Handin 1

« Logistic Regression Softmax Regression »

Rasmus Graversgaard Pedersen, 202006588

Emmanouil Platakis, 202402638

October 9, 2024

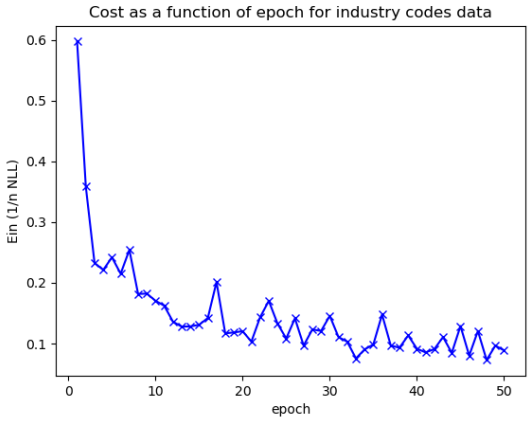
Contents

1 Part I: Logistic Regression

1.1 Code

1.1.1 Summary and Results

Figure 1: Plot of Inner error over time



In Sample Score (accuracy): 0.9587800369685767

Test Score (accuracy): 0.9478935698447893

From the plot we can see that the inner error in the beginning has big improvements, but over time it flattens out. The score also makes sense since we see the test score performing a bit worse than the in sample score.

explaining the plot(s) and comment anything you believe sticks out. Explain if anything does not work.

1.1.2 Actual Code

cost_grad function

```
dot_products = np.dot(X, w)
cost = np.mean(np.log(1+np.exp(-y * dot_products)), axis=0)
grad = -y[:, np.newaxis] * X * logistic(-y * dot_products)[:, np.newaxis]
grad = np.mean(grad, axis=0)
```

fit function

```
i = 0
```

```
while i < epochs:
    permutation = np.random.permutation(X.shape[1])
    X = X[permutation]
    y = y[permutation]

    epoch_cost = 0

    for j in range(int(X.shape[0] / batch_size)):
        to_index = min(j + batch_size, X.shape[0]);
        x_batch = X[j:to_index]
        y_batch = y[j:to_index]

        cost, grad = self.cost_grad(x_batch, y_batch, w)
        epoch_cost += cost
        w = w - lr * grad

    epoch_cost /= int(X.shape[0] / batch_size)
    history.append(epoch_cost)
    i+=1
```

1.2 Theory

Question (i) What is the running time of your mini-batch gradient descent algorithm?

Let:

- n = Number of training samples
- d: dimensionality of training samples
- e: number of epochs run
- b: batch_size for mini_batch_gradient_descent

First step is to calculate the running time of the cost and gradient function.

Running time of dot_products

The running time of the dot_products is $O(b \times d \times 1)$ using the assumption that matrix multiplication running time. Resulting in running time of $O(b \times d)$

Running time of cost

Here we calculate the exponential and log for each sample. So the running time of this step is $O(b)$. The total running time is thus $O(b \times d)$.

Running time of grad

Computing the gradient involves performing element-wise multiplication and summation across the mini-batch, both of which are $O(b \times d)$.

The total running time of cost and gradient is $O(b \times d)$.

Total running time of full mini batch gradient descent

We have $\frac{n}{b}$ batches which each has a running time of $O(b \times d)$. Thus the running time for all the batches in an epoch is $O(\frac{n}{b} \times b \times d) = O(n \times d)$

This is done for e epochs meaning the running time is $O(e \times n \times d)$

Question (ii) **Sanity check** This makes sense since the running time scales with the number of classes represented by the dimension d, data points n and epochs e. So increasing the epoch, class or dimension makes it scale linearly.

Question (iii) **Linearly separable data** When the data is linearly separable, logistic regression's weights grow indefinitely because the model tries to push predicted probabilities towards 0 or 1. This causes the negative log-likelihood to keep decreasing as the model attempts to perfectly classify all points. The process will continue indefinitely, with each gradient step reducing the log-likelihood further, leading to unbounded weight growth.

2 Part II: Softmax

2.1 Code

2.1.1 Summary and Results

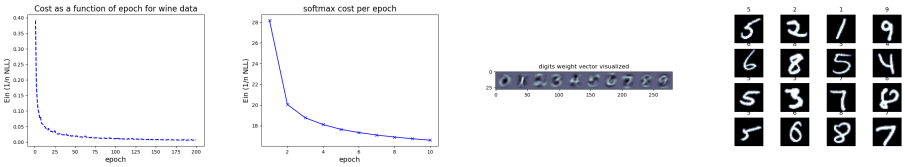


Figure 2: Plots from softmax

Softmax Wine Classifier

In Sample Score (accuracy): 1.0

Test Score (accuracy): 0.978

digits test

In Sample Score (accuracy): 0.92

Test Score (accuracy): 0.92

2.1.2 Actual Code

cost_grad function

```
z = np.dot(X, W)
cost = -np.sum(Yk * np.log(softmax(z) + 1e-12)) / X.shape[0]
grad = np.dot(X.T, (softmax(z) - Yk)) / X.shape[0]
```

fit function

```
n, d = X.shape
for epoch in range(epochs):
    ind = np.random.permutation(n)
    X_sh = X[ind]
    Y_sh = Y[ind]
    epoch_cost = 0
    for i in range(0, n, batch_size):
        X_batch = X_sh[i:min(i + batch_size, n)]
        Y_batch = Y_sh[i:min(i + batch_size, n)]
        cost, grad = self.cost_grad(X_batch, Y_batch, W)
        epoch_cost += cost
        W -= lr * grad
    history.append(epoch_cost/batch_size)
    print(f'Epoch {epoch + 1}/{epochs}, Cost: {epoch_cost}')
```

2.2 Theory

Running time of cost_grad

To analyze the running time of the `cost_grad` function, we need to break down its main operations. The running time of this function depends on several factors. Let **n** be the number of data points, **d** the dimensionality of the feature space, and **K** the number of classes:

Main operations in `cost_grad`:

- **Matrix multiplication $z = XW$:**
  - This matrix multiplication has a time complexity of $O(ndK)$, as it involves multiplying an $n \times d$ matrix with a $d \times K$ matrix.
- **Softmax computation:**
  - The softmax function is applied to each row of z to get class probabilities.
  - For each data point, computing softmax involves $O(K)$ operations. Since there are n data points, the overall complexity is $O(nK)$.
- **Cost computation:**
  - The cost (negative log-likelihood) is computed by summing over all data points and classes, which involves computing $Y_k \cdot \log(\text{softmax}(z))$ for every class K across n data points, so again $O(nK)$ complexity.
- **Gradient computation:**
  - The gradient of the cost function is computed as $\nabla W = X^T(\text{softmax}(z) - Y_k)$, where $X^T \in \mathbb{R}^{d \times n}$ and $(\text{softmax}(z) - Y_k) \in \mathbb{R}^{n \times K}$. So it is another matrix multiplication but with $d \times n$ and $n \times k$ matrices, so $O(ndK)$ complexity.

Total time complexity of `cost_grad` is the dominant term, thus $O(ndK)$.

This means that the `cost_grad` function scales linearly with the number of data points n, the number of features d, and the number of classes K.