NYU Center for Data Science

DS-GA 1003: Machine Learning (Spring 2020)

Homework 6: Multiclass, Trees, Gradient Boosting

Name: Yibo Liu (yl6769)

Due: Tuesday, May 1, 2020 at 11:59pm

Instructions. You should upload your code and plots to Gradescope. Please map the Gradescope entry on the rubric to your name and NetId. You must follow the policies for submission detailed in Homework 0.

---

1 SGD for Multiclass Linear SVM

Suppose our output space and our action space are given as follows: $\mathcal{Y} = \mathcal{A} = \{1, \ldots, k\}$. Given a non-negative class-sensitive loss function $\Delta : \mathcal{Y} \times \mathcal{A} \to [0, \infty)$ and a class-sensitive feature mapping $\Psi : \mathcal{X} \times \mathcal{Y} \to \mathbf{R}^d$. Our prediction function $f : \mathcal{X} \to \mathcal{Y}$ is given by

$$f_w(x) = \arg\max_{y \in \mathcal{Y}} \langle w, \Psi(x, y) \rangle$$

For training data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, let $J(w)$ be the ℓ_2-regularized empirical risk function for the multiclass hinge loss. We can write this as

$$J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max_{y \in \mathcal{Y}} \left[\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right],$$

for some $\lambda > 0$.

1. Show that $J(w)$ is a convex function of w. You may use any of the rules about convex functions described in our notes on Convex Optimization, in previous assignments, or in the Boyd and Vandenberghe book, though you should cite the general facts you are using. [Hint: If $f_1, \ldots, f_m : \mathbf{R}^n \to \mathbf{R}$ are convex, then their pointwise maximum $f(x) = \max\{f_1(x), \ldots, f_m(x)\}$ is also convex.]

   Answer:
   $\max_{y \in \mathcal{Y}} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle]$ is the maximum over the linear function of w, so it is convex. Thus the point-wise sum $\sum_{i=1}^{n} \max_{y \in \mathcal{Y}} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle]$ is also convex. Besides, since $w \in \mathcal{R}^d$, its L2 norm $||w||$ is convex. Therefore, $J(w)$ is convex.

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. (Hint: It may be helpful to refer to $\hat{y}_i = \arg\max_{y \in \mathcal{Y}} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle]$.)

   Answer:
   $\hat{y}_i = \arg\max_{y \in \mathcal{Y}} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle]$
   $\Rightarrow J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} [\Delta(y_i, \hat{y}) + \langle w, \Psi(x_i, \hat{y}) - \Psi(x_i, y_i) \rangle]$
   The subgradient of $J(w)$ on w: $2\lambda w + \frac{1}{n} \sum_{i=1}^{n} [\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)]$

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

   Answer:
   $2\lambda w + [\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)]$

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

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

2 [Optional] Hinge Loss is a Special Case of Generalized Hinge Loss

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:

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

Answer:
When $y = y'$, $l = 0$.
When $y \neq y'$,
$l(h, (x, 1)) = \max_{y' \in \mathcal{Y}}[1 + h(x, -1) - h(x, 1)] = \max(0, 1 - g(x)) = \max(0, 1 - yg(x))$
$l(h, (x, -1)) = \max_{y' \in \mathcal{Y}}[1 + h(x, 1) - h(x, -1)] = \max(0, 1 + g(x)) = \max(0, 1 - yg(x))$

3 Multiclass Classification - Implementation

In this problem we will work on a simple three-class classification example. The data is generated and plotted for you in the skeleton code.

3.1 One-vs-All (also known as One-vs-Rest)

In this problem we will implement one-vs-all multiclass classification. Our approach will assume we have a binary base classifier that returns a score, and we will predict the class that has the highest score.

1. Complete the class OneVsAllClassifier in the skeleton code. Following the OneVsAllClassifier code is a cell that extracts the results of the fit and plots the decision region. Include these results in your submission.

Answer:

[20]:
```
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_function" that
    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
        """
        #Your code goes here
        y_fit={}
        for i in range(self.n_classes):
            y_fit[i]=(np.where(y==i,1,0))
        for i in range(self.n_classes):
            self.estimators[i].fit(X,y_fit[i])
        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 sklearn SVMs do),
        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.
↪")

        #Replace the following return statement with your code
        score=np.zeros([self.n_classes,X.shape[0]])
        for i in range(self.n_classes):
            score[i]=self.estimators[i].decision_function(X)
        return score.T

    def predict(self, X):
        """
        Predict the class with the highest score.
        @param X: array-like, shape = [n_samples,n_features] input data
```

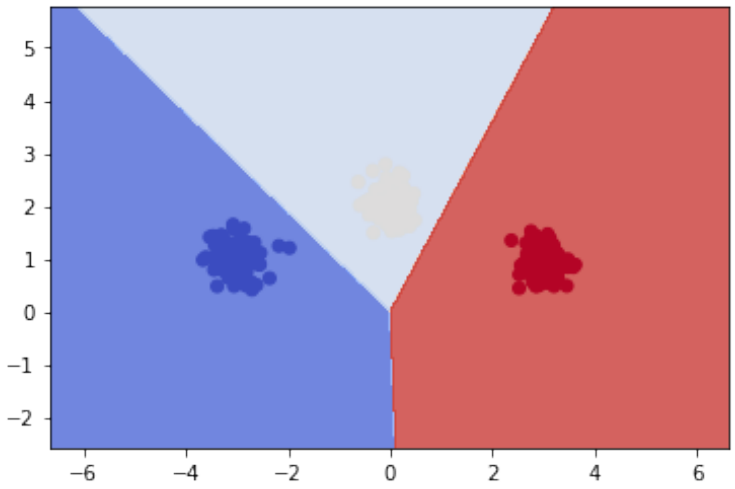
```
        @returns array-like, shape = [n_samples,] the predicted classes for␣
 ↪each input
        """
        #Replace the following return statement with your code
        score=self.decision_function(X)
        y=np.zeros([score.shape[0]])
        for i in range(len(y)):
            y[i]=np.where(score[i]==max(score[i]))[0][0]
        return y
#         return np.zeros(X.shape[0])
```

[15]:
```
#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 yet

# 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))
```

```
/opt/conda/envs/dsga-1003/lib/python3.7/site-packages/sklearn/svm/_base.py:
 ↪947:
ConvergenceWarning: Liblinear failed to converge, increase the number of
iterations.
  "the number of iterations.", ConvergenceWarning)
/opt/conda/envs/dsga-1003/lib/python3.7/site-packages/sklearn/svm/_base.py:
 ↪947:
ConvergenceWarning: Liblinear failed to converge, increase the number of
iterations.
  "the number of iterations.", ConvergenceWarning)

Coeffs 0
[[-1.05852368 -0.90296438]]
Coeffs 1
```

```
[[-0.34082071 -0.14261791]]
Coeffs 2
[[ 0.89079539 -0.82578715]]
```

[15]:
```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```



3.2 Multiclass SVM

In this question, we will implement stochastic subgradient descent for the linear multiclass SVM, as described in lecture and in this problem set. We will use the class-sensitive feature mapping approach with the "multivector construction", as described in our multiclass classification lecture (slide 24).

1. Complete the skeleton code for multiclass SVM. Following the multiclass SVM implementation, we have included another block of test code. Make sure to include the results from these tests in your assignment, along with your code.

Answer:

[59]:
```
import random
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 features for input data
```

```
    @param y: a target class (in range 0,..,num_classes-1)
    @return array-like, shape = [n_samples,n_outFeatures], the class␣
↪sensitive features for class y
    '''
    #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],X.shape[1])
    #your code goes here, and replaces following return
    if len(X.shape)==1:
        X=X[np.newaxis,:]
    if type(y)!=np.ndarray:
        y=np.array([y])
    n_outFeatures=num_classes*num_inFeatures
    out=np.zeros([num_samples,n_outFeatures])
    for i in range(num_samples):
        out[i][y[i]*num_inFeatures:(y[i]+1)*num_inFeatures]=X[i]
    return out

