hw3 - to submit

October 21, 2017

1 CMPS 242 Homework Assignment 3

1.1 Sanjay Krishna Gouda

λ	0.001	0.01	0.1	1	10
Training accuracy	97.37	97.33	97.16	95.33	80.70
validation accuracy	**95.63**	95.53	95.16	93.60	78.9
Test accuracy (with l2 penalty)	**95.48**	-	-	-	-
Test accuracy (with l1 penalty)	**86.74**	-	-	-	-

Results *Time taken for stochastic dradient descent :621.94 seconds Time taken for mini batch dradient descent with batch size = 50 :468.02 seconds Time taken for batch gradient descent = 833.349572* ### Implementing Logistic Regression with Batch Gradient Descent. #### Logistic Regression hypothesis:

$$y^{'} = sigmoid(X.w)$$

With cost function

$$J(w) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\,log\,(h_w\,(x^{(i)})) - (1-y^{(i)})\,log\,(1-h_w(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{m}w_j^2$$

or in the vectorized form

$$J(w) = \frac{1}{m}((\,log\,(g(Xw))^T y + (\,log\,(1-g(Xw))^T(1-y)) + \frac{\lambda}{m}(||w||_2^2)$$

where m is the number of examples and g(z) is the sigmoidal activation given by

$$g(z) = \frac{1}{1+e^z}$$

Choosing regularizer (the λ) value based on 10 fold cross validation scores.

Adding regularizer to the cost function. Initially, just the L2 norm of weights but in later cells, other norms. (for extra credit).

In the cell below: * import train and test csvs * map spam/ham to 1/0 * remove stop words from train file * use tf-idf and vectorize the train file * use the vocabulary of the above vectorization and vectorize the test file * print the shapes of train and test matrices with last column being the mapped labels

the matrix built using tfidfvectorizer normalizes the matrix by default using norm = 'l2'

```
In [204]: import pandas as pd
          import numpy as np
          import math,os,time,itertools
          import matplotlib.pyplot as plt
          import pylab as pl
          import seaborn as sns
          from sklearn import metrics
          from sklearn.metrics import confusion_matrix
          import matplotlib.pyplot as plt
          import pylab as pl
          import seaborn as sns
          %matplotlib inline

          os.chdir("M:\Course stuff\Fall 17\CMPS 242\hw3")
          data = pd.read_csv("new_train.csv", encoding = "ISO-8859-1")
          #mapping spam/ham to 1/0
          data['label']=data['label'].map({'spam':1,'ham':0})
          y_train = data.iloc[:,data.columns=='label']
          # using nltk to remove stopwords
          text = data['sms']
          import nltk
          from nltk.corpus import stopwords
          stop = stopwords.words('english')
          for i in range(text.shape[0]):
                  text[i] = ' '.join([w for w in data['sms'][i].split() if not w in stopwords.w

          # tf-idf on train data
          from sklearn.feature_extraction.text import CountVectorizer,TfidfTransformer,TfidfVe
          vectorizer = TfidfVectorizer(stop_words = 'english')

          #temp_x = vectorizer.fit_transform(text)
          x_train = vectorizer.fit_transform(text).toarray()

          #storing the vocabulary
          vocab_dict = vectorizer.vocabulary_

          temp_x = vectorizer.fit_transform(text)
          x_train = temp_x.toarray()

          # now we have both x and y matrices which are the
          # input text and data and corresponding spam/ham labels
          import numpy as np
          train = np.concatenate((x_train,y_train), axis = 1)
          #print(train.shape)

          ### test data ###

          test_data = pd.read_csv("new_test.csv", encoding = "ISO-8859-1")
```

```
test_matrix = test_data
test_matrix['label'],test_matrix['sms']=test_data['label'].map({'spam':1,'ham':0}),te

### VECTORIZING WITH TRAIN VOCABULARY ###
temp_test = TfidfVectorizer(stop_words = 'english',vocabulary = vocab_dict).fit_trans
y_test = test_data.iloc[:,test_data.columns=='label']
test = np.concatenate((temp_test,y_test), axis = 1)
print("shape of train matrix %s\nshape of test matrix %s"%(train.shape,test.shape))

def sigmoid(z):
        return(1/(1+np.exp(-z)))
#sigmoid(train[:,-1])


def predict(yhat):
        for _ in range(yhat.shape[0]):
                if yhat[_,0]>=0.5:
                        yhat[_,0] = 1
                else:
                        yhat[_,0] = 0
        return yhat
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:25: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html


shape of train matrix (3000, 6023)
shape of test matrix (2572, 6023)
```

