

HANDIN 1

Machine Learning Q1/Q2 2014
Group Whiner

Morten Nygaard
2011 4582

Jonas Hovmand
2011 3884

Kasper Eenberg
2011 7679

September 23, 2014

Contents

Contents	i
1 Introduction	1
2 Logistic Regression	1
2.1 Gradient Descent and OCS	1
2.1.1 logCost	1
2.1.2 logRun	1
2.1.3 Results	2
2.1.4 The AU digit dataset	2
2.1.4.1 Getting around digit size	3
2.2 Sanity Check	3
3 Softmax Regression	3
3.1 Calculation	3
3.2 Results	4
A Figures	6

1 Introduction

In this exercise we implemented classifiers for optical character recognition, referred to as OCR. This was done by two different methods, *logistic regression* and *multinomial/softmax regression*.

To verify the implementations of the two different regressions, we were given the training sets *mnistTrain.mat*, *auTrains.mat*, and the test set *mnist-Test.mat*. The training sets were used to train on and the test set was used to test on.

As we did not have any test data for the *au* data, we did *k-fold cross validation* on the *au* dataset, as well as a few experiments on classifying by using the classifiers trained on the *mnistTrain.mat* data. This way we could verify that the implementations worked as expected on this data as well.

Our final classifiers for *au* data is built from all the training data, due to the fact that more training data improves the classifications we are able to give.

2 Logistic Regression

2.1 Gradient Descent and OCS

2.1.1 logCost

The file *logCost.m* contains the function `logCost(X, y, θ)`, which calculates the cost function E_{in} and the gradient ∇E_{in} . The input dimensions are $X \subseteq \mathbb{R}^{n \times d}$, $y \subseteq \mathbb{R}^{n \times 1}$ and $\theta \subseteq \mathbb{R}^{d \times 1}$.

The cost function was calculated using matrix multiplications with the formula

$$NLL(D | \theta) = - \sum (y \cdot \log(\sigma(X\theta)) + (1 - y) \cdot \log(1 - \sigma(X\theta))) \quad (1)$$

where σ is the logistic function

$$\sigma(z) = 1 / (1 + e^{-z}), \quad (2)$$

and the operator “.” denotes element by element multiplication.

The gradient ∇E_{in} was calculated using the formula

$$\nabla NLL(\theta) = - \frac{X^T(Y - \sigma(X\theta))}{\|X^T(Y - \sigma(X\theta))\|} \quad (3)$$

We discovered that the values of the gradient grew extremely large, this lead to θ having really large values in later iterations which leads to NaN's in the logistic function σ . This was fixed by normalizing $\nabla NLL(\theta)$ to unit length.

2.1.2 logRun

In *logRun.m* file the function `logRun(X, y)` was defined using the same dimensions as above.

We start by adding a bias column of ones to the matrix X and a bias row of value zero to the vector θ . Next we decided on a learning rate and the

number of iterations to run, experimentation showed that learning rates of 0.01, 0.05 and 0.1 made sense.

The rest of the algorithm was basically defined the same way as for the example for linear regression, except that the algorithm used `logCost()` to calculate E_{in} and ∇E_{in} . We did not implement any line search algorithm, but instead just used ∇E_{in} multiplied with the learning rate to do our gradient descent.

Now that we had implemented gradient descent, we could actually run the algorithm on the data and express how well the algorithm recognized the characters.

2.1.3 Results

The functions described above were used to test our OCR capabilities on the *Mnist* dataset. The main experimentation at this stage was changing the learning rate and the amount of iterations we ran.

As is visible on [Figure 1](#), the learning rate affected the convergence speed towards the optimal recognition, which turned out to be 90.62% with a learning rate of 0.1 at iteration 133.

In [Figure 1](#) it is obvious that the recognized digits zig-zag jump up and down from one iteration to the next. This might be a side effect of not directly optimizing for the amount of recognized digits, but instead having $NLL(D | \theta)$ as a cost function. This cost function can be verified in [Figure 2](#) to be, more or less, constantly decreasing.

A plot of the θ 's as we progress through the iterations can be seen in [Figure 3](#). These θ 's were reshaped into a 28×28 pixel image, the same size as our input.

A plot of the original array actually gives some more meaning to what we are seeing. As can be seen, the variables θ_i are in the range $-1 < \theta_i < 1$. The [Figure 3](#) was generated by rescaling $(\theta + \min(\theta))$ to values between 0 and 1, which means that 0's are represented as a grey colour, black are the negative values and white are the positive values.

