Homework 6: Decision Trees and Boosting

Due: Friday, April 22nd, 2022 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. **The optional problems should not take you too much time and help you navigate the material, consider taking a shot at them.**

---

1 Decision Tree Implementation

In this problem we'll implement decision trees for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the loss function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here. We'll work with the classification and regression data sets from previous assignments.

1. Complete the `compute_entropy` and `compute_gini` functions.

2. Complete the class `Decision_Tree`, given in the skeleton code. The intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable self.depth, with the root node having depth 0. The main job of the fit function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.

3. Run the code provided that builds trees for the two-dimensional classification data. Include the results. For debugging, you may want to compare results with sklearn's decision tree (code provided in the skeleton code). For visualization, you'll need to install `graphviz`.

4. Complete the function `mean_absolute_deviation_around_median` (MAE). Use the code provided to fit the `Regression_Tree` to the krr dataset using both the MAE loss and median predictions. Include the plots for the 6 fits.

2 Ensembling

Recall the general gradient boosting algorithm , for a given loss function ℓ and a hypothesis space $\mathcal{F}$ of regression functions (i.e. functions mapping from the input space to $\mathbb{R}$):

0: Initialize $f_0(x) = 0$.

1: For $m = 1$ to M:

(a) Compute:

$$\mathbf{g}_m = \left(\frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i)\right) \right)_{j=1}^{n}$$

(b) Fit regression model to $-\mathbf{g}_m$:

$$h_m = \underset{h \in \mathcal{F}}{\arg\min} \sum_{i=1}^{n} \left((-\mathbf{g}_m)_i - h(x_i) \right)^2.$$

(c) Choose fixed step size $\nu_m = \nu \in (0, 1]$, or take

$$\nu_m = \underset{\nu > 0}{\arg\min} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + \nu h_m(x_i)\right).$$

(d) Take the step:

$$f_m(x) = f_{m-1}(x) + \nu_m h_m(x)$$

3: Return f_M.

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. One of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function.

Gradient Boosting Regression Implementation

First we'll keep things simple and consider the standard regression setting with square loss. In this case the we have $\mathcal{Y} = \mathbb{R}$, our loss function is given by $\ell(\hat{y}, y) = 1/2\,(\hat{y} - y)^2$, and at the m'th round of gradient boosting, we have

$$h_m = \underset{h \in \mathcal{F}}{\arg\min} \sum_{i=1}^{n} \left[(y_i - f_{m-1}(x_i)) - h(x_i) \right]^2.$$

5. Complete the `gradient_boosting` class. As the base regression algorithm to compute the argmin, you should use sklearn's regression tree. You should use the square loss for the tree splitting rule (`criterion` keyword argument) and use the default sklearn leaf prediction rule from the `predict` method [1]. We will also use a constant step size ν.

6. Run the code provided to build gradient boosting models on the regression data sets `krr-train.txt`, and include the plots generated. For debugging you can use the sklearn implementation of `GradientBoostingRegressor`[2].

[1] Examples of usage are given in the skeleton code to debug previous problems, and you can check the docs https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

[2] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

Classification of images with Gradient Boosting

In this problem we will consider the classification of MNIST, the dataset of handwritten digits images, with ensembles of trees. For simplicity, we only retain the '0' and '1' examples and perform binary classification.

First we'll derive a special case of the general gradient boosting framework: BinomialBoost. Let's consider the classification framework, where $\mathcal{Y} = \{-1, 1\}$. In lecture, we noted that AdaBoost corresponds to forward stagewise additive modeling with the exponential loss, and that the exponential loss is not very robust to outliers (i.e. outliers can have a large effect on the final prediction function). Instead, let's consider the logistic loss

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

where $m = yf(x)$ is the margin.

7. Give the expression of the negative gradient step direction, or pseudo residual, $-\mathbf{g}_m$ for the logistic loss as a function of the prediction function f_{m-1} at the previous iteration and the dataset points $\{(x_i, y_i)\}_{i=1}^n$. What is the dimension of g_m?

