ECE421

Assignment 1:

Linear and Logistic Regression

Alexander APOSTOLOV and Anthony Kemmeugne

# 1.Linear Regression

Recall: We have the Loss function:

L=

## 1.1.Loss function and gradient

We first calculate the derivative of L with respect to dwi

So, the gradient according to the weights is:

Then we calculate the derivative with respect to the bias:

Here is our implementation in Python for the gradient:

def gradMSE(W, b, x, y, reg):

pre\_prediction = np.matmul(x, W)

prediction = np.add(pre\_prediction, b)

e = np.subtract(prediction,y)

regTerm = np.multiply(W, reg)

tempor = np.matmul(np.transpose(x), e)

scaled = np.multiply(tempor, 1/N)

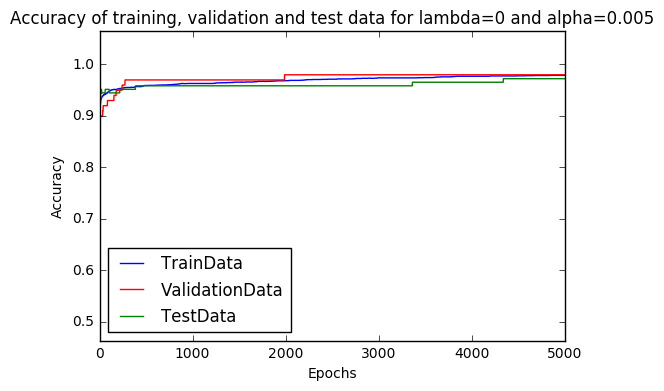
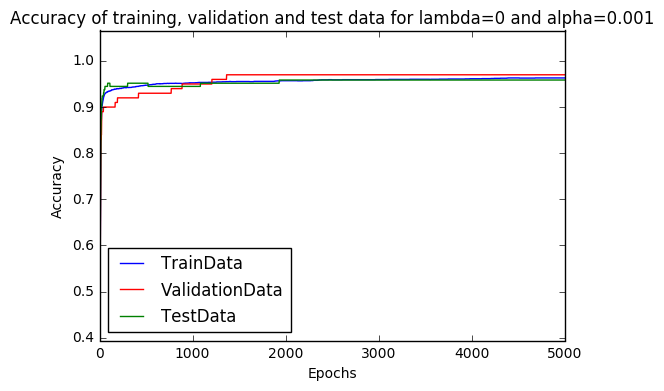
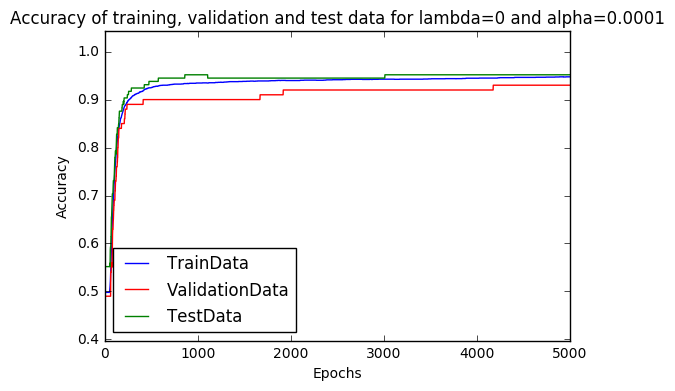
gradW = np.add(scaled, regTerm)

gradb=(np.sum(e))/N

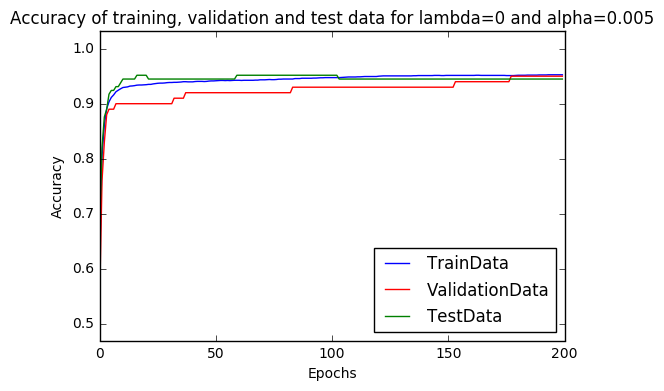
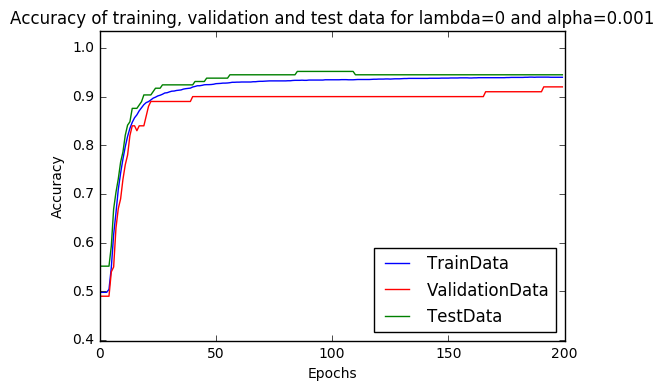
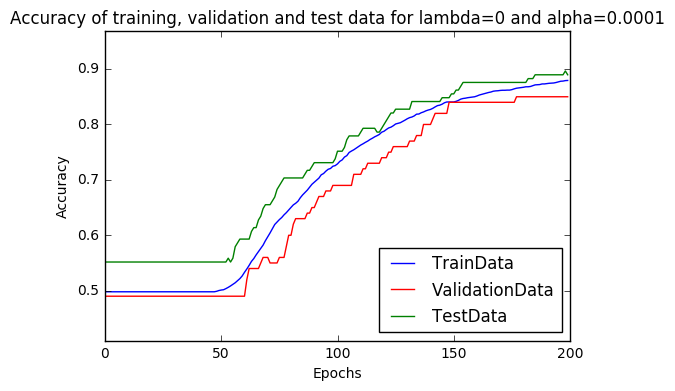
return gradW, gradb

## 1.3. Tuning the learning rate

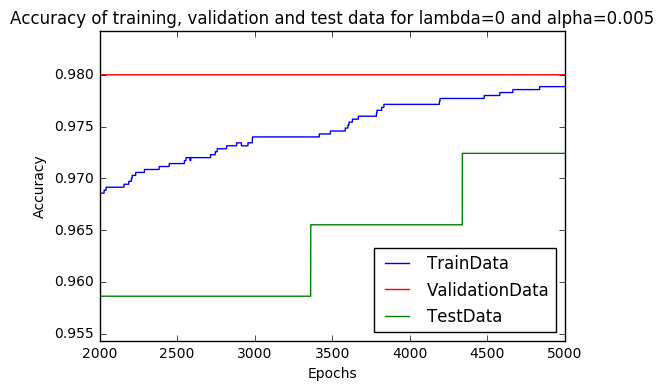
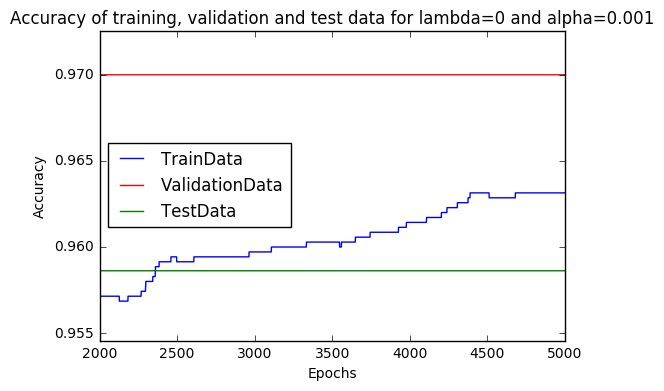
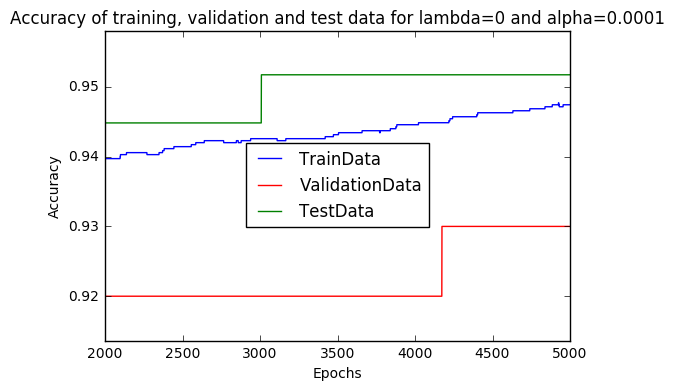
We have tested the implementation of Gradient Descent with 5000 epochs and λ=0 and here are the results. We first investigate the accuracy, the accuracy is determined as the portion of correctly predicted data points, we say that a prediction of less than 0.5 is 0 and more than that is 1.



We see that the three choices of alpha have similar curves for accuracy and that the accuracy of the training data is increasing in a continuous manner whereas the accuracies for validation and test data increases in a jumpy manner. This is mainly due to the fact that there are not a lot of datapoints in the training set and the validation set, so whenever a new data point changes its prediction, the change will be visible in the graph. We have taken only the first 200 epochs of these graphs to analyse the learning curve in the beginning.

