CART-GBM-skeleton-code

April 29, 2019

```
In [23]: 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
         import graphviz
         import pdb
         from IPython.display import Image

         %matplotlib inline
```

1 Load Data

```
In [24]: 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 [25]: # 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
```

2 Decision Tree Class

```
In [26]: 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 for splitting node
                 :param leaf_value_estimator: method for estimating leaf value
                 :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

    def is_pure(self, y):
        return len(set(y.flatten())) <= 1


    def fit(self, X, y=None):
        '''
        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 u
        splitting the node, we should also init self.left and self.right to be Decision
        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
        '''

        if self.depth >= self.max_depth or len(y) <= self.min_sample or self.is_pure(y)
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)

        else:
            self.is_leaf = False
            splits = [] # contains (feature_index, split_value, loss) tuples
            for i, x in enumerate(X.T): # iterate over columns
                losses = [] # contain (split_value, loss) pairs
                for split_val in x:
                    left_y = y[x<=split_val]
                    right_y = y[x>split_val]
                    loss = len(left_y)*self.split_loss_function(left_y) + len(right_y)*
                    losses.append([split_val, loss])
                min_loss = min(losses, key=lambda x: x[1])
                splits.append([i] + min_loss)

            #pdb.set_trace()
            min_split = min(splits, key=lambda x: x[2])
            self.split_id = min_split[0]
            self.split_value = min_split[1]
```

```python
            left_x = X[X[:,self.split_id] <= self.split_value]
            right_x = X[X[:,self.split_id] > self.split_value]
            left_y = y[X[:,self.split_id] <= self.split_value]
            right_y = y[X[:,self.split_id] > self.split_value]

            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)
            #pdb.set_trace()
            try:
                self.left.fit(left_x, left_y)
                self.right.fit(right_x, right_y)
            except:
                pdb.set_trace()

        return self

    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 instan
        '''
        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)
```

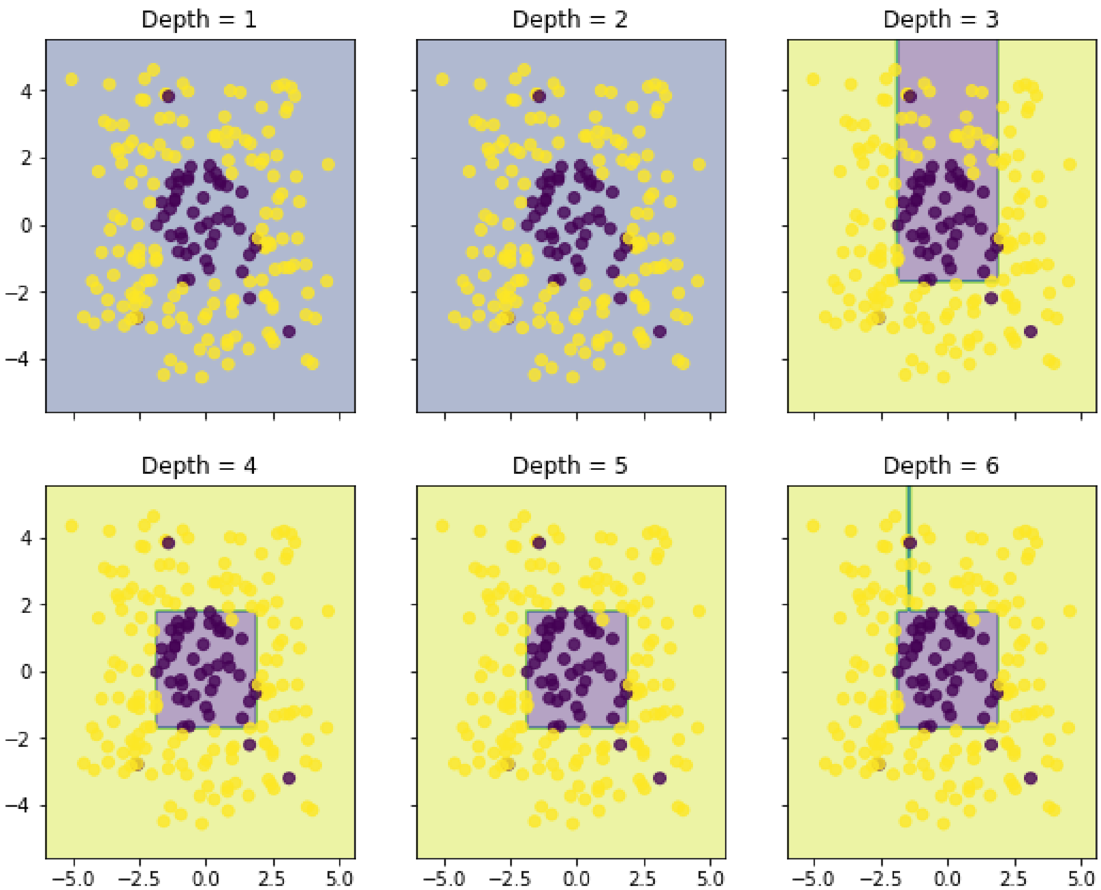
3 Decision Tree Classifier

```
In [27]: def compute_entropy(label_array):
             '''
             Calulate the entropy of given label list

             :param label_array: a numpy array of labels shape = (n, 1)
             :return entropy: entropy value
             '''
             counter = Counter(label_array.flatten())
             entropy = 0
             # pdb.set_trace()
             for c, n in counter.items():
                 p_c = float(n)/len(label_array)
                 entropy -= p_c*np.log2(p_c)
             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
             '''
             gini = 0
             for c, n in Counter(label_array.flatten()).items():
                 p_c = float(n)/len(label_array)
                 # gini += p_c * (1-p_c)
                 gini += p_c**2
             return 1 - gini

In [50]: def most_common_label(y):
             '''
             Find most common label
             '''
             label_cnt = Counter(y.reshape(len(y)))
             #pdb.set_trace()

             label = label_cnt.most_common(1)[0][0]
             #pdb.set_trace()


             return label

In [51]: class Classification_Tree(BaseEstimator, ClassifierMixin):

             loss_function_dict = {
                 'entropy': compute_entropy,
```

```python
        '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
```

