ITMAL

Hand-in: Journal 2

Af Gruppe 12

|  |  |
| --- | --- |
| Navn | Studie Nummer |
| Max Barly Jørgensen | 201401694 |
| Nicholas Ladefoged | 201500609 |
| Kristoffer Villadsen | 201607406 |
| Søren Katborg |  |

Indhold

[Gradient descent 4](#_Toc5613301)

[Qa: 4](#_Toc5613302)

[Qb: 5](#_Toc5613303)

[Qd: 5](#_Toc5613304)

[Forklaring: 5](#_Toc5613305)

[Qe: 6](#_Toc5613306)

[Capacity under overfitting 6](#_Toc5613307)

[Qa: 6](#_Toc5613308)

[Qb: 7](#_Toc5613309)

[Generalization error 8](#_Toc5613310)

[Qa: 8](#_Toc5613311)

[Qb: 8](#_Toc5613312)

[Qb part II: 9](#_Toc5613313)

[Naïve Bayes 9](#_Toc5613314)

[Regulizers 10](#_Toc5613315)

[Qa: 10](#_Toc5613316)

[Qb: 11](#_Toc5613317)

[Qc: 13](#_Toc5613318)

[Qd: 13](#_Toc5613319)

[Gridsearch 13](#_Toc5613320)

[Qa 13](#_Toc5613321)

[Cell 1 14](#_Toc5613322)

[Cell 2 14](#_Toc5613323)

[Qb 15](#_Toc5613324)

[Qc 16](#_Toc5613325)

[Qd 18](#_Toc5613326)

[Speed up by compression 20](#_Toc5613327)

[Noise reduction 20](#_Toc5613328)

[PCA -> t-SNE features 21](#_Toc5613329)

[Neurons 21](#_Toc5613330)

[Qa: 21](#_Toc5613331)

[Qb: 21](#_Toc5613332)

[Perceptron 22](#_Toc5613333)

[Qa: 22](#_Toc5613334)

[Qb: 22](#_Toc5613335)

[Qc: 23](#_Toc5613336)

[Qd: 23](#_Toc5613337)

[Multi-layers Perceptrons (MLP) 24](#_Toc5613338)

[Qa: 24](#_Toc5613339)

[Cell 1 24](#_Toc5613340)

[Cell 2 24](#_Toc5613341)

[Cell 3 25](#_Toc5613342)

[Qb: 26](#_Toc5613343)

[Qc: 26](#_Toc5613344)

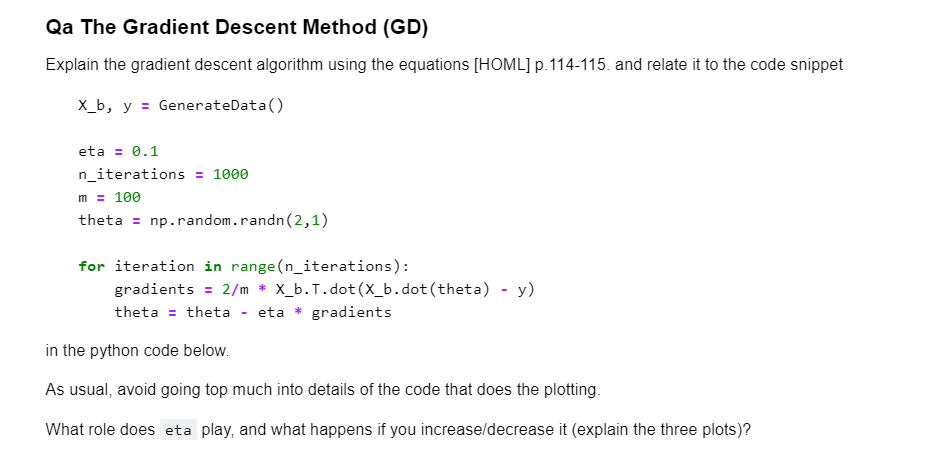
[Keras Multi-Layer Perceptrons (MLP’s) on MNIST-data 28](#_Toc5613345)

[Qa: 29](#_Toc5613346)

[Qb: 31](#_Toc5613347)

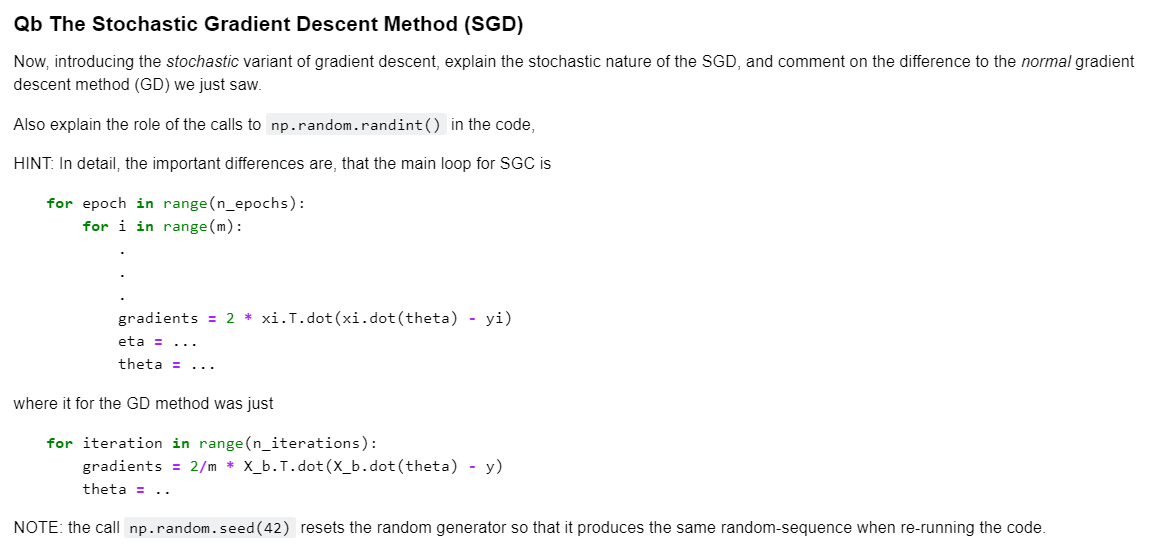
# Gradient descent

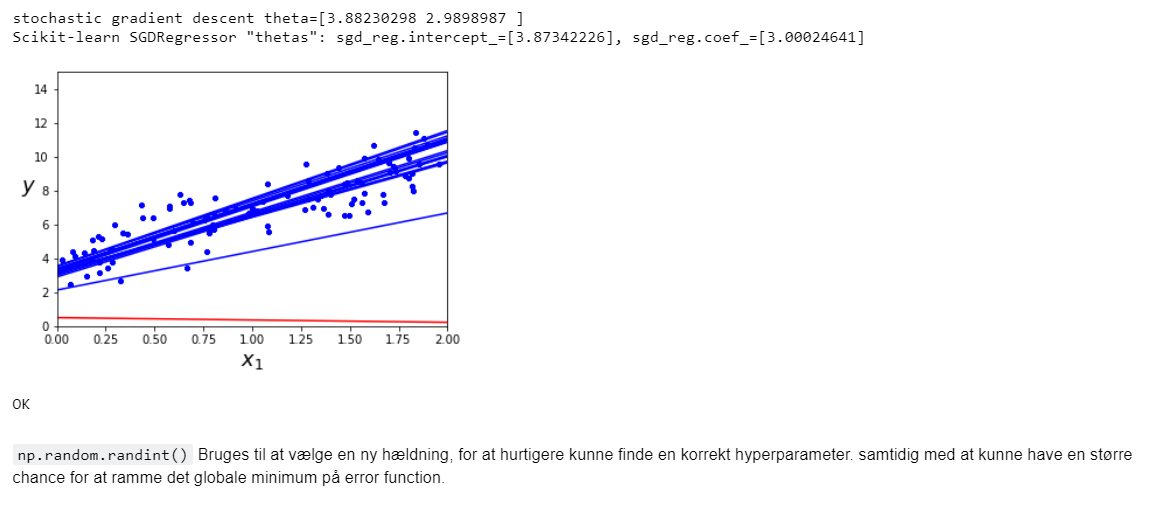
## Qa:



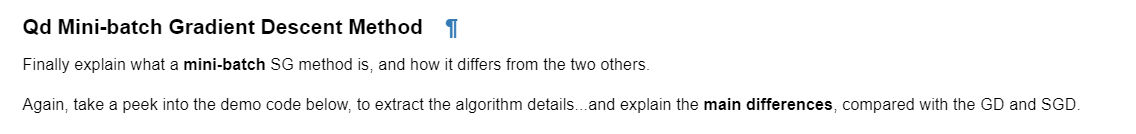


## Qb:





## Qd:

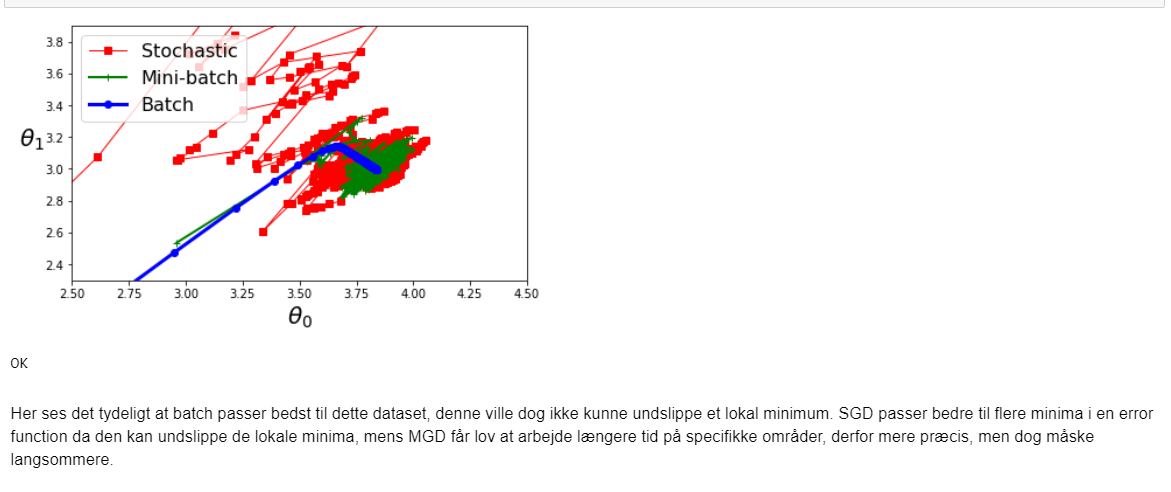


### Forklaring:



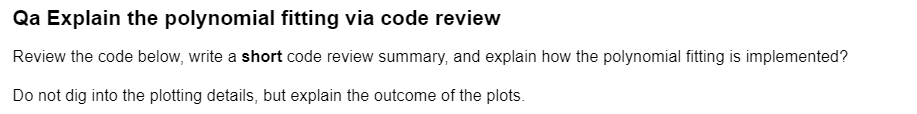
## Qe:

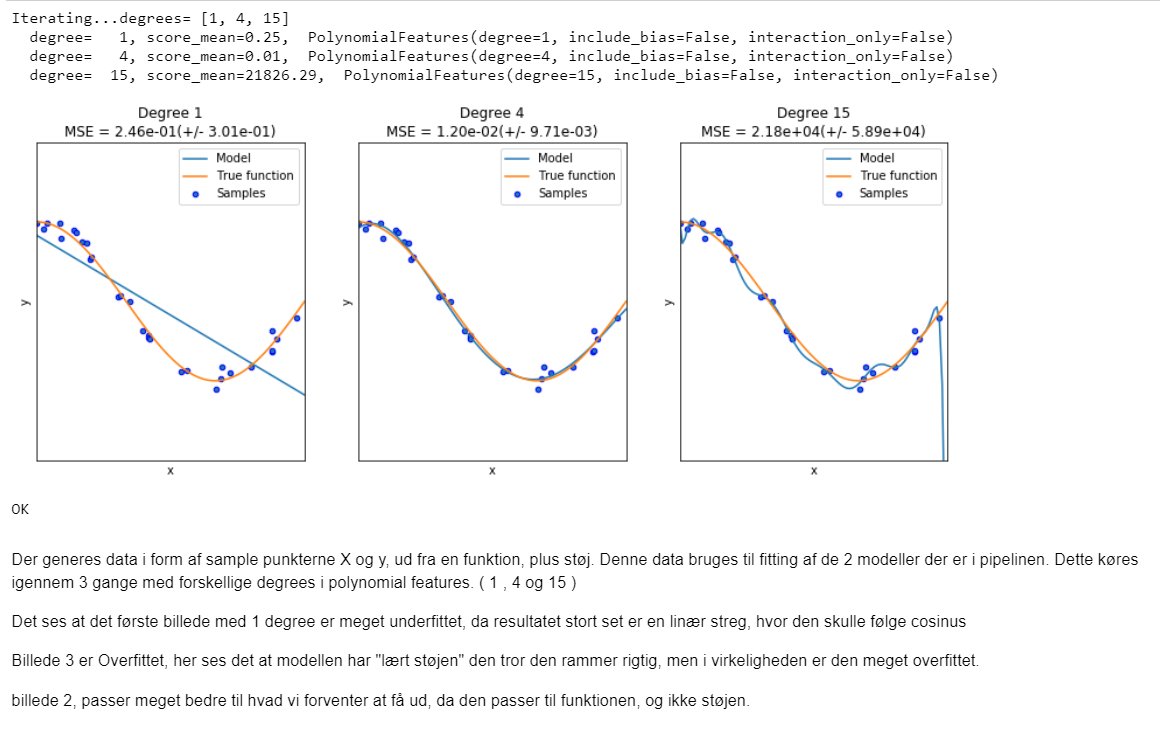




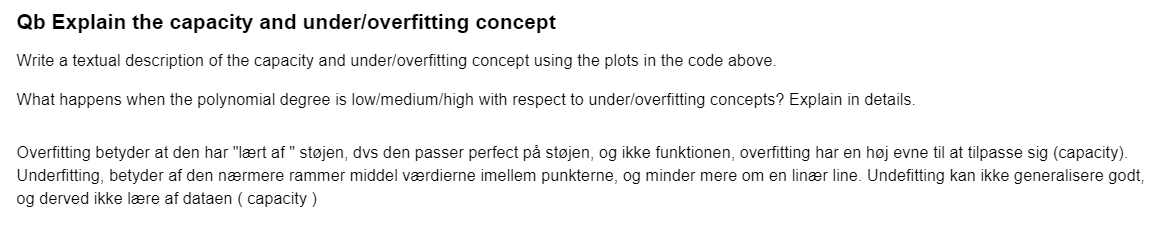
# Capacity under overfitting

## Qa:



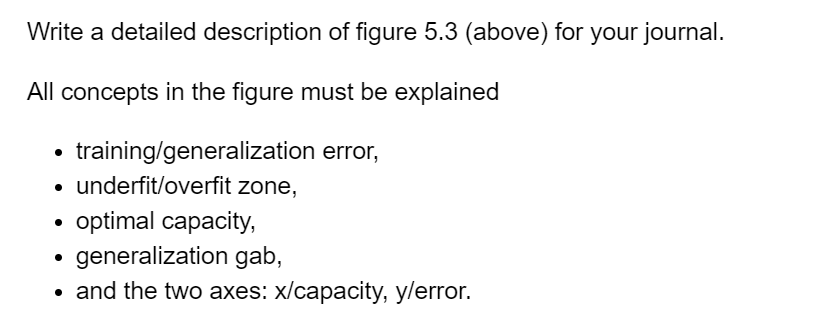


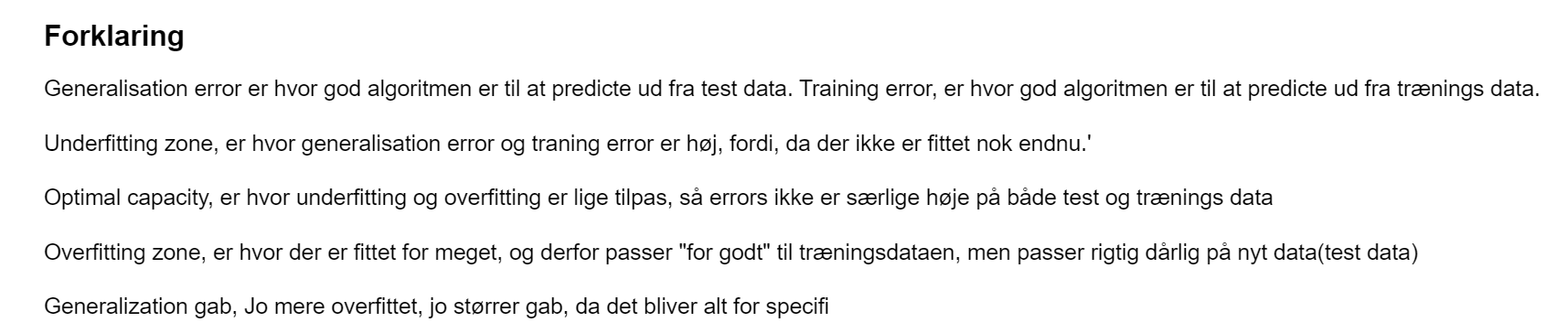
## Qb:



