Machine Learning HW6

Ben Zhang, bz957

April 23, 2018

1.2 Two versions of multiclass hinge loss (or generalized hinge loss)

1.2

1) $\because \Delta(y,y)=0$ for all $y \in Y$

$\therefore \max_{y\in Y} [\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i)]$

$= \max[\Delta(y_i,y_i)+h(x_i,y_i)-h(x_i,y_i),\ \max_{y\in Y-\{y_i\}}[\Delta(y_i,y)+h(x_i,y)-h(x_i,y_i)]]$

$= \max[0,\ \max_{y\in Y-\{y_i\}}[\Delta(y_i,y)-m_{i,y}(h)]]$

$= \max_{y\in Y-\{y_i\}} (\max[0, \Delta(y_i,y)-m_{i,y}(h)])$

$\Rightarrow \ell_2(h,(x_i,y_i)) = \ell_1(h,(x_i,y_i))$

2a) $\ell_1(h,(x_i,y_i)) = \max_{y\in Y-\{y_i\}} (\max[0, \Delta(y_i,y)-m_{i,y}(h)])$

$\because m_{i,y}(h) = h(x_i,y_i)-h(x_i,y) \geq \Delta(y_i,y)$

for all $y \in Y-\{y_i\}$, $\Rightarrow \ell_1(h,(x_i,y_i))=0$

if $y=y_i$, $\Delta(y_i,y)=0$, $\ell_1(h,(x_i,y_i))$

$= \max_{y=y_i}(\max[0, 0-(h(x_i,y_i)-h(x_i,y_i))])$

$= 0$

$\because \Delta(y_i,y)=0$ for $y=y_i \Rightarrow \Delta(y,y)=0$ for all $y\in Y$,

from 1st question, $\ell_2(h,(x_i,y_i)) = \ell_1(h,(x_i,y_i)) = 0$

2-b)

$$\mathcal{F} = \{ f(x) = \arg\max_{y \in Y} h(x, y) \mid h \in \mathcal{H} \}$$

$\because$ $h(x_i, y_i) - h(x_i, y) \geq \Delta(y_i, y)$.

for all $y \in Y - \{y_i\}$,

and $\Delta(y_i, y) > 0 \ \forall y \neq y_i$, $\Delta(y_i, y) = 0$ for $y = y_i$,

$\therefore$ $h(x_i, y_i) - h(x_i, y) \geq 0$

$h(x_i, y_i) \geq h(x_i, y)$ for all $y \in Y$

$\therefore$ $\arg\max_{y \in Y} h(x_i, y) = y_i$

$\therefore$ $f(x_i) = \arg\max_{y \in Y} h(x_i, y) = y_i$

2 SGD for Multiclass Linear SVM

2.1 $f_i(w) = \Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle$

is convex,

$\Rightarrow \max_{y \in Y} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$

is convex,

$\because \lambda\|w\|^2$ is convex

$\Rightarrow J(w)$ is convex.

2.2 $\hat{y}_i = \arg\max_{y \in Y} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$

Let $Q = \frac{1}{n}\sum_{i=1}^{n} [\Delta(y_i, \hat{y}_i) + \langle w, \Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)\rangle]$

$\frac{\partial Q}{\partial w} = \frac{1}{n}\sum_{i=1}^{n}[\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)]$.

from HW II we know if f is convex and differentiable, then its gradient at x is a subgradient.

$\because \lambda\|w\|^2$ gradient is $2\lambda w$

$\Rightarrow$ subgradient of $J(w)$ at w is

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

2.3

at point (x_i, y_i), $g = 2\lambda w + [\Phi(x_i, \hat{y}_i) - \Phi(x_i, y_i)]$

2.4 based on the points $(x_i, y_i), \cdots, (x_{i+m-1}, y_{i+m-1})$.

$$g = 2\lambda w + \frac{1}{m}\sum_{i=1}^{m} [\Phi(x_i, \hat{y}_i) - \Phi(x_i, y_i)]$$

3. $\ell(h,(x,y)) = \max_{y' \in Y} [\Delta(y,y') + h(x,y') - h(x,y)]$

let $f(x) = \Delta(y,y') + h(x,y') - h(x,y)$

if $y \neq y'$, $\begin{cases} y'=1, & f(x) = 1 + h(x,1) - h(x,-1) = 1 - g(x) = 1 - yg(x) \\ y'=-1, & f(x) = 1 + h(x,-1) - h(x,1) = 1 + g(x) = 1 - yg(x) \end{cases}$

