1. Reformulations of Multiclass Hinge Loss

Problem 1.2.1. Show that if $\Delta(y, y) = 0$ for all $y \in \mathcal{Y}$, then $\ell_2\left(h, (x_i, y_i)\right) = \ell_1(h, (x_i, y_i))$.

Solution Assume $\Delta(y, y) = 0$ for all $y \in \mathcal{Y}$, then

$$\begin{aligned}
\ell_2(h, (x_i, y_i)) &= \max_{y \in \mathcal{Y}} \left[\Delta\left(y_i, y\right) + h(x_i, y) - h(x_i, y_i)\right] \\
&= \max\left[\Delta\left(y_i, y_i\right) + h(x_i, y_i) - h(x_i, y_i), \max_{y \neq y_i}\left[\Delta\left(y_i, y\right) + h(x_i, y) - h(x_i, y_i)\right]\right] \\
&= \max\left[0, \max_{y \neq y_i}\left[\Delta\left(y_i, y\right) + h(x_i, y) - h(x_i, y_i)\right]\right] \\
&= \max_{y \neq y_i}\left[\max\left[0, \Delta\left(y_i, y\right) + h(x_i, y) - h(x_i, y_i)\right]\right] \\
&= \ell_1(h, (x_i, y_i))
\end{aligned}$$

Problem 1.2.2.a. Show that under the conditions above, $\ell_1(h, (x_i, y_i)) = \ell_2(h, (x_i, y_i)) = 0$.

Solution Since $\Delta(y, y) = 0$, $\ell_1 = \ell_2$.
Also, since $m_{i,y}(h) = h(x_i, y_i) - h(x_i, y) \geq \Delta(y_i, y)$

$$\Delta(y_i, y) - m_{i,y}(h) = \Delta(y_i, y) + h(x_i, y) - h(x_i, y_i) \leq 0 \qquad \forall y \neq y_i$$

Then it is clear that $\ell_1(h, (x_{i,}y_i)) = 0 = \ell_2(h, (x_{i,}y_i))$

Problem 1.2.2.b. Show that under the conditions above, we make the correct prediction on x_i. That is, $f(x_i) = \arg\max_{y \in \mathcal{Y}} h(x_i, y) = y_i$.

Solution Assume $f(x_i) = \arg\max_{y \in \mathcal{Y}} h(x_i, y) \neq y_i$.
Then $\exists\ y'$ such that $h(x_i, y') > h(x_i, y_i)$.
Then $h(x_i, y_i) - h(x_i, y) < 0$. But this contradicts the fact that

$$h(x_i, y_i) - h(x_i, y') \geq \Delta(y_i, y) > 0$$

Thus we conclude that $f(x_i) = \arg\max_{y \in \mathcal{Y}} h(x_i, y) = y_i$

2. SGD for Multiclass Linear SV

Problem 2.2. Since $J(w)$ is convex, it has a subgradient at every point. Give an expression for a subgradient of $J(w)$. You may use any standard results about subgradients, including the result from an earlier homework about subgradients of the pointwise maxima of functions.

Solution

$$\Delta J(w) = 2\lambda w + \frac{1}{n}\sum_{i=1}^{n}\left[\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)\right]$$

Problem 2.3. Give an expression for the stochastic subgradient based on the point (x_i, y_i).

Solution

$$\Delta J(w) = 2\lambda w + \Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)$$

Problem 2.4.

Give an expression for a minibatch subgradient, based on the points $(x_i, y_i), \ldots, (x_{i+m-1}, y_{i+m-1})$.

Solution

$$\Delta J(w) = 2\lambda w + \frac{1}{m}\sum_{i=1}^{m}\left[\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)\right]$$

3. Hinge Loss is a Special Case of Generalized Hinge Loss

Problem 3. Let $\mathcal{Y} = \{-1, 1\}$. Let $\Delta(y, \hat{y}) = 1(y \neq \hat{y})$. If $g(x)$ is the score function in our binary classification setting, then define our compatibility function as

$$\begin{aligned} h(x,1) &= g(x)/2 \\ h(x,-1) &= -g(x)/2. \end{aligned}$$

Show that for this choice of h, the multiclass hinge loss reduces to hinge loss:

$$\ell\left(h,(x,y)\right) = \max_{y' \in \mathcal{Y}} \left[\Delta\left(y,y'\right)) + h(x,y') - h(x,y)\right] = \max\left\{0, 1 - yg(x)\right\}$$

Solution Note that

$$\ell\left(h,(x,y)\right) = \max\left[\Delta\left(-1,y'\right)) + h(x,y') - h(x,-1), \Delta\left(1,y'\right)) + h(x,y') - h(x,1)\right]$$