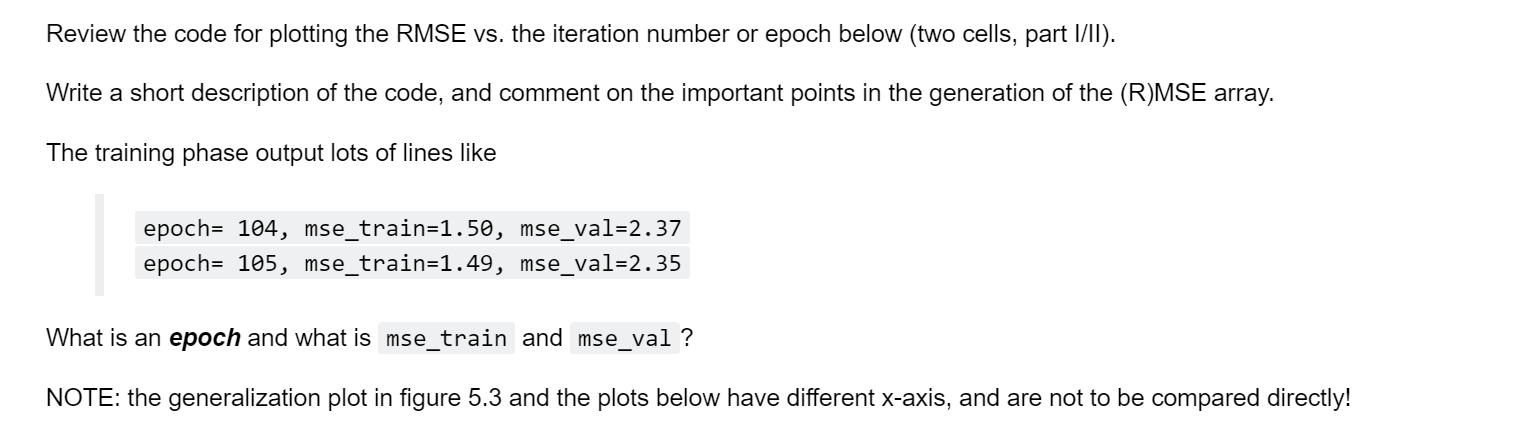
# Generalization error

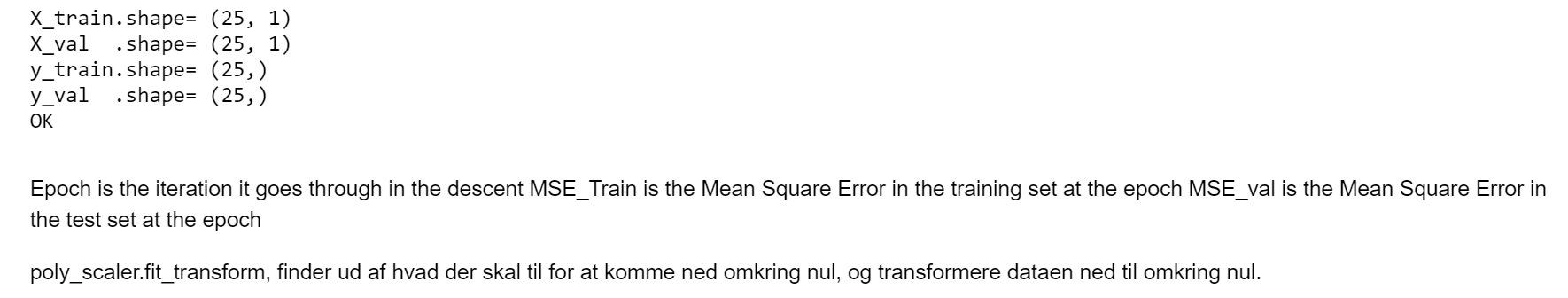
## Qa:



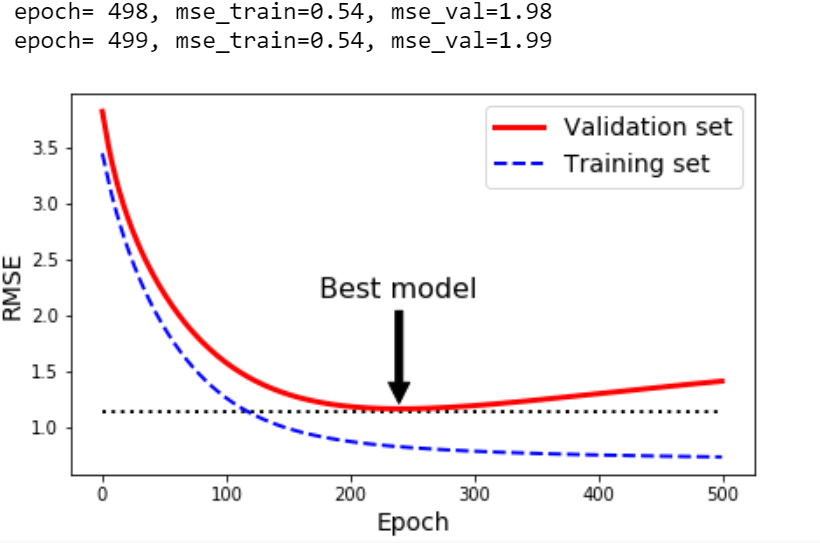


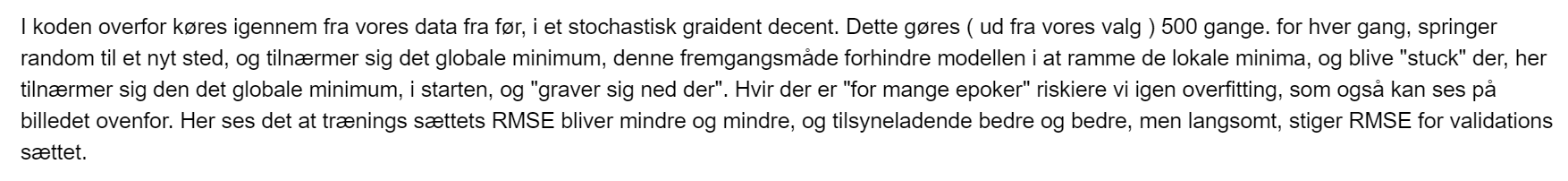
## Qb:



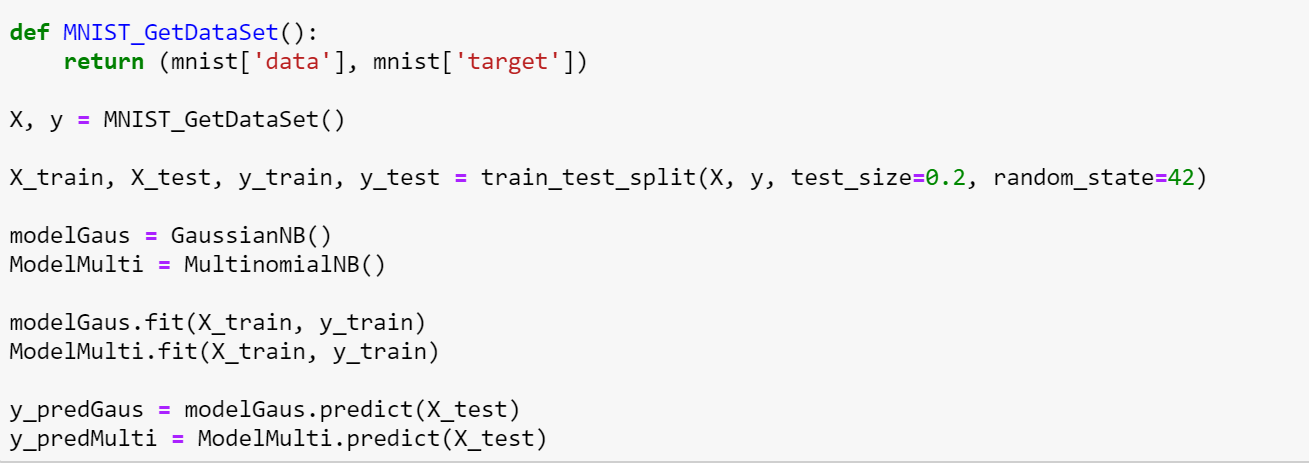


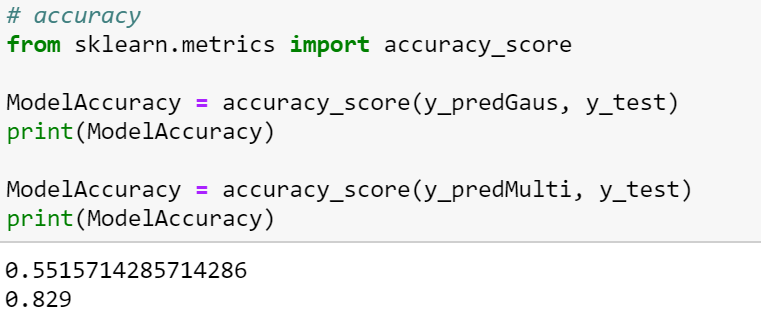
## Qb part II:

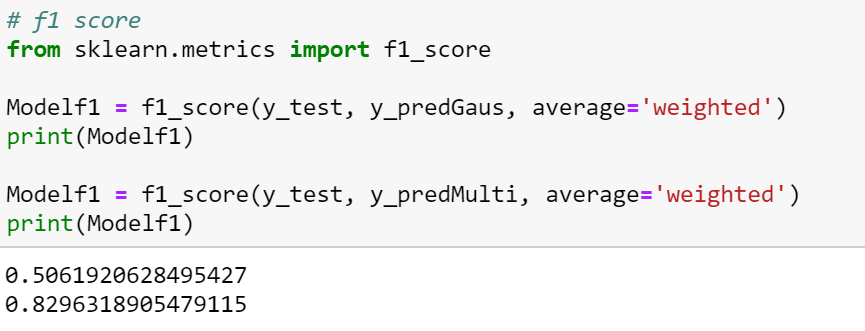


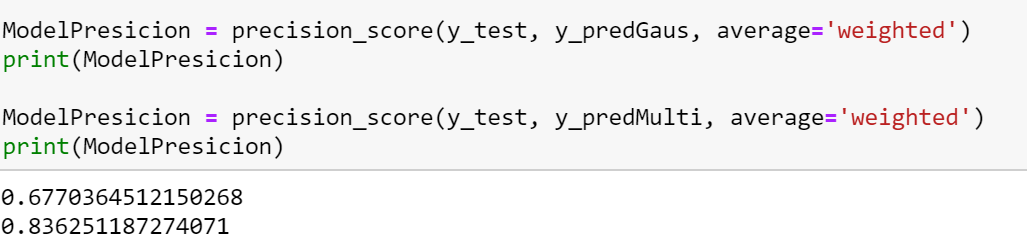


# Naïve Bayes



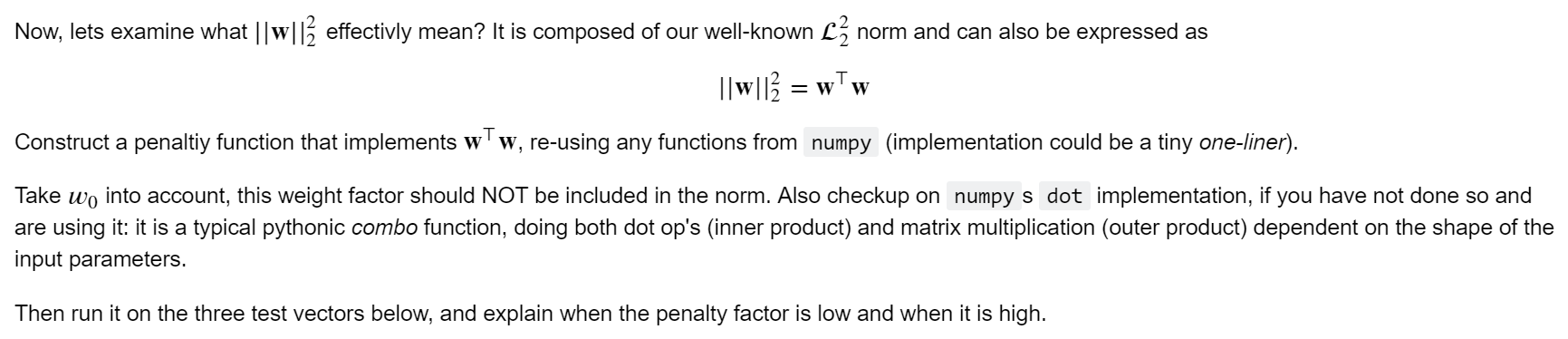


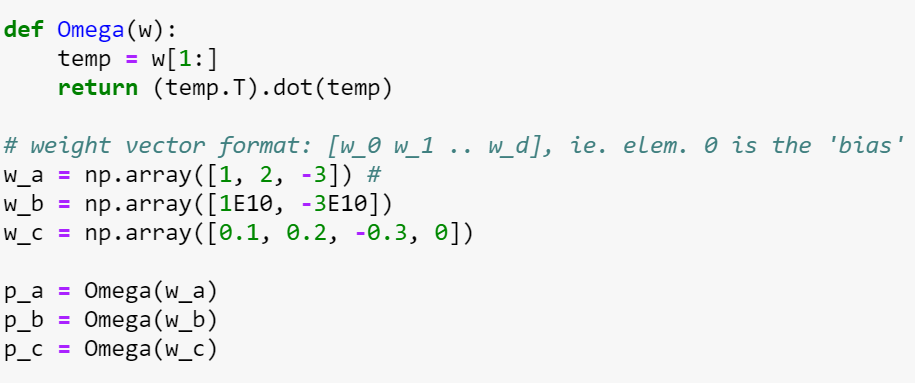




# Regulizers

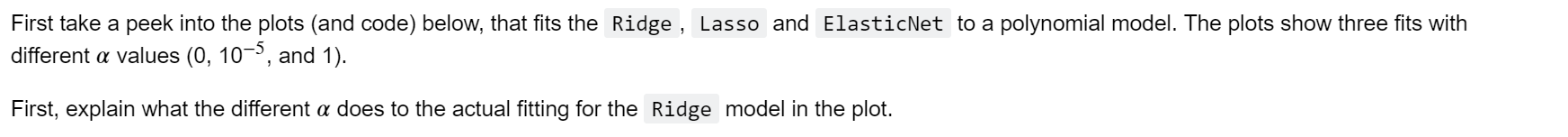
## Qa:

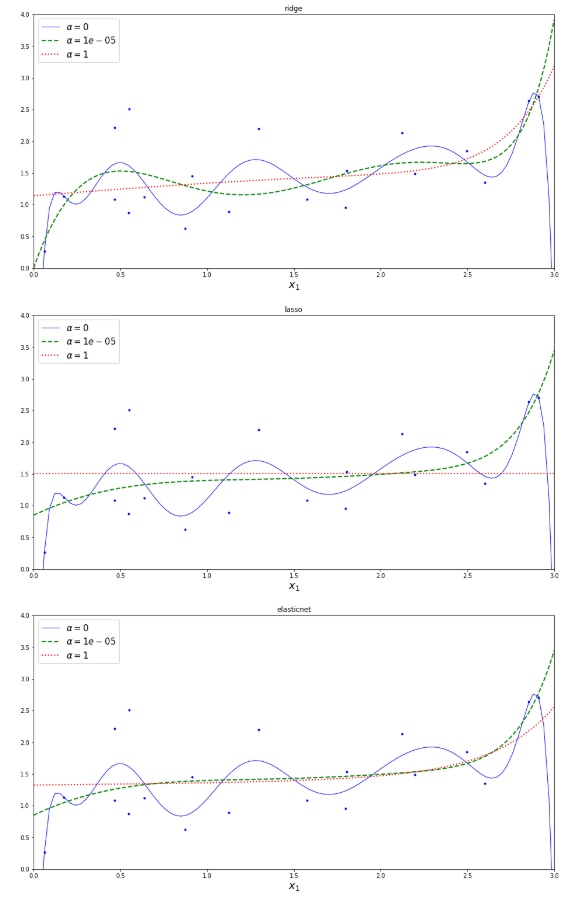


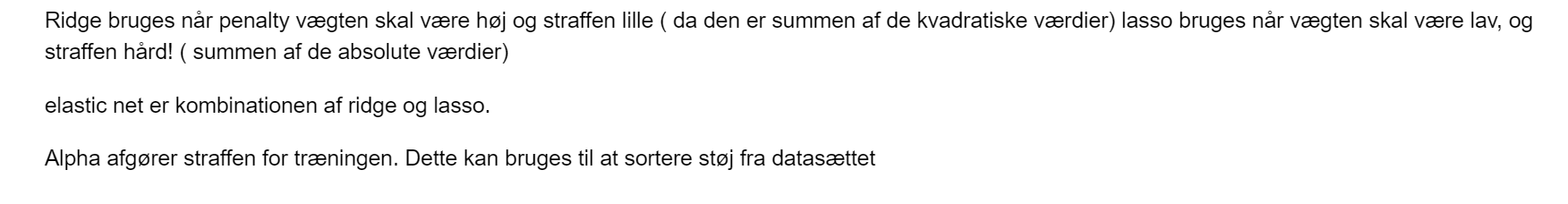




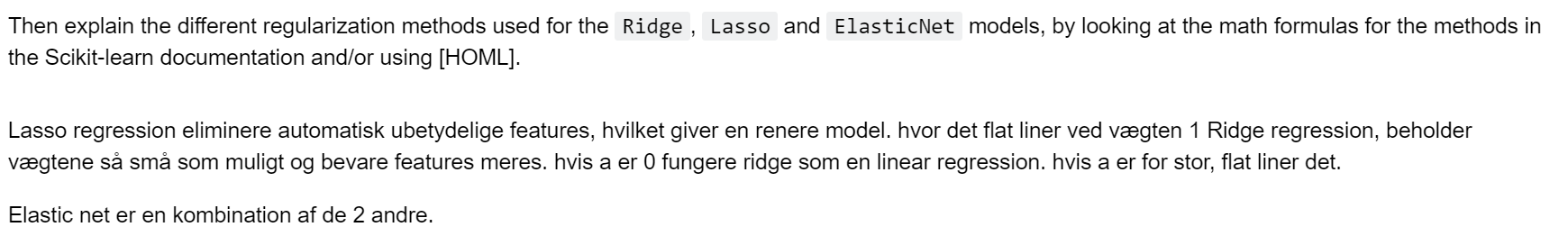
## Qb:



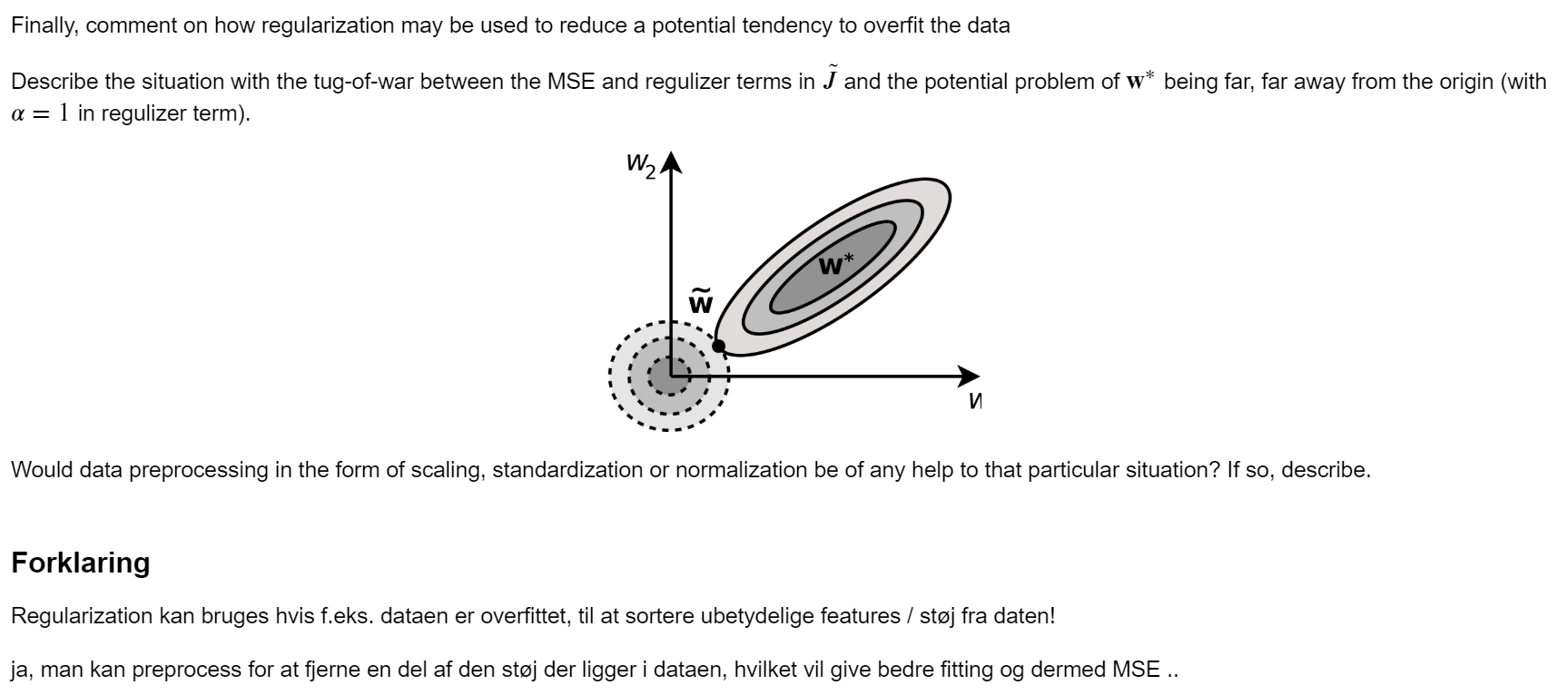




## Qc:

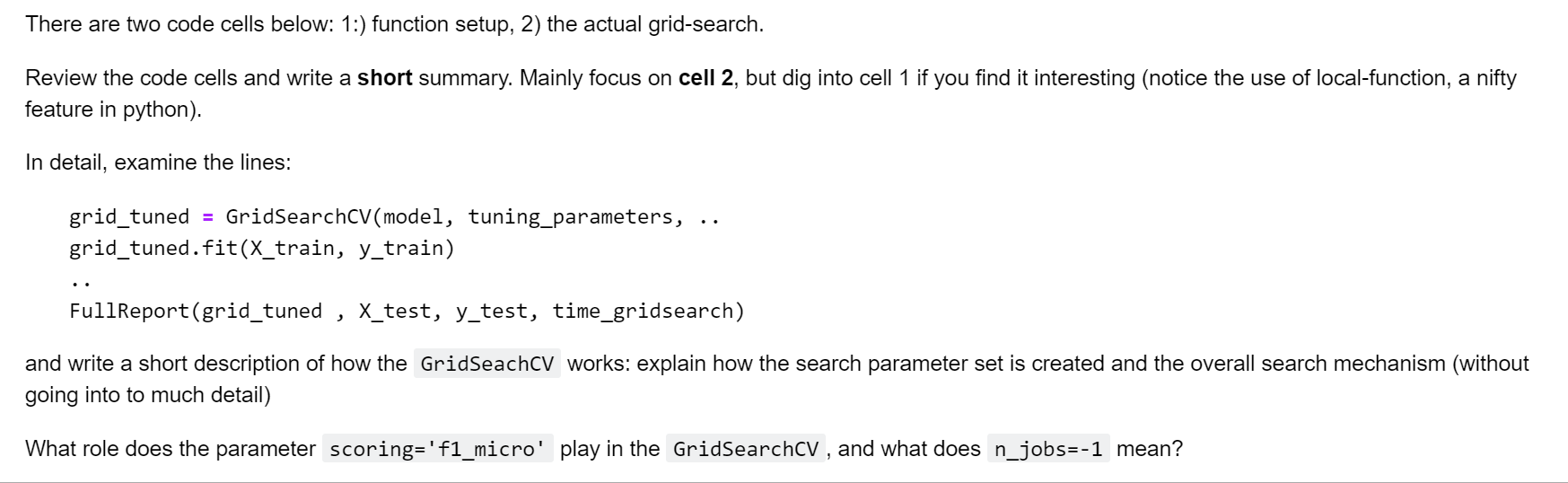


## Qd:

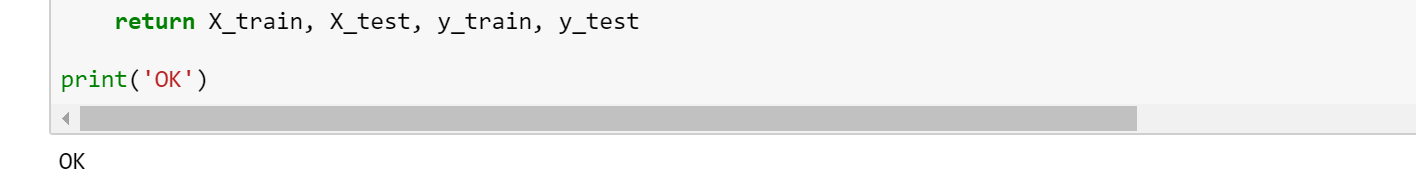


# Gridsearch

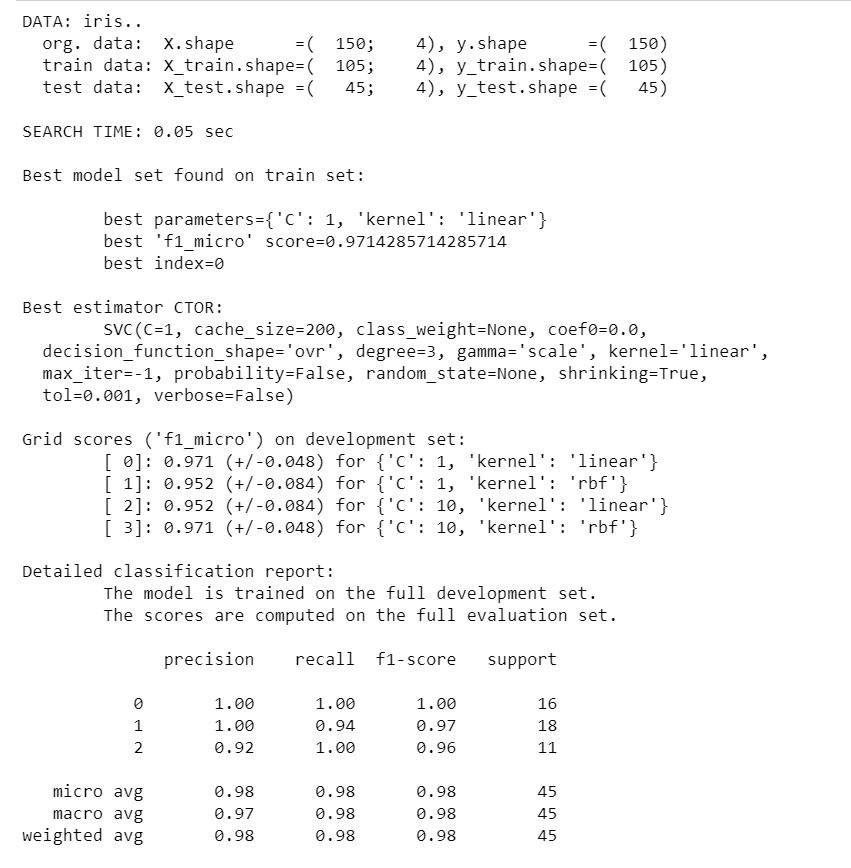
## Qa



### Cell 1



### Cell 2





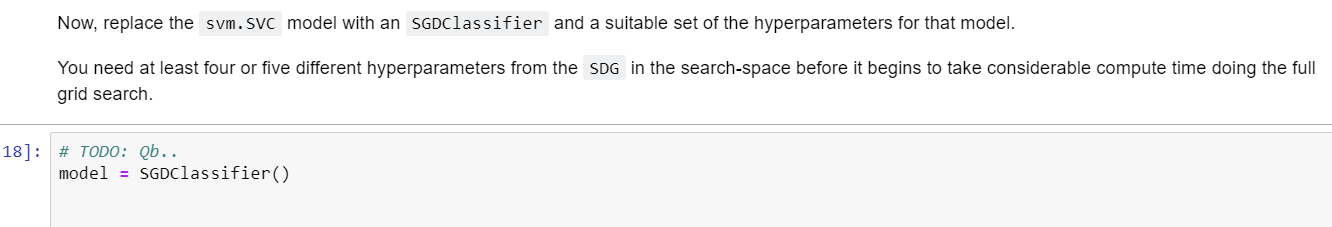
### Forklaring:

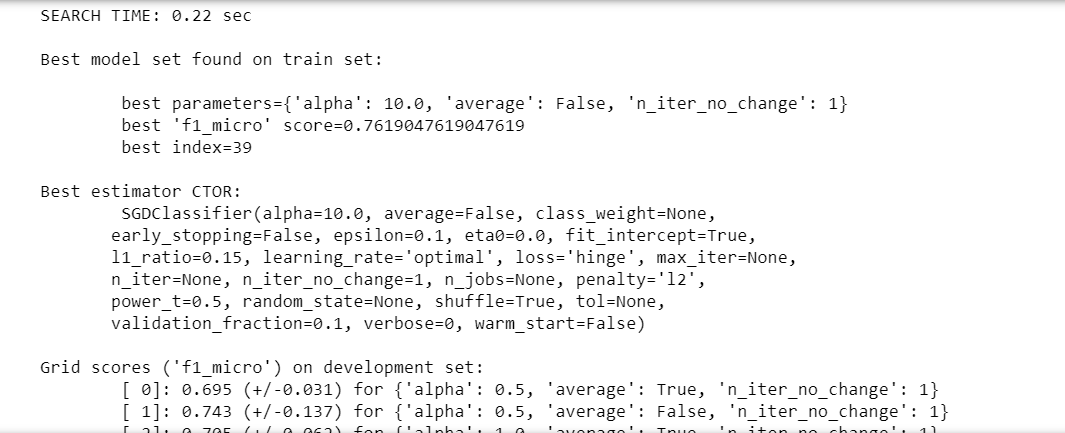
Grid search o andre modeller har en masse parameter som kan instillies men i mange tilfælde vil man lade disse da forblive på default.

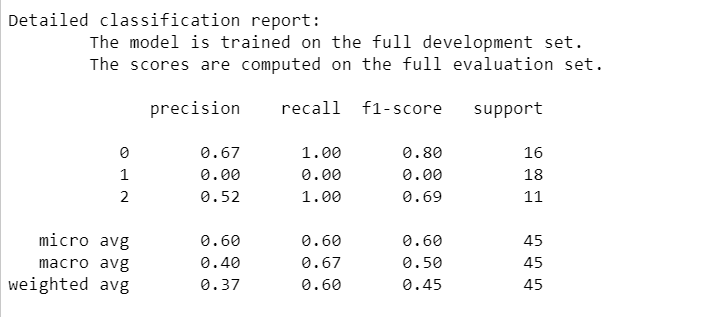
I grid search cv forsøges der at tilpasse disse parameter til vores model. Dette gøres ved at prøve sig frem i mange forsøg, dog kan nogle parametre vælges i forvejen. Der beregnes ved hver forsøg en precision, recall, f1-score og en support. Disse værdier bruges derefter til at bestemme hvilken sæt af parametere der er bedst.