   The dimension of g_m is $Dim(g_m) = n$, (aka: $g_m \in \mathbb{R}^n$) since there are n preditions corresponding to data points. Setting up our gradient as:

   $$-g_{m_i} = \ell(y_i, f_{m-1}(x_i))$$

   Where m is the variable which we differentiate with respect to, and i is the i^{th} entry of $-g_m$. The i^{th} entry in the negative gradient step direction is given by the following:

   $$-g_{m_i} = \frac{\partial \ell(m)}{\partial m} = \frac{y_i \times -e^{-y_i f_{m-1}(x_i)}}{1 + e^{-y_i f_{m-1}(x_i)}} = \frac{1}{1 + e^{y_i f_{m-1}(x_i)}}$$

   We showed what the negative gradient is for the i^{th} entry, here's the full thing:

   $$-g_m = \left(\frac{1}{1 + e^{y_i f_{m-1}(x_i)}}, \ldots, \frac{1}{1 + e^{y_n f_{m-1}(x_n)}}\right)$$

8. Write an expression for h_m as an argmin over functions h in $\mathcal{F}$.

   We need to minimize the following expression:

   $$h_m = \arg\min_{h \in H} \sum_{i=1}^{n} \left[-g_i - h_i(x_i)\right]^2$$

   Substitutign our identity we found in problem 7 for $-g_i$, we can write the expression for the argmin of h_m as follows:

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

9. Load the MNIST dataset using the helper preprocessing function in the skeleton code.Using the scikit learn implementation of `GradientBoostingClassifier`, with the logistic loss (`loss='deviance'`) and trees of maximum depth 3, fit the data with 2, 5, 10, 100 and 200 iterations (estimators). Plot the train and test accurary as a function of the number of estimators.

Classification of images with Random Forests (Optional)

10. Another type of ensembling method we discussed in class are random forests. Explain in your own words the construction principle of random forests.

    Random forests is an ensemble method in which many trees are built and then used to create a final decision tree.

    To build random forests we take advantage of ensemble methods and bootstrapping.

    Ensemble methods are ML methods which combine many weak models into one powerful model. In our case, this means combining many trees (low bias, high variance) into a single tree that hopefully achieves lower variance, and higher accuracy on test set performance.

    Bootstrapping is a method in which we re-sample our data with replacement to simulate taking additional independent samples of a true underlying distribution from which we have attained our training data. Independence here is a strong word, as the bootstrapped samples are independent of the training data, but not the underlying distribution, $P_{\mathcal{X}\times\mathcal{Y}}$.

    The benefit of this, is that when we achieve some sample statistic θ, it is an unbiased estimator of the θ of the true underlying distribution. However, it also has some variance, σ^2, so to try to decrease the variance of our estimator, θ, we use bootstrapping which decreases the variance to $\frac{\sigma^2}{n}$, while not changing the expectation of our sample statistic.

    Our procedure to build a random forest is as follows:

    - Simulate data by using resampling methods like Bootstrapping.
    - Train many decision trees on separate portions of the simulated data (this can be done in parallel!). We restrict our choice of splitting variable to a randomly chosen subset of features of size m. This hopefully avoids the situation in which smaller trees are dominated by highly correlated features that explain most of the variance. A good size for m is $m = \sqrt{p}$, where p is the number of features.
    - Combine the output of our random forest model (the many trees we just created), by averaging the decision criteria or taking majority vote (the method you will use will depend on what prediction task you are trying to accomplish).

11. Using the scikit learn implementation of `RandomForestClassifier`[3], with the entropy loss (`criterion='entropy'`) and trees of maximum depth 3, fit the preprocessed binary MNIST dataset with 2, 5, 10, 50, 100 and 200 estimators.

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier`

12. What general remark can you make on overfitting for Random Forests and Gradient Boosted Trees? Which method achieves the best train accuracy overall? Is this result expected? Can you think of a practical disadvantage of the best performing method? How do the algorithms compare in term of test accuracy?

Model Results:

Both models are prone to overfitting, with Gradient Boosted Trees overfittitng slightly more than the Random Forest model.

In the experiments we ran as part of the homework, both methods achieved very high test accuracy at 99.99%. However, Gradient Boosting Methods achieved 100% training accuracy, meaning it was more prone to overfitting, than Random Forest, which approach 99.98% (though that margin is razor thin).

This result is expected, as Gradient Boost continues to find functions that explain the variance of the residuals, and Random Forest can continually improve its performance as more estimators are added. That being said, for the dataset we used in the homework, the Random Forest model approached its highest test accuracy much quicker than Gradient Boosted Trees.

