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, ClassifierMixi
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingClassifier, GradientBoost
import graphviz

from IPython.display import Image

%matplotlib inline
```

Load Data

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_trair
```

Decision Tree Class

Problem 1

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
    #Only considering binary classification
    entropy = 0 #Initialize entropy
    prob_one = label_array.sum() / len(label_array) #Calculate the pro
    prob_array = [1-prob_one,prob_one]              #Likewise for clas

    #Make sure we don't bug out trying to take log_2(0)
    if prob_one == 0 or prob_one == 1:
        return 0

    #Calculate entropy sum(-log_2(prob)*prob)
    for i in [0,1]:
        entropy -= np.log2(prob_array[i]) * prob_array[i]

    #Return entropy
    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
    '''
    #Only considering binary classification
    gini = 0                                        #Initialize gini i
    prob_one = label_array.sum() / len(label_array) #Calculate the pro
    prob_array = [1-prob_one,prob_one]              #Likewise for clas

    #Calculate gini index -> sum(prob_class(1-prob_class))
    for i in [0,1]:
        gini += prob_array[i]*(1-prob_array[i])
    return gini #Return gini index
```

Problem 2

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
        :param leaf_value_estimator: method for estimating leaf value
        :param depth: depth indicator, default value is 0, representin
        :param min_sample: an internal node can be splitted only if it
        :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

        #Add these variables to the constructor
        self.right = None         #Left child node
        self.left = None         #Right child node
        self.split_id = None      #Best column to split on
        self.split_value = None   #Best value to split on within best
        self.value = None         #Value to return if

    def fit(self, x, y):
        '''
        This should fit the tree classifier by setting the values self
        self.split_id (the index of the feature we want ot split on, i
        self.split_value (the corresponding value of that feature wher
        and self.value, which is the prediction value if the tree is a
        splitting the node, we should also init self.left and self.rig
        objects corresponding to the left and right subtrees. These su
        the data that fall to the left and right,respectively, of self
        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

        #Check break condition, if we've exceeded max depth or are leq
        if self.depth >= self.max_depth or len(y) <= self.min_sample:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)

#             if y.sum() / len(y) <= .5:
#                 self.value = 0
#             else:
#                 self.value = 1
```

```
        else:
            #Calculate best splitting point
            self.find_best_feature_split(x,y)
            #Split data in two depending on criteria
            all_data = np.append(x,y,axis=1) #Create one big matrix (e
            #Filter data by split column / split point
            left_data = all_data[all_data[:,self.split_id]<=self.split
            #Again but look for greater for right tree
            right_data = all_data[all_data[:,self.split_id]>self.split
            left_x_node = left_data[:,0:-1]
            left_y_node = left_data[:,-1].reshape(-1,1)
            right_x_node = right_data[:,0:-1]
            right_y_node = right_data[:,-1].reshape(-1,1)

            #Create left and right nodes
            self.left = Decision_Tree(self.split_loss_function, #Pass
                                      self.leaf_value_estimator, #Pass
                                      depth=self.depth+1,        #Pass
                                      min_sample = self.min_sample, #Pa
                                      max_depth = self.max_depth
                                      )
            self.right = Decision_Tree(self.split_loss_function, #Pass
                                      self.leaf_value_estimator, #Pass
                                      depth=self.depth+1,        #Pass
                                      min_sample = self.min_sample, #Pa
                                      max_depth = self.max_depth
                                      )
            #Fit the left/right nodes
            self.left.fit(left_x_node,left_y_node)
            self.right.fit(right_x_node,right_y_node)

        return self

    def find_best_split(self, x_node, y_node, feature_id):
        '''
        For feature number feature_id, returns the optimal splitting p
        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
        #x_copy = x_node.copy()
        y_copy = y_node.copy()
        feature_vals = x_node[:,feature_id].copy() #Grab the feature v
        sorting = feature_vals.argsort()    #Prepare index for arg sor
        y_copy = y_copy[sorting]            #Sort the y_node by x inde
        feature_vals.sort()                 #Sort the feature grabbed

        #Initialize entropy variable
        best_loss = 100
```

```
        split_value = -1
        #Iterate over the feature vals
        for i in range(1,len(feature_vals)):
            #Seperate sorted (single) feature vals into two halves
            top_half = y_copy[:i]
            bottom_half = y_copy[i:]

            #Calculate weighted entropy for each half
            top_ratio = (len(top_half)/len(y_copy))
            bottom_ratio = (len(bottom_half)/len(y_copy))
            top_half_entropy =  top_ratio * self.split_loss_function(t
            bottom_half_entropy = bottom_ratio * self.split_loss_funct

            #Calculate Loss = Total Weighted Entropy
            loss = top_half_entropy + bottom_half_entropy

            #Check if we've reached a smaller loss
            if loss <= best_loss:
                best_loss = loss #Update smaller loss

                if i == 1:
                    split_value = (feature_vals[i]+feature_vals[i+1])/
                #Update the best split value via midpoint value
                #Take midpoint of point before after if best split poi
                else:
                    split_value = (feature_vals[i]+feature_vals[i-1])/



        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)
        '''
        best_feature_loss = 100
        #Iterate over all of the columns
        for i in range(x_node.shape[1]):
            #Use self.find_best_split to
            split_value, best_loss = self.find_best_split(x_node,y_noc

            #Check if we've found a column to split better on
            if best_loss <= best_feature_loss:
                best_feature_loss = best_loss      #Update Loss Accord
                self.split_id = i                  #Update the column
                self.split_value = split_value     #Update the column

    def predict_instance(self, instance):
```

```
    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
        '''
        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)
```