It is somewhat obvious that they look reminiscent of the digits they were actually supposed to represent. We can interpret these thetas as vectors that attempt to maximize the value of $x\theta$ (a single digit) for the digit they are classifying, and minimizing otherwise.

2.1.4 The AU digit dataset

Testing show that you cannot directly use the classifiers trained on the *mnist* dataset to classify the AU digits dataset. The results from attempting this are only slightly better than taking random guesses. The reason for this is that, when plotted, the AU digits are smaller and do not occupy as much space as the *mnist* digits in the 28×28 matrix.

With k-fold cross validation we divided the AU dataset into 5 and 10 parts, and trained on 4(9) while testing on 1. The data from the tests are visible below, and the correct classifications are around 84.5%.

10 Parts

Test Set	%	Count
1	82.78	495/598
2	86.79	519/598
3	84.28	504/598
4	84.78	507/598
5	85.12	509/598
6	84.62	506/598
7	83.44	499/598
8	86.79	519/598
9	86.12	515/598
10	83.61	500/598

5 Parts

Test Set	%	Count
1	84.28	1008/1196
2	84.11	1006/1196
3	85.20	1019/1196
4	84.62	1012/1196
5	84.45	1010/1196

2.1.4.1 Getting around digit size

In an attempt to get around digit size we implemented a function `maxImage`, that removes all the blank rows and columns, and scales the remaining image up to be 28×28 .

This means that we could now train on the rescaled *mnist* dataset, and classify on the AU dataset.

2.2 Sanity Check

We generated a normally distributed vector of length d with center 0.5 and standard deviation 0.1, and multiplied every image vector with it.

Before testing applying random noise to the input, we expected the amount of correct classifications to be approximately the same. This based on the assumption that the same noise added to every image would, by the intuition we used above for θ , lessen the impact of the random noise so they don't towards the original sum quite as much.

However we were wrong in our prediction, as can be seen in [Figure 4](#). As it clearly shows, classifiers trained on the noisy input data consistently underperform compared to classifiers trained on proper data. It can be argued, though, that this is still a decent classification level.

3 Softmax Regression

3.1 Calculation

The `softmax` implementation was written to make as much use of matrix multiplications as possible. The following dimensions are used:

- $X \subseteq \mathbb{R}^{n \times d}$
- $Y \subseteq \mathbb{R}^{n \times k}$
- $\theta \subseteq \mathbb{R}^{d \times k}$
- $k = t$ as the amount of digits.

Firstly $X \cdot \theta$ is calculated, then each row is subtracted the maximal value c that is found in each row, using matrix calculations in Matlab. Each value in each row is then exponentiated, and the sum of each row is calculated. This gives us a $n \times 1$ column vector on which each row has the logarithm taken, and is added c . The calculations above gives us

$$S = \log \left(\sum_{i=1}^k e^{x_i} \right) = c + \log \left(\sum_{i=1}^k e^{x_i - c} \right). \quad (4)$$

Where we define S_j as the value S value for the j 'th row. Then we for each entry $x_{i,j}$ in X calculate $\exp(x_{i,j} - S_j)$. This gives the softmax calculation for the entire input X . We define this as $\text{softmax}(X)$.

To calculate the cost, we calculate the element by element multiplication of $\text{softmax}(X)$ and Y , which by then definition of Y gives us a $n \times k$ matrix, with only one non-zero entry in each row. This non-zero entry is

$$\sum_{j=1}^k [y_j = 1] (\text{softmax}(\theta^T x)_j), \quad (5)$$

after which we take the logarithm of this value, and sum all the rows to find the cost.

The gradient is calculated with

$$- \left(\frac{X^T (Y - \text{softmax}(X\theta))}{\|X^T (Y - \text{softmax}(X\theta))\|} \right), \quad (6)$$

to get a gradient of unit length.

3.2 Results

Using a learning rate of 0.1 and up to 300 iterations, we get a maximal matching of 9102 correct out of 10000.

When training on the scaled *mnist* digits, we were able to achieve a maximum of 93.57% correctly classified. A plot of progressions can be seen in [Figure 5](#).

Trying to classify the scaled AU digits with the softmax classifiers trained on the scaled *mnist* dataset results in 3884 of 5982, which is around 65% accuracy.

