Homework 5: ML

Vishakh Padmakumar
DS-GA 1003 · Spring 2021 · NYU Center for Data Science

1 Bayesian Logistic Regression with Gaussian Priors

Q1. For the same dataset D described at the beginning of the Section, give an expression for the posterior density $p(w|D)$ in terms of the negative log-likelihood function $NLL_D(w)$ and the prior density $p(w)$

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

The negative log likelihood is related to $p(D|w)$ as $p(D|w) = exp(-NLL_D(w))$ because by definition, the negative of NLL_D gives us log likelihood which we exponentiate to get the likelihood of the data given the prior.

Therefore posterior $p(w|D) = \frac{p(w)\, exp(-NLL_D(w))}{p(D)}$

Q2. 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?

This is not a conjugate prior because the prior is a gaussian distribution and we calculate likelihood using a bernoulli so when you multiply the two to obtain an expression for the posterior, you won't reobtain a gaussian.

Q3. Show that there exist a covariance matrix Σ such that MAP (maximum a posteriori) estimate for w after observing data D is the same as the minimizer of the regularized logistic regression function defined in Regularized Logistic Regression paragraph above, and give its value.

$p(w|D) = \frac{p(w)\, exp(-NLL_D(w))}{p(D)}$

$-\log p(w|D) = -\log p(w) + NLL_D(w) - \log p(D)$

$p(w) = \frac{exp(\frac{-1}{2} \times w^T \Sigma^{-1} w)}{\sqrt{(2\pi)^{d/2}|\Sigma|}}$

$-\log p(w) = \frac{1}{2}(w^T \Sigma^{-1} w) - \frac{1}{2}((2\pi)^{d/2}|\Sigma|)$

We proved in HW4 that $NLL_D(w) = n\hat{R}_n(w)$

So the MAP i.e. minimizing the negative log posterior can be written as:

$\arg\min_w -\log p(w|D) = \arg\min_w n\hat{R}_n(w) + \frac{1}{2}(w^T\Sigma^{-1}w) - \frac{1}{2}((2\pi)^{d/2}|\Sigma|)$
$= \arg\min_w n\hat{R}_n(w) + \frac{1}{2}(w^T\Sigma^{-1}w)$ **.... (1)**

And we know that the regularized logistic regression objective, $J(w) = \hat{R}_n(w) + \lambda||w||^2$.

So minimizing the loss can be written as:

$\arg\min_w J(w) = \arg\min_w nJ(w) = \arg\min_w n\hat{R}_n(w) + n\lambda||w||^2$ **... (2)**

From **(1)** and **(2)**, we can observe that the term of empirical risk is the same, so the minimizer of the $p(w)$ term needs to correspond to the minimizer of the regularization term for us to obtain the required Σ.

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

$\Sigma = \frac{1}{2n\lambda}I$

So when $w \sim \mathcal{N}(0, \frac{1}{2n\lambda}I)$, $w_{MAP} = \arg\min_w -\log p(w|D) = \arg\min_w J(w)$

Q4. 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 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.

We have just shown that when $w \sim \mathcal{N}(0, \frac{1}{2n\lambda}I)$, the MAP estimate corresponds to the minima of our regularized LR. So if our prior knowledge indicates that $w \sim \mathcal{N}(0, I)$ is an appropriate prior, we can equate the 2 to get the reguired step size λ.

$\frac{1}{2n\lambda}I = I$

$\frac{1}{2n\lambda} = 1$

$\lambda = \frac{1}{2n}$

Q5. We additionally obtained a set of clean results 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.

The new likelihood $L(\theta_1, \theta_2) = p(D_r, D_c|\theta_1, \theta_2) = p(D_r|\theta_1, \theta_2)p(D_c|\theta_1, \theta_2)$

And here $p(D_c)$ is the clean value so it only depends on θ_1. So taking log, we get the log likelilihood as:

$LL(D_r, D_C|\theta_1, \theta_2) = n_h \log(\theta_1\theta_2) + n_t \log(1 - \theta_1\theta_2) + c_h \log(\theta_1) + c_t \log(1 - \theta_1)$

Now when we take the derivative w.r.t. each parameter, we get:

