

Machine Learning (E23) Assignment 1

Tobias Washeim: 202007525, Alexander Myrtoft Skjødt: 202004110

October 2nd, 2023

Contents

I	Logistic Regression	2
1	Code	2
1.1	Summary and Results	2
1.2	Actual Code	2
2	Theory	2
2.1	Running Time	2
2.2	Sanity Check	3
2.3	Linearly Separable Data	3
II	Softmax Regression	3
3	Code	4
3.1	Summary and Results	4
3.2	Actual Code	4
4	Theory	4

Part I

Logistic Regression

1 Code

1.1 Summary and Results

Running our code for logistic regression a test accuracy of 94.46% . The in-sample accuracy running the code is 95.12%. On fig. 1 the graph for the cost as a function of epoch can be seen.

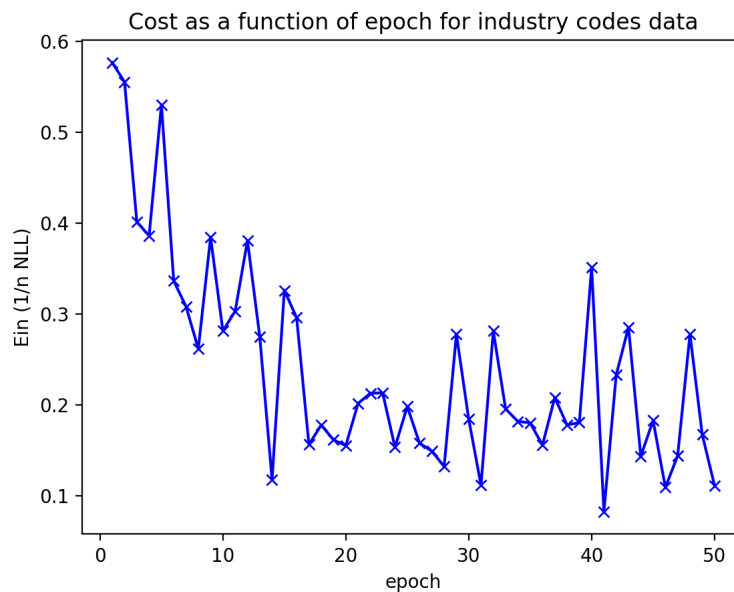


Figure 1: Evolution of the cost over each epoch when training the logistic regression model. We see a downward trend as the model is trained on more data, and end with an acceptable accuracy.

1.2 Actual Code

2 Theory

2.1 Running Time

The running time to compute a matrixproduct is proportional to $O(abc)$ where O is big O-notation and the matrix product is composed of matrices with dimensions of $a \times b$ and $b \times c$.

We compute the running time of the cost function. For each epoch, e , calculations in regard to the mini-batches are calculated. It is clear that the running time is proportional to e so $\mathcal{O}_{\text{epoch}}(e)$. For each epoch the number of batches computed is n/b so the time is $\mathcal{O}_{\text{Batch}}(n/b)$ where b is the size of the batch. Next, for each batch the cost function and cost gradient decent is computed.

```
def cost_grad(self, X, y, w):
    """
    Compute the average negative log likelihood and 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 average negative log likelihood for logistic regression with data X, y
        grad: np.array shape(d, ) gradient of the average negative log likelihood at w
    """
    n, d = np.shape(X)
    cost = np.sum([np.log(1+np.exp(-y[i]*np.dot(w,X[i]))) for i in range(n)]) / n
    grad = np.zeros(w.shape)

    grad_vector=[]
    for i in range(n):
        grad_vector.append(-1*y[i]*X[i,:]*logistic(-1*y[i]*np.dot(w,X[i,:])))
    grad = np.sum(grad_vector,axis=0)/n

    assert grad.shape == w.shape
    return cost, grad
```

Figure 2: Cost Grad function for logistic regression

```
def fit(self, X, y, w=None, lr=0.1, batch_size=16, 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.
    Remember 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 loss function (ln-sample error) after every epoch. Used for plotting
    """
    if w is None: w = np.zeros(X.shape[1])
    history = []
    for e in range(epochs):
        data = np.random.permutation(np.insert(X,0,y,axis=1))
        batch_start, batch_end = 0, batch_size
        while batch_end < len(data):
            batch = data[batch_start:batch_end]
            cost, grad = self.cost_grad(batch[:,1:],batch[:,0],w)
            w -= lr*grad
            batch_start += batch_size
            batch_end += batch_size
            print("Cost after epoch #{} is {}".format(e, cost))
            history.append(cost)

    self.w = w
    self.history = history
```