Either $y = y'$ or $y \neq y'$.
If $y = y'$, then $\ell(h,(x,y)) = 0$.
Otherwise

$$\begin{aligned} \ell(h,(x,y)) &= \Delta(y,y') + h(x,y') - h(x,y) \\ &= \begin{cases} 1 + g(x) & \text{if } y = -1 \\ 1 - g(x) & \text{if } y = 1 \end{cases} \\ &= 1 - yg(x) \end{aligned}$$

Thus $\ell\left(h,(x,y)\right) = \max\left\{0, 1 - yg(x)\right\}$

Gradient Boosting Machines

Problem 7.1. Consider the regression framework, where $\mathcal{Y} = \mathbf{R}$. Suppose our loss function is given by

$$\ell(\hat{y}, y) = \frac{1}{2}\left(\hat{y} - y\right)^2,$$

and at the beginning of the m'th round of gradient boosting, we have the function $f_{m-1}(x)$. Show that the h_m chosen as the next basis function is given by

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

In other words, at each stage we find the base prediction function $h_m \in \mathcal{F}$ that is the best fit to the residuals from the previous stage.

Solution Note that

$$\begin{aligned}
(g_m)_i &= \frac{\partial}{\partial f(x_i)} \sum_{j=1}^{n} \ell(y_j, f(x_j)) \\
&= \frac{\partial}{\partial f(x_i)} \frac{1}{2}(y_i - f(x_i))^2 \\
&= f(x_i) - y_i
\end{aligned}$$

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

Problem 7.2. Now 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. Similar to what we did in the ℓ_2-Boosting question, write an expression for h_m as an argmin over $\mathcal{F}$.

Solution Note that $\ell(y, f(x)) = \ln\left(1 + e^{-yf(x)}\right)$.
Then

$$\begin{aligned}
(g_m)_i &= \frac{\partial}{\partial f(x_i)} \sum_{j=1}^{n} \ln\left(1 + e^{-y_j f(x_j)}\right) \\
&= \frac{\partial}{\partial f(x_i)} \ln\left(1 + e^{-y_i f(x_i)}\right) \\
&= \frac{-y_i e^{-y_i f(x_i)}}{1 + e^{-y_i f(x_i)}}
\end{aligned}$$

Thus $h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left[\frac{-y_i e^{-y_i f(x_i)}}{1+e^{-y_i f(x_i)}} - h(x_i)\right]^2$

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]
```

```
            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
```

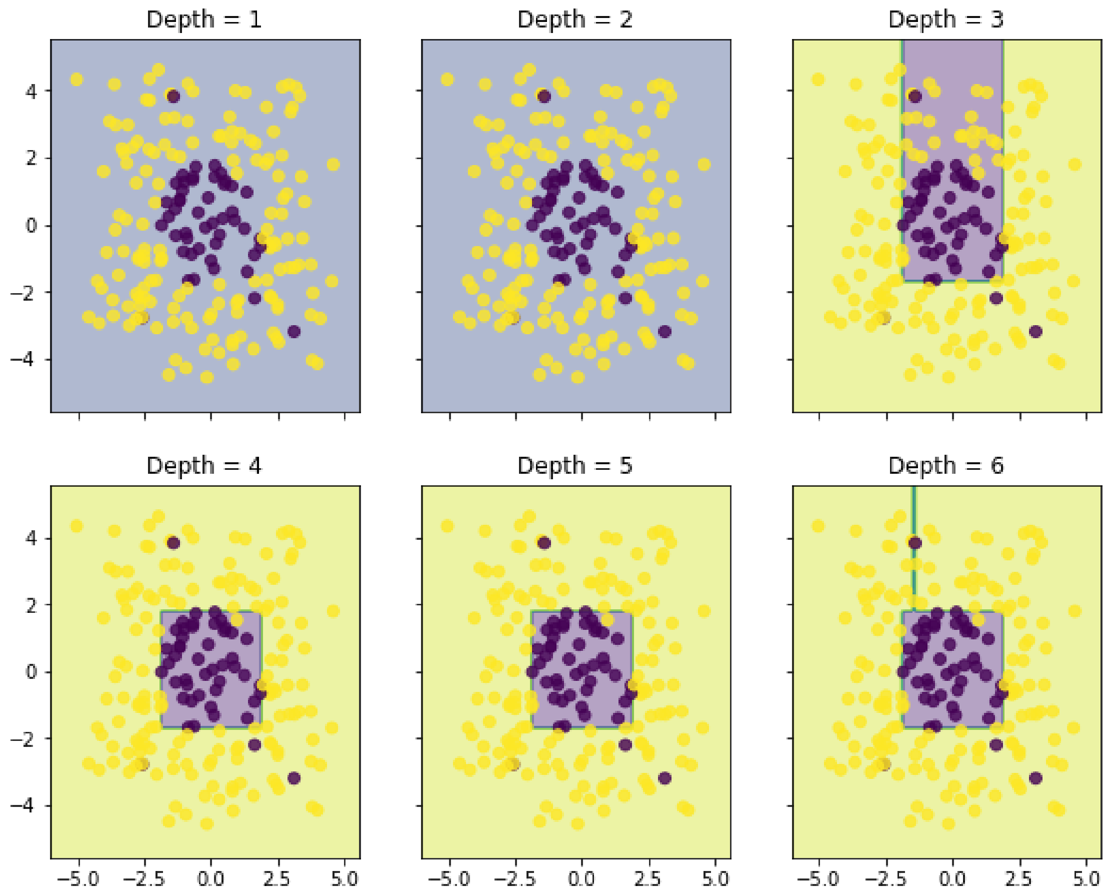
In [50]:
```
def most_common_label(y):
    '''
    Find most common label
    '''
    label_cnt = Counter(y.reshape(len(y)))
    #pdb.set_trace()