$\frac{dLL(D_r, D_c|\theta_1, \theta_2)}{d\theta_1} = \frac{n_h}{\theta_1\theta_2}\theta_2 - \frac{n_t}{1-\theta_1\theta_2}\theta_2 + \frac{c_h}{\theta_1} - \frac{c_t}{1-\theta_1} = \theta_2(\frac{n_h}{\theta_1\theta_2} - \frac{n_t}{1-\theta_1\theta_2}) + \frac{c_h}{\theta_1} - \frac{c_t}{1-\theta_1} = 0$

$\frac{dLL(D_r, D_c|\theta_1, \theta_2)}{d\theta_2} = \frac{n_h}{\theta_1\theta_2}\theta_1 - \frac{n_t}{1-\theta_1\theta_2}\theta_1 = \theta_1(\frac{n_h}{\theta_1\theta_2} - \frac{n_t}{1-\theta_1\theta_2}) = 0$

And we know that $\theta_1 = 0$ is not appropriate since they appear in the denominators of the first equation. So $\frac{n_h}{\theta_1\theta_2} - \frac{n_t}{1-\theta_1\theta_2} = 0$

Solving the two equations, we get:

$\frac{c_h}{\theta_1} - \frac{c_t}{1-\theta_1} = 0$

$\theta_{1\,MLE} = \frac{c_h}{c_h+c_t} = \frac{c_h}{N_c}$

$\theta_1\theta_2 = \frac{n_h}{n_h+n_t}$

$\theta_{2\,MLE} = \frac{1}{\theta_1}\frac{n_h}{n_h+n_t} = \frac{c_h+c_t}{c_h}\frac{n_h}{n_h+n_t} = \frac{N_c n_h}{N_r c_h}$

To confirm that this is indeed a maximizer of likelihood, we can do the double derivative test:

$\frac{d^2LL(D_r,D_c|\theta_1,\theta_2)}{d\theta_1^2} = \frac{-n_h}{2\theta_1^2} - \frac{n_t}{2\times(1-\theta_1\theta_2)^2}\theta_2^2 - \frac{c_h}{2\theta_1^2} - \frac{c_t}{2(1-\theta_1)^2}$ which is always negative.

$\frac{d^2LL(D_r,D_c|\theta_1,\theta_2)}{d\theta_2^2} = \frac{-n_h}{2\theta_2^2} - \frac{n_t}{2\times(1-\theta_1\theta_2)^2}\theta_1^2$ which is always negative.

Hence the estimates we found are indeed the maximizers of likelihood.

Q6. 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 $p(\theta_1) = Beta(h,t)$. Derive the MAP estimates for θ_1 and θ_2.

We are given a Beta prior for θ_1 as $p(\theta_1) = Beta(h,t)$. Hence the posterior is also going to be Beta distribution with parameters affected by the clean counts observed.

From the prior, we know that $p(\theta_1) \propto \theta_1^{h-1}(1-\theta_1)^{t-1}$

$p(\theta_1|D_c) \propto p(\theta_1)p(D_c|\theta_1)$

$\propto \theta_1^{h-1}(1-\theta_1)^{t-1} \times \theta_1^{c_h}(1-\theta_1)^{c_t}$

$\propto \theta_1^{h+c_h-1}(1-\theta_1)^{t+c_t-1}$

Or the posterior $p(\theta_1|D_c) = Beta(h+c_h, t+c_t)$

So MAP estimate can be $\theta_{1\,MAP} = \frac{h+c_h-1}{h+t+N_c-2}$.

We solve for MLE of θ_2 from likelihood of D_r and D_c:
$LL(D_r, D_c|\theta_1,\theta_2) = n_h\log(\theta_1\theta_2) + n_t\log(1-\theta_1\theta_2) + c_h\log(\theta_1) + c_t\log(1-\theta_1)$

And taking a derivative w.r.t θ_2, we obtain:
$\frac{dLL(D_r,D_c|\theta_1,\theta_2)}{d\theta_2} = \frac{n_h}{\theta_1\theta_2}\theta_1 - \frac{n_t}{1-\theta_1\theta_2}\theta_1 = 0$

$\theta_{2\,MLE} = \frac{1}{\theta_1}\frac{n_h}{n_h+n_t} = \frac{1}{\theta_{1\,MAP}}\frac{n_h}{n_h+n_t} = \frac{n_h}{N_r}\frac{h+t+N_c-2}{h+c_h-1}$

Note that in Q5, we showed the double derivative test for the same expression.