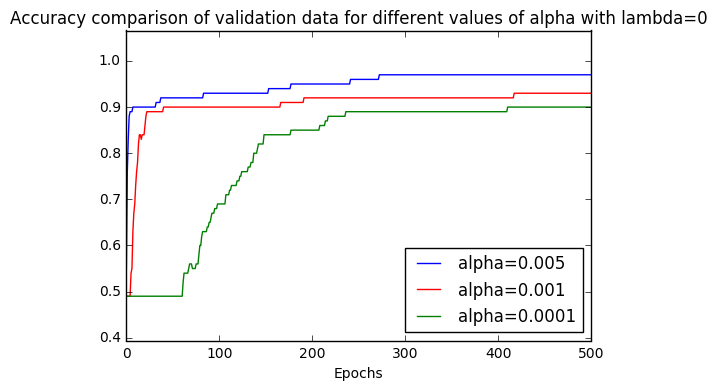
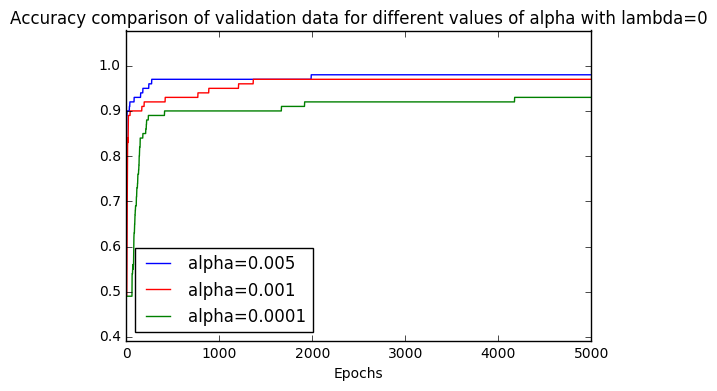
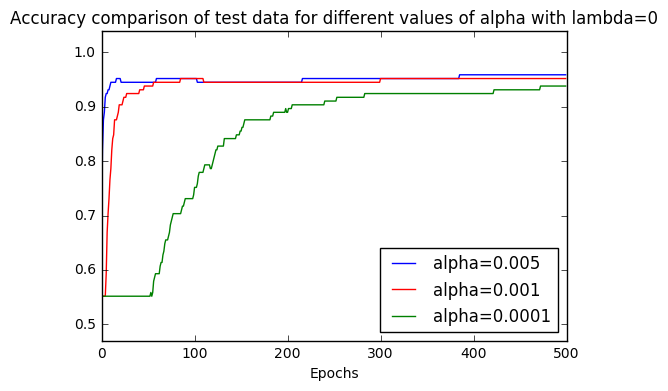
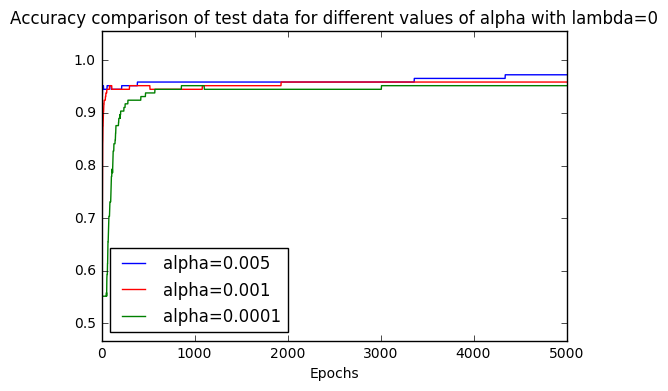
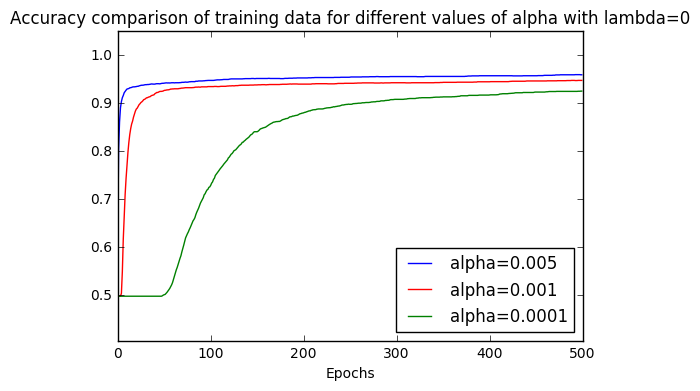
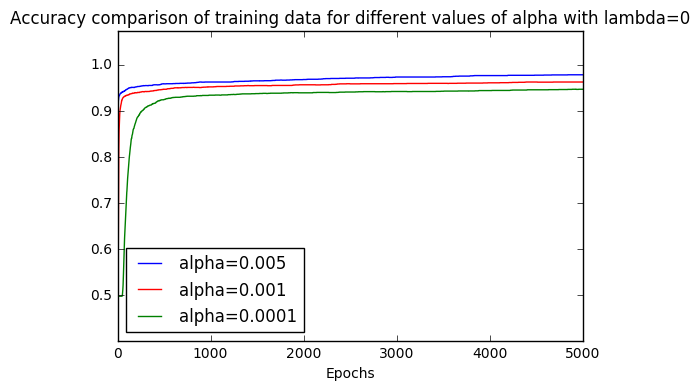


First, we see that the initial value is always close to 0.5, which is what one would expect since in the beginning the weights are all set to zeros, so the prediction is random. We then notice that, the bigger alpha is the faster the curve will grow. This is also what is expected since alpha is the factor that determines how much we are updating the weights. We have also take a look at the last 2000 epochs:

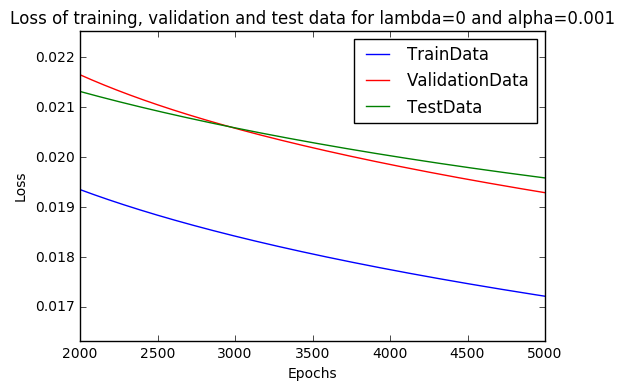
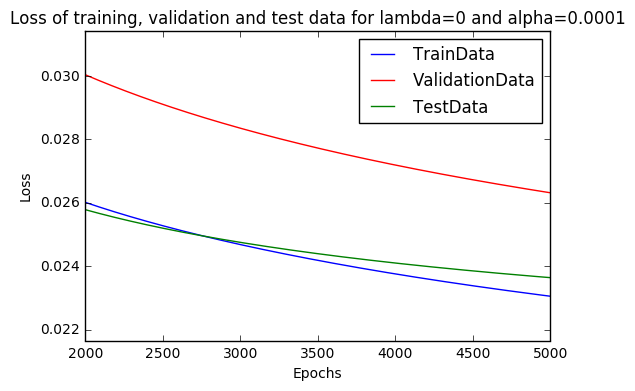
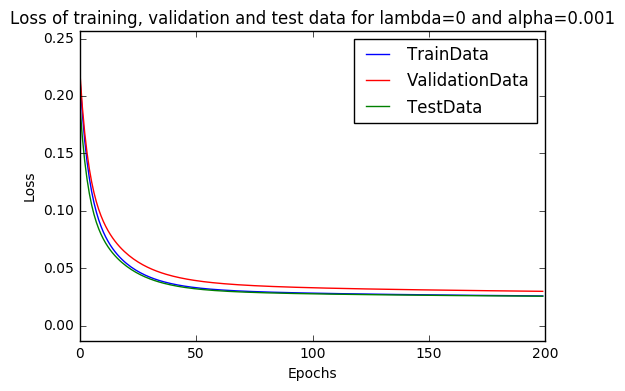
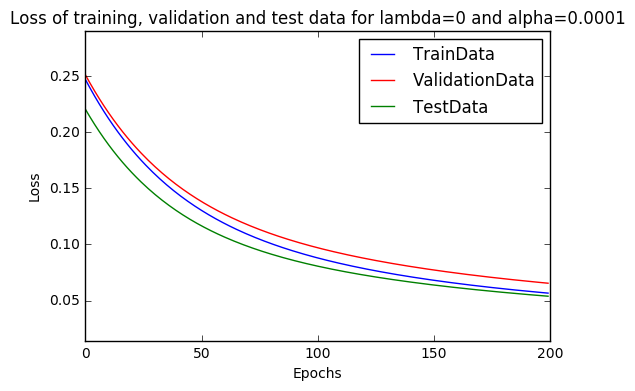
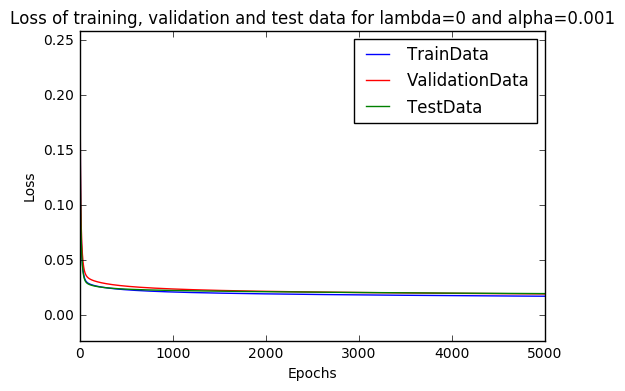
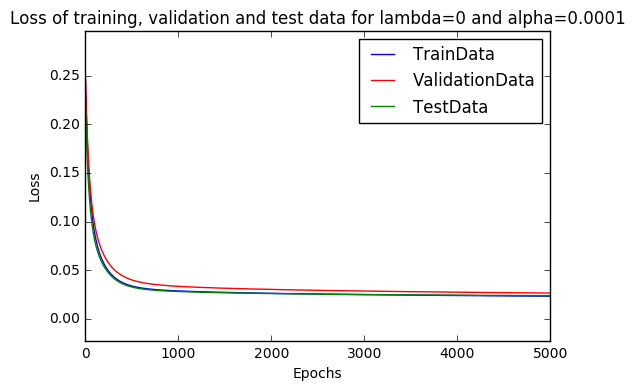