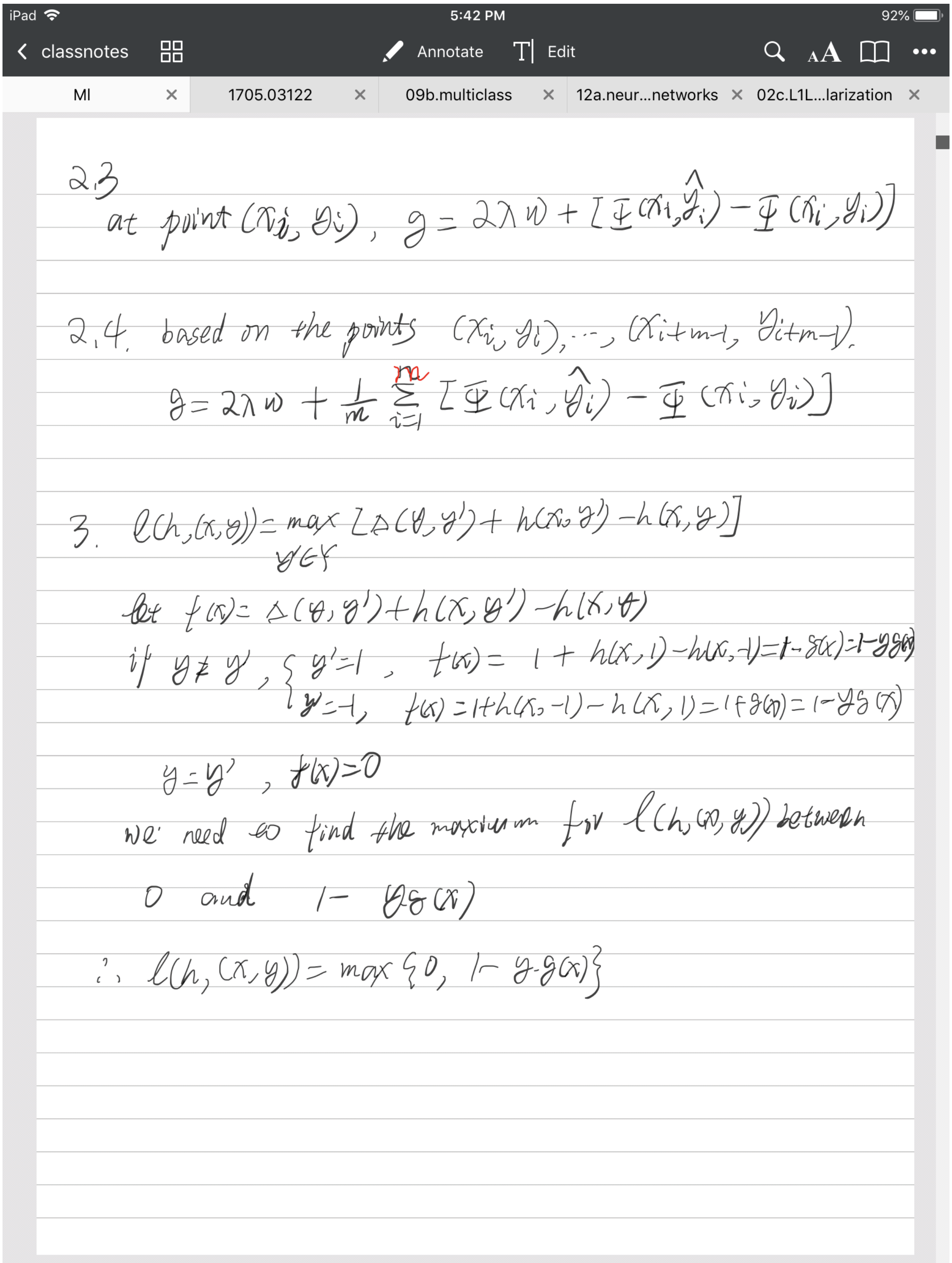
$y = y'$, $f(x) = 0$

we need to find the maximum for $\ell(h,(x,y))$ between

0 and $1 - yg(x)$

$\therefore$ $\ell(h,(x,y)) = \max\{0, 1 - y \cdot g(x)\}$

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



2.3
at point (x_i, y_i), $g = 2\lambda w + [\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)]$

2.4 based on the points $(x_i, y_i), \cdots, (x_{i+m-1}, y_{i+m-1})$.

