Problem Set III

Implementing Regression and Augmentation

Kunsh Singh

11/2/22

Problem 1

Regression: Given data (X, Y) with $X \in \mathbb{R}^d$ and $Y \in \{0, 1\}$, our goal is to train a classifier that will predict an unknown class label$\widetilde{y}$ from a new data point$\widetilde{x}$. Consider the following model:

(a) Write down the formula for the unnormalized posterior of $\beta|Y$, i.e.,

$$p(\beta|y; x; \sigma) \propto \Pi_{i=1}^{n} p(y_i|\beta; x_i)p(\beta; \sigma)$$

For the following equation, we determine the unnormalized posterior as a multivariate term to be the likelihood multiplied by the prior, such that the likelihood is represented by a Bernoulli distribution and the prior is represented by the Gaussian distribution. We get the following equation:

$$\sigma(x_i^T\beta)^{y_i}\sigma(1 - x_i^T\beta)^{1-y_i}\frac{1}{\sqrt{2\pi}\sigma}e^{-|\frac{\beta}{\sigma}|^2}$$

(b) Show that this posterior is proportional to $exp(-U(\beta)$, where

$$U(\beta) = \sum(x_i^T\beta)^{y_i}\sigma(1 - x_i^T\beta)^{1-y_i}\frac{1}{\sqrt{2\pi}\sigma}e^{-|\frac{\beta}{\sigma}|^2}$$

We start by computing $e^{-U(\beta)}$, getting the following equation for our multivariate conditional probability.

$$e^{(y_i-1)x_i^T\beta} \cdot e^{-ln(1+e^{-x_i^T\beta})} - \frac{1}{2\sigma^2}||B_k||^2$$

Using rules of logarithms and exponentiation, we can simplify our equation of $e^{-U(\beta)}$ down to the following terms when we take the natural log.

$$ln(\sigma(x_i^T\beta)^{y_i}) + ln(1 - \sigma(x_i^T\beta)^{1-y_i}) + ln(\frac{1}{\sqrt{2\pi}\sigma}e^{-|\frac{\beta}{\sigma}|^2})$$

Simplifying the following equation further we end up with the following:

$$ln(e^{-x_i^T\beta}) - ln(1 + e^{-x_i^T\beta}) = -x_i^T\beta - ln(1 + e^{-x_i^T\beta})$$

And finally, after further simplification and adding back our sigma term that we took our, we get the following equation which is similar to our original equation from (a) of our unormalized posterior.

$$e^{-U(\beta)} = (y_i - 1)x_i^T\beta - ln(1 + e^{-x_i^T\beta}) - |\frac{\beta}{\sigma}|^2$$

(c) Implement maximum a posterior (MAP) to infer β.
(d) Use your code to analyze the minist dataset (provided in PS2), looking only two digits 0 and 1. The 0/1 labels are your Y data, and the images are your X data. Also, add a constant term, i.e., a column of 1's to your X matrix. Make sure to trained β from the provided training data, and then use the trained β to get a prediction, $\widetilde{y}$, of the class labels for the test data.

We combine (c) and (d) since the solutions to both parts share a great deal of commonality with one another. For c-f, we need to use data from our dataset in mnist.mat, containing distorted images of the numbers 1-9. In order to implement MAP, we must first import our data from mnist.mat.

```
def main(self,mat=loadmat('code/mnist.mat')):
    DELTA = 0.8 #Step-size/Learning Rate
    SIGMA = 10 #Scalar
    EPSILON = 8 #CONVERGENCE INTERVAL based on gradient

    x = mat['trainX']
    testX = mat['testX']
    y = mat['trainY']
    testY = mat['testY']

    dataX = []
    dataY = []

    col = 1
```

Next, we must process our data from our trainX and trainY within our mnist.mat. The variables dataX and dataY are used to store training data, such that dataX contains images stored as a 1 dimensional array of pixels (1x784 for each image, 1x785 to account for added column of 1s), and dataY contains the digit contained in each image (1-9), which we will use to verify our answer. We also append the integer '1' to the end of each row in order to add the column of 1s as described by part d.

```
for i in range(len(x)):
    arr = x[i].tolist()
    if y[i][0] == 0:
        arr.append(1)
        dataX.append(arr)
        dataY.append(0)
        col+=1
    if y[i][0] == 1:
        arr.append(1)
        dataX.append(arr)
        dataY.append(1)

dataX = np.array(dataX)
dataY = np.array([dataY])
```

Let's proceed to implementing maximum a posterior. We do so by determining the l value at every pixel for every image. We use this to determine our gradient, and then finally add on $\frac{\beta}{\sigma^2}$. We use the following equation to determine our l:

$$\frac{dl}{dB_k} = \sum_{i=1}^{n}[(y_i - 1) + \frac{e^{-X_i \cdot \beta}}{1+e^{-X_i \cdot \beta}}]x_{ik}$$

After determining our value for the gradient, we can perform gradient ascent to determine our value for β. However, we use a unique approach in our code below, by changing the $y_i - 1$ in the equation in order to use gradient descent on our β, simplifying our process. Gradient ascent was modified to gradient descent also due to the fact that the output for gradient ascent was producing a greater error (error = 38.2%) (c).

```
BETA = np.zeros((dataX.shape[1])) #Initial beta value
l = 1
gradient = np.zeros(BETA.shape)
plotY = []
swatch = True #makes while loop a do while loop
```

```
while(swatch or self.magnitude(gradient)>EPSILON):
    for c in range (0, 785): #Image pixels
        l = 0.3
        for r in range (0, len(dataX)): #Num of images
            try:
                #Start of single line equation (see above for l)
                l += ((dataY[0][r]) -
                math.exp(-np.dot(np.array(dataX[r]).T,BETA))/
                (1 + (math.exp(-np.dot(np.array(dataX[r]).T,BETA))))
                ) * dataX[r][c]
                #End of single line equation (see above for l)
            except OverflowError:
                l += 0
        gradient[c] = l

    gradient += BETA/(SIGMA**2)
    plotY.append(self.magnitude(gradient))
    BETA -= DELTA * gradient
    swatch = False
```

Lets use our training data from mnist.mat to test our effectiveness of β. We start by importing our values for testX and testY, such that testX is all the test images (1x784 for each image, 1x785 to account for added column of 1s), and testY is the true digit value of each of the testX image (1-9). (d)

```
#Getting Testing Data
tX = []
tY = []
for i in range(len(testX)):
    arr = testX[i].tolist()
    if testY[i][0] == 0:
        arr.append(1)
        tX.append(arr)
        tY.append(0)
        col+=1
    if testY[i][0] == 1:
        arr.append(1)
        tX.append(arr)
        tY.append(1)

tX = np.array(tX)
tY = np.array([tY])
```

(e) Compare this to the true class labels, y, and see how well you did by estimating the average error rate, $E[|y - \widetilde{y}|]$ (a.k.a. the zero-one loss). What values of σ and the MAP stepsize Δ did you use?

Finally, we use our β to predict the value of each testY. If we are incorrect, we can increment our error by 1/n, such that n is the number of images contained in our image testing set. Finally, we multiply our error by 100 to determine the percent error we have for 0/1 regression.

```
#Generate predictions based on BETA
#Check if our predictions match our expected value

predictions = np.array([])
error = 0
for i in range(len(tX)):
    try:
        #Start of single line
        predictions = np.append(predictions,
        np.round(1/(1+math.exp(np.dot(tX[i], BETA.T)))))
        #End of single line
        if predictions[i] != tY[0][i]:
            error += 1
    except OverflowError:
        predictions = np.append(predictions, 0)
        error += 1

#Print error/step-size/sigma
error /= len(tX)
error *=  100 #Error percent

print("Error: "+str(error)+"%")
```

Using $\sigma = 10, \Delta = 0.8$, and $\epsilon = 8$, we were able to generate an error of 5.217% for our regression/estimation model for the 0 and 1 labels. σ is a hyper-parameter, Δ is our learning rate, and ϵ is our convergence interval. (e)

(f) Rerun your code to analyze the other two digits 6 and 8. Make sure to re-tune the model parameters for best performance.

We can simply replace our training data to check for labels '6' and '8' in our trainY/testY. However, we still append 0 and 1 to our trainY/testY respectively, such that the number '0' in our trainY/testY represents '6', and '1' in our trainY/testY represents '8'. This pattern effectively normalizes our data around 0 and 1, and thus we able to preserve our equation to determine our gradient l, as well as most of our other code.

Using $\sigma = 5, \Delta = 0.9$, and $\epsilon = 8$, we were able to generate an error of 0.8929% for our regression/estimation model for the 6 and 8 labels. σ is a hyper-parameter, Δ is our learning rate, and ϵ is our convergence interval. (e)

END OF 1

Problem 2

From our PS2, we have the following function to determine principal component analysis.

```
def pca(self,x):
    #Determines Principal Component Analysis
    # Normalize the input matrix
    x = (x - x.mean()) / x.std()

    # Determine the covariance matrix
    x = np.matrix(x)
    cov = (np.transpose(x) * x) / x.shape[1]

    # Perform Singular-Value Decomposition
    U, S, V = np.linalg.svd(cov)

    return U, S
```

We can load our data/eigenvalues/eigenvectors and pass them into our pca function:

```
# Load data from mnist.mat
data = mat
input = data['trainX']

# Get the principal components
eigenvecs, eigenvals = self.pca(input)
```

And we can reuse the following function such that e is in range of pc=10, pc=20, and pc=30. We do this for both dataX (input images from mnist.mat) and testX (testing images from mnist.mat) — this will project our images onto a low-dimensional space based on principal components, speeding up our classification processes significantly.

```
dataDownsized = []
for img in dataX:
    nft = []
    for e in range(10):
        nft.append(np.matmul(eigenvecs[e],img))
    dataDownsized.append(nft)
```

We use $\sigma = 10, \Delta = 0.8$, and $\epsilon = 8$ for all three of the following cases of PC. For $PC = 10$, we get an error of 1.739% in approximately 13 milliseconds. For $PC = 20$, we get an error of 0.0% in approximately 30 milliseconds. Finally, for $PC = 10$, we get an error of 1.739% again in approximately 48 milliseconds. All times presented are our training times.

END OF 2