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]
    #your code goes here and replaces following return statement
    w=np.zeros([num_outFeatures])
    for t in range(T):
        idx=np.arange(num_samples)
        random.shuffle(idx)
        X=X[idx]
        y=y[idx]
        w=w-eta*subgd(X[i],y[i],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=featureMap):
        '''
        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 margin)
        @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 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
        '''
        #Your code goes here and replaces the following return statement
        y_max=0
        local_max=self.Delta(y,y_max)+np.dot(w,(self.Psi(x,y_max)-self.
↪Psi(x,y)).T)
        for y_i in range(self.num_classes):
            if local_max < self.Delta(y,y_i)+np.dot(w,(self.Psi(x,y_i)-self.
↪Psi(x,y)).T):
                local_max=self.Delta(y,y_i)+np.dot(w,(self.Psi(x,y_i)-self.
↪Psi(x,y)).T)
                y_max=y_i
        return 2*self.lam*w+self.Psi(x,y_max)-self.Psi(x,y)

    def fit(self,X,y,eta=0.001,T=10000): # 0.1
        '''
        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]
```

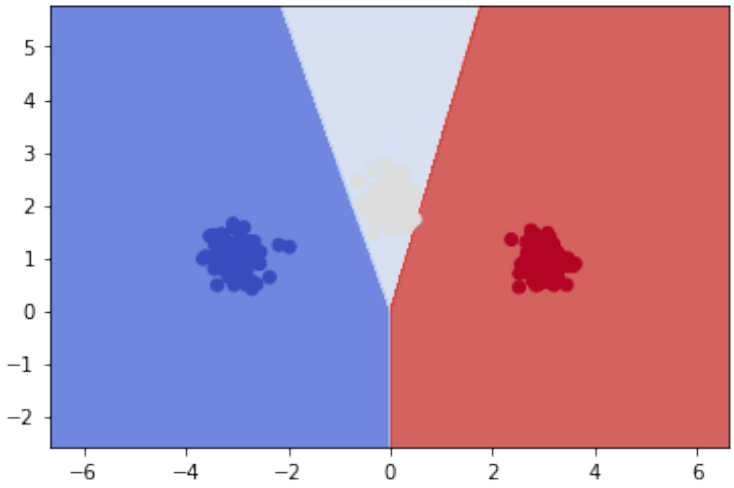
```
        @return array-like, shape = [n_samples, n_classes] giving scores␣
↪for each sample,class pairing
        '''
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting␣
↪data.")

        #Your code goes here and replaces following return statement
        n=X.shape[0]
        score=np.zeros([n,self.num_classes])
        for i in range(n):
            for j in range(self.num_classes):
                score[i][j]=np.dot(self.coef_,self.Psi(X[i],j).T)
        return score

    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 point
        '''

        #Your code goes here and replaces following return statement
        n=X.shape[0]
        score=self.decision_function(X)
        pred=np.zeros(n)
        for i in range(n):
            pred[i]=np.where(score[i]==max(score[i]))[0][0]
        return pred
```

[60]:
```
#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)
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))
```

```
w:
[[-0.33489929 -0.03930003 -0.0200967   0.07715895  0.35499599 -0.03785892]]
```

[60]:
```
array([[100,   0,   0],
       [  0, 100,   0],
```

