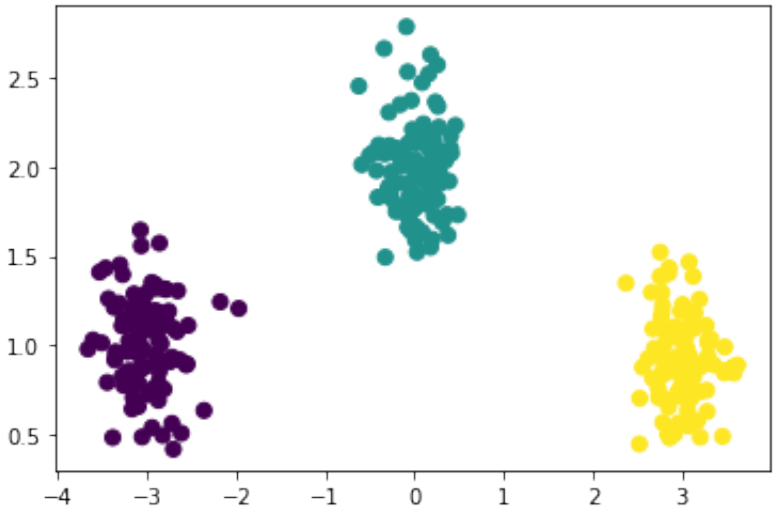
In [39]:
```
import numpy as np
import matplotlib.pyplot as plt
try:
    from sklearn.datasets.samples_generator import make_blobs
except:
    from sklearn.datasets import make_blobs

%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

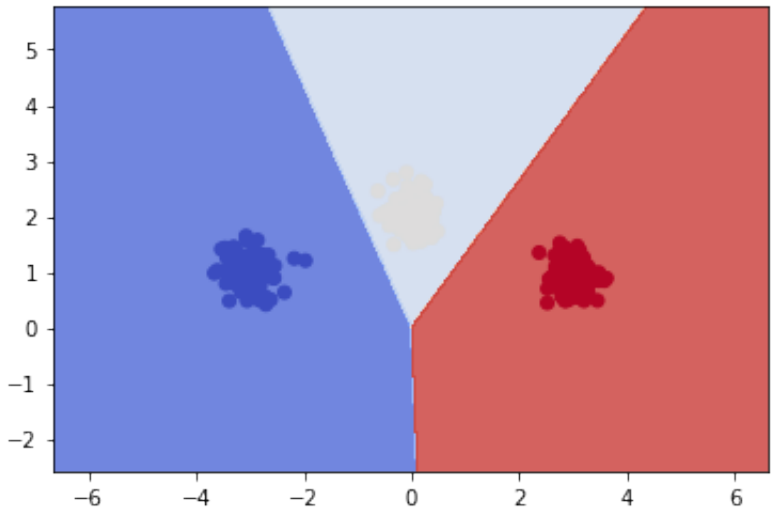
In [82]:

```
#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

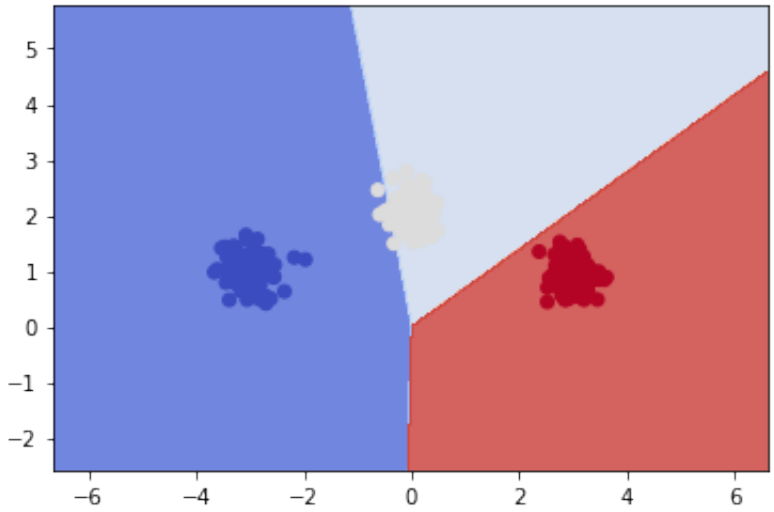
In [ ]:

```
#### 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))
```

Out[76]:
```
w:
[[-0.36339978 -0.02792297  0.1423984   0.07055951  0.22100137 -0.04263654]]
array([[100,   0,   0],
       [  7,  93,   0],
       [  0,   0, 100]])
```



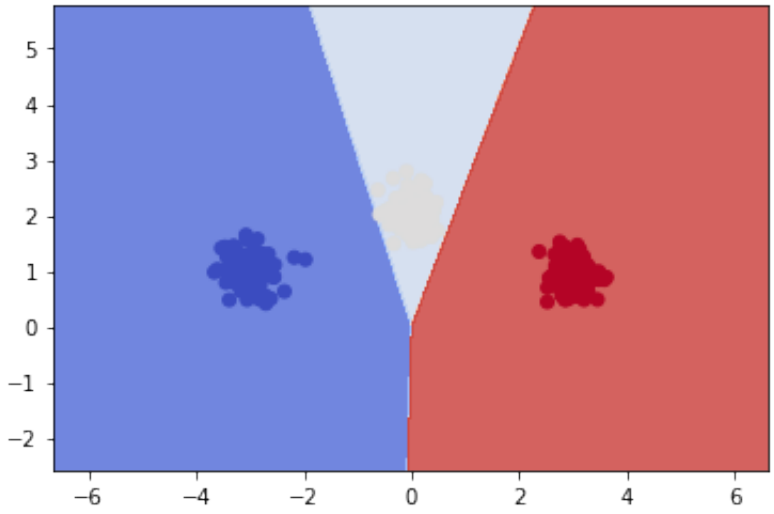
After testing different step sizes for η, we find that $\eta = .001$ works great

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]])
```



Homework 5: SGD for Multiclass Linear SVM

Due: Friday, April 8, 2022 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

---

1 Bayesian Modeling

Bayesian Logistic Regression with Gaussian Priors

This question analyzes logistic regression in the Bayesian setting, where we introduce a prior $p(w)$ on $w \in \mathbb{R}^d$. Consider a binary classification setting with input space $\mathcal{X} = \mathbb{R}^d$, outcome space $\mathcal{Y}_\pm = \{-1, 1\}$, and a dataset $\mathcal{D} = \big((x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\big)$.