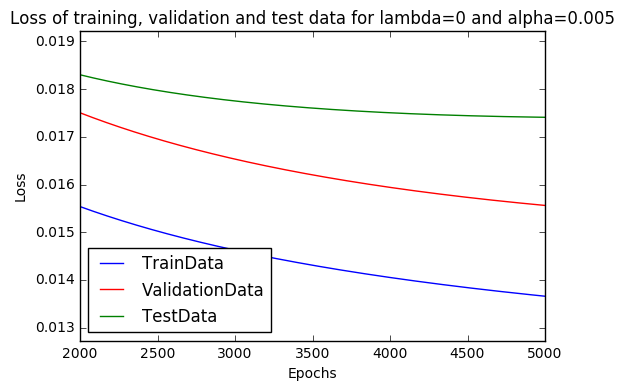
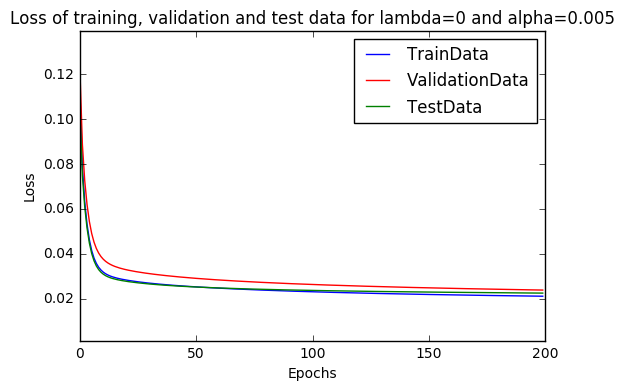
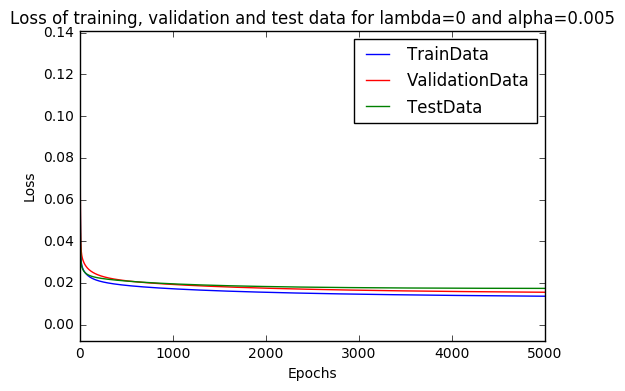
We notice that in the three scenarios the accuracy for the training, validation and test data sets is getting really close. Also, we see that the bigger alpha is the bigger the final accuracy is. This might be explained by the fact that a bigger alpha makes the convergence towards the minimum of the loss function faster.

We also see these same conclusions on the following graphs, which show the accuracy on the same data set with different values of alpha:



We then took a look at the evolution of the loss during the learning process:





We first observe that the shape of the curve is similar. However, the bigger alpha is the faster the curve will go down. This is what we would expect since alpha determines the speed at which the weights are updated. We also see that the final loss is the smallest when alpha is the biggest, the same explanation as before can be applied here.

Here is a table that summarizes the final accuracies in the different scenarios:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Training data | Validation data | Test data |
| α = 0.0001 | 0.9474 | 0.93 | 0.9517 |
| α = 0.001 | 0.9631 | 0.97 | 0.9586 |
| α = 0.005 | 0.9789 | 0.98 | 0.9724 |

And here is the same table with the final losses:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Training data | Validation data | Test data |
| α = 0.0001 | 0.0231 | 0.0263 | 0.0236 |
| α = 0.001 | 0.0172 | 0.0193 | 0.0196 |
| α = 0.005 | 0.0137 | 0.0156 | 0.0174 |

We have also conducted measures to see what is the execution time of the implementation to reach a mean square error of 0.017:

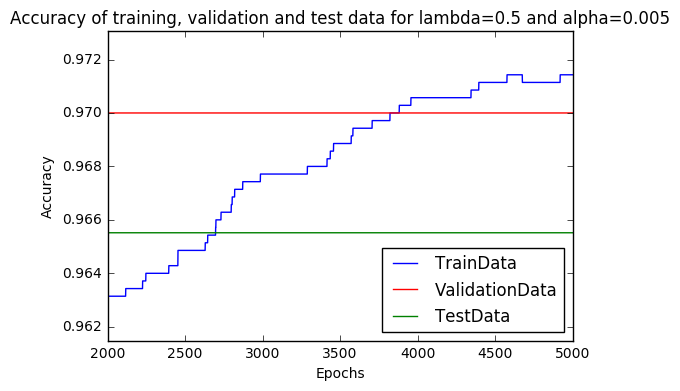
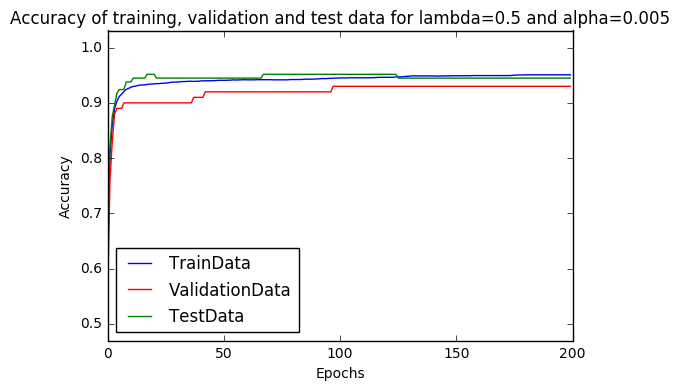
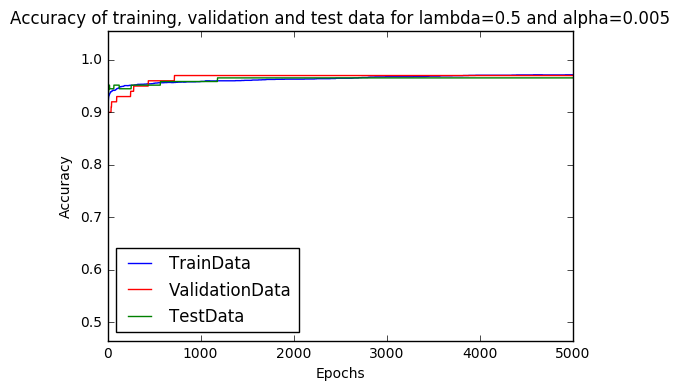
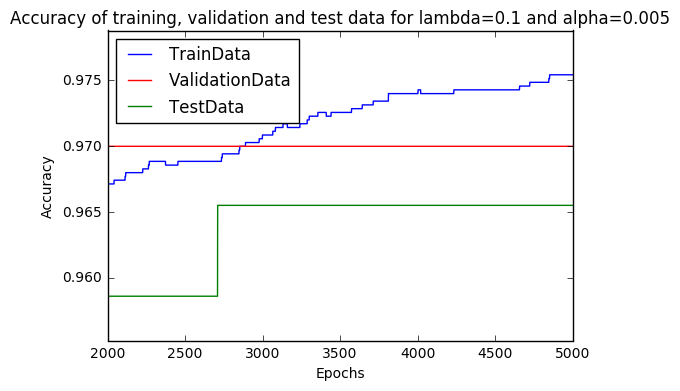
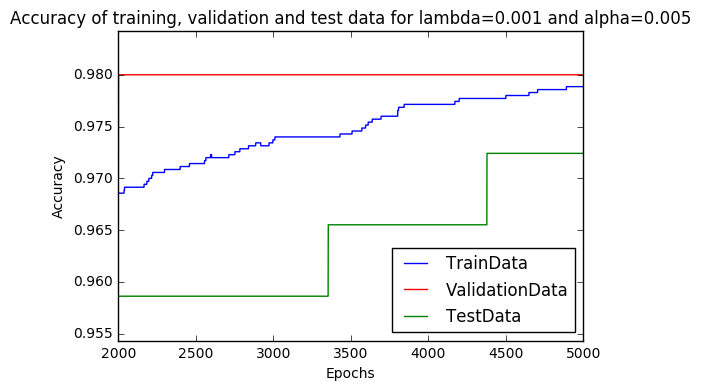
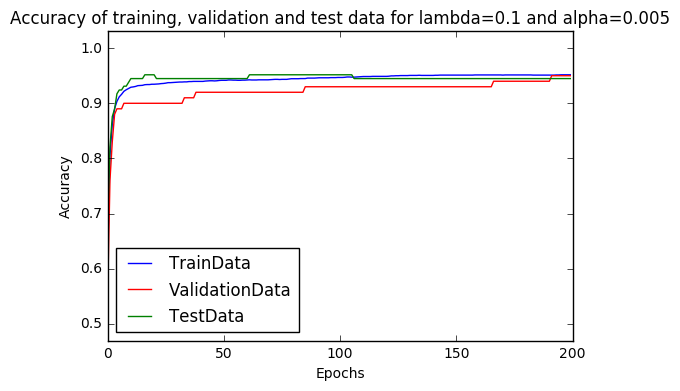
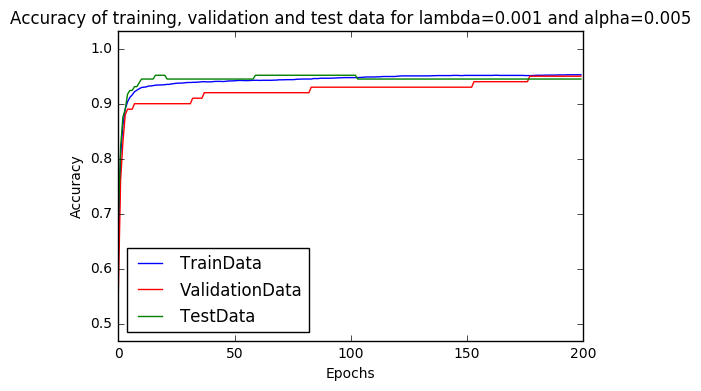
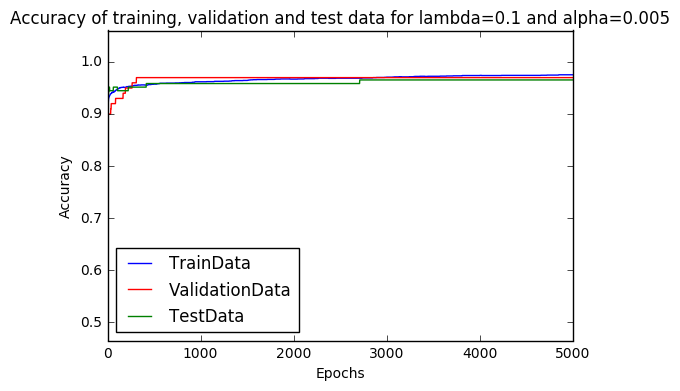
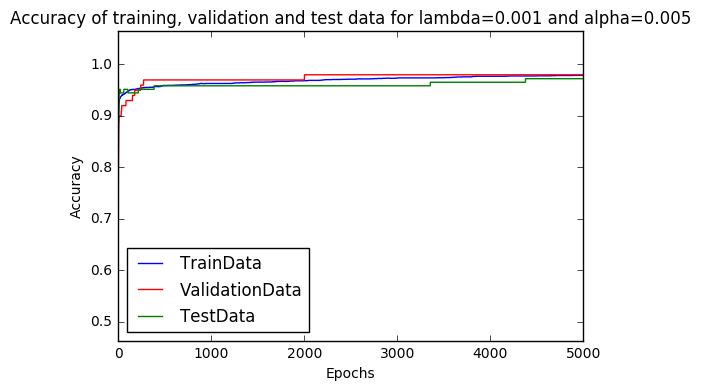
|  |  |  |
| --- | --- | --- |
|  | Execution time | Number of iterations |
| α = 0.0001 | 245,74s | 54’697 |
| α = 0.001 | 24.59s | 5’470 |
| α = 0.005 | 4.89s | 1’094 |

We see that for a give MSE, the number of iterations and thus the execution time increases linearly as alpha decreases.

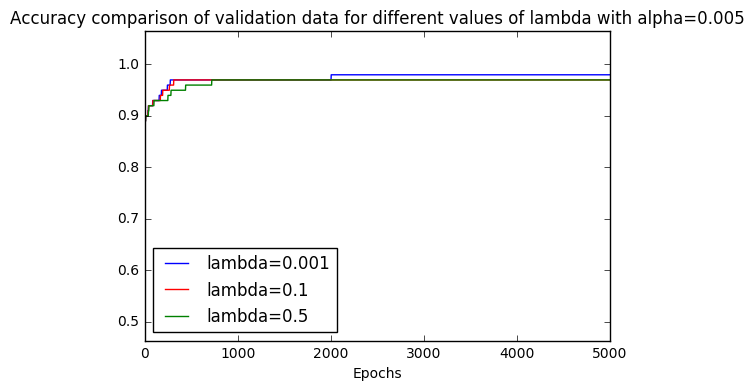
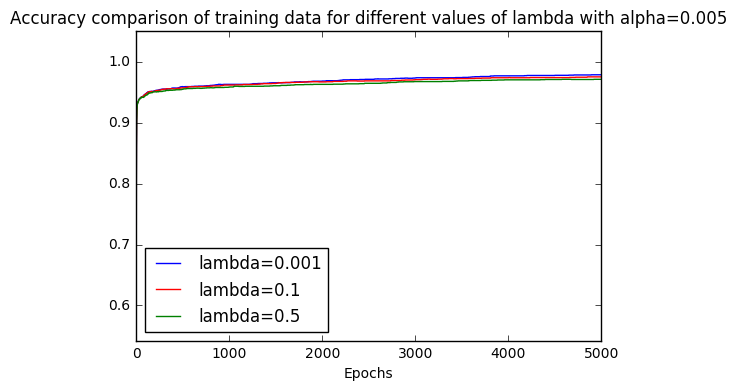
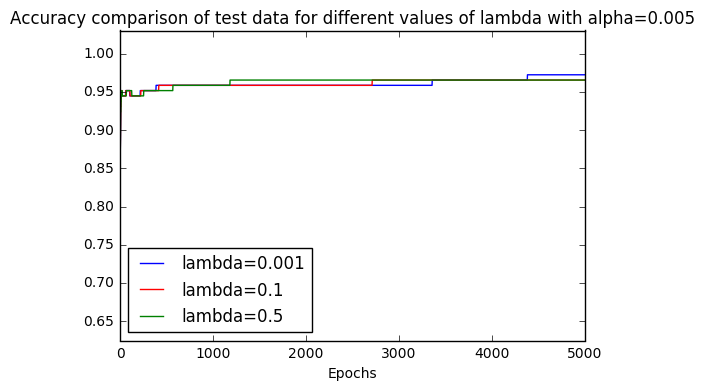
## 1.3.Generalization

We know investigate the impact of the regularization constant λ, for 5000 epochs and a fixed alpha of 0.005. The values for λ are {0.001, 0.1, 0.5}.

We first investigate the accuracy in the different scenarios.

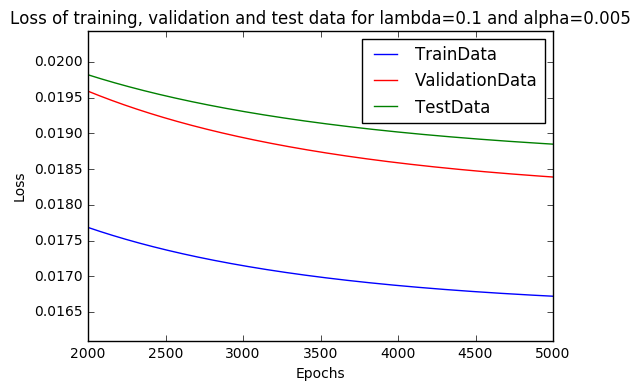
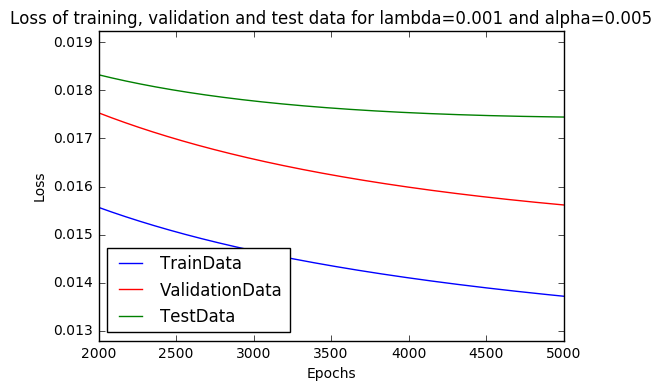
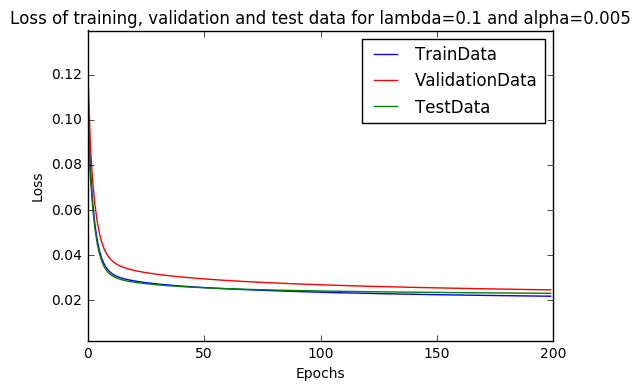
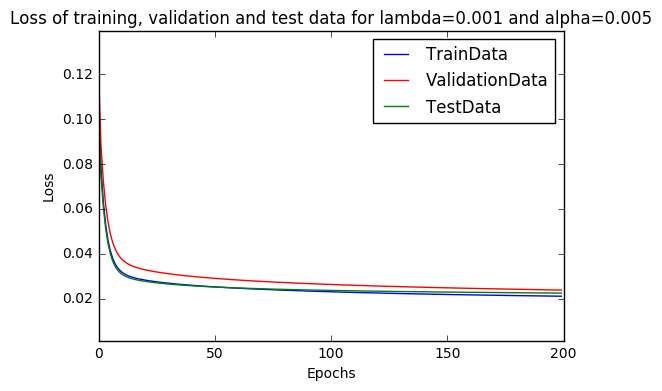
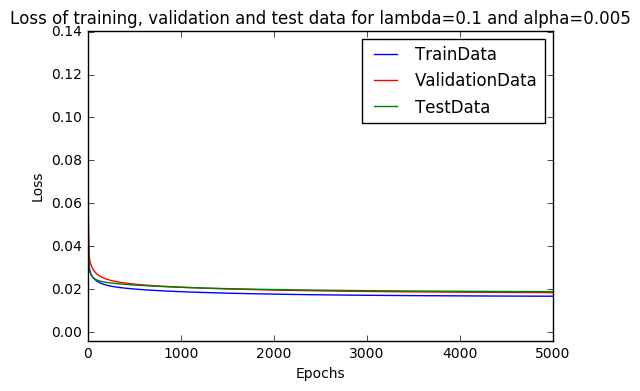
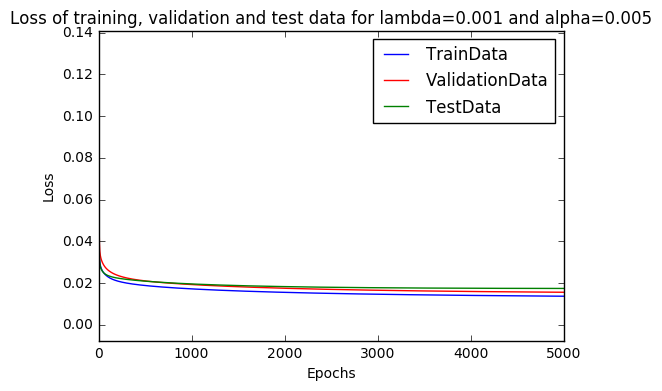


