ITMAL

Hand-in: Journal 2

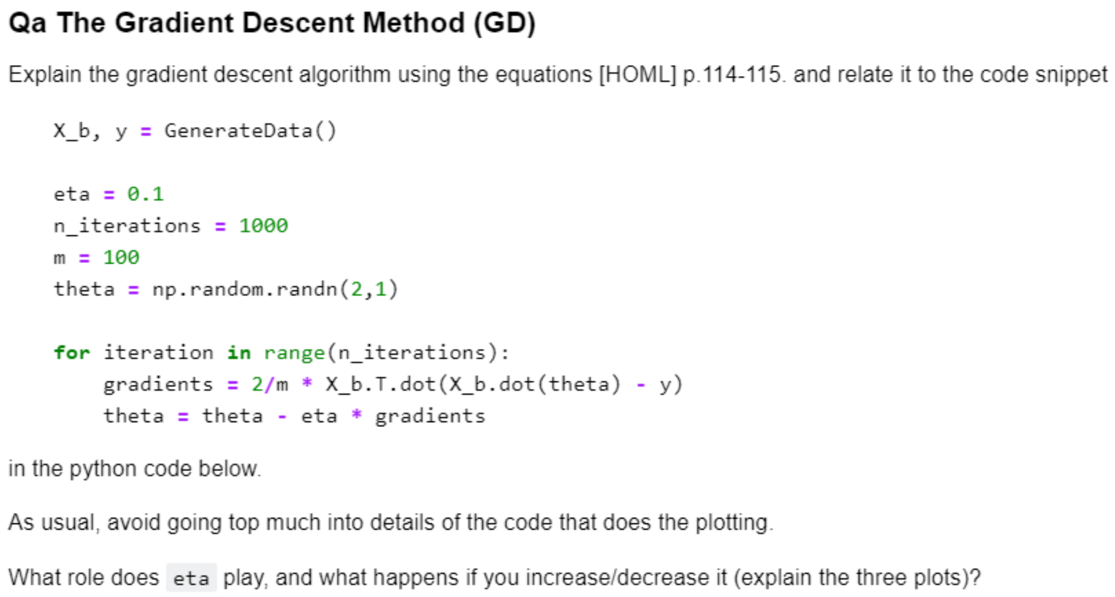
Af Gruppe 12

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

Indhold

Gradient descent

Qa:



Qa The Gradient Descent Method (GD)

Explain the gradient descent algorithm using the equations [HOML] p.114-115. and relate it to the code snippet

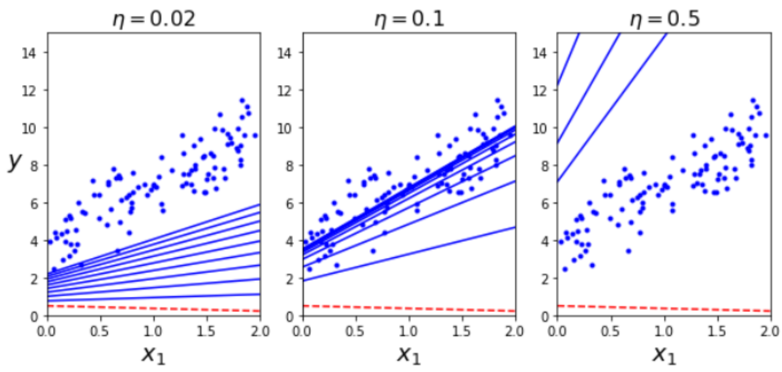
```
X_b, y = GenerateData()

eta = 0.1
n_iterations = 1000
m = 100
theta = np.random.randn(2,1)

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

in the python code below.

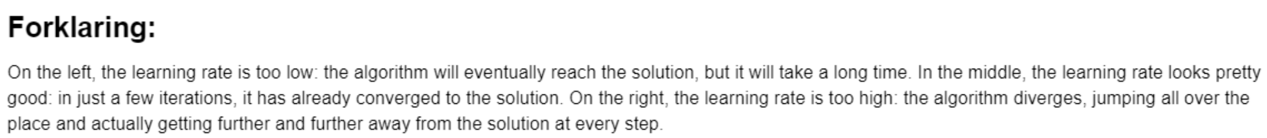
As usual, avoid going top much into details of the code that does the plotting.

What role does `eta` play, and what happens if you increase/decrease it (explain the three plots)?

```
stochastic gradient descent theta=[3.86865647 2.98760205]
```



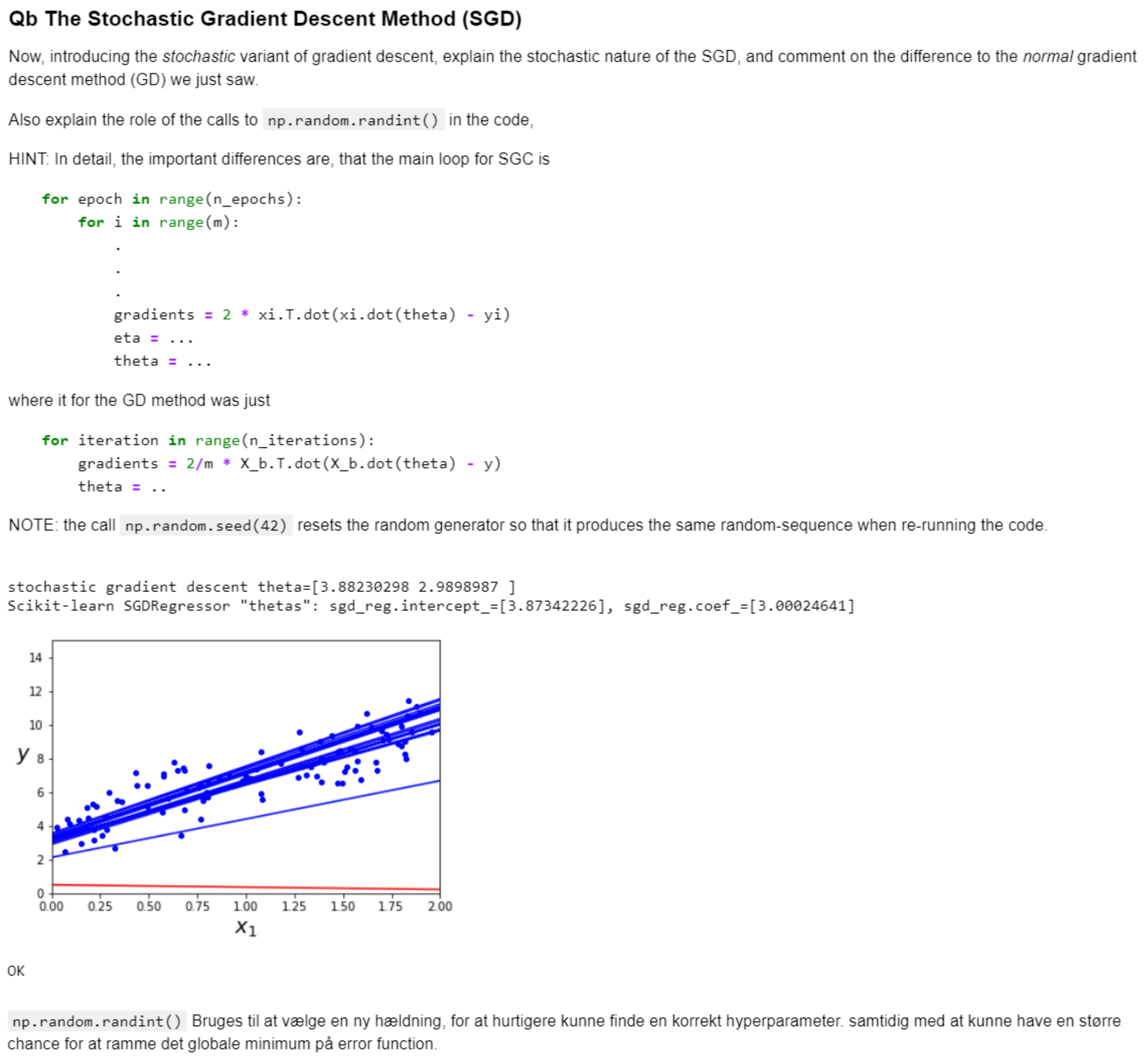
```
OK
```



Forklaring:

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution. On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

Qb:



Qb The Stochastic Gradient Descent Method (SGD)

Now, introducing the *stochastic* variant of gradient descent, explain the stochastic nature of the SGD, and comment on the difference to the *normal* gradient descent method (GD) we just saw.

Also explain the role of the calls to `np.random.randint()` in the code,

HINT: In detail, the important differences are, that the main loop for SGC is

```
for epoch in range(n_epochs):
    for i in range(m):
        .
        .
        .
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = ...
        theta = ...
```

where it for the GD method was just

```
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = ..
```

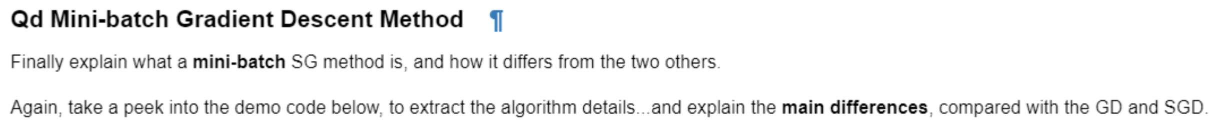
NOTE: the call `np.random.seed(42)` resets the random generator so that it produces the same random-sequence when re-running the code.

```
stochastic gradient descent theta=[3.88230298 2.9898987 ]
Scikit-learn SGDRegressor "thetas": sgd_reg.intercept_=[3.87342226], sgd_reg.coef_=[3.00024641]
```

```
OK
```

`np.random.randint()` Bruges til at vælge en ny hældning, for at hurtigere kunne finde en korrekt hyperparameter. samtidig med at kunne have en større chance for at ramme det globale minimum på error function.

Qd:

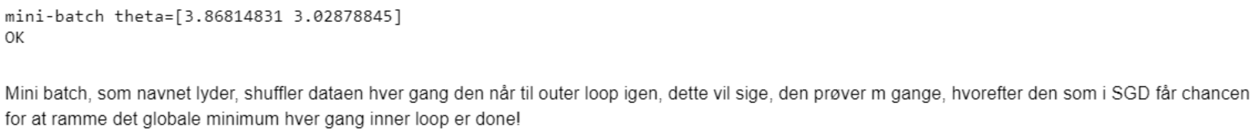


Qd Mini-batch Gradient Descent Method ¶

Finally explain what a **mini-batch** SG method is, and how it differs from the two others.

Again, take a peek into the demo code below, to extract the algorithm details...and explain the **main differences**, compared with the GD and SGD.

Forklaring:



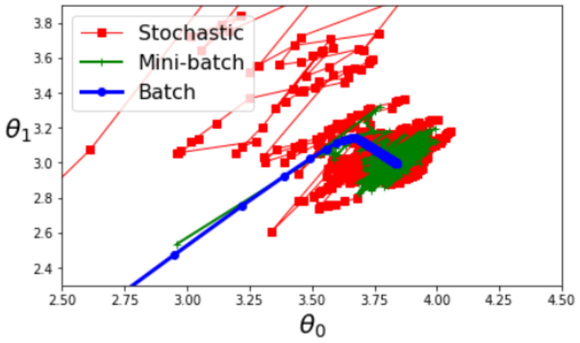
```
mini-batch theta=[3.86814831 3.02878845]
OK
```

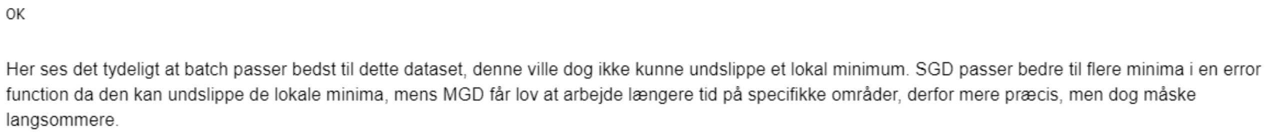
Mini batch, som navnet lyder, shuffler dataen hver gang den når til outer loop igen, dette vil sige, den prøver m gange, hvorefter den som i SGD får chancen for at ramme det globale minimum hver gang inner loop er done!

Qe:

Qe Choosing a Gradient Descent Method

Explain the $\theta_0 - \theta_1$ plot below, and make a comment on when to use GD/SGD/mini-batch gradient descent (pros and cons for the different methods).



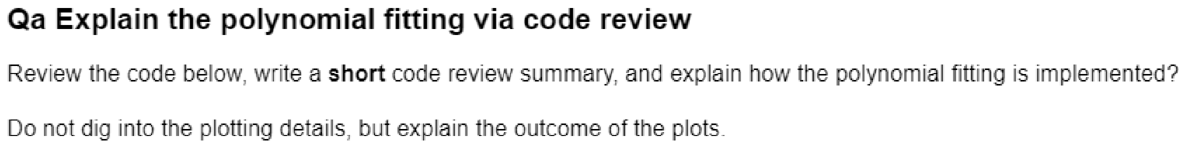


OK

Her ses det tydeligt at batch passer bedst til dette dataset, denne ville dog ikke kunne undslippe et lokal minimum. SGD passer bedre til flere minima i en error function da den kan undslippe de lokale minima, mens MGD får lov at arbejde længere tid på specifikke områder, derfor mere præcis, men dog måske langsommere.

Capacity under overfitting

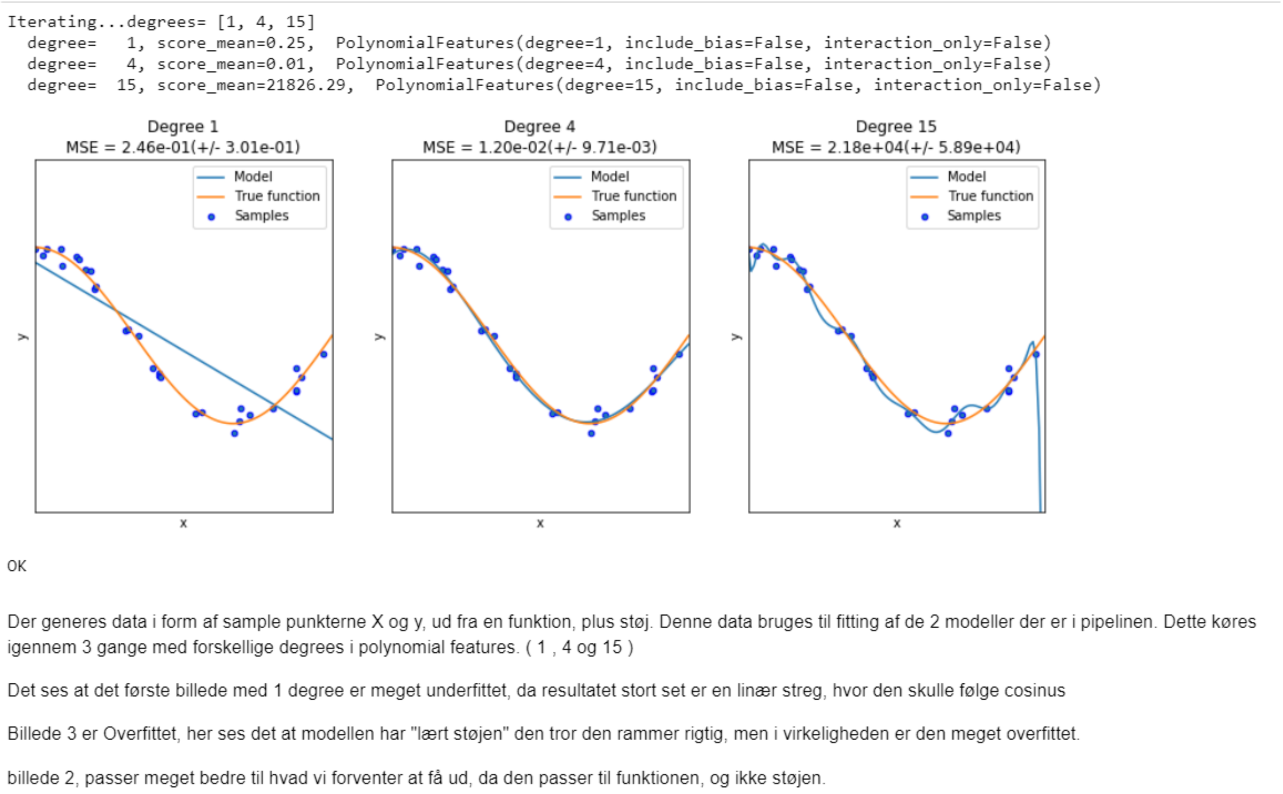
Qa:



Qa Explain the polynomial fitting via code review

Review the code below, write a **short** code review summary, and explain how the polynomial fitting is implemented?

Do not dig into the plotting details, but explain the outcome of the plots.



```
Iterating...degrees= [1, 4, 15]
  degree=   1, score_mean=0.25,  PolynomialFeatures(degree=1, include_bias=False, interaction_only=False)
  degree=   4, score_mean=0.01,  PolynomialFeatures(degree=4, include_bias=False, interaction_only=False)
  degree=  15, score_mean=21826.29,  PolynomialFeatures(degree=15, include_bias=False, interaction_only=False)
```

```
OK
```

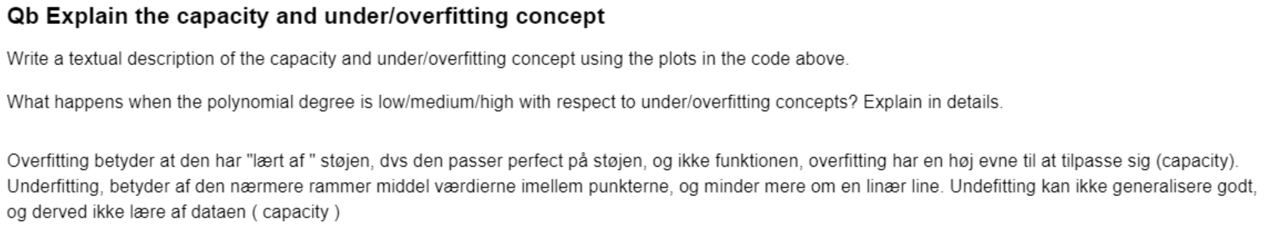
Der generes data i form af sample punkterne X og y, ud fra en funktion, plus støj. Denne data bruges til fitting af de 2 modeller der er i pipelinen. Dette køres igennem 3 gange med forskellige degrees i polynomial features. (1 , 4 og 15)

Det ses at det første billede med 1 degree er meget underfittet, da resultatet stort set er en linær streg, hvor den skulle følge cosinus

Billede 3 er Overfittet, her ses det at modellen har "lært støjen" den tror den rammer rigtig, men i virkeligheden er den meget overfittet.

billede 2, passer meget bedre til hvad vi forventer at få ud, da den passer til funktionen, og ikke støjen.

Qb:



Qb Explain the capacity and under/overfitting concept

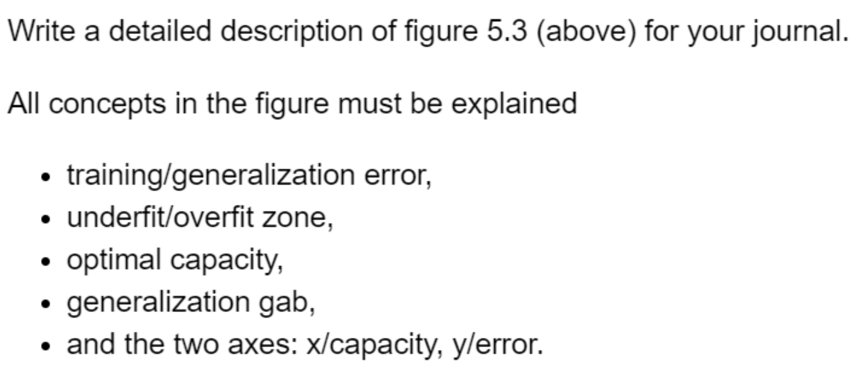
Write a textual description of the capacity and under/overfitting concept using the plots in the code above.

What happens when the polynomial degree is low/medium/high with respect to under/overfitting concepts? Explain in details.

Overfitting betyder at den har "lært af " støjen, dvs den passer perfect på støjen, og ikke funktionen, overfitting har en høj evne til at tilpasse sig (capacity). Underfitting, betyder af den nærmere rammer middel værdierne imellem punkterne, og minder mere om en linær line. Undefitting kan ikke generalisere godt, og derved ikke lære af dataen (capacity)

Generalization error

Qa:



Write a detailed description of figure 5.3 (above) for your journal.

All concepts in the figure must be explained

- training/generalization error,
- underfit/overfit zone,
- optimal capacity,
- generalization gab,
- and the two axes: x/capacity, y/error.

Forklaring

Generalisation error er hvor god algoritmen er til at predicte ud fra test data. Training error, er hvor god algoritmen er til at predicte ud fra trænings data.

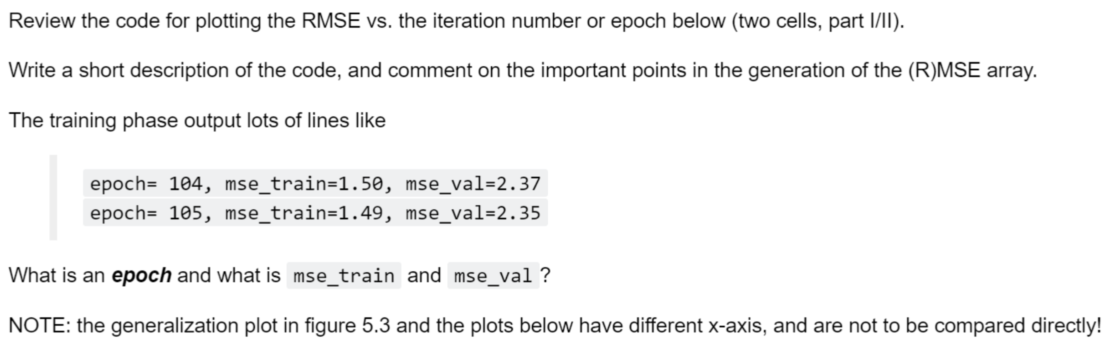
Underfitting zone, er hvor generalisation error og traning error er høj, fordi, da der ikke er fittet nok endnu.'

Optimal capacity, er hvor underfitting og overfitting er lige tilpas, så errors ikke er særlige høje på både test og trænings data

Overfitting zone, er hvor der er fittet for meget, og derfor passer "for godt" til træningsdataen, men passer rigtig dårlig på nyt data(test data)

Generalization gab, Jo mere overfittet, jo størrer gab, da det bliver alt for specifi

Qb:



Review the code for plotting the RMSE vs. the iteration number or epoch below (two cells, part I/II).

Write a short description of the code, and comment on the important points in the generation of the (R)MSE array.

The training phase output lots of lines like

> ```
> epoch= 104, mse_train=1.50, mse_val=2.37
> epoch= 105, mse_train=1.49, mse_val=2.35
> ```

What is an ***epoch*** and what is `mse_train` and `mse_val` ?

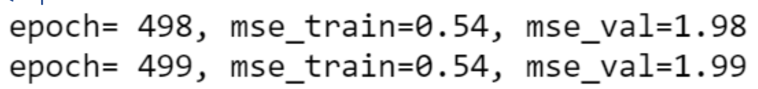
NOTE: the generalization plot in figure 5.3 and the plots below have different x-axis, and are not to be compared directly!

