PART I: Logistic Regression

Theory

1. What is the running time of your mini batch gradient descent algorithm?
   Cost is calculated in every round of the algorithm in order to obtain new gradient and weight vector, therefore, time will be dependent of number of batch size and dimensions of the data, so which is *O(bd)*. Computing cost and gradient includes basic mathematical operation but we are focusing on dot products. Total running time would depend on our batch size, sample size (n) and number of epochs we are going to repeat the process, which makes total time of *O(e n d)*

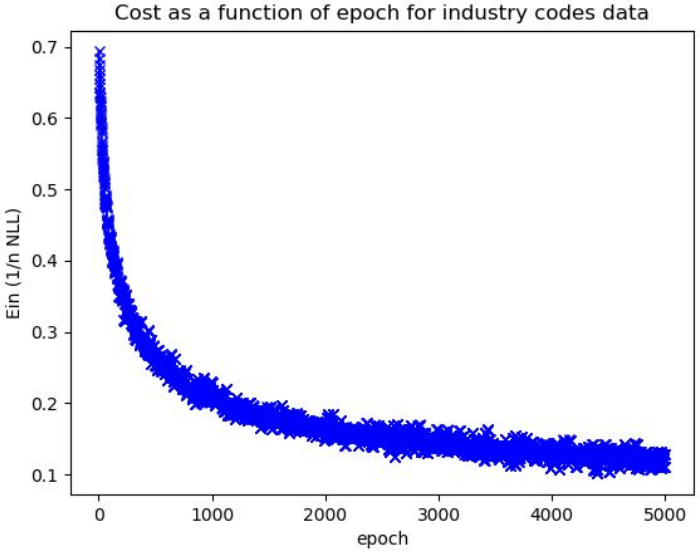
2. Sanity Check
   First thought would be that it will make it worse since the example of handwritten digits position of pixel matters as the build up the structure of the handwritten digits. It depends whether the permutation is random and different for each image. In this case, the model will perform worse. However, if the permutation is random but the same for each image, or systematic non-random, then the model will perform the same as the one using no permutation.

3. Linear Separable Data
   We can always drive function upwards, towards infinity, therefore function has no maximum and attempting to find maximum will be a forever work or “gap” between classes will be bigger and bigger and weights increase to infinity. Problem will have many solutions and logistic regression results would oscillate a lot. In terms of maximum likelihood, the step function will fit data perfectly, which means that maximum likelihood estimate will select parameters of infinite magnitude and will allow for many different parameters.

Code

Summary and result

By running "*python logistic_regression.py*" command with batch size of 1000, learning rate of 0.1 and 1000 epochs, which are quite big numbers for those arguments we got results of:
In Sample Score: 0.9637707948243993
Test Score: 0.9517738359201774
Which is what we expected from the description



The Cost function plot (right) indicates decrease of cost as the epochs were were increasing. In other words, cost function is here to estimate how wrong the model is the model in terms of its ability to estimate parameters. In our example we can see that cost decreases over time but it should stabilize after some time(meaning we cannot significantly improve accuracy). Running with more iteration could show that on the plot. Computing cost takes O(nd) time if we consider that logistic functions i in constant time.

Actual code

```
def cost_grad(self, X, y, w):
        """
        Compute the average cross entropy and the gradient under the logistic regression model
        using data X, targets y, weight vector w

        np.log, np.sum, np.choose, np.dot may be useful here
        Args:
           X: np.array shape (n,d) float - Features
           y: np.array shape (n,)  int - Labels
           w: np.array shape (d,)  float - Initial parameter vector

        Returns:
           cost: scalar the cross entropy cost of logistic regression with data X,y
           grad: np.arrray shape(n,d) gradient of cost at w
        ""
        cost = 0
        grad = np.zeros(w.shape)
        ### YOUR CODE HERE 5 - 15 lines
        cost = -np.sum( y*np.log(logistic(X.dot(np.transpose(w)))) +
(1-y)*np.log(1-logistic(X.dot(np.transpose(w)))) )
        cost = 1.0/len(X) * cost

        deltaNLL = y - logistic(X.dot(w.transpose()))
        deltaNLL = -deltaNLL.transpose().dot(X)
        grad = (1.0 / len(X)) * deltaNLL


        ### END CODE
        assert grad.shape == w.shape
        return cost, grad
```

```
def fit(self, X, y, w=None, lr=0.1, batch_size=3, epochs=10):
    """
    Run mini-batch stochastic Gradient Descent for logistic regression
    use batch_size data points to compute gradient in each step.
```

```
    The function np.random.permutation may prove useful for shuffling the data before each epoch
    It is wise to print the performance of your algorithm at least after every epoch to see if progress is being made.
    Remeber the stochastic nature of the algorithm may give fluctuations in the cost as iterations increase.

    Args:
        X: np.array shape (n,d) dtype float32 - Features
        y: np.array shape (n,) dtype int32 - Labels
        w: np.array shape (d,) dtype float32 - Initial parameter vector
        lr: scalar - learning rate for gradient descent
        batch_size: number of elements to use in minibatch
        epochs: Number of scans through the data

    sets:
        w: numpy array shape (d,) learned weight vector w
        history: list/np.array len epochs - value of cost function after every epoch. You know for plotting
    """
    if w is None: w = np.zeros(X.shape[1])
    history = []
    n = X.shape[0]
    ### YOUR CODE HERE 14 - 20 lines
    for i in range(epochs):
        X_shuffle,Y_shuffle = shuffle(X, y)
        for j in range(n // batch_size):
            X_subset = resample(X_shuffle, n_samples = batch_size, random_state=0)
            y_subset = resample(Y_shuffle, n_samples = batch_size, random_state=0)
            cost, grad = self.cost_grad(X_subset, y_subset, w)  # compute cost and gradient of the data with weights
            history.append(cost)  # remember the loss in iteration
            w -= lr * grad  # upgrade weights depending on gradient
            #lr = lr * 0.99
            #print("Cost",cost)
            #print("Gradient", grad)

    ### END CODE
    self.w = w
    self.history = history
```

PART II: Softmax

Theory

Cost grad time complexity: *O(ndK)* since the time consuming step is multiplying a n*d matrix with a d*K matrix

Total time complexity: *O(endK)* since we have to compute cost and gradient which depends on dimensionality of our data, sample size we take and repeat it for e- number of epochs. Also we have to encode K number of classes.