4 Decision Tree Boundary

```
In [80]: # 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=6)
         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)
             #pdb.set_trace

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

         plt.show()
```



Depth = 1
Depth = 2
Depth = 3
Depth = 4
Depth = 5
Depth = 6
4
2
0
−2
−4
−5.0
−2.5
0.0
2.5
5.0

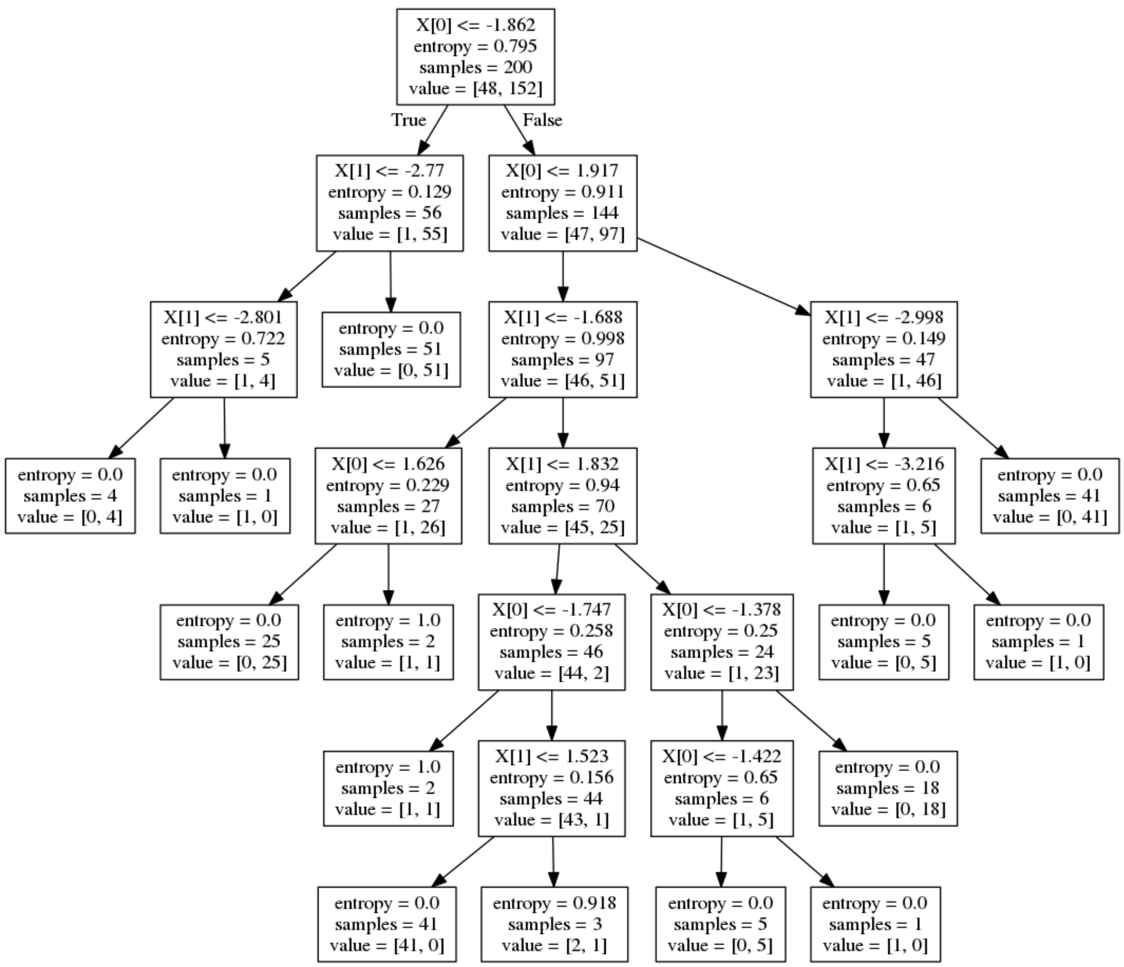
5 Compare decision tree with tree model in sklearn

In [75]:
```
clf = DecisionTreeClassifier(criterion='entropy', max_depth=6, min_samples_split=5)
clf.fit(x_train, y_train_label)
export_graphviz(clf, out_file='tree_classifier.dot')
```

In [76]:
```
# Visualize decision tree
!dot -Tpng tree_classifier.dot -o tree_classifier.png
```

In [77]:
```
Image(filename='tree_classifier.png')