```
X_train.shape= (25, 1)
X_val  .shape= (25, 1)
y_train.shape= (25,)
y_val  .shape= (25,)
OK
```

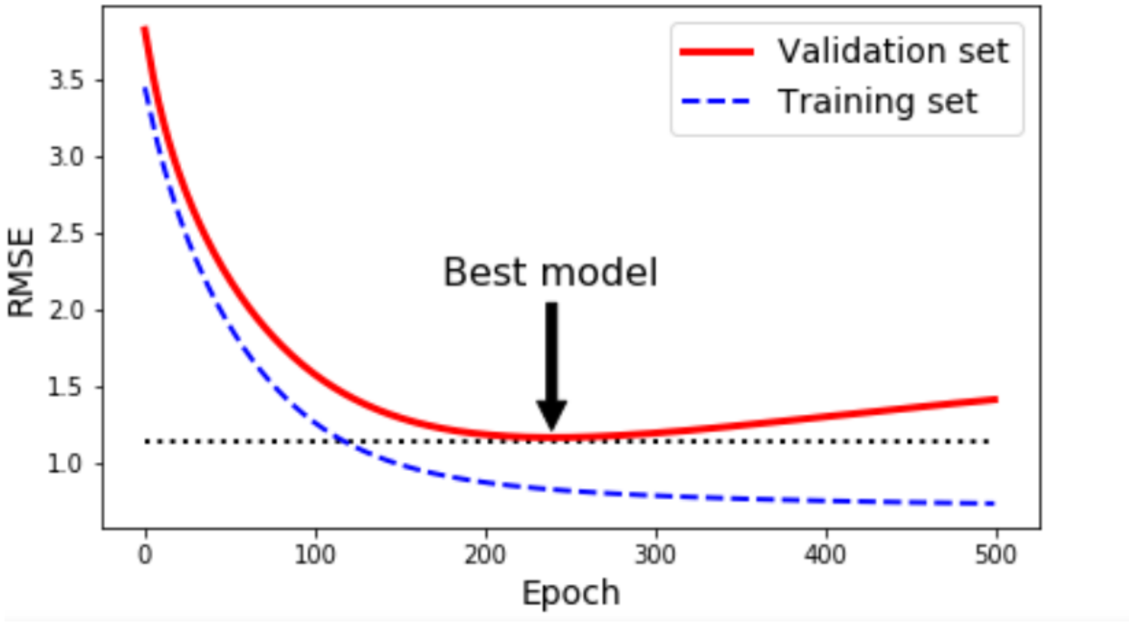
Epoch is the iteration it goes through in the descent MSE_Train is the Mean Square Error in the training set at the epoch MSE_val is the Mean Square Error in the test set at the epoch

poly_scaler.fit_transform, finder ud af hvad der skal til for at komme ned omkring nul, og transformere dataen ned til omkring nul.

Qb part II:

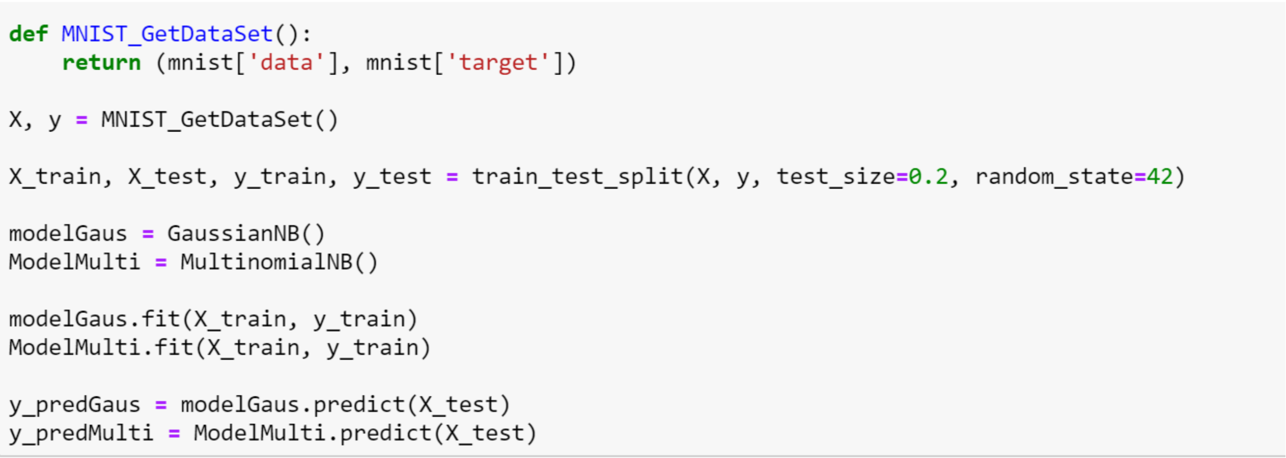


```
epoch= 498, mse_train=0.54, mse_val=1.98
epoch= 499, mse_train=0.54, mse_val=1.99
```

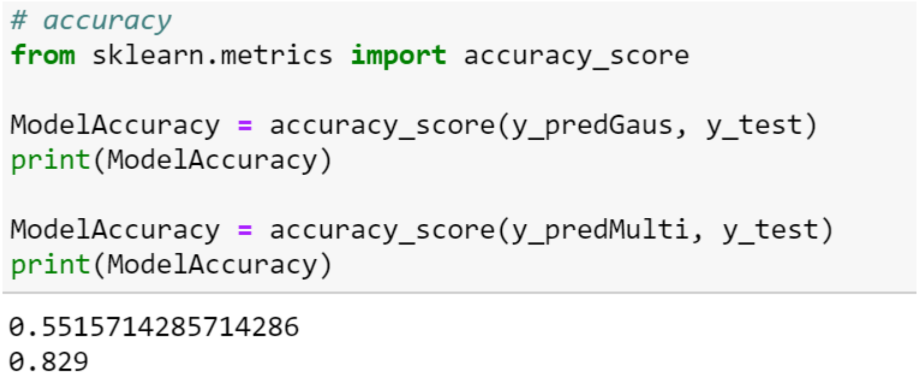


I koden overfor køres igennem fra vores data fra før, i et stochastisk graident decent. Dette gøres (ud fra vores valg) 500 gange. for hver gang, springer random til et nyt sted, og tilnærmer sig det globale minimum, denne fremgangsmåde forhindre modellen i at ramme de lokale minima, og blive "stuck" der, her tilnærmer sig den det globale minimum, i starten, og "graver sig ned der". Hvir der er "for mange epoker" riskiere vi igen overfitting, som også kan ses på billedet ovenfor. Her ses det at trænings sættets RMSE bliver mindre og mindre, og tilsyneladende bedre og bedre, men langsomt, stiger RMSE for validations sættet.

Naïve Bayes

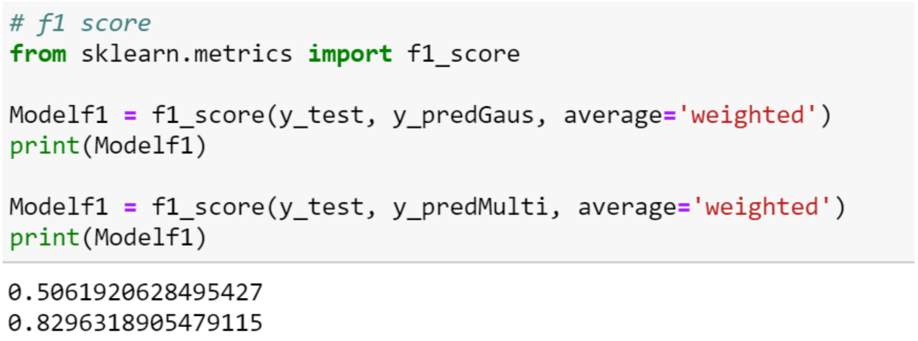


```
def MNIST_GetDataSet():
    return (mnist['data'], mnist['target'])

X, y = MNIST_GetDataSet()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

modelGaus = GaussianNB()
ModelMulti = MultinomialNB()

modelGaus.fit(X_train, y_train)
ModelMulti.fit(X_train, y_train)

y_predGaus = modelGaus.predict(X_test)
y_predMulti = ModelMulti.predict(X_test)
```



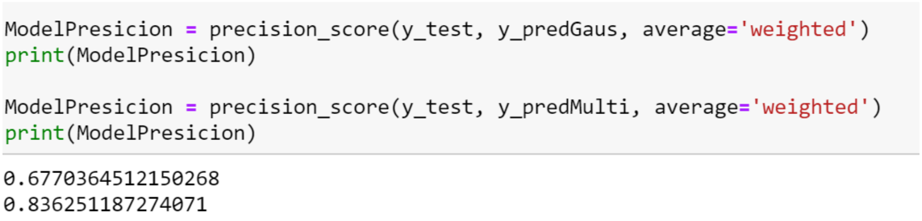
```
# accuracy
from sklearn.metrics import accuracy_score

ModelAccuracy = accuracy_score(y_predGaus, y_test)
print(ModelAccuracy)

ModelAccuracy = accuracy_score(y_predMulti, y_test)
print(ModelAccuracy)
```

```
0.5515714285714286
0.829
```



```
# f1 score
from sklearn.metrics import f1_score

Modelf1 = f1_score(y_test, y_predGaus, average='weighted')
print(Modelf1)

Modelf1 = f1_score(y_test, y_predMulti, average='weighted')
print(Modelf1)
```

```
0.5061920628495427
0.8296318905479115
```

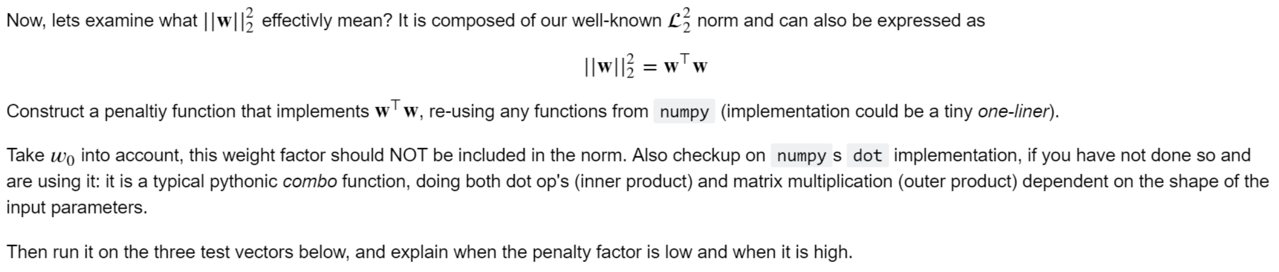


```
ModelPresicion = precision_score(y_test, y_predGaus, average='weighted')
print(ModelPresicion)

ModelPresicion = precision_score(y_test, y_predMulti, average='weighted')
print(ModelPresicion)
```

```
0.6770364512150268
0.836251187274071
```

Regulizers

Qa:



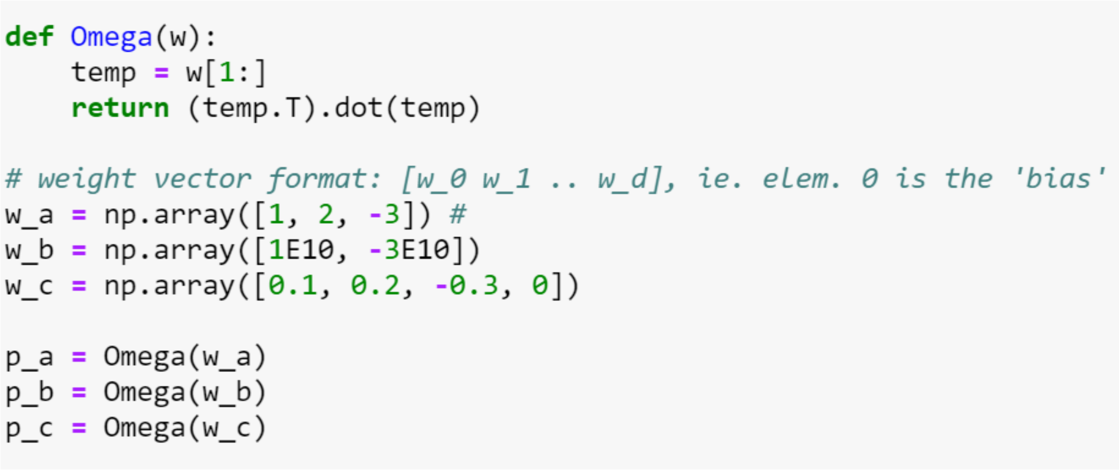
Now, lets examine what $||\mathbf{w}||_2^2$ effectivly mean? It is composed of our well-known $\mathcal{L}_2^2$ norm and can also be expressed as

$$||\mathbf{w}||_2^2 = \mathbf{w}^\top \mathbf{w}$$

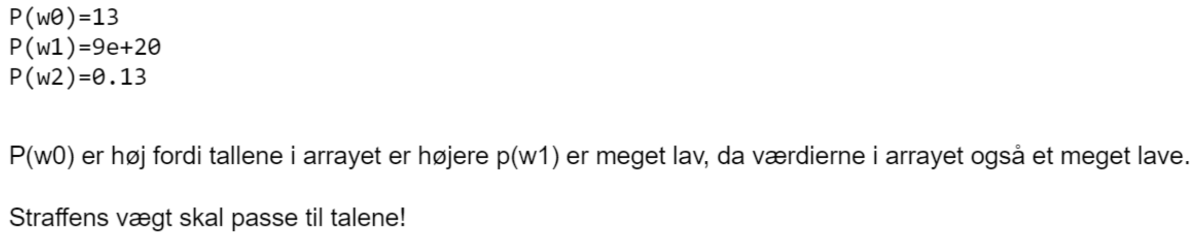
Construct a penaltiy function that implements $\mathbf{w}^\top \mathbf{w}$, re-using any functions from `numpy` (implementation could be a tiny *one-liner*).

Take w_0 into account, this weight factor should NOT be included in the norm. Also checkup on `numpy` s `dot` implementation, if you have not done so and are using it: it is a typical pythonic *combo* function, doing both dot op's (inner product) and matrix multiplication (outer product) dependent on the shape of the input parameters.

Then run it on the three test vectors below, and explain when the penalty factor is low and when it is high.



```python
def Omega(w):
    temp = w[1:]
    return (temp.T).dot(temp)

# weight vector format: [w_0 w_1 .. w_d], ie. elem. 0 is the 'bias'
w_a = np.array([1, 2, -3]) #
w_b = np.array([1E10, -3E10])
w_c = np.array([0.1, 0.2, -0.3, 0])

p_a = Omega(w_a)
p_b = Omega(w_b)
p_c = Omega(w_c)
```

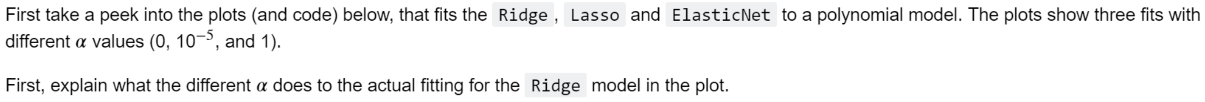