costfn Inputs : weights vector, training matrix (including labels) and regularizer that defaults to λ = 0.01 Splits the input training matrix into x and y matrices where y is the last column and x is all columns except the last. This matrix x is the one I use for training. Returns the cost based on the equation:

$$J(w) = \frac{1}{m}((\ log\ (g(Xw))^T y + (\ log\ (1 - g(Xw))^T (1 - y))$$

without regularization and

$$J(w) = \frac{1}{m}((\ log\ (g(Xw))^T y + (\ log\ (1 - g(Xw))^T (1 - y)) + \frac{\lambda}{m}(||w||_2^2)$$

where m is the number of examples and g(z) is the sigmoidal activation given by

$$g(z) = \frac{1}{1 + e^z}$$

** grads ** Inputs: Returns the gradient of cost function taken with respect to w. This is given by

$$\frac{\delta J(w)}{\delta w_j} = \frac{1}{m} X^T (g(Xw) - y)$$

without regularization and

$$\frac{\delta J(w)}{\delta w_j} = \frac{1}{m} X^T (g(Xw) - y) + \frac{\lambda}{m} w_j$$

with L2 regularization

```
In [149]: def costfn(w,matrix,reg = 0.01, penalty = 'l1'):
                m = matrix.shape[0]
                x= matrix[:,:-1]
                y= matrix[:,-1]
                h = sigmoid(np.dot(x,w))
                if penalty == 'l1':
                        cost = -(1/m)*(np.log(h).T.dot(y)+np.log(1-h).T.dot(1-y))+(reg/(2*m)
                if penalty == 'l2':
                        cost = -(1/m)*(np.log(h).T.dot(y)+np.log(1-h).T.dot(1-y))+(reg/(2*m)
                return cost[0]

          def grads(w,matrix,reg = 0.01, penalty = 'l1'):
                #print(matrix.shape)#calculates the derivative of cost function at the given
                m = matrix.shape[0]
                x= matrix[:,:-1]
                y= matrix[:,-1]
                w_reg = w
                h = sigmoid(np.dot(x,w)) #yhat
                yhatdiffy = np.subtract(h,y.reshape(y.shape[0],1))
                if penalty == 'l1':
                        grad = np.add((1/m)*(x.T.dot(yhatdiffy)),(reg/m)*(w))
                if penalty == 'l2':
                        grad = np.add((1/m)*(x.T.dot(yhatdiffy)),(reg/m)*np.sign(w))
                return grad.reshape(grad.shape[0],1)
```

** bgd_optimizer(w,matrix,n_iters = 100,reg = 0.01, penalty = 'l2'): ** ##### Batch gradient descent function Inputs: * n_iters = Number of times the weights update process is repeated. * reg = the λ (regularizer) value. Defaults to 0.01 * penalty = if 'l1', uses l1 norm regularizer and if 'l2' uses l2 norm regularizer. * w = initial weights matrix * matrix = the matrix on which to train on. Returns: * w_opt = Returns the weight vector after finishing the update mechanism/ optmization.

Algorithm: * Actual Learning Rate:

$$\eta = \eta_0 . t^{-\alpha}$$

where $\alpha = 0.9$ here I set $\eta_0 = 50$ and used more number of iterations so that the net learning rate is not too small. * Update Rule:

$$w := w - \eta . grads(weights = w, matrix = entireinputmatrix, reg = reg)$$

* Updates the parameters just once after computing the grads on entire matrix.

```
In [98]: def bgd_optimizer(w,matrix,n_iters = 100,reg = 0.01, penalty = 'l2'):
```

```
        # updates the weights matrix by computing the delta over entire input matrix
        # w := w- sum(delta(wx-y))
        for i in range(1,n_iters+1):
                if i%250==0:
                        print("\t iteration %d of %d. Cost = %r"%(i,n_iters,costfn(w,m
                        learning_rate = 50*np.power(i,-0.9)  #eta = eta0*(iteration^-0
                        delta = grads(w,matrix = matrix,reg = reg, penalty = penalty)
                learning_rate = 100*np.power(i,-0.9)  #eta = eta0*(iteration^-0.9)
                delta = grads(w,matrix = matrix,reg = reg, penalty = penalty)
                w = w - learning_rate * (delta) # w:= w - delta*()
        return w
```