    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,
```

```
    '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]:
```python
# 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

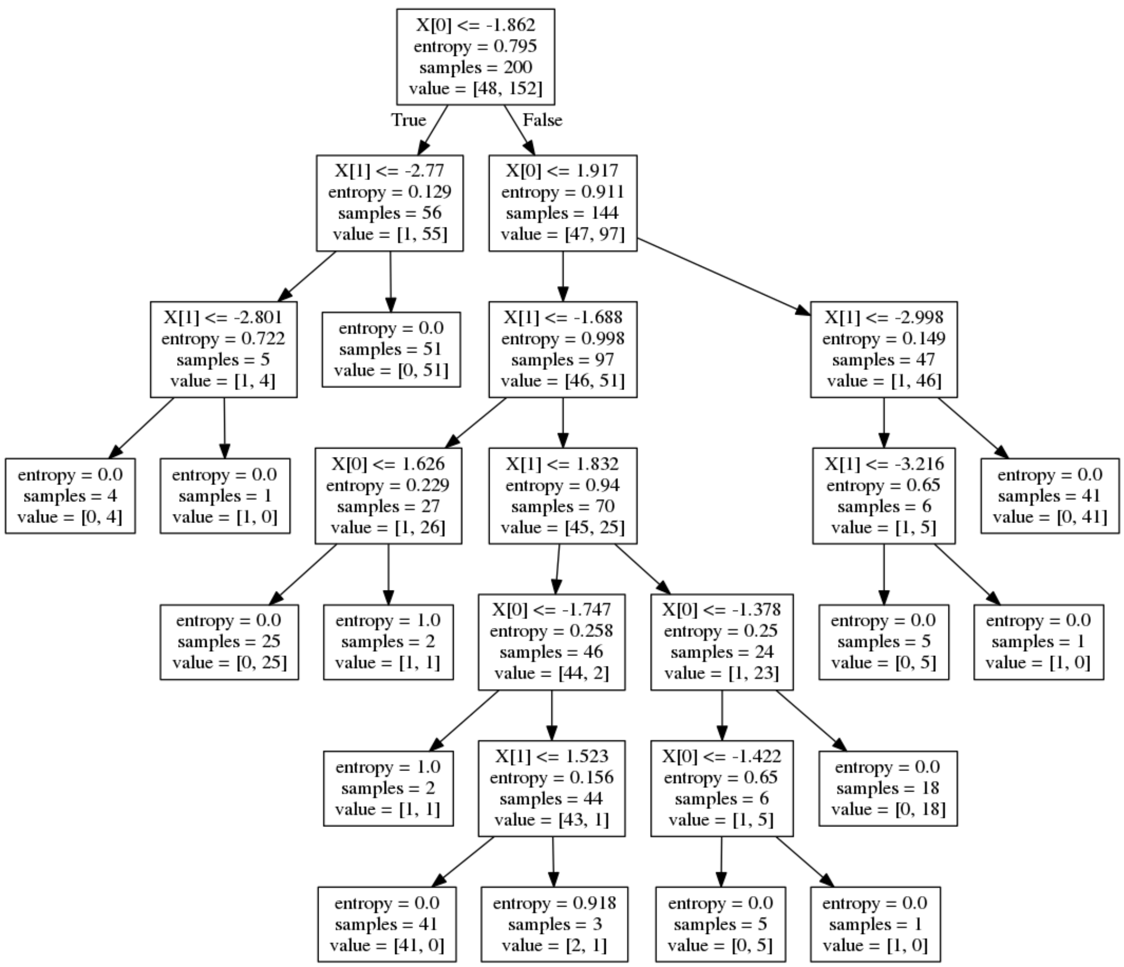
5 Compare decision tree with tree model in sklearn

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
```