1. Give an expression for the posterior density $p(w \mid \mathcal{D})$ in terms of the negative log-likelihood function $\text{NLL}_\mathcal{D}(\text{w})$ and the prior density $p(w)$ (up to a proportionality constant is fine).

   A Bayesian model is defined by two pieces: a parametric family of densities, $\{p(D|w \mid w \in \Theta\}$,and a prior distribution, $P(w)$. Putting the two pieces together we get the following joint distribution:

   $$P(D, w) = P(D|w)P(w)$$

   Using Bayes rule, we can write the posterior distribution as:

   $$p(w|D) = \frac{p(D|w)P(w)}{p(D)}$$

   Since we don't know $p(D)$, but understand that it is a probability in $[0, 1]$, we write the identity using a proportionality constant:

   $$p(w|D) \propto P(D|w)P(w)$$

   From now on, we'll denote this relationship by applying a proportionality constant, $k = \frac{1}{P(D)}$, to the RHS. We can express $P(D|w)$ using the Likelihood Estimation methods we explored in the last homework. Since we're asked to express this probability in the form on NLL, we'll need to adjust it like so:

   $$P(w|D) = k \times e^{-NLL_D(w)} \times P(w)$$

2. Suppose we take a prior on w of the form $w \sim \mathcal{N}(0, \Sigma)$, that is in the Gaussian family. Is this a conjugate prior to the likelihood given by logistic regression?

