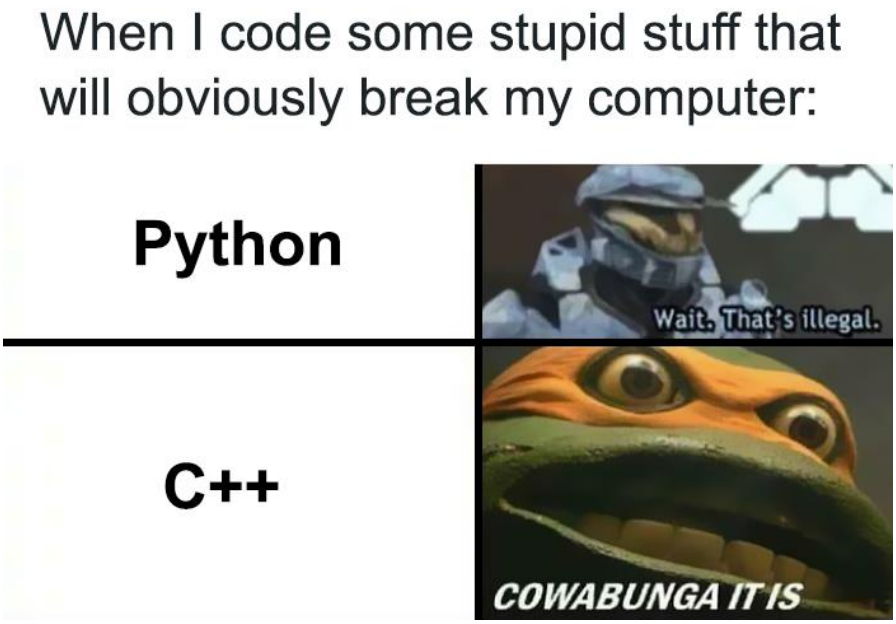
O2

SWMAL-01 Group 13



Group members
Christoffer Regnar Mølck - 202009347
Martin Teit Bruun Andersen - 202109592
Morten Kierkegaard Erichsen - 201807571

12. marts 2024

1 Resume Chapter 2: End-to-End

1.0.1 Look at the Big Picture

The sub-chapter starts with an introduction to the working life of a data science person. Here the book assigns the reader with the task to build a housing price model of Californian homes. The reader is then introduced to the Machine Learning (from now on mentioned as "ML") check list and how to go about it. Should the reader not want to read the sub-chapter it is mentioned a ML check list can be found in appendix B.

First step on the check list is getting the whole picture as the sub-chapter is called.
- Should the model be linked to something else?
- How does the outcome of the model come to use later on?

Afterwards the sub-chapter introduces the reader to Pipepline. Here it is explained that a pipeline is a data processing component, where different components are being handled to be processed by another component. Second step is to frame the problem.
- Which learning type to use.
- Which task should one use? (regression vs. classification)

The sub-chapters assignment calls for a supervised regression model to find the median housing price. The third step focuses on finding the models performance measure. Here the reader is introduced to:
- RMSE (Root Mean Square Error)
- MAE (Mean Absolute Error)

1.0.2 Get the Data

The sub-chapter starts by encouraging the reader to look up the code examples on the github page for the book. Henceforth, it is explained how to setup the needed Jupyter envoriorment using Anaconda as the choosen IDE.

Later on in the sub-chapters section "Download the Data"it is shown how the reader might fetch the CSV file that contains the housing data needed to create and start the model introduced in previous sub-chapter "Look at the Big Picture". The reader is also introduced to the loading of set Housing data and what the reader might discover using modules such as:
- .head()
- .intro()
- .describe()

For example is it mentioned how a entry within the Housing dataset "Total_bedrooms"only contains 20,433 entities, which leaves out 207 entities without any information regarding the

amount of bedrooms the house contains. This leaves the reader with a better understanding of the data.

The sub-chapter ends asking the reader to create a set set and not to look at it.

1.0.3 Explore and Visualize the Data to Gain Insights

In the beginning the sub-chapter suggests to make a copy of the training set, so the original set can be left untouched while the copied set can be messed around with.