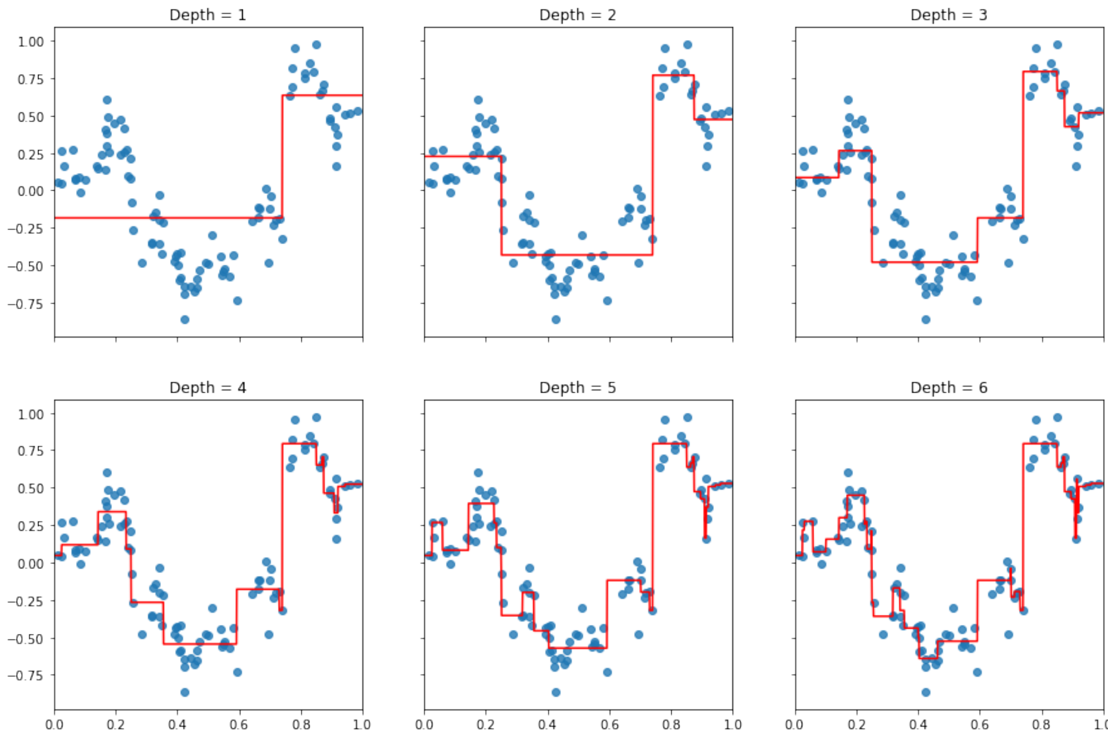
```
In [68]: class Regression_Tree():
             '''
             :attribute loss_function_dict: dictionary containing the loss functions used for sp
             :attribute estimator_dict: dictionary containing the estimation functions used in l
             '''

             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

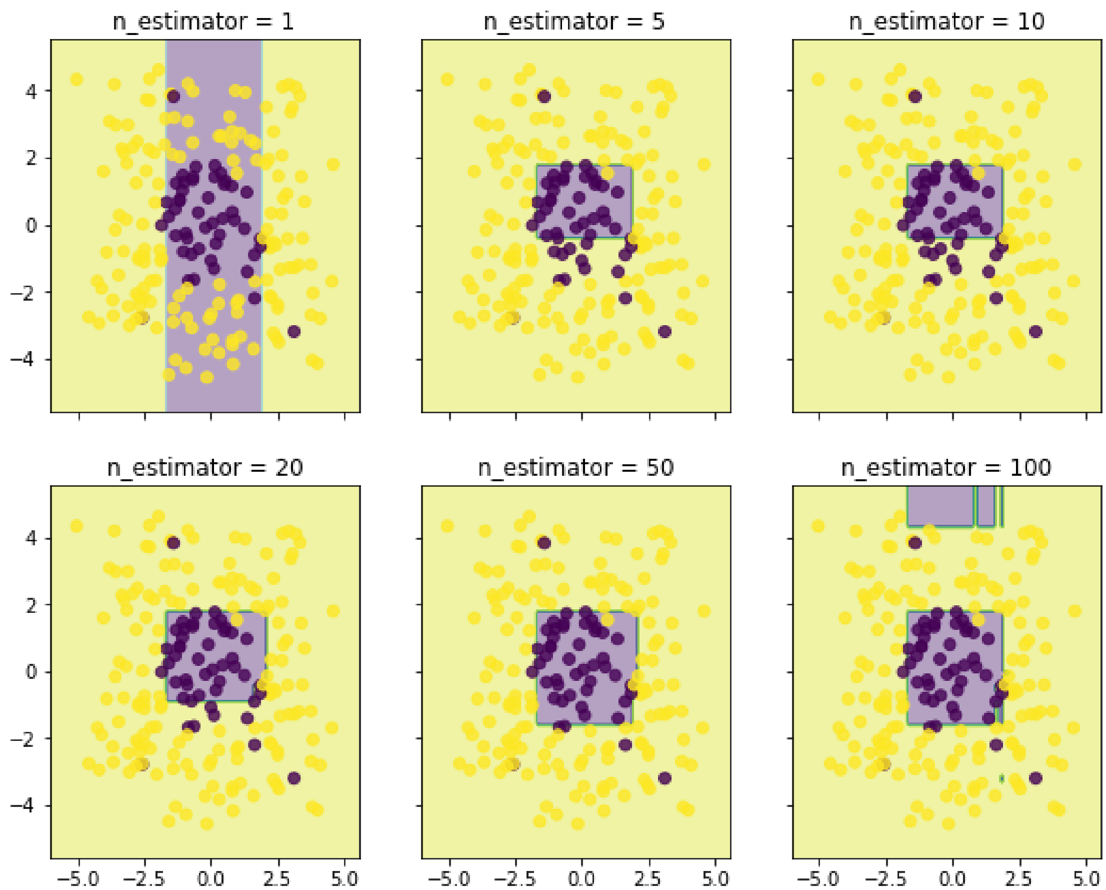
```
In [15]: #Pseudo-residual function.
         #Here you can assume that we are using L2 loss