   The Gaussian family is not a prior to our Logistic Regression classification task, as the conditional probability $P(D|w)$ takes the form of a Bernoulli distribution:

$$P(D|w) = w^{n+} \times (1-w)^{n-}$$

Where $n+$ signifies positive class predictions and $n-$ signifies negative class predictions. Multiplying the likelihood and the prior together does not result in another Gaussian distribution, and thus the resulting new posterior will not be Gaussian, negating any conclusion we could make that the Gaussian family is a conjugate prior to Logistic Regression.

A Gaussian family being a conjugate prior only applies to **Linear Regression**, as we have shown that the MLE methods we explored for a Gaussian family are equivalent to solving the least squares objective function.

3. Show that there exist a covariance matrix Σ such that MAP (maximum a posteriori) estimate for w after observing data $\mathcal{D}$ is the same as the minimizer of the regularized regression function defined in Regularized Logistic Regression paragraph above, and give its value. [Hint: Consider minimizing the negative log posterior of w. Also, remember you can drop any terms from the objective function that don't depend on w. You may freely use results of previous problems.]

Firstly lets consider our Gaussian prior, which we define as

$$P(w) = \frac{\exp -\frac{1}{2} w^T \Sigma^{-1} w}{\sqrt{(2\pi)^d det(\Sigma)}}$$

We'll start solving by using the identity we found in problem 1, remove the proportionality constant, k, as it is a constant, substitute in the definition of a Gaussian distribution then minimize the negative log posterior of w:

$$\begin{aligned} P(w|D) &= k \times e^{-NLL_D(w)} \times P(w) \\ -Log(P(w|D)) &= -log(e^{-NLL_D(w)}) \times -log(P(w)) \\ &= -log(e^{-NLL_D(w)}) + -log(\frac{\exp -\frac{1}{2} w^T \Sigma^{-1} w}{\sqrt{(2\pi)^d det(\Sigma)}}) \\ &= NLL_D(w) + \frac{1}{2} w^T \Sigma^{-1} w - \frac{d}{2}(2\pi det(\Sigma)) \end{aligned} \tag{1}$$

We can now remove the constant term, $\frac{d}{2}(2\pi det(\Sigma))$, substitute in our identity for NLL, and take the argmin with respect to w:

$$argmin_w \left(-\log(P(w|D))\right) = W_{map} = \sum_{i=1}^{n} log(1 + e^{-y_i w^T x_i}) + \frac{1}{2} w^T \Sigma^{-1} w \tag{2}$$

Comparing this to our Empirical Risk Minimizer for Regularized (L2 Norm) Logistic Regression we find they are incredibly similar:

$$\text{Regularized Logistic Regression ERM} = \frac{1}{n} \sum_{i=1}^{n} log(1 + e^{-y_i w^T x_i}) + \lambda ||w||^2$$

The two expressions are indeed equivalent, as we can arbitrarily divide our W_{map} expression by $1/n$ and define $\Sigma = I \times \frac{1}{2n\lambda}$, without changing the w value that our minimization returns.

$$\begin{aligned} argmin_w(-log(P(w|D))) = W_{map} &= \frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i w^T x_i}) + \frac{1}{2} w^T \Sigma^{-1} w \\ &= \frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i w^T x_i}) + \frac{1}{2} w^T \Sigma^{-1} w \qquad (3) \\ &= \frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i w^T x_i}) + \lambda ||w||^2 \quad \square \end{aligned}$$

4. In the Bayesian approach, the prior should reflect your beliefs about the parameters before seeing the data and, in particular, should be independent on the eventual size of your dataset. Imagine choosing a prior distribution $w \sim \mathcal{N}(0, I)$. For a dataset $\mathcal{D}$ of size n, how should you choose λ in our regularized logistic regression objective function so that the ERM is equal to the mode of the posterior distribution of w (i.e. is equal to the MAP estimator).

   We would simply chose $\lambda = \frac{1}{2n}$ which when we evaluate our derived W_{map} estimator, the $\frac{1}{2} w^T \Sigma^{-1} w$ term would become:

   $$\frac{1}{2n} w^T \Sigma^{-1} w = \frac{1}{2n} w^T (Id_n * 2n) w = w^T w = ||w||^2$$

   And again, we would have equivalence between our ERM model and our Bayesian MLE W_{MAP} estimate.