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
```

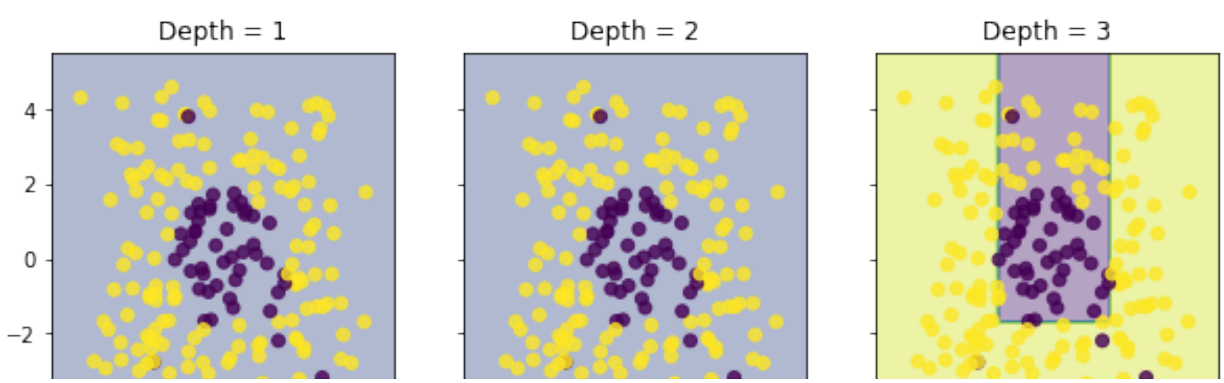
In [7]:
```
class Classification_Tree(BaseEstimator, ClassifierMixin):

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

    def __init__(self, loss_function='entropy', min_sample=5, max_dept
        '''
        :param loss_function(str): loss function for splitting interna
        '''

