DS-GA 1003 - Homework 5

Eric Niblock

April 3, 2021

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 $\lambda > 0$.

1. **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.]**

   Let us call,

   $$f_i(w) = \Delta\left(y_i, y\right) + \langle w, \Psi(x_i, y) - \Psi(x_i, y_i) \rangle \tag{1}$$

   Then, obviously,

   $$f_i(w) = \Delta\left(y_i, y\right) + w^T \left(\Psi(x_i, y) - \Psi(x_i, y_i) \right) \tag{2}$$

   Which is of the form $y = w^T A + b$, and is thus an affine function of w. We know that affine functions are convex, and thus every $f_i(w)$ is convex. Then, as given in the problem

statement, if $f_i(w)$ is convex, then,

$$f(w) = \max_i \{f_i(w)\} \tag{3}$$

Is convex. So, rewriting the original function in terms of $f(w)$ yields,

$$J(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} f(w) \tag{4}$$

The sum of convex functions is convex, and norms are convex. Therefore $J(w)$ is convex.

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

If we can show that $J(w + z) \geq J(w) + g^T z$, then g would be a valid subgradient of $J(w)$. We propose,

$$g(w) = 2\lambda w + \frac{1}{n} \sum_{i=1}^{n} [\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)] \tag{5}$$

Is valid subgradient of $J(w)$. The following is a short proof:

$$
\begin{aligned}
J(w+z) &= \lambda||w+z||^2 + \frac{1}{n}\sum_{i=1}^{n}\max_{y\in\mathcal{Y}}\left[\Delta\left(y_i,y\right) + \langle (w+z), \Psi(x_i,y) - \Psi(x_i,y_i)\rangle\right] \\
&= \lambda||w+z||^2 + \frac{1}{n}\sum_{i=1}^{n}\left[\Delta\left(y_i,\hat{y}_i\right) + \langle (w+z), \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right] \\
&= \lambda||w||^2 + 2\lambda w^T z + \lambda||z||^2 + \frac{1}{n}\sum_{i=1}^{n}\left[\Delta\left(y_i,\hat{y}_i\right) + \langle w, \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right] \\
&\qquad + \sum_{i=1}^{n}\left[\langle z, \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right] \\
&= \left(\lambda||w||^2 + \frac{1}{n}\sum_{i=1}^{n}\left[\Delta\left(y_i,\hat{y}_i\right) + \langle w, \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right]\right) + 2\lambda w^T z \\
&\qquad + \lambda||z||^2 + \frac{1}{n}\sum_{i=1}^{n}\left[\langle z, \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right] \\
&= J(w) + 2\lambda w^T z + \lambda||z||^2 + \frac{1}{n}\sum_{i=1}^{n}\left[\langle z, \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right] \\
&\geq J(w) + 2\lambda w^T z + \frac{1}{n}\sum_{i=1}^{n}\left[\langle z, \Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)\rangle\right] \\
&= J(w) + \left(2\lambda w + \frac{1}{n}\sum_{i=1}^{n}[\Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)]\right)^T z
\end{aligned}
\tag{6}
$$

Note that the second to last line is a result of $\lambda||z||^2 \geq 0$ always. In summary we have,

$$
J(w+z) \geq J(w) + g^T z \tag{7}
$$

$$
J(w+z) \geq J(w) + \left(2\lambda w + \frac{1}{n}\sum_{i=1}^{n}[\Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)]\right)^T z \tag{8}
$$

$$
g(w) = 2\lambda w + \frac{1}{n}\sum_{i=1}^{n}[\Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)] \tag{9}
$$

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

The stochastic subgradient based on a single point (x_i, y_i) would be given by,

$$
g_s(w) = 2\lambda w + [\Psi(x_i,\hat{y}_i) - \Psi(x_i,y_i)] \tag{10}
$$

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

The minibatch subgradient based on points $(x_i, y_i), \ldots, (x_{i+m-1}, y_{i+m-1})$ is given by,

$$g_{mb}(w) = 2\lambda w + \frac{1}{m} \sum_{i}^{i+m-1} \left[\Psi(x_i, \hat{y}_i) - \Psi(x_i, y_i)\right] \tag{11}$$

(Optional) 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:

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

In general, we have two possibilities. Either, $y' = y$ or $y' \neq y$.

First let's assume deal with $y' = y$. We know that $\Delta(y, y') = 1(y' \neq y)$ simply represents the zero-one loss, and would therefore be zero in this instance. Then,

$$\ell\left(h, (x, y = y')\right) = \max_{y' \in \mathcal{Y}} \left[h(x, y') - h(x, y)\right] = 0 \tag{12}$$

Since $y' = y$ implies $h(x, y') = h(x, y)$.

Then we have the case when $y' \neq y$, which will entail two sub-cases (either y=1 or y=-1). Note that in either case, $\Delta(y, y') = 1$ since $y' \neq y$. So when $y' = 1$ and $y = -1$ we have,

$$\ell\left(h, (x, -1)\right) = \max_{y' \in \mathcal{Y}} \left[1 + h(x, 1) - h(x, -1)\right] = 1 + g(x) \tag{13}$$

And notice that $1 - yg(x) = 1 + g(x)$ when $y = -1$, as in this case. Then we have the alternative: $y' = -1$ and $y = 1$. This yields,

$$\ell\left(h, (x, 1)\right) = \max_{y' \in \mathcal{Y}} \left[1 + h(x, -1) - h(x, 1)\right] = 1 - g(x) \tag{14}$$

With $1 - yg(x) = 1 - g(x)$ when $y = 1$, as in this case. So, having gone through all of the cases, we've confirmed that correct classification provide zero loss, and incorrect classifications provide a loss of $1 - yg(x)$. This means,

$$\ell\left(h, (x, y)\right) = \max_{y' \in \mathcal{Y}} \left[\Delta\left(y, y'\right)) + h(x, y') - h(x, y)\right] = \max\left\{0, 1 - yg(x)\right\} \tag{15}$$

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.

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.

5. **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.**

   The following is the completion of the above three methods within our One vs. All Classifier,

```
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
    """

    self.y = y
    self.X = X
    for i in range(self.n_classes):
        self.temp = self.y.copy()
        self.temp[self.temp!=i] = -2
        self.temp[self.temp==i] = -1
```

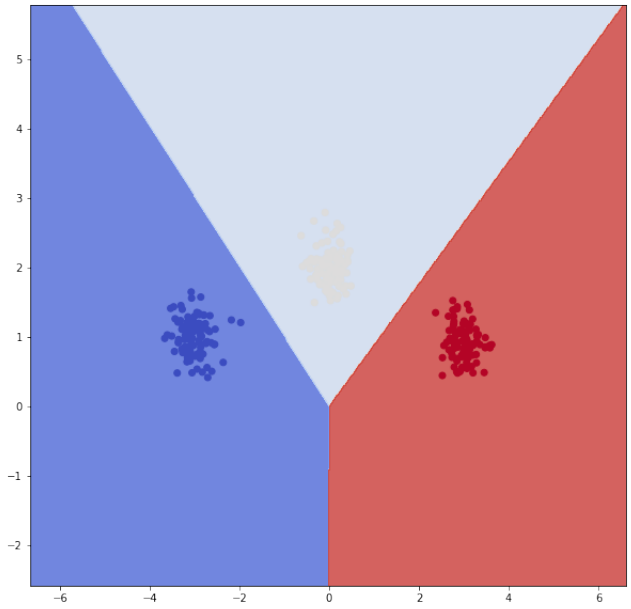
```
            self.estimators[i].fit(self.X,self.temp)

        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 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 data.")

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

        self.X = X
        return np.array([self.estimators[i].decision_function(self.X)
                         for i in range(self.n_classes)]).T

    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
        """

        self.X = X
        self.scores = self.decision_function(self.X)
        preds = [e.argmax() for e in self.scores]
        return np.array(preds)
```

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

After setting $C = 0.01$ and the number of iterations to $30,000$, we achieved the following results. We provide the values of the coefficients, the confusion matrix, and the resulting plot,

```
Coeffs 0
[[-0.55701685 -0.52074541]]
```

```
Coeffs 1
[[0.02891682 0.0595564 ]]
Coeffs 2
[[ 0.54538409 -0.52655583]]

array([[100,   0,   0],
   [  0, 100,   0],
   [  0,   0, 100]], dtype=int64)
```



As you can see, we were able to achieve perfect separation with reasonable margins.