Figure 3: Fit function for logistic regression

Cost function:

The cost function running time must be related to the dotproduct between a row of the matrix X and w which is proportional to $\mathcal{O}(d)$. The dotproduct is computed b times, once for each data point in the batch.

$$\mathcal{O}_{\text{cost}}(dne) = \mathcal{O}_{\text{epoch}}(e)\mathcal{O}_{\text{loop}}(n/b)\mathcal{O}_{\text{datapoints}}(b)\mathcal{O}_{\text{dotproduct}}(d) \quad (1)$$

Gradient descent:

The running time for the gradient descent is related to the produt between the rows of matrix X (which gives a proportionality of d), the dotproduct between w and $X[i,:]$ (proportional to d), we run over the entire batch size (proportional to b). We do the calculations for all batches which is n/b and repeat for all epochs (e). The running time is

$$\mathcal{O}_{\text{grad}}(d^2ne) = \mathcal{O}_X(d)\mathcal{O}_{\text{dot}}(d)\mathcal{O}_{\text{range}}(b)\mathcal{O}_{\text{batches}}(n/b)\mathcal{O}_{\text{epoch}}(e). \quad (2)$$

2.2 Sanity Check

Doing a random permutation of a pixel with the same permutation makes the classifier worse, as the locality of pixels matter when doing image recognition. Our model do not use locality, so we can permutate data points randomly without problem.

2.3 Linearly Separable Data

When the data is linearly separable the gradient descent will eventually find a perfect minima where the gradient is 0. Each component (i.e. weight) will then be a constant.

Part II

Softmax Regression

3 Code

3.1 Summary and Results

The accuracy of the test of Softmax regression is 97.78% and the in-sample test is 100%.

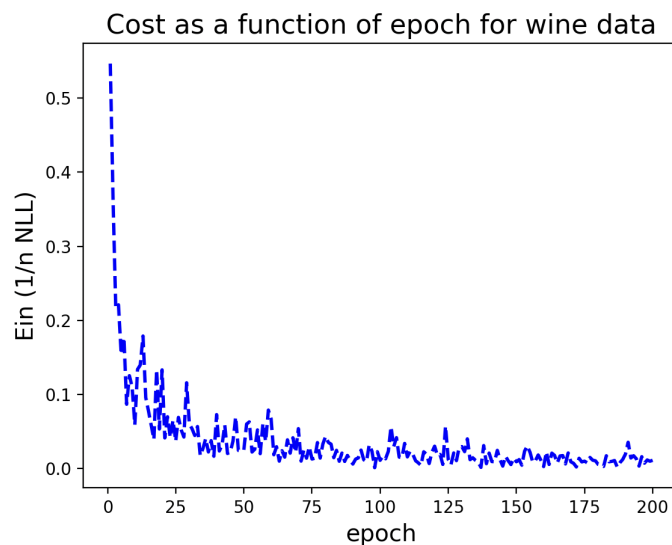


Figure 4: Evolution of the cost over each epoch when training the logistic regression model. The cost very quickly declines over the first few epochs, and then flattens out at low cost/good accuracy.

3.2 Actual Code

4 Theory

Again our running time is proportional to e , the number of epochs. As before the cost-grad function is called once for each mini-batch, which is done n/b times per epoch, so the running time gets a factor n/b .

Say the softmax function takes a $n \times d$ matrix in. It runs a loop over the first axis in the matrix ($\mathcal{O}(n)$), runs over the second axis to find "amax" ($\mathcal{O}(d)$), then runs a loop over the second axis ("for d in n") ($\mathcal{O}(d)$), and lastly runs a loop over the second axis again. This this gives $\mathcal{O}(n \cdot 3d) = \mathcal{O}(nd)$.