        self.tree = Decision_Tree(self.loss_function_dict[loss_functio
                                  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
```

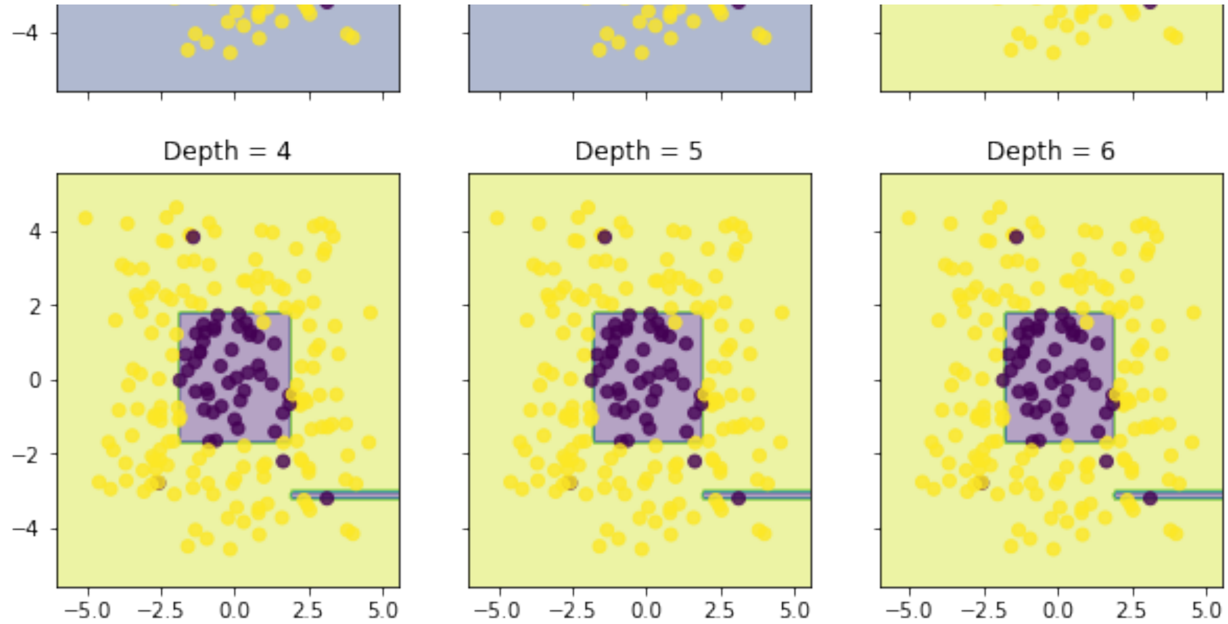
Problem 3

Decision Tree Boundary

In [8]:

```
# 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,

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(), y
    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_tr
    axarr[idx[0], idx[1]].set_title(tt)