As can be seen below, k-fold validation results with softmax regression is approximately equal to logistic regression.

10 Parts

Test Set	%	Count
1	84.28	504/598
2	87.46	523/598
3	84.78	507/598
4	84.45	505/598
5	84.62	506/598
6	85.12	509/598
7	84.62	506/598
8	87.63	524/598
9	86.45	517/598
10	82.44	493/598

5 Parts

Test Set	%	Count
1	85.37	1021/1196
2	84.95	1016/1196
3	85.54	1023/1196
4	85.62	1024/1196
5	84.95	1016/1196

A Figures

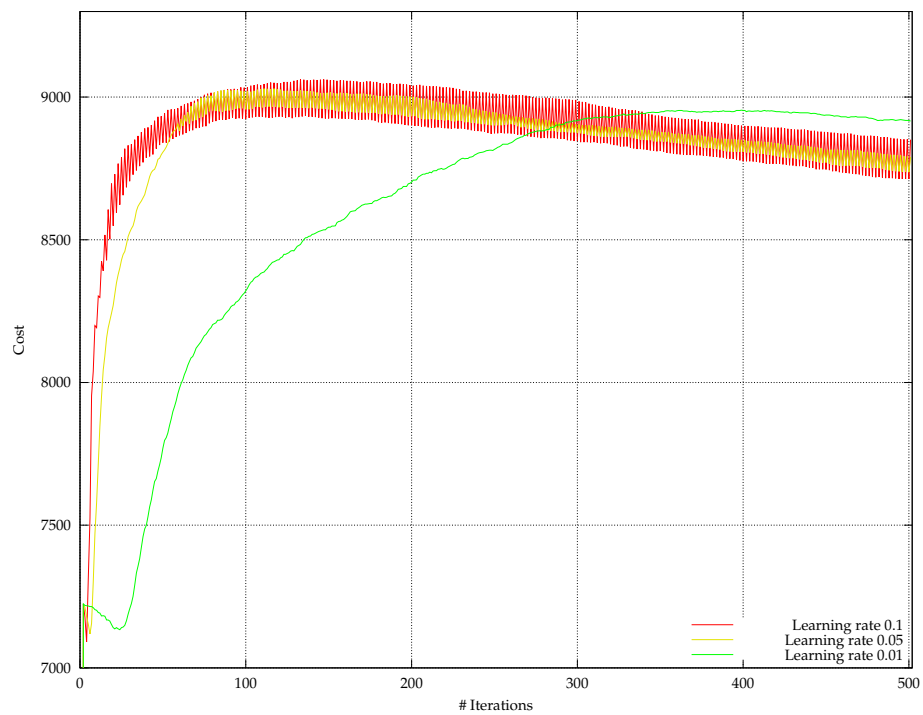


Figure 1: Plot of characters recognized per iteration, divided by learning rate.