Disadvantages:

The fact that the Random Forest task can be computed in parallel is incredibly valuable. Gradient Boosted Trees have additional hyper-parameters like learning rate, hypothesis space, that must be tuned to work optimally, which can be a disadvantage. They also cannot be computed in parallel, and require a lot of CPU power.

For Random forest, you must choose how many features to include in each tree, which is another hyper-parameter you must tune (though $m = \sqrt{p}$ is used in practice mostly.

Both methods can suffer in interpretability, and the significance of each variable can be hard to gauge. Both models are also prone to overfitting.

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, expor
from sklearn.ensemble import GradientBoostingClassifier, GradientBoostingRegr
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_train))).res
```

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 probabilit
    prob_array = [1-prob_one,prob_one]              #Likewise for class = 0

    #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 index
    prob_one = label_array.sum() / len(label_array) #Calculate the probabilit
    prob_array = [1-prob_one,prob_one]              #Likewise for class = 0

    #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 loss
        :param leaf_value_estimator: method for estimating leaf value from ar
        :param depth: depth indicator, default value is 0, representing root 
        :param min_sample: an internal node can be splitted only if it contai
        :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 column
        self.value = None         #Value to return if

    def fit(self, x, y):
        '''
        This should fit the tree classifier by setting the values self.is_lea
        self.split_id (the index of the feature we want ot split on, if we're
        self.split_value (the corresponding value of that feature where the s
        and self.value, which is the prediction value if the tree is a leaf n
        splitting the node, we should also init self.left and self.right to b
        objects corresponding to the left and right subtrees. These subtrees 
        the data that fall to the left and right,respectively, of self.split_
        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 min_sa
        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 (easier t
            #Filter data by split column / split point
            left_data = all_data[all_data[:,self.split_id]<=self.split_value]
            #Again but look for greater for right tree
            right_data = all_data[all_data[:,self.split_id]>self.split_value]
            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 split f
                                      self.leaf_value_estimator, #Pass leaf_va
                                      depth=self.depth+1,        #Pass self.de
                                      min_sample = self.min_sample, #Pass min
                                      max_depth = self.max_depth
                                      )
            self.right = Decision_Tree(self.split_loss_function, #Pass split
                                      self.leaf_value_estimator, #Pass leaf_va
                                      depth=self.depth+1,        #Pass self.de
                                      min_sample = self.min_sample, #Pass min
                                      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 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
        #x_copy = x_node.copy()
        y_copy = y_node.copy()
        feature_vals = x_node[:,feature_id].copy() #Grab the feature vals
        sorting = feature_vals.argsort()    #Prepare index for arg sorting fe
        y_copy = y_copy[sorting]            #Sort the y_node by x index
        feature_vals.sort()                 #Sort the feature grabbed from x 

        #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(top_half
            bottom_half_entropy = bottom_ratio * self.split_loss_function(bot

            #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])/2 #Take
                #Update the best split value via midpoint value
                #Take midpoint of point before after if best split point is f
                else:
                    split_value = (feature_vals[i]+feature_vals[i-1])/2 #Take



        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_node,i)

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

    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 contain
        '''
        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]:

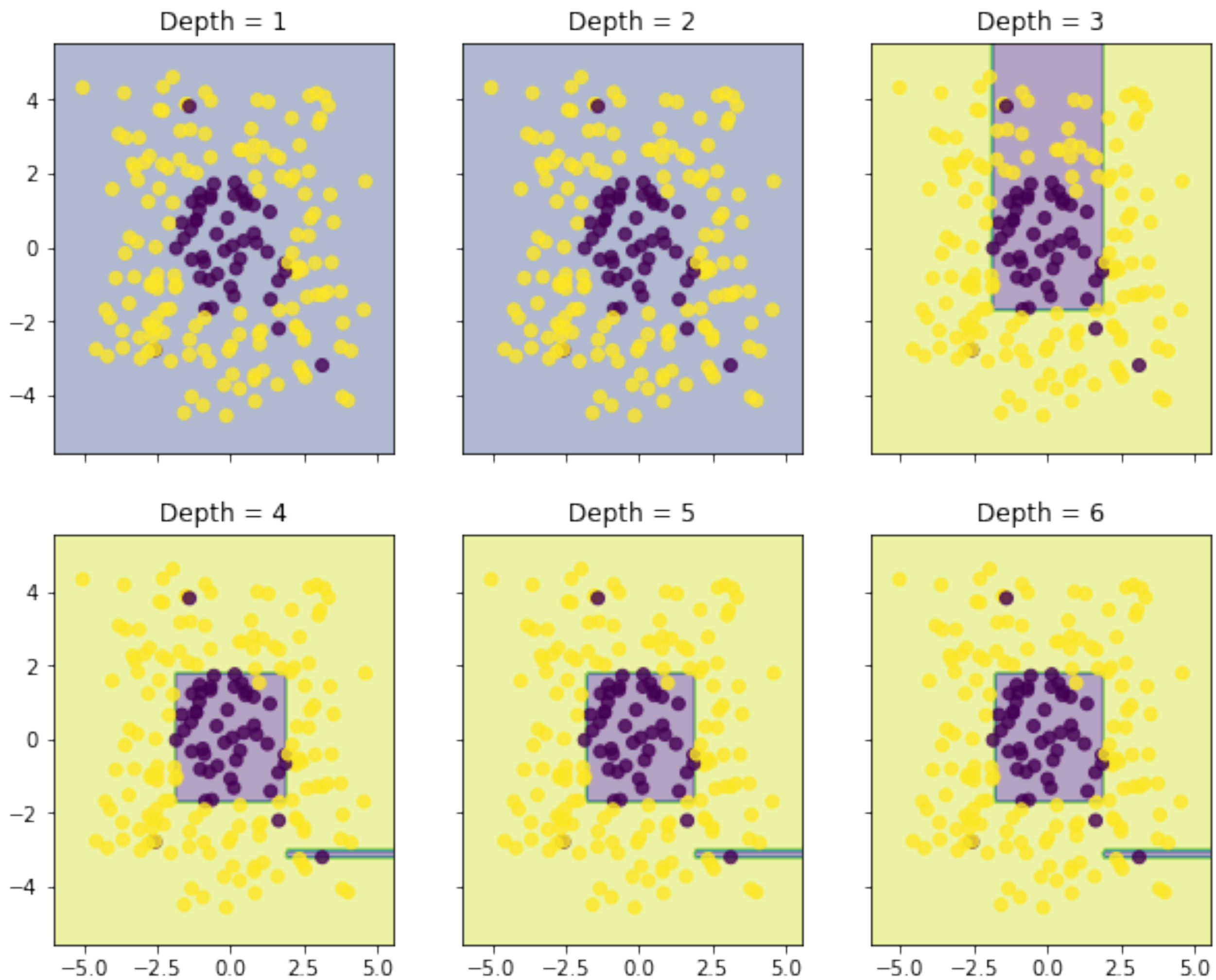
```
class Classification_Tree(BaseEstimator, ClassifierMixin):