         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

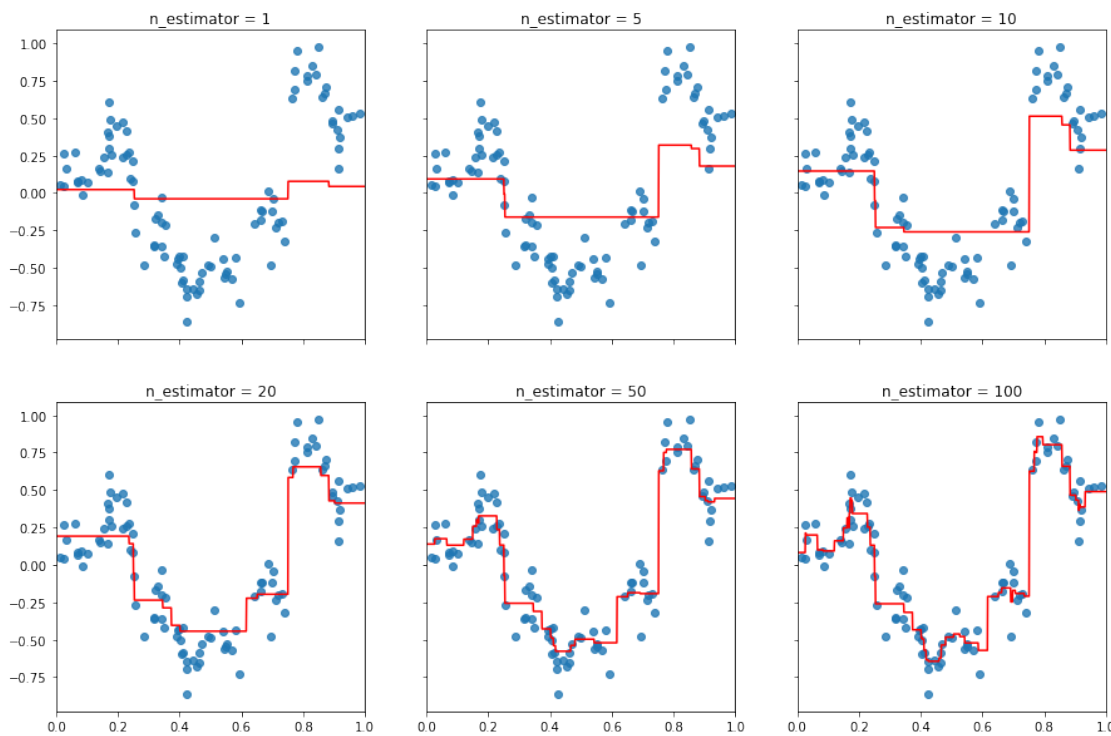
In [19]:
```
# Plotting decision regions
x_min, x_max = x_train[:, 0].min() - 1, x_train[:, 0].max() + 1
y_min, y_max = x_train[:, 1].min() - 1, x_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

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

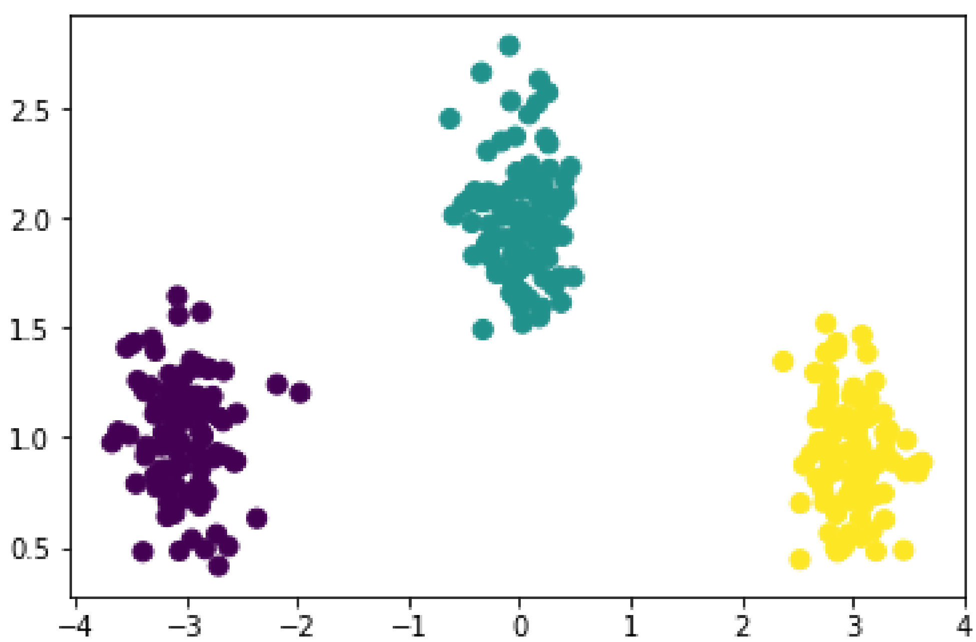
In [22]:
```
plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)

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)
```



multiclass-skeleton-code

April 29, 2019

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from sklearn.datasets.samples_generator import make_blobs
        import pdb
        from collections.abc import Iterable
        from tqdm import tqdm

        %matplotlib inline

In [2]: # Create the  training data
        np.random.seed(2)
        X, y = make_blobs(n_samples=300,cluster_std=.25, centers=np.array([(-3,1),(0,2),(3,1)]))
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50)

Out[2]: <matplotlib.collections.PathCollection at 0x7fa982119470>
```



```
In [3]: from sklearn.base import BaseEstimator, ClassifierMixin, clone

        class OneVsAllClassifier(BaseEstimator, ClassifierMixin):
            """
            One-vs-all classifier
            We assume that the classes will be the integers 0,..,(n_classes-1).
            We assume that the estimator provided to the class, after fitting, has a "decision_f
            returns the score for the positive class.
            """
            def __init__(self, estimator, n_classes):
                """
                Constructed with the number of classes and an estimator (e.g. an
                SVM estimator from sklearn)
                @param estimator : binary base classifier used
                @param n_classes : number of classes
                """
                self.n_classes = n_classes
                self.estimators = [clone(estimator) for _ in range(n_classes)]
                self.fitted = False

            def fit(self, X, y=None):
                """
                This should fit one classifier for each class.
                self.estimators[i] should be fit on class i vs rest
                @param X: array-like, shape = [n_samples,n_features], input data
                @param y: array-like, shape = [n_samples,] class labels
                @return returns self
                """
                for class_n, estimator in enumerate(self.estimators):
                    labels = (y == class_n).astype(int)
                    estimator.fit(X, labels)
                self.fitted = True
                return self

            def decision_function(self, X):
                """
                Returns the score of each input for each class. Assumes
                that the given estimator also implements the decision_function method (which skl
                and that fit has been called.
                @param X : array-like, shape = [n_samples, n_features] input data
                @return array-like, shape = [n_samples, n_classes]
                """
                if not self.fitted:
                    raise RuntimeError("You must train classifer before predicting data.")

                if not hasattr(self.estimators[0], "decision_function"):
                    raise AttributeError(
                        "Base estimator doesn't have a decision_function attribute.")
```