```
[  0,   0, 100]])
```



4 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 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 manyDecision_Tree objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.

[22]:
```
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 more than min_smaple
        :param max_depth: restriction of tree depth.
```

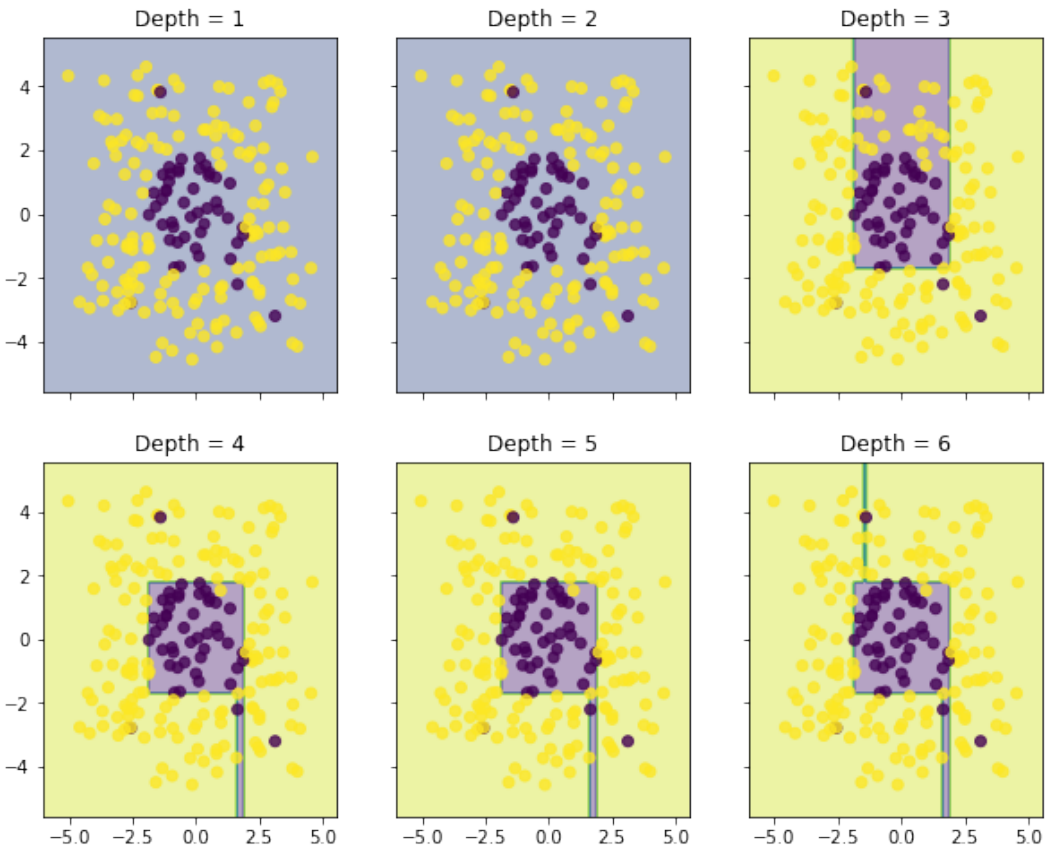
```
        '''
        self.split_loss_function = split_loss_function
        self.leaf_value_estimator = leaf_value_estimator
        self.depth = depth
        self.min_sample = min_sample
        self.max_depth = max_depth

        self.left=None
        self.right=None
        self.value=None
        self.split_value=None
        self.split_id=None
        self.is_leaf=None

    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 we are
        splitting the node, we should also init self.left and self.
↪right to be Decision_Tree
        objects corresponding to the left and right subtrees. These␣
↪subtrees should be fit on
        the data that fall to the left and right,respectively, of self.
↪split_value.
        This is a recurisive tree building procedure.

        :param X: a numpy array of training data, shape = (n, m)
        :param y: a numpy array of labels, shape = (n, 1)

        :return self
        '''

        # Your code goes here
        if self.depth==self.max_depth or len(y)<=self.min_sample:
            self.is_leaf=True
            self.value=self.leaf_value_estimator(y)
            return self
        n_feat=X.shape[1]
        n_samples=X.shape[0]
        X = np.concatenate([X,y],1)
        min_loss=self.split_loss_function(y)
        found=False
        for i in range(n_feat):
            X = np.array(sorted(X,key=lambda x:x[i]))
            for j in range(n_samples):
                left_y = X[:j+1,-1].reshape(-1,1)
                right_y = X[j+1:,-1].reshape(-1,1)
```