Coin Flipping with Partial Observability

This is continuing your analysis done in HW4, you may use the results you obtained in HW4

Consider flipping a biased coin where $p(z = H \mid \theta_1) = \theta_1$. However, we cannot directly observe the result z. Instead, someone reports the result to us, which we denotey by x. Further, there is a chance that the result is reported incorrectly *if it's a head*. Specifically, we have $p(x = H \mid z = H, \theta_2) = \theta_2$ and $p(x = T \mid z = T) = 1$.

5. We additionally obtained a set of clean results $\mathcal{D}_c$ of size N_c, where x is directly observed without the reporter in the middle. Given that there are c_h heads and c_t tails, estimate θ_1 and θ_2 by MLE taking the two data sets into account. Note that the likelihood is $L(\theta_1, \theta_2) = p(\mathcal{D}_r, \mathcal{D}_c \mid \theta_1, \theta_2)$.

   We set up the problem by defining n_h as the number of heads and n_t as the number of tails from D_r.

   Since D_r is conditionally independent from D_c and θ_2, we have:

   $$P(D_c|\theta_1, \theta_2, D_r) = P(D_c|\theta_1)$$

Using the definition of MLE :

$$\begin{aligned} L(\theta_1, \theta_2) &= p(\mathcal{D}_r, \mathcal{D}_c \mid \theta_1, \theta_2) \\ &= P(\mathcal{D}_r|\theta_1, \theta_2)P(D_c|\theta_1) \\ &= (\theta_1\theta_2)^{n_h} * (1-\theta_1\theta_2)^{n_t} * (\theta_1)^{c_h} * (1-\theta_1)^{c_t} \end{aligned} \tag{4}$$

We can take the log-likelihood and take partial derivatives, and set to 0 to solve for the MLE getting us:

$$\begin{aligned} \frac{\partial Log(L(\theta_1, \theta_2)}{\partial \theta_1} &= \frac{c_h}{\theta_1} + \frac{c_t}{1-c_t} + \frac{n_h}{\theta_1} - \frac{n_t\theta_2}{1-\theta_2\theta_1} \\ \frac{\partial Log(L(\theta_1, \theta_2)}{\partial \theta_2} &= \frac{n_h}{\theta_2} - \frac{n_t\theta_1}{1-\theta_2\theta_1} \end{aligned} \tag{5}$$

When we set the partial derivative with respect to θ_2 to 0 we have:

$$\frac{n_h}{\theta_2} = \frac{n_t\theta_1}{1-\theta_2\theta_1}$$

Multiplying both sides by θ_2/θ_1:

$$\frac{n_h}{\theta_1} = \frac{n_t\theta_2}{1-\theta_2\theta_1}$$

And:

$$\frac{n_h}{\theta_1} - \frac{n_t\theta_2}{1-\theta_2\theta_1} = 0$$

Using what we found we plug this back in and simplify:

$$\begin{aligned} \frac{\partial Log(L(\theta_1, \theta_2))}{\partial \theta_1} = 0 &= \frac{c_h}{\theta_1} - \frac{c_t}{1-c_t} \\ \theta_1 MLE &= \frac{c_h}{c_h + c_t} \end{aligned} \tag{6}$$

Plugging θ_{1MLE} the expression of the partial derivative of MLE with respect to θ_2:

$$0 = \frac{n_h}{\theta_2} - \frac{n_t\theta_{1MLE}}{1-\theta_{1MLE}\theta_2}$$

Solving for θ_2 we get, we can factor out the θ_{1MLE} in the denominator and simplify to get:

$$\begin{aligned} \theta_2 MLE &= \frac{n_h}{n_h\theta_{1MLE} + n_t\theta_{1MLE}} \\ \theta_2 MLE &= \frac{n_h}{(\theta_{1MLE}(n_h + n_t))} \\ \theta_2 MLE &= \frac{n_h}{(n_h + n_t)} * \frac{1}{\theta_{1MLE}} \\ \theta_2 MLE &= \frac{n_h}{(n_h + n_t)} * \frac{c_h + c_t}{c_h} \end{aligned} \tag{7}$$

This implies that the MLE estimate we receive for θ_2 is the proportion of "dirty" observed heads divided by the proportion of "clean" heads.