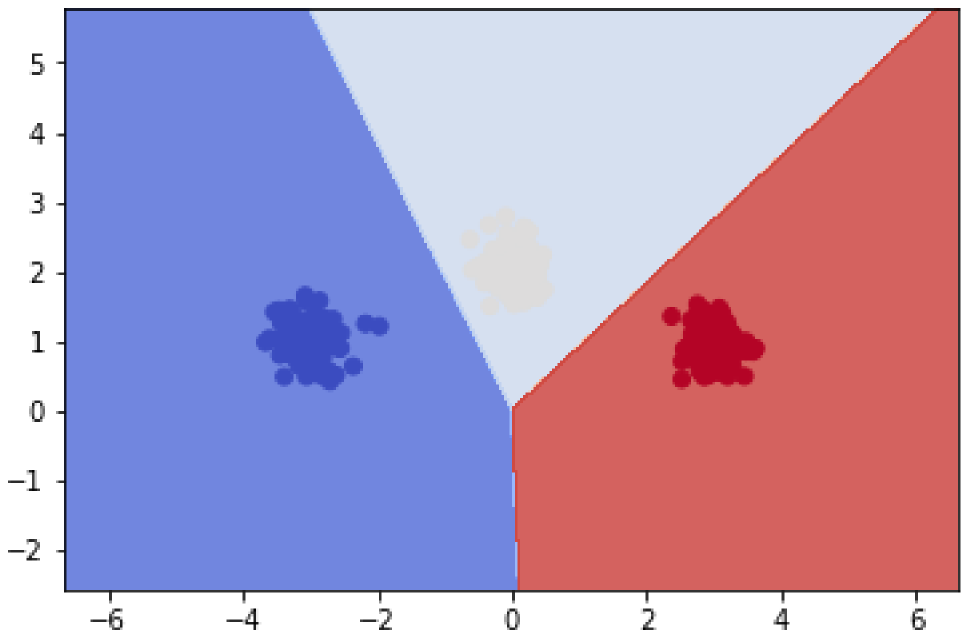
```
            return np.array([e.decision_function(X) for e in self.estimators]).T

        def predict(self, X):
            """
            Predict the class with the highest score.
            @param X: array-like, shape = [n_samples,n_features] input data
            @returns array-like, shape = [n_samples,] the predicted classes for each input
            """
            return np.argmax(self.decision_function(X), axis=1)
```

```
In [4]: #Here we test the OneVsAllClassifier
        from sklearn import svm
        svm_estimator = svm.LinearSVC(loss='hinge', fit_intercept=False, C=200)
        clf_onevsall = OneVsAllClassifier(svm_estimator, n_classes=3)
        clf_onevsall.fit(X,y)

        for i in range(3) :
            print("Coeffs %d"%i)
            print(clf_onevsall.estimators[i].coef_) #Will fail if you haven't implemented fit ye

        # create a mesh to plot in
        h = .02  # step size in the mesh
        x_min, x_max = min(X[:,0])-3,max(X[:,0])+3
        y_min, y_max = min(X[:,1])-3,max(X[:,1])+3
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))
        mesh_input = np.c_[xx.ravel(), yy.ravel()]

        Z = clf_onevsall.predict(mesh_input)
        Z = Z.reshape(xx.shape)
        plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
        # Plot also the training points
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)


        from sklearn import metrics
        metrics.confusion_matrix(y, clf_onevsall.predict(X))
```

```
Coeffs 0
[[-1.05852964 -0.90293998]]
Coeffs 1
[[ 0.31047003 -0.19010007]]
Coeffs 2
[[ 0.8916347  -0.82599479]]
```

```
/home/cfizette/anaconda3/envs/ml/lib/python3.6/site-packages/sklearn/svm/base.py:931: Convergenc
  "the number of iterations.", ConvergenceWarning)
```

```
Out[4]: array([[100,   0,   0],
               [  0, 100,   0],
               [  0,   0, 100]])
```