plt.show()
```



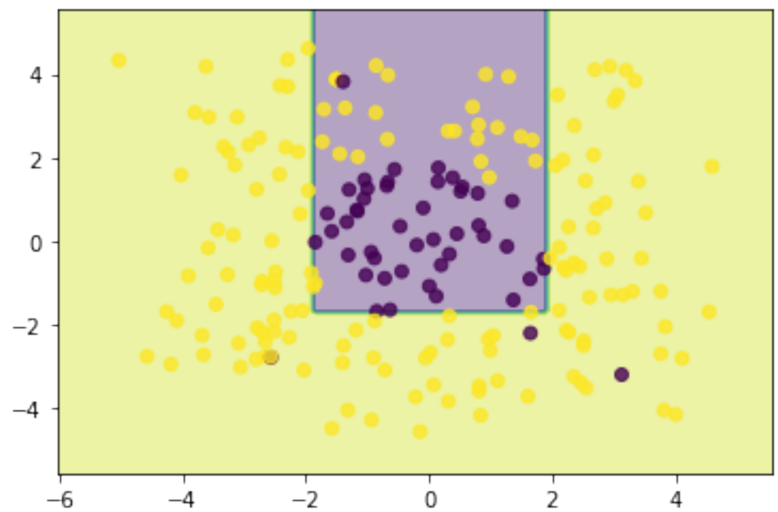


Compare decision tree with tree model in sklearn

In [9]:
```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=3, min_sam
clf.fit(x_train, y_train_label)
export_graphviz(clf, out_file='tree_classifier.dot')
```

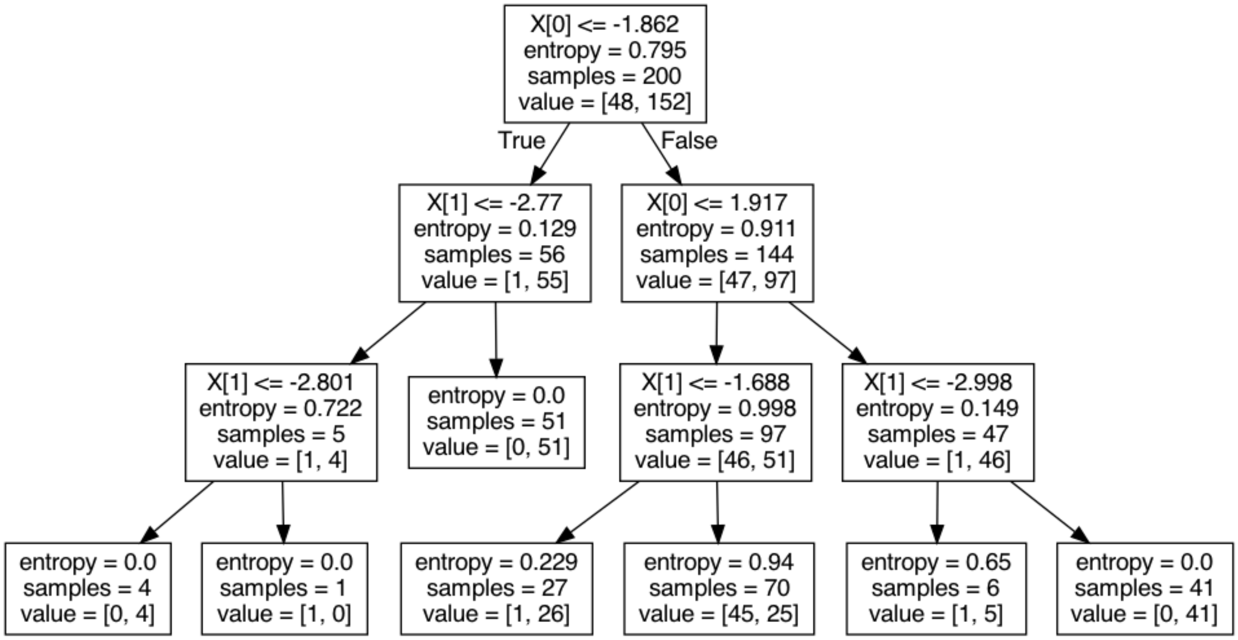
In [10]:
```
# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = np.array([clf.predict(x[np.newaxis,:]) for x in np.c_[xx.ravel(),
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 0x7ff5505a2670>
```



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

Out[11]:



Problem 4

Decision Tree Regressor

In [12]:

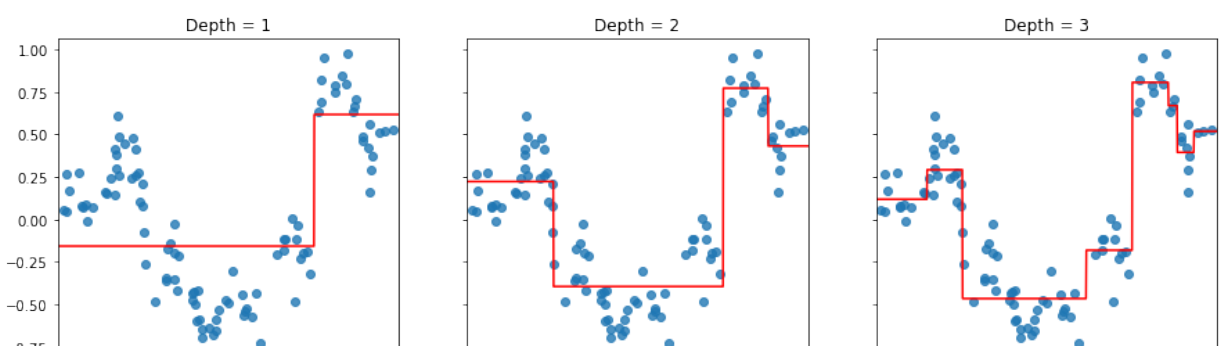
```python
# Regression Tree Specific Code
def mean_absolute_deviation_around_median(y):
    '''
    Calulate the mean absolute deviation around the median of a given