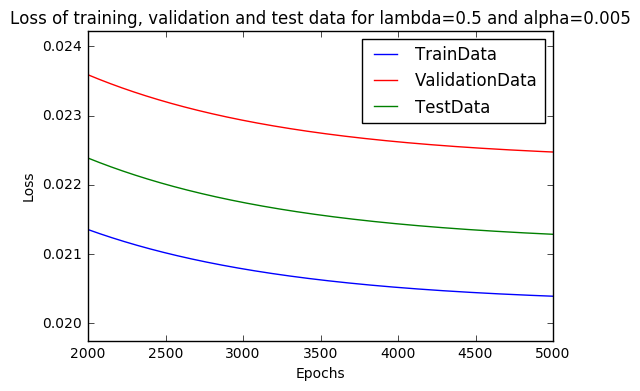
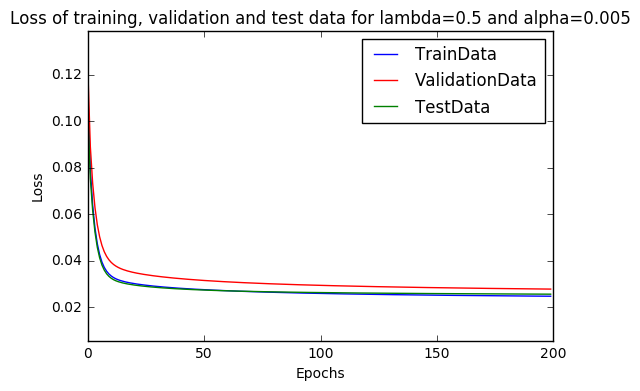
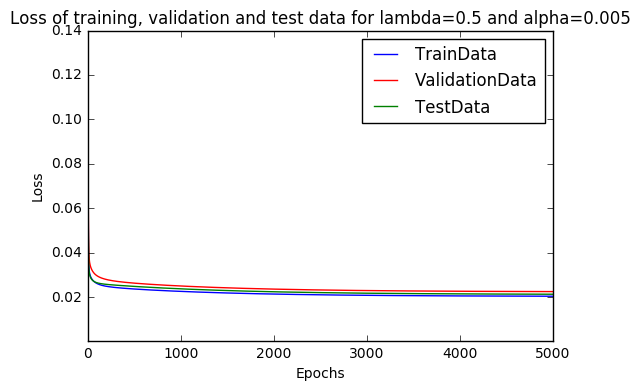
And here are three plots which compare the accuracy of the same dataset over the three different values of λ:



We first observe that the curves are really similar. They all get quite high quite fast and at the same rate. However, we see that over the end, when λ is smaller, the accuracy is slightly better. We also see that for a given λ, the accuracy over the three datasets is really close.

We then studied the loss function in the different possible scenarios.





We first notice that the curves are very similar for different values of λ. We see that the training set after some iterations, has the smallest loss value, which is what one would expect since we are optimizing the weights over this set. We also notice that the loss is smaller when λ is smaller.

Here is a table that shows the final accuracies in every scenario:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Training data | Validation data | Test data |
| λ = 0.001 | 0.9789 | 0.98 | 0.9724 |
| λ = 0.1 | 0.9754 | 0.97 | 0.9655 |
| λ = 0.5 | 0.9714 | 0.97 | 0.9655 |

And here ae the final losses:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Training data | Validation data | Test data |
| λ = 0.001 | 0.0137 | 0.0156 | 0.0174 |
| λ = 0.1 | 0.0167 | 0.0184 | 0.0188 |
| λ = 0.5 | 0.0204 | 0.0225 | 0.0213 |

We also took measures to check how much time is needed to reach a MSE of 0.017.

|  |  |  |
| --- | --- | --- |
|  | Execution time | Number of iterations |
| λ = 0.001 | 4.54s | 1’102 |
| λ = 0.1 | 14.54s | 3’454 |
| λ = 0.5 | Not reached after an hour |  |

So, we see that when λ is smaller, we reach better results faster. We also notice that when λ is 0.001, the results are the same as when it is equal to zero. So from this we can conclude that for the regularization factor to make a difference it has to be bigger than 0.001.

The reason why we want to use λ, the regularization parameter is to avoid overfitting. Indeed, the data we have might be noisy, so if we try to fit our weights perfectly to the data, we might end up with a prediction that is equal to the training data points but overshoots and undershoots around them. We introduce this term to penalize big weights, which are related to overfitting scenarios.

So even though, smaller λ’s converge faster, we have to make a trade off between convergence time and preciseness and avoiding overfitting.

## 1.5 Comparing Batch GD with normal equation

The normal equation optimizes the tuning of the weights on the training data and can be computed as follow:

Using the normal equation, calculating the weights is done in an average of 0.16 s and here are the achieved performances:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Training data | Validation data | Test data |
| Accuracy | 0.9843 | 0.94 | 0.9172 |
| Loss | 0.0116 | 0.0301 | 0.9172 |

And here is a table with the same information for the best results we have seen over all scenarios in batch GD:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Training data | Validation data | Test data |
| Accuracy | 0.9784 | 0.98 | 0.9724 |
| Loss | 0.0137 | 0.0156 | 0.0174 |

With the best choice of α and λ, we reach the same accuracy over the training set in 35.87s (12’344 iterations), so for the same performance the normal equation is over 200 times faster. Also, we see that the normal equation gives weights which are better than batch GD for the training set, but worse than batch GD for the validation and test set. This happens because the normal equation finds the optimal weights to fit to the training data, but in doing that it can overfit the weights to the training data and thus the prediction for the rest of the points will be incorrect.

# II. Logistic regression

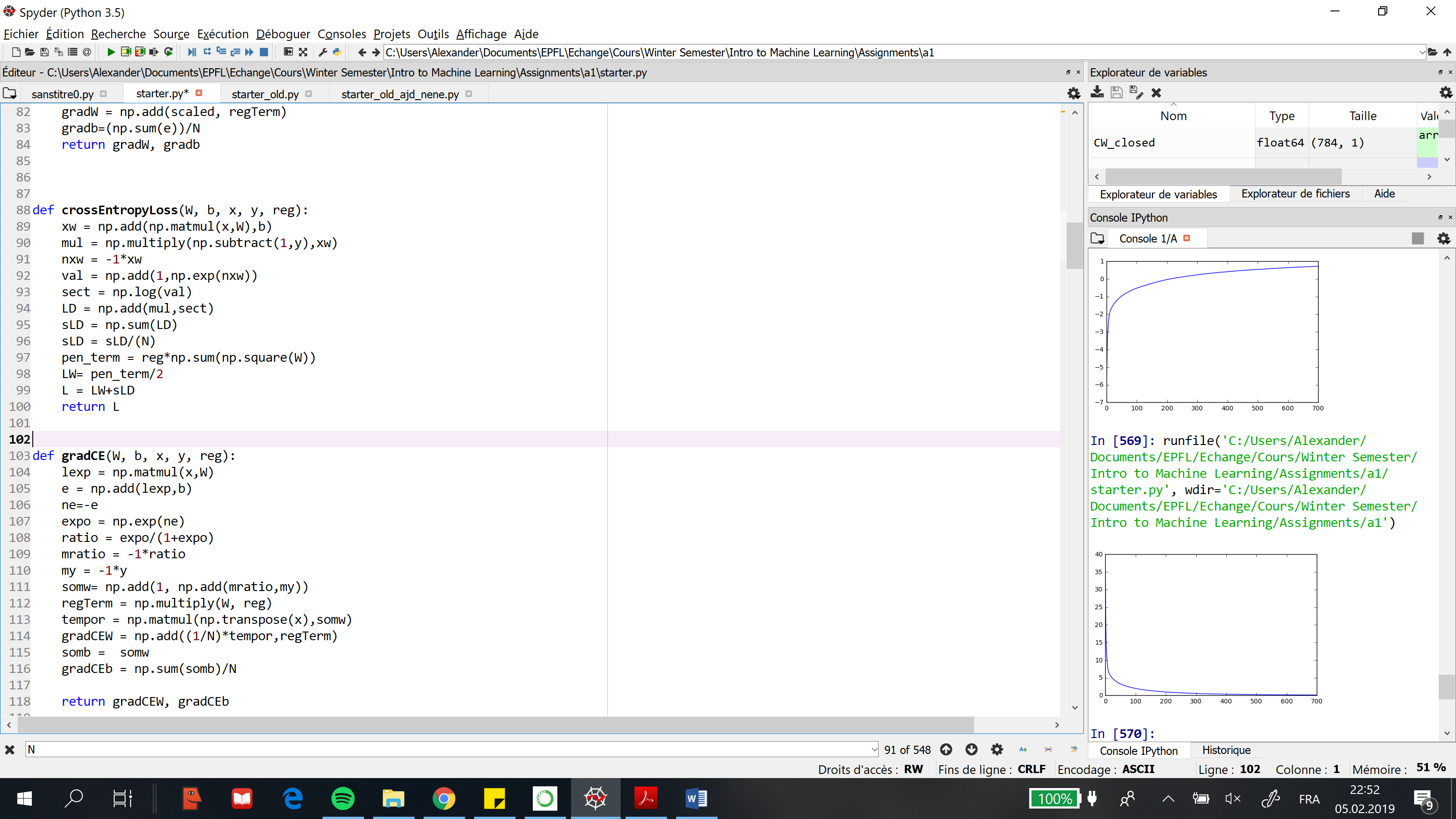
## II.1 Loss Function and Gradient

The Loss function is:

The derivative according to the weight is:

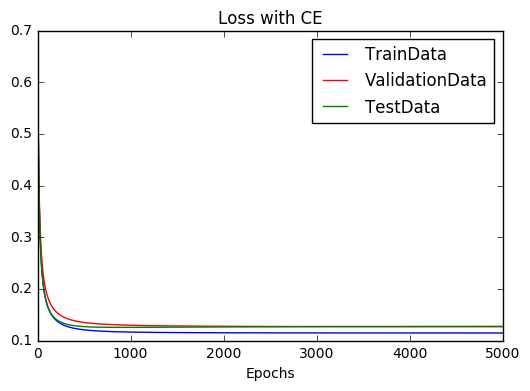
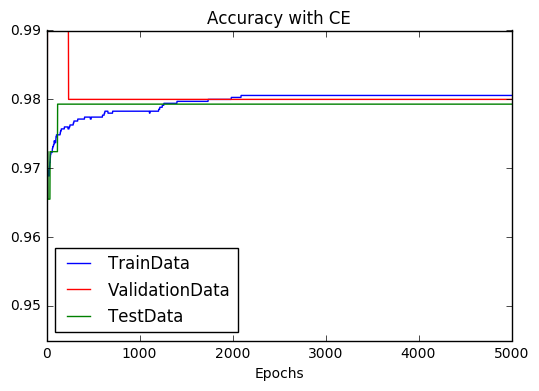
And the derivative according to the bias is:

Here is a snippet of our Python code implementing this:



## II.2 Learning

We have plotted the Accuracy and Loss over the learning process:



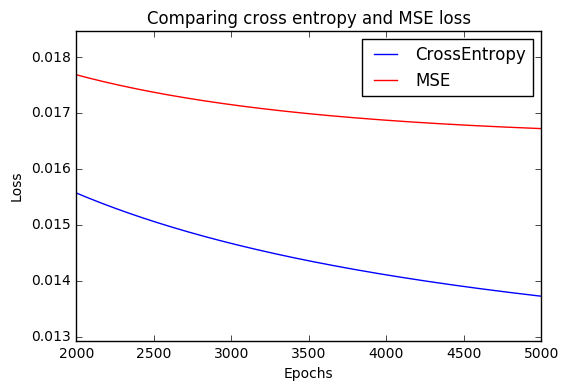
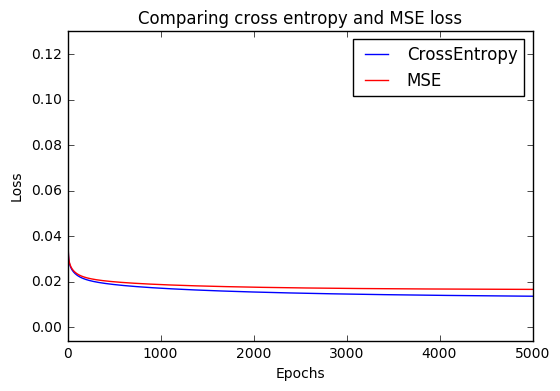
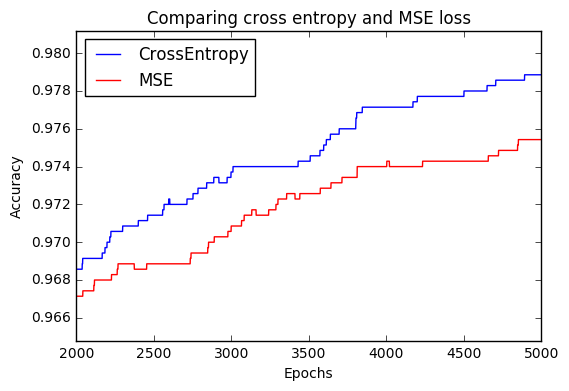
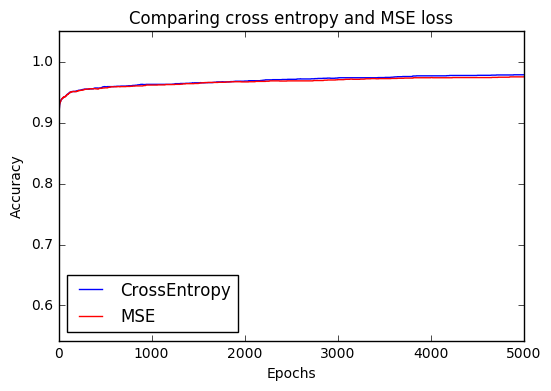
We see that this method has similar curves for the accuracy and history to the curves we have seen. However, this method reaches better accuracy performances faster. This can be explained because the MSE method can be overly sensitive to misclassified points, on the contrary binary cross-entropy deals better with these misclassified points by using the log odds. Here are the final performances of this method. Also, we notice that the losses with this method are bigger than the previous method (MSE), but this is due to the fact that we are not using the same loss functions, so at the minimum, the value is different.

|  |  |  |
| --- | --- | --- |
|  | Accuracy | Loss |
| Training Set | 0.9806 | 0.1145 |
| Validation Set | 0.98 | 0.1266 |
| Test Set | 0.9793 | 0.1274 |

## II.3 Comparison to Linear Regression

We have compared the Accuracy and Loss for the two loss functions (MSE and CE) and here are the results (the plots on the right only consider the values after 2000 epochs):

accuracy



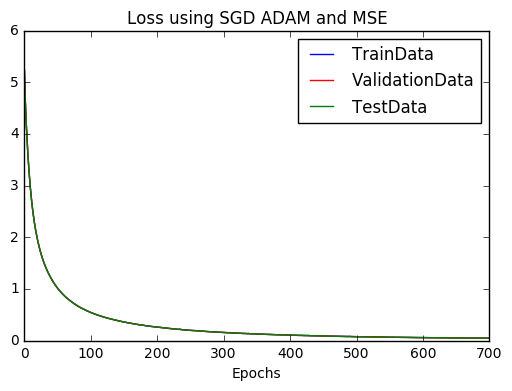
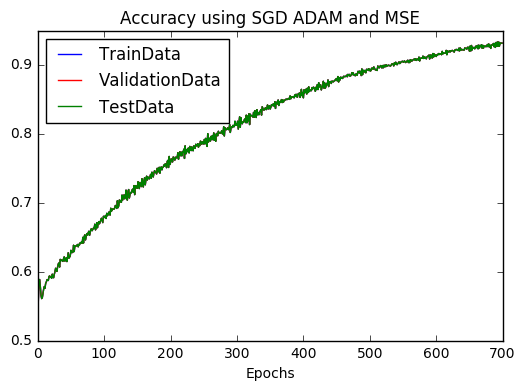
accuracy

As seen before the Cross entropy and the MSE have similar results, however (due to possible misclassified data points) the cross entropy is slightly better in the two measurements. We also see that the cross entropy gets really smaller values for the loss function when the regularization parameter is set to 0. Indeed, the final loss is now around 0.01 and was 0.1 in the previous question. So Cross entropy is more sensible to the regularization parameter. This is probably due to the fact that we are considering log odds, so using the regulation parameters has a bigger effect on the loss function using CE than the one using MSE.

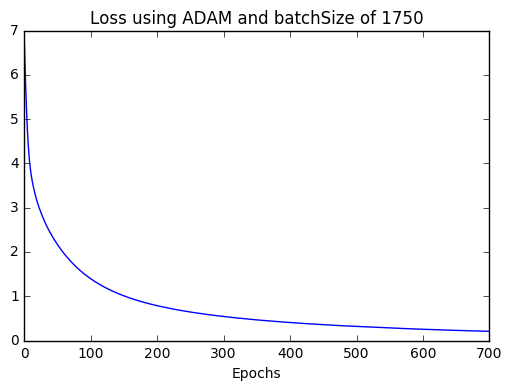
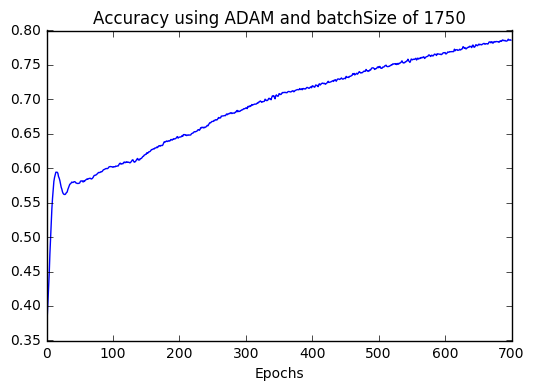
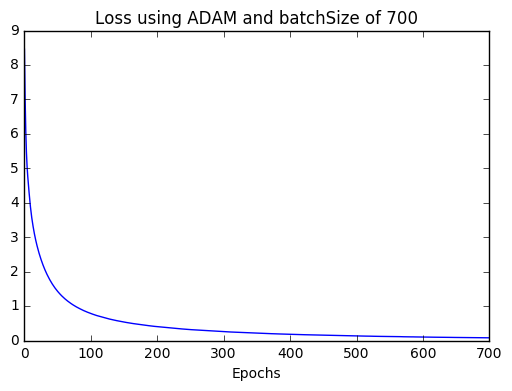
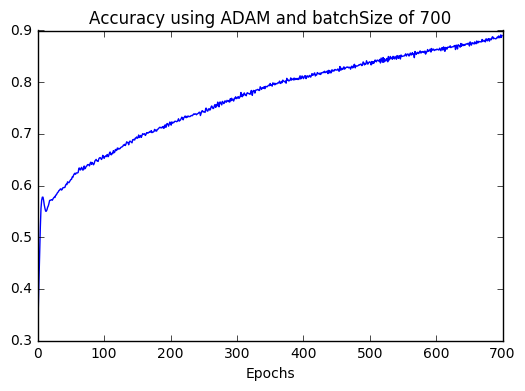
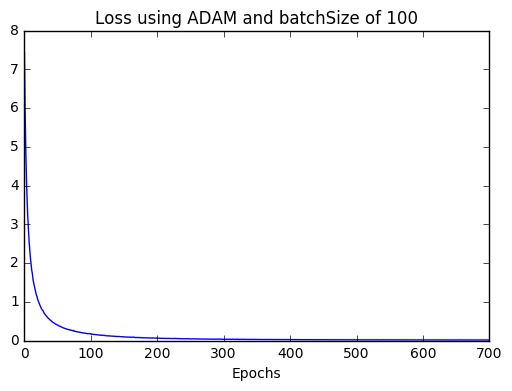
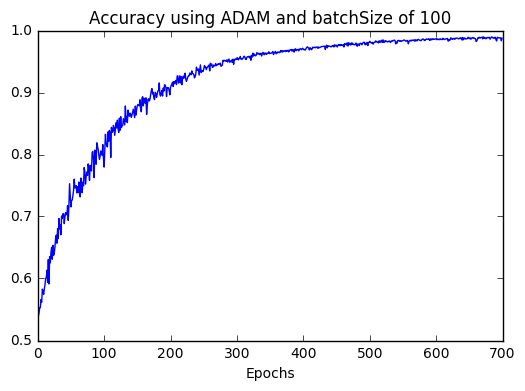
# III. Batch gradient Descent vs. SGD and Adam