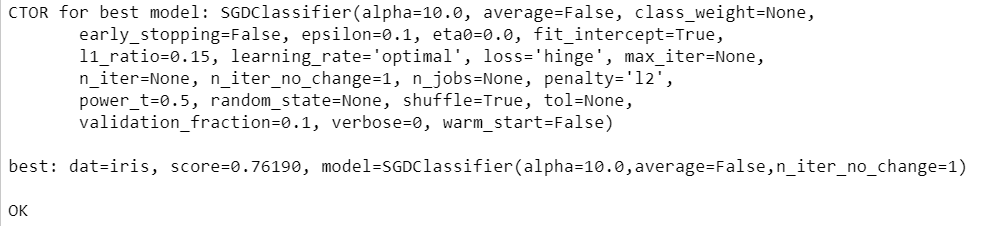
scoring f1\_micro er den score der bruges til at regulere med. n\_jobs=-1 kører parralelt.

## Qb

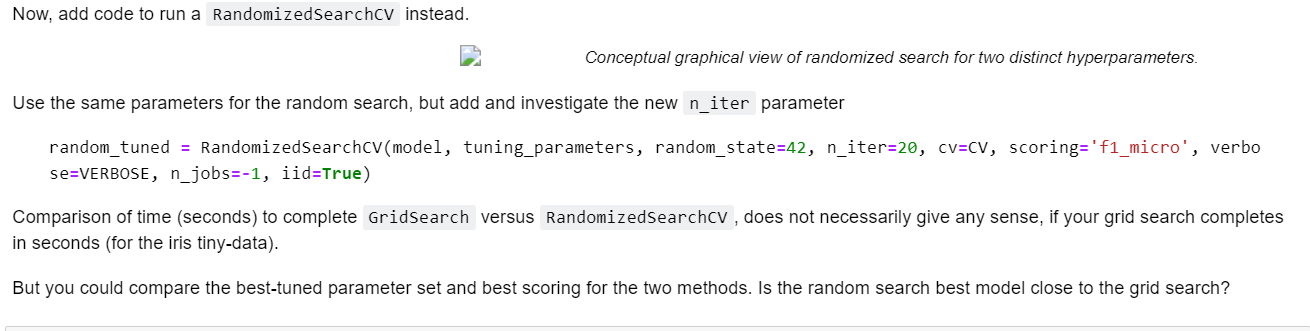




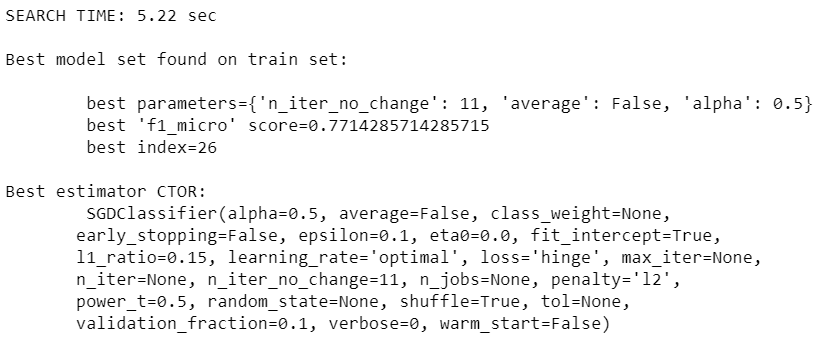


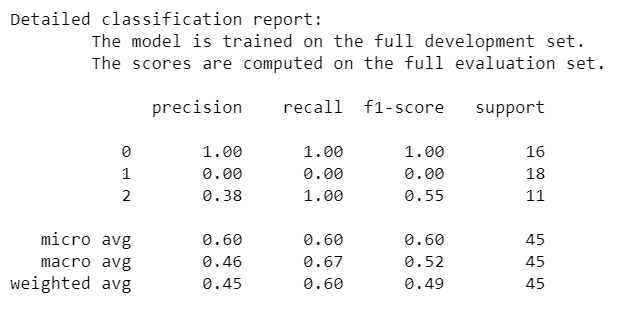


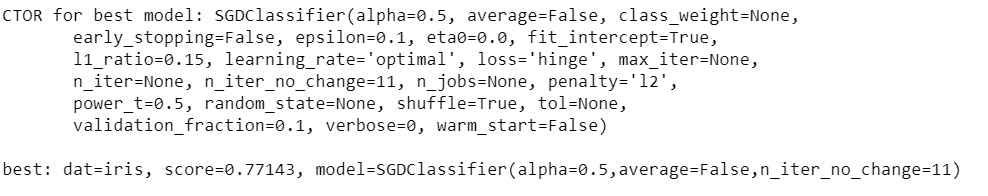
## Qc

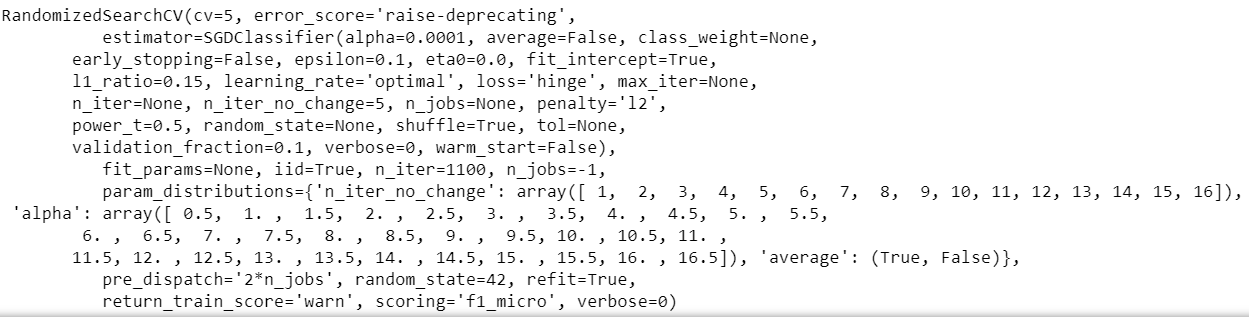




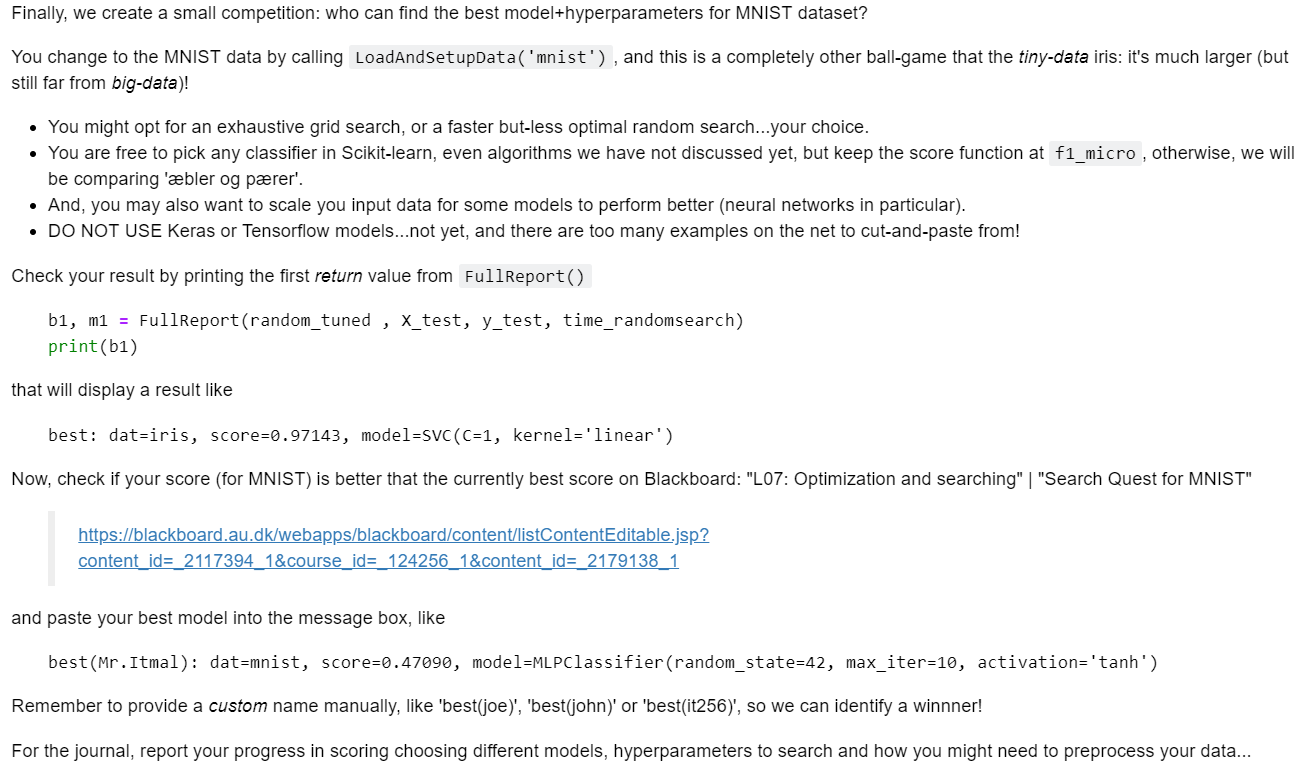


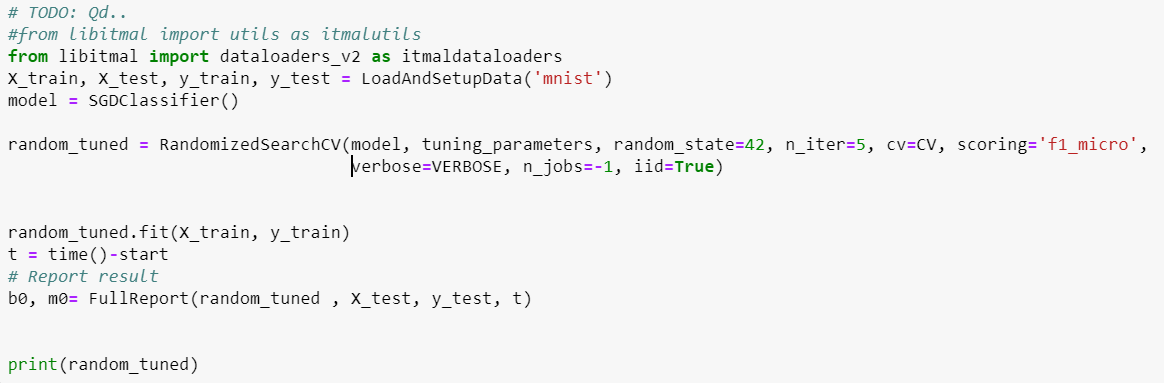


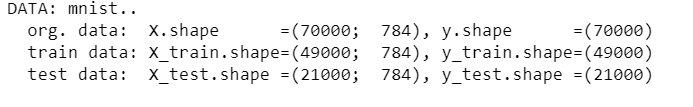


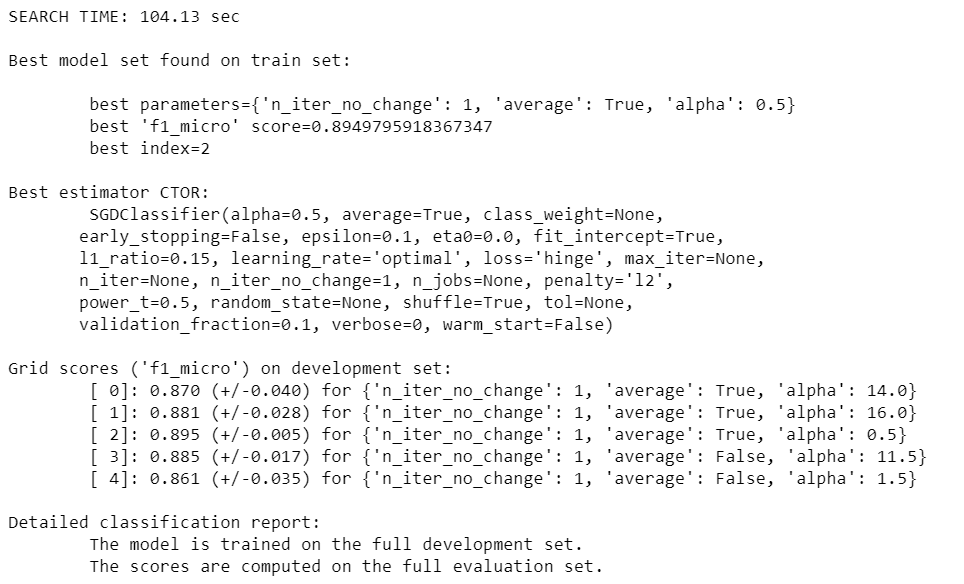


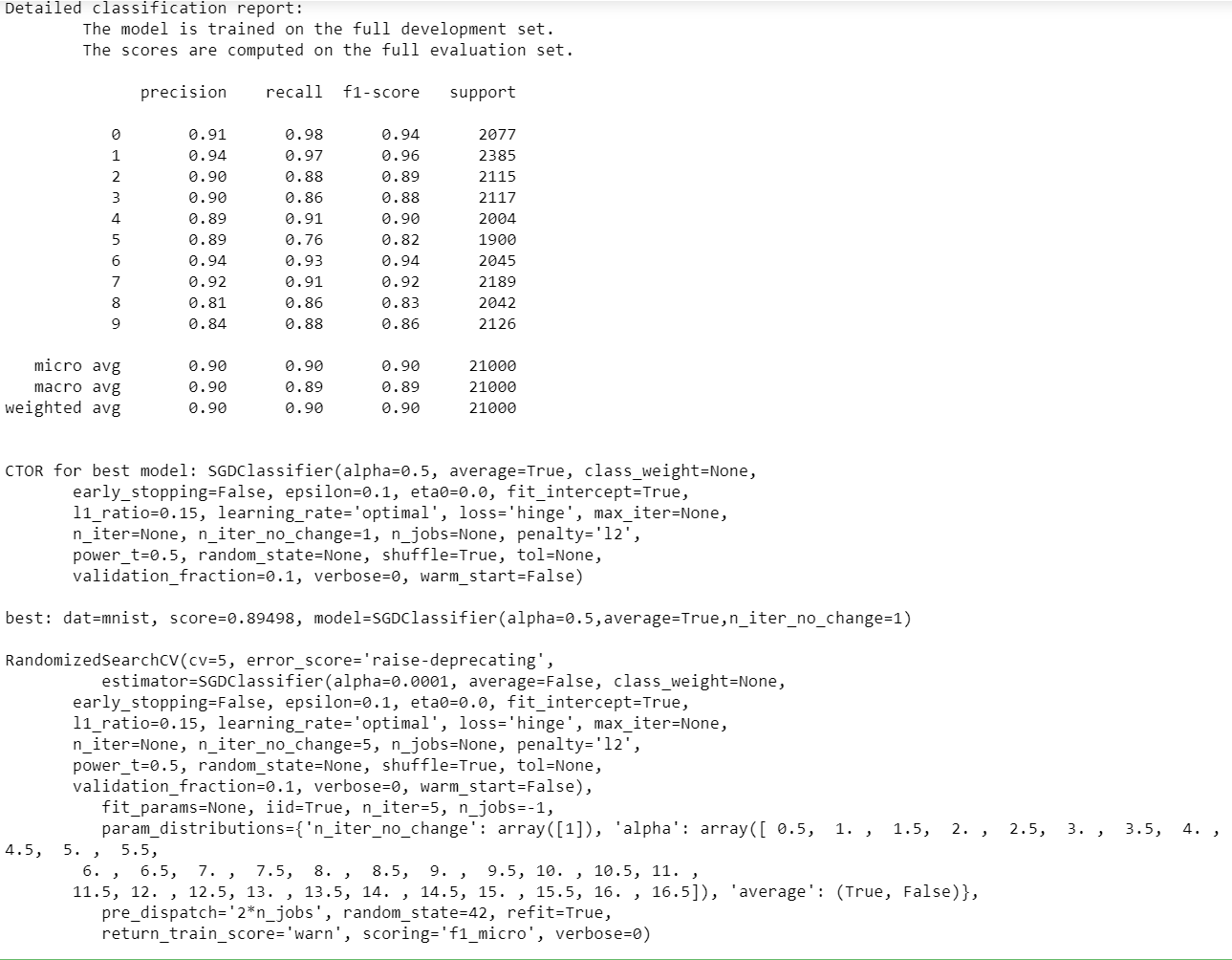