Later on scatter plots of the training data is shown and how the reader with the use of the "alpha"attribute large and dense located entries easier can be located. Previous mentioned in earlier sub-chapters, the textbook dataset is a rather small dataset for training a ML model. However, the size of the Housing dataset makes it easier looking for correlations. Later on the sub-chapter looks into the correlation in respect to "median_house_value". Here it shows that there trends to be a slight trend/correlation between "median_house_value"and "median_house_income". This trend is then shown using scatter plots again (where the "alpha"attribute is used once more) for a better understanding of the data and its correlation.

1.0.4 Prepare the Data for Machine Learning Algorithms

The sub-chapter focuses on the implementation of functions to clean the data for multiple reasons such as:
- The option to reproduce the transformations for future projects.
- Creating a library with many different types of transformations.

Here the sub-chapter also informs the reader how to handle datasets with missing values such as:
- Deleting/Removing the missing entries from the dataset.
- Completely removing the attribute from the model.
- Providing a prefix value where they are missing.
- The values being either zero, mean etc..

The sub-sub-chapter then describes what to do if the third option was chosen. One of the transformers introduced is scikit learns "OrdinalEncoder". This transformer turns strings into categories since ML models prefer numerical input. However, categories can be tricky so the tect also introduces the reader to transformer "OneHot". OneHot takes one attribute that is "Hot"(Binary == 1), while leaving the rest as zero's, "cold"(Binary == 0). One of the "nedd to"transformers to add is feature scaling. The text introduces

two ways to feature scale:
1. The Min-Max scale (otherwise known as normalization)
- Values are shifted and rescaled so they instead range from 0 to 1. This is done by taking the minimum value in set dataset, the maximum value and a target value. These values are then put to use the following way: $X = \frac{Target-Min}{Max-Min}$.

2. Standardization
- $X = \frac{Target-Mean}{Std.Variance}$
The standardization differs from the Min_Max not only but outcome but is also much less affected by "hick-ups"in the dataset. For example should the reader have a dataset where values should be 0-15, but a 100 (a value) sneaked its way into the dataset. The Min_Max would then completely crush the transformation, where standardization would not be affected as much.

Last in the sub-chapter the reader is introduced to scikit's transformations pipeline class and its sequencing functionality.

1.0.5 Select and Train a Model

The introduction to the sub-chapter congratulates the reader on (should he/she/them) having followed along and tackled the questions/problems along the way. The reader should by now have created a traning set, test set and transformation pipelines to clean up the data. The model is then ready and with the introduction of a LinearRegressor and RMSE it is shown how poorly the model is performing due to the LinearRegressor under-fitting the data.

Since the LinearRegressor performed badly the reader is introduced to another model type: DecisionTreeRegressor. However this model type ends up over-fitting the data instead. The last model introduced in the sub-chapter is the RandomForestRegressor (RFR). This regressor works by training many Decision Trees on subset of the features and then take the average of their predictions. These types of models are called Ensemble Learning. The RFR out performs the models, but is still over-fitting the model. Henceforth, the reader is advised to shortlist 2-5 models and not spend to much time tweaking the hyper-parameters of each model.

1.0.6 Fine-Tune Your Model

When a shortlist is created it is time to tune the models selected. The reader is then told that this can be done manually. However, this is a tedious task. From there it is advised to use SciKits GridSearchCV (GScv). This search method takes in the hyperparameters the

reader wants to optimize. The SGcv uses Cross-Validation to optimize the hyperparameters. An example is given where the GScv is used on RFR. Here the reader is advised not to think too hard about what parameters RFR takes since it is introduced in chapter 7.

The example shows how the GScv is used and how it works. For the sub-chapter example the GScv goes through 90 iterations of training before deciding on what hyperparameter performed best. Afterwards another search method is introduced called RandomSearch (RS). The reader is told to use this method instead of GScv, when the amount of hyperparameters tuned is "too many". RS has the benefits of xxxx iterations of testing using a random numerical value instead of a prefixed number.