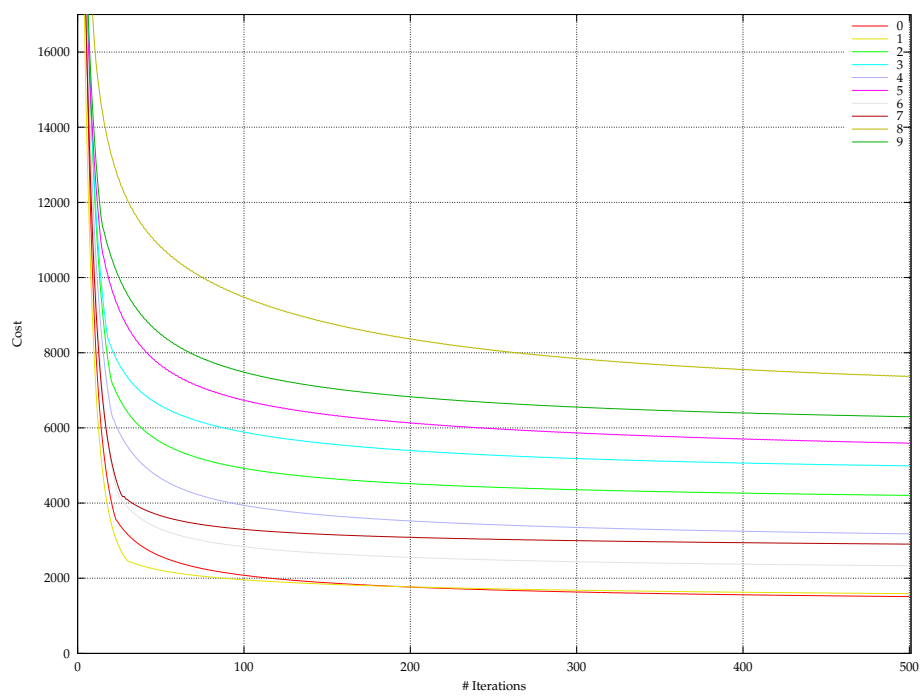


Figure 2: Plot of cost progression for each digit, and learning rate 0.1.

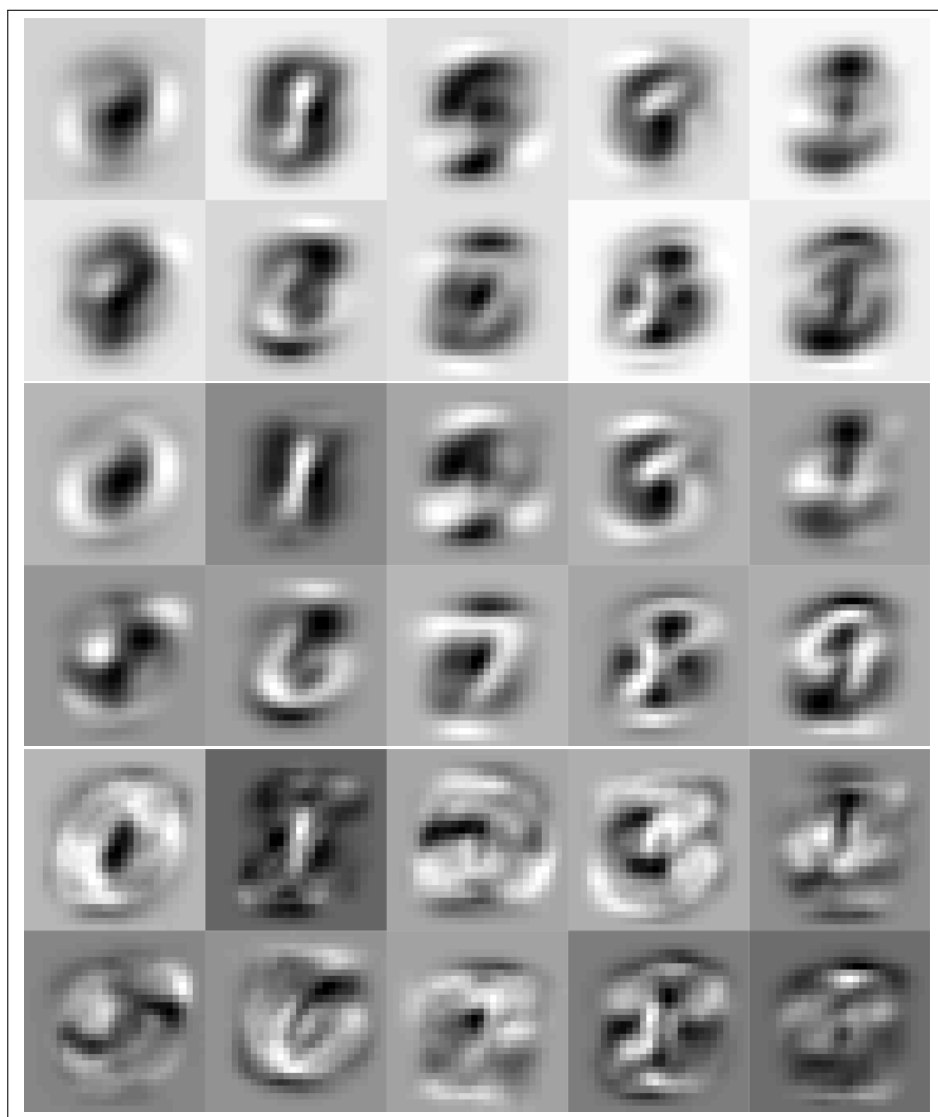


Figure 3: Plot of θ s for the best classifier found using logistic regression after, from top 5, 10 and 133 iterations.