```

Out[77]:



In [46]:
```
clf2.tree.left.right.value
```

Out[46]: 1

6 Decision Tree Regressor

```
In [67]: # 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
             '''
             mean = np.mean(y)
             n = len(y)
             mae = np.abs(y - mean).sum()/float(n)
             return mae
```

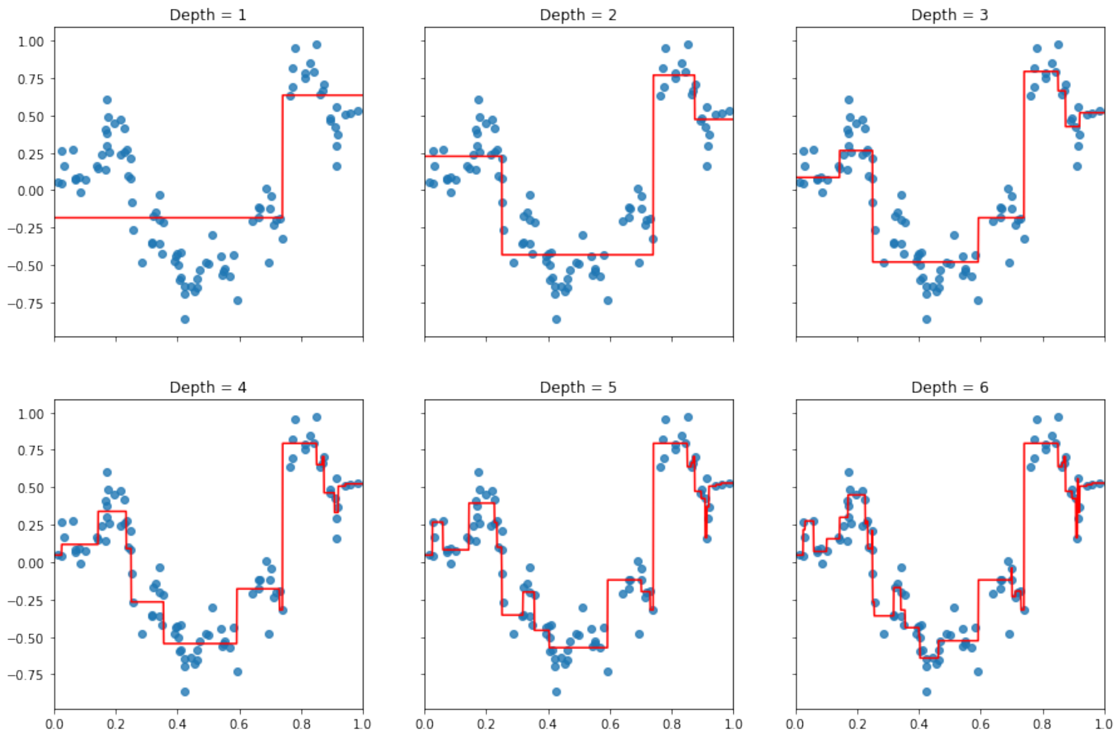
```
In [68]: class Regression_Tree():
             '''
             :attribute loss_function_dict: dictionary containing the loss functions used for sp
             :attribute estimator_dict: dictionary containing the estimation functions used in l
             '''


             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=1
                 '''
                 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
```

7 Fit regression tree to one-dimensional regression data

In [69]:
```
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].reshap
x_krr_test, y_krr_test = data_krr_test[:,0].reshape(-1,1),data_krr_test[:,1].reshape(-1

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

clf2 = Regression_Tree(max_depth=2,  min_sample=1, loss_function='mae', estimator='medi
clf2.fit(x_krr_train, y_krr_train)

clf3 = Regression_Tree(max_depth=3,  min_sample=1, loss_function='mae', estimator='medi
clf3.fit(x_krr_train, y_krr_train)

clf4 = Regression_Tree(max_depth=4,  min_sample=1, loss_function='mae', estimator='medi
clf4.fit(x_krr_train, y_krr_train)

clf5 = Regression_Tree(max_depth=5,  min_sample=1, loss_function='mae', estimator='medi
clf5.fit(x_krr_train, y_krr_train)

clf6 = Regression_Tree(max_depth=6,  min_sample=1, loss_function='mae', estimator='medi
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,

    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()
```

```
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/numpy/core/fromnumeric.py:3118: Run
  out=out, **kwargs)
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/numpy/core/_methods.py:85: RuntimeW
  ret = ret.dtype.type(ret / rcount)
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/ipykernel/__main__.py:11: RuntimeWa
```


Depth = 1
Depth = 2
Depth = 3
Depth = 4
Depth = 5
Depth = 6
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0.0
0.2
0.4
0.6
0.8
1.0

8 Gradient Boosting Method

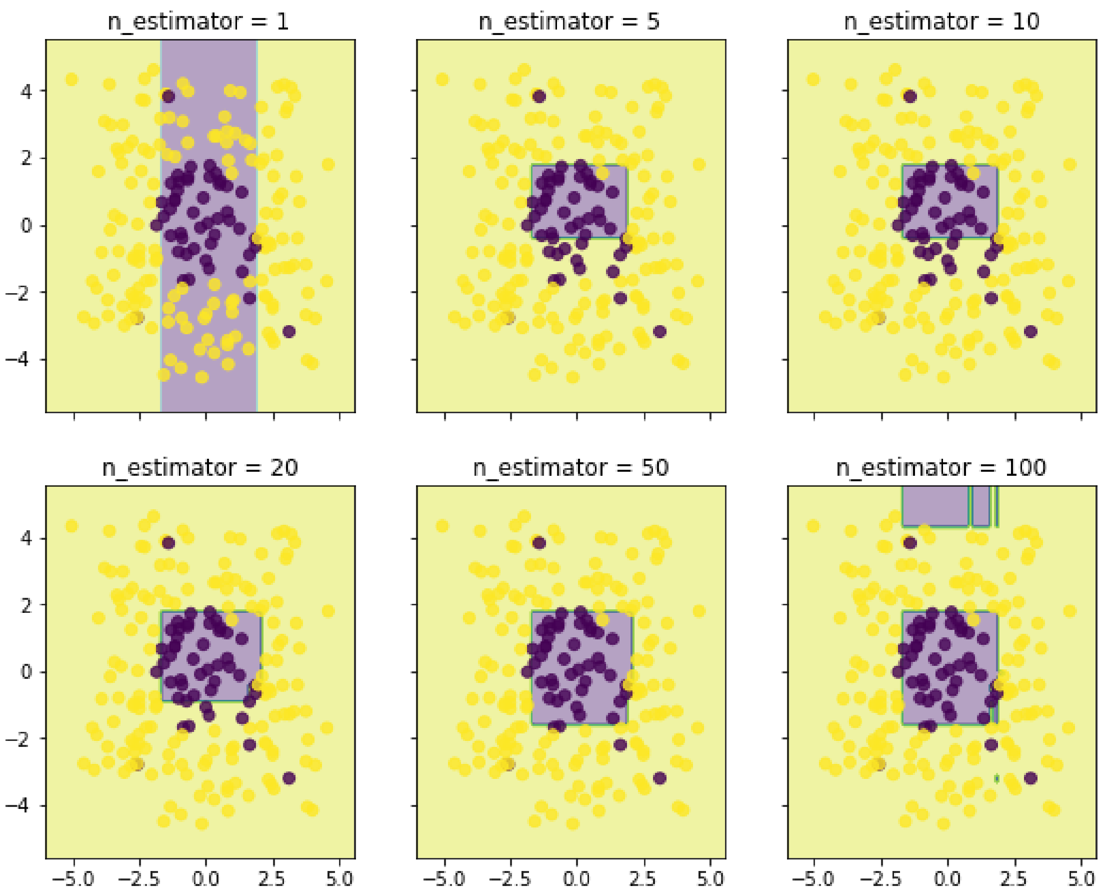
In [15]:
```
#Pseudo-residual function.
#Here you can assume that we are using L2 loss

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

class ConstantModel(BaseEstimator, RegressorMixin):
    def __init__(self, c):
        self.c = c

    def fit(self, x, y=None):
        pass

    def predict(self, x):
        return self.c * np.ones(len(x))
```

In [18]:
```
class gradient_boosting():
    '''
    Gradient Boosting regressor class
    :method fit: fitting model
    '''
    def __init__(self, n_estimator, pseudo_residual_func, learning_rate=0.1, min_sample
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounds of gradient boo
        :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 = [ConstantModel(c=0)]

    def calc_pseudo_residual(self, X, y):
        return self.predict(X) - y.flatten()

    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        '''
```

```python
        for _ in range(self.n_estimator):
            pseudo_residual = self.calc_pseudo_residual(train_data, train_target)
            #pdb.set_trace()
            new_estimator = DecisionTreeRegressor(max_depth=self.max_depth,
                                                  min_samples_split=self.min_sample)
            new_estimator.fit(X=train_data, y=-pseudo_residual)
            #pdb.set_trace()
            self.estimators.append(new_estimator)


    def predict(self, test_data):
        '''
        Predict value
        '''
        preds = np.zeros(len(test_data))
        for estimator in self.estimators:
            preds += self.learning_rate * estimator.predict(test_data)
        return preds
```

9 2-D GBM visualization - SVM data

In [19]:
```
# 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, 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]])

    gbt = gradient_boosting(n_estimator=i, pseudo_residual_func=pseudo_residual_L2, max
    gbt.fit(x_train, y_train)

    Z = np.sign(gbt.predict(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.flatten
    axarr[idx[0], idx[1]].set_title(tt)
```


n_estimator = 1
n_estimator = 5
n_estimator = 10
n_estimator = 20
n_estimator = 50
n_estimator = 100
4
2
0
−2
−4
−5.0
−2.5
0.0
2.5
5.0

10 1-D GBM visualization - KRR data

In [22]:
```
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,
    gbm_1d.fit(x_krr_train, y_krr_train)

    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)
```