Multiclass SVM

```
In [16]: def zeroOne(y,a) :
             '''
             Computes the zero-one loss.
             @param y: output class
             @param a: predicted class
             @return 1 if different, 0 if same
             '''
             return int(y != a)

         def featureMap(X,y,num_classes) :
             '''
             Computes the class-sensitive features.
             @param X: array-like, shape = [n_samples,n_inFeatures] or [n_inFeatures,], input fe
             @param y: a target class (in range 0,..,num_classes-1)
             @return array-like, shape = [n_samples,n_outFeatures], the class sensitive features
             '''
             #The following line handles X being a 1d-array or a 2d-array
             num_samples, num_inFeatures = (1,X.shape[0]) if len(X.shape) == 1 else (X.shape[0],
             if num_samples == 1:
                 X = [X]

             # n_out_features = n_classes * n_in_features
             # feature_map[y*n_in_features : y*n_in_features + n_in_features] = X

             feature_map = np.zeros((num_samples, num_inFeatures * num_classes))

             for i, xi in enumerate(X):
                 start = y*num_inFeatures
                 end = start + num_inFeatures
                 feature_map[i,start:end] = xi

             '''
             for i in range(num_samples):
                 xi = X[i]
                 yi = y
                 start = yi*num_inFeatures
                 end = start + num_inFeatures
                 feature_map[i,start:end] = xi
             '''
             return feature_map

         def sgd(X, y, num_outFeatures, subgd, eta = 0.1, T = 10000):
             '''
             Runs subgradient descent, and outputs resulting parameter vector.
             @param X: array-like, shape = [n_samples,n_features], input training data
             @param y: array-like, shape = [n_samples,], class labels
```

```
    @param num_outFeatures: number of class-sensitive features
    @param subgd: function taking x,y and giving subgradient of objective
    @param eta: learning rate for SGD
    @param T: maximum number of iterations
    @return: vector of weights
    '''
    num_samples = X.shape[0]
    w = np.zeros(num_outFeatures)
    for _ in tqdm(range(T)):
        for xi, yi in zip(X, y):
            w -= eta * subgd(xi, yi, w)
    return w

class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Psi=feat
        '''
        Creates a MulticlassSVM estimator.
        @param num_outFeatures: number of class-sensitive features produced by Psi
        @param lam: l2 regularization parameter
        @param num_classes: number of classes (assumed numbered 0,..,num_classes-1)
        @param Delta: class-sensitive loss function taking two arguments (i.e., target
        @param Psi: class-sensitive feature map taking two arguments
        '''
        self.num_outFeatures = num_outFeatures
        self.lam = lam
        self.num_classes = num_classes
        self.Delta = Delta
        self.Psi = lambda X,y : Psi(X,y,num_classes)
        self.fitted = False

    def generalized_hinge_loss(self, x, y, w):
        return max([self.class_hinge_loss(x,y,w,y_pred) for y_pred in range(self.n_clas

    def class_hinge_loss(self, x, y, w, y_pred):
        return self.Delta(y, y_pred) + (self.Psi(x, y_pred) - self.Psi(x, y))@w

    def subgradient(self,x,y,w):
        '''
        Computes the subgradient at a given data point x,y
        @param x: sample input
        @param y: sample class
        @param w: parameter vector
        @return returns subgradient vector at given x,y,w
        '''
        y_hat = np.argmax([self.class_hinge_loss(x,y,w,y_pred) for y_pred in range(self
```

```
            sgd = 2 * self.lam * w + self.Psi(x,y_hat).flatten() - self.Psi(x,y).flatten()
            return sgd

        def fit(self,X,y,eta=0.1,T=10000):
            '''
            Fits multiclass SVM
            @param X: array-like, shape = [num_samples,num_inFeatures], input data
            @param y: array-like, shape = [num_samples,], input classes
            @param eta: learning rate for SGD
            @param T: maximum number of iterations
            @return returns self
            '''
            self.coef_ = sgd(X,y,self.num_outFeatures,self.subgradient,eta,T)
            self.fitted = True
            return self

        def decision_function(self, X):
            '''
            Returns the score on each input for each class. Assumes
            that fit has been called.
            @param X : array-like, shape = [n_samples, n_inFeatures]
            @return array-like, shape = [n_samples, n_classes] giving scores for each sampl
            '''
            if not self.fitted:
                raise RuntimeError("You must train classifer before predicting data.")

            return np.array([self.Psi(X, y)@self.coef_ for y in range(self.num_classes)]).T

        def predict(self, X):
            '''
            Predict the class with the highest score.
            @param X: array-like, shape = [n_samples, n_inFeatures], input data to predict
            @return array-like, shape = [n_samples,], class labels predicted for each data
            '''
            raw_pred = self.decision_function(X)
            pred = np.argmax(raw_pred, axis=1)
            return pred
```

In [17]:
```
#the following code tests the MulticlassSVM and sgd
#will fail if MulticlassSVM is not implemented yet
est = MulticlassSVM(6,lam=1)
est.fit(X,y, T=500, eta=0.01)
print("w:")
print(est.coef_)
Z = est.predict(mesh_input)
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
# Plot also the training points
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)


from sklearn import metrics
metrics.confusion_matrix(y, est.predict(X))
```

```
100%|| 500/500 [00:10<00:00, 46.90it/s]


w:
[-0.34533686 -0.04675874  0.02345881  0.06871364  0.32187805 -0.02195491]
```

```
Out[17]: array([[100,   0,   0],
                [  0, 100,   0],
                [  0,   0, 100]])
```