Solution to Question 1: Show that $J(w)$ is a convex function of w, where $J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max_{y \in Y} \left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$

- Since $\max_{y \in Y} \left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$ is a convex function of w, the sum of it (i.e. $\sum_{i=1}^{n} \max_{y \in Y} \left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$) is also a convex function of w.
- Since $\|w\|$ is a convex function of w, $\|w\|^2$ and $\lambda \|w\|^2$ are both convex functions of w.
- By summation of convex functions is still convex, thereupon, it's proved that $J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max_{y \in Y} \left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$ is a convex function of w.

Solution to Question 2: Give an expression for a subgradient of $J(w)$.

- In this setting, $J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max_{y \in Y} \left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$
- Given that $\hat{y}_i = \arg\max_{y \in Y} \left[\Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \right]$
- we have $J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \left[\Delta\left(y_i, \hat{y}_i\right) + \langle w, \Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i) \rangle \right]$
- Thus the subgradient of $J(w)$ is $\bigtriangledown J(w) = 2\lambda w + \frac{1}{n} \sum_{i=1}^{n} \left[\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i) \right]$

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

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

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

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

Solution to the (optional) Question: show that for the choice of h mentioned, the multiclass hinge loss reduces to hinge loss: $\ell\left(h, (x, y)\right) = \max_{y' \in Y} \left[\Delta\left(y, y'\right) + h(x, y') - h(x, y) \right] = \max\left\{0, 1 - yg(x)\right\}$

- When $y = y'$, we have $\Delta(y, \hat{y}) = 0$
$$l(h,(x,y)) = \max_{y' \in Y} \left[0 + h(x,y') - h(x,y)\right] = 0$$
- When $y \neq y'$, with y=1 & y'=-1 , we have $\Delta(y, \hat{y}) = 1$
$$\begin{aligned} l(h,(x,y)) &= \max_{y' \in Y} \left[\Delta\left(y,y'\right) + h(x,y') - h(x,y)\right] \\ &= \max_{y' \in Y}[1 + h(x,-1) - h(x,1)] \\ &= \max_{y' \in Y}(1 - \frac{g(x)}{2} - \frac{g(x)}{2}) \\ &= \max\{0, 1 - yg(x)\} \end{aligned}$$
- When $y \neq y'$, with y=-1 & y'=1, we have $\Delta(y, \hat{y}) = 1$
$$\begin{aligned} l(h,(x,y)) &= \max_{y' \in Y} \left[\Delta\left(y,y'\right) + h(x,y') - h(x,y)\right] \\ &= \max_{y' \in Y}[1 + h(x,1) - h(x,-1)] \\ &= \max_{y' \in Y}(1 + \frac{g(x)}{2} + \frac{g(x)}{2}) \\ &= \max\{0, 1 - yg(x)\} \end{aligned}$$

To sum up, it is shown that for this choice of h, we have
$$\ell(h,(x,y)) = \max_{y' \in Y} \left[\Delta\left(y,y'\right) + h(x,y') - h(x,y)\right] = \max\left\{0, 1 - yg(x)\right\}$$

In [1]:

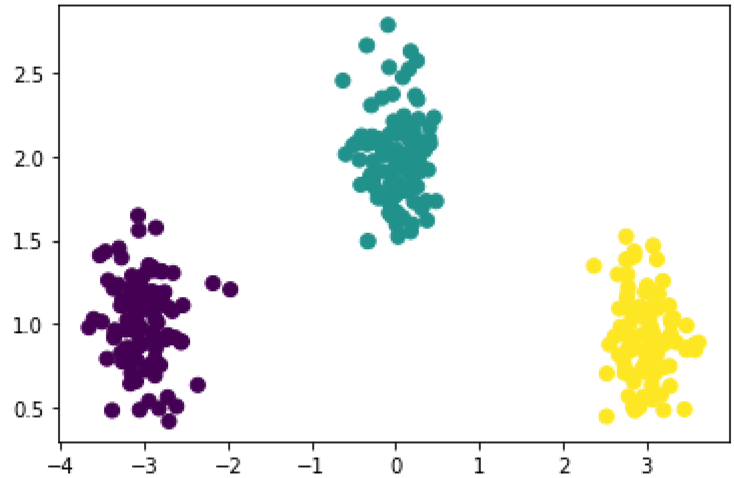
```
import numpy as np
import matplotlib.pyplot as plt
# try:
#     from sklearn.datasets.samples_generator import make_blobs
# except:
from sklearn.datasets import make_blobs

%matplotlib inline
```

In [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 0x7fa95573fe80>
```



One VS All

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 "de
cision_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
        for i in range(self.n_classes):
            y_fit = np.zeros(y.shape[0])
            y_fit[y==i] = 1
            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 (w
hich 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 dat
a.")

        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)
        return np.argmax(score, axis=1)
```

In [4]:

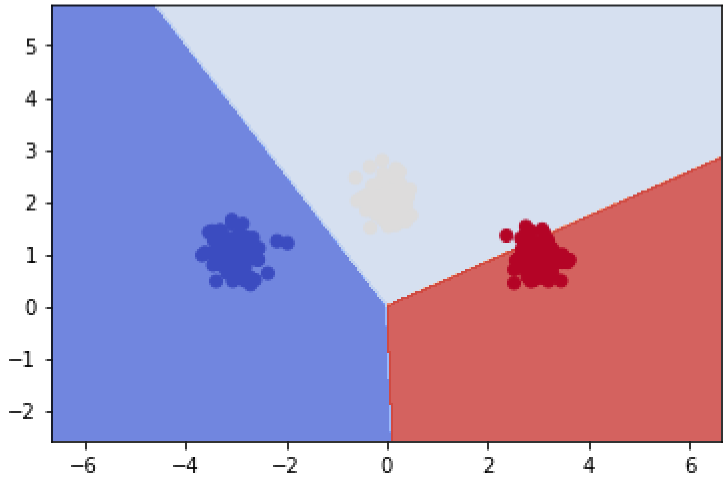
```
#Here we test the OneVsAllClassifier
from sklearn import svm
svm_estimator = svm.LinearSVC(loss='hinge', fit_intercept=False, C=200)
clf_onevsall = OneVsAllClassifier(svm_estimator, n_classes=3)
clf_onevsall.fit(X,y)

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

```
Coeffs 0
[[-1.05853334 -0.90294603]]
Coeffs 1
[[0.42121645 0.27171776]]
Coeffs 2
[[ 0.89164752 -0.82601734]]
```

```
/opt/anaconda3/lib/python3.8/site-packages/sklearn/svm/_base.py:976:
ConvergenceWarning: Liblinear failed to converge, increase the numbe
r of iterations.
  warnings.warn("Liblinear failed to converge, increase "
```

Out[4]:

```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,  11,  89]])
```



Multiclass SVM

In [5]:

```
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,], i
nput 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 f
eatures 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.s
hape[0], X.shape[1])

    #your code goes here, and replaces following return
    if len(X.shape) == 1:
        X = X[np.newaxis, :]
    X_construct = np.zeros((num_samples, num_classes*num_inFeatures))
    X_construct[:, y*num_inFeatures:(y+1)*num_inFeatures] = X

    return X_construct

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_i = np.zeros(num_outFeatures).reshape(1, 6)
    w = np.zeros(num_outFeatures).reshape(1, 6)
    for i in range(T):
        index = np.random.randint(num_samples)
        w_i -= eta * subgd(X[index], y[index], w_i)
        w += w_i

    return w/T

class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, P
si=featureMap):
```

```
        '''
        Creates a MulticlassSVM estimator.
        @param num_outFeatures: number of class-sensitive features produced by P
si
        @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
        margin = [self.Delta(y, y_dash) + w@np.ravel(self.Psi(x, y_dash)-self.Ps
i(x, y)) for y_dash in range(self.num_classes)]
        y_hat = np.argmax(margin)
        subgd = 2*self.lam*w + self.Psi(x, y_hat) - self.Psi(x, y)

        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 eac
h sample,class pairing
        '''
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting dat
a.")

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

```
        num_samples = X.shape[0]
        score = np.zeros((num_samples, self.num_classes))
        for i in range(num_samples):
            score[i, :] = [self.coef_ @ self.Psi(X[i], y_j).T for y_j in range(s
elf.num_classes)]

        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 p
redict
        @return array-like, shape = [n_samples,], class labels predicted for eac
h data point
        '''

        #Your code goes here and replaces following return statement
        score = self.decision_function(X)

        return np.argmax(score, axis=1)
```

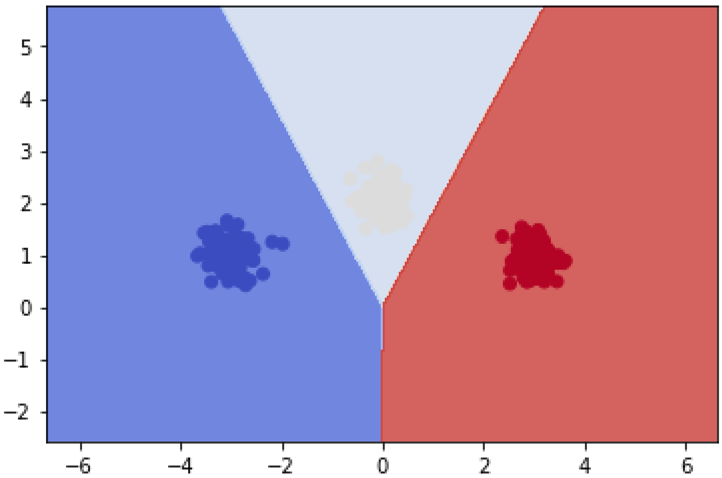
In [6]:

```
#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:
[[-0.29422556 -0.05284238 -0.00136812  0.10902631  0.29559368 -0.056
18393]]
```

Out[6]:

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



In [ ]: