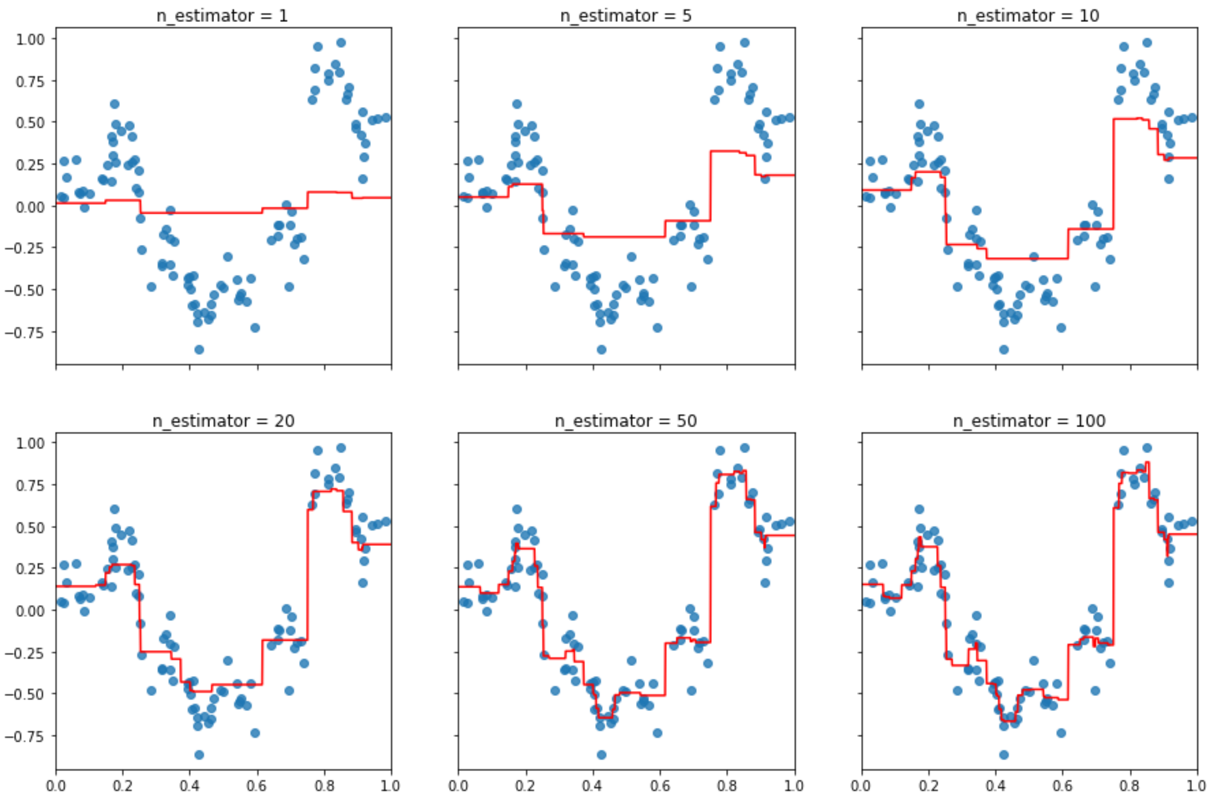
Q6

1-D GBM visualization - KRR data

In [17]:

```
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)
```



Classification of images with Gradient Boosting

Q7

The logistic loss:

$$\ell\left(y_i, f_{m-1}(x_i)\right) = \ln\left(1 + e^{-y_i f_{m-1}(x_i)}\right)$$

The pseudo residual:

$$\begin{aligned}
-\mathbf{g}_m &= -\left(\frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i)\right)\right)_{j=1}^{n} \\
&= -\left(\frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} \ln\left(1 + e^{-y_i f_{m-1}(x_i)}\right)\right)_{j=1}^{n} \\
&= -\left(\frac{\partial}{\partial f_{m-1}(x_j)} \ln\left(1 + e^{-y_j f_{m-1}(x_j)}\right)\right)_{j=1}^{n} \\
&= \left(\frac{y_j}{1 + e^{y_j f_{m-1}(x_j)}}\right)_{j=1}^{n}
\end{aligned}$$

The dimension of $\mathbf{g}_m$:

$$\dim(\mathbf{g}_m) = n$$

Q8

$$
\begin{aligned}
h_m &= \underset{h \in \mathcal{F}}{\operatorname{argmin}} \sum_{i=1}^{n} \left((-\mathbf{g}_m)_i - h(x_i) \right)^2 \\
&= \underset{h \in \mathcal{F}}{\operatorname{argmin}} \sum_{i=1}^{n} \left(\frac{y_i}{1 + e^{y_i f_{m-1}(x_i)}} - h(x_i) \right)^2
\end{aligned}
$$

Q9

In [18]:

```
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

from sklearn.ensemble import GradientBoostingClassifier
```

In [19]:

```
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
```

In [20]:

```
X_train, X_test, y_train, y_test = pre_process_mnist_01()
```

Train the model

In [21]:

```
estimators = [2, 5, 10, 100, 200]
train_acc_set = []
test_acc_set = []

for n_estimators in estimators:
    clf = GradientBoostingClassifier(loss='deviance', n_estimators=n_estimators, max_
```

```
    clf.fit(X_train, y_train)
    train_acc = clf.score(X_train, y_train)
    test_acc = clf.score(X_test, y_test)
    train_acc_set.append(train_acc)
    test_acc_set.append(test_acc)
```

Plot the accuracy

In [22]:

```
fig, ax = plt.subplots(figsize=[10,6])
ax.set_title('Accuracy vs Num of Estimators')
ax.set_xlabel('Estimators')
ax.set_ylabel('Accuracy')
ax.plot(estimators, train_acc_set, '--x', label='train accuracy')
ax.plot(estimators, test_acc_set, '--x', label='test accuracy')
ax.legend()
plt.show()
```