1.0.7 Launch, Monitor, and Maintain Your System

The sub-chapter informs the reader that it is important to check the code's live performance. This is important since the model might worsen over time, due to the input data changing because measurements, that did not play a factor at the time of the models creation. For housing prices this could be hyperinflation fx. The systems performance requires a human analysis. This can be done through Crowd-platforms such as: Amazon Mechanical Turk or CrowdFlower. At last the reader is advised to, on a regular basis train its model on a regular basis with fresh data, and that it would be a good idea to automate the process.

1.0.8 Try It Out!

This sub-chapter sums up the whole chapter. And tells the reader that emphasizing the importance of data preparation, monitoring tools, human evaluation pipelines, and regular model training are go-to when creating a model/project. While algorithms are crucial, mastering the overall process and a few key algorithms is more beneficial than focusing solely on advanced ones. The reader is then advised to start their own journey by selecting a dataset and working through the process from start to finish. Here a website like Kaggle.com is pointed out due to its great amount of datasets, clear goals, and a communities join on the readers journey.

2 Data analysis of O4 dataset.

2.1 Data Introduction and purpose

2.1.1 A short introduction to our O4 project

Our idea is to use data from kaggle, specifically a bank churn dataset. The idea is to train a neural net on the data to predict income based on a number of variables.

2.1.2 The data:

That data as comes from here: Bank Churn. And includes 14 features and about 275k rows. The data includes the following columns:

Id, CustomerId, Surname, CreditScore, Geography, Gender, Age, Tenure, Balance, NumOfProducts, HasCrCard, IsActiveMember, EstimatedSalary, Exited.

This includes a mixture of string and numeric data. The Geography is unfortunately limited to Germany, France and Spain. Ideally we would have wanted data for the whole world, but we were unable to find such, possible due to laws as the data could be considered sensitive.

The original point of the dataset is to describe the churn of bank accounts. Which is another word for leaving the account like going to another bank etc. We will however use it to predict salary instead.

The data is already quite clean, which will be shown more in the next section as we dive deeper into data analysis.

Ultimately what we want to build is a feed forward regressor that predicts the income for any bank account given the CreditScore, Geograpghy, Gender, Age, Balance, NumOfProducts and HasCrCard. This could possibly give us some ideas of the correlation between say, age, gender and Geography in relations to income.

Another possibility we could possibly investigate is XGboost aka Boosted trees, as the training time for this type of regressor is almost instant. It could be interesting to see the performance of a traditional ffnn vs Xgboost.

2.2 Data analysis

The data comes presplit into train and test data, we dont want that so the first thing i did was to combine it.

```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

#combine train and test data to one dataframe
data = pd.concat([train, test], ignore_index=True)
```

There are several columns we want to drop as they are not relevant, This includes:

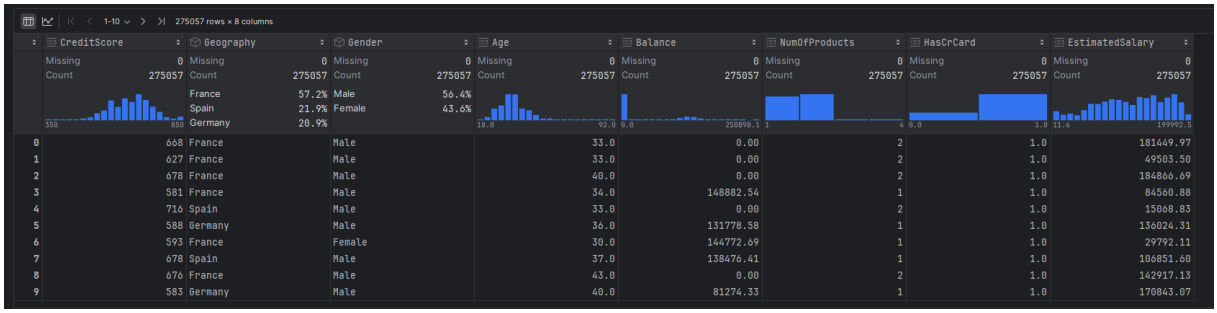
- id
- CustomerId
- Surname
- Tenure
- isActiveMember
- Exited

these are not relevant as they add nothing of meaningful value to the training data and would only act as noise. of cause, this is only our opinion and hidden correlations could be possible. But it is reasonable to assume that these columns have no correlation with the data we want to predict on.

so we drop it.

```
data = data.drop(['id', 'CustomerId', 'Surname', 'Tenure', 'IsActiveMember',
    ↪ 'Exited'], axis=1)
```

our data now looks like this:

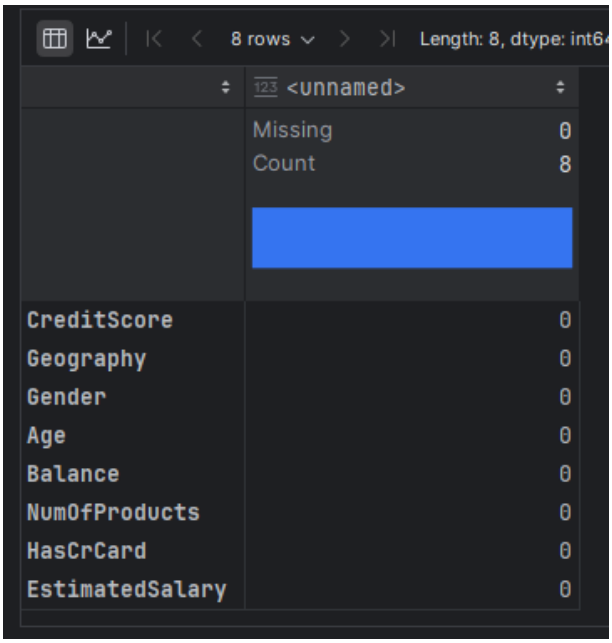


	CreditScore	Geography	Gender	Age	Balance	NumOfProducts	HasCrCard	EstimatedSalary
Missing	0	0	0	0	0	0	0	0
Count	275057	275057	275057	275057	275057	275057	275057	275057
		France 57.2%	Male 56.4%					
		Spain 21.9%	Female 43.6%					
		Germany 20.9%						
0	668	France	Male	33.0	0.00	2	1.0	181449.97
1	627	France	Male	33.0	0.00	2	1.0	49503.50
2	678	France	Male	40.0	0.00	2	1.0	184866.69
3	581	France	Male	34.0	148882.54	1	1.0	84560.88
4	716	Spain	Male	33.0	0.00	2	1.0	15068.83
5	588	Germany	Male	36.0	131778.58	1	1.0	136024.31
6	593	France	Female	30.0	144772.69	1	1.0	29792.11
7	678	Spain	Male	37.0	138476.41	1	1.0	106851.60
8	676	France	Male	43.0	0.00	2	1.0	142917.13
9	583	Germany	Male	40.0	81274.33	1	1.0	170843.07

I do a quick check to see if there is any data that is null.

```
1 data.isnull().sum()
```

output:



	<unnamed>
Missing	0
Count	8
CreditScore	0
Geography	0
Gender	0
Age	0
Balance	0
NumOfProducts	0
HasCrCard	0
EstimatedSalary	0

there are 0 places where our data is null, the data seems quite clean. But lets continue to check.

Next i check how many unique values there are, to see if there are any errors in the binary columns like gender and HasCrCard. Its reasonable to assume that an error could have been made somewhere. Like Male could have been mispelled and so on.

```
for column in data.columns:
    print(column, data[column].nunique())
```

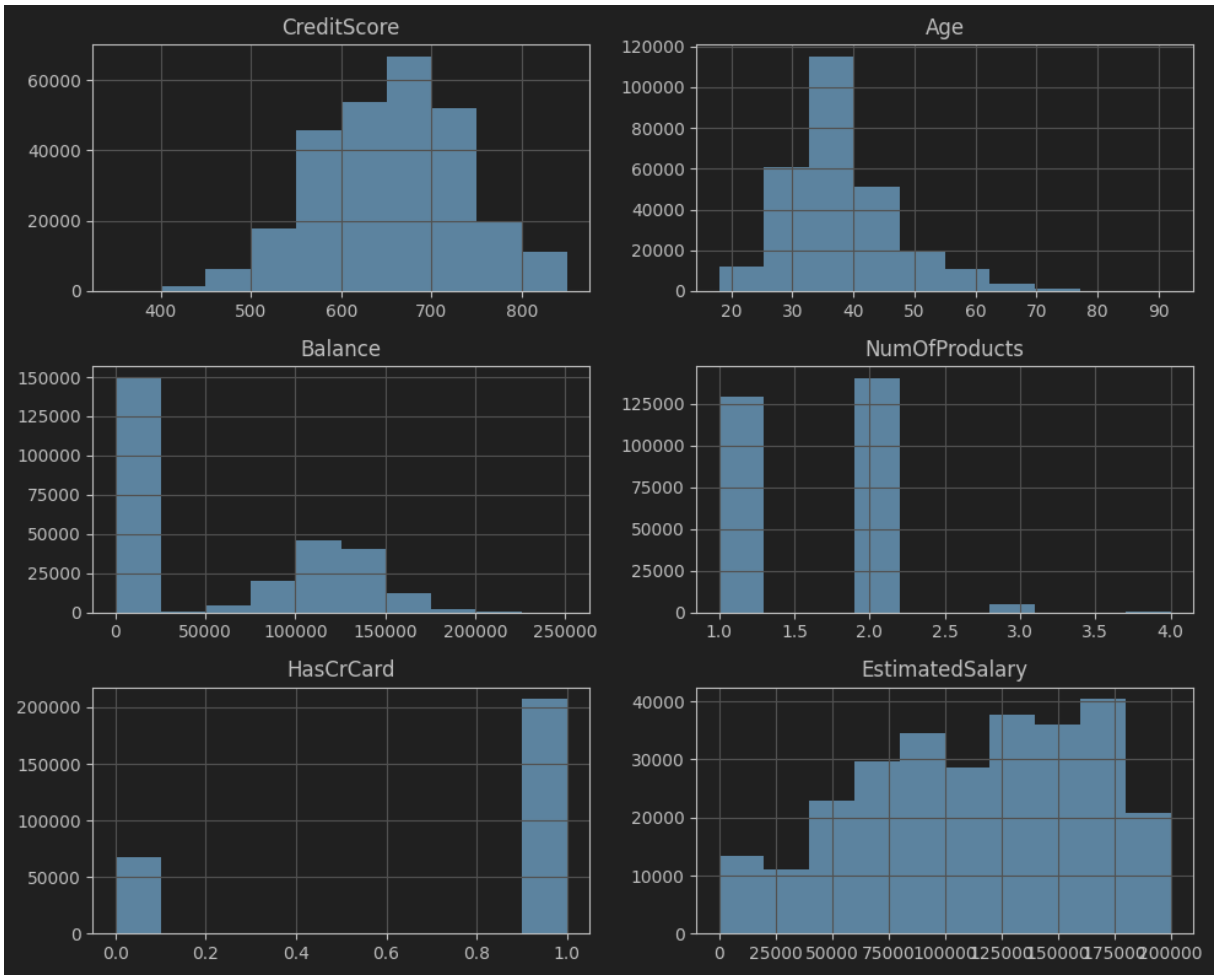
```
CreditScore 458
Geography 3
Gender 2
Age 74
Balance 43875
NumOfProducts 4
HasCrCard 2
EstimatedSalary 79868
```

This is where i discovered that there are only 3 countries represented in the data. Unfortunately. Otherwise everything else looks good.

Now i wanted some histograms, scatter plots and boxplots to show the data.

```
data.hist(figsize=(10, 8))
plt.tight_layout()
plt.show()

# Create scatter plots
plt.figure(figsize=(10, 8))
plt.subplot(2, 2, 1)
plt.scatter(data['Age'], data['Balance'])
plt.xlabel('Age')
plt.ylabel('Balance')

plt.subplot(2, 2, 2)
plt.scatter(data['Age'], data['EstimatedSalary'])
plt.xlabel('Age')
plt.ylabel('Estimated Salary')

plt.subplot(2, 2, 3)
plt.scatter(data['Balance'], data['EstimatedSalary'])
plt.xlabel('Balance')
plt.ylabel('Estimated Salary')

plt.tight_layout()
plt.show()
```