## Qd



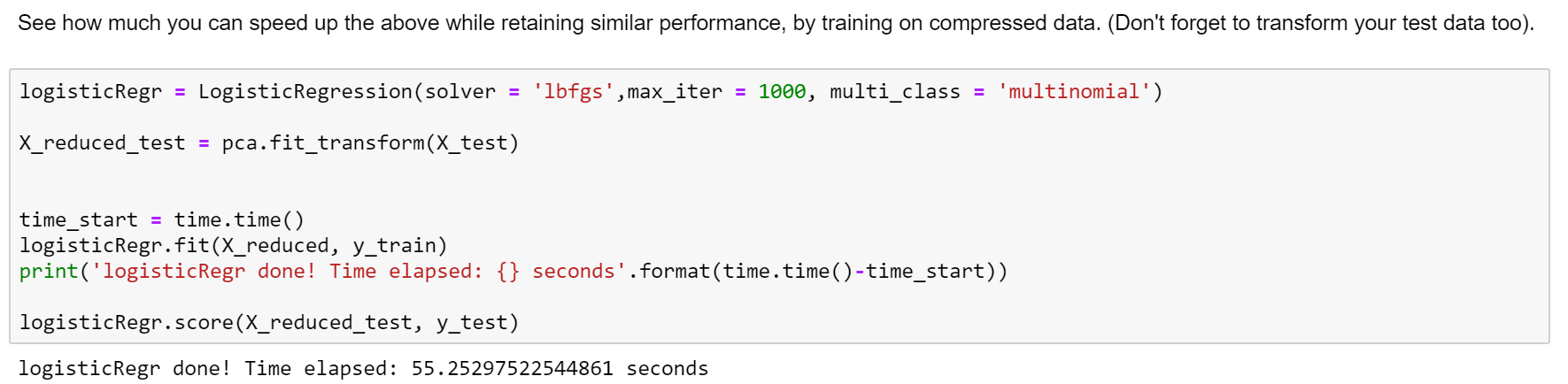


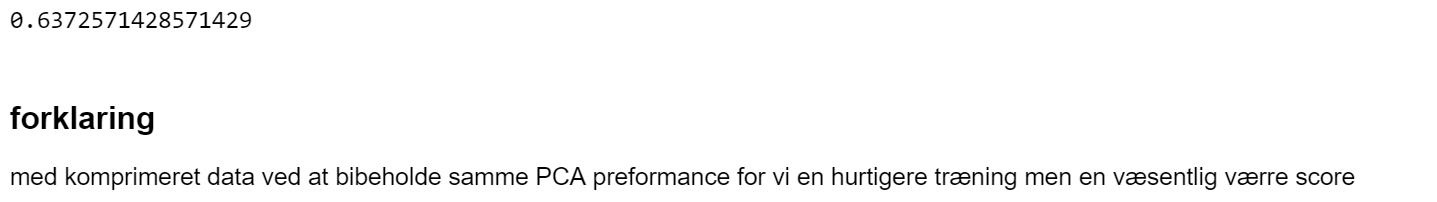




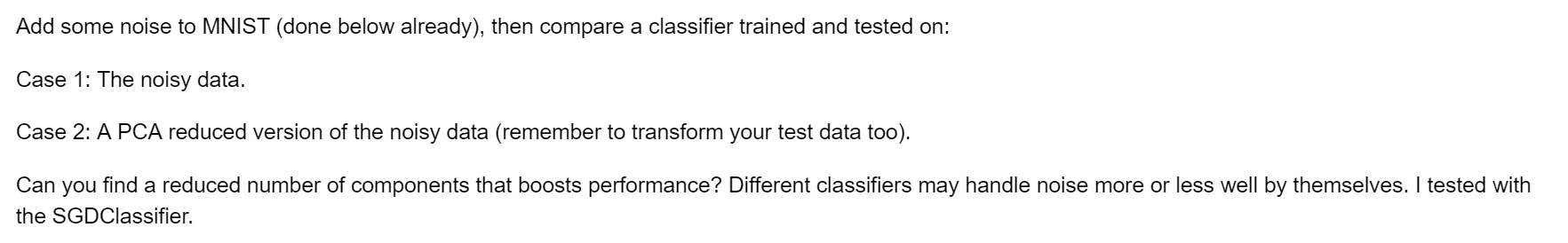


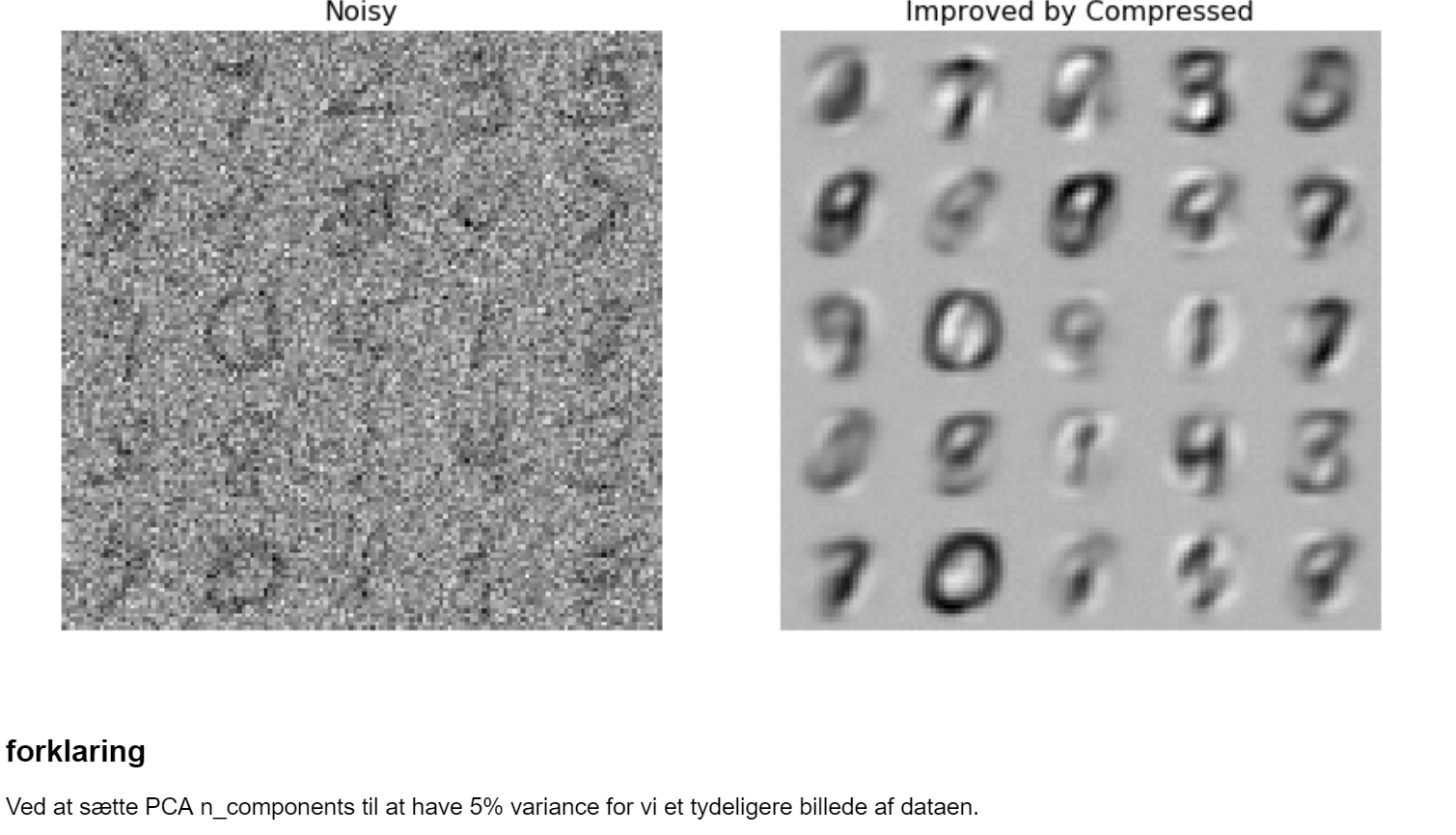
# Speed up by compression

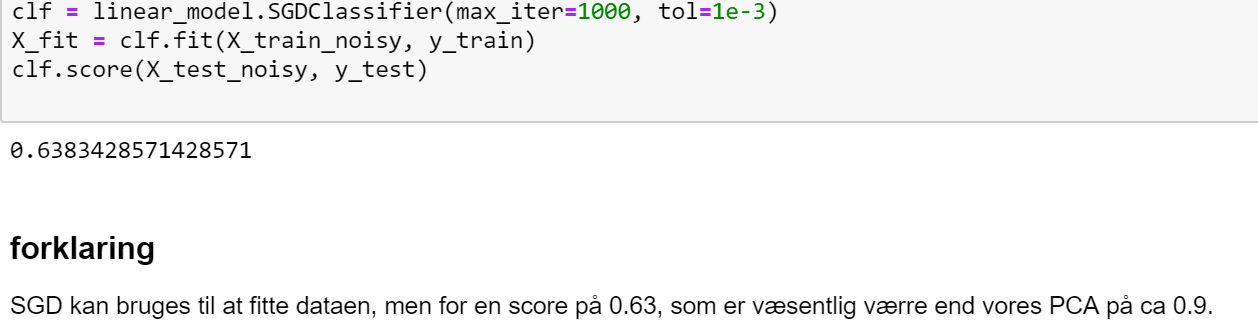




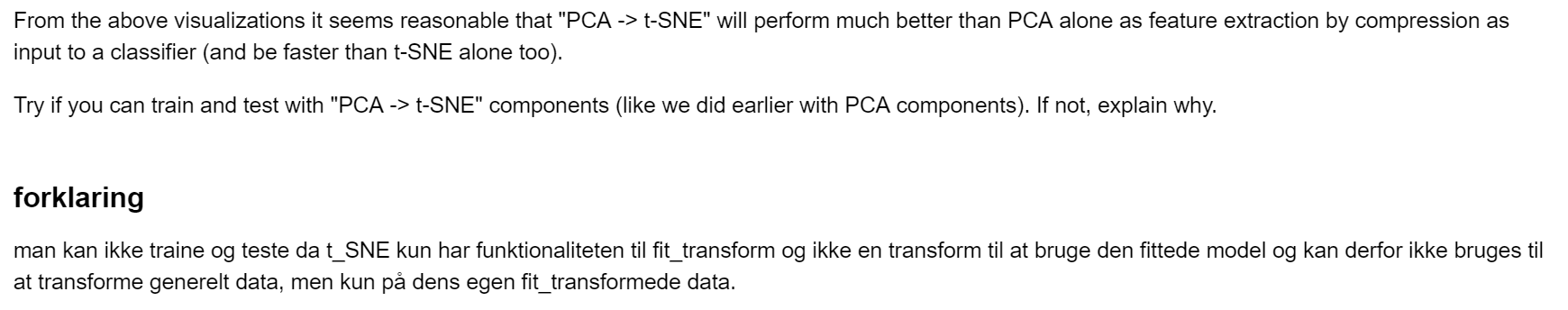
# Noise reduction





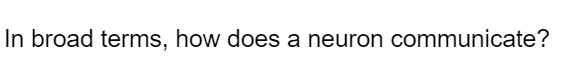


# PCA -> t-SNE features

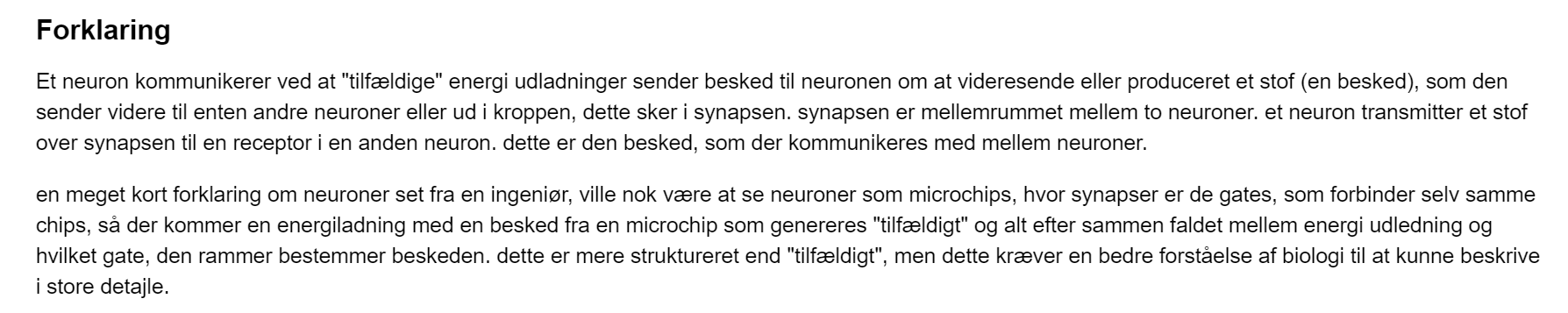


# Neurons

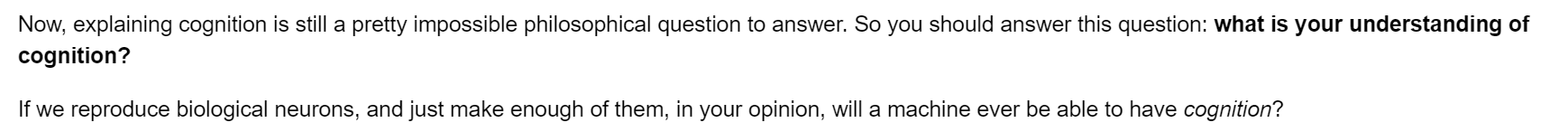
## Qa:

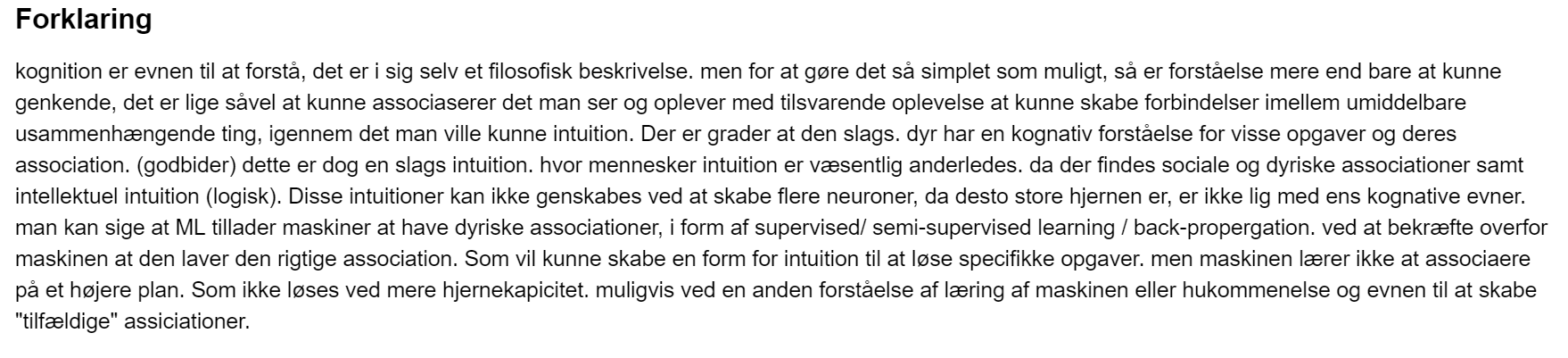






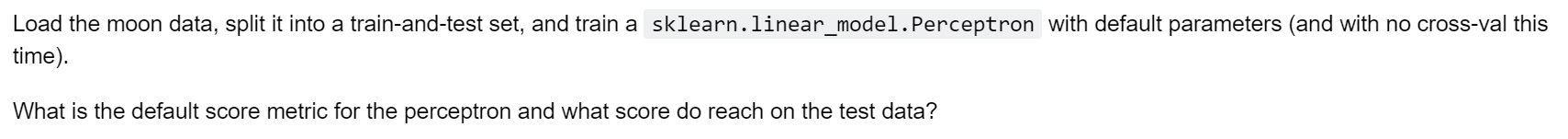
## Qb:

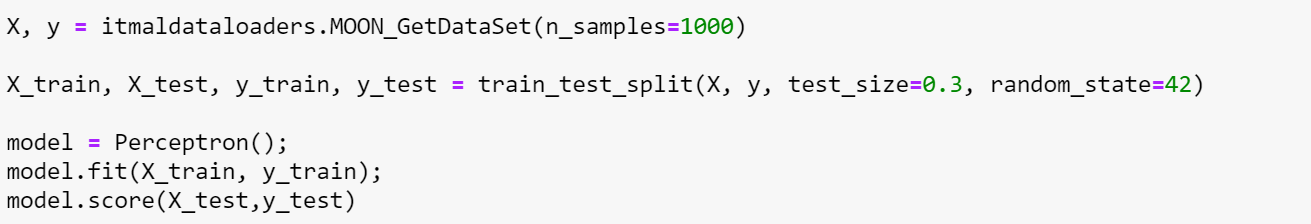


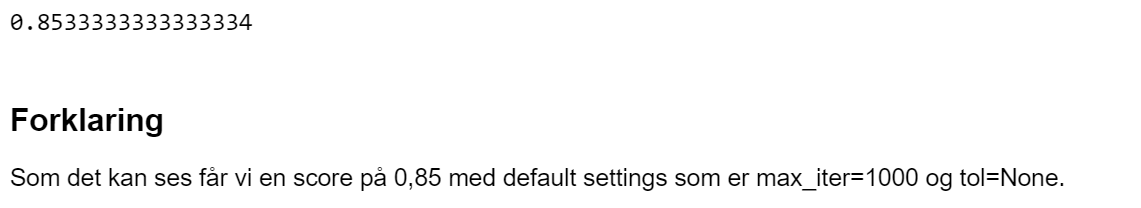


# Perceptron

## Qa:

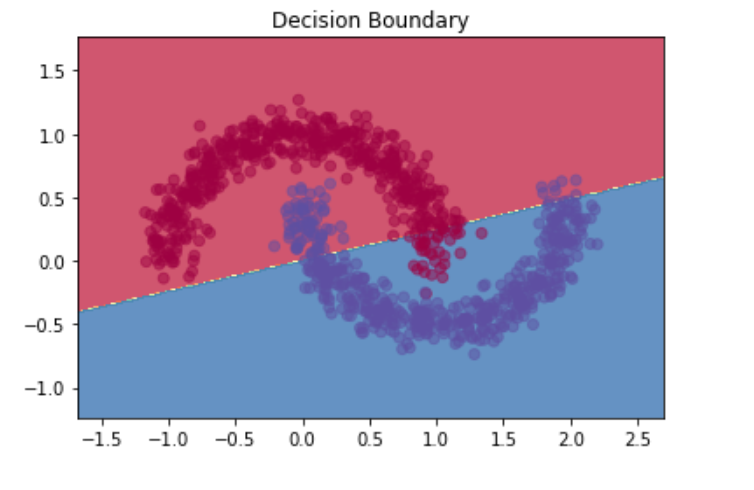




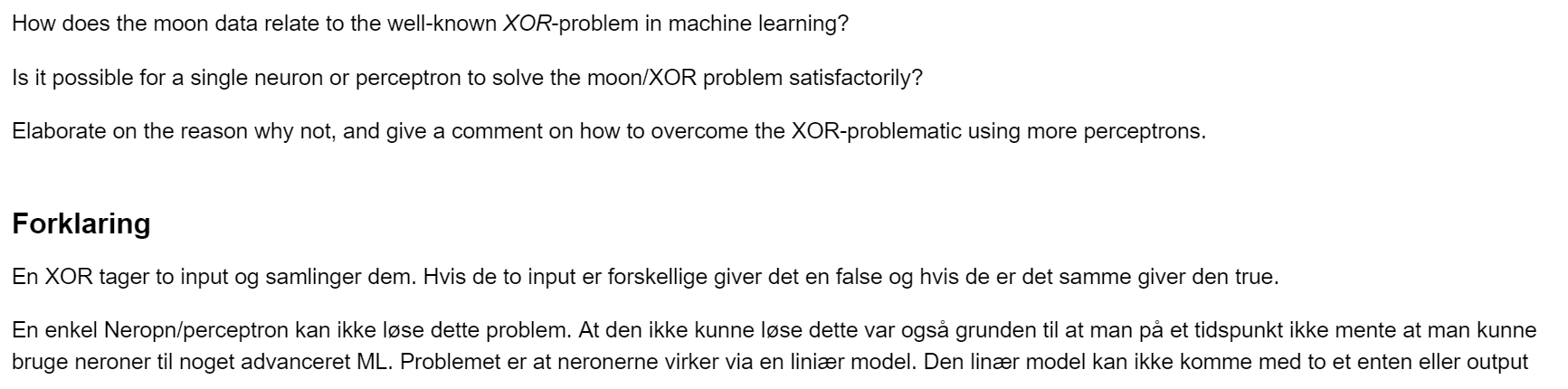


## Qb:

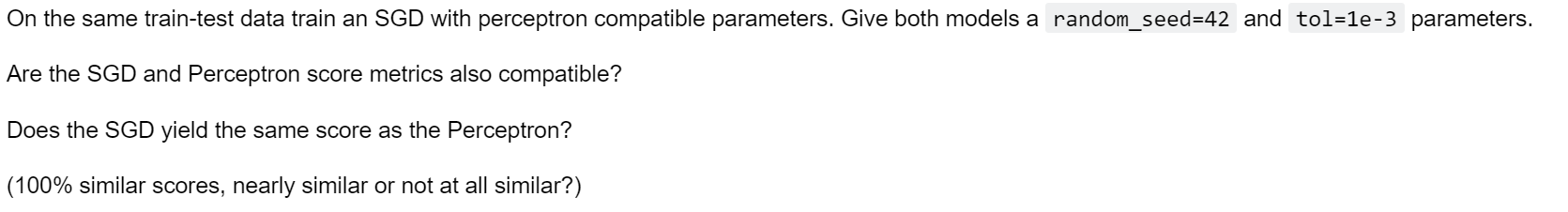


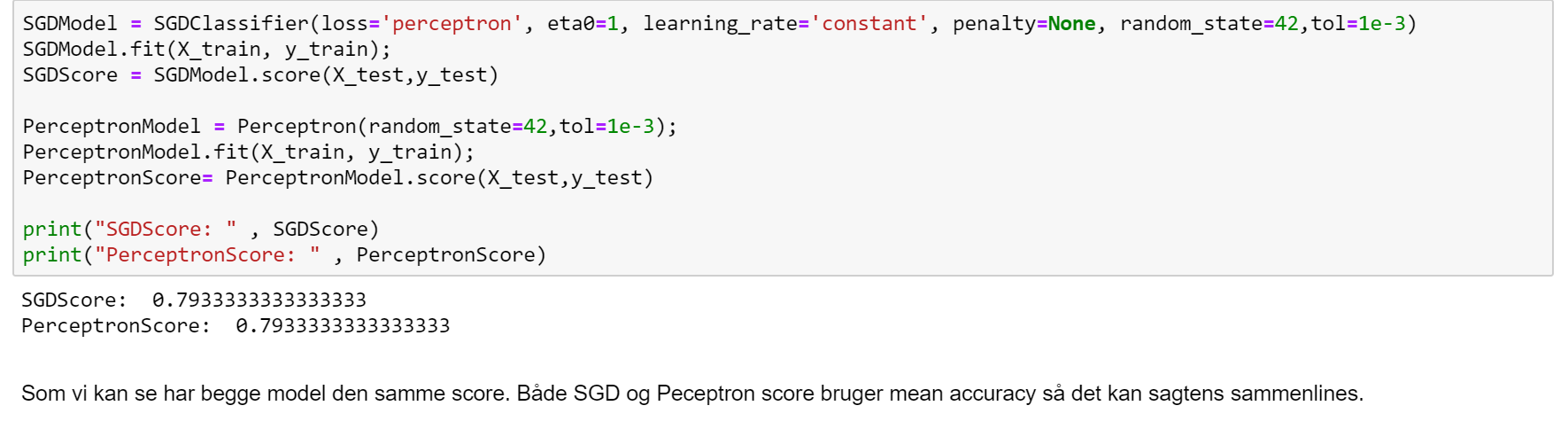


## Qc:



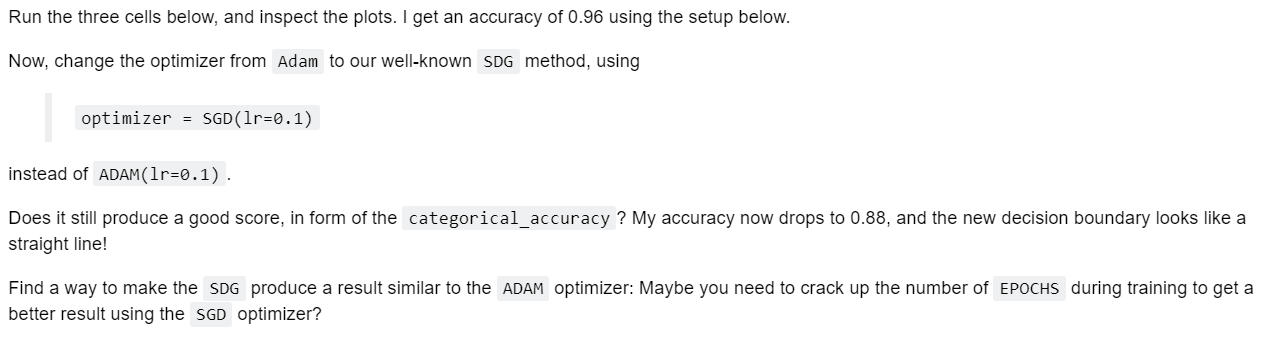
## Qd:





# Multi-layers Perceptrons (MLP)

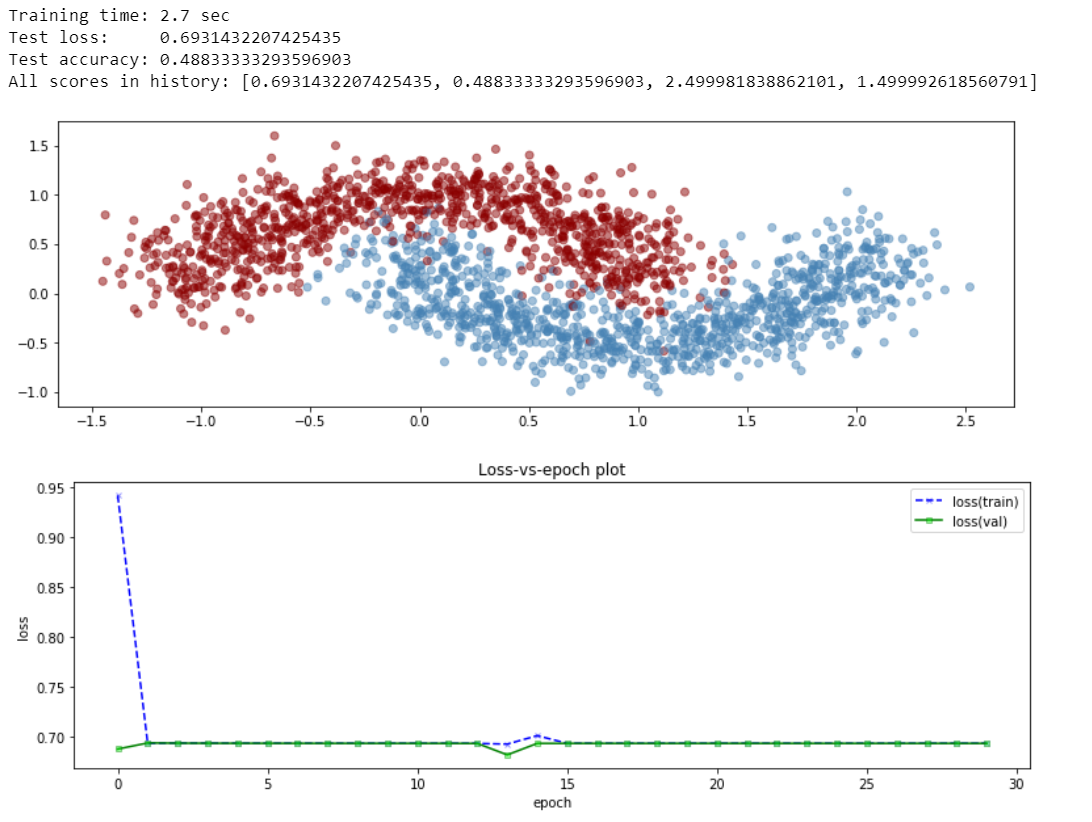
## Qa:

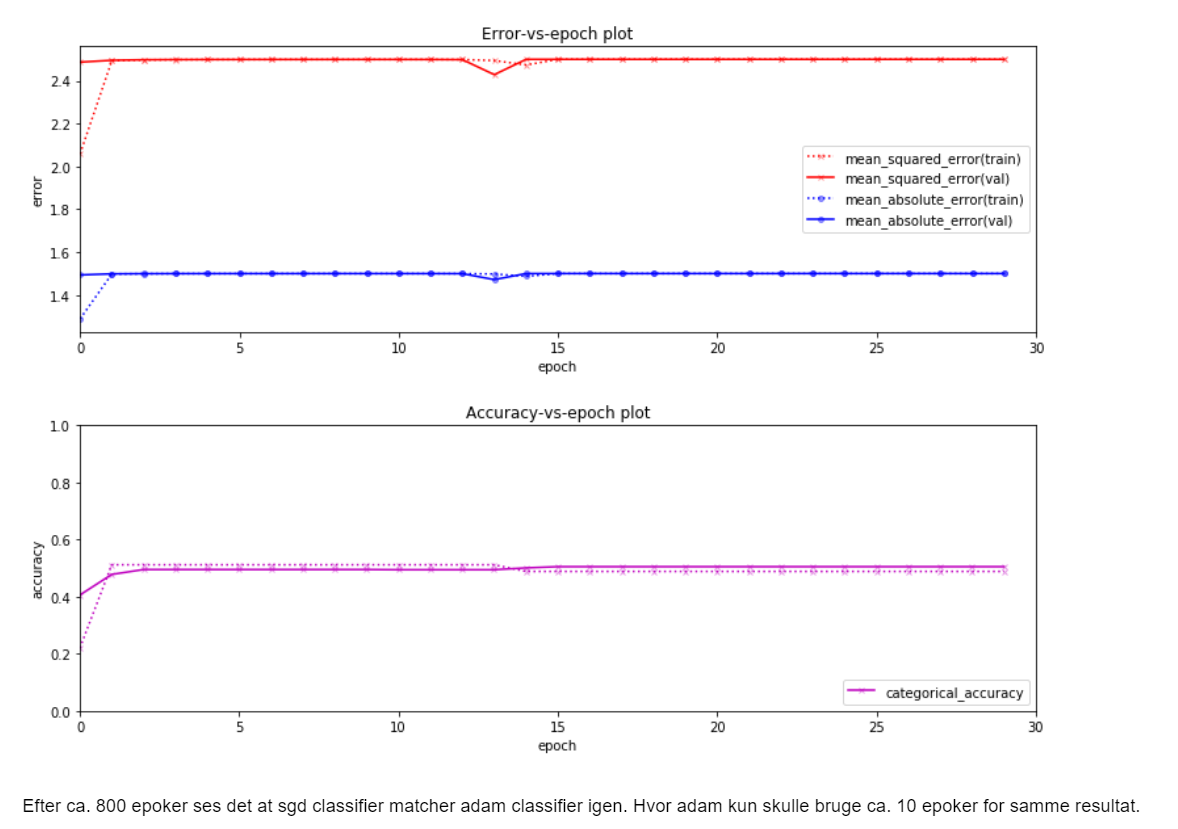


### Cell 1

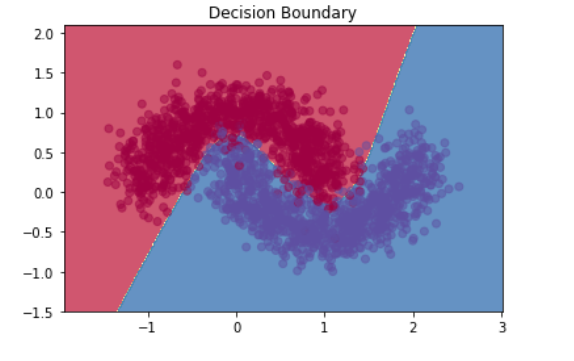


### Cell 2

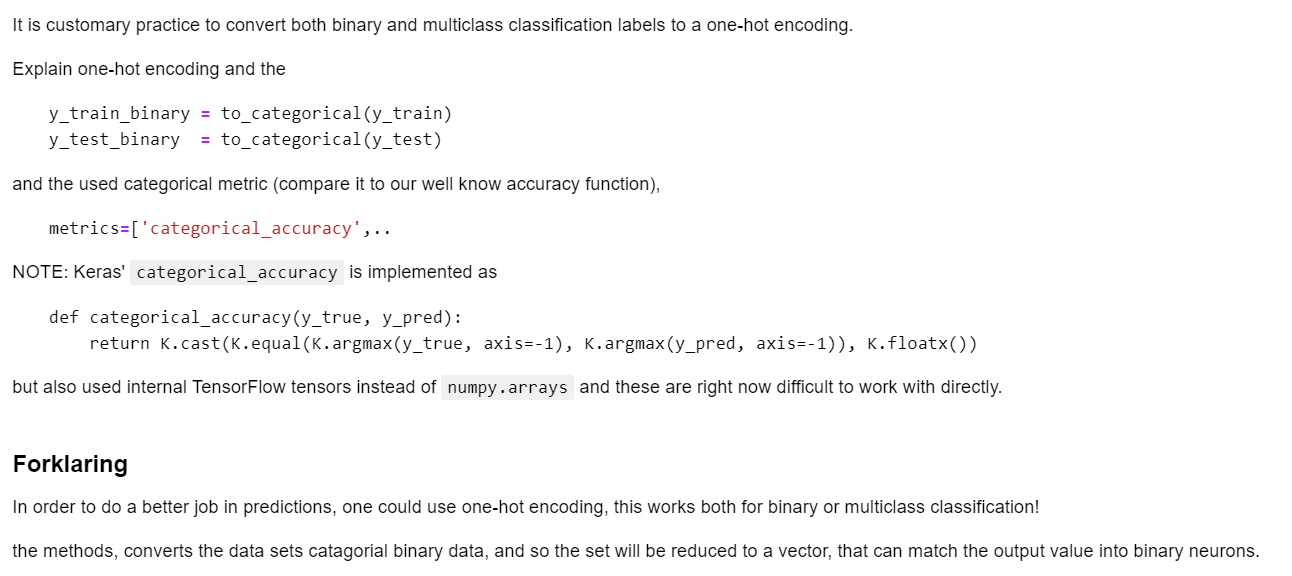




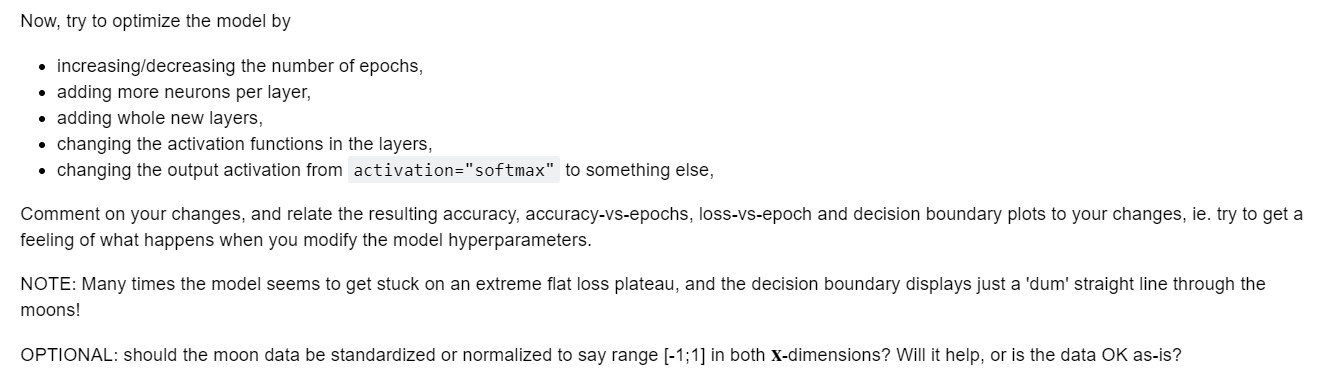
### Cell 3



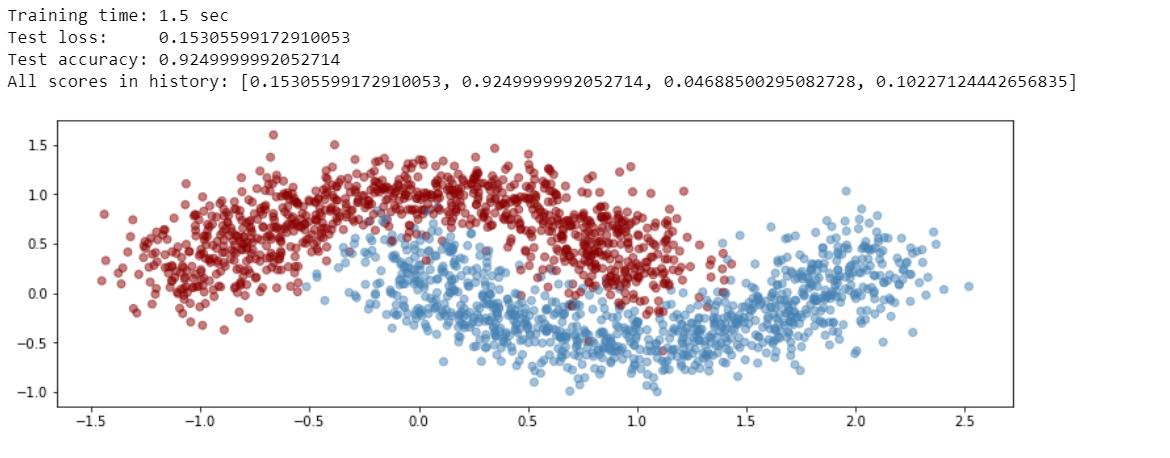
## Qb:

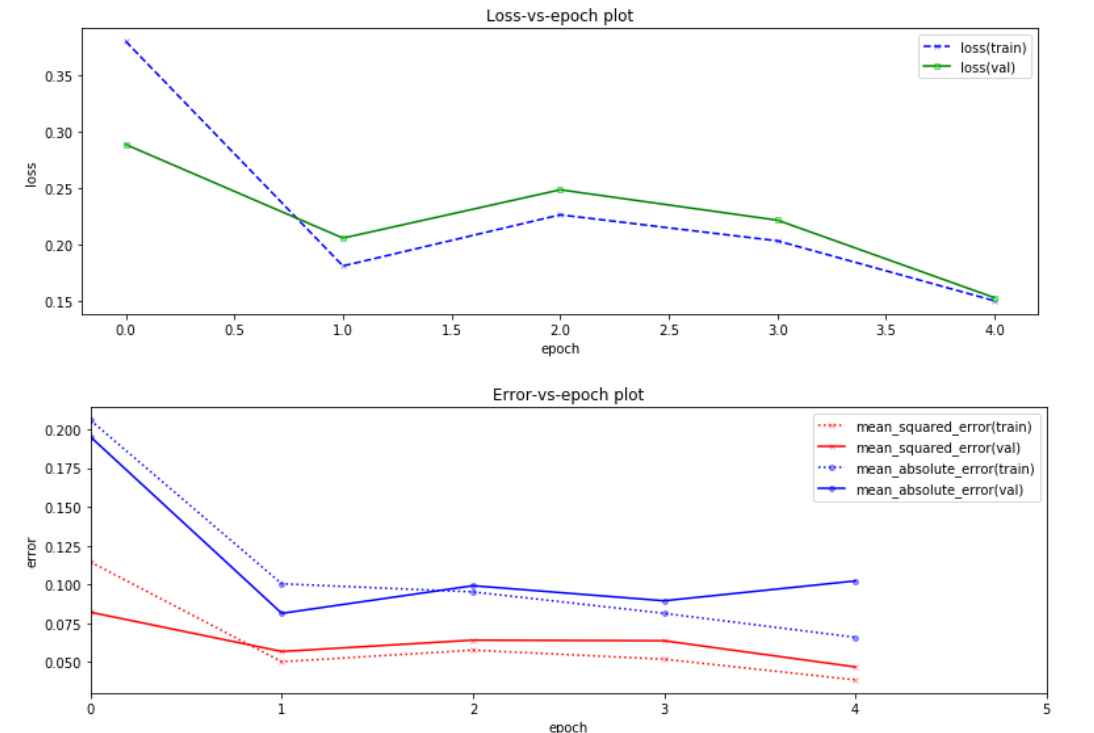


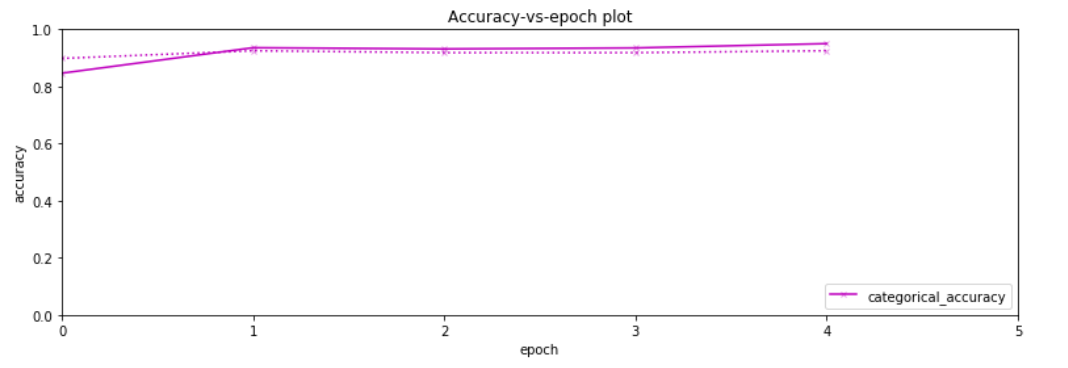
## Qc:

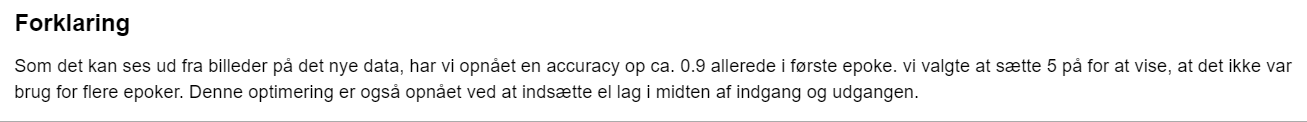












# Keras Multi-Layer Perceptrons (MLP’s) on MNIST-data

Here we will once again work on the MNIST data, but this time using layers and perceptrons, wich is a very different way of computing outputs than before.

Using biology, more exact the the human brain as fundamental concept of this, we will use the neurons as a kind of “best optimized” to solve the problems as training.

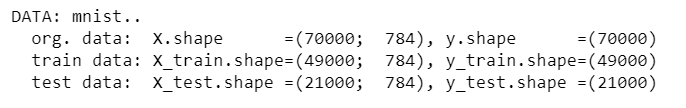
The key concept here, is to try and optimize the number of epochs, and adding in more units, as well as layers. Optimizing all these in relation to each other, will resolve in much faster computing rates.

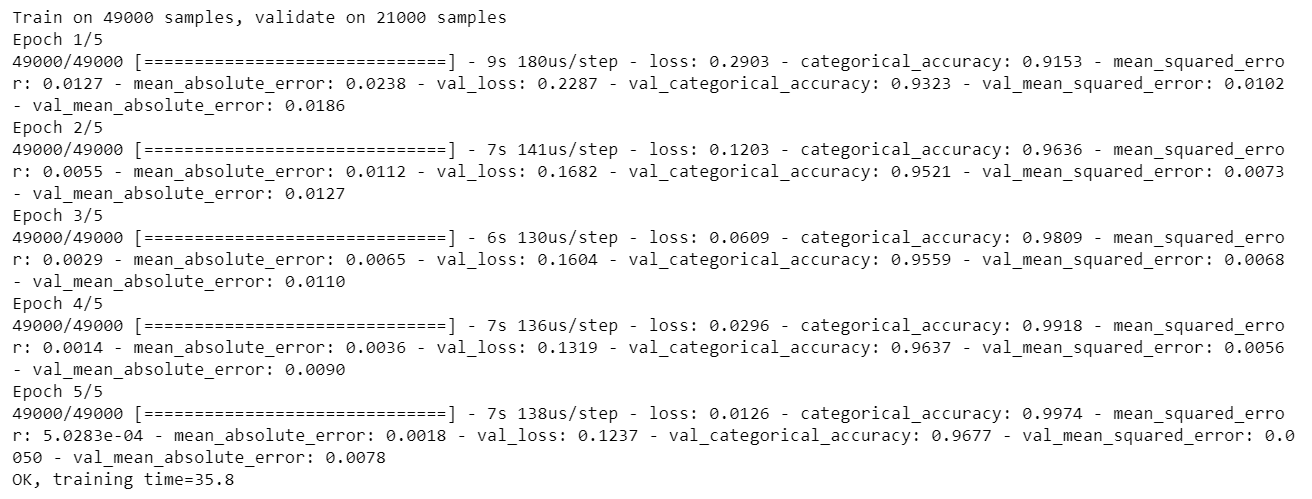
## Qa:

In this first question, we will use the concepts explained in the introduction, to redo the mlp\_moon.ipynb, into this new way of thinking.

The code below, shows our implementation of this, as seen, we found that 3 layers was enough for realizing this. Finally, we used 5 epochs, however, 1 epoch was really enough, but we needed to demonstrate a little more, that the epochs was still making a bit better.

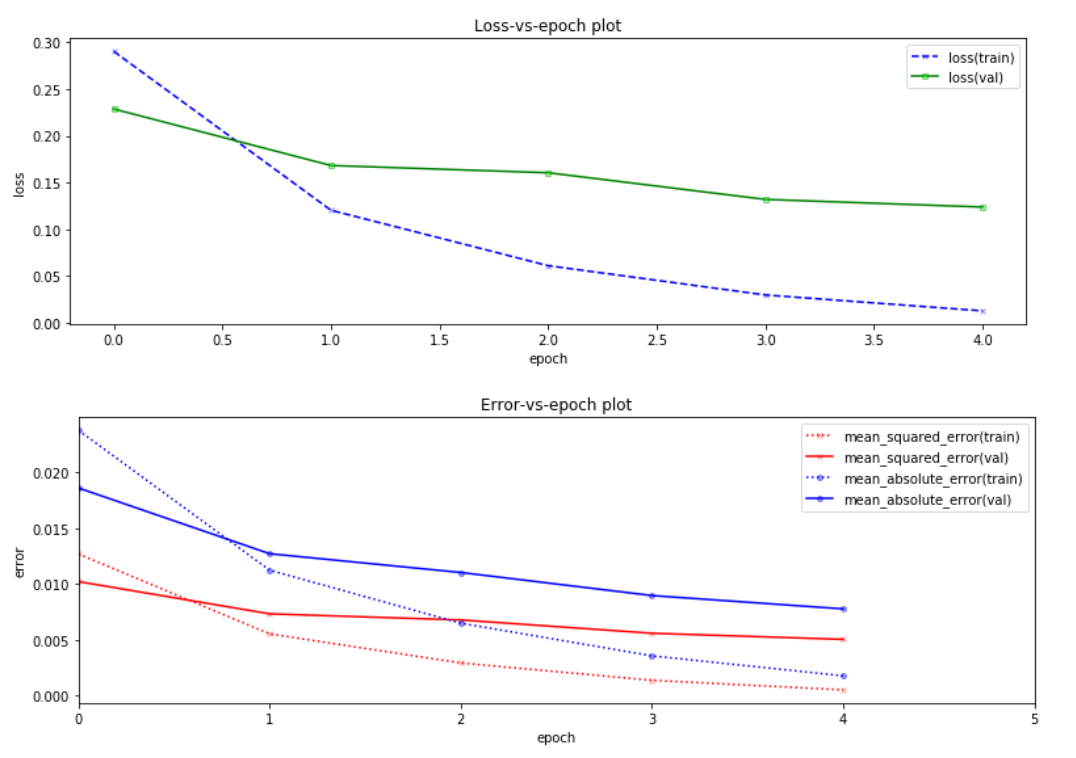




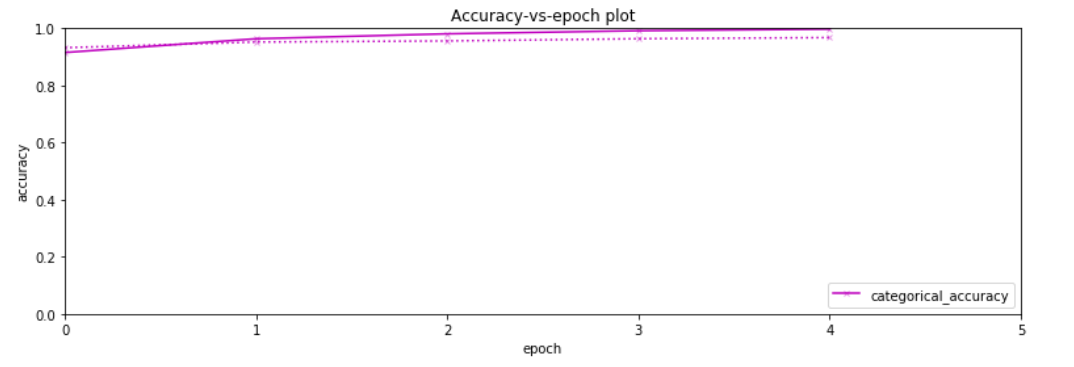


After running the code, the training time is shown below, because we used 5 epocs the training time got a little steep, we ended up using 35 seconds realizing this data, on the GPU cluster.