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

    def __init__(self, loss_function='entropy', min_sample=5, max_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
```

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

plt.show()
```



Compare decision tree with tree model in sklearn

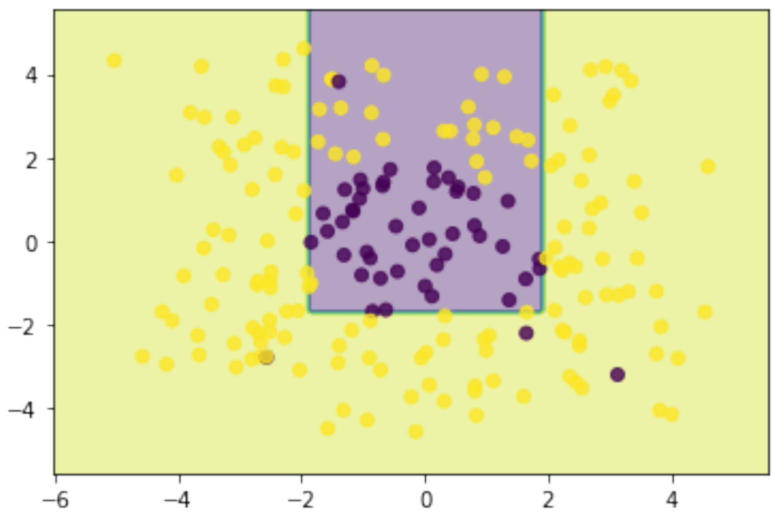
In [9]:

```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=3, min_samples_sp
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.rave
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)
```