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.

7. Complete the function `featureMap` in the skeleton code.

The following is our implementation of the feature map,

```
def featureMap(X,y,num_classes) :
    '''
    Computes the class-sensitive features.
    @param X: array-like, shape = [n_samples,n_inFeatures] or [n_inFeatures,],
    input 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 features 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.shape[0],X.shape[1])
    if len(X.shape) == 1:
        X=X[np.newaxis,:]
        yuse = np.array([y])
    else:
        yuse = y

    num_outFeatures = num_inFeatures*num_classes
    P = np.zeros([num_samples,num_outFeatures])
    for i in range(num_samples):
        P[i][num_inFeatures*yuse[i]:(num_inFeatures*yuse[i]+num_inFeatures)]=X[i,:]

    return P
```

8. Complete the function `sgd`**.**

The following is our implementation of stochastic gradient descent,

```
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]
    w = np.zeros(num_outFeatures)
    for t in range(int(T/num_samples)):
        shuf = [i for i in range(num_samples)]
        np.random.shuffle(shuf)
        X = X[shuf]
```

```
            y = y[shuf]
            for i in range(num_samples):
                move = subgd(X[i],y[i],w)
                w = w-eta*move

        return w
```

9. **Complete the methods** `subgradient`, `decision_function` **and** `predict` **from the class** `MulticlassSVM`.

The following is our implementation of the above three methods,

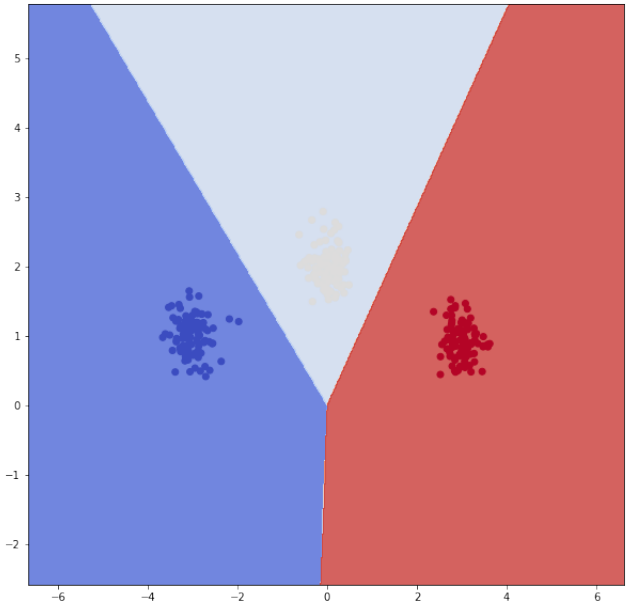
```
    def subgradient(self,x,y,w):
        '''
        Computes the subgradient at a given data point x,y
        @param x: sample input
        @param y: sample class
        @param w: parameter vector
        @return returns subgradient vector at given x,y,w
        '''

        y_max = 0
        psi_vec = self.Psi(x,y_max) - self.Psi(x,y)
        val_max = zeroOne(y_max,y) + w@psi_vec[0]
        for c in range(self.num_classes):
            psi_vec = self.Psi(x,c) - self.Psi(x,y)
            val = zeroOne(c,y) + w@psi_vec[0]
            if val > val_max:
                y_max = c
                val_max = val

        return(2*self.lam*w + self.Psi(x,y_max) - self.Psi(x,y))


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

        num_samples = X.shape[0]
```

```
        scores = np.zeros((num_samples,self.num_classes))

        for i in range(self.num_classes):
            for j in range(num_samples):
                scores[j,i] = self.coef_@self.Psi(X[j],i)[0]

        return scores


    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 point
        '''

        scores = self.decision_function(X)
        preds = np.zeros(X.shape[0])
        for s in range(scores.shape[0]):
            preds[s] = np.argmax(scores[s])

        return preds
```

10. 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.

After setting $\eta = 0.1$ and $\lambda = 0.1$, we achieved the following results. We provide the weight vector w, the confusion matrix, and the resulting plot,

```
    w:
[[-0.78052079 -0.17157619 -0.11191613  0.43794729  0.89243692 -0.2663711 ]]

    array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]], dtype=int64)
```



As you can see, we were able to achieve perfect separation with reasonable margins.