Homework 5

Name: Zheyuan Hu

NetID: zh2095

Derivation

Q1

The Object function is given by
$J(w) = \lambda\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\max_{y\in\mathbf{y}}\left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle\right]$

Since $\Delta\left(y_i, y\right)$ is a constant of w and $\langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle$ is a inner product of w, $\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle$ is convex.

According to the Convex Optimization note 3.2.4, their pointwise maximum $\max_{y\in\mathbf{y}}\left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle\right]$ is also convex.

Since the sum of convex functions is also convex,
$\sum_{i=1}^{n}\max_{y\in\mathbf{y}}\left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle\right]$ is convex

Then from note 3.1.3, we know that every norm is convex, so $\lambda\|w\|^2$ is convex.

Therefore, $J(w)$ is a convex function of w.

Q2

Define

$$\hat{y}_i = argmax_{y\in\mathbf{y}}\left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle\right]$$

Then $J(w)$ can be written as

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

Since $J(w)$ is convex and differentiable, the subgradient of $J(w)$ is its gradient,

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

Q3

An expression for the stochastic subgradient based on the point (x_i, y_i):

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

Q4

An expression for a minibatch subgradient, based on the points
$(xi, yi), \ldots, (xi + m - 1, yi + m - 1)$:

$$2\lambda w + \frac{1}{m}\sum_{j=i}^{i+m-1}[\Psi(x_j, \hat{y}_j) - \Psi(x_j, y_j)]$$

Implementation

Q5

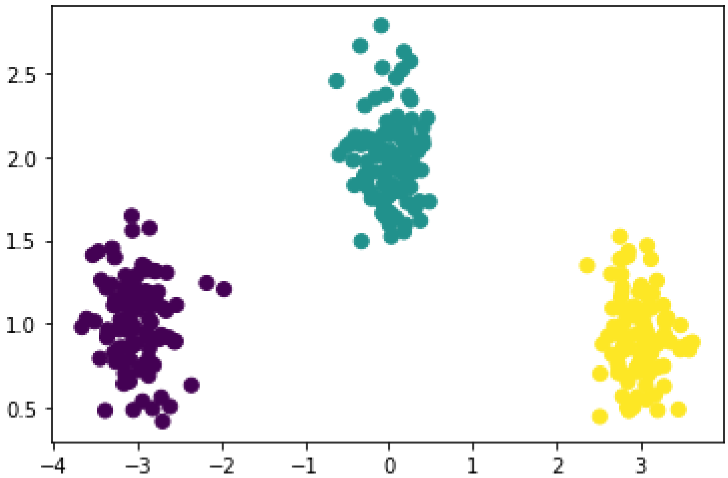
In [6]:

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

%matplotlib inline
```

In [7]:

```
# 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[7]: <matplotlib.collections.PathCollection at 0x288d18c2130>



In [8]:

```
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 "decisior
    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
        for i in range(self.n_classes):
            y_fit = np.array(y==i).astype(int)
            self.estimators[i].fit(X, y_fit)
        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 s
        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([X.shape[0], self.n_classes])
        for i in range(self.n_classes):
            score[:,i] = self.estimators[i].decision_function(X)
        return score

    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_pred = []
        for i in range(X.shape[0]):
            y_pred.append(np.argmax(score[i,:]))
        return np.array(y_pred)
```

Q6

I modified the regularization weight c=100 to get a better seperating hyperplane

In [9]:

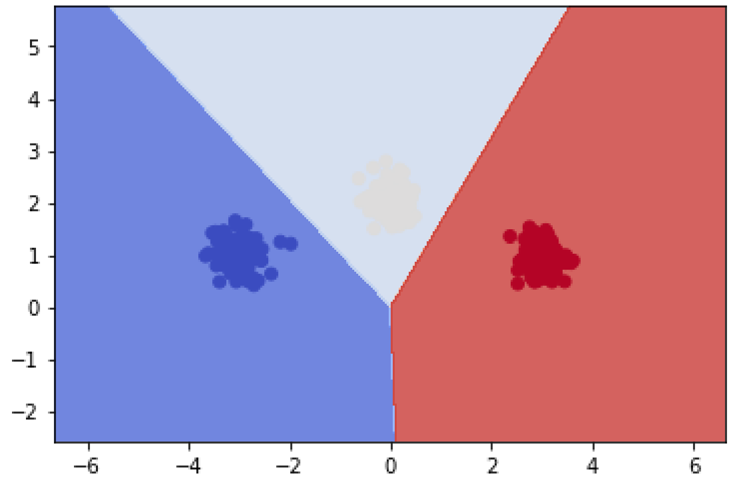
```
#Here we test the OneVsAllClassifier
from sklearn import svm
svm_estimator = svm.LinearSVC(loss='hinge', fit_intercept=False, C=100)
clf_onevsall = OneVsAllClassifier(svm_estimator, n_classes=3)
clf_onevsall.fit(X,y)

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)
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))
```

```
C:\Users\52673\anaconda3\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWarnin
g: Liblinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
Coeffs 0
[[-1.05853334 -0.90294603]]
Coeffs 1
[[-0.25046466 -0.12232678]]
Coeffs 2
[[ 0.89164752 -0.82601734]]
```

Out[9]:

```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]], dtype=int64)
```



Q7

In [63]:

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

Q8

In [64]:

```
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,w 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([1,num_outFeatures])
    for i in range(T):
        k = np.random.randint(num_samples)
        w -= eta * subgd(X[k], y[k],w)
    return w
```

Q9

In [65]:

```
class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Psi=fe
        '''
        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 subgradient(self,x,y,w):
        '''
```

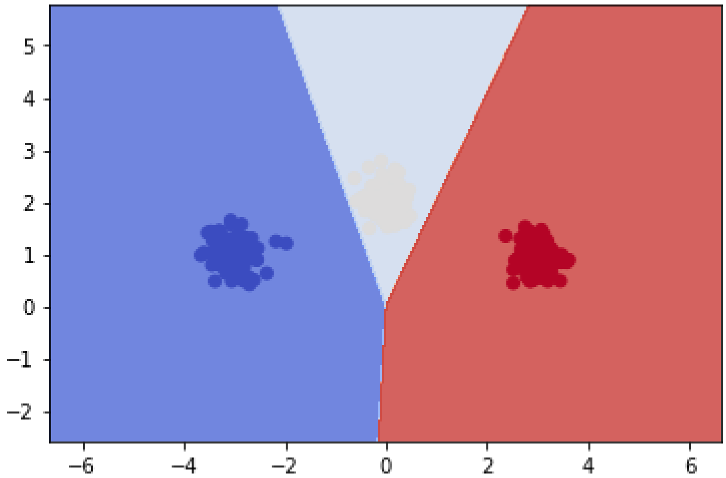
```
        Computes the subgradient at a given data point x,y
        @param x: sample input
        @param y: sample class
        @param w: parameter vector
        @return returns subgradient vector at given x,y,w
        '''
        #Your code goes here and replaces the following return statement
        loss = 0
        for yi in range(self.num_classes):
            a = self.Delta(yi, y) + np.dot(w, (self.Psi(x, y)-self.Psi(x, yi)).T)
            if a > loss:
                loss = a
                y_hat = yi
        subgrad = 2*self.lam*w + self.Psi(x, y_hat) - self.Psi(x, y)
        return subgrad

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

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

        #Your code goes here and replaces following return statement
        score = self.decision_function(X)
        y_pred = []
        for i in range(X.shape[0]):
            y_pred.append(np.argmax(score[i,:]))
        return np.array(y_pred)
```

Q10

In [66]:
```
#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,eta=0.1)
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:
[[-1.11076154 -0.09503514  0.00667478  0.31704149  1.10408676 -0.22200634]]
```

Out[66]:
```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]], dtype=int64)
```



In [ ]: