question3

May 16, 2019

1 CS 520 Final: Question 3 - Classification

1.0.1 a) Construct a model to classify images as Class A or B, and train it on the indicated data. Specify your trained model. What does your model predict for each of the unlabeled images? Give the details of your model, its training, and the final result. Do the predictions make sense, to you?

1.0.2 To solve this question, I contract a logistic regression classfier model to classfy the images. First of all, I use the sigmoid fuction $f_w(x) = \frac{1}{1+e^{-(w*x)}}$ as activate function to transifer all varibles into 0 to 1.

```
In [181]: import numpy as np
          def sigmoid(t):
              return (1 / (1 + np.exp(-t)))
```

1.0.3 There are 'forward' and 'backward' propagation steps for learning the features. The probalities of each samples can be computed by $P = sigmoid(w^T x + b)$, The loss of forwad propagation is $J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} log(a^{(i)}) + (1 - y^{(i)}) log(1 - a^{(i)})$ Then we can use $\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y(i))$ to compute the gradient of the loss with respect to weights and bias.

```
In [182]: def initialize(x):
              w = np.zeros((x.shape[0],1))
              bias = 0
              return w, bias

          def propagate(weights, bias, x, y):
              size = x.shape[1]

              # forward propagation
              p = sigmoid(np.dot(weights.T,x)+bias)
              loss = (-1/size)*np.sum((y*np.log(p)) + ((1-y)*np.log(1-p)))

              # backward propagration
              d_weights = (1/size)*(np.dot(x,np.subtract(p,y).T))
              d_bais = (1/size)*(np.sum(p-y))
              return loss, d_weights, d_bais
```

1.0.4 For training part, I minimize the loss funnction to learn the weights and bias. $weights = weights - \alpha\partial w$, α is the learing rate. I set it to 0.5

```
In [183]: def train(weights, bias, x, y, iterations, learing_rate):
              z = []
              for i in range(iterations):
                  loss, d_weights, d_bais = propagate(weights, bias, x, y)
                  weights = weights-learing_rate*d_weights
                  bias = bias-learing_rate*d_bais

                  if i %100 ==0:
                      z.append(loss)

              return weights, bias, d_weights, d_bais, z
```

1.0.5 For training part, I calculate the probability according to the trained weights. If the probability is less or equal to 0.5, the image will classfy to class A, else it is class B.

```
In [187]: def predict(weights, bias, x):
              size = x.shape[1]
              predict_y = np.zeros((1,size))
              weights = weights.reshape(x.shape[0],1)
              p = sigmoid(np.dot(weights.T,x)+bias)
              print(p)
              for i in range(p.shape[1]):
                  if p[0,i] <=0.5:
                      predict_y[0,i] = 0
                  elif p[0,i] >0.5:
                      predict_y[0,i] = 1
              return predict_y, p
```

1.0.6 To extract the features, my strategy is to use 1 to represent black cell and 0 for white cell. Each cell in the image is a feature, then I get 25 features for total. My training set contains all 10 samples from classA and classB. This kind of input contain the information of black cell position and white cell position, but it may loss some information of neighbors. Since the size of images is small, and the data set is also small. I think this input is enough for the simple model to do the classification. The test set cotain the 5 mystery images. Then, I use 0 to represent classA and use 1 to represent classB.

```
In [262]: class_a = np.loadtxt('ClassA.txt', usecols=range(5))
          class_b = np.loadtxt('ClassB.txt', usecols=range(5))
          mystery = np.loadtxt('Mystery.txt', usecols=range(5))
          mystery1 = np.reshape(mystery[0:5],(1,25))
          mystery2 = np.reshape(mystery[5:10],(1,25))
          mystery3 = np.reshape(mystery[10:15],(1,25))
          mystery4 = np.reshape(mystery[15:20],(1,25))
          mystery5 = np.reshape(mystery[20:25],(1,25))
          class_a1 = np.reshape(class_a[0:5],(1,25))
```

```
class_a2 = np.reshape(class_a[5:10],(1,25))
class_a3 = np.reshape(class_a[10:15],(1,25))
class_a4 = np.reshape(class_a[15:20],(1,25))
class_a5 = np.reshape(class_a[20:25],(1,25))
class_b1 = np.reshape(class_b[0:5],(1,25))
class_b2 = np.reshape(class_b[5:10],(1,25))
class_b3 = np.reshape(class_b[10:15],(1,25))
class_b4 = np.reshape(class_b[15:20],(1,25))
class_b5 = np.reshape(class_b[20:25],(1,25))
train_x = np.concatenate((class_a1,class_a2,class_a3,class_a4,class_a5,class_b1,class_
train_x = train_x.T
train_y = np.array([0,0,0,0,0,1,1,1,1,1])
test_x = np.concatenate((mystery1,mystery2,mystery3,mystery4,mystery5))
test_x = test_x.T
```

1.0.7 Here are the labels and probabilities of each test samples

```
In [196]: weights,bias = initialize(train_x)
          weights, bias, d_weights, d_bais, z = train(weights,bias,train_x,train_y,200,0.5)
          predict_y, p1 = predict(weights, bias, test_x)
          predict_y
```

```
[[0.84547059 0.01149389 0.99932725 0.05827251 0.69033637]]
```

```
Out[196]: array([[1., 0., 1., 0., 1.]])
```

1.0.8 Result1

1.0.9 Mystery1: ClassB

1.0.10 Mystery2: ClassA

1.0.11 Mystery3: ClassB

1.0.12 Mystery4: ClassA

1.0.13 Mystery5: ClassB

1.0.14 This result makes some sense to me. For images in classA, the black cells are distributed at left top corner in my view. So mystery2 and mystery3 should be in classA from my subjective feeling. In opposite, the black cells are distributed at the right bottom corner. Therefore, mystery1, mystery3 and mystery5 should be in classB.

1.0.15 b) The data provided is quite small, and overfitting is a serious risk. What steps can you take to avoid it?

1.0.16 The data set of this questions is so small, so the overfitting is a serious problem. The model does well o the training set certainly, but it may not works well for new samples. The method which I used to avoid overfitting is L2 regularization. It penalizing the loss function to reduce the bias. We add the regularization term to our original loss function. The new loss fuction become to $L(x,y) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} log(a^{(i)}) + (1-y^{(i)}) log(1-a^{(i)}) + \lambda \sum_{i=1}^{n} \theta_i^2$ The regularization term keep weights small and simplifier the model. Lambda is the penalty term. It determines how much you want to penalizes the weights.

In [193]:
```
def lossWithRegularization(weights,size,lambd):
    loss = lambd/(2*size)*(np.sum(np.square(weights)))
    return loss


def regularizationPropagate(weights, bias, x, y):
    size = x.shape[1]

    # forward propagation
    p = sigmoid(np.dot(weights.T,x)+bias)
    loss = (-1/size)*np.sum((y*np.log(p)) + ((1-y)*np.log(1-p)))
    loss = loss + lossWithRegularization(weights,size, 0.1)

    # backward propagration
    d_weights = (1/size)*(np.dot(x,np.subtract(p,y).T))+0.1/size*weights
    d_bais = (1/size)*(np.sum(p-y))
    return loss, d_weights, d_bais

def regularizationTrain(weights, bias, x, y, iterations, learing_rate):
    z = []
    for i in range(iterations):
        loss, d_weights, d_bais = regularizationPropagate(weights, bias, x, y)
        weights = weights-learing_rate*d_weights
```

```
        bias = bias-learing_rate*d_bais

        if i %100 ==0:
            z.append(loss)

    return weights, bias, d_weights, d_bais, z
```

1.0.17 We can see that result of classifying does ot change, but the the standard deviation of probalilities without regularization is bigger than withing regularization. That means the model fit the trainning set less.

```
In [194]: weights,bias = initialize(train_x)
          weights, bias, d_weights, d_bais, z = regularizationTrain(weights,bias,train_x,train_y
          predict_y2, p2= predict(weights, bias, test_x)
          print(predict_y2)
          print('standard deviation of nonregularization', np.std(p1))
          print('standard deviation of regularization', np.std(p2))
```

```
[[0.77412117 0.03608859 0.99508565 0.11446669 0.63768133]]
[[1. 0. 1. 0. 1.]]
standard deviation of nonregularization 0.4090149464917842
standard deviation of regularization 0.37480612296630783
```

1.0.18 c) Construct and train a second type of model. Specify its details. How do its predictions compare to the first model? Are there any differences, and what about the two models caused the differences?

1.0.19 I choose linear regression as my second model. The linear regression is pretty similar as logistic regression, the it do not use sigmoid funtion to compute the probability. Now, $p = w^T x + b$. The loss function is different also. $J = \sum_{i=1}^{m}(y_i - wx_i - b)^2$, $\frac{\partial J}{\partial w} = \frac{2}{m}x(p-y)^T$ For the linear regression, I also used L2 regrularization to avoid overfitting

```
In [311]: def LRpropagate(weights, bias, x, y):
              size = x.shape[1]

              # forward propagation
              p = np.dot(weights.T,x)+bias
              loss = np.sum(np.square(y-np.dot(weights.T,x)))*1/size
              loss = loss + lossWithRegularization(weights,size, 0.005)

              # backward propagration
              d_weights = (2/size)*np.dot(x, (p-y).T)+0.005/size*weights
              d_bais = (2/size)*(p-y)
              return loss, d_weights, d_bais

          def LRtrain(weights, bias, x, y, iterations, learing_rate):
              z = []
              for i in range(iterations):
```

```
            loss, d_weights, d_bais = LRpropagate(weights, bias, x, y)
            weights = weights-learing_rate*d_weights

            if i %100 ==0:
                z.append(loss)

        return weights, bias, d_weights, d_bais, z

    def LRpredict(weights, bias, x):
        size = x.shape[1]
        predict_y = np.zeros((1,size))
        weights = weights.reshape(x.shape[0],1)
        p = np.dot(weights.T,x)
        print(p)
        for i in range(p.shape[1]):
            if abs(p[0,i]-0) <= abs(p[0,i]-1):
                predict_y[0,i] = 0
            elif abs(p[0,i]-0) > abs(p[0,i]-1):
                predict_y[0,i] = 1
        return predict_y, p
```

1.0.20 Result2

1.0.21 Mystery1: ClassB

1.0.22 Mystery2: ClassA

1.0.23 Mystery3: ClassB

1.0.24 Mystery4: ClassA

1.0.25 Mystery5: ClassB

1.0.26 The result given by linear regression is similar as the output of logistic regression. The only difference is mystery5. Mystery5 is a pretty tricky simple. The black cells are distributed evenly at 4 corners and the center. It looks like containning the all patterns from classA and classB. So it is really sensetive about the models. The different models may give different output of mystery5. Since the linear regression classifys the sample by a line. It may not fit the sample that is near the middle very well.

In [312]:
```
LRtrain_x = np.append(train_x,np.array([[1,1,1,1,1,1,1,1,1,1]]),axis=0)
weights,bias = initialize(train_x)
weights, bias, d_weights, d_bais, z = LRtrain(weights,bias,train_x,train_y,2000,0.05)
predict_y3, p3 = LRpredict(weights, bias, test_x)
predict_y3
```

```
[[0.84524005 0.1307516  1.29274917 0.15672514 0.27608897]]
```

Out[312]:
```
array([[1., 0., 1., 0., 0.]])
```