Plotting below is the loss-vs-epoch and the error-vs epoch, as shown. The training data is getting really good, however the test data is only getting slightly better per epoch, this is a scenario of overfitting, but we think, that its still within acceptable ranges.

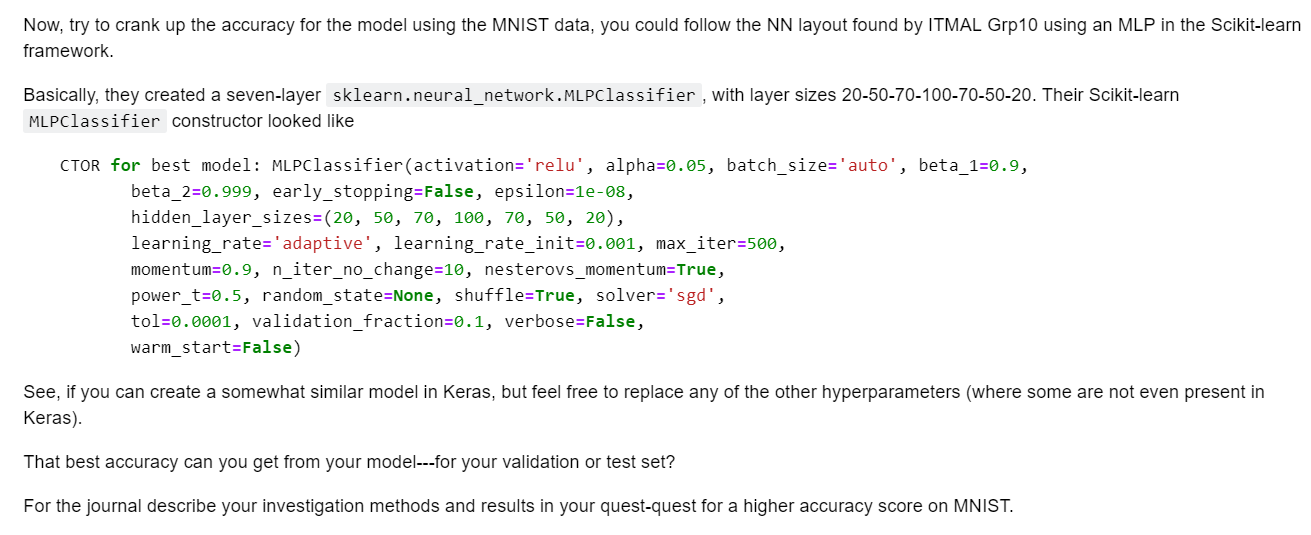


Finally, after all we get an accuracy that is quite acceptable. We are getting really near one here



## Qb:

Now we will use the MLPClassifier with the parameters from group 10, and try to work on this data the same way as before.



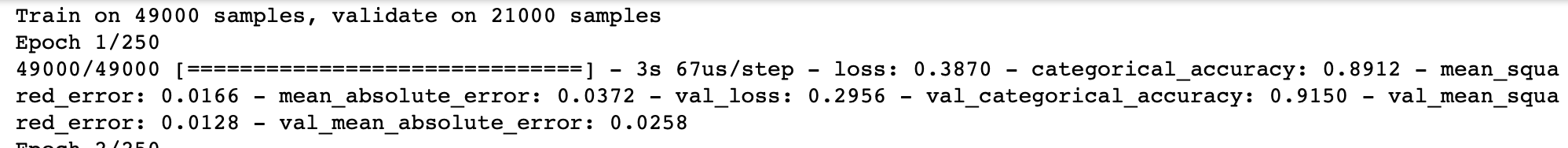


### 

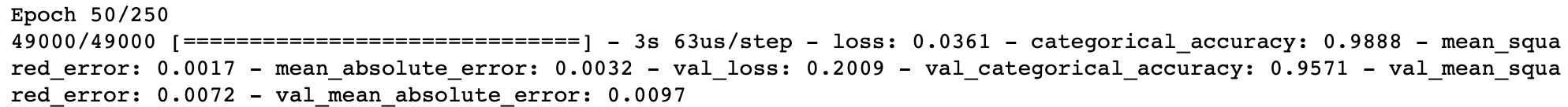
After running epoch number 250, we have finished our training.

We then have an accuracy of 0.996 and an MSE of 0.0013 on the training data.

We also have and accuracy of 0.9579 and an MSE of 0.073 on the test data. Comparing to the first epoch (see below)



And the 50. Epoch (see below)

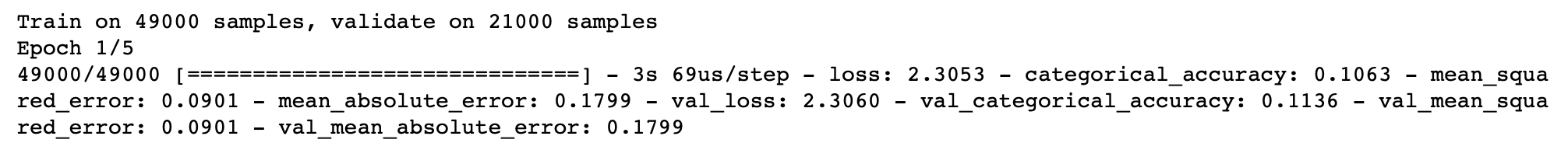


We can see that 1 epoch was not really so good with a MSE on test data of 1.2 % but if we go the the 50. Epoch we are ok at MSE on test data at 0.7%. we could have concluded the training here, as it is almost the same as for epoch 250.

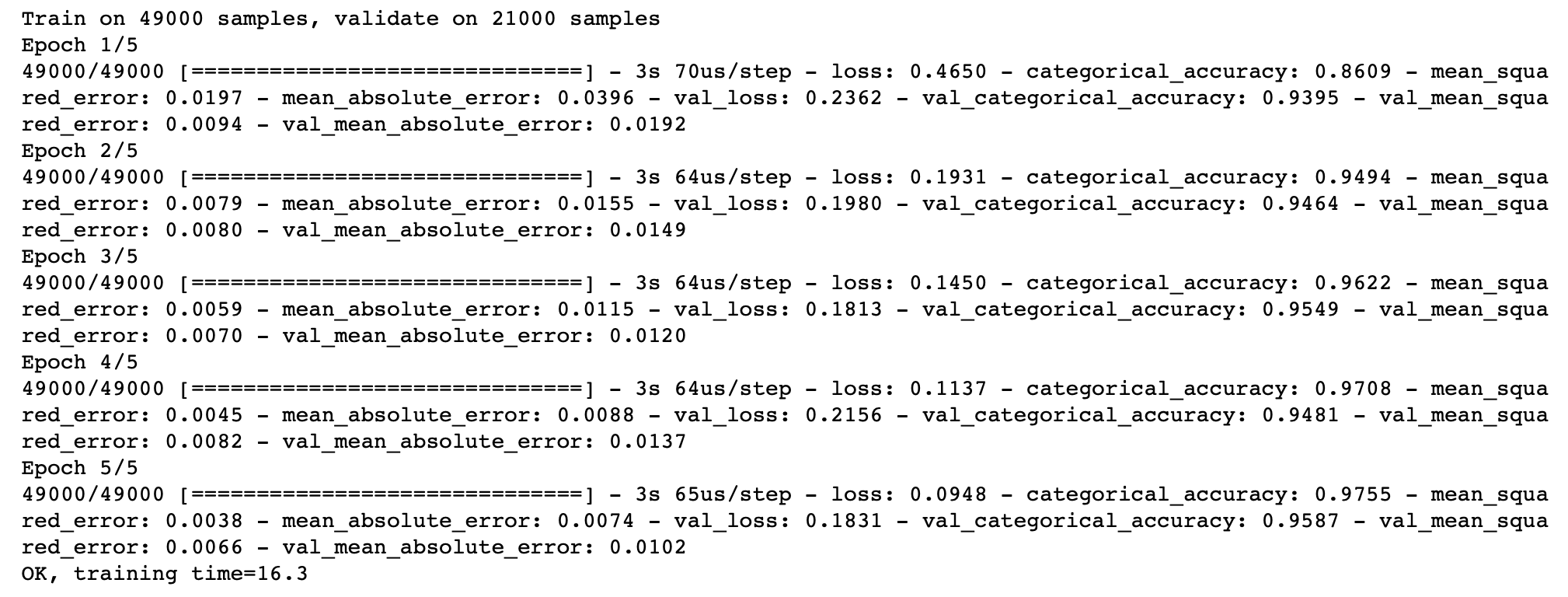
**Different activasions**

All the different activations we tried out, still ended in a soft max activation at the last layer.

Other than tanh activations ( in the code ) we also tried sigmoid, which gave a much worse performance already at epoch 1 (see below ) , we tried using this activation since sigmoid is also a nonlinear activation function



Lastly, we tried relu activation, which proved to be the most optimal (see below)



As seen we got test accuary of 96% and MSE of 0.6%, which is already better than the tanh activation, at epoch 250 compared to epoch 5 on relu activation.

**Conclusion**

In conclusion, our error rate is slightly higher on the test data, than on the training data, which could lead to some overfitting, however this is still so low, that we see it as not relevant. The whole training took 928 seconds of runtime on the GPU cluster, which is due to the many epochs, where we could have ended at epoch 50.

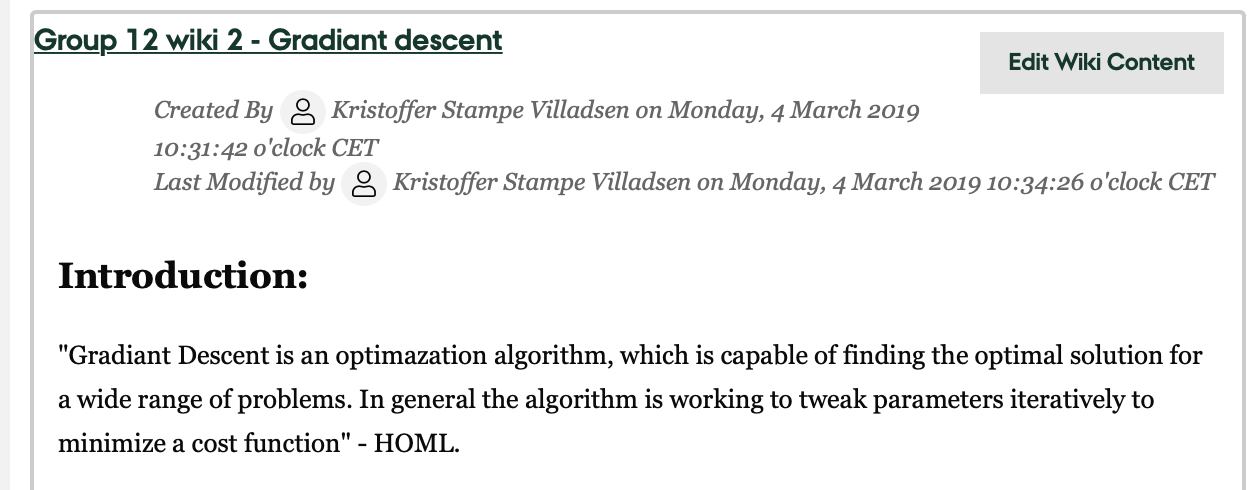
As seen in the section where we try different activations, the one using relu, proved much better, and concluded in 16 second at 5 epochs, with an MSE on test data at only 0.6%.

# Wiki – Gradient Decent

For the wiki, we chose to dig deeper into the concept of the algorithm which is gradient decent. Here we both talk about the stochastic gradient decent, as well as the mini-batch gradient decent.

A snapshot of the wiki, is seen on the image below, followed by the full wiki.

The Wiki can also be seen directly on the wiki’s page at <https://blackboard.au.dk/webapps/Bb-wiki-BBLEARN/wikiView?course_id=_124256_1&wiki_id=_75492_1&page_guid=bbddb5e60c9349b385069368ce1d8f9e>



## Introduction to Gradient decent

"Gradiant Descent is an optimazation algorithm, which is capable of finding the optimal solution for a wide range of problems. In general the algorithm is working to tweak parameters iteratively to minimize a cost function" - HOML.

When looking at Gradiant Descent we need to understand that the algorithm is working on a cost function, the cost function is a function looking at the models MSE, as long as the cost function has a gradiant steeper than zero, it should keep looking for a minimum where the gradiant is zero. 

This is done by using random initialization, where a random theta value for the model is chosen and than the algorithm starts looking for where the gradiant is steepest and than, dependent on the learning rate, it start working towards a minimum.

the learning rate, is a parameter of our algorithm. it has an effect on how 'fast' we travels towards zero. But it doesn´t mean that a high learning rate is the answer. a high learning rate, might cause our algorithm to miss the minimum and have us end up further from our goal. and a to small learning rate, might have us never reaching our minimum. 

as we stated the algorithm should keep looking for a minium until it reaches zero. but this is specified by yet another parameter. Making sure we don't use all processing on an constant search for a minimum we have a limit of operations before we state we have reach a minimum.

with this knowledge we can look a little closer at some problems and some deviations of GD.

if a cost function is more complex, and may have a more than one minimum or have plateau. than we will have problems with our basic version of GD, this basic version of GD is also called Batch Gradiant Descent.

a possible way of solving the problems of local minimas and plateaus is using the deviation of GD called Stochastic GD

## The stochastic gradient decent

 The problem that we a faced with using BGD is that from our random initialization is that it computes every step from that point, and works only from that point. this can be an extremely slow process and might not be reliable with the concern of multiple minimas. an alternativ to deal with these problems we could use Stochastic GD or SGD. This model takes an random stance for every point it works and looks for the steepest gradiant. this results in an more volitale start, but will be faster to find a shallow gradiant, but as a result of always picking a random stance it is far more likely to never end on the absolute minimum value. but it will be better at avoiding local minimas and plateaus.

## The Mini-batch gradient decent

When looking for the best model, best of both worlds is a possiblity. instead of looking at every point in the data set as in batch or taking a random point only mini-batch takes a smaller set of the total set. the smaller set is randomly chosen, taking the advantages from SGD, and works on every point in the smaller set taking the advantages from BGD. the advantage of MBGD is it gets closer than SGD and is faster than BGD, but is also inherits the problems from both. It can have problems with local minimas in some cases. and it will most likely never reach the absolute minimum value even if it ends up close to the global minimum.

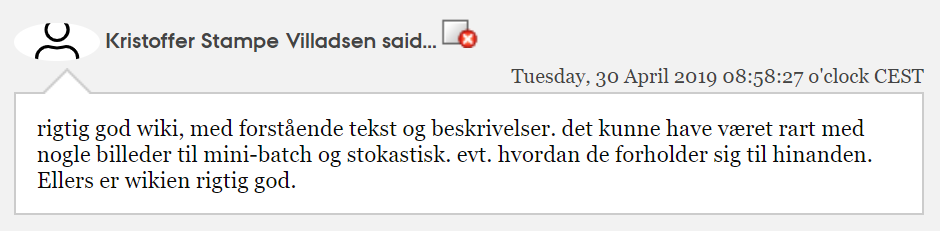
## References

HOML - 4. Training Models - Gradiant Descent

# Comments of others wiki’s

In this section, we will post the snapshots of comments to the wiki’s of others groups.

## Gruppe 45 – Gradient decent



## Gruppe 1 – supervised/ unsupervised