```
                l_loss=self.split_loss_function(left_y)*len(left_y)/
↪len(y)
                r_loss=self.split_loss_function(right_y)*len(right_y)/
↪len(y)
                loss=r_loss+l_loss
                if loss < min_loss:
                    found=j
                    min_loss=loss
                    self.split_id=i
                    self.split_value=X[j,i]
        if not found:
            self.is_leaf=True
            self.value=self.leaf_value_estimator(y)
        else:
            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)
            self.left.fit(X[:found+1,:-1],X[:found+1,-1])
            self.right.fit(X[found+1:,:-1],X[found+1:,-1])
        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 instance
        '''
        if self.is_leaf:
            return self.value
        if instance[self.split_id] <= self.split_value:
            #print('split at x[%s]<=%s'%(self.split_id,self.
↪split_value))
            return self.left.predict_instance(instance)
        else:
            return self.right.predict_instance(instance)
```

2. Complete either the compute_entropy or compute_gini functions. 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. For visualization, you'll need to install graphviz.

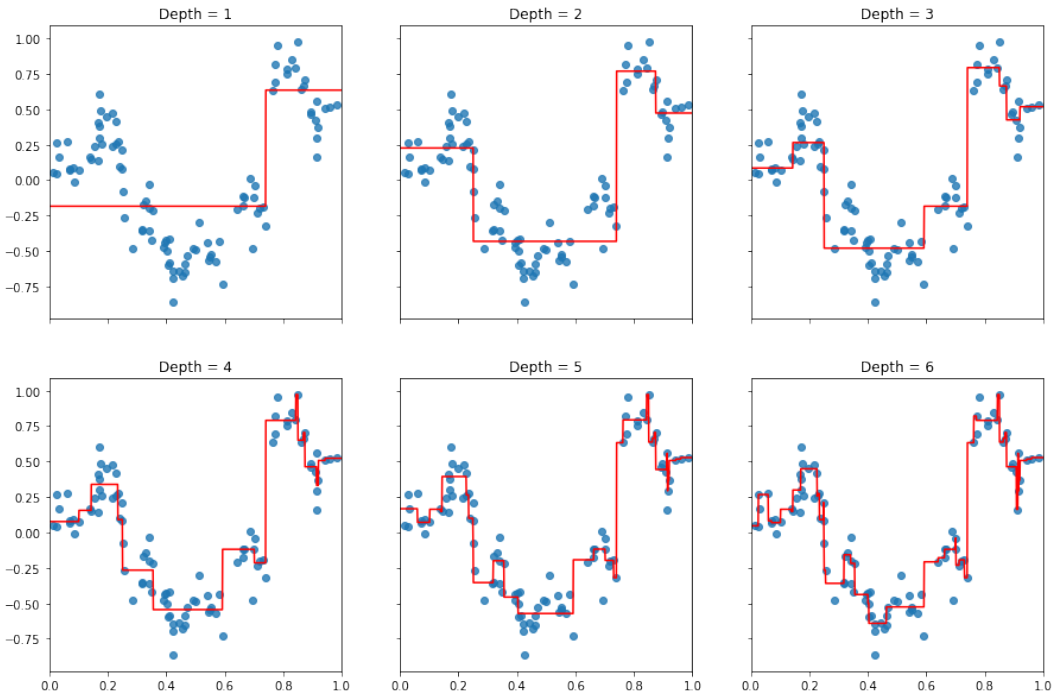
[23]:
```
def compute_entropy(label_array):
    '''
    Calulate the entropy of given label list

    :param label_array: a numpy array of labels shape = (n, 1)
    :return entropy: entropy value
    '''
    n_classes = np.unique(label_array)
```

```
    entropy = 0
    for label in n_classes:
        p = np.sum(label_array==label)/float(len(label_array))
        entropy += -p*np.log(p)
    return entropy

def compute_gini(label_array):
    '''
    Calulate the gini index of label list

    :param label_array: a numpy array of labels shape = (n, 1)
    :return gini: gini index value
    '''
    n_classes = np.unique(label_array)
    gini = 0
    for label in n_classes:
        p = np.sum(label_array==label)/len(label_array)
        gini += p*(1-p)
    return gini
```



3. [Optional] 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.

[33]:
```python
# Regression Tree Specific Code
def mean_absolute_deviation_around_median(y):
    '''
    Calulate the mean absolute deviation around the median of a given target list