2 Derivation

Q7. Show that J(w) is a convex function of w.

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

Consider function: $f_y(w) = [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$ for any one particular (x_i, y_i) and a fixed class y.

With respect to w, the terms $\Delta(y_i, y)$ and $\Psi(x_i, y) - \Psi(x_i, y_i)$ are only dependant on the class y and hence are both constants. So $f_y(w)$ is an affine function function of w and is hence convex w.r.t w.

Now we can look at $\max_{y\in\mathcal{Y}}[\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$ as $\max_{y\in\mathcal{Y}} f_y(w)$. And we know that each $f_y(w)$ is an affine function w.r.t. w. The pointwise maximum of a set of convex functions is also convex.

As a result $\frac{1}{n}\sum_{i=1}^{n} \max_y[\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$ is sum of a set of convex functions (all we do is vary the (x_i, y_i) pair we choose) and is also convex w.r.t. w.

$\lambda||w||^2$ is a convex function of w since it is just the norm in $\mathbb{R}^d$

Therefore $J(w)$ is a sum of two functions both of which are convex w.r.t. w and we can conclude that $J(w)$ is also a convex function of w

Q8. Give an expression for a subgradient of $J(w)$.

We want to find the subgradient g of $J(w)$.
Consider $\hat{y} = \text{argmax}_{y\in\mathcal{Y}}[\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$ for a particular (x_i, y_i).

We can rewrite J as $J(w) = \lambda\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}[\Delta(y_i, \hat{y}) + \langle w, \Psi(x_i, \hat{y}) - \Psi(x_i, y_i)\rangle]$. Note that the selected $\hat{y}$ varies from point to point because we defined $\hat{y}$ for a particular (x_i, y_i).

Now if we only had one fixed y, the subgradient of the inner product term $f_y(w) = [\Delta(y_i, y) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i)\rangle]$, would be just $(\Psi(x_i, y) - \Psi(x_i, y_i))$ (Since the $\Delta(y_i, y)$ term is a constant w.r.t. w).

Therefore the subgradient, $g = 2\lambda w + \frac{1}{n}\sum_{i=1}^{n}(\Psi(x_i, \hat{y}) - \Psi(x_i, y_i))$

Q9. Give an expression for the stochastic subgradient based on the point (x_i, y_i).

The cost function J for SGD becomes $J(w) = \lambda\|w\|^2 + [\Delta(y_i, \hat{y}) + \langle w, \Psi(x_i, \hat{y}) - \Psi(x_i, y_i)\rangle]$ where the definition of $\hat{y}$ carries over from the previous question.

So the subgradient $g = 2\lambda w + (\Psi(x_i, \hat{y}) - \Psi(x_i, y_i))$

Q10. Give an expression for a minibatch subgradient, based on the points $(x_i, y_i), ..., (x_{i+m-1}, y_{i+m-1})$.

We can extend the answers from the previous 2 questions to obtain the required subgradient:
$g = 2\lambda w + \frac{1}{m} \sum_{j=i}^{i+m-1} (\Psi(x_j, \hat{y}) - \Psi(x_j, y_j))$
Again here $\hat{y}$ is selected for each point so it varies.

Optional question Let $y = \{-1, 1\}$. Let $\Delta(y, \hat{y}) = 1\{y \neq \hat{y}\}$. If $g(x)$ is the score function in our binary classification setting, then define our compatibility function as $h(x, 1) = g(x)/2, h(x, -1) = -g(x)/2$.

Show that for this choice of h, the multiclass hinge loss reduces to hinge loss: $l(h, (x, y)) = \max_{y' \in y}[\Delta(y, y')) + h(x, y') - h(x, y)] = \max\{0, 1 - yg(x)\}$

$l(h, (x, y)) = \max_{y' \in y}[\Delta(y, y')) + h(x, y') - h(x, y)]$

We know that $y' \in \mathcal{Y} = \{-1, 1\}$ so y' can either be y or $-y$.