Code

Cost grad

```
def cost_grad(self, X, y, W):

        Yk = one_in_k_encoding(y, self.num_classes)
        input_size = X.shape[0]
        cost = np.nan
        grad = np.zeros(W.shape) * np.nan
        soft = np.log(softmax(np.dot(X, W)))
        cost = -np.sum((Yk.T.dot(soft[Yk == 1])) / input_size)

        grad = -np.transpose(X).dot(Yk - softmax(X.dot(W))) / input_size

        ### END CODE
        return cost, grad
```

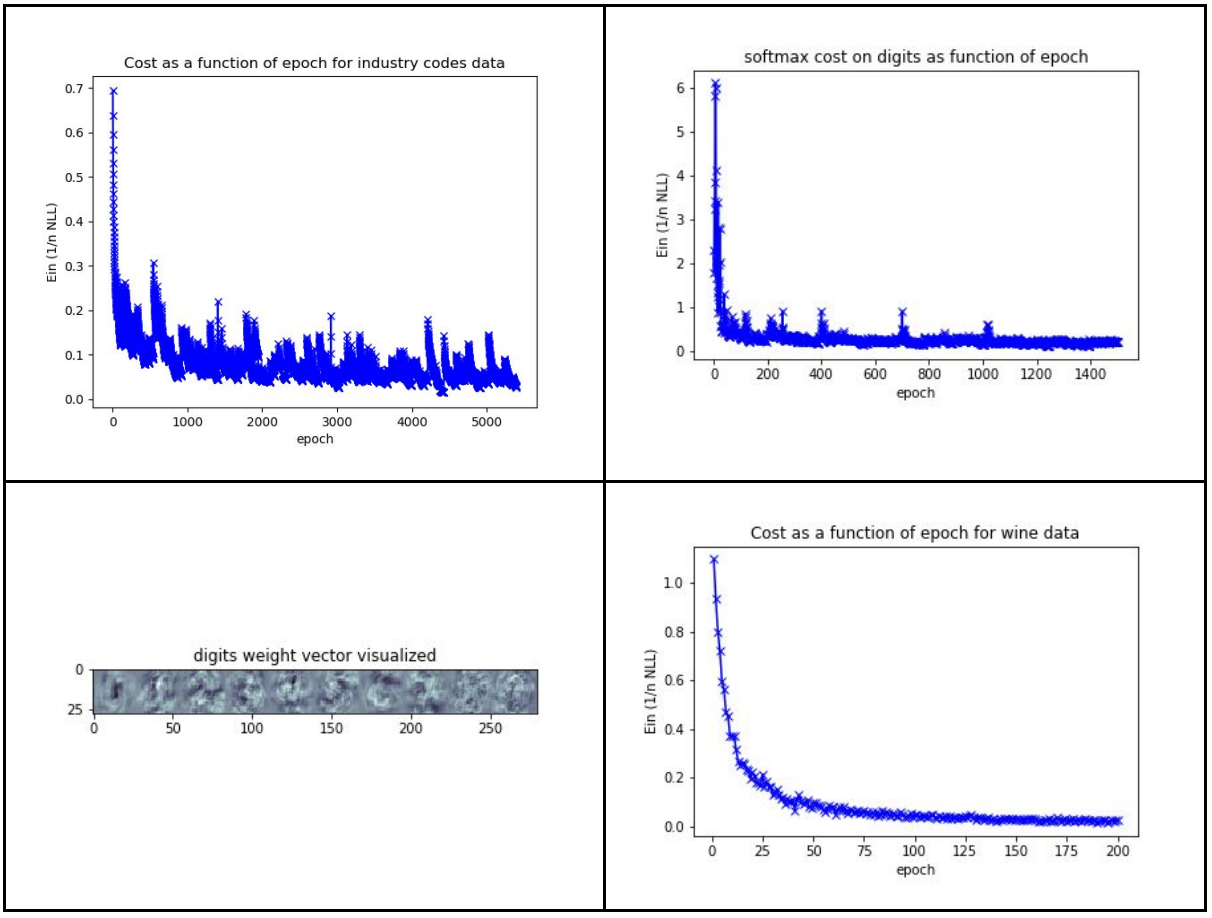
Fit

```
def fit(self, X, Y, W=None, lr=0.01, epochs=10, batch_size=2):

         if W is None: W = np.zeros((X.shape[1], self.num_classes))
         history = []
         ### YOUR CODE HERE
         n = X.shape[0]
         for j in range(epochs):
             X_shuff, Y_shuff = shuffle(X, Y)
             for i in range(n // batch_size):
                 X_mini = resample(X_shuff, n_samples=batch_size, random_state=0)
                 Y_mini = resample(Y_shuff, n_samples=batch_size, random_state=0)
                 cost, grad = self.cost_grad(X_mini, Y_mini, W)
                 history.append(cost)
                 W -= lr * grad
```

```
            #print("Cost", cost)
            #print("W", W)
        ### END CODE
        self.W = W
        self.history = history
```

Graphs and visualisations:



Wine in sample error: 0.906647398844
Wine test error: 0.884496124031