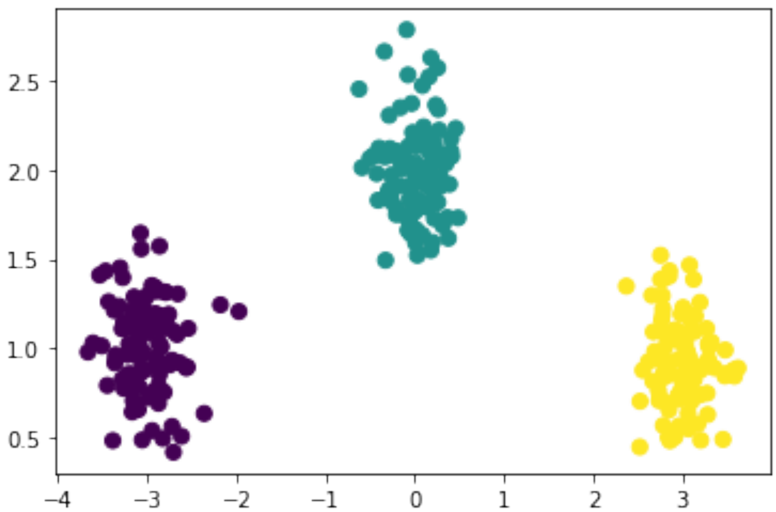
In [39]:
```
import numpy as np
import matplotlib.pyplot as plt
try:
    from sklearn.datasets.samples_generator import make_blobs
except:
    from sklearn.datasets import make_blobs

%matplotlib inline
```

In [40]:
```
# Create the  training data
np.random.seed(2)
X, y = make_blobs(n_samples=300,cluster_std=.25, centers=np.array([(-3,1),(0,
plt.scatter(X[:, 0], X[:, 1], c=y, s=50)
```

Out[40]: <matplotlib.collections.PathCollection at 0x7ff37117db20>



One VS All

Problem 11

In [49]:
```
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
    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
        """
        #My Code
        #Create helper dictionary
        class_dict = {}
        #for each class, create a np array with length n, that takes value 1
        for class_n in range(self.n_classes):
            class_dict[class_n] = np.where(y==class_n,1,0)
        #Iterate through our classes and evaluate SVM
        for class_n in range(self.n_classes):
            self.estimators[class_n].fit(X,class_dict[class_n])
        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
        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 da

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

        #Initialize a matrix size rows*classes
        score_matrix = np.zeros([X.shape[0],self.n_classes])

        #For each col (representing a unique class) estimate class viaw decis
        for i in range(self.n_classes):
            score_matrix[:,i] = self.estimators[i].decision_function(X)

        return score_matrix
```

```
    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 e
        """
        #Calculate 1 vs all Scores
        score = self.decision_function(X)
        #Return the class that satisfies the arg max
        highest_score_class = np.argmax(score, axis=1)
        return highest_score_class
```

Problem 12

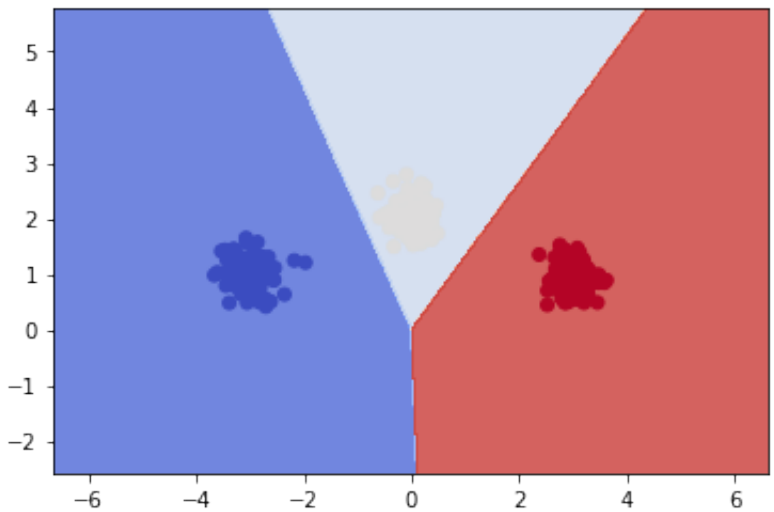
In [82]:

```
#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 impleme

# 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
[[-1.05852753 -0.90296523]]
Coeffs 1
[[ 0.21690689 -0.31543985]]
Coeffs 2
[[ 0.89106893 -0.82499987]]
```

```
/opt/anaconda3/lib/python3.9/site-packages/sklearn/svm/_base.py:1206: Converge
nceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
/opt/anaconda3/lib/python3.9/site-packages/sklearn/svm/_base.py:1206: Converge
nceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

Out[82]:
```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```



Multiclass SVM

Problem 13

In [67]:
```
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,]
    @param y: a target class (in range 0,..,num_classes-1)
    @return array-like, shape = [n_samples,n_outFeatures], the class sensitiv
    '''

    if len(X.shape) == 1:
        X = X.reshape((1,2))

    #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

    #My Code
    #We will need Num Feature Map Cols = Num Features * Num Classes
    n_outFeatures = num_inFeatures*num_classes
    #Initalize np.array of zeros
    feature_matrix = np.zeros([num_samples, n_outFeatures])
    #Enforce that y is np.array
    if type(y) != np.ndarray:
            y = np.array([y])

    #Iterate over the number of rows
    for row in range(num_samples):
        #Calculate Start/Stop index for each class
        start = y[row] * num_inFeatures
        stop = start + num_inFeatures
        #Update feature_matrix
        feature_matrix[row,start:stop] = X[row,:]
    #Return featureMap
    return feature_matrix
```

Problem 14

In [68]:
```
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]

    #My Code
    #Initialize w vector
    w = np.zeros(num_outFeatures)
    #Loop over for however many iterations
    for i in range(T):
        #Select an index at random
        index = np.random.choice(np.arange(num_samples))
        #Take gradient step with the shuffled index
        w = w - eta*subgd(X[index,:],y[index],w)
    #Return weight vector
    return w
```

Problem 15

In [71]:
```
class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne
        '''
        Creates a MulticlassSVM estimator.
        @param num_outFeatures: number of class-sensitive features produced b
        @param lam: l2 regularization parameter
        @param num_classes: number of classes (assumed numbered 0,..,num_clas
        @param Delta: class-sensitive loss function taking two arguments (i.e
        @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
        '''
        #My Code

        #Initiliaze two variables, our first guess at a class, and the result
        class_guess = 0
        max_margin_guess = self.Delta(y,class_guess) + (w@self.Psi(x,class_gu

        #Loop over all classes
        for class_number in range(self.num_classes):
            #Calculate Margin for Class = Class_Number
            class_margin = self.Delta(y,class_number) + (w@self.Psi(x,class_n
            #Check if we've improved our margin
            if class_margin > max_margin_guess:
                max_margin_guess = class_margin
                class_guess = class_number
        #Once we've tried all the classes, return the highest margin one
        subgradient = (2*self.lam*w) + self.Psi(x,class_guess) - self.Psi(x,y
        #Return gradient
        return subgradient

    def fit(self,X,y,eta=0.1,T=10000):
        '''
        Fits multiclass SVM
        @param X: array-like, shape = [num_samples,num_inFeatures], input dat
        @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 
        '''
        if not self.fitted:
            raise RuntimeError("You must train classifer before predicting da

        #My code
        #Initialize matrix of 0s
        score_matrix = np.zeros([X.shape[0],self.num_classes])
        #Iterate over each class for each row and predict
        for row in range(X.shape[0]):
            for class_col in range(self.num_classes):
```

```
                #Update score_matrix for the scores in each class
                m = self.Psi(X[row],class_col).T
                score_matrix[row][class_col] =  np.dot(self.coef_,m)
        #Return matrix
        return score_matrix

    def predict(self, X):
        '''
        Predict the class with the highest score.
        @param X: array-like, shape = [n_samples, n_inFeatures], input data t
        @return array-like, shape = [n_samples,], class labels predicted for 
        '''
        #My Code
        #Calculate the score for each class
        class_scores = self.decision_function(X)
        #Get the largest scores for each row
        max_scores = np.argmax(class_scores,axis=1)
        #Return max scores
        return max_scores
```

Problem 16

In [ ]:

```
#### Code as Given
```

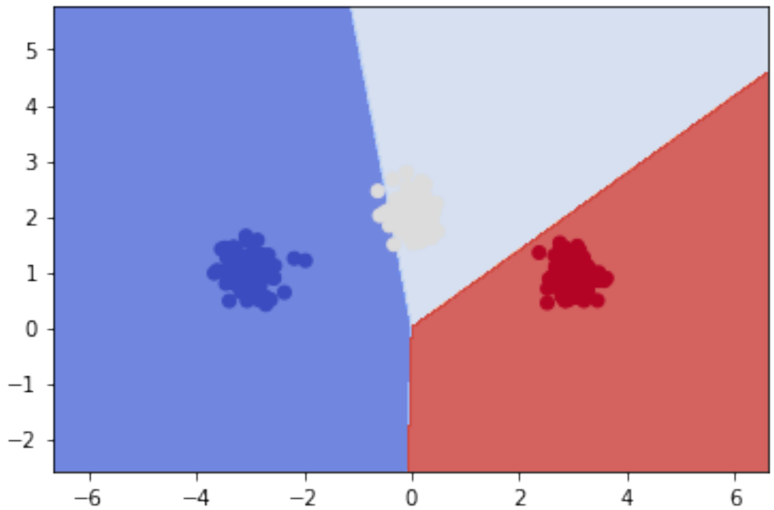
In [76]:

```
#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:
[[-0.36339978 -0.02792297  0.1423984   0.07055951  0.22100137 -0.04263654]]
```

Out[76]:

```
array([[100,   0,   0],
       [  7,  93,   0],
       [  0,   0, 100]])
```



After testing different step sizes for η, we find that $\eta = .001$ works great

In [78]:

```
#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=0.001)
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)
metrics.confusion_matrix(y, est.predict(X))
```

```
w:
[[-0.34154083 -0.03142942  0.0041913   0.08156499  0.33734953 -0.05013557]]
```

Out[78]:

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