** minibatch_optimizer(n_iters,batch_size,w,matrix,reg = 0.01, print_cost = True): ** Inputs: * n_iters = Number of times the weights update process is repeated. * batch_size = Number of examples using which the costfn and grads functions are used to update the weights vector. * reg = the λ (regularizer) value. Defaults to 0.01 * penalty = if 'l1', uses l1 norm regularizer and if 'l2' uses l2 norm regularizer. * w = initial weights matrix * matrix = the matrix on which to train on. Returns: * w_opt = Returns the weight vector after finishing the update mechanism/ optmization.

Algorithm: * Actual Learning Rate:

$$\eta = \eta_0.t^{-\alpha}$$

where $\alpha = 0.9$ here I set $\eta_0 = 1$ * Update Rule:

$$w := w - \eta.grads(weights = w, matrix = current_batch, reg = reg)$$

* Updates are applied iteratively using the batches. * Batch construction: Used two pointers with fixed distance which equals the batch size and iteratively around the values each time passing these two pointers as indices of the matrix in the grads() function call.

** Note ** : Setting batch_size to 1 gives Stochastic Gradient Descent.

```
In [153]: def minibatch_optimizer(w,matrix,n_iters = 100,batch_size = 50,reg = 0.01, penalty =

                for i in range(1,n_iters+1):
                        np.random.shuffle(matrix) #shuffle data first
                        init = 0
                        b_s = init+batch_size
                        batches = int(matrix.shape[0]/batch_size) #total number of batches
                        learning_rate = 1*np.power(i,-0.9)  #eta = eta0*(iteration^-0.9)

                        if batches == matrix.shape[0]:
                                for j in range(batches-1):
                                        #print(init,init+1)
                                        delta = grads(w,matrix = matrix[init:init+1,:],reg =
                                        w = w - learning_rate * (delta)
                                        init += 1
                                last = matrix[-2:-1,:]
                                w = w - learning_rate*(grads(w,matrix = last,reg = reg, pena
                        if i%500==0:
```

```
                        print("\niteration %d of %d"%(i,n_iters))
                        current_cost = costfn(w,matrix = matrix ,reg = reg,penalty =
                        print("current cost = ",current_cost)
                # update rule applied to each batch
                if batches != matrix.shape[0]:
                        for j in range(1,batches+1):
                                if b_s > matrix.shape[0]: #in case batch size is not
                                        b_s = b_s - matrix.shape[0]
                                        delta = grads(w,matrix = matrix[:-b_s,:],reg
                                        w = w - learning_rate * (delta)

                                if b_s < matrix.shape[0]:
                                        delta = grads(w,matrix = matrix[init:b_s,:],
                                        w = w - learning_rate * (delta) # w:= w - de
                                        init = b_s+1
                                        b_s += batch_size

        return w
```

Helper function to plot the confusion matrix

```
In [61]: def plot_confusion_matrix(cm, classes=['ham','spam'],
                                   normalize=False,
                                   title='Confusion matrix',
                                   cmap=plt.cm.Blues):
             """
             This function prints and plots the confusion matrix.
             Normalization can be applied by setting `normalize=True`.
             """
             if normalize:
                 cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
                 print("Normalized confusion matrix")
             else:
                 print('Confusion matrix, without normalization')

             print(cm)

             plt.imshow(cm, interpolation='nearest', cmap=cmap)
             plt.title(title)
             plt.colorbar()
             tick_marks = np.arange(len(classes))
             plt.xticks(tick_marks, classes, rotation=45)
             plt.yticks(tick_marks, classes)

             fmt = '.2f' if normalize else 'd'
             thresh = cm.max() / 2.
             for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
                 plt.text(j, i, format(cm[i, j], fmt),
```