```
P(w0)=13
P(w1)=9e+20
P(w2)=0.13
```

P(w0) er høj fordi tallene i arrayet er højere p(w1) er meget lav, da værdierne i arrayet også et meget lave.

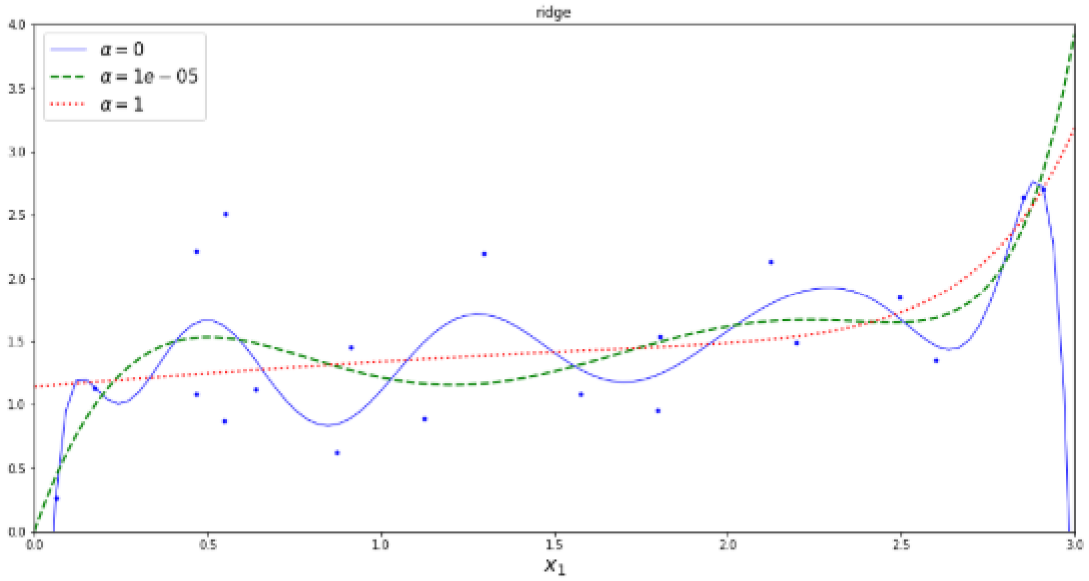
Straffens vægt skal passe til talene!

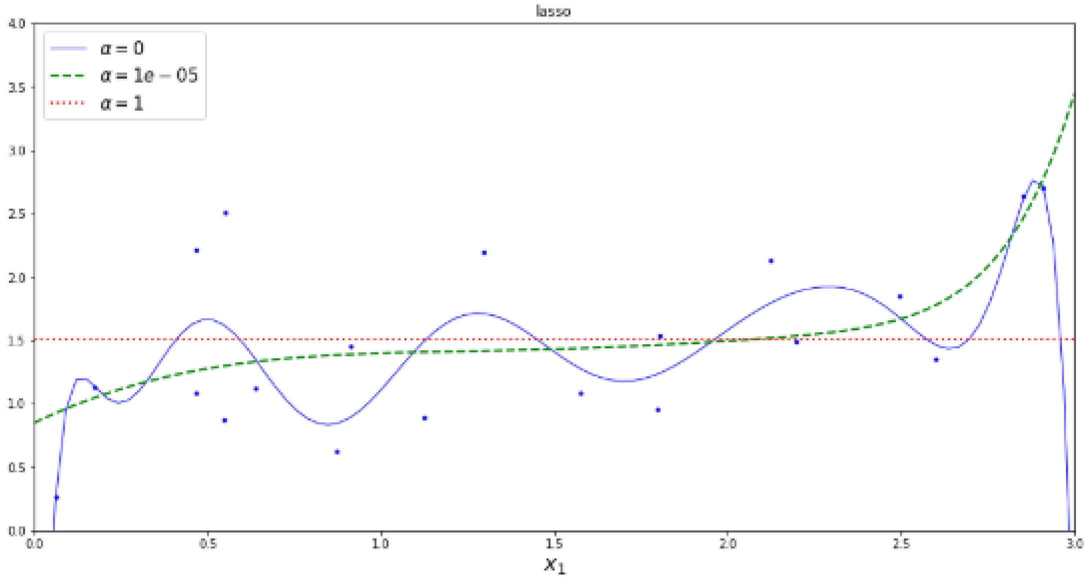
Qb:

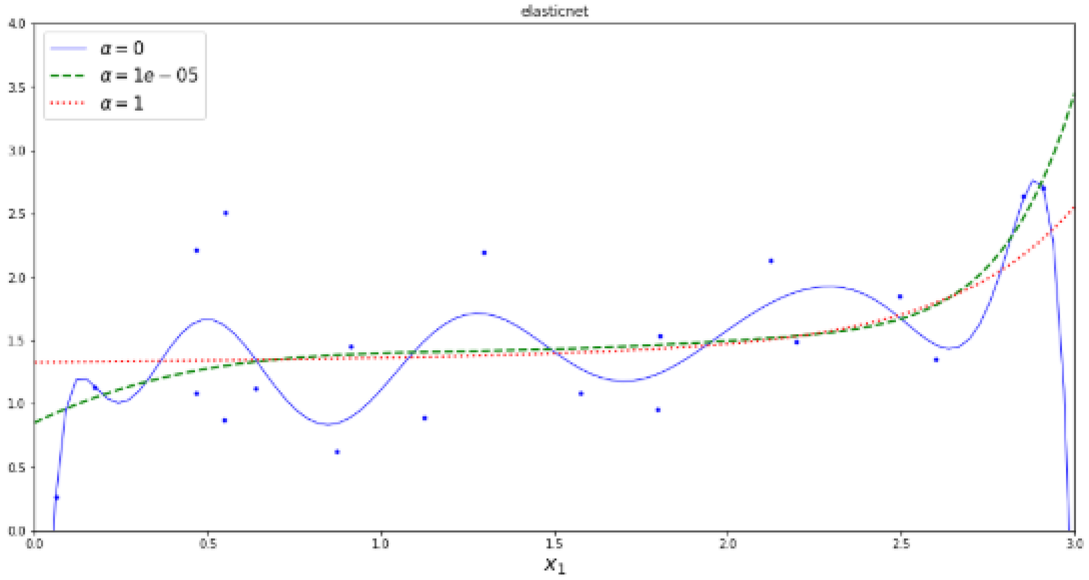


First take a peek into the plots (and code) below, that fits the `Ridge`, `Lasso` and `ElasticNet` to a polynomial model. The plots show three fits with different α values (0, 10^{-5}, and 1).

First, explain what the different α does to the actual fitting for the `Ridge` model in the plot.


ridge
$\alpha = 0$
$\alpha = 1e-05$
$\alpha = 1$
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
0.0 0.5 1.0 1.5 2.0 2.5 3.0
x_1


lasso
$\alpha = 0$
$\alpha = 1e-05$
$\alpha = 1$
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
0.0 0.5 1.0 1.5 2.0 2.5 3.0
x_1


elasticnet
$\alpha = 0$
$\alpha = 1e-05$
$\alpha = 1$
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
0.0 0.5 1.0 1.5 2.0 2.5 3.0
x_1

Ridge bruges når penalty vægten skal være høj og straffen lille (da den er summen af de kvadratiske værdier) lasso bruges når vægten skal være lav, og straffen hård! (summen af de absolute værdier)

elastic net er kombinationen af ridge og lasso.

Alpha afgører straffen for træningen. Dette kan bruges til at sortere støj fra datasættet

Qc:

Then explain the different regularization methods used for the `Ridge`, `Lasso` and `ElasticNet` models, by looking at the math formulas for the methods in the Scikit-learn documentation and/or using [HOML].

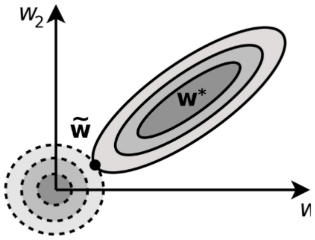
Lasso regression eliminere automatisk ubetydelige features, hvilket giver en renere model. hvor det flat liner ved vægten 1 Ridge regression, beholder vægtene så små som muligt og bevare features meres. hvis a er 0 fungere ridge som en linear regression. hvis a er for stor, flat liner det.

Elastic net er en kombination af de 2 andre.

Qd:

Finally, comment on how regularization may be used to reduce a potential tendency to overfit the data

Describe the situation with the tug-of-war between the MSE and regulizer terms in $\tilde{J}$ and the potential problem of $\mathbf{w}^*$ being far, far away from the origin (with $\alpha = 1$ in regulizer term).



Would data preprocessing in the form of scaling, standardization or normalization be of any help to that particular situation? If so, describe.

Forklaring

Regularization kan bruges hvis f.eks. dataen er overfittet, til at sortere ubetydelige features / støj fra daten!

ja, man kan preprocess for at fjerne en del af den støj der ligger i dataen, hvilket vil give bedre fitting og dermed MSE ..

Gridsearch

Qa

There are two code cells below: 1:) function setup, 2) the actual grid-search.

Review the code cells and write a **short** summary. Mainly focus on **cell 2**, but dig into cell 1 if you find it interesting (notice the use of local-function, a nifty feature in python).

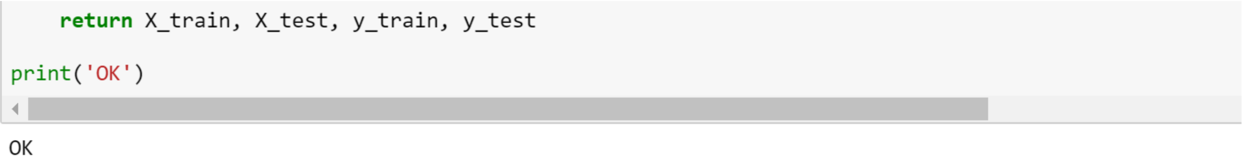
In detail, examine the lines:

```
grid_tuned = GridSearchCV(model, tuning_parameters, ..
grid_tuned.fit(X_train, y_train)
..
FullReport(grid_tuned , X_test, y_test, time_gridsearch)
```

and write a short description of how the `GridSeachCV` works: explain how the search parameter set is created and the overall search mechanism (without going into to much detail)

What role does the parameter `scoring='f1_micro'` play in the `GridSearchCV`, and what does `n_jobs=-1` mean?

Cell 1

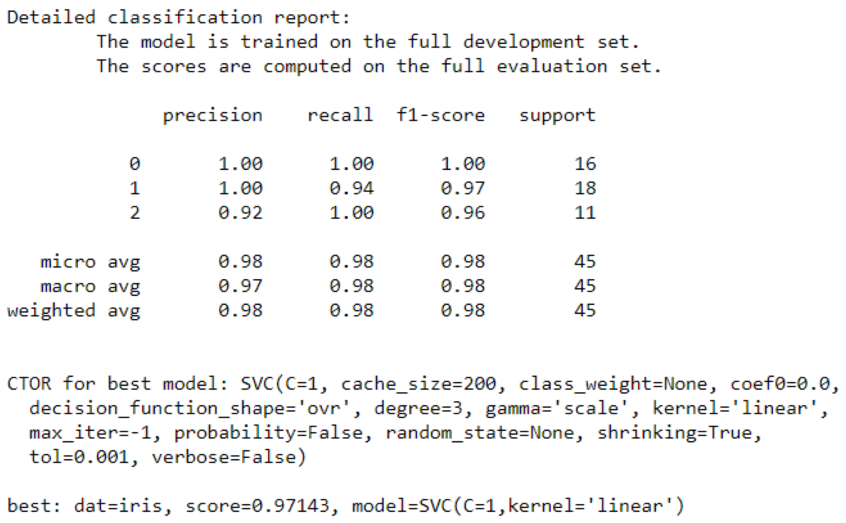


```
    return X_train, X_test, y_train, y_test

print('OK')
```

```
OK
```

Cell 2

```
DATA: iris..
  org. data:  X.shape      =(  150;    4), y.shape      =(  150)
  train data: X_train.shape=(  105;    4), y_train.shape=(  105)
  test data:  X_test.shape =(   45;    4), y_test.shape =(   45)

SEARCH TIME: 0.05 sec

Best model set found on train set:

        best parameters={'C': 1, 'kernel': 'linear'}
        best 'f1_micro' score=0.9714285714285714
        best index=0

Best estimator CTOR:
        SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

Grid scores ('f1_micro') on development set:
        [ 0]: 0.971 (+/-0.048) for {'C': 1, 'kernel': 'linear'}
        [ 1]: 0.952 (+/-0.084) for {'C': 1, 'kernel': 'rbf'}
        [ 2]: 0.952 (+/-0.084) for {'C': 10, 'kernel': 'linear'}
        [ 3]: 0.971 (+/-0.048) for {'C': 10, 'kernel': 'rbf'}

Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      0.94      0.97        18
           2       0.92      1.00      0.96        11

   micro avg       0.98      0.98      0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```



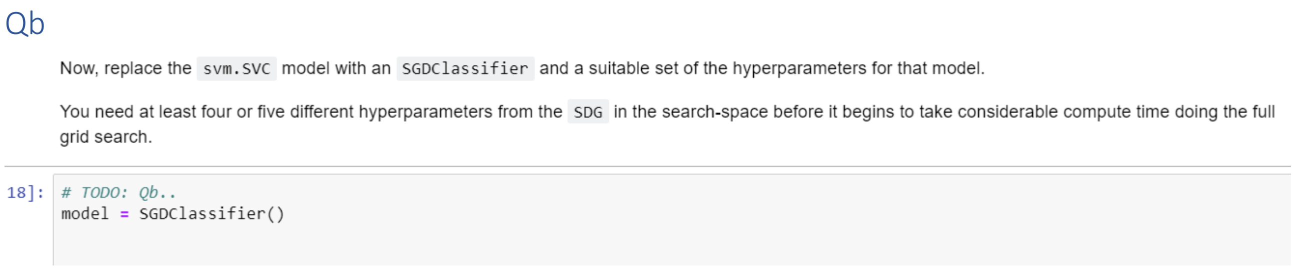
```
Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      0.94      0.97        18
           2       0.92      1.00      0.96        11

   micro avg       0.98      0.98      0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45


CTOR for best model: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

best: dat=iris, score=0.97143, model=SVC(C=1,kernel='linear')
```

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.

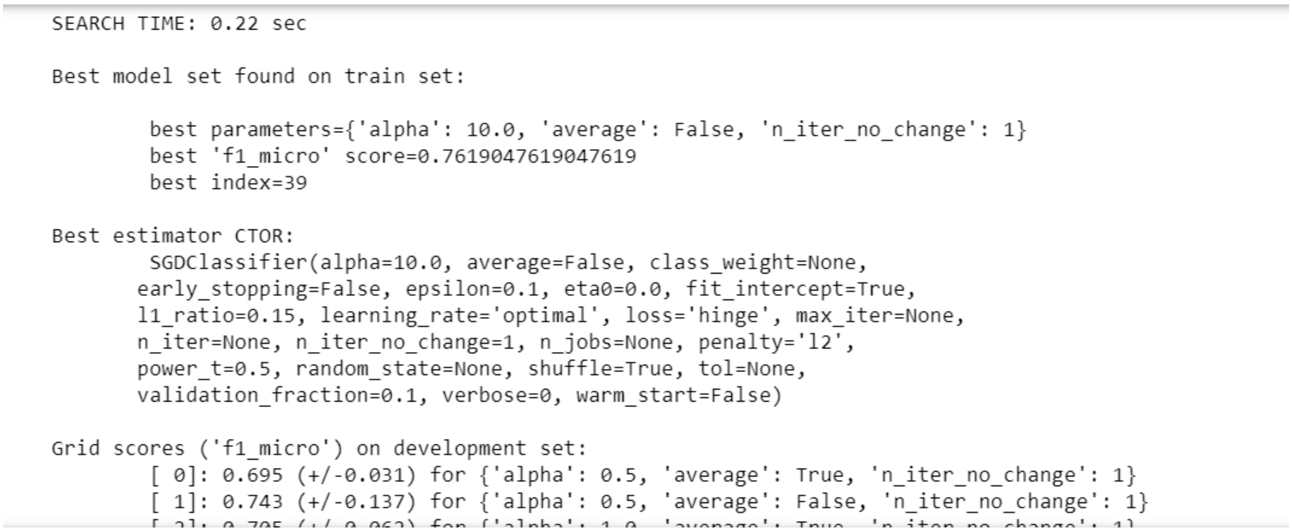


Qb

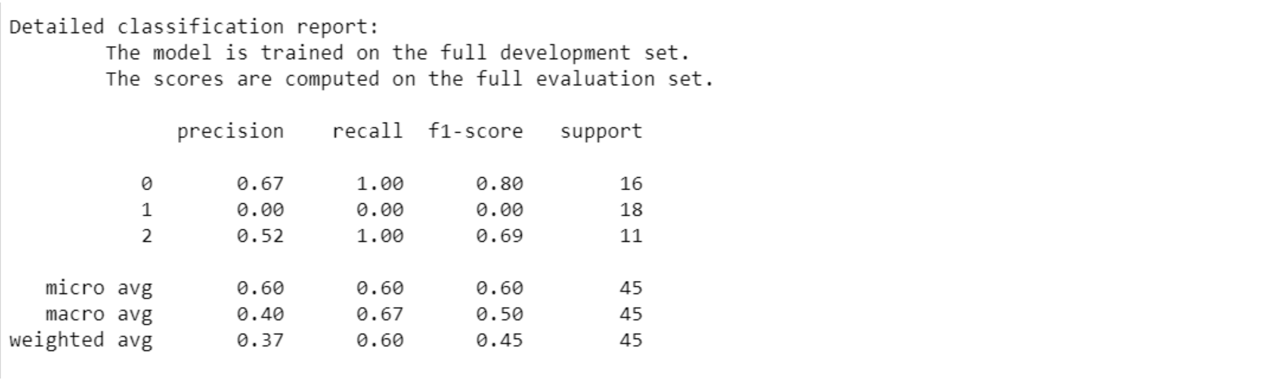
Now, replace the `svm.SVC` model with an `SGDClassifier` and a suitable set of the hyperparameters for that model.

You need at least four or five different hyperparameters from the `SDG` in the search-space before it begins to take considerable compute time doing the full grid search.

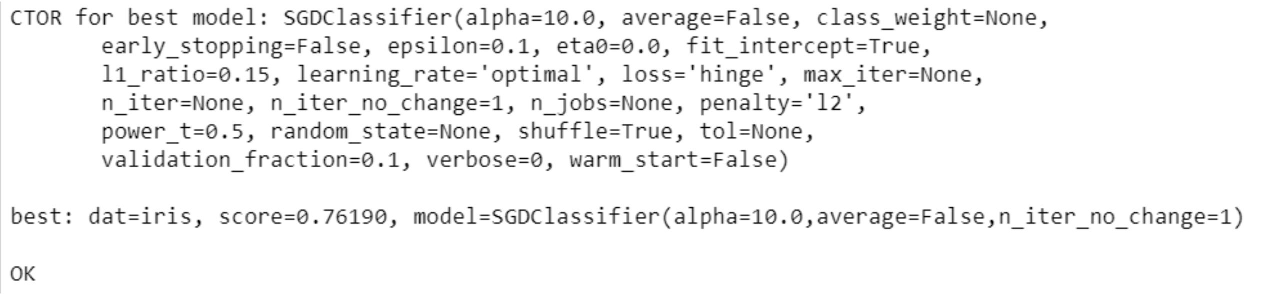
18]:
```
# TODO: Qb..
model = SGDClassifier()
```



```
SEARCH TIME: 0.22 sec

Best model set found on train set:

        best parameters={'alpha': 10.0, 'average': False, 'n_iter_no_change': 1}
        best 'f1_micro' score=0.7619047619047619
        best index=39

Best estimator CTOR:
        SGDClassifier(alpha=10.0, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=1, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)

Grid scores ('f1_micro') on development set:
        [ 0]: 0.695 (+/-0.031) for {'alpha': 0.5, 'average': True, 'n_iter_no_change': 1}
        [ 1]: 0.743 (+/-0.137) for {'alpha': 0.5, 'average': False, 'n_iter_no_change': 1}
        [ 2]: 0.705 (+/-0.062) for {'alpha': 1.0, 'average': True, 'n_iter_no_change': 1}
```

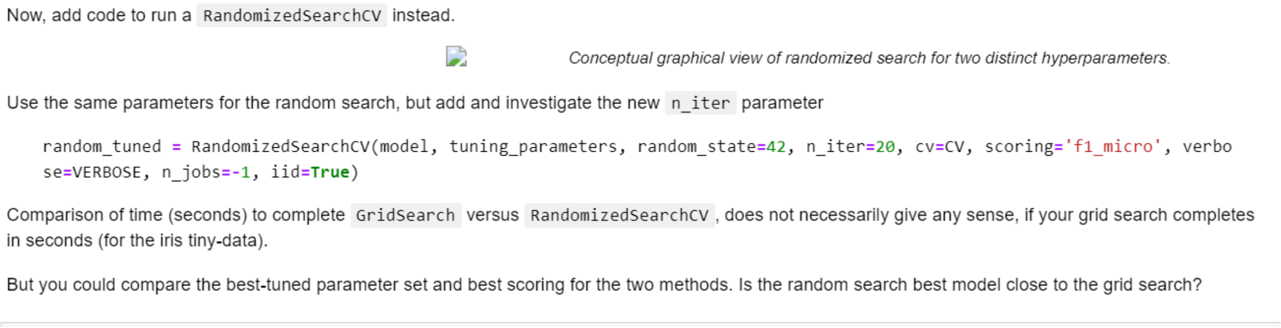


```
Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       0.67      1.00      0.80        16
           1       0.00      0.00      0.00        18
           2       0.52      1.00      0.69        11

   micro avg       0.60      0.60      0.60        45
   macro avg       0.40      0.67      0.50        45
weighted avg       0.37      0.60      0.45        45
```



```
CTOR for best model: SGDClassifier(alpha=10.0, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=1, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)

best: dat=iris, score=0.76190, model=SGDClassifier(alpha=10.0,average=False,n_iter_no_change=1)

OK
```

Qc



Now, add code to run a `RandomizedSearchCV` instead.