## III.2. Implementing Stochastic Gradient Descent

Here are the plots we obtain after running the method:



## 3.Batch Size Investigation



We see that the accuracy grows faster and gets closer to 1 when the batch sizes are smaller, this is due to the fact that when the batch sizes are smaller, we have to run the optimizer more times to get to the desired 700 epochs. For example, when we use batch sizes of 100, we update the weights 35\*700 compared to only 2\*700 when using batch sizes of 1750. However, we see that the computation time is longer, when we use smaller batch sizes. This is because we need to compute the gradient more times and this is the bottleneck in the operations. So, we have to find a compromise between efficiency and accuracy.

## 4.Hyperparameter Investigation

Here are the results of the different scenarios:

**Training Data**

|  |  |  |
| --- | --- | --- |
|  | Final Accuracy | Final Loss |
| Betha1= 0.95 | 0.9300 | 0.0457 |
| Betha1=0.99 | 0.9269 | 0.0434 |
| Betha2=0.99 | 0.9489 | 0.0355 |
| Betha2=0.9999 | 0.9066 | 0.0622 |
| Epsilon = 1e-9 | 0.794 | 0.1705 |
| Epsilon=1e-4 | 0.8651 | 0.0958 |

**Validation Data**

|  |  |  |
| --- | --- | --- |
|  | Final Accuracy | Final Loss |
| Betha1= 0.95 | 0.89 | 0.0763 |
| Betha1=0.99 | 0.88 | 0.2350 |
| Betha2=0.99 | 0.89 | 0.0683 |
| Betha2=0.9999 | 0.75 | 0.1732 |
| Epsilon = 1e-9 | 0.87 | 0.0863 |
| Epsilon=1e-4 | 0.74 | 0.3 |

**Test Data**

|  |  |  |
| --- | --- | --- |
|  | Final Accuracy | Final Loss |
| Betha1= 0.95 | 0.9103 | 0.0635 |
| Betha1=0.99 | 0.8483 | 0.1714 |
| Betha2=0.99 | 0.9034 | 0.0559 |
| Betha2=0.9999 | 0.8414 | 0.1887 |
| Epsilon = 1e-9 | 0.9103 | 0.0669 |
| Epsilon=1e-4 | 0.8069 | 0.2813 |

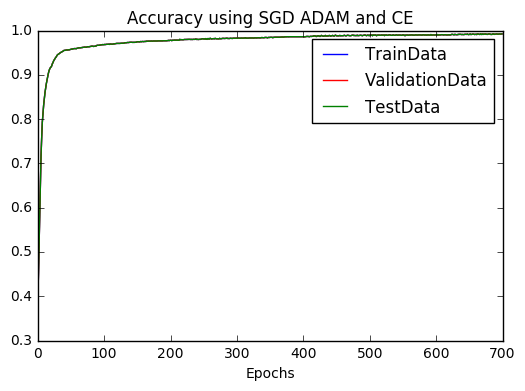
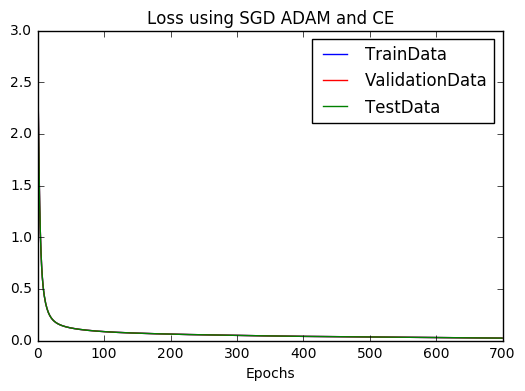
We see that the performances are the best on the training data set. This is what one would expect since we re optimizing the weights over these data points. The performance is less than the batch algorithm. This is because here we are updating roughly the same number of times the weights (4900 compared to 5000 times), but each update only looks at one of the minibatches. As seen on the different papers published about the ADAM method, it is an algorithm that on the long term generates better results more efficiently.

Here we see that for better results seem to be achieved when:

* Beta1 is 0.95
* Beta2 is 0.9
* Epsilon is 1e-8 (default value)

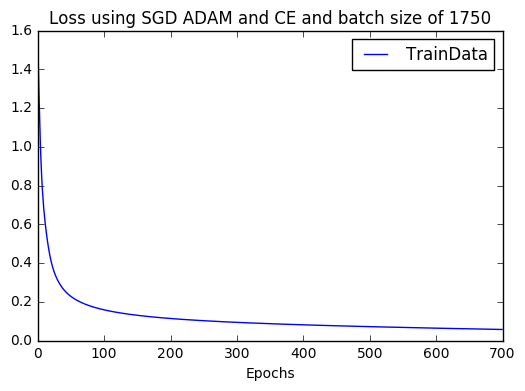
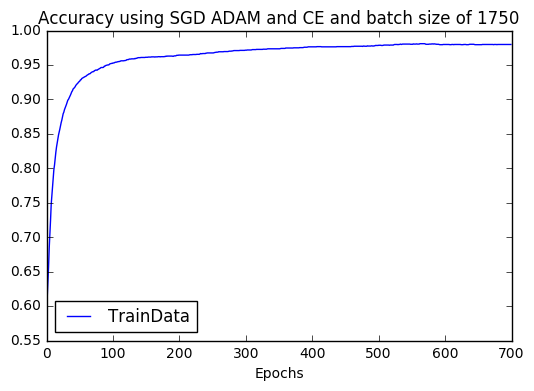
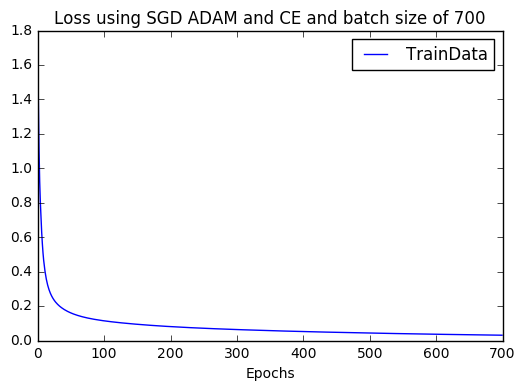
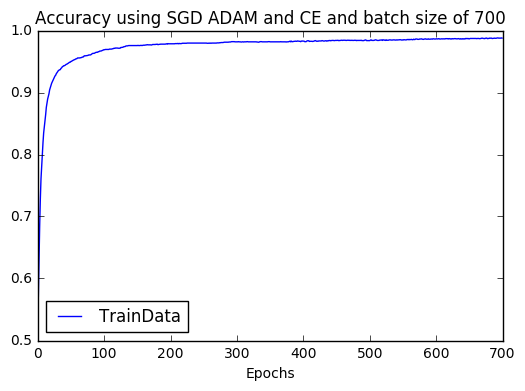
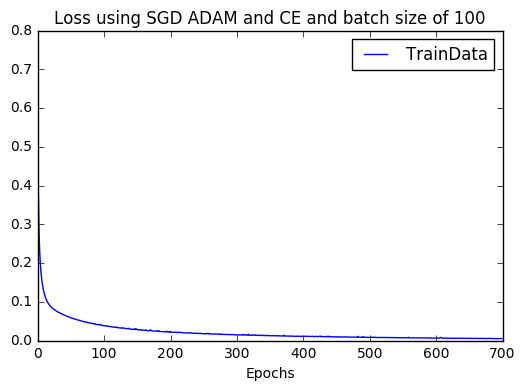
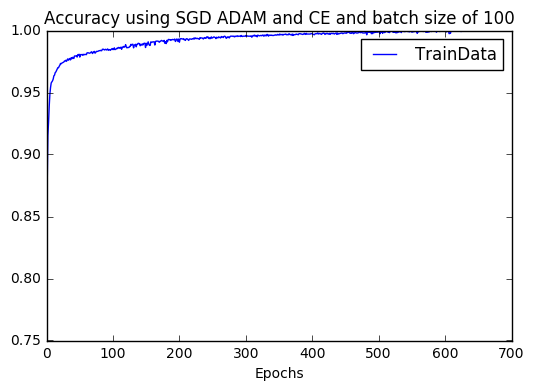
### 5. Cross entropy loss investigation

We first did the plots with the default parameters and alpha=0.001 using the cross entropy function:

We see th

We see that this model has better performances faster that the one using the MSE loss function.

We then study the changes of the batch size.