If $y' = y$ then the term inside the maximizer is:
$[\Delta(y, y)) + h(x, y) - h(x, y)] = [1\{y \neq y\} + 0] = 0$

If $y' = -y$ then the term inside the maximizer is:
$[\Delta(y, -y)) + h(x, -y) - h(x, y)] = [1\{y \neq -y\} + h(x, -y) - h(x, y)] = 1 + h(x, -y) - h(x, y)$

We are given that $h(x, 1) = g(x)/2$ and $h(x, -1) = -g(x)/2$.
In other words $h(x, y) = yg(x)/2$.

So $1 + h(x, -y) - h(x, y) = 1 + (\frac{-yg(x)}{2}) - \frac{yg(x)}{2} = 1 - yg(x)$

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

Q11. 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.

```
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 "decision
    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
        assert self.n_classes == len(self.estimators)
        print("Starting fit: ")
        print("X: ", X.shape, " y:", y.shape)
        for current_label in range(self.n_classes):
            labels = []
            for label in y:
                if label == current_label:
                    labels.append(1)
                else:
                    labels.append(0)
            self.estimators[current_label].fit(X, labels)
```

```
        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
        res = []
        for current_label in range(self.n_classes):
            scores = self.estimators[current_label].decision_function(X)
            res.append(scores)

        res = np.asarray(res).T
        print("Result of decision function: ", res.shape)
        return res#0

    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
        scores = self.decision_function(X)
        pred = np.argmax(scores, axis=1)
        return pred#np.zeros(X.shape[0])
```

Q12. Include the results of the test cell in your submission.

Relevant Output from Jupyter Notebook-

```
Starting fit:
X:  (300, 2)  y: (300,)
Coeffs 0
```

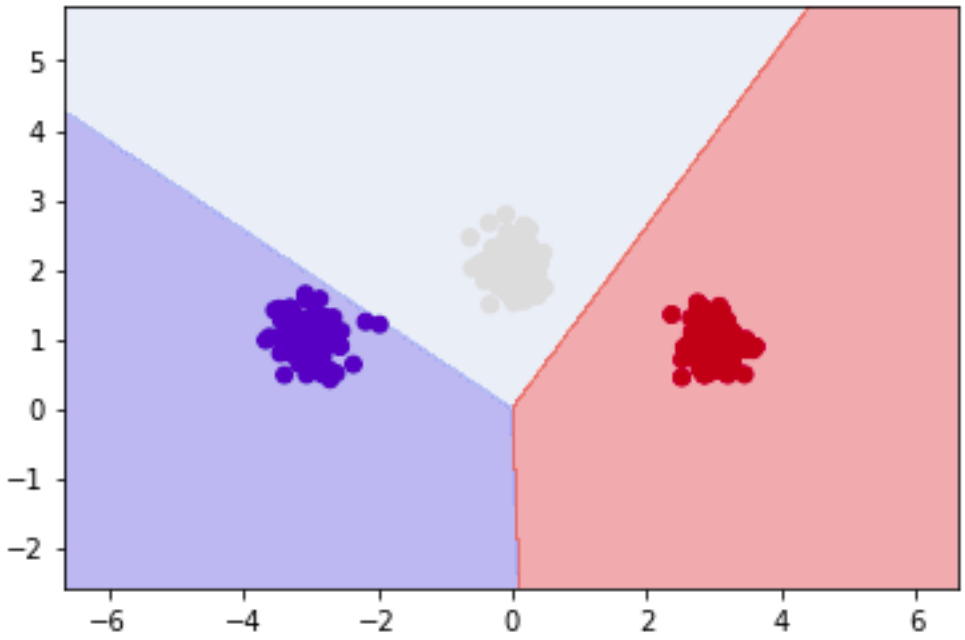

Figure 1: Decision boundary and training points of SVM for Q12

```
[[-1.05849489 -0.90295803]]
Coeffs 1
[[-0.38386961  0.14783551]]
Coeffs 2
[[ 0.89140301 -0.82558736]]
Result of decision function:  (278635, 3)
Result of decision function:  (300, 3)
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```

Image of decision boundary from Notebook Fig. 1

Q13. Complete the function featureMap in the skeleton code.

