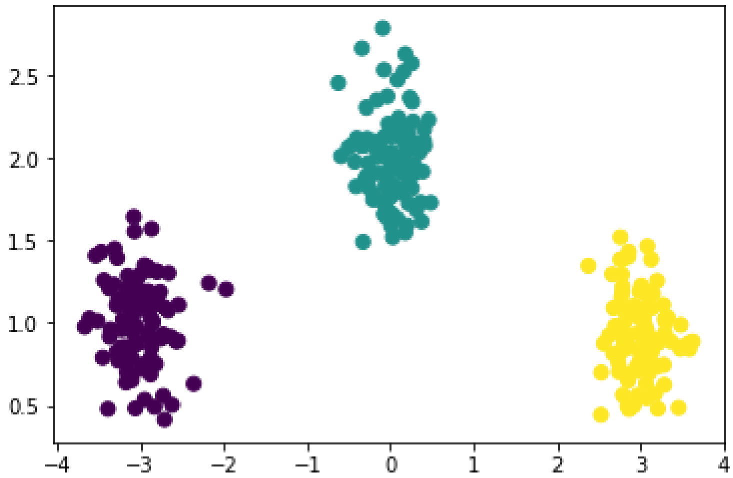
In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs

%matplotlib inline
```

In [13]:
```
# 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),(
plt.scatter(X[:, 0], X[:, 1], c=y, s=50)
```

Out[13]:
```
<matplotlib.collections.PathCollection at 0x1a1c149208>
```



4.1 one-vs-all

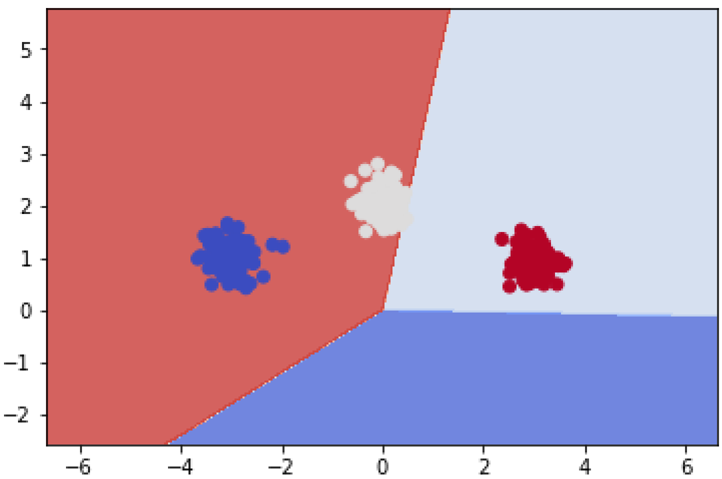
In [14]:
```python
from sklearn.base import BaseEstimator, ClassifierMixin, clone

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 "dec
    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 yi, estimator in enumerate(self.estimators):
            #Create binary labels for each y
            label = np.zeros(len(y))
            label[y == i] = 1
            #Fit binary classification
            estimator.fit(X,label)
        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 (wh
        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
        n_samples = X.shape[0]
        #initialize score
        score = np.zeros((n_samples,self.n_classes))
        for idx, estimator in enumerate(self.estimators):
            score[:,idx] = estimator.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
        """
        #Replace the following return statement with your code

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

In [22]:
```
#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

# 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))
```

```
/Users/jr/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:922: Conver
genceWarning: Liblinear failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/jr/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:922: Conver
genceWarning: Liblinear failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/jr/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:922: Conver
genceWarning: Liblinear failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)

Coeffs 0
[[ 0.89134624 -0.82548752]]
Coeffs 1
[[ 0.89136243 -0.82461249]]
Coeffs 2
[[ 0.89073021 -0.82446619]]
```

Out[22]:
```
array([[  0,   0, 100],
       [  0,   1,  99],
       [  0, 100,   0]])
```



4.2 Multiclass SVM

In [24]:
```
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,], in
    @param y: a target class (in range 0,..,num_classes-1)
    @return array-like, shape = [n_samples,n_outFeatures], the class sensitive fe
    '''
    #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.sha
    #your code goes here, and replaces following return
    num_outFeatures = num_classes * num_inFeatures
    output_X = np.zeros(num_samples*num_outFeatures).reshape(num_samples,num_outF
    if num_samples == 1:
        feature_mapped = np.zeros(num_outFeatures)
        feature_mapped[y*num_inFeatures:(y+1)*num_inFeatures] = X
        return feature_mapped
    for idx, sample in enumerate(X):
        yi = y[idx]
        feature_mapped = np.zeros(num_outFeatures)
        feature_mapped[yi*num_inFeatures:(yi+1)*num_inFeatures] = sample
        output_X[idx,:] = feature_mapped
    return output_X

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)
    avg_w = np.zeros(num_outFeatures)
    for t in range (T):
        idx = np.random.randint(num_samples)
        x_sample = X[idx]
        y_sample = y[idx]
        sgd = subgd(x_sample, y_sample, w)
        w -= eta*sgd
        avg_w += w
    return avg_w/T

class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Ps
        '''
        Creates a MulticlassSVM estimator.
        @param num_outFeatures: number of class-sensitive features produced by Ps
```

```
        @param lam: l2 regularization parameter
        @param num_classes: number of classes (assumed numbered 0,..,num_classes-
        @param Delta: class-sensitive loss function taking two arguments (i.e., t
        @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

        h = [self.Delta(y,y_prime)+w.dot(self.Psi(x,y_prime))-w.dot(self.Psi(x,y)
               for y_prime in range(self.num_classes)]
        yhat = np.argmax(h)
        return 2*self.lam*w.T+self.Psi(x,yhat)-self.Psi(x,y)

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

        #Your code goes here and replaces following return statement
        hxy = np.zeros(len(X)*self.num_classes).reshape(len(X),self.num_classes)
        for i,xi in enumerate(X):
            hxy[i,:] = [self.coef_.dot(self.Psi(xi,yi)) for yi in range(self.num_
        return hxy

    def predict(self, X):
        '''
        Predict the class with the highest score.
        @param X: array-like, shape = [n_samples, n_inFeatures], input data to pr
        @return array-like, shape = [n_samples,], class labels predicted for each
        '''

        #Your code goes here and replaces following return statement
        def getmaxpos(arr1d):
```

```
            return np.where(arr1d==max(arr1d))[0][0]
        decision_mat = self.decision_function(X)
        return np.apply_along_axis(arr=decision_mat,axis=1,func1d=getmaxpos)
```

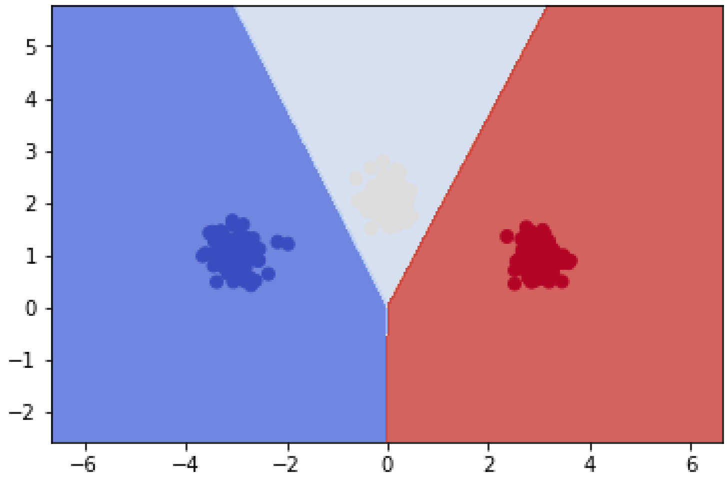
In [25]:

```
#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:
[-0.29978099 -0.05096267  0.00147252  0.10720836  0.29830847 -0.05624569]
```

Out[25]:

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



In [ ]: