Machine Learning - Assignment 3

Frederik Predut - StudieNr: 201502305

Magnus Hauge Kyneb - StudieNr: 201807279

Sebastian Laczek Nielsen - StudieNr: 201806678

Gruppe 22 - November 2020

Table of Contents

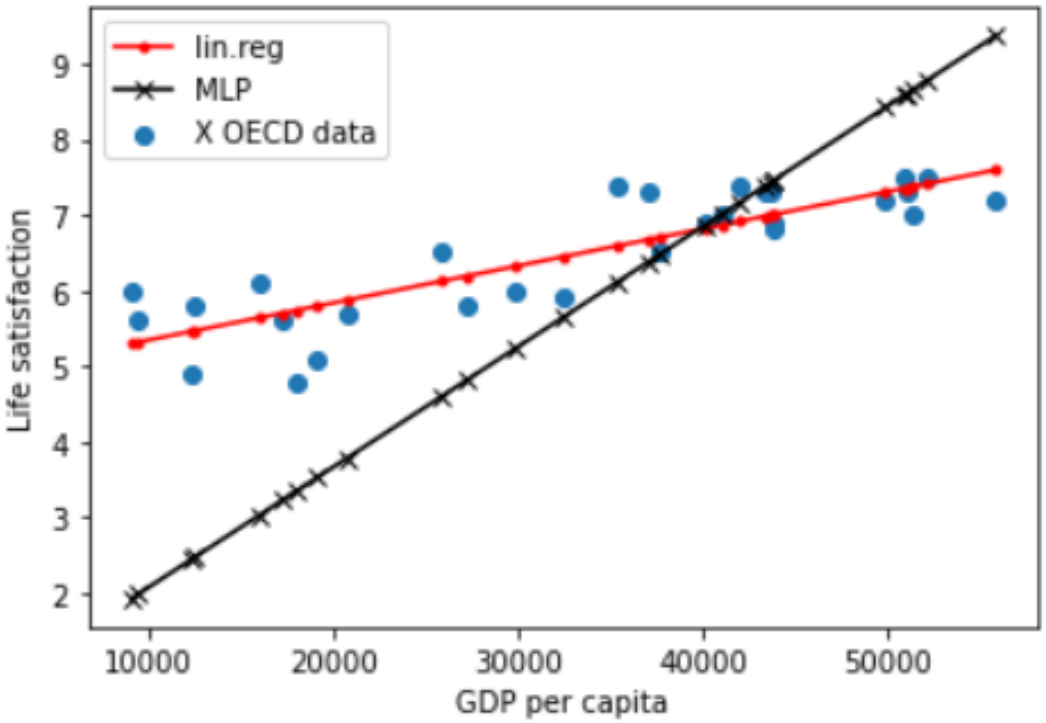
1 L07

1.1 pipelines

1.1.1 Qa - Create a Min/max scaler for the MLP

We see on figure 1 that the MLPregressor produces a negativ $R^2 score = -3.67$

```
The MLP mis-fits the data, seen in the bad R^2 score..
lin.reg.score(X, y)=0.73
MLP.score(X, y)=-3.67
```



Figur 1: negativ MLP score with unnormalized data

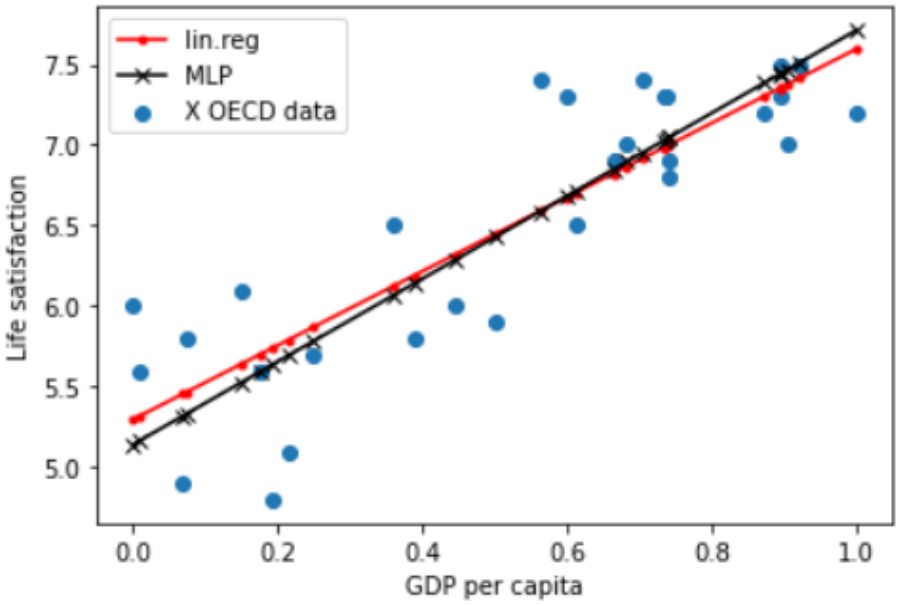
Try to manually scale X to a range of [0;1], re-train the MLP, re-plot and find the new score from the rescaled input. Any better?:

we have a function that normalizes our data in a range of [0;1] with the normalizeData() function we've implemented ourselfs see figure 2 we then retrain our model and plot the various result see figure 3. The MLP now has a R2 score that matches the linear reg R2 score

```python
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

# function normalizes data
def normalizeData(x):
    return (x-np.min(x)) / (np.max(x) - np.min(x))

# normalizing X (OECD data)
X = normalizeData(X)

# print X, values range between 0 and 1
#print('normalized data to range [0;1]: ', X)
#print(X)

# retaining the model
mlp.fit(X, y)
linreg.fit(X, y)

#print("The MLP mis-fits the data, seen in the bad R^2 score..")
PlotModels(linreg, mlp, X, y)
```

Figur 2: manually scaled input, retrained and plotted

```
lin.reg.score(X, y)=0.73
MLP.score(X, y)=0.72
```



Figur 3: Positiv R2 score

Explain the fundamental problem with a min-max scaler and outliers. Will a sklearn.preprocessing.Stan do better here, in the case of abnormal feature values/outliers?:

The fundamental problem is that the min-max scaler is sensitive to outliers, when it scales data between 0-1 if there is a few outliers with extreme values, it almost negates the rest of the data.

1.1.2 Qb Scikit-learn Pipelines

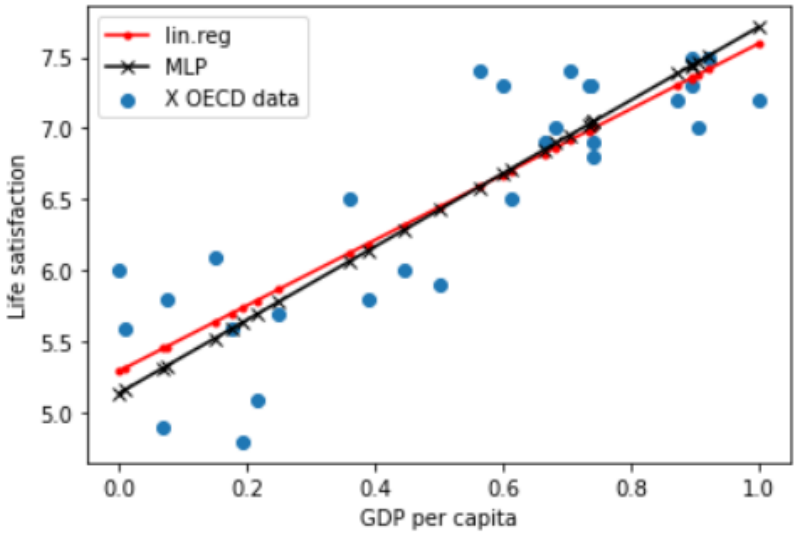
Now, rescale again, but use the sklearn.preprocessing.MinMaxScaler.When this works put both the MLP and the scaler into a composite construction via sklearn.pipeline.Pipeline. This composite is just a new Scikit-learn estimator, and can be used just like any other fit-predict models, try it, and document it for the journal:

We've implemted the pipeline, see figure 4 with the MLP and scaler and plot the results see figure 5

```
from sklearn.pipeline import Pipeline

#using MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
mlp.fit(X_scaled, y)

y_pred_mlp = mlp.predict(X_scaled)
print(mlp.score(X_scaled, y)) # 0.72

# pipeline - from lesson L07 lecture
pipe = Pipeline(
    [
        ('scaler', MinMaxScaler()),
        ('mlp', mlp)
    ]
)

pipe.fit(X,y)
print(pipe.score(X_scaled, y))
PlotModels(linreg, mlp, X_scaled, y)
```

Figur 4: Positiv R2 score

```
0.7239243376870621
0.7239243376870621
lin.reg.score(X, y)=0.73
MLP.score(X, y)=0.72
```



Figur 5: Positiv R2 score

as seen on figure 5 we run the pipeline and get the result for it.

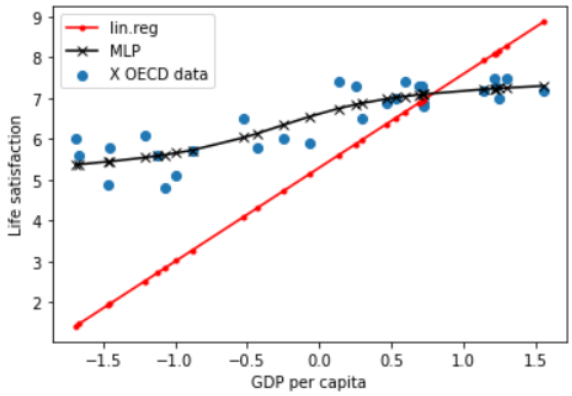
1.1.3 Qc Will a sklearn.preprocessing.StandardScaler do better here, in the case of abnormal feature values/outliers?

We changed the scaler from MinMax to StandardScaler and reused the pipeline from Qb see 6

```python
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

#using StandardScaler
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
mlp.fit(X_scaled, y)

y_pred_mlp = mlp.predict(X_scaled)
print(mlp.score(X_scaled, y)) # 0.72

# pipeline - from lesson L07 lecture
pipe = Pipeline(
    [
        ('scaler', StandardScaler()),
        ('mlp', mlp)
    ]
)

pipe.fit(X,y)
print(pipe.score(X_scaled, y))
PlotModels(linreg, mlp, X_scaled, y)
```

Figur 6: Standard Scaler with pipeline

```
0.7364875605477428
-0.9474726811262661
lin.reg.score(X, y)=-5.07
MLP.score(X, y)=0.78
```



Figur 7: Plot of Standard Scaler with pipeline

1.1.4 Qd Modify the MLP Hyperparameters

Finally, try out some of the hyperparameters associated with the MLP. Specifically, test how few neurons the MLP can do with—still producing a sensible output, i.e. high R2. Also try-out some other activation functions, ala sigmoid, and solvers, like sgd.

we've tried out some of the activation functions see figure and played with hyperparameters 8 and printet the score for them 8

```python
# 3 neurons - activation function = identity
mlp = MLPRegressor(hidden_layer_sizes=(1,),
                   solver='lbfgs',
                   activation='identity',
                   tol=1E-5,
                   max_iter=100000,
                   early_stopping=True,
                   verbose=False)

# train/predict the model
mlp3 = mlp.fit(X,y)
mlp3 = mlp.predict(X)
mlp3_score_R2 = mlp.score(X,y)

# print R^2 score for 3 neurons and activation function =
print('R^2 score with 3 neurons',mlp3_score_R2)


# 6 neurons  - activation function =lbgs
mlp = MLPRegressor(hidden_layer_sizes=(6, ),
                   solver='sgd',
                   activation='logistic',
                   tol=1E-5,
                   max_iter=100000,
                   early_stopping=True,
                   verbose=False)

# train/predict the model
mlp6 = mlp.fit(X,y)
mlp6 = mlp.predict(X)
mlp6_score_R2 = mlp.score(X,y)

# print R^2 score for 6 neurons and activation function =
print('R^2 score with 6 neurons', mlp6_score_R2)



# 9 neurons - activation function = adam
mlp = MLPRegressor(hidden_layer_sizes=(9, ),
                   solver='adam',
                   activation='tanh',
                   tol=1E-5,
                   max_iter=100000,
                   verbose=False)

# train/predict the model
mlp9 = mlp.fit(X,y)
mlp9 = mlp.predict(X)
mlp9_score_R2 = mlp.score(X,y)

# print R^2 score for 9 neurons and activation function = adam
print('R^2 score with 9 neurons',mlp9_score_R2)
```

```
R^2 score with 3 neurons -8.71426017993114
R^2 score with 6 neurons -0.05417423498312113
R^2 score with 9 neurons -10.57287205263725
```

Figur 8: Different activation functions

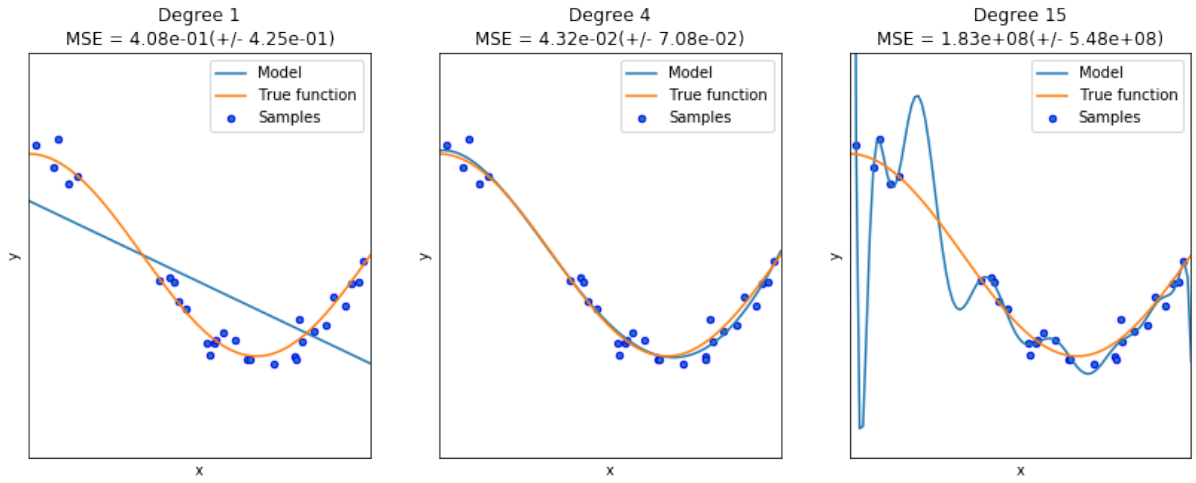
1.2 capacity_underfitting_v2

1.2.1 Qa

code review:
In this code review the plot functions are all skipped to focus on the interesting parts of the code. First we generate data and save it into X and y. Then the degrees array is defined, these are the degrees the polynomial is gonna use later. The interesting happens in the `for loop`, where we add the first Polynomial feature with degrees equal to the first value in the degrees array. This is pipelined with a linearRegression, and then fitted. Hereafter a cross validation score is calculate, and averaged into the score_mean variable. After we predict on the data. This happens for each variable in the degrees array.

All the results are plotted together and shown in the end.

The plots on figure 9 show the how the predictions of three different fitted models.



Figur 9: Model with three different capacities

1.2.2 Qb

Describe capacity and under/overfitting concepts using the plots on figure 9

Model Capacity:
This describes how complex data a model can handle. With simple data, there would be no reason to have a high capacity as this would lead to overfitting. And to understand a complex data set, it would not be wise to use a low capacity, as there would be no meaningful outcome, and would result in underfitting.

Uderfitting:
When the capacity is too low for a complex problem underfitting happens, this is shown on the first figure with a 1 degree polynomial. In this case the model is so simple it tries to predict linearly for a problem that is not linear.

Overfitting:
When a capactity is too high for a given problem, overfitting happens. This is shown on figure 3, with a 15 degree polynomial. In this case the model overcompensates, for indivual values, and makes wide swing predictions, when in reality it is much simpler.

1.2.3 Qc Score method

Why is the scoring method called `neg_mean_squared_error` in the code?
Since we are measuring the performance in errors the lower scores are preferable, instead of high scores are better. This means we negate the high scores, so they properly reflect what the desired scores are.

What happens if you try to set it to mean_squared_error?
Changing this just throws an exception, since it is not a valid argument, as seen on figure 10

```
ValueError: 'mean_squared_error' is not a valid scoring value.
```

Figur 10: Exception from changing the scoring

Why does the 15 degree model produce a low error score and why is this not the best model
When overfitted the model produces a low error score because it overcompensates to reach all the data points, however it does not produce a general summation of understanding the data. This means new data points would not be predicted properly.

1.3 generalizaation_error

1.3.1 Qa

Describe the following concepts:

training/generalization error:
When training a machine learning model, the training set for said model, can be computed to some error measure. This measure is called the training error.

The generalization error is when using a previously unseen input, and measure the error on this set. This is also called the test error. It can also be described as the expected value of the error on a new input

underfit/overfit zone:
Underfitting is when the training set does not produce a low enough error rate on the training set. And overfitting is when the error gap between training and generalization is too large.

optimal capacity:
This fitting problem can be controlled through capacity. Capacity a way for the model to fit a variety complexities. This means increasing the capacity allows the model to fit better on more complex problems, however on simple problems this may overfit. The figure shows this well where increasing the capacity allows the model to decrease the gap between both test and training error, and if the capacity gets too large, it makes the training error lower, but it makes wild assumptions on the new input, because it predicts the data to be more complex.

generalization gap:
This is the gap between training error and generalization error. At optimal capacity this gap is at it lowest.

The two axis X-capacity, y-error:
The x-axis shows the capacity of the model, while the y-axis shows the performance of the model. The best model for the data would then be at a low y-value, which allows to find the optimal capacity.

1.3.2 Qb

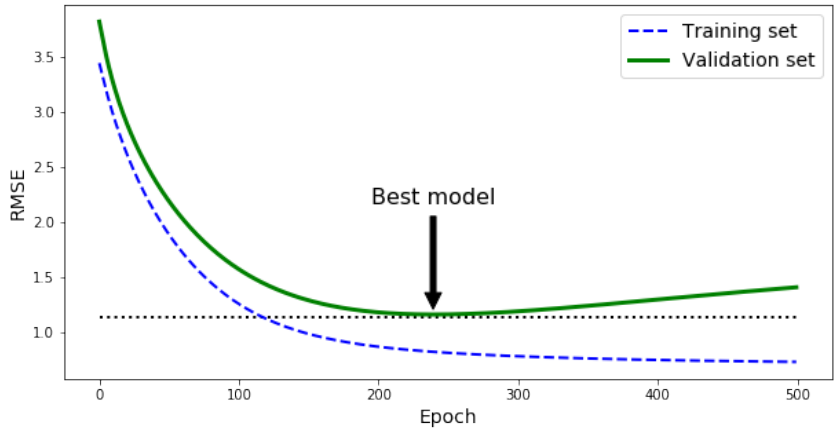
Code review:
Firstly the function GenerateData() just simply generates nonlinear data. Then a train test split is made of this generated data.

The train data is then prepocessed through a pipeline using StandardScaler and PolynomialFeatures.

cell 2:
Here the train function is defined which takes the training and validation data, along with the iteration number(epoch). The epochs determines the amount of times the data is fitted to the SGDRegressor. The results of both the training and validation is appended to their respective arrays.

cell 3:
Both arrays are plotted with the best epoch. this is calculated by finding the index of the minimum value in the validation error array. This is also optimal capacity.

The result can be seen on figure 11.



Figur 11: Graph showing when the best model is achieved

What is an epoch?
epoch is the iteration number of the model. iIt describes how many times the model iterates on its predictions.

What is `mse_train`?
mse_train is the error array for the train predict. It is this array that through each iteration is appended on the train_errors array. **what is `mse_val`?**
mse_val is almost the same as mse_train, it is just the error array for the validation predict. And is appended on the val_errors array.

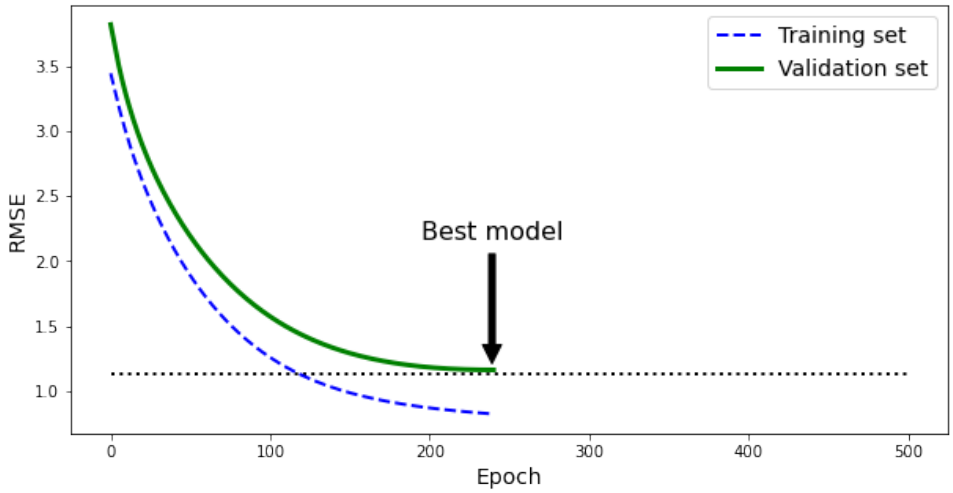
1.3.3 Qc Early stopping

Early stopping is when the model stops after reaching optimal capacity. In the case above the easiest way to early stop is to compare each calculation of the validation set to the previous, this way if it starts increasing it will indicate that it is now overfitting. However it should be mentioned that this will not always work on all datasets, as the curve might swing up and down multiple times but for this dataset its fine.

code for early stopping:

```
1.— if((epoch > 0) and mse_val > mse_prev_val):
2.—       break;
3.— mse_prev_val = mse_val
```

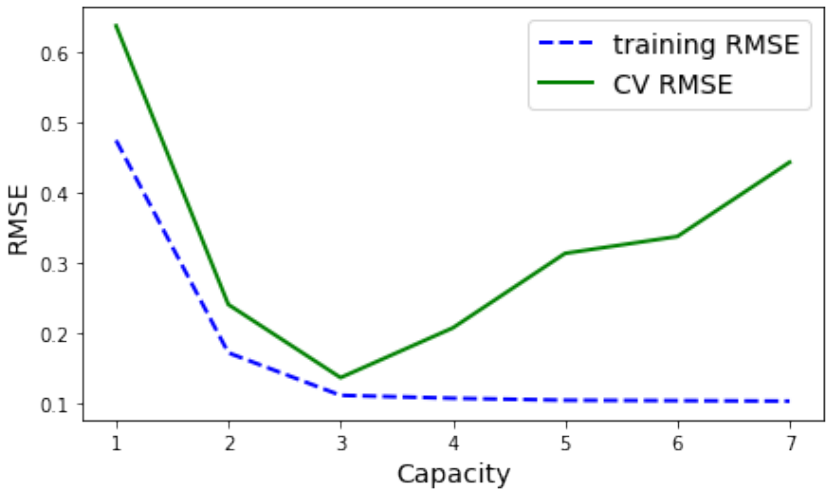
This code should be inserted into the `for loop`. This saves the previous value and compares it to the newest value, and if the new value is higher than the previous it stops the loop, because we are now starting to overfit. On figure 12 is shown the result of running the code now with early stop implemented.



Figur 12: Early stop implemented

1.3.4 Qd Explain the polynomial RMSE-capacity plot

The plot generated from the code is seen on figure **??**

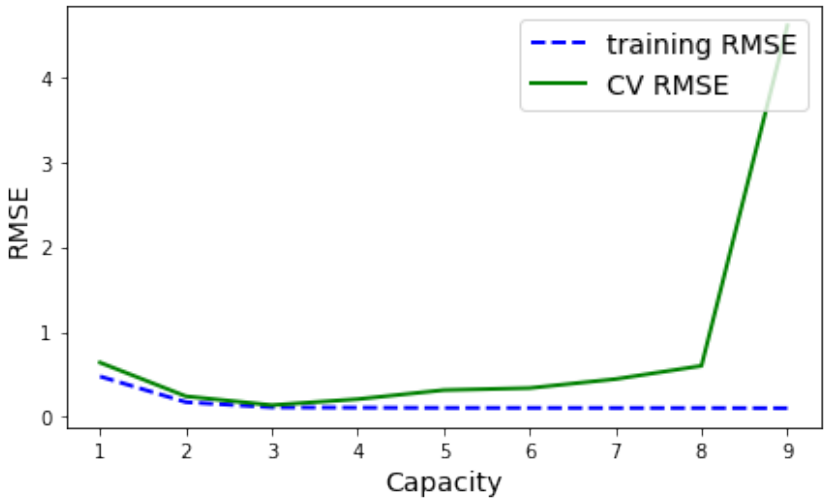


Figur 13: Plot of RMSE-capacity plot

why does the training error keep dropping, while the CV-error drops until around capacity 3? After the CV-error has reached its optimal capacity any further complexities to the prediction of the data, makes the model predict the data to be more complex than in reality, this results in these wild swings in the prediction curve, which in turn makes it more error prone. However for the training data, increasing the complexity just makes the model better at fitting the data.

what does the x-axis Capacity and y-axis RMSE represent:
Capacity is the complexety of the model, and as show here as the amount of degrees the polynomial adds. The y-axis is the performance axis, it measures the proportion of correct predictions.

Try increasing the capacity what happens?
Increasing the capacity does mostly nothing, as it just continuesly overfits, giving no meaningful new information as seen on figure 14. It does however hide the actual information, since it becomes so bad at predicting that it almost hides the original graph. This is also due to the optimal capacity being at 3. This also becomes clearer when comparing the two figures with different capacities, as seen on both figure 14 and 13.



Figur 14: Plot with increased capacity to 10

2 L08

2.1 Regulizers

2.1.1 Qa - The Penalty Factor

We implement the regulizer penalty using the dot product, after removing the bias.

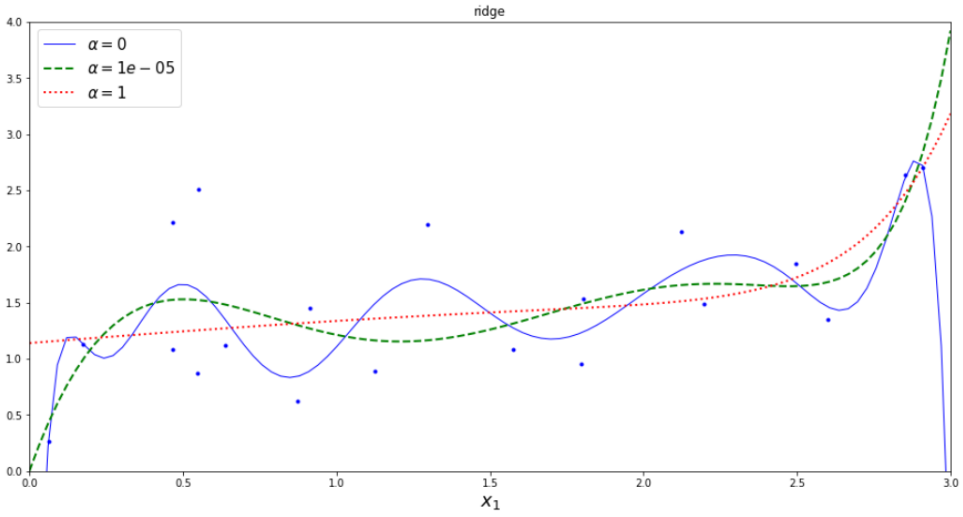
```
def Omega(w):
    w_nobias = np.delete(w, 0)
    w_dot = np.dot(w_nobias, w_nobias)
    return w_dot

# 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)

print(f"P(w0)={p_a}")
print(f"P(w1)={p_b}")
print(f"P(w2)={p_c}")

# TEST VECTORS
e0 = 2*2+(-3)*(-3)
e1 = 9e+20
e2 = 0.13

CheckInRange(p_a, e0)
CheckInRange(p_b, e1)
CheckInRange(p_c, e2)

print("OK")
```

```
P(w0)=13.0
P(w1)=9e+20
P(w2)=0.13
OK
```

Figur 15: Regulizer penalty implementation using dot and without the bias.

2.1.2 Qb - Explain the Ridge plot

Below you can see the resulting ridge plot for different models using various alpha values.



Figur 16: Ridge plot using different levels of alpha (penalty factor constant).

We can see that the different alpha values reguralizes the generated model. The more reguralized (the higher the alpha) the more flat and less overfitted the models becomes. Alpha is a measure of how penalized the model will be for having large weights. At alpha 0, the model could be considered overfitted, as extreme weights are in no way penalized.

2.1.3 Qc - Explain Ridge, Lasso and ElasticNet Regulized Methods

Qc explanation:

Ridge: Ridge regression imposes a penalty on the size of the coefficients (weights). The ridge coefficients minimize a penalized risidual sum of squares. As we understand it, penalize weights going to crazy high values.

This is the function used:

$$\min_w ||Xw - y||_2^2 + \alpha||w||_2^2$$

Lasso: Lasso favors solutions with fewer non-zero coefficients. It effectively reduces the number of features a solution is dependant on. This smells a bit like feature extraction.

This is the function used:

$$\min_w \frac{1}{2n_{\text{samples}}}||Xw - y||_2^2 + \alpha||w||_1$$

ElasticNet: Linear regressionsmodel training with both L1 and L2-norm regularization fo the coefficients. It's a bit of a middle ground between ridge and lasso, in the sense that it penalizes crazy high weight values, while favoring solutions with fewer non-zero weights.

This is the function used:

$$\min_w \frac{1}{2n_{\text{samples}}}||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$

Figur 17: Our explanation of Ridge, Lasso and ElasticNet regulized methods.

2.1.4 Qd - Regularization and Overfitting

In this exercise we explain the tug of war between the loss function and regulizer. The tug of war is that between how well the model fits the data, and how close the weights are to zero.

There's both a penalty for not fitting the data, but also one of having extreme weights. The model must therefore find a middleground, catering to both. This is described as a tug of war.

This can be used to combat overfitting, as overfitting often requires weights of extreme values to match the data points as closely as possible.

2.2 Grid Search

2.2.1 Qa - Explain GridSearchCV

GridSearchCV is a method that allows us to try a bunch of different combinations of hyperparamters, to get a feel for what works best. The method includes fit and score-functionality like we're used to, but takes a parameter-grid that we can specify. The parameter grid specifies what different combinations we're interested in trying.

The report functionality is created to better show us the results. In this case, it seems like the best estimator was C=1 with kernel='linear'.

The following lines...

gridtuned = GridSearchCV(model, tuningparameters, ..
and
gridtuned.fit(X-train, y-train)

...are setting up the model and search space for the grid search funciton and then running a fit-algorithm across the all the different parameter combinations. The model parameter defines what kind of model we're using and trying to fit the data. Could be a linear model, an SGDClassifier or an SCV-model. This affects what kind of function we will end up. All these models have different hyperparameters, which are sort of a meta-level stuff, like how many times the model should iterated to fit better, or what kind of loss-function is used to determine its fit.
The tuningparameters input determines what hyperparameters we want to try. GridSearchCV will try combinations across the hyperparameters specified.
Once the model and the hyparameter-search-space is chosen, the fit is run across all the different hyperparameter combinations and the best models, based on various metrics related to score and loss, are highlighted. This gives us information about what sets of hyperparameters work best.

Tuning parameters can be created as a set of different arrays for each hyperparameter. GridSearch will then search "in a grid" as the name suggests, trying every combination between the array members. In the documentation, the search is described as "exhaustive", meaning every combination is examined.

2.2.2 Qb - Hyperparameter Grid Search using an SGD classifier

To switch to the SGDClassifier, we simple swap the model in the code from the previous exercise. In addition, we change up the tuning parameters to match that of the SGDClassifier. We tried adding enough parameters to the search space, so that the search would take several seconds.

```
# Code copied form Qa, but SVC is replaced with SGDClassifier.

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


# Setup search parameters
model = SGDClassifier()


tuning_parameters_SGD = {
    'penalty':('l1','l2','elasticnet'),
    'alpha':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
    'shuffle':[True, False],
    'epsilon':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
    'l1_ratio':[0, 0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
}


CV = 5
VERBOSE = 0

# Run GridSearchCV for the model
start = time()
grid_tuned_SGD = GridSearchCV(model,
                           tuning_parameters_SGD,
                           cv=CV,
                           scoring='f1_micro',
                           verbose=VERBOSE,
                           n_jobs=-1,
                           iid=True)
grid_tuned_SGD.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(grid_tuned_SGD, X_test, y_test, t)
print('OK(grid-search)')
```

Figur 18: Setup grid search using the SGDClassifier.

Here's the result from the grid search. The helper function FullReport gives us a bunch of feedback. We see the best model found as well as the constructor for the method. We also see an extremely long list of the combinations searched through. The F1-score achieved with the grid search method is 0.99 with a search time of 32.59 seconds.

```
SEARCH TIME: 32.59 sec

Best model set found on train set:

        best parameters={'alpha': 0.01, 'epsilon': 0.0001, 'l1_ratio': 0.8, 'penalty': 'elasticnet', 'shuffle': True}
        best 'f1_micro' score=0.9904761904761905
        best index=1306

Best estimator CTOR:
        SGDClassifier(alpha=0.01, epsilon=0.0001, l1_ratio=0.8, penalty='elasticnet')

Grid scores ('f1_micro') on development set:
        [ 0]: 0.686 (+/-0.311) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0, 'penalty': 'l1', 'shuffle': True}
        [ 1]: 0.781 (+/-0.253) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0, 'penalty': 'l1', 'shuffle': False}
        [ 2]: 0.781 (+/-0.245) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0, 'penalty': 'l2', 'shuffle': True}
        [ 3]: 0.752 (+/-0.185) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0, 'penalty': 'l2', 'shuffle': False}
        [ 4]: 0.819 (+/-0.175) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0, 'penalty': 'elasticnet', 'shuffle': True}
        [ 5]: 0.752 (+/-0.185) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0, 'penalty': 'elasticnet', 'shuffle': False}
        [ 6]: 0.876 (+/-0.166) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.1, 'penalty': 'l1', 'shuffle': True}
        [ 7]: 0.781 (+/-0.253) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.1, 'penalty': 'l1', 'shuffle': False}
        [ 8]: 0.695 (+/-0.253) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.1, 'penalty': 'l2', 'shuffle': True}
        [ 9]: 0.752 (+/-0.185) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.1, 'penalty': 'l2', 'shuffle': False}
        [10]: 0.743 (+/-0.155) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.1, 'penalty': 'elasticnet', 'shuffle': True}
        [11]: 0.705 (+/-0.071) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.1, 'penalty': 'elasticnet', 'shuffle': False}
        [12]: 0.790 (+/-0.267) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.2, 'penalty': 'l1', 'shuffle': True}
        [13]: 0.781 (+/-0.253) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.2, 'penalty': 'l1', 'shuffle': False}
        [14]: 0.810 (+/-0.276) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.2, 'penalty': 'l2', 'shuffle': True}
        [15]: 0.752 (+/-0.185) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.2, 'penalty': 'l2', 'shuffle': False}
        [16]: 0.762 (+/-0.241) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.2, 'penalty': 'elasticnet', 'shuffle': True}
        [17]: 0.705 (+/-0.126) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.2, 'penalty': 'elasticnet', 'shuffle': False}
        [18]: 0.648 (+/-0.344) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.3, 'penalty': 'l1', 'shuffle': True}
        [19]: 0.781 (+/-0.253) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.3, 'penalty': 'l1', 'shuffle': False}
        [20]: 0.610 (+/-0.285) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.3, 'penalty': 'l2', 'shuffle': True}
        [21]: 0.752 (+/-0.185) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.3, 'penalty': 'l2', 'shuffle': False}
        [22]: 0.705 (+/-0.212) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.3, 'penalty': 'elasticnet', 'shuffle': True}
        [23]: 0.724 (+/-0.111) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.3, 'penalty': 'elasticnet', 'shuffle': False}
        [24]: 0.800 (+/-0.353) for {'alpha': 1e-05, 'epsilon': 1e-05, 'l1_ratio': 0.4, 'penalty': 'l1', 'shuffle': True}
```

Figur 19: Result from grid search using the SGDClassifier part 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       1.00      1.00      1.00        16
           1       1.00      0.83      0.91        18
           2       0.79      1.00      0.88        11

    accuracy                           0.93        45
   macro avg       0.93      0.94      0.93        45
weighted avg       0.95      0.93      0.93        45


CTOR for best model: SGDClassifier(alpha=0.01, epsilon=0.0001, l1_ratio=0.8, penalty='elasticnet')

best: dat=iris, score=0.99048, model=SGDClassifier(alpha=0.01,epsilon=0.0001,l1_ratio=0.8,penalty='elasticnet',shuffle=True)

OK(grid-search)
```

Figur 20: Result from grid search using the SGDClassifier part 2.

2.2.3 Qc - Hyperparameter Random Search using an SGD classifier

In this exercise, we do the same as in the previous exercise, but using a randomized search rather than a grid search. Below you see the setup and result.

```
# Code copied form Qb, but replaced GridSearchCV with RandomizedSearch

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


# Setup search parameters
model = SGDClassifier()


tuning_parameters_SGD = {
    'penalty':('l1','l2','elasticnet'),
    'alpha':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
    'shuffle':[True, False],
    'epsilon':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
    'l1_ratio':[0, 0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
}


CV = 5
VERBOSE = 0

# Run RandomizedSearchCV for the model
start = time()
random_tuned_SGD = RandomizedSearchCV(model,
                           tuning_parameters_SGD,
                           n_iter=20,
                           cv=CV,
                           scoring='f1_micro',
                           random_state = 42,
                           verbose=VERBOSE,
                           n_jobs=-1,
                           iid=True)
random_tuned_SGD.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(random_tuned_SGD, X_test, y_test, t)
print('OK(random-search)')
```

Figur 21: Setup randomized search using the SGDClassifier.

The F1-score achieved with the random search method is 0.96, lower than grid search's 0.99. However, random search had a search time of only 0.37 seconds compared to grid search's 32.59 seconds. Here we see the trade-off between search speed and solution performance.

```
SEARCH TIME: 0.37 sec

Best model set found on train set:

        best parameters={'shuffle': True, 'penalty': 'l1', 'l1_ratio': 0.5, 'epsilon': 0.1, 'alpha': 0.01}
        best 'f1_micro' score=0.9619047619047619
        best index=9

Best estimator CTOR:
        SGDClassifier(alpha=0.01, l1_ratio=0.5, penalty='l1')

Grid scores ('f1_micro') on development set:
        [ 0]: 0.876 (+/-0.245) for {'shuffle': True, 'penalty': 'l2', 'l1_ratio': 0, 'epsilon': 0.0001, 'alpha': 0.001}
        [ 1]: 0.924 (+/-0.114) for {'shuffle': True, 'penalty': 'elasticnet', 'l1_ratio': 0.6, 'epsilon': 0.0001, 'alpha': 0.01}
        [ 2]: 0.829 (+/-0.196) for {'shuffle': True, 'penalty': 'l2', 'l1_ratio': 0.1, 'epsilon': 1, 'alpha': 0.001}
        [ 3]: 0.952 (+/-0.085) for {'shuffle': False, 'penalty': 'l2', 'l1_ratio': 0.6, 'epsilon': 0.1, 'alpha': 0.001}
        [ 4]: 0.838 (+/-0.097) for {'shuffle': True, 'penalty': 'l1', 'l1_ratio': 0.9, 'epsilon': 1e-05, 'alpha': 0.1}
        [ 5]: 0.695 (+/-0.047) for {'shuffle': False, 'penalty': 'l2', 'l1_ratio': 0.9, 'epsilon': 0.001, 'alpha': 1}
        [ 6]: 0.752 (+/-0.388) for {'shuffle': True, 'penalty': 'elasticnet', 'l1_ratio': 0, 'epsilon': 0.0001, 'alpha': 0.0001}
        [ 7]: 0.886 (+/-0.177) for {'shuffle': True, 'penalty': 'l2', 'l1_ratio': 0.8, 'epsilon': 1e-05, 'alpha': 0.01}
        [ 8]: 0.724 (+/-0.304) for {'shuffle': True, 'penalty': 'l1', 'l1_ratio': 0, 'epsilon': 1, 'alpha': 1e-05}
        [ 9]: 0.962 (+/-0.071) for {'shuffle': True, 'penalty': 'l1', 'l1_ratio': 0.5, 'epsilon': 0.1, 'alpha': 0.01}
        [10]: 0.695 (+/-0.047) for {'shuffle': False, 'penalty': 'elasticnet', 'l1_ratio': 0.3, 'epsilon': 0.001, 'alpha': 1}
        [11]: 0.695 (+/-0.214) for {'shuffle': True, 'penalty': 'elasticnet', 'l1_ratio': 1, 'epsilon': 0.0001, 'alpha': 1e-05}
        [12]: 0.819 (+/-0.164) for {'shuffle': False, 'penalty': 'elasticnet', 'l1_ratio': 0.5, 'epsilon': 0.0001, 'alpha': 0.1}
        [13]: 0.914 (+/-0.071) for {'shuffle': False, 'penalty': 'l1', 'l1_ratio': 0.7, 'epsilon': 1, 'alpha': 0.0001}
        [14]: 0.810 (+/-0.241) for {'shuffle': False, 'penalty': 'l2', 'l1_ratio': 1, 'epsilon': 0.1, 'alpha': 0.01}
        [15]: 0.810 (+/-0.241) for {'shuffle': False, 'penalty': 'l2', 'l1_ratio': 0.4, 'epsilon': 1e-05, 'alpha': 0.01}
        [16]: 0.952 (+/-0.104) for {'shuffle': False, 'penalty': 'l1', 'l1_ratio': 0.5, 'epsilon': 0.001, 'alpha': 0.001}
        [17]: 0.695 (+/-0.047) for {'shuffle': True, 'penalty': 'l2', 'l1_ratio': 0.2, 'epsilon': 1, 'alpha': 1}
        [18]: 0.848 (+/-0.140) for {'shuffle': True, 'penalty': 'l2', 'l1_ratio': 1, 'epsilon': 1, 'alpha': 0.001}
        [19]: 0.905 (+/-0.085) for {'shuffle': False, 'penalty': 'l2', 'l1_ratio': 1, 'epsilon': 1e-05, 'alpha': 0.0001}

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      0.78      0.88        18
           2       0.73      1.00      0.85        11

    accuracy                           0.91        45
   macro avg       0.91      0.93      0.91        45
weighted avg       0.93      0.91      0.91        45


CTOR for best model: SGDClassifier(alpha=0.01, l1_ratio=0.5, penalty='l1')

best: dat=iris, score=0.96190, model=SGDClassifier(alpha=0.01,epsilon=0.1,l1_ratio=0.5,penalty='l1',shuffle=True)

OK(random-search)
```

Figur 22: Result from randomized search using the SGDClassifier.

2.2.4 Qd - MNIST Search Quest II

In this exercise, we switch to use MNIST data instead of the iris data. This is a much larger dataset and will take longer to process. We started of trying to same grid search method with the same tuning parameters as in exercise Qb. However, after an hour of waiting with a red hot computer, we decided to gradually decrease the number of tuning parameters in hopes of getting a result.

Unfortunately, even a random search with a single iteration and only 5 different parameters in the tuning parameters, we were not given a solution after waiting for an hour. Calculations were definitely being made though, as the computer was struggling to keep up with the workload. Whether we did something wrong or the task was too complex for the pc and timeframe, we're unsure of.

```
# Setup data
X_train, X_test, y_train, y_test = LoadAndSetupData(
    'mnist')  # 'iris', 'moon', or 'mnist'


# Setup search parameters
model = SGDClassifier()


tuning_parameters_SGD = {
    'penalty':('l1','l2'),
    'alpha':[0.0001, 0.001, 0.01],
}


CV = 5
VERBOSE = 0

# Run RandomizedSearchCV for the model
start = time()
random_tuned_SGD = RandomizedSearchCV(model,
                          tuning_parameters_SGD,
                          n_iter=1,
                          cv=CV,
                          scoring='f1_micro',
                          random_state = 42,
                          verbose=VERBOSE,
                          n_jobs=-1,
                          iid=True)
random_tuned_SGD.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(random_tuned_SGD, X_test, y_test, t)
print('OK(random-search)')
```

```
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)
```

Figur 23: Random search attempt on MNIST data without luck. The program was stuck like this for over an hour while being busy calculating.

After changing model to SVC and trying a search space with only two hyperparameters, we were finally able to get a result from the search after a long search-time, as can be seen on figure 24 and 25.

```
# Setup data
X_train, X_test, y_train, y_test = LoadAndSetupData(
    'mnist')  # 'iris', 'moon', or 'mnist'


# Setup search parameters
model = svm.SVC(
    gamma=0.001
)


tuning_parameters_SVC = {
    'kernel':('linear','poly')
}


CV = 5

VERBOSE = 0

# Run RandomizedSearchCV for the model
start = time()
random_tuned_SVC = RandomizedSearchCV(model,
                           tuning_parameters_SVC,
                           n_iter=1,
                           cv=CV,
                           scoring='f1_micro',
                           random_state = 42,
                           verbose=VERBOSE,
                           n_jobs=-1,
                           iid=True)
random_tuned_SVC.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(random_tuned_SVC, X_test, y_test, t)
print('OK(random-search)')
```

Figur 24: Random search on MNIST-data using the SVC-model and a very small search space.

We got an f1-score of 0.976, which is surprisingly good results, at least much higher than a randomly guessing model. The result can be seen on figure 25.

```
Best model set found on train set:

        best parameters={'kernel': 'poly'}
        best 'f1_micro' score=0.9758571428571429
        best index=0

Best estimator CTOR:
        SVC(gamma=0.001, kernel='poly')

Grid scores ('f1_micro') on development set:
        [ 0]: 0.976 (+/-0.002) for {'kernel': 'poly'}

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       0.98      0.99      0.99      2077
           1       0.98      0.99      0.99      2385
           2       0.97      0.98      0.97      2115
           3       0.98      0.97      0.97      2117
           4       0.97      0.98      0.98      2004
           5       0.97      0.97      0.97      1900
           6       0.98      0.98      0.98      2045
           7       0.98      0.97      0.98      2189
           8       0.97      0.96      0.97      2042
           9       0.97      0.96      0.97      2126

    accuracy                           0.98     21000
   macro avg       0.98      0.98      0.98     21000
weighted avg       0.98      0.98      0.98     21000


CTOR for best model: SVC(gamma=0.001, kernel='poly')

best: dat=mnist, score=0.97586, model=SVC(kernel='poly')
```

Figur 25: Result from random search on MNIST-data using the SVC-model and a very small search space.