Cost:

When calculating the cost function we run a loop over the number of data points (batch size) so gets a factor b to the running time. The dotproduct that is first calculated ($\text{dot}(X[i, :], w)$) has time $\mathcal{O}(dK)$. In the same loop it also calculates the softmax of a $1 \times K$ ($\mathcal{O}(K)$) and a dotproduct of two K -length vectors ($\mathcal{O}(K)$). Resulting in $\mathcal{O}(b \cdot d \cdot 3K) = \mathcal{O}(bdK)$.

```
def cost_grad(self, X, y, W):
    """
    Compute the average negative log likelihood cost and the gradient under the softmax model
    using data X, Y and weight matrix W.

    the functions np.log, np.nonzero, np.sum, np.dot (@), may come in handy
    Args:
        X: numpy array shape (n, d) float - the data each row is a data point
        y: numpy array shape (n, ) int - target values in 0,1,...,k-1
        W: numpy array shape (d x K) float - weight matrix
    Returns:
        totalcost: Average Negative Log Likelihood of w
        gradient: The gradient of the average Negative Log Likelihood at w
    """
    cost = 0.
    grad = np.zeros(W.shape)*np.nan
    Yk = one_in_k_encoding(y, self.num_classes) # may help - otherwise you may remove it
    n = len(y)
    for i in range(n):
        innerproduct = np.dot(X[i,:],W)
        sum_element = np.log(np.dot(Yk[i,:],softmax(np.array([innerproduct])))[0])
        cost += sum_element

    cost = (-1/n)*cost

    grad = (-1/n) * np.matmul(np.transpose(X),Yk-softmax(np.matmul(X,W)))

    return cost, grad
```

Figure 5: Cost Grad function for the softmax regression

```
def fit(self, X, Y, W=None, lr=0.01, epochs=10, batch_size=16):
    """
    Run Mini-Batch Gradient Descent on data X,Y to minimize the in sample error (1/n)NLL for softmax regression.
    Printing the performance every epoch is a good idea to see if the algorithm is working

    Args:
        X: numpy array shape (n, d) - the data each row is a data point
        Y: numpy array shape (n, ) int - target labels numbers in {0, 1,..., k-1}
        W: numpy array shape (d x K)
        lr: scalar - initial learning rate
        batchsize: scalar - size of mini-batch
        epochs: scalar - number of iterations through the data to use

    Sets:
        W: numpy array shape (d, K) learned weight vector matrix W
        history: list/numpy array len epochs - value of cost function after every epoch. You know for plotting
    """
    if W is None: W = np.zeros((X.shape[1], self.num_classes))
    history = []
    # if w is None: w = np.zeros(X.shape[1])
    # history = []
    for e in range(epochs):
        data = np.random.permutation(np.insert(X,0,Y,axis=1))
        batch_start, batch_end = 0, batch_size
        while batch_end < len(data):
            batch = data[batch_start:batch_end]
            x_batch = batch[:,1:]
            y_batch = np.array(batch[:,0],dtype=int)
            cost, grad = self.cost_grad(x_batch,y_batch,W)
            W = lr*grad + W*(1-lr) - vars: W = lr*grad
            batch_start += batch_size
            batch_end += batch_size
            print(f"Cost after epoch #{e} is {cost}")
            history.append(cost)

    self.W = W
    self.history = history
```

Figure 6: Fit function for the softmax regression

With the mini-batches and epochs we get in total

$$\mathcal{O}_{\text{Cost}}(dKne) = \mathcal{O}(e)\mathcal{O}(n/b)\mathcal{O}(bdK) \quad (3)$$

Gradient descent:

The gradient descent is calculated in one line. Unpacking this from the inside out (right to left): first we take a matrix product of an $b \times d$ and $d \times K$ matrix ($\mathcal{O}(bdK)$), then taking the softmax of that resulting $b \times K$ matrix ($\mathcal{O}(bK)$) and lastly taking the matrix product of an $d \times b$ matrix and the resulting $b \times K$ of the softmax ($\mathcal{O}(dbK)$). In total we have $\mathcal{O}(d^2b^3K^3)$. With the number of mini-batches and epochs we finally have:

$$\mathcal{O}_{\text{Grad}}(d^2b^2K^3ne) = \mathcal{O}(e)\mathcal{O}(n/b)\mathcal{O}(d^2b^3K^3) \quad (4)$$