    :param y: a numpy array of targets shape = (n, 1)
    :return mae
    '''
    # Your code goes here
    median = np.median(y)
    mae = np.mean(np.abs(y-median))
    return mae
```



5 Gradient Boosting Machines

Recall the general gradient boosting algorithm[1], for a given loss function ℓ and a hypothesis space $\mathcal{F}$ of regression functions (i.e. functions mapping from the input space to $\mathbf{R}$):

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:

[1] Besides the lecture slides, you can find an accessible discussion of this approach in http://www.saedsayad.com/docs/gbm2.pdf, in one of the original references http://statweb.stanford.edu/~jhf/ftp/trebst.pdf, and in this review paper http://web.stanford.edu/~hastie/Papers/buehlmann.pdf.

(a) Compute:

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

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

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

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

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

(d) Take the step:

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

3. Return f_M.

In this problem we'll derive a special case of the general gradient boosting framework: BinomialBoost.

1. 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 $\mathcal{F}$.

Answer:

$$(g_m)_i = \frac{\partial ln(1 + e^{-y_i f_{m-1}(x_i)})}{\partial f_{m-1}(x_i)} = \frac{-y_i e^{-y_i f_{m-1}(x_i)}}{1 + e^{-y_i f_{m-1}(x_i)}}$$

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

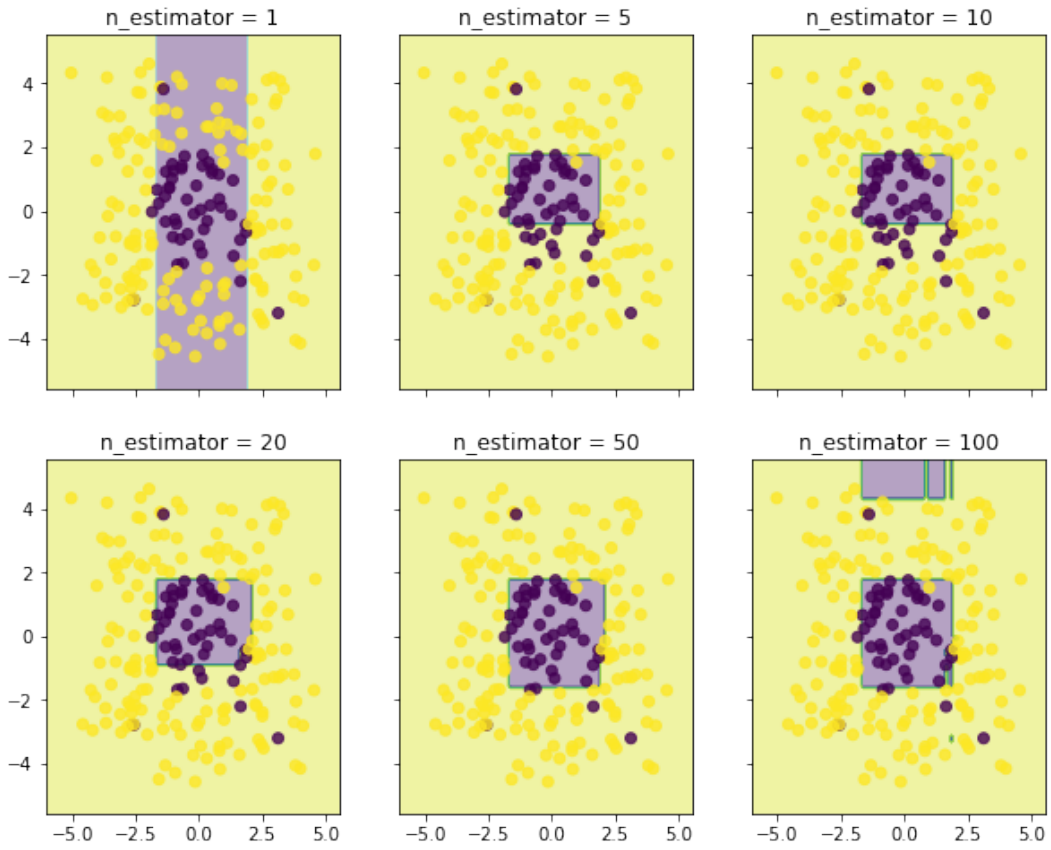
6 Gradient Boosting Implementation

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. Although one of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function, here we'll keep things simple and consider the standard regression setting with square loss.

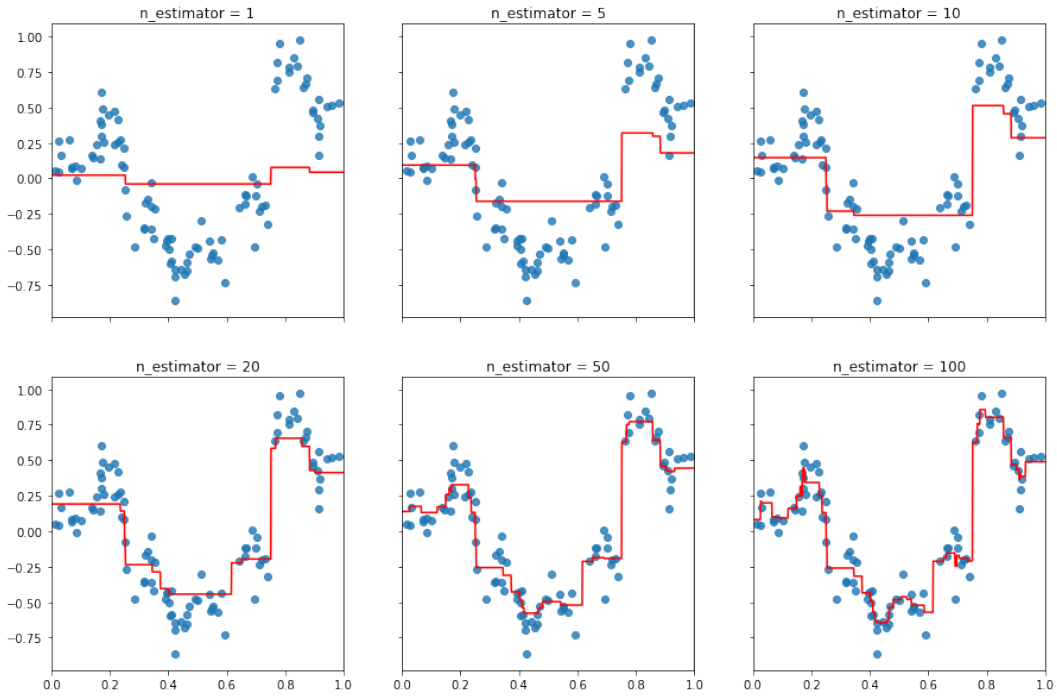
1. Complete the gradient_boosting class. As the base regression algorithm, you may use sklearn's regression tree. You should use the square loss for the tree splitting rule and the mean function for the leaf prediction rule. Run the code provided to build gradient boosting models on the classification and regression data sets, and include the plots generated. Note that we are using square loss to fit the classification data, as well as the regression data.

[37]:
```
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=5, max_depth=3):
        '''
        Initialize gradient boosting class