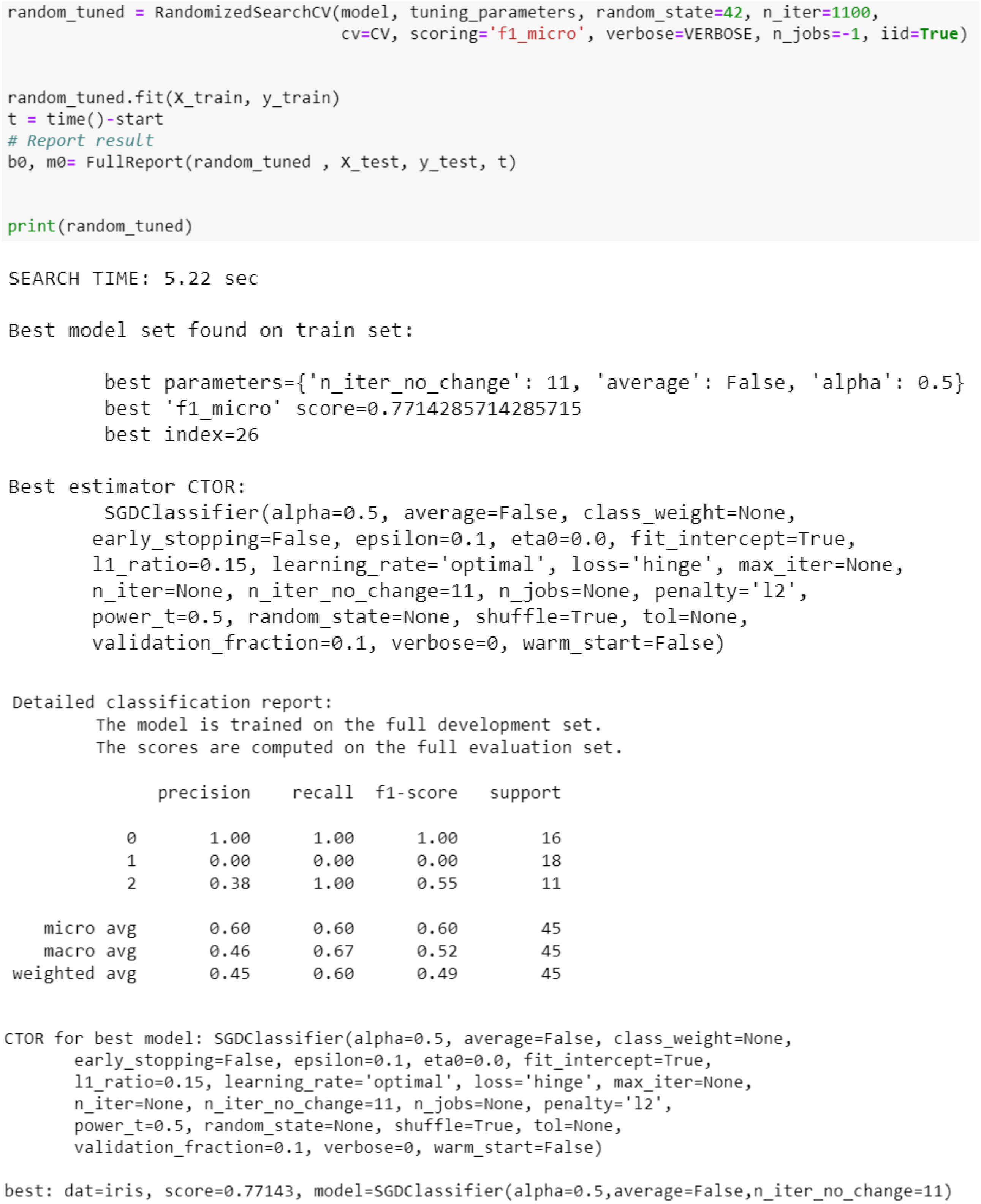
Conceptual graphical view of randomized search for two distinct hyperparameters.

Use the same parameters for the random search, but add and investigate the new `n_iter` parameter

```
random_tuned = RandomizedSearchCV(model, tuning_parameters, random_state=42, n_iter=20, cv=CV, scoring='f1_micro', verbo
se=VERBOSE, n_jobs=-1, iid=True)
```

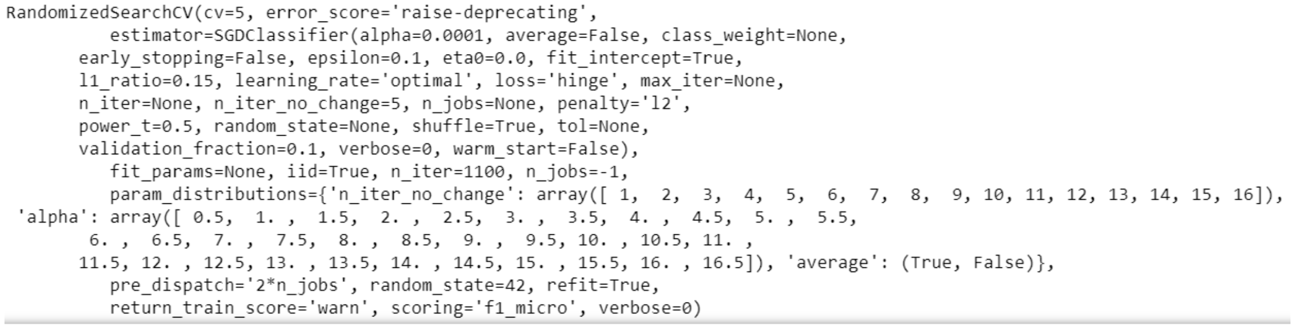
Comparison of time (seconds) to complete `GridSearch` versus `RandomizedSearchCV`, does not necessarily give any sense, if your grid search completes in seconds (for the iris tiny-data).

But you could compare the best-tuned parameter set and best scoring for the two methods. Is the random search best model close to the grid search?

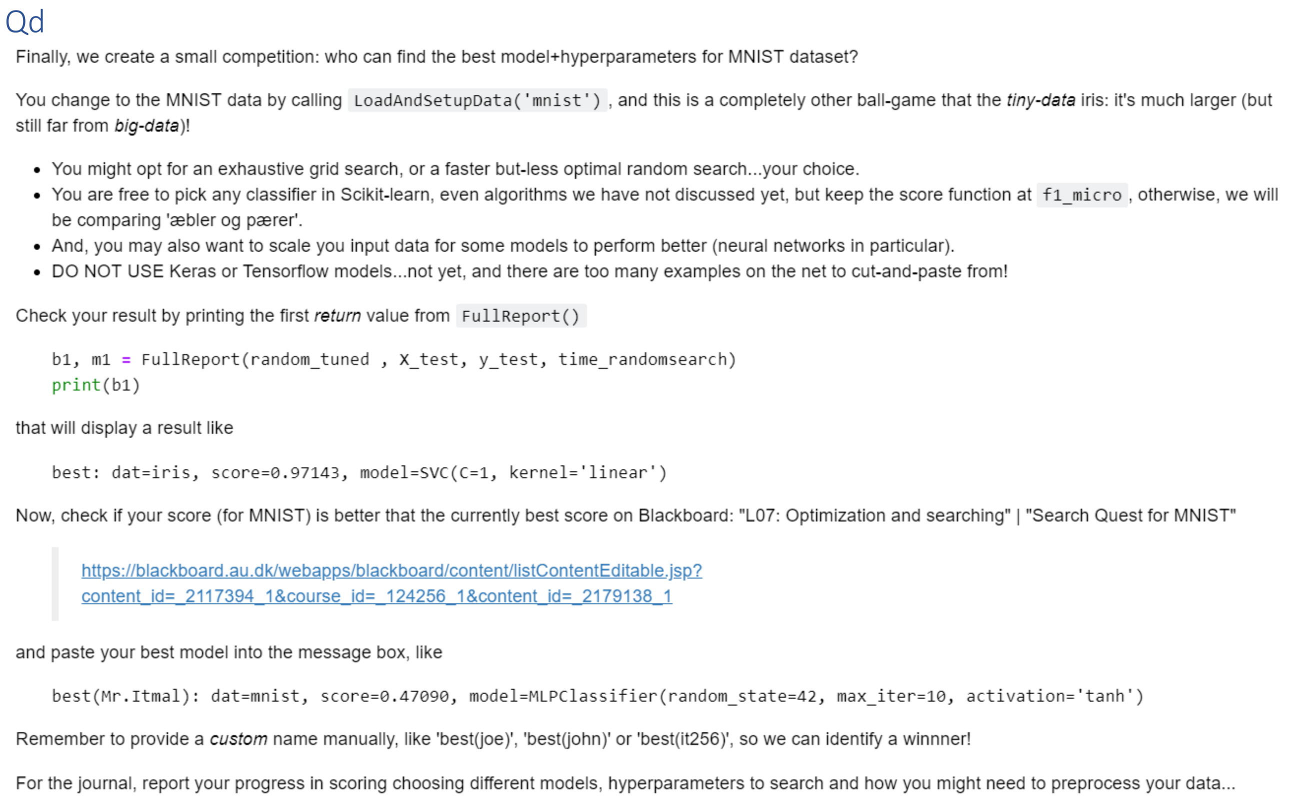


```
random_tuned = RandomizedSearchCV(model, tuning_parameters, random_state=42, n_iter=1100,
                                  cv=CV, scoring='f1_micro', verbose=VERBOSE, n_jobs=-1, iid=True)


random_tuned.fit(X_train, y_train)
t = time()-start
# Report result
b0, m0= FullReport(random_tuned , X_test, y_test, t)


print(random_tuned)
```

```
SEARCH TIME: 5.22 sec

Best model set found on train set:

        best parameters={'n_iter_no_change': 11, 'average': False, 'alpha': 0.5}
        best 'f1_micro' score=0.7714285714285715
        best index=26

Best estimator CTOR:
        SGDClassifier(alpha=0.5, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=11, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)


Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       0.00      0.00      0.00        18
           2       0.38      1.00      0.55        11

   micro avg       0.60      0.60      0.60        45
   macro avg       0.46      0.67      0.52        45
weighted avg       0.45      0.60      0.49        45


CTOR for best model: SGDClassifier(alpha=0.5, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=11, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)

best: dat=iris, score=0.77143, model=SGDClassifier(alpha=0.5,average=False,n_iter_no_change=11)
```



```
RandomizedSearchCV(cv=5, error_score='raise-deprecating',
          estimator=SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=5, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False),
          fit_params=None, iid=True, n_iter=1100, n_jobs=-1,
          param_distributions={'n_iter_no_change': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16]),
 'alpha': array([ 0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,
        6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. , 10.5, 11. ,
       11.5, 12. , 12.5, 13. , 13.5, 14. , 14.5, 15. , 15.5, 16. , 16.5]), 'average': (True, False)},
          pre_dispatch='2*n_jobs', random_state=42, refit=True,
          return_train_score='warn', scoring='f1_micro', verbose=0)
```



Qd

Finally, we create a small competition: who can find the best model+hyperparameters for MNIST dataset?

You change to the MNIST data by calling `LoadAndSetupData('mnist')`, and this is a completely other ball-game that the *tiny-data* iris: it's much larger (but still far from *big-data*)!

- You might opt for an exhaustive grid search, or a faster but-less optimal random search...your choice.
- You are free to pick any classifier in Scikit-learn, even algorithms we have not discussed yet, but keep the score function at `f1_micro`, otherwise, we will be comparing 'æbler og pærer'.
- And, you may also want to scale you input data for some models to perform better (neural networks in particular).
- DO NOT USE Keras or Tensorflow models...not yet, and there are too many examples on the net to cut-and-paste from!

Check your result by printing the first *return* value from `FullReport()`

```
b1, m1 = FullReport(random_tuned , X_test, y_test, time_randomsearch)
print(b1)
```

that will display a result like

```
best: dat=iris, score=0.97143, model=SVC(C=1, kernel='linear')
```

Now, check if your score (for MNIST) is better that the currently best score on Blackboard: "L07: Optimization and searching" | "Search Quest for MNIST"

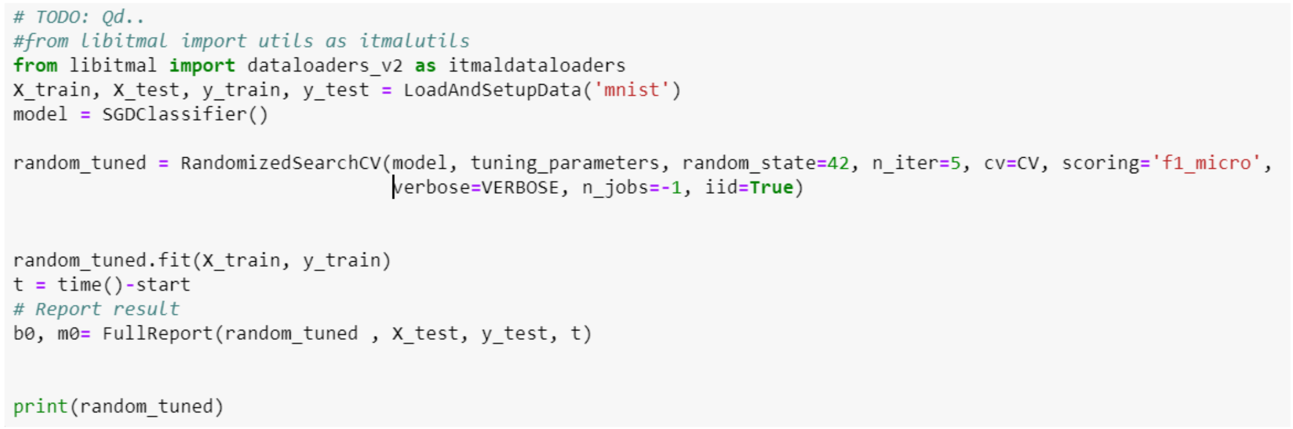
> https://blackboard.au.dk/webapps/blackboard/content/listContentEditable.jsp?content_id=_2117394_1&course_id=_124256_1&content_id=_2179138_1

and paste your best model into the message box, like

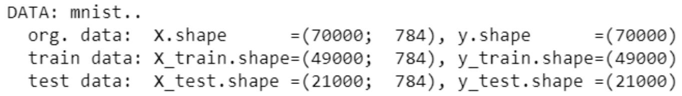
```
best(Mr.Itmal): dat=mnist, score=0.47090, model=MLPClassifier(random_state=42, max_iter=10, activation='tanh')
```

Remember to provide a *custom* name manually, like 'best(joe)', 'best(john)' or 'best(it256)', so we can identify a winnner!

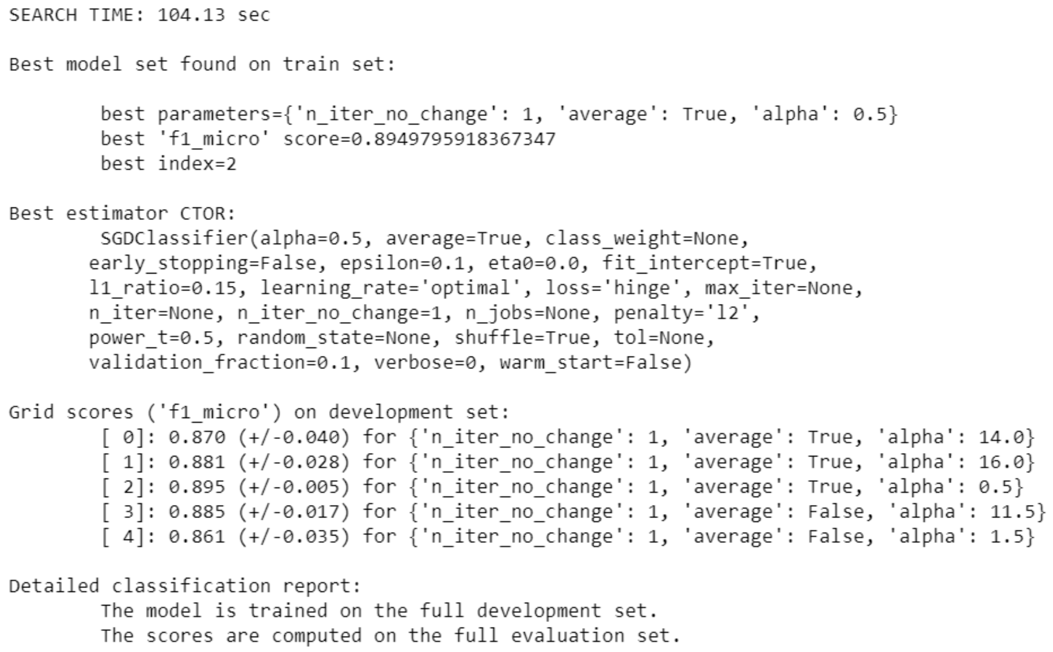
For the journal, report your progress in scoring choosing different models, hyperparameters to search and how you might need to preprocess your data...



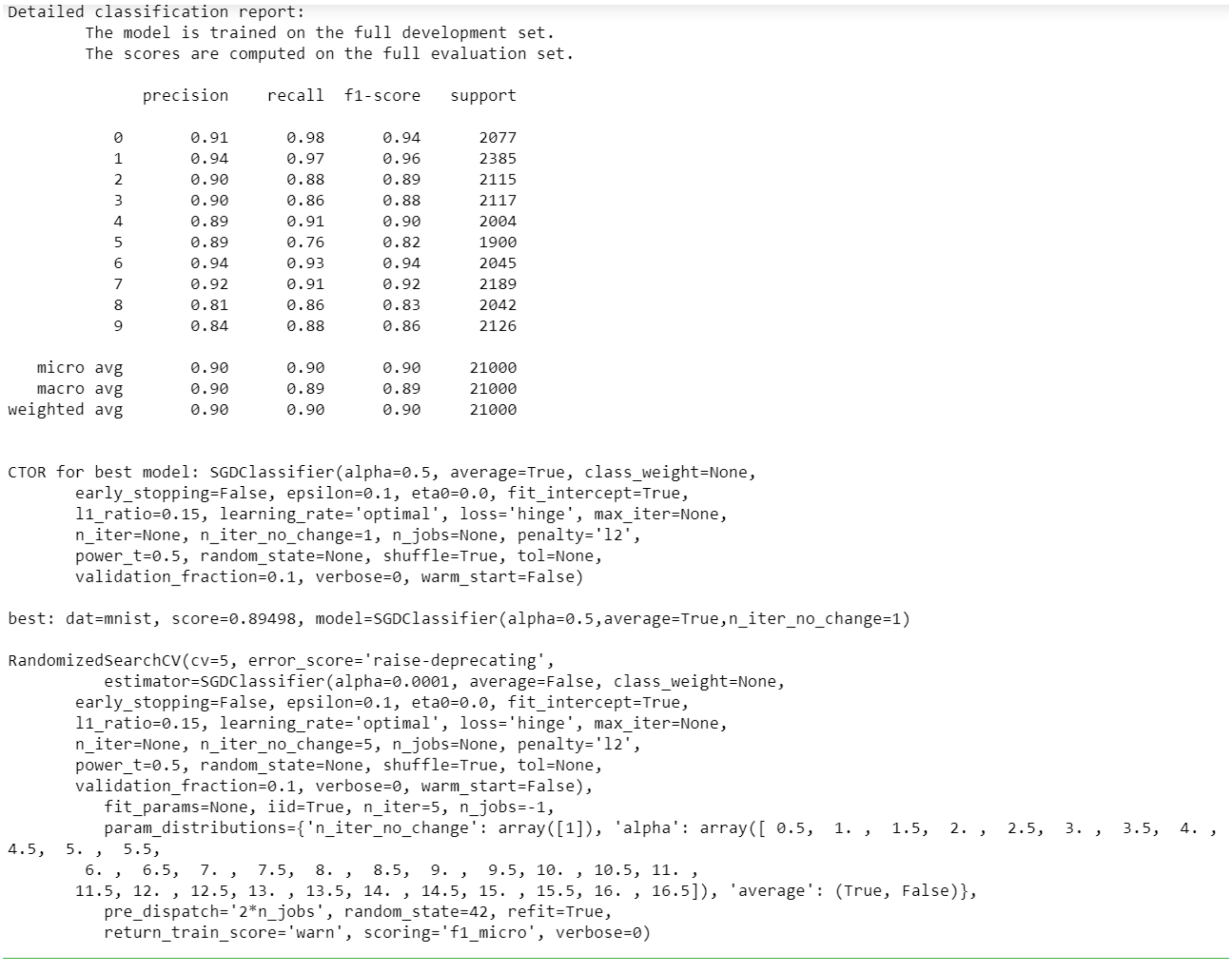
```
# TODO: Qd..
#from libitmal import utils as itmalutils
from libitmal import dataloaders_v2 as itmaldataloaders
X_train, X_test, y_train, y_test = LoadAndSetupData('mnist')
model = SGDClassifier()

random_tuned = RandomizedSearchCV(model, tuning_parameters, random_state=42, n_iter=5, cv=CV, scoring='f1_micro',
                                  verbose=VERBOSE, n_jobs=-1, iid=True)


random_tuned.fit(X_train, y_train)
t = time()-start
# Report result
b0, m0= FullReport(random_tuned , X_test, y_test, t)


print(random_tuned)
```



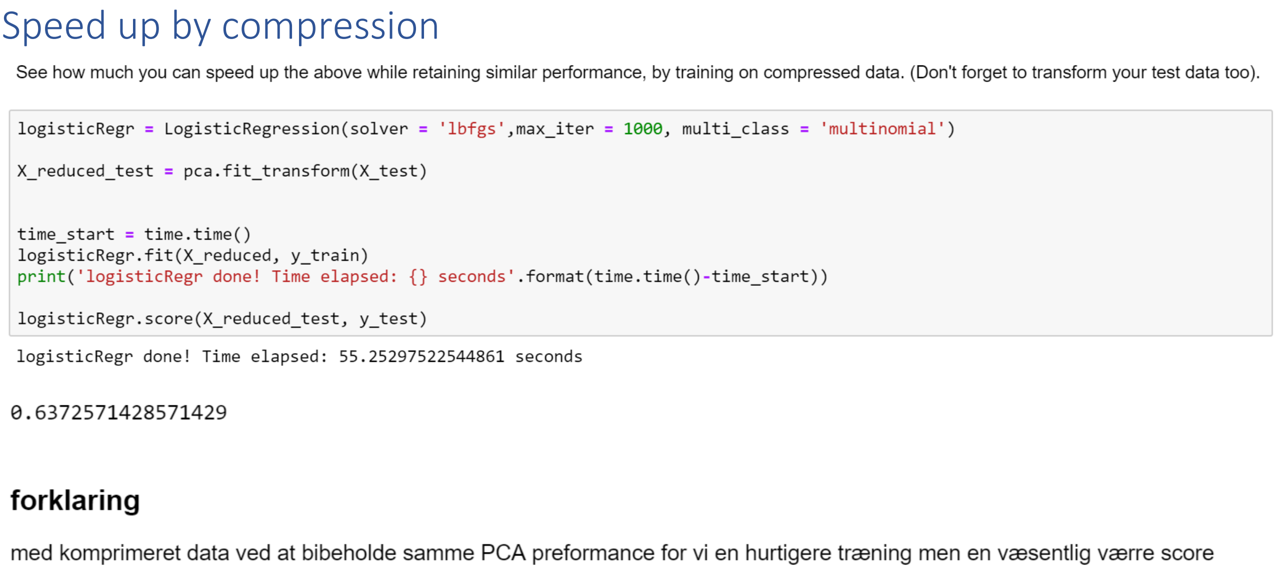
```
DATA: mnist..
  org. data:  X.shape      =(70000;  784), y.shape      =(70000)
  train data: X_train.shape=(49000;  784), y_train.shape=(49000)
  test data:  X_test.shape =(21000;  784), y_test.shape =(21000)
```



```
SEARCH TIME: 104.13 sec

Best model set found on train set:

        best parameters={'n_iter_no_change': 1, 'average': True, 'alpha': 0.5}
        best 'f1_micro' score=0.8949795918367347
        best index=2

Best estimator CTOR:
        SGDClassifier(alpha=0.5, average=True, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=1, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)

Grid scores ('f1_micro') on development set:
        [ 0]: 0.870 (+/-0.040) for {'n_iter_no_change': 1, 'average': True, 'alpha': 14.0}
        [ 1]: 0.881 (+/-0.028) for {'n_iter_no_change': 1, 'average': True, 'alpha': 16.0}
        [ 2]: 0.895 (+/-0.005) for {'n_iter_no_change': 1, 'average': True, 'alpha': 0.5}
        [ 3]: 0.885 (+/-0.017) for {'n_iter_no_change': 1, 'average': False, 'alpha': 11.5}
        [ 4]: 0.861 (+/-0.035) for {'n_iter_no_change': 1, 'average': False, 'alpha': 1.5}

Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.
```



```
Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       0.91      0.98      0.94      2077
           1       0.94      0.97      0.96      2385
           2       0.90      0.88      0.89      2115
           3       0.90      0.86      0.88      2117
           4       0.89      0.91      0.90      2004
           5       0.89      0.76      0.82      1900
           6       0.94      0.93      0.94      2045
           7       0.92      0.91      0.92      2189
           8       0.81      0.86      0.83      2042
           9       0.84      0.88      0.86      2126

   micro avg       0.90      0.90      0.90     21000
   macro avg       0.90      0.89      0.89     21000
weighted avg       0.90      0.90      0.90     21000


CTOR for best model: SGDClassifier(alpha=0.5, average=True, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=1, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False)

best: dat=mnist, score=0.89498, model=SGDClassifier(alpha=0.5,average=True,n_iter_no_change=1)

RandomizedSearchCV(cv=5, error_score='raise-deprecating',
          estimator=SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=None,
       n_iter=None, n_iter_no_change=5, n_jobs=None, penalty='l2',
       power_t=0.5, random_state=None, shuffle=True, tol=None,
       validation_fraction=0.1, verbose=0, warm_start=False),
          fit_params=None, iid=True, n_iter=5, n_jobs=-1,
          param_distributions={'n_iter_no_change': array([1]), 'alpha': array([ 0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,
4.5,  5. ,  5.5,
        6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. , 10.5, 11. ,
       11.5, 12. , 12.5, 13. , 13.5, 14. , 14.5, 15. , 15.5, 16. , 16.5]), 'average': (True, False)},
          pre_dispatch='2*n_jobs', random_state=42, refit=True,
          return_train_score='warn', scoring='f1_micro', verbose=0)
```



Speed up by compression

See how much you can speed up the above while retaining similar performance, by training on compressed data. (Don't forget to transform your test data too).

```
logisticRegr = LogisticRegression(solver = 'lbfgs',max_iter = 1000, multi_class = 'multinomial')

X_reduced_test = pca.fit_transform(X_test)


time_start = time.time()
logisticRegr.fit(X_reduced, y_train)
print('logisticRegr done! Time elapsed: {} seconds'.format(time.time()-time_start))

logisticRegr.score(X_reduced_test, y_test)
```

```
logisticRegr done! Time elapsed: 55.25297522544861 seconds

0.6372571428571429
```

forklaring

med komprimeret data ved at bibeholde samme PCA preformance for vi en hurtigere træning men en væsentlig værre score

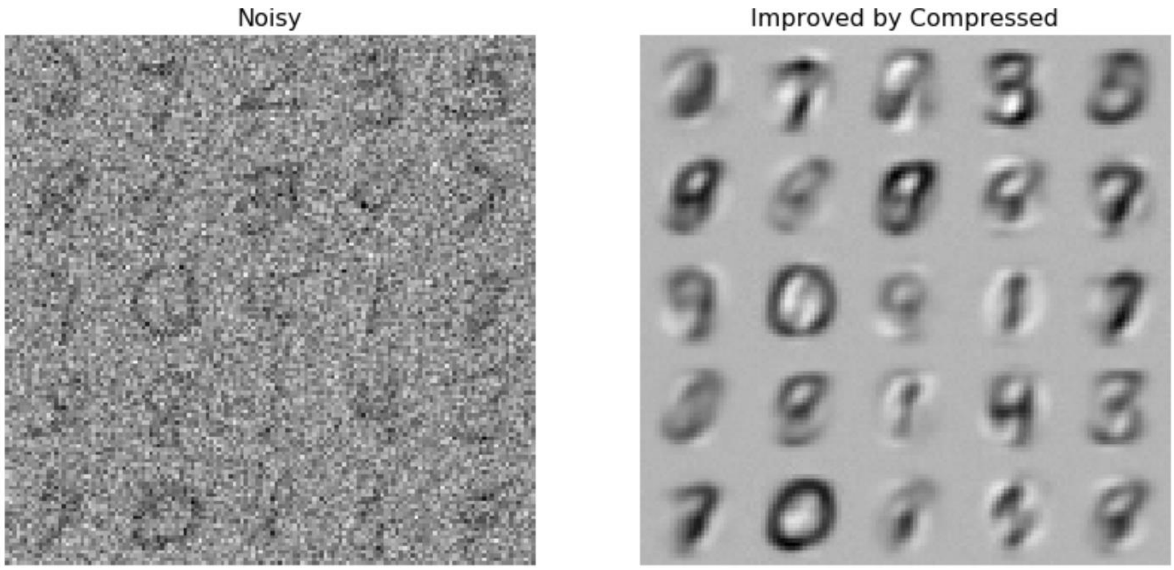
Noise reduction

Add some noise to MNIST (done below already), then compare a classifier trained and tested on:

Case 1: The noisy data.

Case 2: A PCA reduced version of the noisy data (remember to transform your test data too).

Can you find a reduced number of components that boosts performance? Different classifiers may handle noise more or less well by themselves. I tested with the SGDClassifier.



forklaring

Ved at sætte PCA n_components til at have 5% variance for vi et tydeligere billede af dataen.

```
clf = linear_model.SGDClassifier(max_iter=1000, tol=1e-3)
X_fit = clf.fit(X_train_noisy, y_train)
clf.score(X_test_noisy, y_test)
```

```
0.6383428571428571
```

forklaring

SGD kan bruges til at fitte dataen, men for en score på 0.63, som er væsentlig værre end vores PCA på ca 0.9.

PCA -> t-SNE features

From the above visualizations it seems reasonable that "PCA -> t-SNE" will perform much better than PCA alone as feature extraction by compression as input to a classifier (and be faster than t-SNE alone too).

Try if you can train and test with "PCA -> t-SNE" components (like we did earlier with PCA components). If not, explain why.

forklaring

man kan ikke traine og teste da t_SNE kun har funktionaliteten til fit_transform og ikke en transform til at bruge den fittede model og kan derfor ikke bruges til at transforme generelt data, men kun på dens egen fit_transformede data.

Neurons

Qa:

In broad terms, how does a neuron communicate?

Then explain, again with the wording of an engineer, not a biologist, how the neurons are structured in the brain.

Forklaring

Et neuron kommunikerer ved at "tilfældige" energi udladninger sender besked til neuronen om at videresende eller produceret et stof (en besked), som den sender videre til enten andre neuroner eller ud i kroppen, dette sker i synapsen. synapsen er mellemrummet mellem to neuroner. et neuron transmitter et stof over synapsen til en receptor i en anden neuron. dette er den besked, som der kommunikeres med mellem neuroner.

en meget kort forklaring om neuroner set fra en ingeniør, ville nok være at se neuroner som microchips, hvor synapser er de gates, som forbinder selv samme chips, så der kommer en energiladning med en besked fra en microchip som genereres "tilfældigt" og alt efter sammen faldet mellem energi udledning og hvilket gate, den rammer bestemmer beskeden. dette er mere struktureret end "tilfældigt", men dette kræver en bedre forståelse af biologi til at kunne beskrive i store detajle.

Qb:

Now, explaining cognition is still a pretty impossible philosophical question to answer. So you should answer this question: **what is your understanding of cognition?**

If we reproduce biological neurons, and just make enough of them, in your opinion, will a machine ever be able to have *cognition*?

Forklaring

kognition er evnen til at forstå, det er i sig selv et filosofisk beskrivelse. men for at gøre det så simplet som muligt, så er forståelse mere end bare at kunne genkende, det er lige såvel at kunne associaserer det man ser og oplever med tilsvarende oplevelse at kunne skabe forbindelser imellem umiddelbare usammenhængende ting, igennem det man ville kunne intuition. Der er grader at den slags. dyr har en kognativ forståelse for visse opgaver og deres association. (godbider) dette er dog en slags intuition. hvor mennesker intuition er væsentlig anderledes. da der findes sociale og dyriske associationer samt intellektuel intuition (logisk). Disse intuitioner kan ikke genskabes ved at skabe flere neuroner, da desto store hjernen er, er ikke lig med ens kognative evner. man kan sige at ML tillader maskiner at have dyriske associationer, i form af supervised/ semi-supervised learning / back-propergation. ved at bekræfte overfor maskinen at den laver den rigtige association. Som vil kunne skabe en form for intuition til at løse specifikke opgaver. men maskinen lærer ikke at associaere på et højere plan. Som ikke løses ved mere hjernekapicitet. muligvis ved en anden forståelse af læring af maskinen eller hukommenelse og evnen til at skabe "tilfældige" assiciationer.

Perceptron

Qa:

Load the moon data, split it into a train-and-test set, and train a `sklearn.linear_model.Perceptron` with default parameters (and with no cross-val this time).

What is the default score metric for the perceptron and what score do reach on the test data?

```
X, y = itmaldataloaders.MOON_GetDataSet(n_samples=1000)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

model = Perceptron();
model.fit(X_train, y_train);
model.score(X_test,y_test)
```

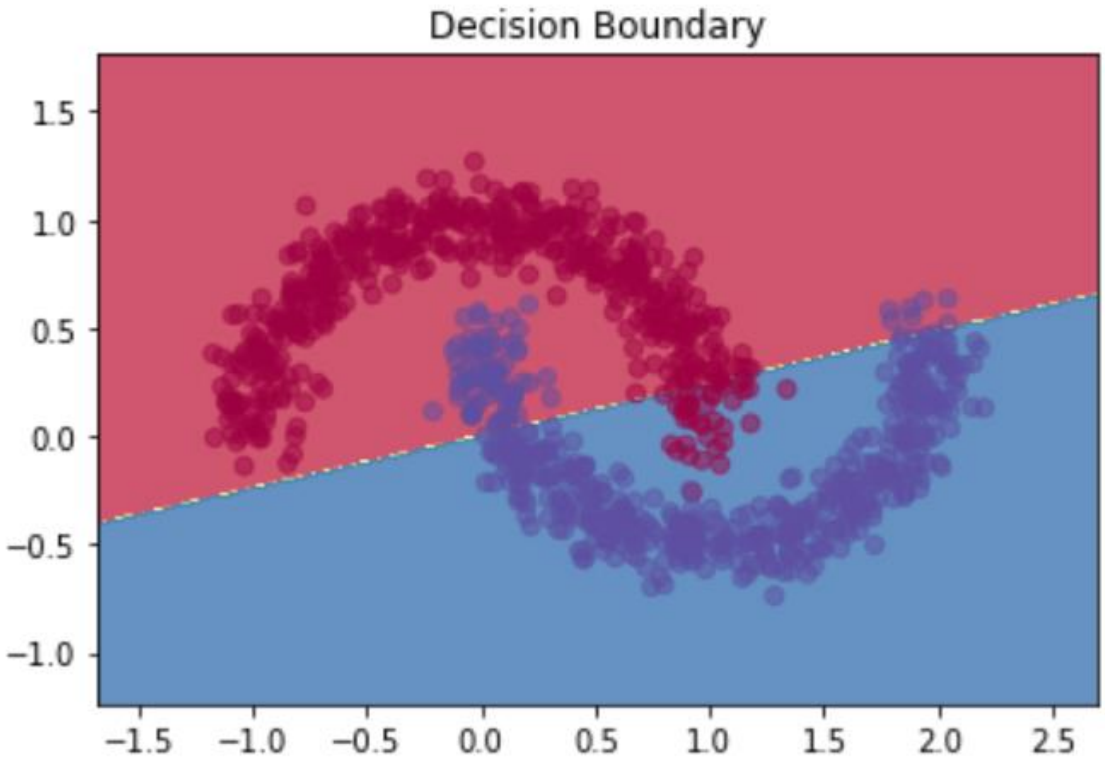
```
0.8533333333333334
```

Forklaring

Som det kan ses får vi en score på 0,85 med default settings som er max_iter=1000 og tol=None.

Qb:

Use the helper code below to plot the decision boundary for the perceptron on the moon-data.



Qc:

How does the moon data relate to the well-known *XOR*-problem in machine learning?

Is it possible for a single neuron or perceptron to solve the moon/XOR problem satisfactorily?

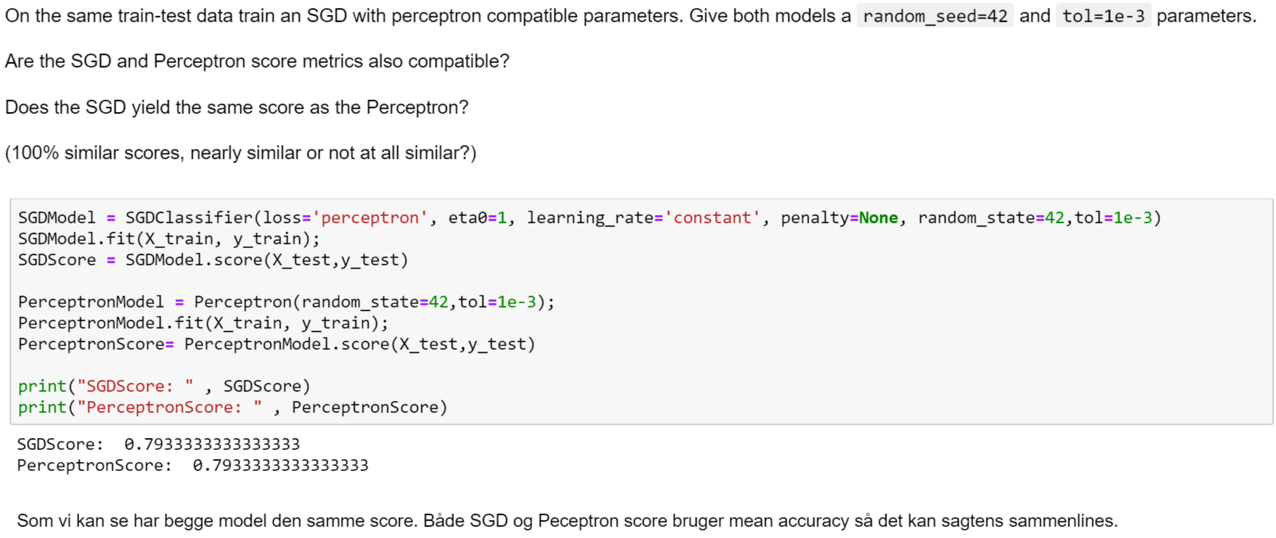
Elaborate on the reason why not, and give a comment on how to overcome the XOR-problematic using more perceptrons.

Forklaring

En XOR tager to input og samlinger dem. Hvis de to input er forskellige giver det en false og hvis de er det samme giver den true.

En enkel Neropn/perceptron kan ikke løse dette problem. At den ikke kunne løse dette var også grunden til at man på et tidspunkt ikke mente at man kunne bruge neroner til noget advanceret ML. Problemet er at neronerne virker via en liniær model. Den linær model kan ikke komme med to et enten eller output

Qd:



On the same train-test data train an SGD with perceptron compatible parameters. Give both models a `random_seed=42` and `tol=1e-3` parameters.

Are the SGD and Perceptron score metrics also compatible?

Does the SGD yield the same score as the Perceptron?

(100% similar scores, nearly similar or not at all similar?)

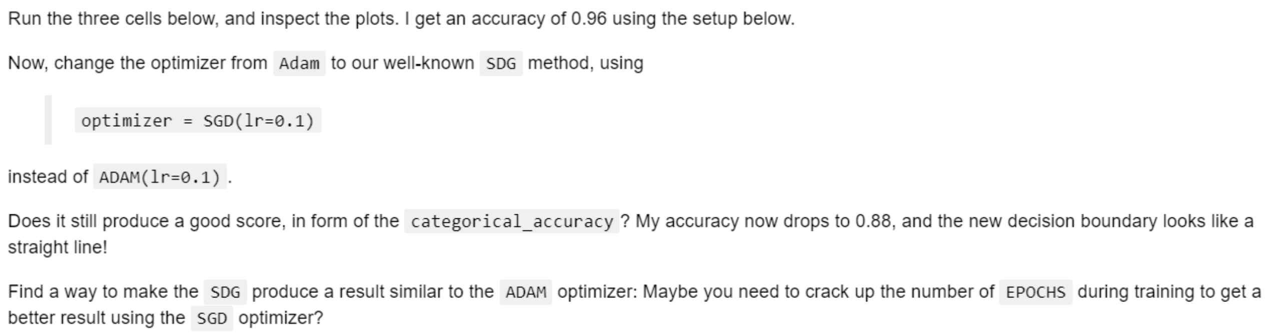
```
SGDModel = SGDClassifier(loss='perceptron', eta0=1, learning_rate='constant', penalty=None, random_state=42,tol=1e-3)
SGDModel.fit(X_train, y_train);
SGDScore = SGDModel.score(X_test,y_test)

PerceptronModel = Perceptron(random_state=42,tol=1e-3);
PerceptronModel.fit(X_train, y_train);
PerceptronScore= PerceptronModel.score(X_test,y_test)

print("SGDScore: " , SGDScore)
print("PerceptronScore: " , PerceptronScore)
```

```
SGDScore:  0.7933333333333333
PerceptronScore:  0.7933333333333333
```

Som vi kan se har begge model den samme score. Både SGD og Peceptron score bruger mean accuracy så det kan sagtens sammenlines.

Multi-layers Perceptrons (MLP)

Qa:



Run the three cells below, and inspect the plots. I get an accuracy of 0.96 using the setup below.

Now, change the optimizer from `Adam` to our well-known `SDG` method, using

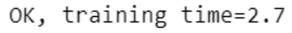
> `optimizer = SGD(lr=0.1)`

instead of `ADAM(lr=0.1)` .

Does it still produce a good score, in form of the `categorical_accuracy` ? My accuracy now drops to 0.88, and the new decision boundary looks like a straight line!

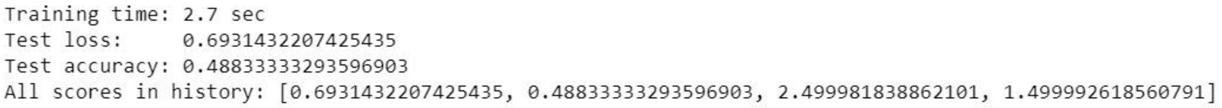
Find a way to make the `SDG` produce a result similar to the `ADAM` optimizer: Maybe you need to crack up the number of `EPOCHS` during training to get a better result using the `SGD` optimizer?

Cell 1

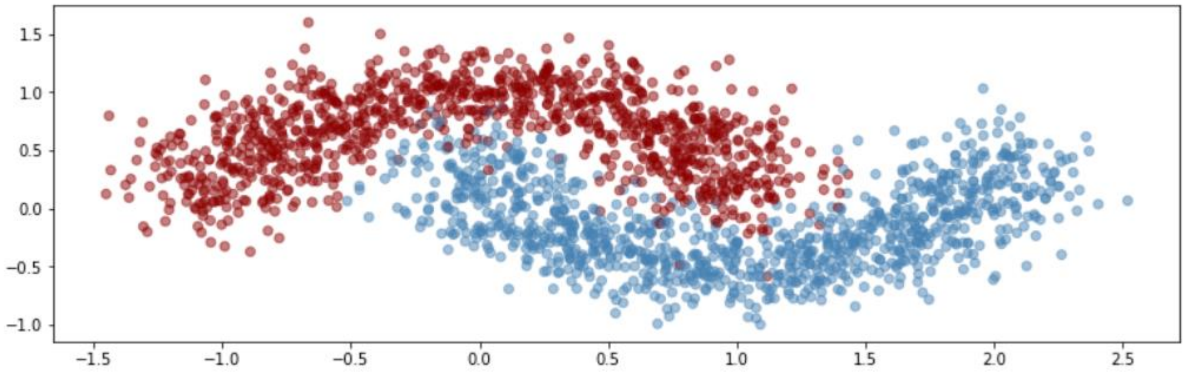


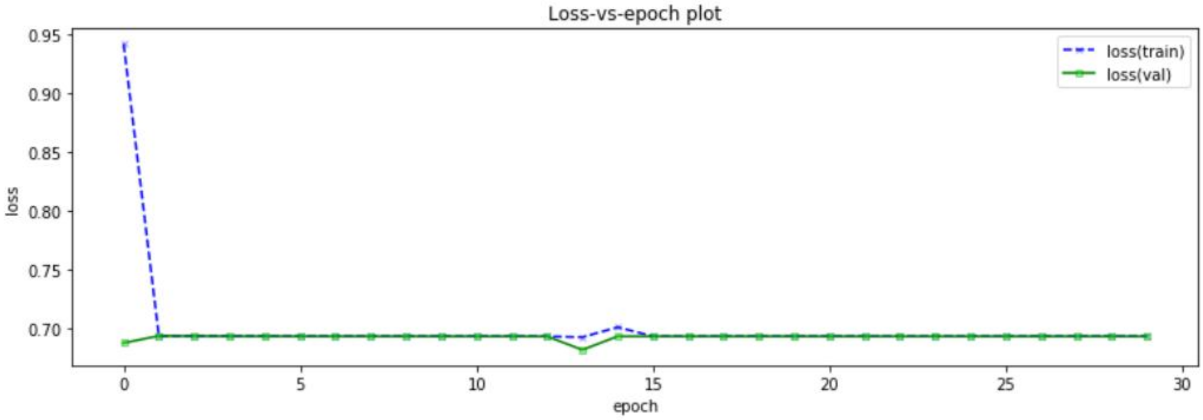
```
OK, training time=2.7
```

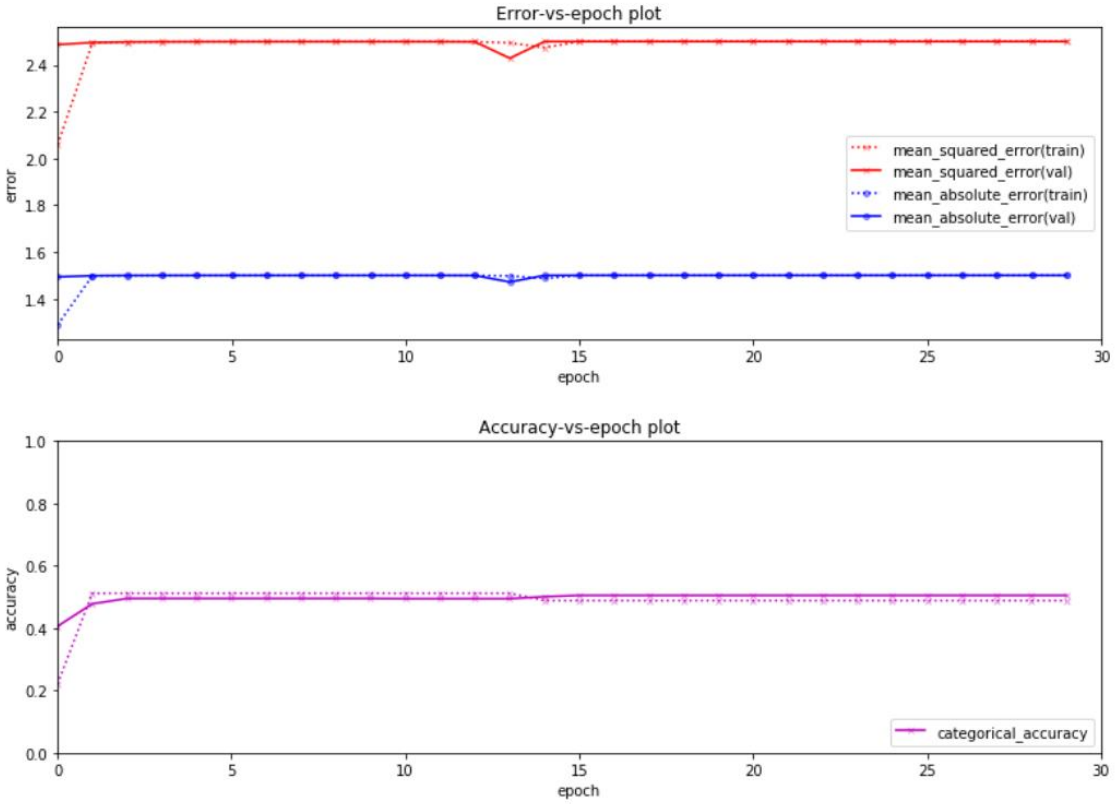
Cell 2



```
Training time: 2.7 sec
Test loss:     0.6931432207425435
Test accuracy: 0.48833333293596903
All scores in history: [0.6931432207425435, 0.48833333293596903, 2.499981838862101, 1.499992618560791]
```

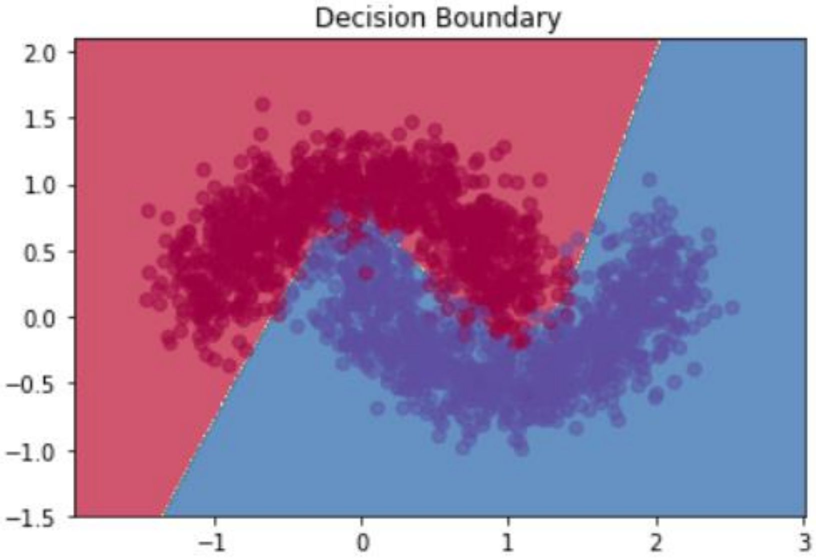




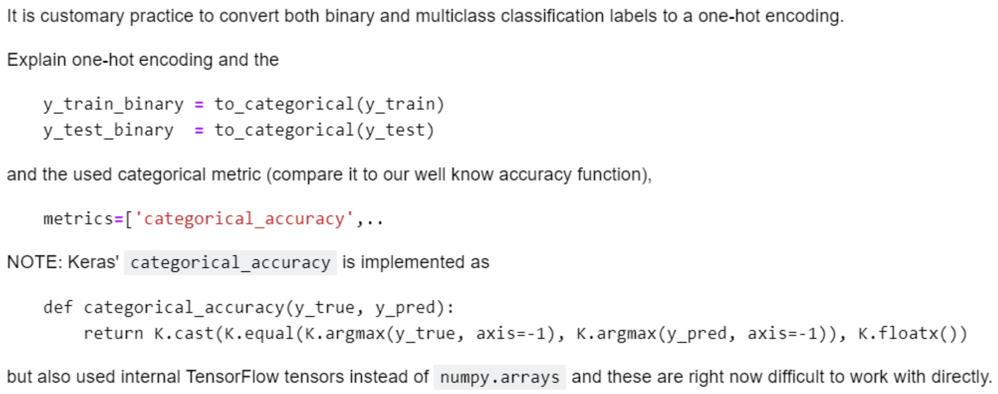


Efter ca. 800 epoker ses det at sgd classifier matcher adam classifier igen. Hvor adam kun skulle bruge ca. 10 epoker for samme resultat.

Cell 3



Qb:



It is customary practice to convert both binary and multiclass classification labels to a one-hot encoding.

Explain one-hot encoding and the

```
y_train_binary = to_categorical(y_train)
y_test_binary  = to_categorical(y_test)
```

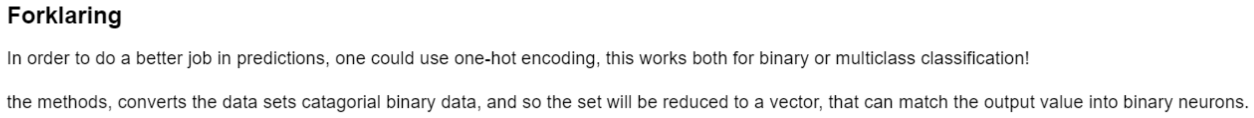
and the used categorical metric (compare it to our well know accuracy function),

```
metrics=['categorical_accuracy',..
```

NOTE: Keras' `categorical_accuracy` is implemented as

```
def categorical_accuracy(y_true, y_pred):
    return K.cast(K.equal(K.argmax(y_true, axis=-1), K.argmax(y_pred, axis=-1)), K.floatx())
```

but also used internal TensorFlow tensors instead of `numpy.arrays` and these are right now difficult to work with directly.

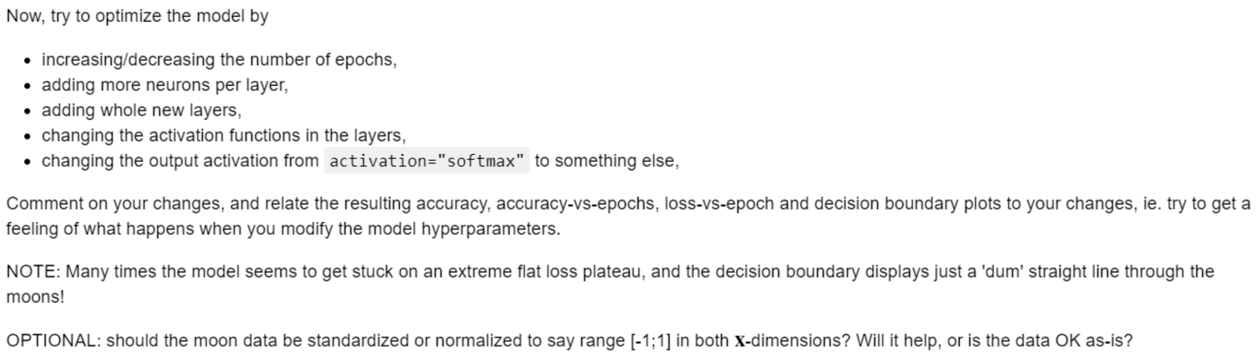


Forklaring

In order to do a better job in predictions, one could use one-hot encoding, this works both for binary or multiclass classification!

the methods, converts the data sets catagorial binary data, and so the set will be reduced to a vector, that can match the output value into binary neurons.

Qc:



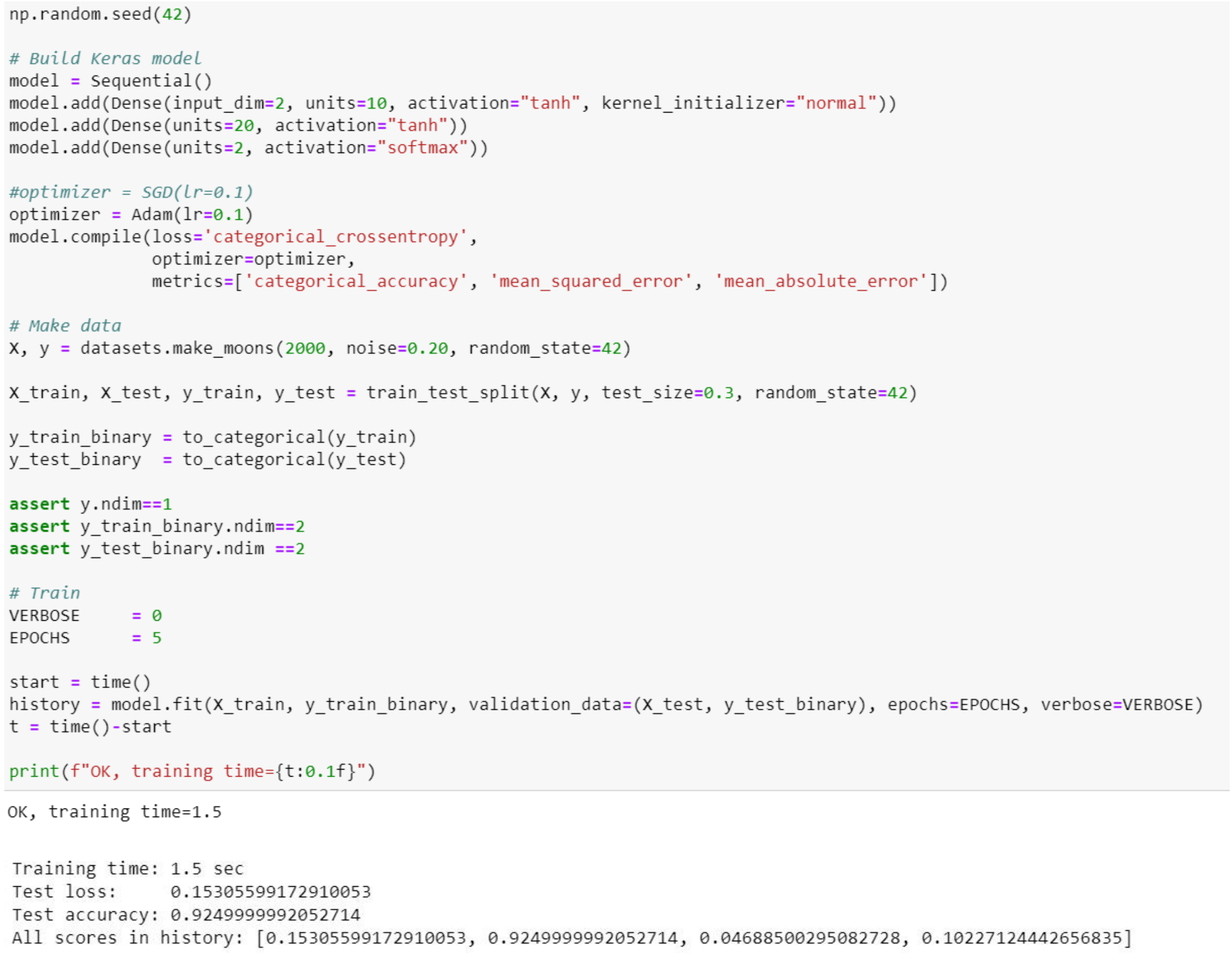
Now, try to optimize the model by

- increasing/decreasing the number of epochs,
- adding more neurons per layer,
- adding whole new layers,
- changing the activation functions in the layers,
- changing the output activation from `activation="softmax"` to something else,

Comment on your changes, and relate the resulting accuracy, accuracy-vs-epochs, loss-vs-epoch and decision boundary plots to your changes, ie. try to get a feeling of what happens when you modify the model hyperparameters.

NOTE: Many times the model seems to get stuck on an extreme flat loss plateau, and the decision boundary displays just a 'dum' straight line through the moons!

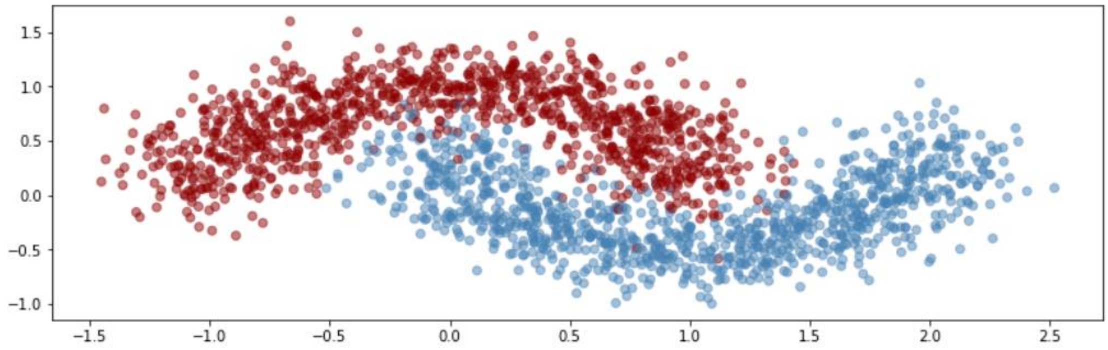
OPTIONAL: should the moon data be standardized or normalized to say range [-1;1] in both $\mathbf{x}$-dimensions? Will it help, or is the data OK as-is?

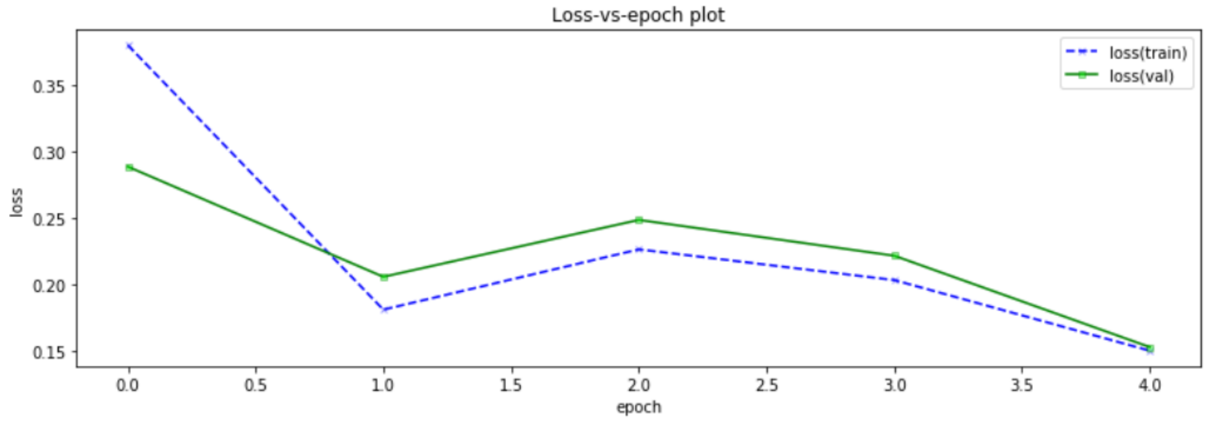


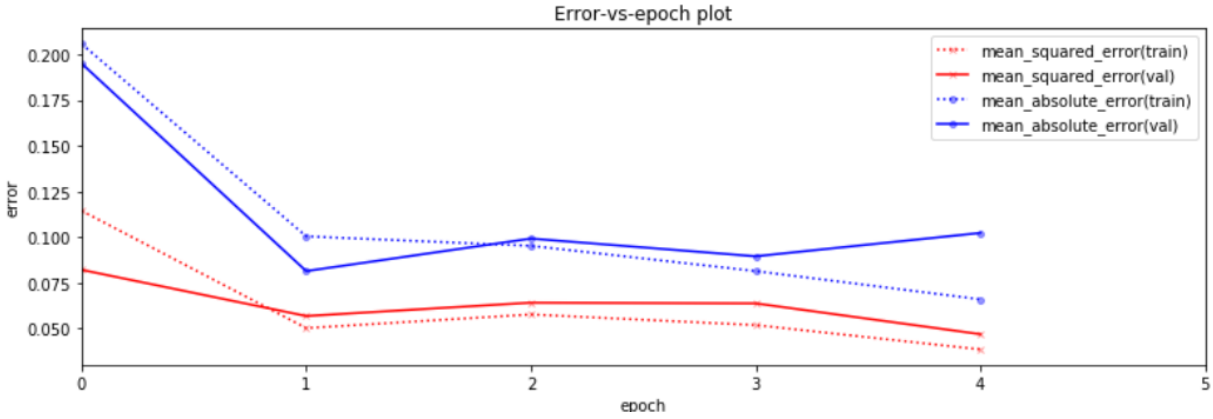
```
np.random.seed(42)

# Build Keras model
model = Sequential()
model.add(Dense(input_dim=2, units=10, activation="tanh", kernel_initializer="normal"))
model.add(Dense(units=20, activation="tanh"))
model.add(Dense(units=2, activation="softmax"))

#optimizer = SGD(lr=0.1)
optimizer = Adam(lr=0.1)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['categorical_accuracy', 'mean_squared_error', 'mean_absolute_error'])

# Make data
X, y = datasets.make_moons(2000, noise=0.20, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

y_train_binary = to_categorical(y_train)
y_test_binary  = to_categorical(y_test)

assert y.ndim==1
assert y_train_binary.ndim==2
assert y_test_binary.ndim ==2

# Train
VERBOSE     = 0
EPOCHS      = 5

start = time()
history = model.fit(X_train, y_train_binary, validation_data=(X_test, y_test_binary), epochs=EPOCHS, verbose=VERBOSE)
t = time()-start

print(f"OK, training time={t:0.1f}")
```

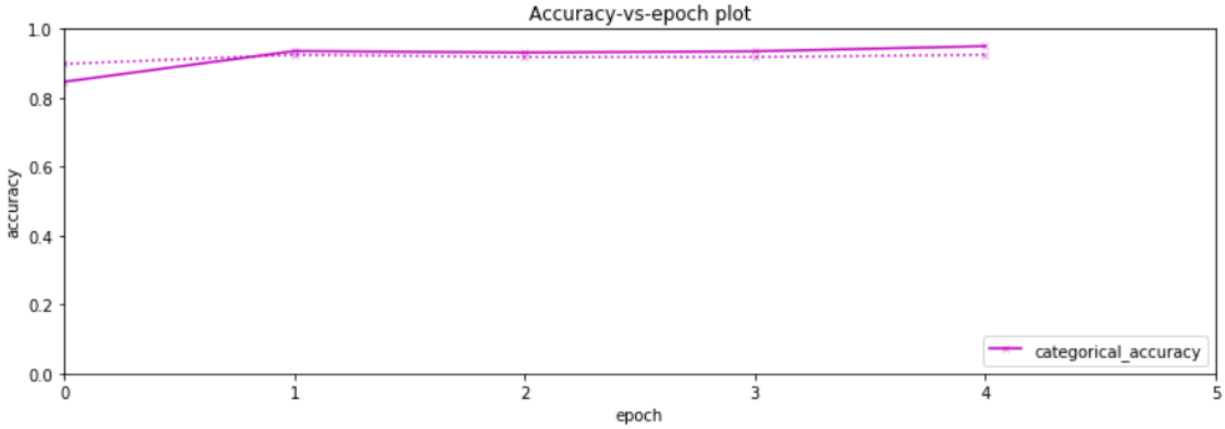
```
OK, training time=1.5
```

```
Training time: 1.5 sec
Test loss:     0.15305599172910053
Test accuracy: 0.9249999992052714
All scores in history: [0.15305599172910053, 0.9249999992052714, 0.04688500295082728, 0.10227124442656835]
```









Forklaring

Som det kan ses ud fra billeder på det nye data, har vi opnået en accuracy op ca. 0.9 allerede i første epoke. vi valgte at sætte 5 på for at vise, at det ikke var brug for flere epoker. Denne optimering er også opnået ved at indsætte el lag i midten af indgang og udgangen.

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.

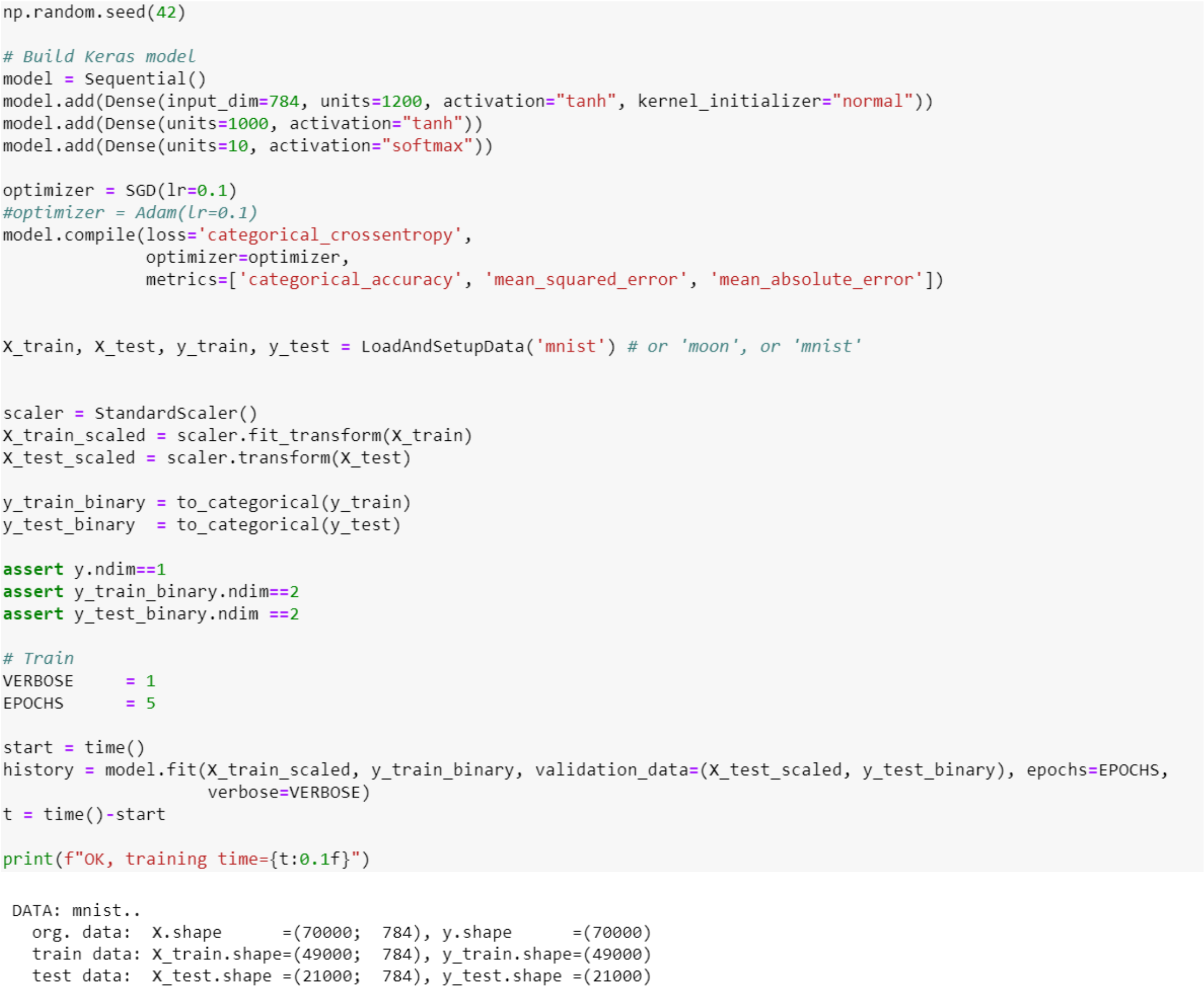
Using biology, more exact the the human brain as fundamental concept of this, we will use the neurons as a kind of "best optimized" to solve the problems as training.

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:

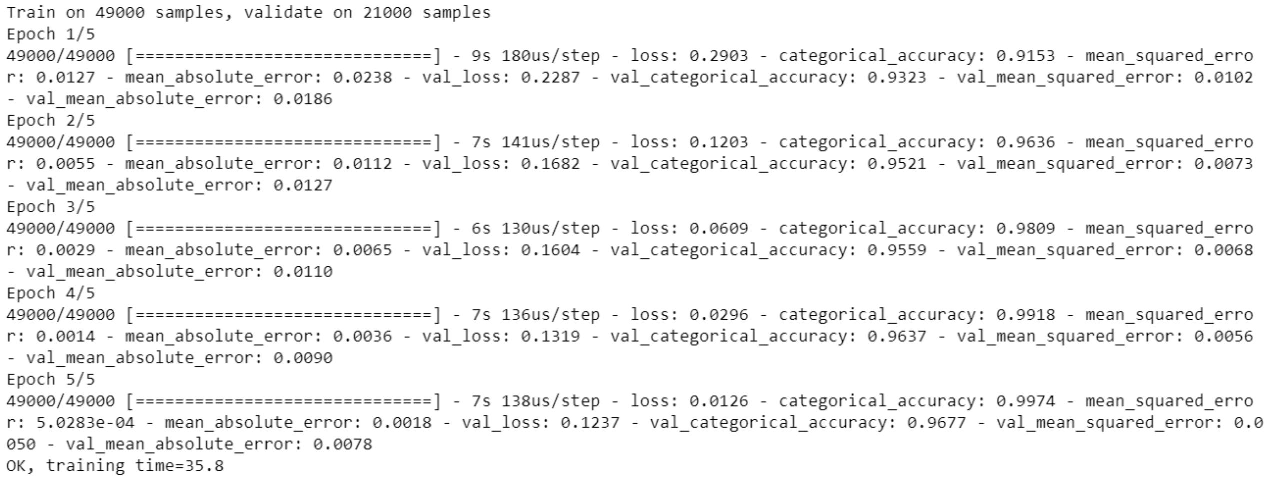
In this first question, we will use the concepts explained in the introduction, to redo the mlp_moon.ipynb, into this new way of thinking.

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.



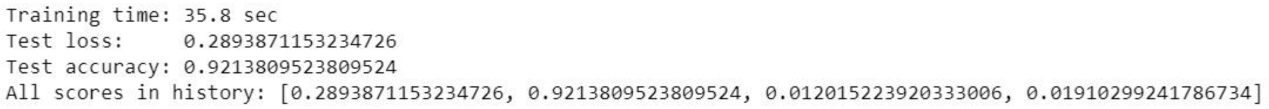
```
np.random.seed(42)

# Build Keras model
model = Sequential()
model.add(Dense(input_dim=784, units=1200, activation="tanh", kernel_initializer="normal"))
model.add(Dense(units=1000, activation="tanh"))
model.add(Dense(units=10, activation="softmax"))

optimizer = SGD(lr=0.1)
#optimizer = Adam(lr=0.1)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['categorical_accuracy', 'mean_squared_error', 'mean_absolute_error'])


X_train, X_test, y_train, y_test = LoadAndSetupData('mnist') # or 'moon', or 'mnist'


scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

y_train_binary = to_categorical(y_train)
y_test_binary  = to_categorical(y_test)

assert y.ndim==1
assert y_train_binary.ndim==2
assert y_test_binary.ndim ==2

# Train
VERBOSE     = 1
EPOCHS      = 5

start = time()
history = model.fit(X_train_scaled, y_train_binary, validation_data=(X_test_scaled, y_test_binary), epochs=EPOCHS,
                    verbose=VERBOSE)
t = time()-start

print(f"OK, training time={t:0.1f}")
```

```
 DATA: mnist..
   org. data:  X.shape      =(70000;  784), y.shape      =(70000)
   train data: X_train.shape=(49000;  784), y_train.shape=(49000)
   test data:  X_test.shape =(21000;  784), y_test.shape =(21000)
```

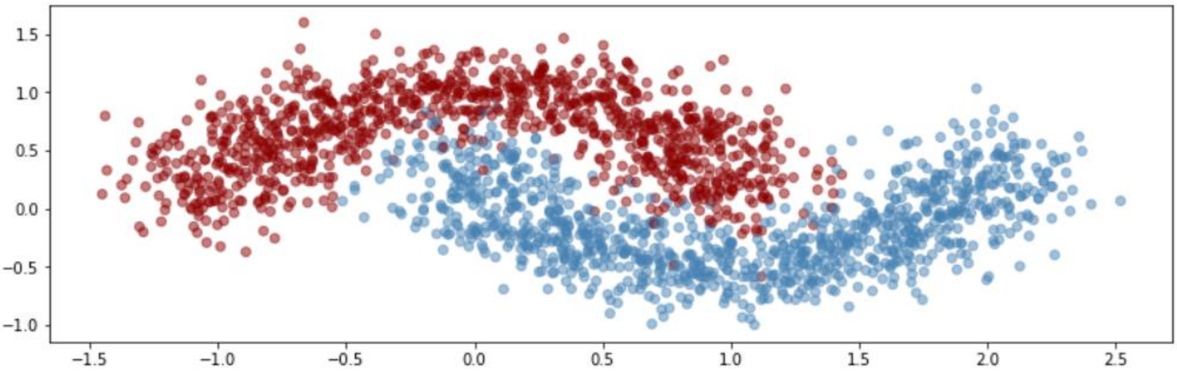


```
Train on 49000 samples, validate on 21000 samples
Epoch 1/5
49000/49000 [==============================] - 9s 180us/step - loss: 0.2903 - categorical_accuracy: 0.9153 - mean_squared_erro
r: 0.0127 - mean_absolute_error: 0.0238 - val_loss: 0.2287 - val_categorical_accuracy: 0.9323 - val_mean_squared_error: 0.0102
- val_mean_absolute_error: 0.0186
Epoch 2/5
49000/49000 [==============================] - 7s 141us/step - loss: 0.1203 - categorical_accuracy: 0.9636 - mean_squared_erro
r: 0.0055 - mean_absolute_error: 0.0112 - val_loss: 0.1682 - val_categorical_accuracy: 0.9521 - val_mean_squared_error: 0.0073
- val_mean_absolute_error: 0.0127
Epoch 3/5
49000/49000 [==============================] - 6s 130us/step - loss: 0.0609 - categorical_accuracy: 0.9809 - mean_squared_erro
r: 0.0029 - mean_absolute_error: 0.0065 - val_loss: 0.1604 - val_categorical_accuracy: 0.9559 - val_mean_squared_error: 0.0068
- val_mean_absolute_error: 0.0110
Epoch 4/5
49000/49000 [==============================] - 7s 136us/step - loss: 0.0296 - categorical_accuracy: 0.9918 - mean_squared_erro
r: 0.0014 - mean_absolute_error: 0.0036 - val_loss: 0.1319 - val_categorical_accuracy: 0.9637 - val_mean_squared_error: 0.0056
- val_mean_absolute_error: 0.0090
Epoch 5/5
49000/49000 [==============================] - 7s 138us/step - loss: 0.0126 - categorical_accuracy: 0.9974 - mean_squared_erro
r: 5.0283e-04 - mean_absolute_error: 0.0018 - val_loss: 0.1237 - val_categorical_accuracy: 0.9677 - val_mean_squared_error: 0.0
050 - val_mean_absolute_error: 0.0078
OK, training time=35.8
```

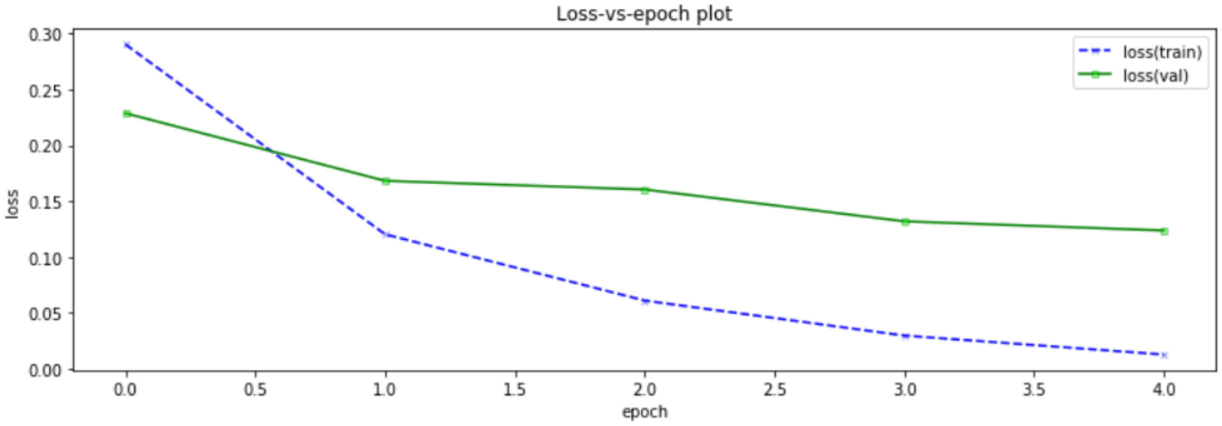
After running the code, the training time is shown below, because we used 5 epocs the training time got a little steep, we ended up using 35 seconds realizing this data, on the GPU cluster.