Here are the final measurements:

|  |  |  |
| --- | --- | --- |
| Batch Size | Accuracy | Loss |
| 100 | 0.9997 | 0.0044 |
| 700 | 0.9914 | 0.0257 |
| 1750 | 0.98 | 0.0565 |

These are the best results we have seen so far. As explained before, the smaller the batch size, the better the accuracy and the loss but the longer the computation times. From here we see that CE loss function is better for predictions.

We finally check the changes obtained from varying the parameters of the optimizer:

**Training Data**

|  |  |  |
| --- | --- | --- |
|  | Final Accuracy | Final Loss |
| Betha1= 0.95 | 0.9909 | 0.0252 |
| Betha1=0.99 | 0.9914 | 0.0236 |
| Betha2=0.99 | 0.9966 | 0.0120 |
| Betha2=0.9999 | 0.9931 | 0.0242 |
| Epsilon = 1e-9 | 0.9911 | 0.0249 |
| Epsilon=1e-4 | 0.9946 | 0.0190 |

**Validation Data**

|  |  |  |
| --- | --- | --- |
|  | Final Accuracy | Final Loss |
| Betha1= 0.95 | 0.97 | 0.0756 |
| Betha1=0.99 | 0.98 | 0.0652 |
| Betha2=0.99 | 0.97 | 0.0486 |
| Betha2=0.9999 | 0.98 | 0.0522 |
| Epsilon = 1e-9 | 0.98 | 0.0253 |
| Epsilon=1e-4 | 0.98 | 0.0608 |

**Test Data**

|  |  |  |
| --- | --- | --- |
|  | Final Accuracy | Final Loss |
| Betha1= 0.95 | 0.9586 | 0.1617 |
| Betha1=0.99 | 0.9724 | 0.1296 |
| Betha2=0.99 | 0.9724 | 0.1480 |
| Betha2=0.9999 | 0.9724 | 0.1876 |
| Epsilon = 1e-9 | 0.9793 | 0.1469 |
| Epsilon=1e-4 | 0.9793 | 0.1145 |

We see once again that the training data has the best performances, this is normal, indeed, the optimization is done on this data set. We see that performances are better when:

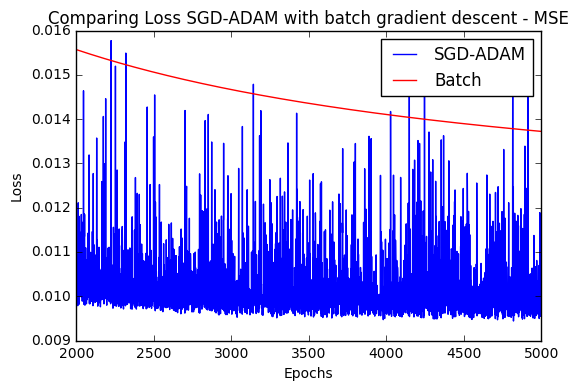
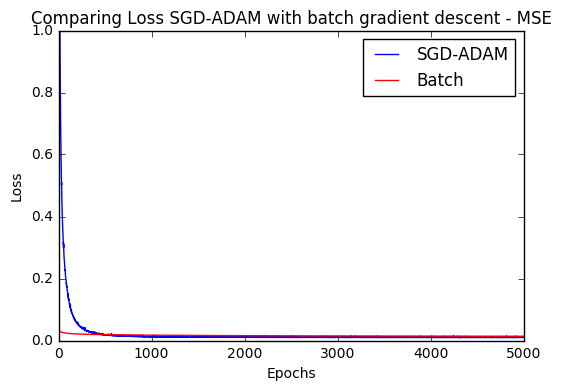
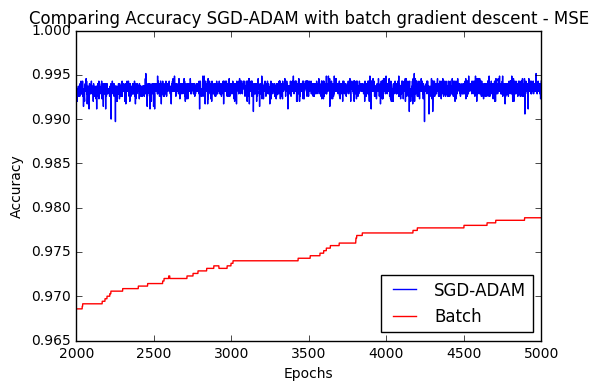
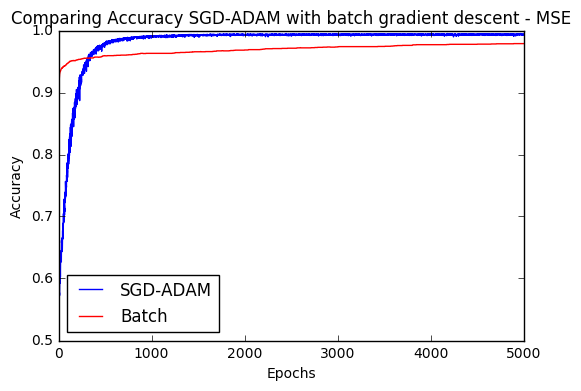
* Betha1 is 0.99
* Betha2 is 0.9(not a big difference though)
* Epsilon is equal to the default value 1e-8

We see that overall, the Cross-entropy function is better than the MSE loss function. This is mainly because the sigmoid and the log-odds allow to better deal with more or less misclassified datapoints.

## III.6 Comparison against Batch GD

To compare the SGD algorithm with ADAM and the batch algorithm we run them bot for 5000 epochs (SGD-ADAM is run with batch sizes of 500, alpha=0.001 and the batch algorithm has =0.001) and here are the results:

First, we note that the calculation for the SGD algorithm took 272s and the batch algorithm took 22s. This is because each epoch in the SGD algorithm computes the gradient 7 times, so this takes more time.

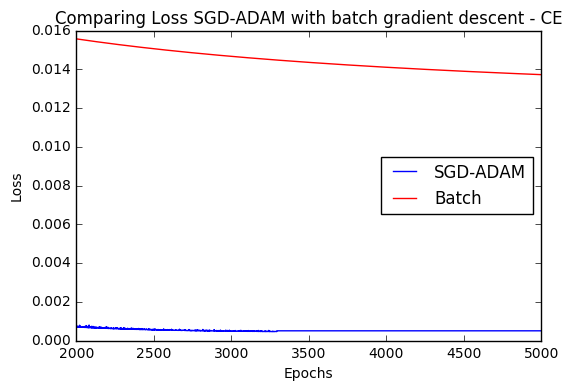
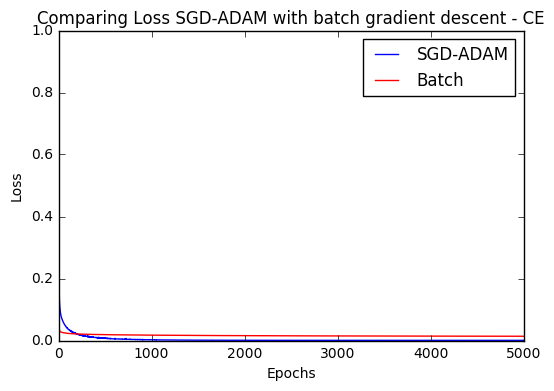
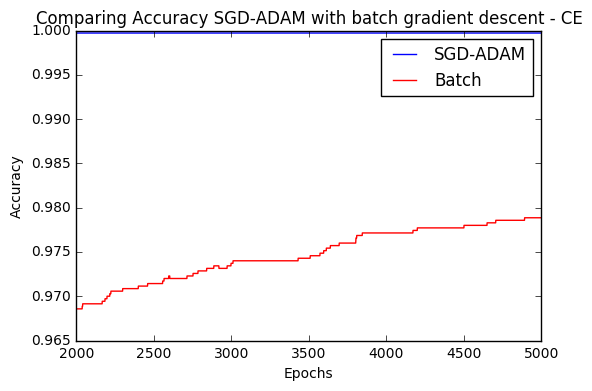
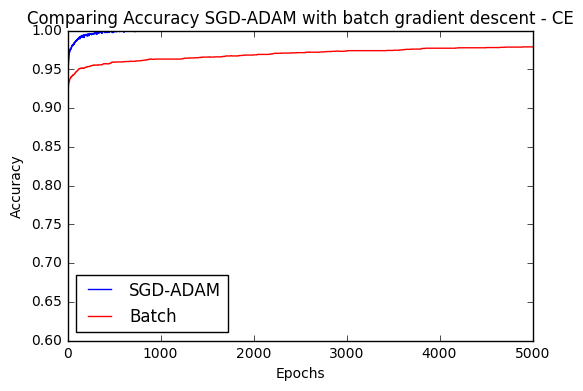


We see that the SGD algorithm is less performant for the first epochs but after roughly 500 epochs, it has almost perfect accuracy and loss.

So, we can conclude that for applications where computation time is scarce, one should better use the batch GD algorithm since it gives good performances faster. SGD-Adam on the other side is better for applications where computational time is available.

The reason for this slow behaviour of SGD-ADAM in the beginning is probably that ADAM has to increase its “velocity” in the beginning.

Also, we see that when we use the batch GD, the loss and accuracy curves are smooth, whereas when we use the SGD-ADAM algorithm the curves are “noisy”. This is because in batch GD we update weights on the whole data set and on the SGD algorithm we only update weights based on minibatches, so every update is not necessarily making the loss function go towards its minimum on the most direct path.

Here are the same results by using the cross-entropy loss function:

The conclusions are the same as with MSE. Here we see that SGD-ADAM has better results than SGD even faster and also, we notice that the performances at the end are significantly better with SGD-ADAM than with GD batch. The computations for SGD-ADAM are done in 297.45s and those for GD batch are done in 22.31s. We can conclude the same once again: SGD-ADAM is better for applications which want better accuracies and have the appropriate computation time available, whereas GD batch is better for applications which want relatively good results fast.