Machine Learning HW2 report

R05944013 網媒一 高滿馨

1.Logistic Regression Function

使用cross entropy 當cost function, 把trainData和weight做內積後, 送進sigmoid function 得到最終的預測值，使用adagrad algorithm來調整learning rate。最後classify的方式是，假如最終值的結果< 0.5 就是 0, 否則就是1。

$$C = -\frac{1}{n}\sum_{x}\left[y \ln a + (1-y)\ln(1-a)\right],$$

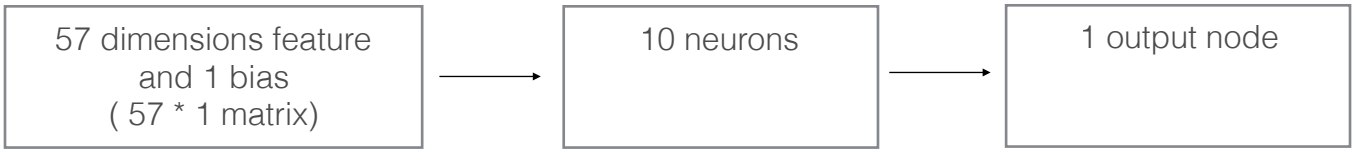
```
def sigmoid( X ):

    d = 1.0/(1.0 + np.exp( -1.0*X))
    return d
```

code fragment: sigmoid function

```
def gradientDescent( trainData, yHead, weight, count ):

    J_History = zeros( shape = ( iteration, 1 ) )
    accumulate = 0

    for x in range( 0, iteration ):


        prediction =  sigmoid( trainData.dot(weight) )


        for i in range( featureNum+1 ):

            #get trainData
            tmp = trainData[ :, i ]
            tmp.shape = ( count, 1 )

            #compute gradient
            derivative = ( ( ( prediction - yHead )*tmp ).sum() )/count

            #update accumulate ( adagrad algorithm )
            accumulate = accumulate + derivative*derivative
            #compute learning rate
            learningRate = alpha/(delta+sqrt(accumulate))

            #update weight
            weight[i][0] = weight[i][0] - learningRate*derivative

        J_History[x][0] = computeErrorRate( trainData, yHead, weight )
        print("finish iteration " + str(x) + ", error is "+ str( J_History[x][0] ) )

    return weight, J_History
```

code fragment: gradient descent

2. 方法2是使用3層的neural network來做training，第一層是input layer，第二層是hidden layer，neuron數可以調整，但試過多種參數後，最後決定使用10個neurons，最後一層是outputnode，可以直接output出最終值，假如小於0.5就判定為0，否則判定為1。sigmoid function則是使用hyper tangent。有使用adagrad algorithm來調整learning rate，另外有再去計算momentum來更新，效果確實有比較好。最後最好的model是使用adagrad和momentum，iteration為100圈的架構。



input layer　　hidden layer　　output layer

neural network架構圖

結果上傳kaggle的部分，最後neural network最好的model有比logistic regression的效果好。可能是因為中間那層hidden layer有做到feature transformation，所以原本無法被分開的一些測資，最後都可以順利地被classify，因此效果比logistic regression的效果好。

```
def train( self, trainData, trainLabel ):

    m = (trainData.shape)[0]
    accumulate = 0

    for i in range( iteration ):
        error = 0.0
        for x in range( m ):
            tmp = trainData[ x, : ]
            tmp.shape = ( featureNum, 1 )
            self.update( tmp )

            tmpLabel = trainLabel[ x, : ]
            tmpLabel.shape = ( outputNode, 1 )
            tmp = self.backPropagate( tmpLabel )
            error += tmp

        errorRate = error/m
        print( "finish iteraion " + str(i) + ", error rate is " + str(errorRate) )

    return self.wi, self.wo
```

train function

```
def update( self, inputs ):

    if (inputs.shape)[0] != self.ni-1:
        print("wrong number of inputs")

    #input activations
    self.ai[ :self.ni-1  , : ] = inputs[ : self.ni-1, :  ]

    #hidden activations
    for j in range( self.nh -1 ):
        total = 0.0
        for i in range( self.ni ):
            total += self.ai[ i, 0 ]*self.wi[ i, j ]
        self.ah[ j, 0 ] = sigmoid(total)

    #output activations
    for k in range( self.no ):
        total = 0.0
        for j in range( self.nh ):
            total += self.ah[ j, 0 ] * self.wo[ j, k ]
        self.ao[k] = sigmoid(total)

    return self.ao
```

update function (forward)

```
def backPropagate( self, targets ):

    if (targets.shape)[0] != self.no:
        print("wrong number of target values")

    #calculate error terms for output
    outputDeltas = zeros( shape = ( self.no, 1 ) )
    for k in range( self.no ):
        outputDeltas[k, 0] = dsigmoid( self.ao[ k, 0 ] )*( targets[ k, 0 ] - self.ao[ k ,0 ] )

    #calculate error terms for hidden
    hiddenDeltas = zeros( shape = ( self.nh, 1 ) )
    for j in range( self.nh ):
        error = 0.0
        for k in range(self.no):
            error += outputDeltas[ k, 0 ]*self.wo[ j, k ]
        hiddenDeltas[ j, 0 ] = dsigmoid( self.ah[ j, 0] )*error

    #update output weights
    for j in range( self.nh ):
        for k in range( self.no ):
            change = outputDeltas[ k, 0 ]*self.ah[ j, 0 ]
            self.sgo[ j, k ] += change*change
            learningRate0 = alpha/( delta+sqrt(self.sgo[ j, k ] ) )
            self.wo[ j, k ] = self.wo[j, k ] + learningRate0*change + M*self.co[ j, k ]
            #self.wo[ j, k ] = self.wo[j, k ] + alpha*change + M*self.co[ j, k ]
            self.co[ j, k ] = change


    #update input weights
    for i in range( self.ni ):
        for j in range( self.nh ):
            change = hiddenDeltas[j]*self.ai[i]
            self.sgi[ i, j ] += change*change
            learningRate1 = alpha/( delta+sqrt(self.sgi[ i, j ] ) )
            self.wi[i, j] = self.wi[i, j] + learningRate1*change + M*self.ci[ i, j ]
            #self.wo[ j, k ] = self.wo[j, k ] + alpha*change + M*self.co[ j, k ]
            self.ci[ i, j ] = change


    #calculate error
    error = 0.0
    for k in range( len(targets) ):
        if self.ao[k] < 0.5:
            if targets == 1:
                error += 1.0
        else:
            if targets == 0:
                error += 0.0

    return error
```

backward propagation

3.這次的作業中，在實作完基本的架構後，optimization的部分比較著重在learning rate調整的部分。有想過要用統計的方式去計算correlation，看是否能夠降維，不過看過data後覺得大部分的data都是0，感覺correlation不會有太好的效果，所以就沒有去測試。adaptive learning rate的部分，有把之前學過的多種演算法都拿來試試看，包括RMSProp，Adam等等，不過這些演算法雖然可以有效降低training error rate，找到真正的最低點，但反而會造成overfitting，準確度反而沒有提升。尤其是RMSProp，training error可以降低到0.0，但public set的準確度很低。因此，可以看出調整learning rate來得到optimize的效果是有限的。

Performance Comparison

	Training Error	Kaggle Score	iteration
Logistic Regression	0.072324418895	0.9333	100000
Neural Network with Adagrad and Momentum	0.02024493876530867	0.94	100
Neural Network with RMSProp	0.0	0.44	100
Neural Network with Adam	0.0204948762	0.936667	150