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

3. $\ell(h,(x,y)) = \max_{y' \in Y} [\Delta(y,y') + h(x,y') - h(x,y)]$

let $f(x) = \Delta(y,y') + h(x,y') - h(x,y)$

if $y' \neq y$, $\begin{cases} y' = 1, & f(x) = 1 + h(x,1) - h(x,-1) = 1 - g(x) = 1 - y g(x) \\ y' = -1, & f(x) = 1 + h(x,-1) - h(x,1) = 1 + g(x) = 1 - y g(x) \end{cases}$

$y = y'$, $f(x) = 0$

we need to find the maximum for $\ell(h,(x,y))$ between 0 and $1 - y g(x)$

$\therefore \ell(h,(x,y)) = \max\{0, 1 - y \cdot g(x)\}$

4 Multiclass Classification - Implementation

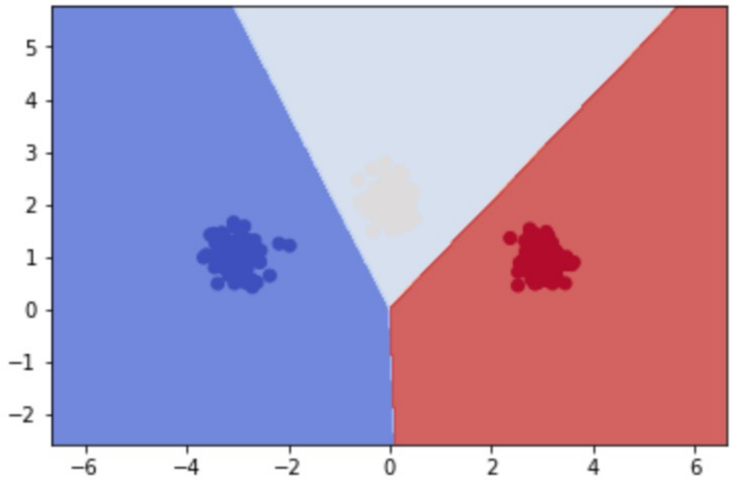
4.1 One-vs-All (also known as One-vs-Rest)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs

# 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)


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
    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
        """
        origin_y = y
        for i in range(self.n_classes):
            y = (y==i)*1
            self.estimators[i].fit(X, y)
            y = origin_y
        #Your code goes here
        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
        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.concatenate([self.estimators[i].decision_function(X).reshape(-1,1)
                               for i in range(self.n_classes)], axis=1)
        #concatenate Y(np.array), Because 3 y have only one axis, as their shape is (
        #and the axis parameter specifically refers to the axis of the elements to co

    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 inpu
        """

        return np.argmax(self.decision_function(X), axis=1)

#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

# 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)
# print(xx.shape,yy.shape,mesh_input.shape,Z.shape)
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
[[-1.05853502 -0.90294878]]
Coeffs 1
[[ 0.2592022  -0.20190078]]
Coeffs 2
[[ 0.89165805 -0.8246809 ]]
```

```
Out[151]: array([[100,   0,   0],
                 [  0, 100,   0],
                 [  0,   0, 100]])
```



4.2 Multiclass SVM

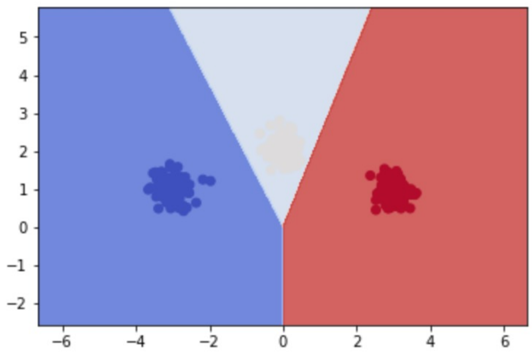
```
def zeroOne(y,a) :
    '''
    Computes the zero-one loss.
    @param y: output class
    @param a: predicted class
    @return 1 if different, 0 if same
    '''
    return int(y != a)

def featureMap(X,y,num_classes) :
    '''
    Computes the class-sensitive features.
    @param X: array-like, shape = [n_samples,n_inFeatures] or [n_inFeatures,], input
    @param y: a target class (in range 0,..,num_classes-1)
    @return array-like, shape = [n_samples,n_outFeatures], the class sensitive featur
    '''
    #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])