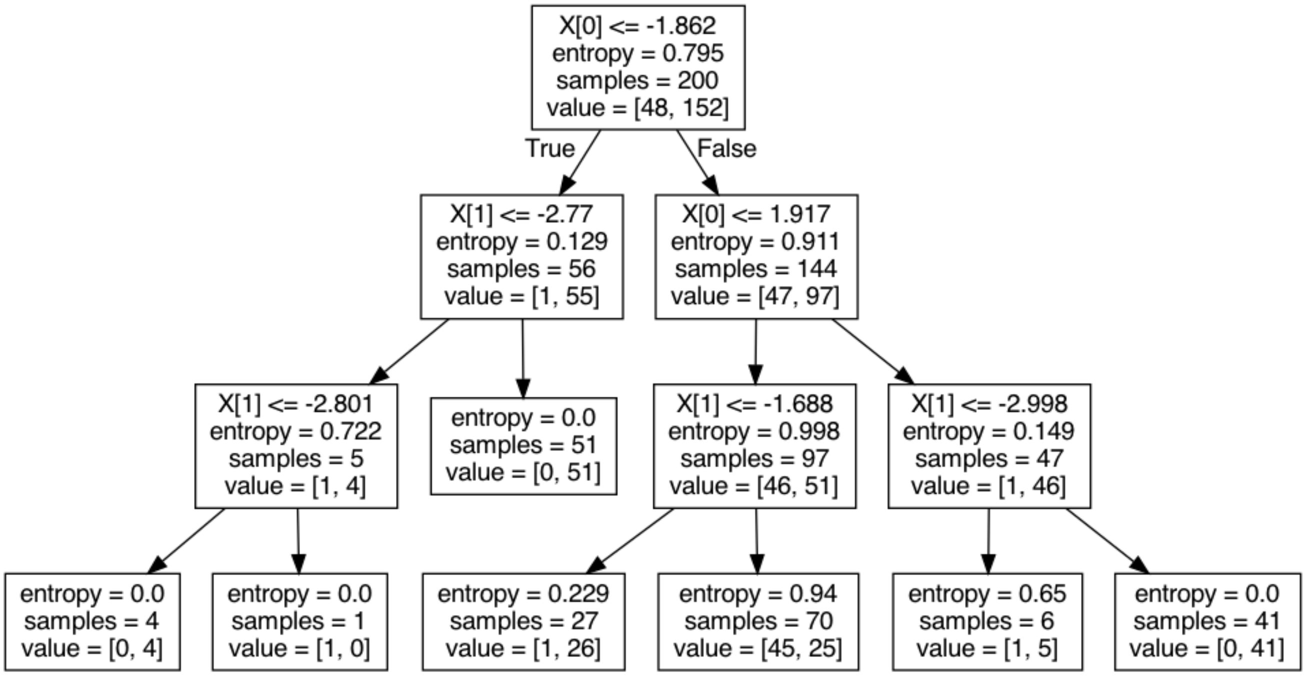
Out[10]: <matplotlib.collections.PathCollection at 0x7ff5505a2670>



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]:

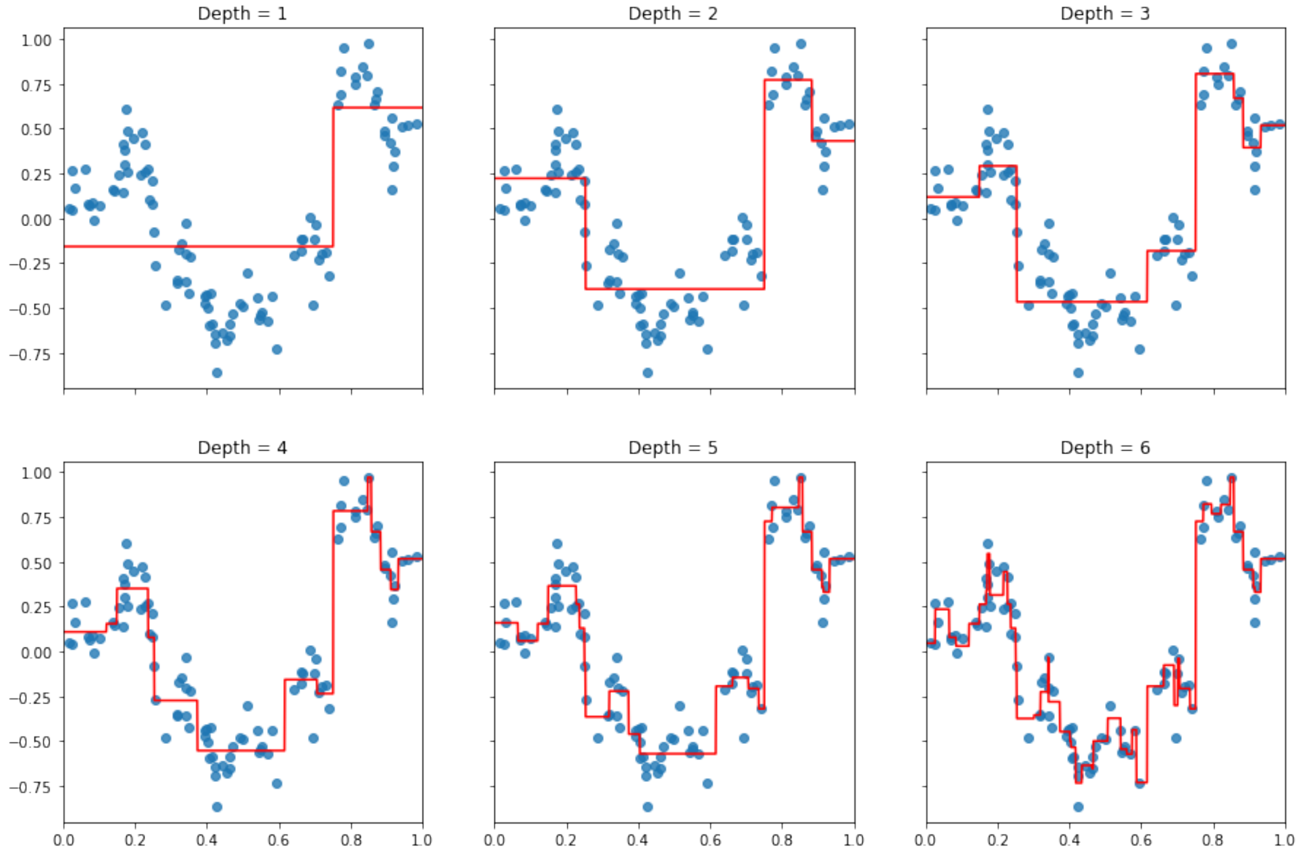
```
# Regression Tree Specific Code
def mean_absolute_deviation_around_median(y):
    '''
    Calulate the mean absolute deviation around the median of a given target

    :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 functions u
    :attribute estimator_dict: dictionary containing the estimation functions
    '''

    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, m
        '''