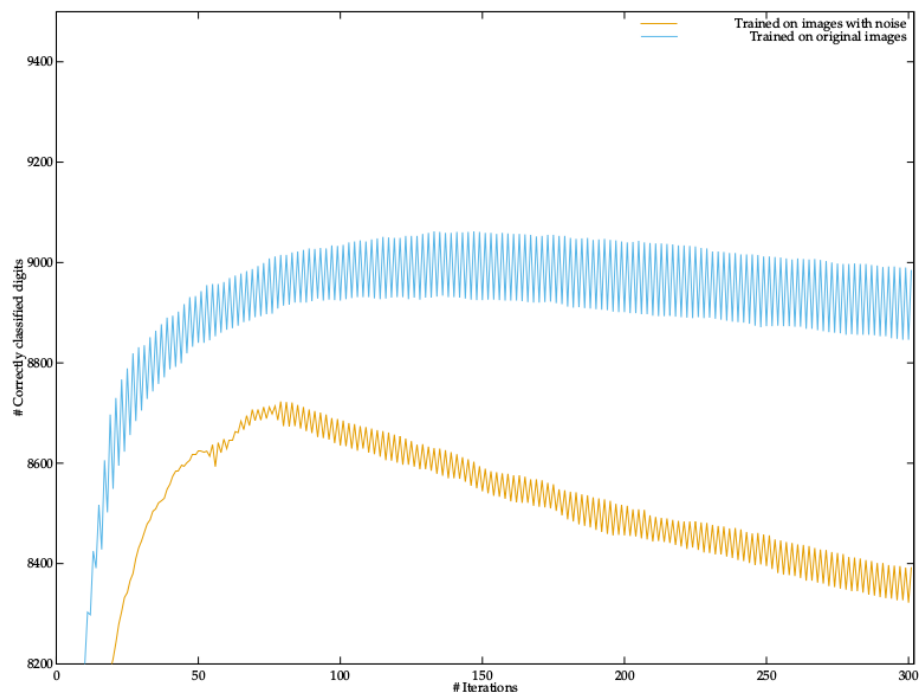


Figure 4: Correct predictions with θ trained on noisy images. Both trained with learning rate 0.1.

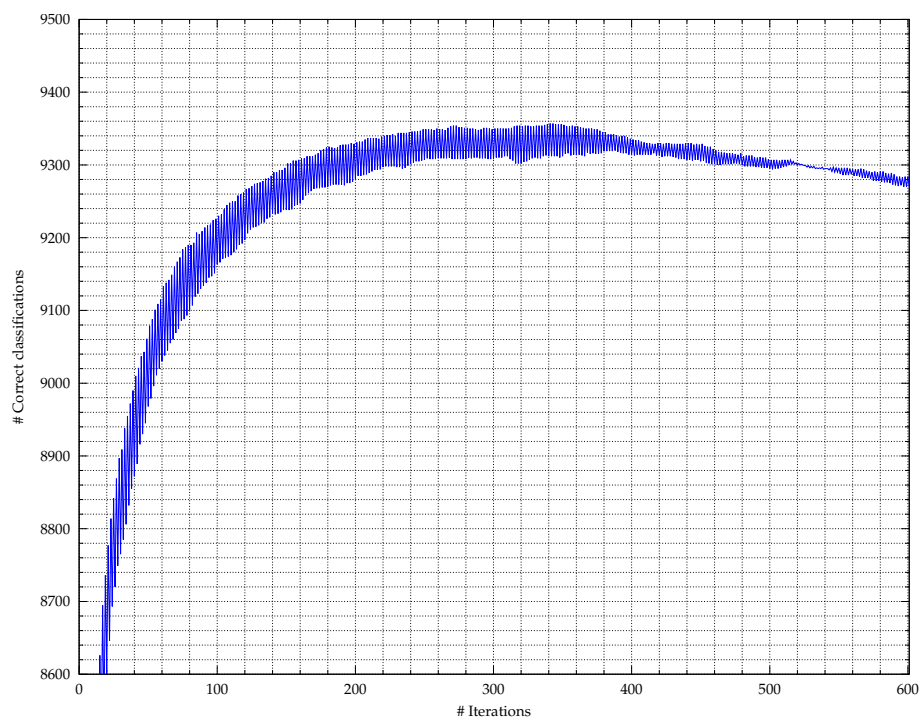


Figure 5: Correct predictions using softmax regression of classifiers trained with images scaled to fill the entire 28×28 image.