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

    #your code goes here, and replaces following return
    X_new[:,y*num_inFeatures:(y+1)*num_inFeatures] = X
    return X_new
    #ndarray:first dimension (shape[0]),here is num_samples;second dimension shape[1]

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((1,num_outFeatures))
    for i in range(T):
        grad = np.zeros((1,num_outFeatures))
        for j in range(num_samples):
            grad += subgd(X[j,:],y[j],w)
        grad = grad/num_samples
        w = w - eta * grad

    #your code goes here and replaces following return statement
    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., targe
        @param Psi: class-sensitive feature map taking two arguments
        '''
        self.num_outFeatures = num_outFeatures #6
        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
        '''
        w = w.reshape(1,-1)
        score = -np.inf
        x_new = self.Psi(x,y)
        y_max = -np.inf
        for i in range(self.num_classes):
            score_new = self.Delta(y,i) + np.dot(w,(self.Psi(x,i)-x_new).T)
```

```
            if score_new >= score:
                y_max = i
                score = score_new

        subgd = 2*self.lam*w + self.Psi(x,y_max) - x_new
        #Your code goes here and replaces the following return statement
        return subgd

    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 sam
        '''
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting data.")

        num_samples = X.shape[0]
        scores = np.zeros((num_samples, self.num_classes))
        for i in range(self.num_classes):
            X_new = self.Psi(X,i)
            scores[:,i] = np.dot(self.coef_,X_new.T)

        #Your code goes here and replaces following return statement
        return scores

    def predict(self, X):
        '''
        Predict the class with the highest score.
        @param X: array-like, shape = [n_samples, n_inFeatures], input data to predic
        @return array-like, shape = [n_samples,], class labels predicted for each data
        '''
        #Your code goes here and replaces following return statement
        return np.argmax(self.decision_function(X), axis=1)

#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:
[[-0.3137493  -0.05275403 -0.02428185  0.10212202  0.33803115 -0.04936798]]

Out[319]: array([[100,   0,   0],
                 [  0, 100,   0],
                 [  0,   0, 100]])
```



7 Gradient Boosting Machines

7.1 at m'th round, we have $f_{m-1}(x)$.

$$\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2 \Rightarrow g_m = \frac{\partial}{\partial f_{m-1}(x)}\left(\frac{1}{2}(f_{m-1}(x) - y)^2\right)$$

$$= f_{m-1}(x) - y$$

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

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

7.2 $\ell(m) = \ln(1+e^{-m}) = \ln(1+e^{-yf(x)})$

$$g_m = \frac{\partial}{\partial f_{m-1}(x)} \ln(1+e^{-yf_{m-1}(x)})$$

$$= -\frac{y}{1+e^{yf_{m-1}(x)}}$$

$$h_m = \arg\min \sum_{i=1}^{n} \left[(-g_m)_i - h(x_i)\right]^2$$

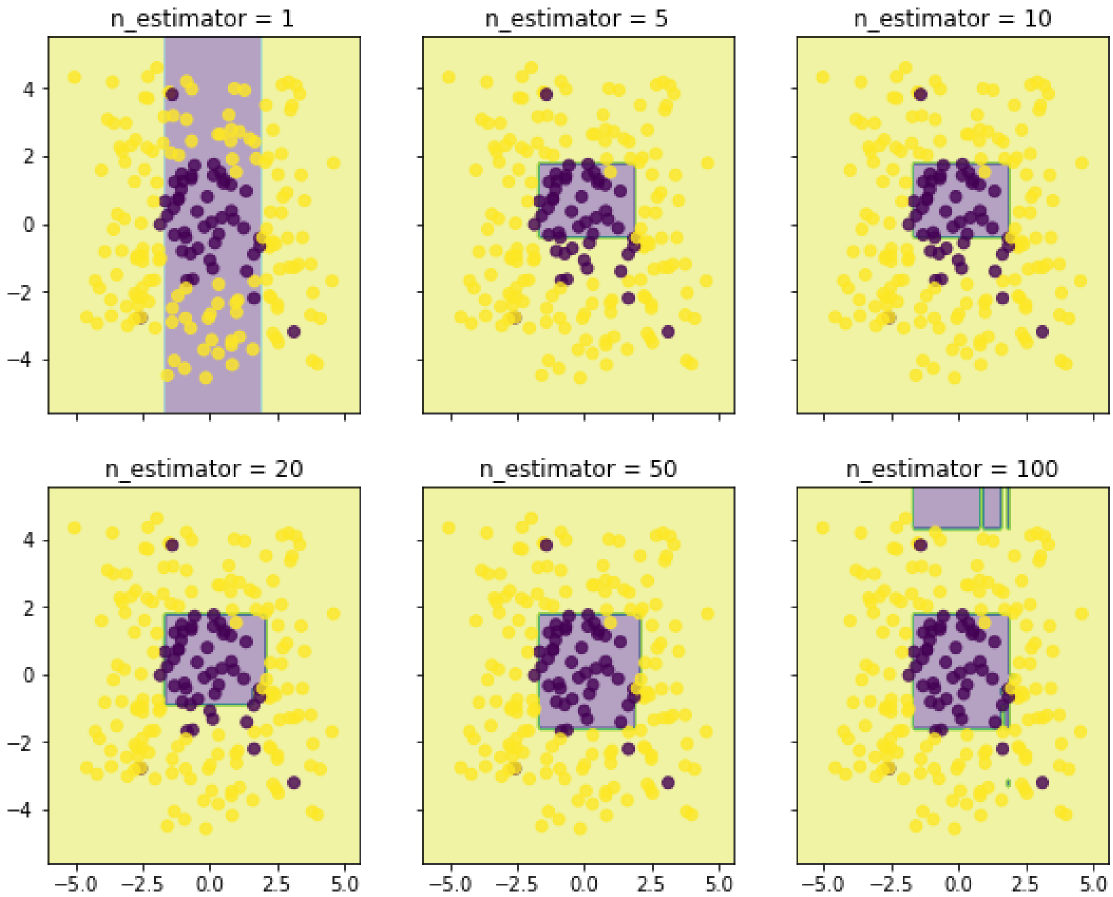
$$= \arg\min \sum_{i=1}^{n} \left[\frac{y_i}{1+e^{y_i f_{m-1}(x_i)}} - h(x_i)\right]^2$$

8 Gradient Boosting Implementation

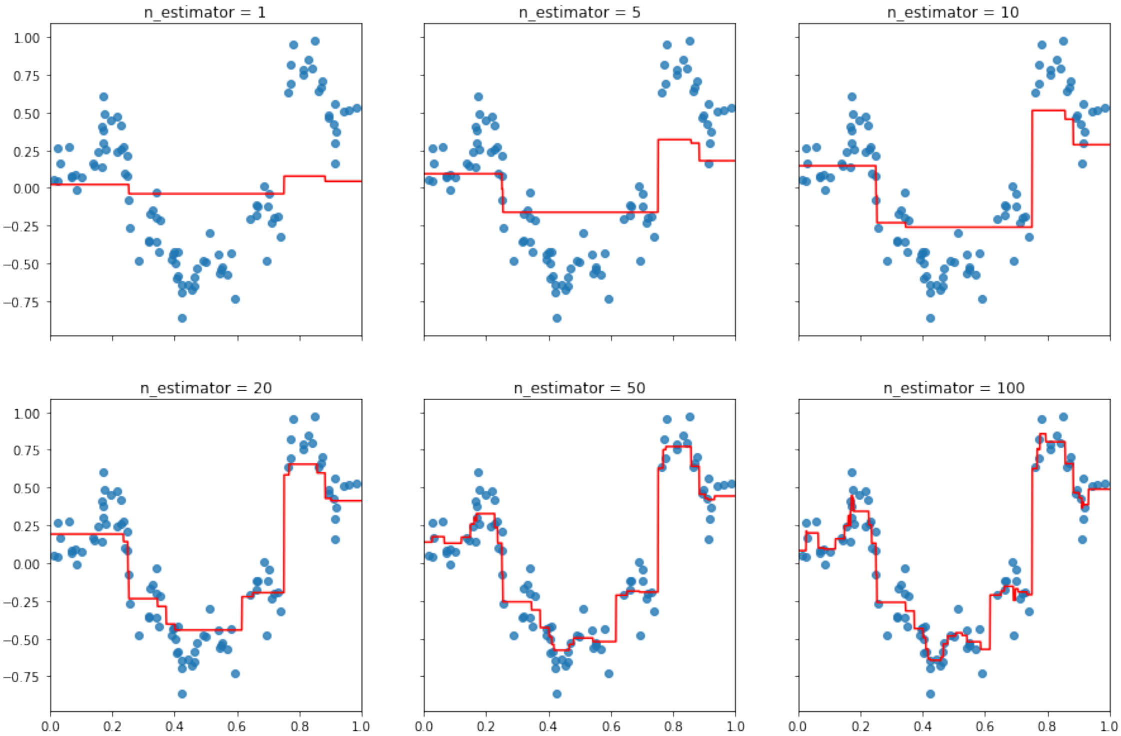
8.1

```
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,
                         export_graphviz
import graphviz

from IPython.display import Image

%matplotlib inline
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)
y_train_label = np.array(list(map(lambda x: 1 if x > 0 else 0, y_train)))


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 b
        :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 fit(self, train_data, train_target):
        '''
        Fit gradient boosting model
        '''
        self.gb=[]

        f = 0
        for i in range(self.n_estimator):
            psedo_r = self.pseudo_residual_func(train_target, f)
            Regressor = DecisionTreeRegressor(max_depth=self.max_depth,
                        min_samples_split=self.min_sample)
            h = Regressor.fit(train_data, psedo_r)
            f = f + self.learning_rate*h.predict(train_data).reshape(-1,1)
            self.gb.append(h)
```

```
        return self

    def predict(self, test_data):
        '''
        Predict value
        '''

        f_test = 0
        for i in range(self.n_estimator):
            h_predict = self.gb[i].predict(test_data)
            f_test = f_test + self.learning_rate*h_predict.reshape(-1,1)
#                print(f_test)
        return f_test


# 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, m
    gbt.fit(x_train, y_train)

    Z = np.sign(gbt.predict(np.c_[xx.ravel(), yy.ravel()]))
#     print(np.any(Z==0))
    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, alpha
    axarr[idx[0], idx[1]].set_title(tt)
```

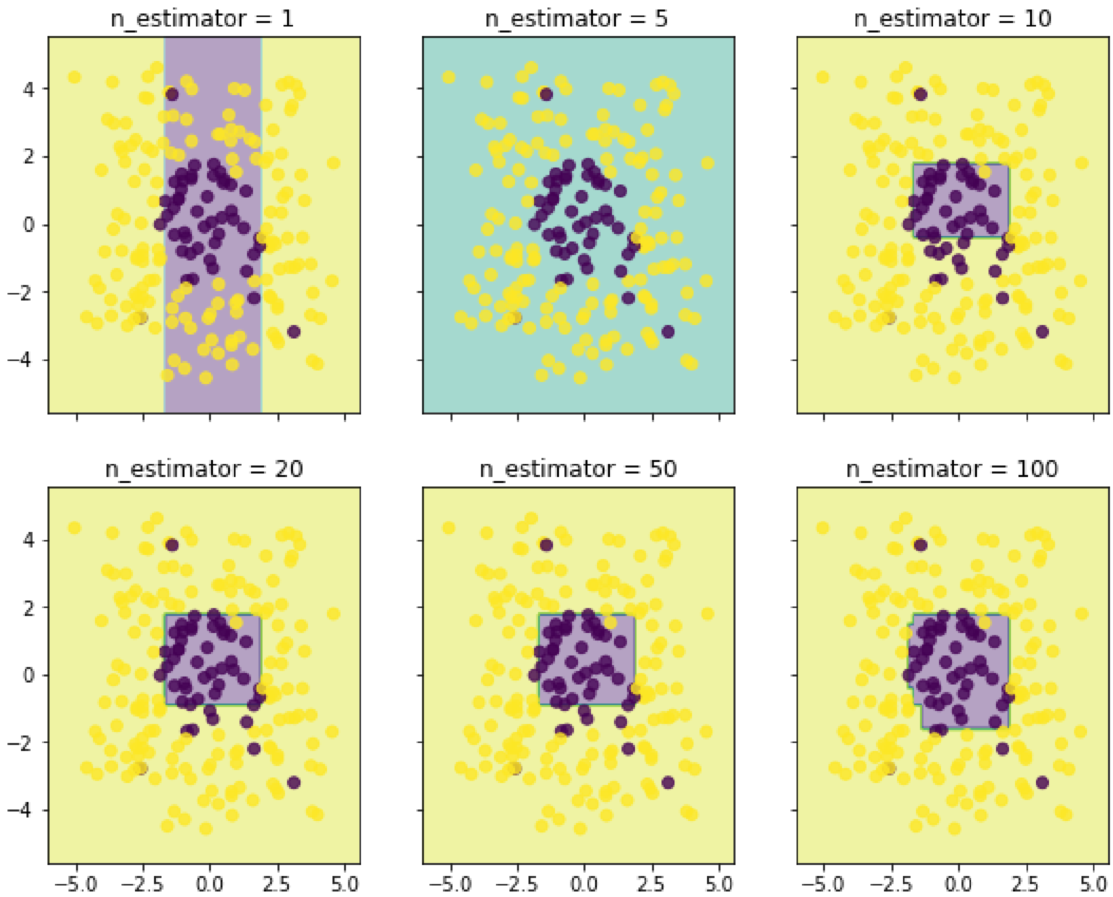

n_estimator = 1
n_estimator = 5
n_estimator = 10
n_estimator = 20
n_estimator = 50
n_estimator = 100
4
2
0
−2
−4
−5.0
−2.5
0.0
2.5
5.0

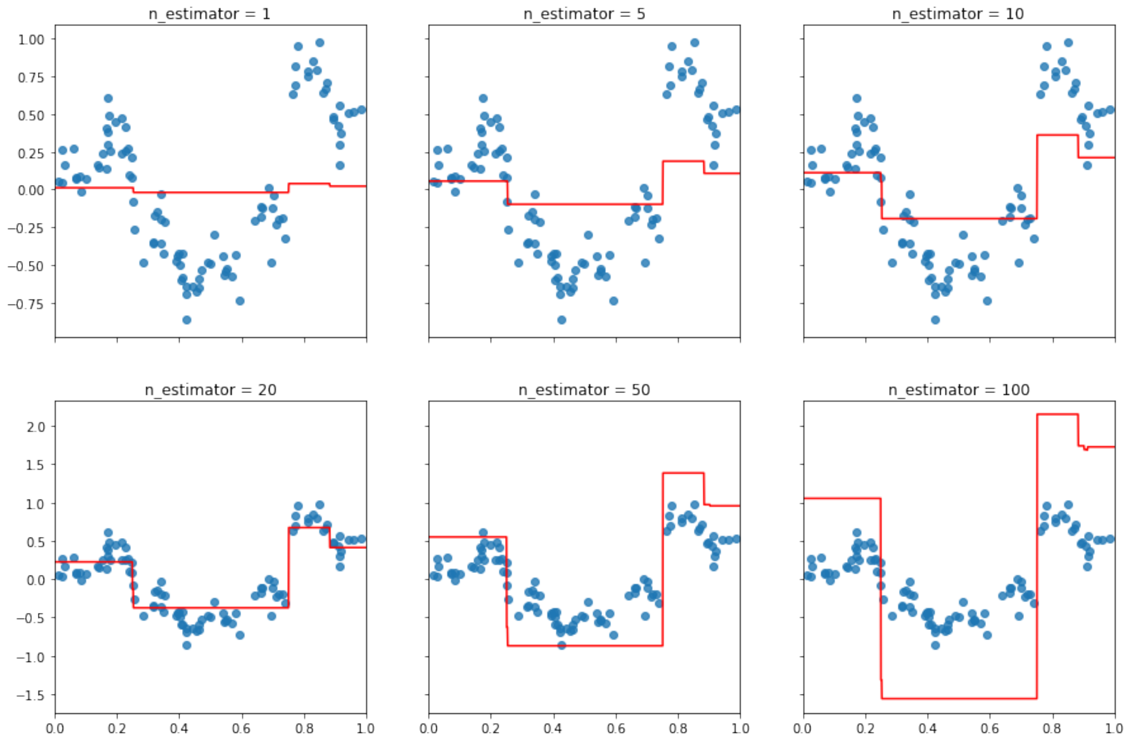
```python
plot_size = 0.001
x_range = np.arange(0., 1., plot_size).reshape(-1, 1)

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)
```



8.2

```python
def pseudo_residual_Logistic(train_target, train_predict):
    '''
    Compute the pseudo-residual based on current predicted value.
    '''
    margin = train_target*train_predict
    loss = train_target/(1+np.exp(margin))
    return loss
```


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


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