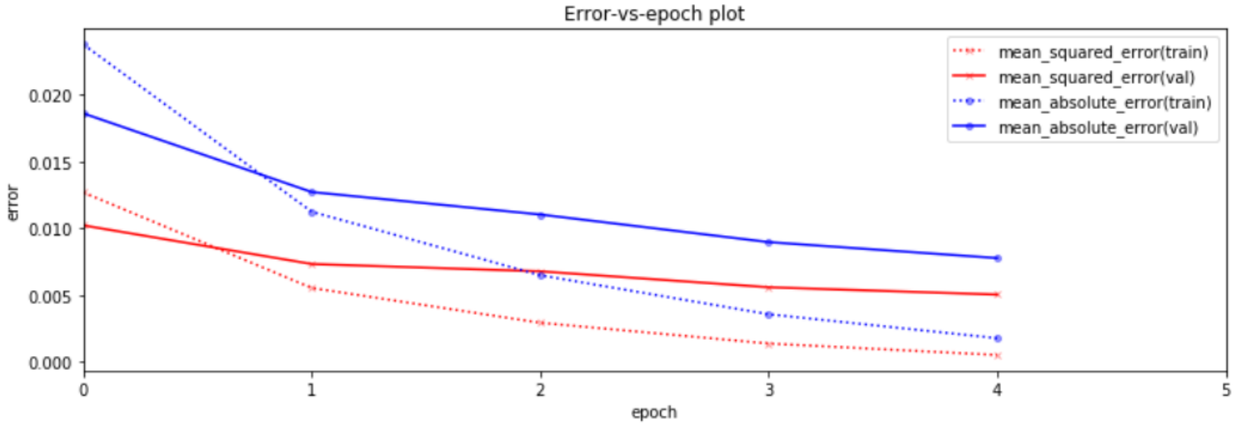


```
Training time: 35.8 sec
Test loss:     0.2893871153234726
Test accuracy: 0.9213809523809524
All scores in history: [0.2893871153234726, 0.9213809523809524, 0.012015223920333006, 0.01910299241786734]
```

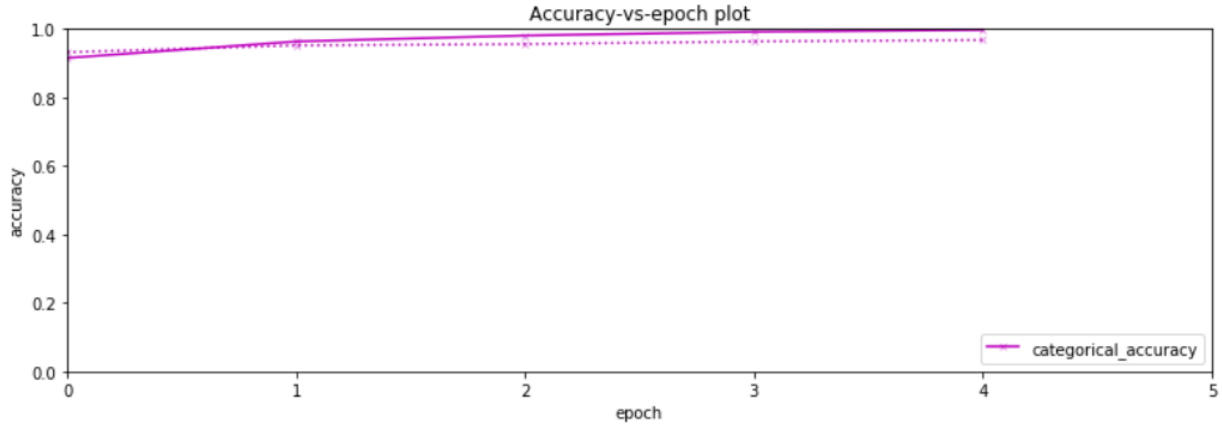


Plotting below is the loss-vs-epoch and the error-vs epoch, as shown. The training data is getting really good, however the test data is only getting slightly better per epoch, this is a scenario of overfitting, but we think, that its still within acceptable ranges.



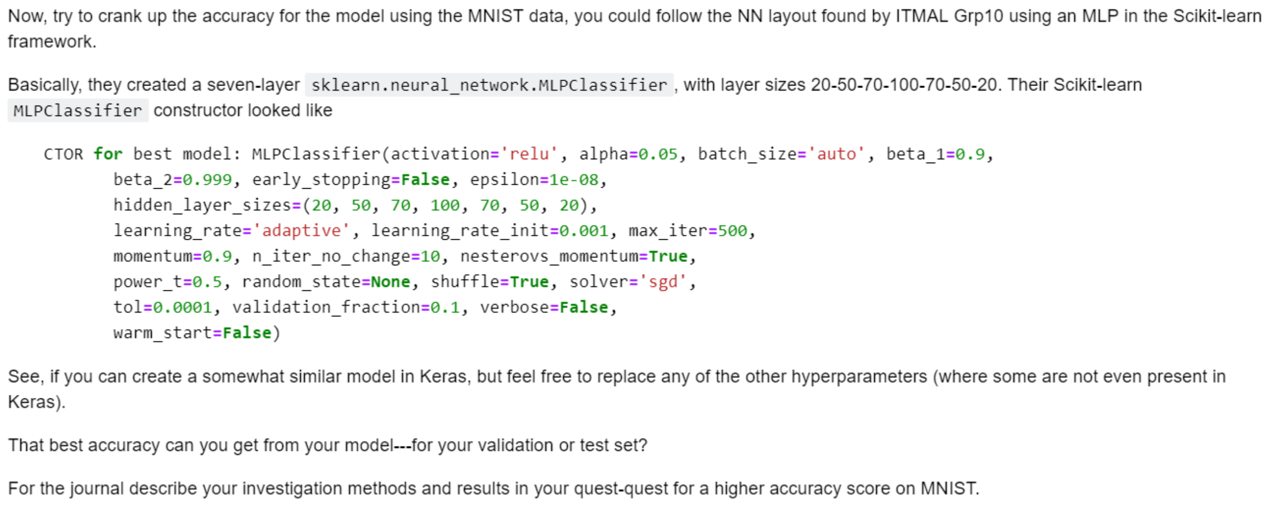


Finally, after all we get an accuracy that is quite acceptable. We are getting really near one here



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.



Now, try to crank up the accuracy for the model using the MNIST data, you could follow the NN layout found by ITMAL Grp10 using an MLP in the Scikit-learn framework.

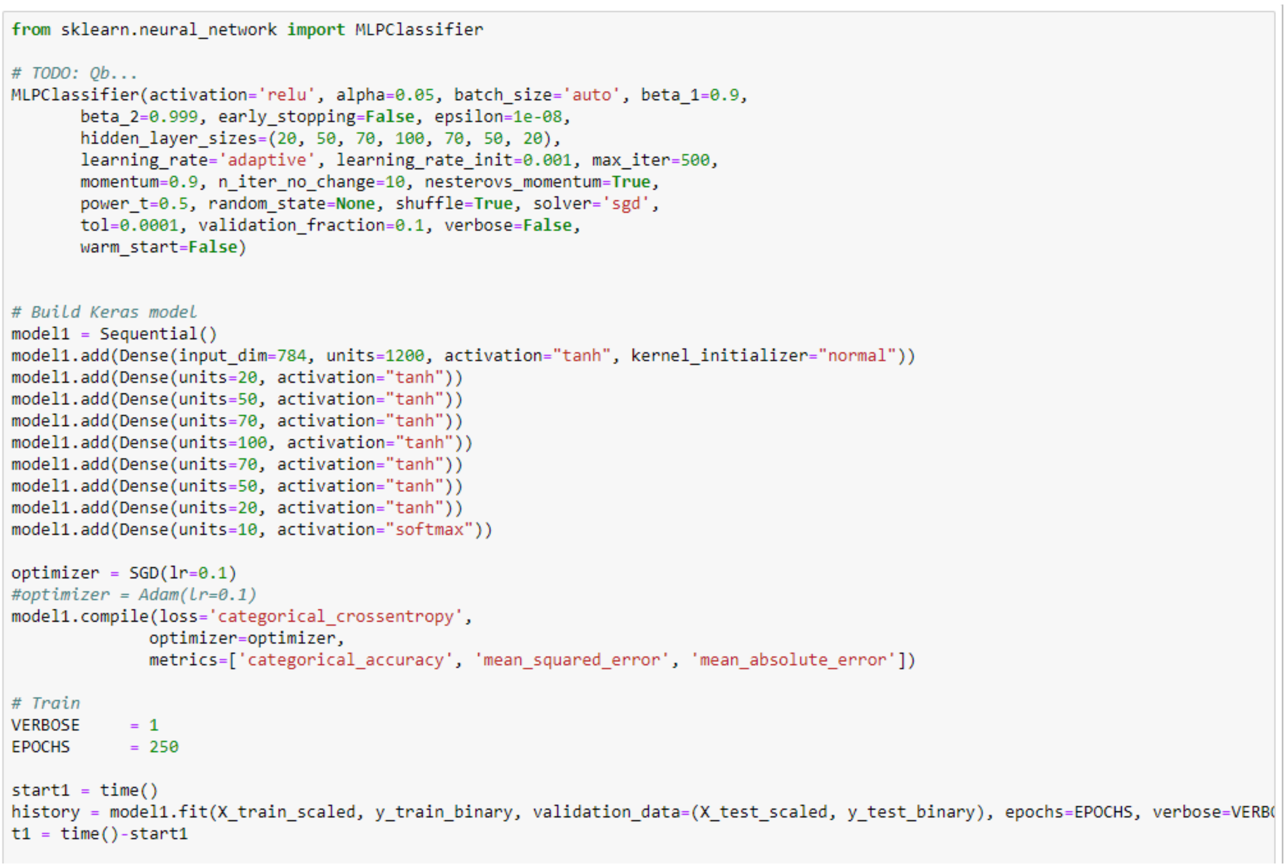
Basically, they created a seven-layer `sklearn.neural_network.MLPClassifier`, with layer sizes 20-50-70-100-70-50-20. Their Scikit-learn `MLPClassifier` constructor looked like

```
CTOR for best model: MLPClassifier(activation='relu', alpha=0.05, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(20, 50, 70, 100, 70, 50, 20),
       learning_rate='adaptive', learning_rate_init=0.001, max_iter=500,
       momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
       power_t=0.5, random_state=None, shuffle=True, solver='sgd',
       tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)
```

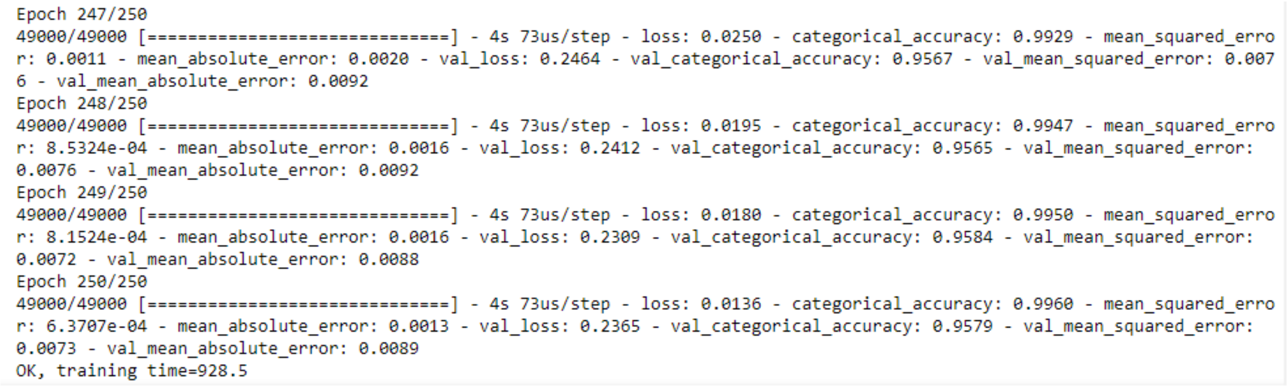
See, if you can create a somewhat similar model in Keras, but feel free to replace any of the other hyperparameters (where some are not even present in Keras).

That best accuracy can you get from your model---for your validation or test set?

For the journal describe your investigation methods and results in your quest-quest for a higher accuracy score on MNIST.

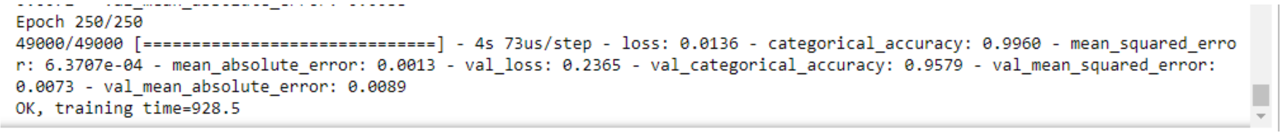


```
from sklearn.neural_network import MLPClassifier

# TODO: Qb...
MLPClassifier(activation='relu', alpha=0.05, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(20, 50, 70, 100, 70, 50, 20),
       learning_rate='adaptive', learning_rate_init=0.001, max_iter=500,
       momentum=0.9, n_iter_no_change=10, nesterovs_momentum=True,
       power_t=0.5, random_state=None, shuffle=True, solver='sgd',
       tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)


# Build Keras model
model1 = Sequential()
model1.add(Dense(input_dim=784, units=1200, activation="tanh", kernel_initializer="normal"))
model1.add(Dense(units=20, activation="tanh"))
model1.add(Dense(units=50, activation="tanh"))
model1.add(Dense(units=70, activation="tanh"))
model1.add(Dense(units=100, activation="tanh"))
model1.add(Dense(units=70, activation="tanh"))
model1.add(Dense(units=50, activation="tanh"))
model1.add(Dense(units=20, activation="tanh"))
model1.add(Dense(units=10, activation="softmax"))

optimizer = SGD(lr=0.1)
#optimizer = Adam(lr=0.1)
model1.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['categorical_accuracy', 'mean_squared_error', 'mean_absolute_error'])

# Train
VERBOSE     = 1
EPOCHS      = 250

start1 = time()
history = model1.fit(X_train_scaled, y_train_binary, validation_data=(X_test_scaled, y_test_binary), epochs=EPOCHS, verbose=VERB
t1 = time()-start1
```



```
Epoch 247/250
49000/49000 [==============================] - 4s 73us/step - loss: 0.0250 - categorical_accuracy: 0.9929 - mean_squared_erro
r: 0.0011 - mean_absolute_error: 0.0020 - val_loss: 0.2464 - val_categorical_accuracy: 0.9567 - val_mean_squared_error: 0.007
6 - val_mean_absolute_error: 0.0092
Epoch 248/250
49000/49000 [==============================] - 4s 73us/step - loss: 0.0195 - categorical_accuracy: 0.9947 - mean_squared_erro
r: 8.5324e-04 - mean_absolute_error: 0.0016 - val_loss: 0.2412 - val_categorical_accuracy: 0.9565 - val_mean_squared_error:
0.0076 - val_mean_absolute_error: 0.0092
Epoch 249/250
49000/49000 [==============================] - 4s 73us/step - loss: 0.0180 - categorical_accuracy: 0.9950 - mean_squared_erro
r: 8.1524e-04 - mean_absolute_error: 0.0016 - val_loss: 0.2309 - val_categorical_accuracy: 0.9584 - val_mean_squared_error:
0.0072 - val_mean_absolute_error: 0.0088
Epoch 250/250
49000/49000 [==============================] - 4s 73us/step - loss: 0.0136 - categorical_accuracy: 0.9960 - mean_squared_erro
r: 6.3707e-04 - mean_absolute_error: 0.0013 - val_loss: 0.2365 - val_categorical_accuracy: 0.9579 - val_mean_squared_error:
0.0073 - val_mean_absolute_error: 0.0089
OK, training time=928.5
```

That best accuracy can you get from your model---for your validation or test set?



```
Epoch 250/250
49000/49000 [==============================] - 4s 73us/step - loss: 0.0136 - categorical_accuracy: 0.9960 - mean_squared_erro
r: 6.3707e-04 - mean_absolute_error: 0.0013 - val_loss: 0.2365 - val_categorical_accuracy: 0.9579 - val_mean_squared_error:
0.0073 - val_mean_absolute_error: 0.0089
OK, training time=928.5
```

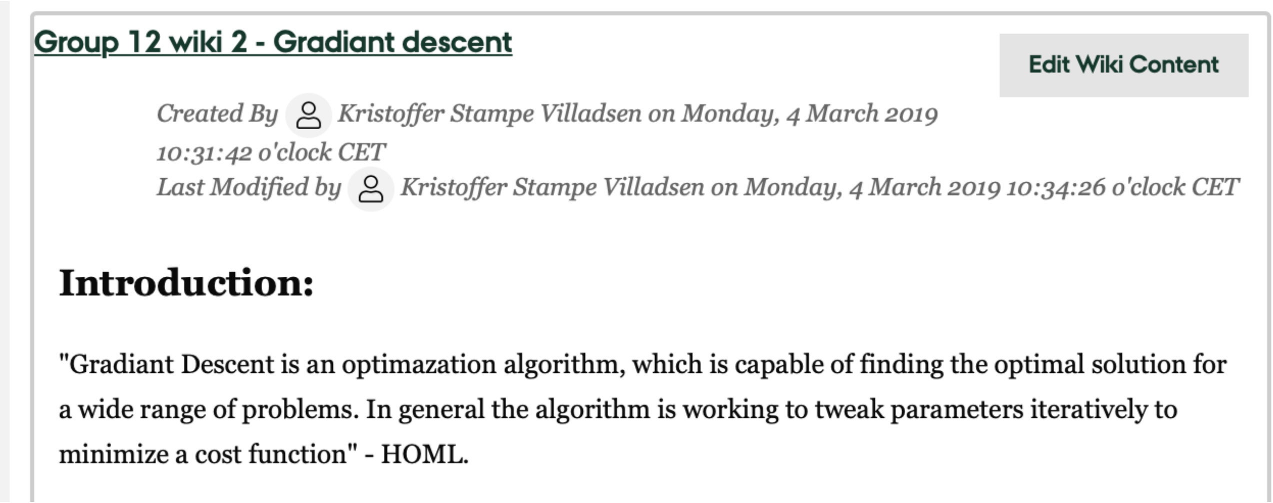
After running epoch number 250, we have realized our code. Now we see an accuracy of 0.9960, and a MSE of only 0.0073 on the test data! We have finally realized something useful, and we can conclude the experiments after 928 seconds of runtime on the GPU cluster.

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



Group 12 wiki 2 - Gradiant descent

Created By Kristoffer Stampe Villadsen on Monday, 4 March 2019 10:31:42 o'clock CET
Last Modified by Kristoffer Stampe Villadsen on Monday, 4 March 2019 10:34:26 o'clock CET

Introduction:

"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.

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.

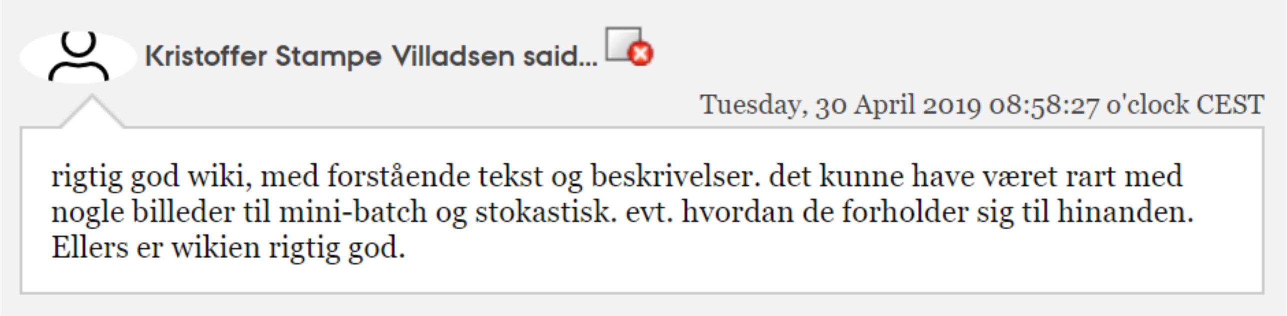
References

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

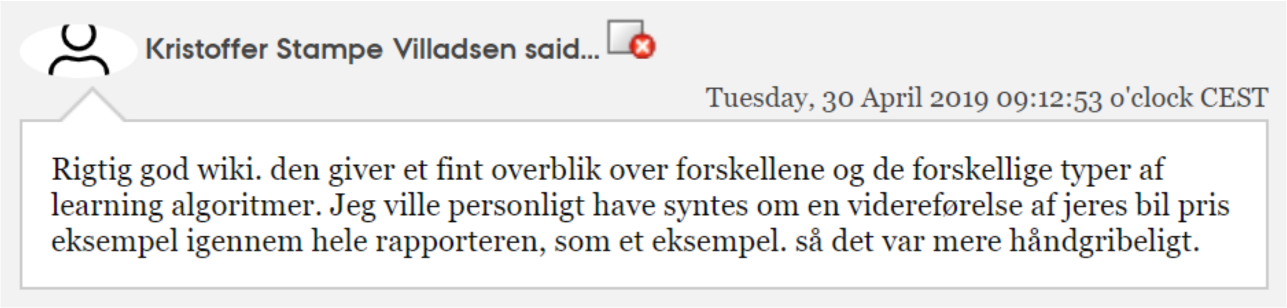


Kristoffer Stampe Villadsen said...

Tuesday, 30 April 2019 08:58:27 o'clock CEST

rigtig god wiki, med forstående tekst og beskrivelser. det kunne have været rart med nogle billeder til mini-batch og stokastisk. evt. hvordan de forholder sig til hinanden. Ellers er wikien rigtig god.

Gruppe 1 – supervised/ unsupervised



Kristoffer Stampe Villadsen said...

Tuesday, 30 April 2019 09:12:53 o'clock CEST

Rigtig god wiki. den giver et fint overblik over forskellene og de forskellige typer af learning algoritmer. Jeg ville personligt have syntes om en videreførelse af jeres bil pris eksempel igennem hele rapporteren, som et eksempel. så det var mere håndgribeligt.