        Initialize Regression_Tree
        :param loss_function(str): loss function used for splitting internal 
        :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]:

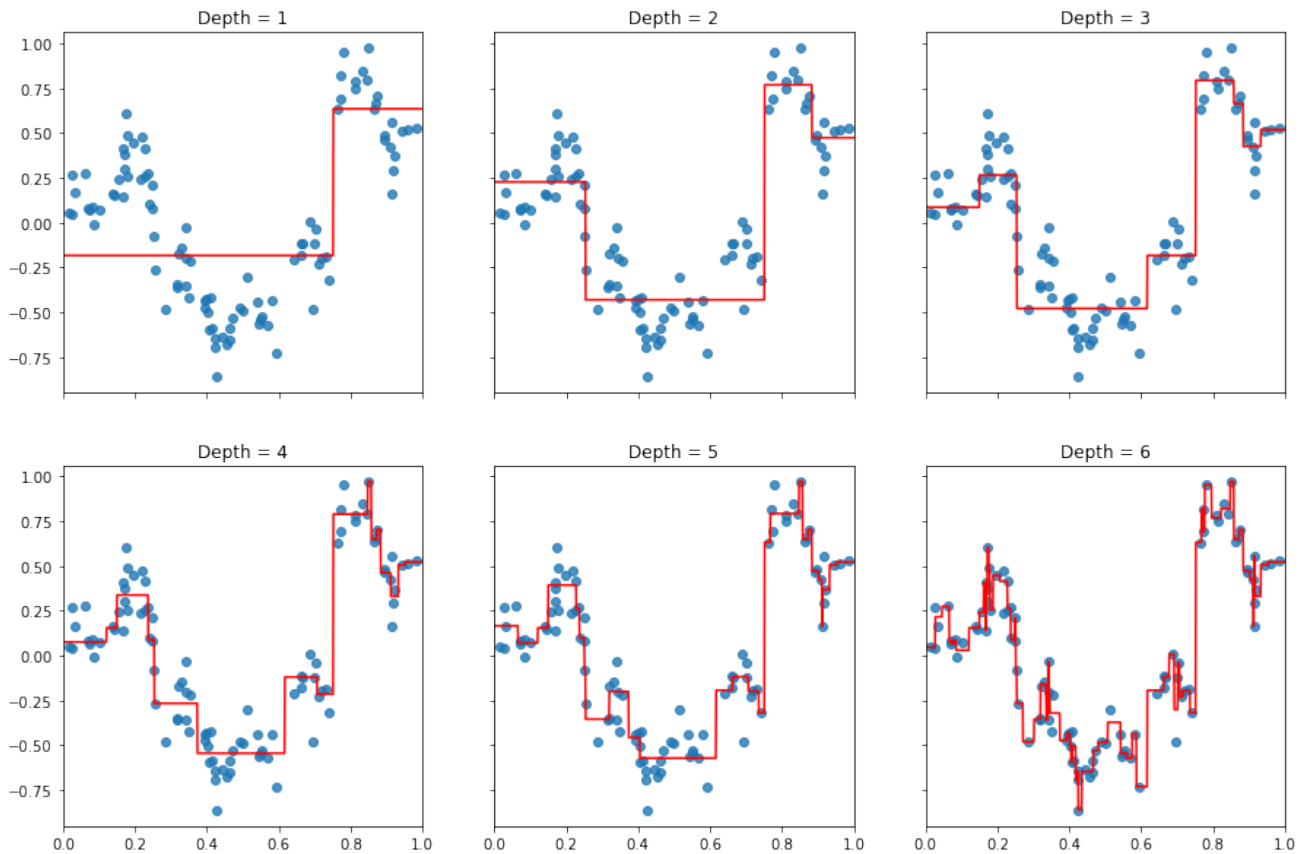
```
data_krr_train = np.loadtxt('krr-train.txt')
data_krr_test = np.loadtxt('krr-test.txt')
x_krr_train, y_krr_train = data_krr_train[:,0].reshape(-1,1),data_krr_train[:
x_krr_test, y_krr_test = data_krr_test[:,0].reshape(-1,1),data_krr_test[:,1].

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

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

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

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

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

clf6 = Regression_Tree(max_depth=10,  min_sample=3, loss_function='mae', esti
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]).re

    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, min_sam
Regression_Tree(max_depth=1,  min_sample=3, loss_function='mae', estimator='m
clf1.fit(x_krr_train, y_krr_train)

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

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

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

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

clf6 = DecisionTreeRegressor(criterion='absolute_error', max_depth=10, min_sa
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=(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 = clf.predict(np.array([x for x in x_range]).reshape(-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()
```



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_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 gr
        :pseudo_residual_func: function used for computing pseudo-residual be
        :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,)
        '''
        #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_sample,
                                          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 using
        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_sample,
                                              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):
        '''
```

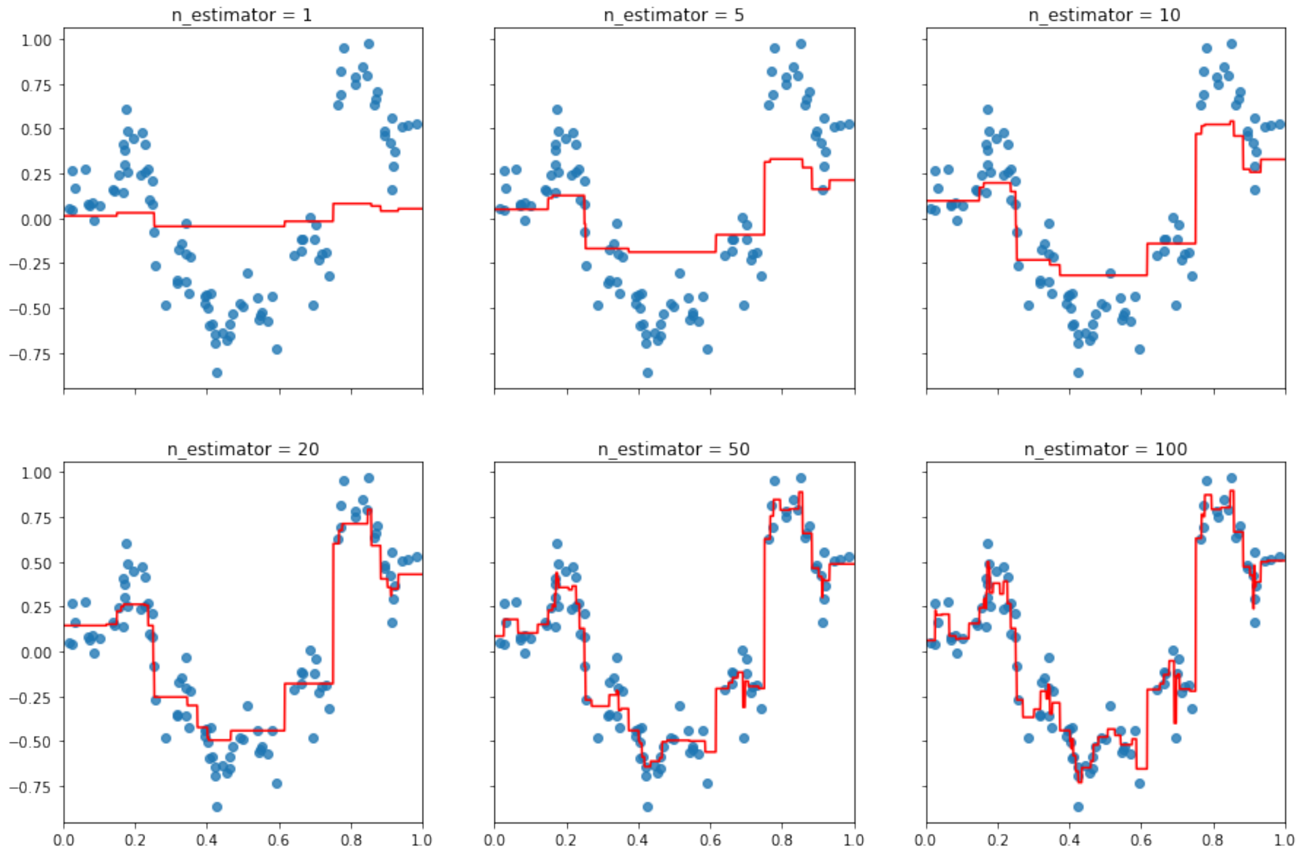
```
        Predict value
        :train_data array of inputs of size (n_samples, m_features)
        '''
        #Initialize prediction
        test_predict = np.zeros(len(test_data))

        #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 step si
            test_predict += self.estimators[i].predict(test_data) * self.lear

        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=(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,

    gbm_1d = gradient_boosting(n_estimator=i, pseudo_residual_func=pseudo_res
                               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_mnist_0
                                                        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]:

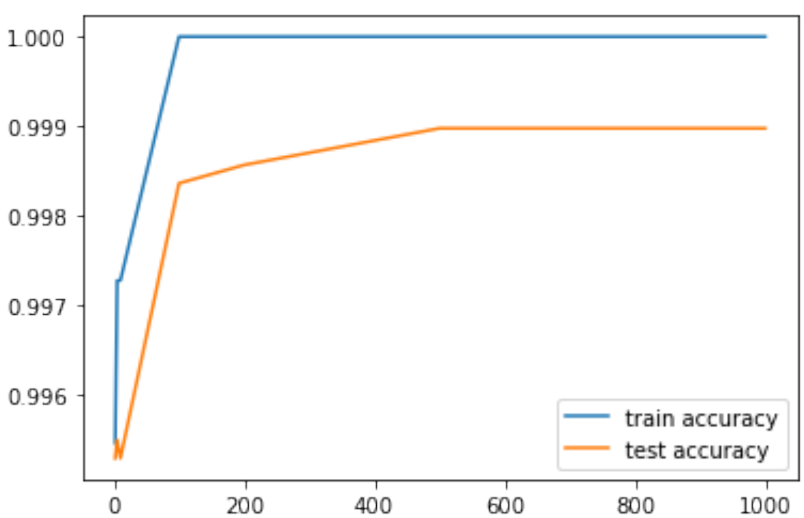
```
X_train, X_test, y_train, y_test = pre_process_mnist_01()
```

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', max_dep
    #Fit the GBC estimator
    gbc.fit(X_train, y_train)

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


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

Out[36]: []