6. Since the clean results are expensive, we only have a small number of those and we are worried that we may overfit the data. To mitigate overfitting we can use a prior distribution on θ_1 if available. Let's imagine that an oracle gave use the prior $p(\theta_1) = \text{Beta}(h, t)$. Derive the MAP estimates for θ_1 and θ_2.

We can derive our MAP estimate by writing out our Bayesian MLE Statement:

$$\begin{aligned} P(\theta_1, \theta_2 | D_r, D_c) &\propto P(D_r|\theta_1, \theta_2) P(D_c|\theta_1) \\ &\propto \theta_1^{c_h} * (1-\theta_1)^{c_t} * (\theta_1\theta_2)^{n_h} * (1-\theta_1\theta_2)^{n_t} * [\theta_1^{h-1} * (1-\theta_1)^{t-1}] * \alpha \end{aligned} \tag{8}$$

Where α is a proportionality constant representing the prior on θ_2. **For sake of notation, we set $\alpha = 1$ as we derive our MAP estimator**.

Taking the log of our above expression we get:

$$Log(P(\theta_1, \theta_2) \propto c_h log(\theta_1) + c_t log(1-\theta_1) + n_h log(\theta_1\theta_2) + n_t log(1-\theta_1\theta 2) + (h-1) log(\theta_1) + (t-1) log(1-\theta_1)$$

And now we take the partial derivative w.r.t both θ_1 and θ_2 we get:

$$\begin{aligned} \frac{\partial Log(P(\theta_1, \theta_2))}{\partial \theta_1} &= \frac{c_h}{\theta_1} - \frac{c_t}{1-\theta_1} + \frac{n_h}{\theta_1} - \frac{n_t\theta_2}{1-\theta_2\theta_1} + \frac{h-1}{\theta_1} - \frac{t-1}{1-\theta_1} \\ \frac{\partial Log(P(\theta_1, \theta_2))}{\partial \theta_2} &= \frac{n_h}{\theta_2} - \frac{n_t\theta_1}{1-\theta_2\theta_1} \end{aligned} \tag{9}$$

Using what we found in problem 5:

$$\frac{n_h}{\theta_1} - \frac{n_t\theta_2}{1-\theta_2\theta_1} = 0$$

$$\begin{aligned} \frac{\partial Log(P(\theta_1, \theta_2))}{\partial \theta_1} &= \frac{c_h}{\theta_1} - \frac{c_t}{1-\theta_1} + \frac{h-1}{\theta_1} - \frac{t-1}{1-\theta_1} \\ \frac{\partial Log(P(\theta_1, \theta_2))}{\partial \theta_2} &= \frac{c_h + h - 1}{\theta_1} - \frac{c_t + t - 1}{1-\theta_1} \end{aligned}$$

Setting to 0 and to solve for θ_{1MAP} we get:

$$\begin{aligned} \theta_{1MAP} = 0 &= (1-\theta_1)(c_h + h - 1) - (\theta_1 c_t + \theta_1 t + \theta_1) \\ \theta_{1MAP} &= \frac{c_h + h - 1}{c_h + h + 2 + c_t + t} \end{aligned} \tag{10}$$

$$\Delta_{\theta_2} Log(L(\theta_1, \theta_2)) = \frac{n_h}{\theta_2} - \frac{n_t\theta_{1MAP}}{1-\theta_2\theta_{1MAP}}$$

Setting to 0 and solving for $_2$ we now arrive at:

$$\begin{aligned}
\theta_{2MAP} = 0 &= (1 - \theta_2\theta_{1MAP})(n_h) - n_t\theta_2\theta_{1MAP} \\
\theta_{2MAP} &= n_h - n_h\theta_2\theta_{1MAP} - nt\theta_2\theta_{1MAP} \\
\theta_{2MAP} &= \frac{n_h}{n_h\theta_{1MAP} + n_t\theta_{1MAP}} \\
\theta_{2MAP} &= \frac{n_h}{n_h + n_t} * \frac{1}{\theta_{1MAP}} \\
\theta_{2MAP} &= \frac{n_h}{n_h + n_t} * \frac{c_h + h + 2 + c_t + t}{c_h + h - 1}
\end{aligned} \tag{11}$$

2 Derivation for multi-class modeling

Suppose our output space and our action space are given as follows: $\mathcal{Y} = \mathcal{A} = \{1, \ldots, k\}$. Given a non-negative class-sensitive loss function $\Delta : \mathcal{Y} \times \mathcal{A} \to [0, \infty)$ and a class-sensitive feature mapping $\Psi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$. Our prediction function $f : \mathcal{X} \to \mathcal{Y}$ is given by

$$f_w(x) = \arg\max_{y \in \mathcal{Y}} \langle w, \Psi(x, y) \rangle .$$

For training data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, let $J(w)$ be the ℓ_2-regularized empirical risk function for the multiclass hinge loss. We can write this as

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

for some $\lambda > 0$.

7. Show that $J(w)$ is a convex function of w. You may use any of the rules about convex functions described in our notes on Convex Optimization, in previous assignments, or in the Boyd and Vandenberghe book, though you should cite the general facts you are using. [Hint: If $f_1, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$ are convex, then their pointwise maximum $f(x) = \max\{f_1(x), \ldots, f_m(x)\}$ is also convex.]

   For our Multi-Class Hinge Loss Objective, there are two parts, the regularization term, $\lambda\|w\|^2$, which is convex as all norms are convex (its Hessian is Positive Definite), and the summation of argmax terms. Note: scaling the norm term by some λ retains the norms convexity.

   We then have $\alpha - m_w$ where $m_w = \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle$, and where α is the output from our $\Delta(y, y_i)$ function. This is a linear function, (specifically affine). Affine functions are both convex and concave, and the addition of two positive convex functions results in a convex function. Furthermore, due to the convexity notes (the maximum between (0, 1-m) is convex, as y=0 is a linear function, and their pointwise maximum is also convex) we know that the argmax section is indeed a convex function, therefore, our MultiClass Hinge Loss is indeed convex.

8. Since $J(w)$ is convex, it has a subgradient at every point. Give an expression for a subgradient of $J(w)$. You may use any standard results about subgradients, including the result

from an earlier homework about subgradients of the pointwise maxima of functions. (Hint: It may be helpful to refer to $\hat{y}_i = \arg\max_{y \in \mathcal{Y}} [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle]$.)

From the definition in the problem statement, let:

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

Now we have:

$$\begin{aligned} J(w) &= \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} [\Delta(\hat{y}, y) + \langle w, \Psi(x_i, \hat{y}) - \Psi(x_i, y_i) \rangle] \\ \frac{\delta J(w)}{\delta w} = \nabla J(w) &= 2\lambda w + \frac{1}{n} \sum_{i=1}^{n} [\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)] \end{aligned} \tag{12}$$

And now we have reached the desired subgradient: $\nabla J(w) = 2\lambda w + \frac{1}{n} \sum_{i=1}^{n} [\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)]$.

9. Give an expression for the stochastic subgradient based on the point (x_i, y_i).

   Using the subgradient we calculated above, we can plug in just a single datapoint, $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$:

$$\underset{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}}{\nabla J(w)} = 2\lambda w + [\Psi(x_i, \hat{y}) - \Psi(x_i, y_i)]$$

   And we achieve the subgradient expression for our stochastic gradient descent with point (x_i, y_i).

10. Give an expression for a minibatch subgradient, based on the points $(x_i, y_i), \ldots, (x_{i+m-1}, y_{i+m-1})$.
    Mini-Batch Gradient Descent:

$$\underset{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}}{\nabla J(w)} = 2\lambda w + \frac{1}{m} \sum_{i}^{i+m-1} (\Psi(x_i, \hat{y_i}) - \Psi(x_i, y_i))$$

(Optional) Hinge Loss is a Special Case of Generalized Hinge Loss

Let $\mathcal{Y} = \{-1, 1\}$. Let $\Delta(y, \hat{y}) = \mathbb{1} y \neq \hat{y}$. If $g(x)$ is the score function in our binary classification setting, then define our compatibility function as

$$\begin{aligned} h(x, 1) &= g(x)/2 \\ h(x, -1) &= -g(x)/2. \end{aligned}$$

Show that for this choice of h, the multiclass hinge loss reduces to hinge loss:

$$\ell(h, (x, y)) = \max_{y' \in \mathcal{Y}} [\Delta(y, y')) + h(x, y') - h(x, y)] = \max\{0, 1 - yg(x)\}$$

Breaking the problem into two cases: where $y \in [-1, 1]$ and $y \neq y'$, and one case where $y = y'$.

We first consider $y = 1$, and then y' $\neq$ y, we would observe $\Delta(y, y') = 1$, giving us a hinge loss that simplifies to:

$$\begin{aligned} l(h(x,y)) &= \max_{y' \in \mathcal{Y}}[1 + h(x,-1) - h(x,1)] \\ &= \max_{y' \in \mathcal{Y}}[1 + -g(x)/2 - g(x)/2] \\ &= \max_{y' \in \mathcal{Y}}[1 - yg(x)] \end{aligned} \tag{13}$$

For the second case, where $y = -1$ and and y' $\neq$ y, we find that we still observe $\Delta(y, y') = 1$, giving us:

$$\begin{aligned} l(h(x,y)) &= \max_{y' \in \mathcal{Y}}[1 + h(x,1) - h(x,-1)] \\ &= \max_{y' \in \mathcal{Y}}[1 + g(x)/2 - -g(x)/2 \\ &= \max_{y' \in \mathcal{Y}}[1 + g(x)] \\ &= \max_{y' \in \mathcal{Y}}[1 - yg(x)] \end{aligned} \tag{14}$$

The last step can be re expressed as stated above since $y = -1$.
Lastly, we have when $y' = y$, in which we would not have any loss, from $\Delta(y, y')$. Since h(x,y) = h(x,y') we have:

$$\begin{aligned} l(h(x,y)) &= \max_{y' \in \mathcal{Y}}[0 + h(x,y) - h(x,y')] \\ &= l(h(x,y)) = \max_{y' \in \mathcal{Y}}[0,0] = 0 \end{aligned} \tag{15}$$

Therefore, we can see that with choice of h, our Multi-Class hinge loss reduces to $\max(0, 1 - yg(x))$.

3 Implementation

In this problem we will work on a simple three-class classification example. The data is generated and plotted for you in the skeleton code.

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

First we will implement one-vs-all multiclass classification. Our approach will assume we have a binary base classifier that returns a score, and we will predict the class that has the highest score.

11. Complete the methods `fit`, `decision_function` and `predict` from `OneVsAllClassifier` in the skeleton code. Following the `OneVsAllClassifier` code is a cell that extracts the results of the fit and plots the decision region. You can have a look at it first to make sure you understand how the class will be used.

12. Include the results of the test cell in your submission.

Multiclass SVM

In this question, we will implement stochastic subgradient descent for the linear multiclass SVM, as described in class and in this problem set. We will use the class-sensitive feature mapping approach with the "multivector construction", as described in the multiclass lecture.

13. Complete the function `featureMap` in the skeleton code.
14. Complete the function `sgd`.
15. Complete the methods `subgradient`, `decision_function` and `predict` from the class `MulticlassSVM`.
16. Following the multiclass SVM implementation, we have included another block of test code. Make sure to include the results from these tests in your assignment, along with your code.