```
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[0],X
    #your code goes here, and replaces following return
    output_size = num_classes*num_inFeatures

    if len(X.shape) == 1:
        output = np.zeros(output_size)
        #print("Output: ", output.shape, y)
        output[y*num_inFeatures:(y+1)*num_inFeatures]=X
        #return output
    else:
        output = np.zeros((num_samples, output_size))
        for i in range(num_samples):
            temp = np.zeros(output_size)
            temp[y[i]*num_inFeatures:(y[i]+1)*num_inFeatures] = X[i]
            output[i, :] = temp
    return output
```

Q14. Complete the function sgd.

```
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(num_outFeatures)
```

```
    c = 0
    for iteration in range(T):
        idx = np.random.randint(len(X))
        #idx = c % len(X)
        c+=1
        subgradient = subgd(X[idx], y[idx], w)
        w = w - eta*subgradient
    return w
```

Q15. Complete the methods subgradient, decision function and predict from the class MulticlassSVM.

```
class MulticlassSVM(BaseEstimator, ClassifierMixin):
    '''
    Implements a Multiclass SVM estimator.
    '''
    def __init__(self, num_outFeatures, lam=1.0, num_classes=3, Delta=zeroOne, Psi=featu
        '''
        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
        margins = []
        for curr_y in range(self.num_classes):
            delta_term = self.Delta(y, curr_y)
```

```
            #print("W: ", w.shape)
            psi_term = self.Psi(x, curr_y) - self.Psi(x, y)
            psi_term = psi_term @ w
            margin = delta_term + psi_term
            margins.append(margin)

        reg = 2*self.lam*w
        yhat = np.argmax(margins)
        loss = self.Psi(x, yhat) - self.Psi(x, y)
        return loss+reg

    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.")

        margins = np.zeros((len(X), self.num_classes))
        for i in range(len(X)):
            margins[i, :] = [self.Psi(X[i], curr_y) @ self.coef_ for curr_y in range(sel
        #Your code goes here and replaces following return statement
        return margins#0

    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
```

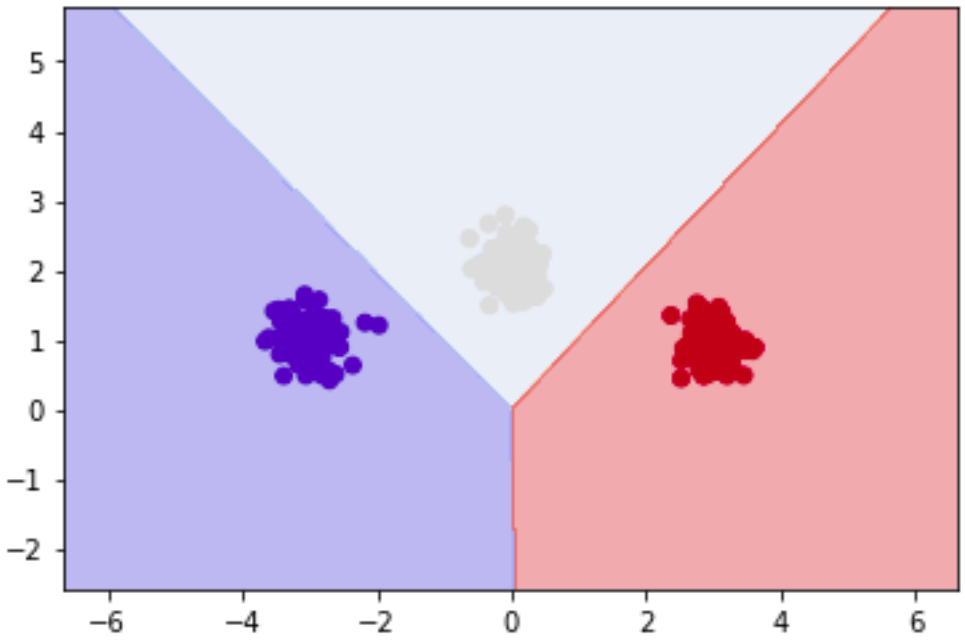

Figure 2: Decision boundary and training points of SVM for Q16

```
        '''
        #Your code goes here and replaces following return statement
        margins = self.decision_function(X)
        return np.argmax(margins, axis=1)#0
```

Q16. 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.

Output from relevant block

```
w:
[-0.58703912 -0.20757253 -0.00126246  0.39134313  0.58830159 -0.1837706 ]
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]])
```

Image of decision boundary from Notebook: Fig. 2