    :param y: a numpy array of targets shape = (n, 1)
    :return mae
    '''
    # Initialize mae / median
    mae = 0
    median = np.median(y)
    #Iterate over y's and calculate absolute deviation from median
    for y_hat in y:
        mae += abs(y_hat-median)
    #Take average
    mae = mae / len(y)
    #Return mae
    return mae
```

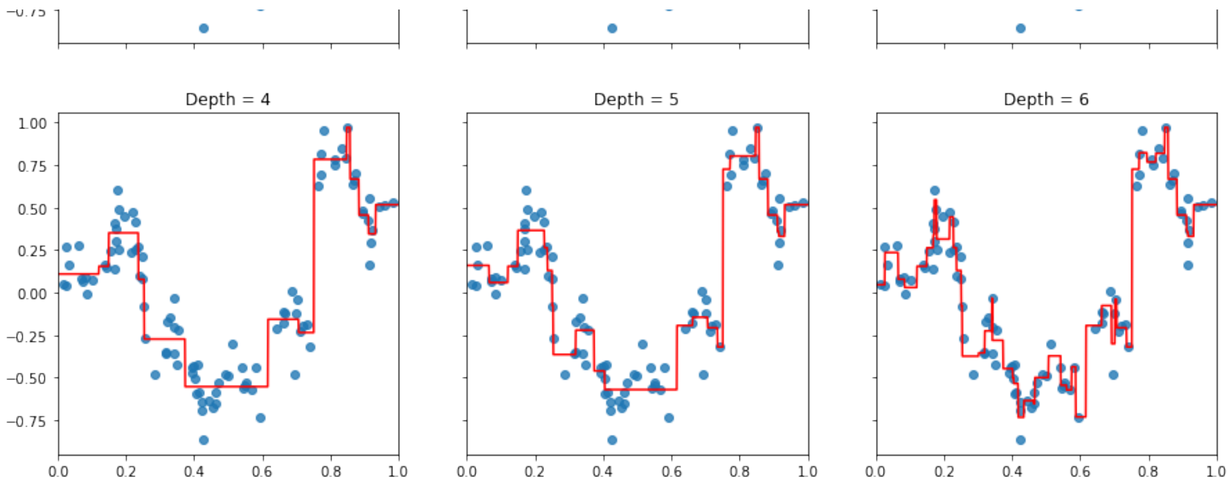
In [13]:
```
class Regression_Tree():
    '''
    :attribute loss_function_dict: dictionary containing the loss func
    :attribute estimator_dict: dictionary containing the estimation fu
    '''

    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_samp
        '''
        Initialize Regression_Tree
        :param loss_function(str): loss function used for splitting ir
        :param estimator(str): value estimator of internal node
        '''

        self.tree = Decision_Tree(self.loss_function_dict[loss_functic
                                  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_
x_krr_test, y_krr_test = data_krr_test[:,0].reshape(-1,1),data_krr_tes

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

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

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

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

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

clf6 = Regression_Tree(max_depth=10,  min_sample=3, loss_function='mae
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=(1

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_ran

    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()
```



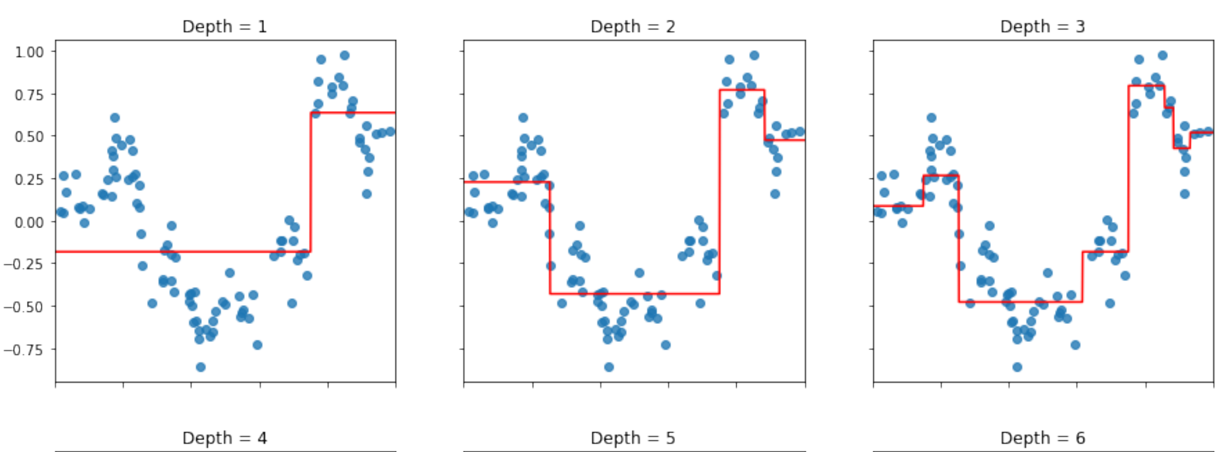


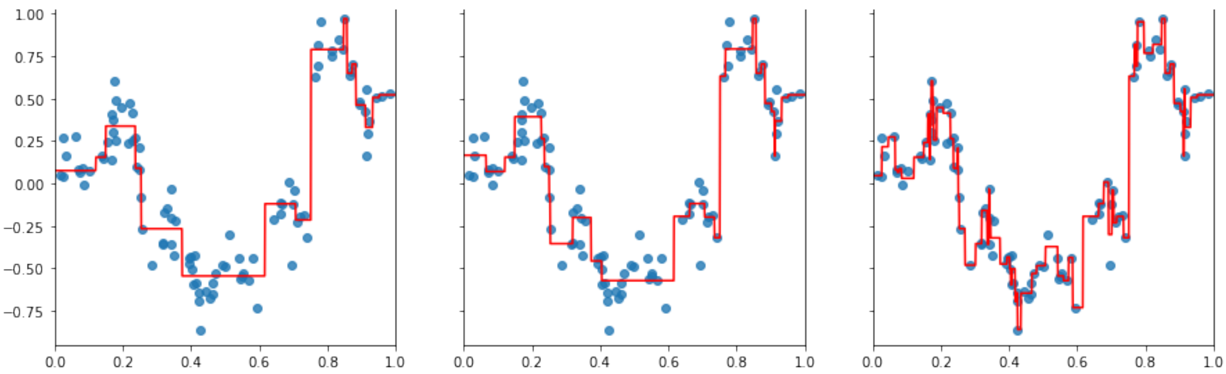
Compare with scikit-learn for debugging

In [15]:

```
```

```
# Training regression trees with different depth
clf1 = DecisionTreeRegressor(criterion='absolute_error', max_depth=1,
Regression_Tree(max_depth=1,  min_sample=3, loss_function='mae', estim
clf1.fit(x_krr_train, y_krr_train)

clf2 = DecisionTreeRegressor(criterion='absolute_error', max_depth=2,
clf2.fit(x_krr_train, y_krr_train)

clf3 = DecisionTreeRegressor(criterion='absolute_error', max_depth=3,
clf3.fit(x_krr_train, y_krr_train)

clf4 = DecisionTreeRegressor(criterion='absolute_error', max_depth=4,
clf4.fit(x_krr_train, y_krr_train)

clf5 = DecisionTreeRegressor(criterion='absolute_error', max_depth=5,
clf5.fit(x_krr_train, y_krr_train)

clf6 = DecisionTreeRegressor(criterion='absolute_error', max_depth=10,
clf6.fit(x_krr_train, y_krr_train)

#Compare Plots
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=(1

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 = clf.predict(np.array([x for x in x_range]).resha

    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()
```





Gradient Boosting Method

Problem 5

In [16]:
```
#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 [27]:
```
class gradient_boosting():
    '''
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func, learning_rat
                 min_sample=5, max_depth=5):
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounc
        :pseudo_residual_func: function used for computing pseudo-resi
        :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):
```

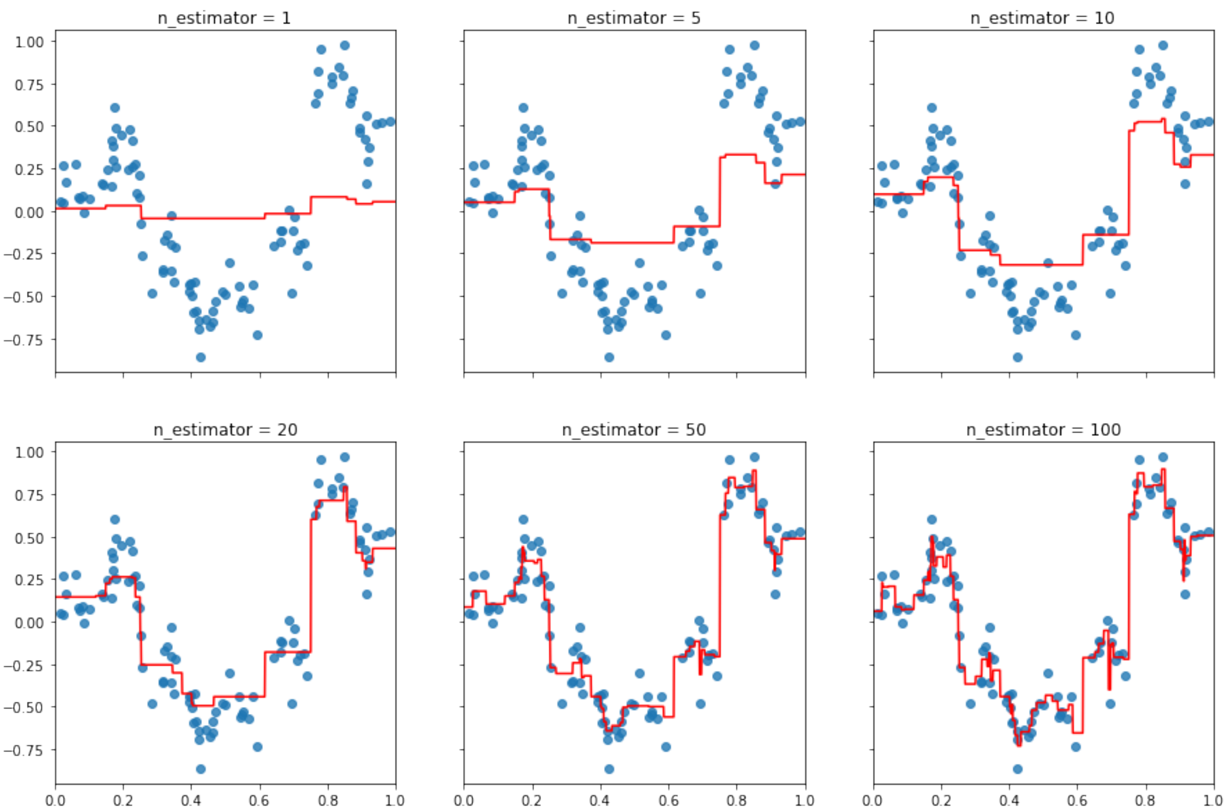
```
    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        :train_data array of inputs of size (n_samples, m_features)
        :train_target array of outputs of size (n_samples,)
        '''
        #Initialize array of 0's of size = (len(train_target))
        base_grad = np.zeros(len(train_target))
        self.estimators.append(base_grad) #Append base case

        #Set up our base case - Sk Learns Regression Tree
        base_case = DecisionTreeRegressor(criterion='squared_error',
                                          min_samples_split=self.min_s
                                          max_depth=self.max_depth)
        #Fit regression model
        base_case.fit(train_data, train_target.flatten())
        #Append Estimators
        self.estimators.append(base_case)

        #Iterate for however many rounds of gradient boosting we're us
        for i in range(1,self.n_estimator):
            #Compute Predictions
            predictions = self.predict(train_data)

            #Compute Residuals
            residuals = train_target.flatten() - predictions

            #Fit regression model to -g
            base_case = DecisionTreeRegressor(criterion='squared_error
                                          min_samples_split=self.min_s
                                          max_depth=self.max_depth)
            #Fit new function
            base_case.fit(train_data, residuals)
            #Append estimators
            self.estimators.append(base_case)
        return self


    def predict(self, test_data):
        '''
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''
        #Initialize prediction
        test_predict = np.zeros(len(test_data))

        #Iterate over the estimators we have saved in our .fit method
        for i in range(1,len(self.estimators)):

            #Add estimator_i prediction to test_predict, but scale by
            test_predict += self.estimators[i].predict(test_data) * se
```

```
        return test_predict
```

1-D GBM visualization - KRR data

Question 6

In [28]:
```
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=(1

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, 1

    gbm_1d = gradient_boosting(n_estimator=i, pseudo_residual_func=pse
                               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)
```



Sklearn implementation for Classification of images

Question 9

Gradient Boosting Classifier

In [29]:
```
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
```

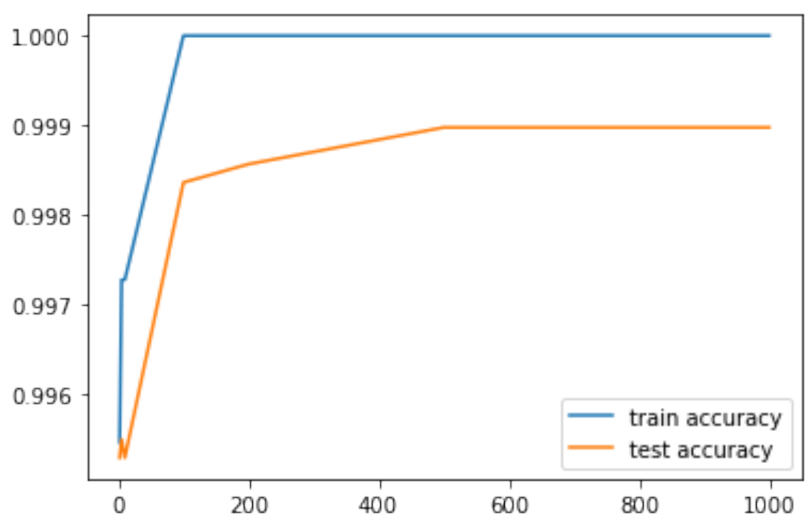
In [30]:
```
def pre_process_mnist_01():
    """
    Load the mnist datasets, selects the classes 0 and 1
    and normalize the data.
    Args: none
    Outputs:
        X_train: np.array of size (n_training_samples, n_features)
        X_test: np.array of size (n_test_samples, n_features)
        y_train: 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_
                                                        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 [31]:
```
X_train, X_test, y_train, y_test = pre_process_mnist_01()
```

In [32]:
```
#Initalize helper variables
loss_dict = {'train':[],'test':[]}
estimators = [2,5,10,100,200,500,1000]

#Iterate over the estimators
for n in estimators:
    #Helper Print Statement to let me know it isn't dead
    print(f'Now fiitting GBC - {n} Estimators used')
    #Initliaze GBC Estimator
    gbc = GradientBoostingClassifier(n_estimators=n, loss='deviance',
    #Fit the GBC estimator
    gbc.fit(X_train, y_train)

    #Append results
    loss_dict['train'].append(gbc.score(X_train, y_train)) #Append Tra
    loss_dict['test'].append(gbc.score(X_test,y_test))     #Append Tes


#Plot our results
plt.plot(estimators, loss_dict['train'], label = 'train accuracy') #Pl
plt.plot(estimators, loss_dict['test'], label = 'test accuracy')   #Pl
plt.legend()
plt.plot()
```

```
fitting gbc with 2 estimators
fitting gbc with 5 estimators
fitting gbc with 10 estimators
fitting gbc with 100 estimators
fitting gbc with 200 estimators
fitting gbc with 500 estimators
fitting gbc with 1000 estimators
```

Out[32]: []



Problem 11

Random Forest Classifier

In [36]:
```
#Initalize helper variables
loss_dict = {'train':[],'test':[]}
estimators = [2,5,10,100,200,500,1000]

#Iterate over our estimators
for n in estimators:
    #Initalize and fit a Random Forest Classifier from sklearn
    gbrf = RandomForestClassifier(n_estimators=n, criterion = 'entropy
    gbrf.fit(X_train, y_train)

    #Append our train / test loss
    loss_dict['train'].append(gbrf.score(X_train, y_train))
    loss_dict['test'].append(gbrf.score(X_test,y_test))

#Plot our results
plt.plot(estimators, loss_dict['train'], label = 'train accuracy') #Pl
plt.plot(estimators, loss_dict['test'], label = 'test accuracy')   #Pl
plt.legend()
plt.plot()
```

Out[36]: []