        :param n_estimator: number of estimators (i.e. number of rounds␣
↪of gradient boosting)
        :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

    def current_estimator(self,x,back=0):
        predicted_y=np.zeros([len(x)])
        for i in range(len(self.estimators)-back):
            predicted_y+=self.learning_rate*self.estimators[i].
↪predict(x)
        predicted_y=predicted_y[:,np.newaxis]
        return predicted_y

    def fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        '''
        # Your code goes here
        self.estimators=[]
        for i in range(self.n_estimator):
            self.estimators.
↪append(DecisionTreeRegressor(min_samples_split=self.
↪min_sample,max_depth=self.max_depth))
            self.estimators[i].fit(train_data,self.
↪pseudo_residual_func(train_target,self.
↪current_estimator(train_data,1)))
        return self

    def predict(self, test_data):
        '''
        Predict value
        '''
        # Your code goes here
        predicted_y=self.current_estimator(test_data,0)
        return predicted_y
```



2. [Optional] Repeat the previous runs on the classification data set, but use a different classification loss, such as logistic loss or hinge loss. Include the new code and plots of your results. Note that you should still use the same regression tree settings for the base regression algorithm.


n_estimator = 1
n_estimator = 5
n_estimator = 10
n_estimator = 20
n_estimator = 50
n_estimator = 100
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