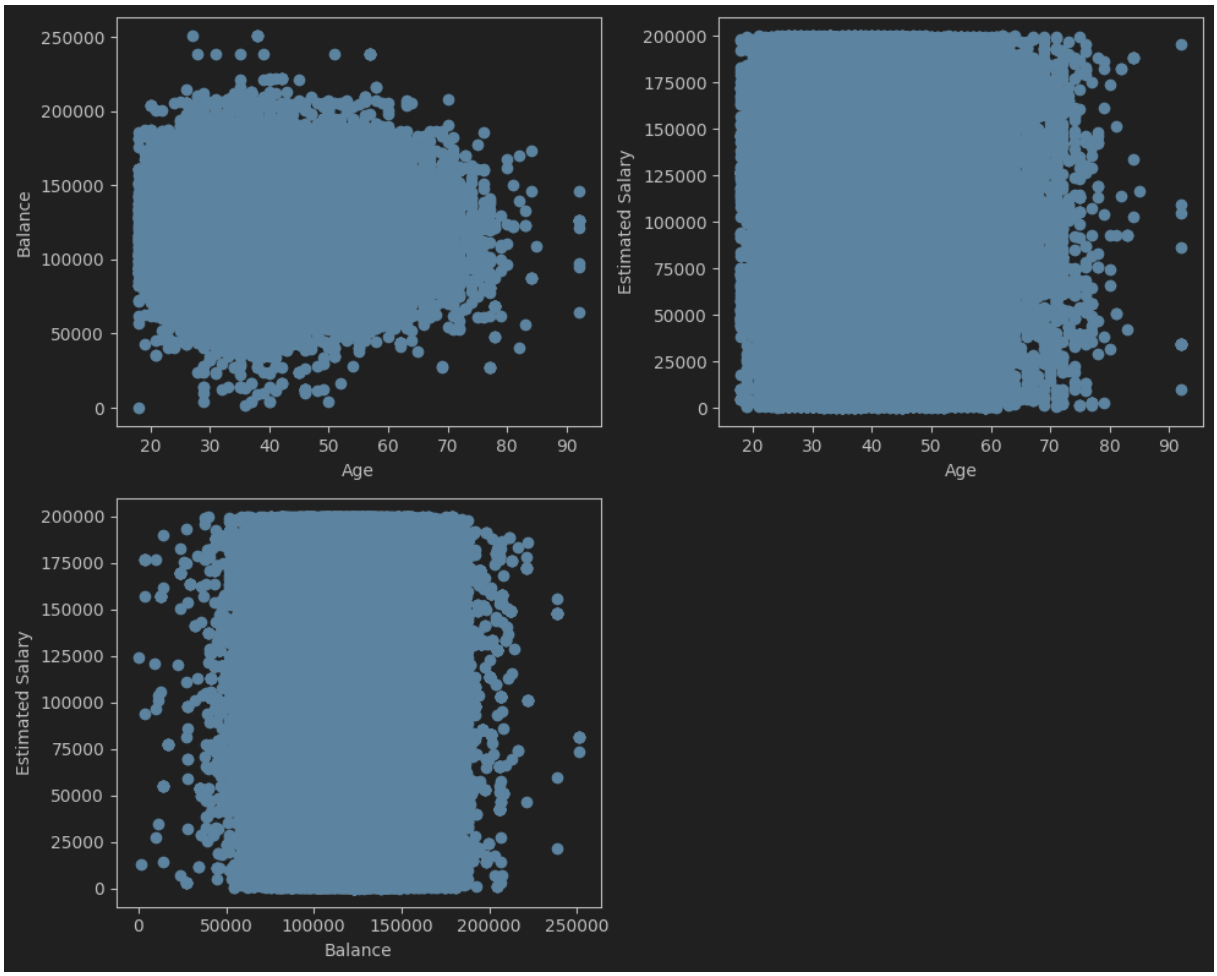
Output:



Hmm, theres something strange going on with the Balance, there are a lot of accounts that are 0. This is quite bad, as it would heavily scew our model.

I decided to get rid of all rows where balance is zero. This reduces our data amount significantly, down to 125k, but i think its the right thing to do in this case.

After removal our scatter plots end up looking like this:



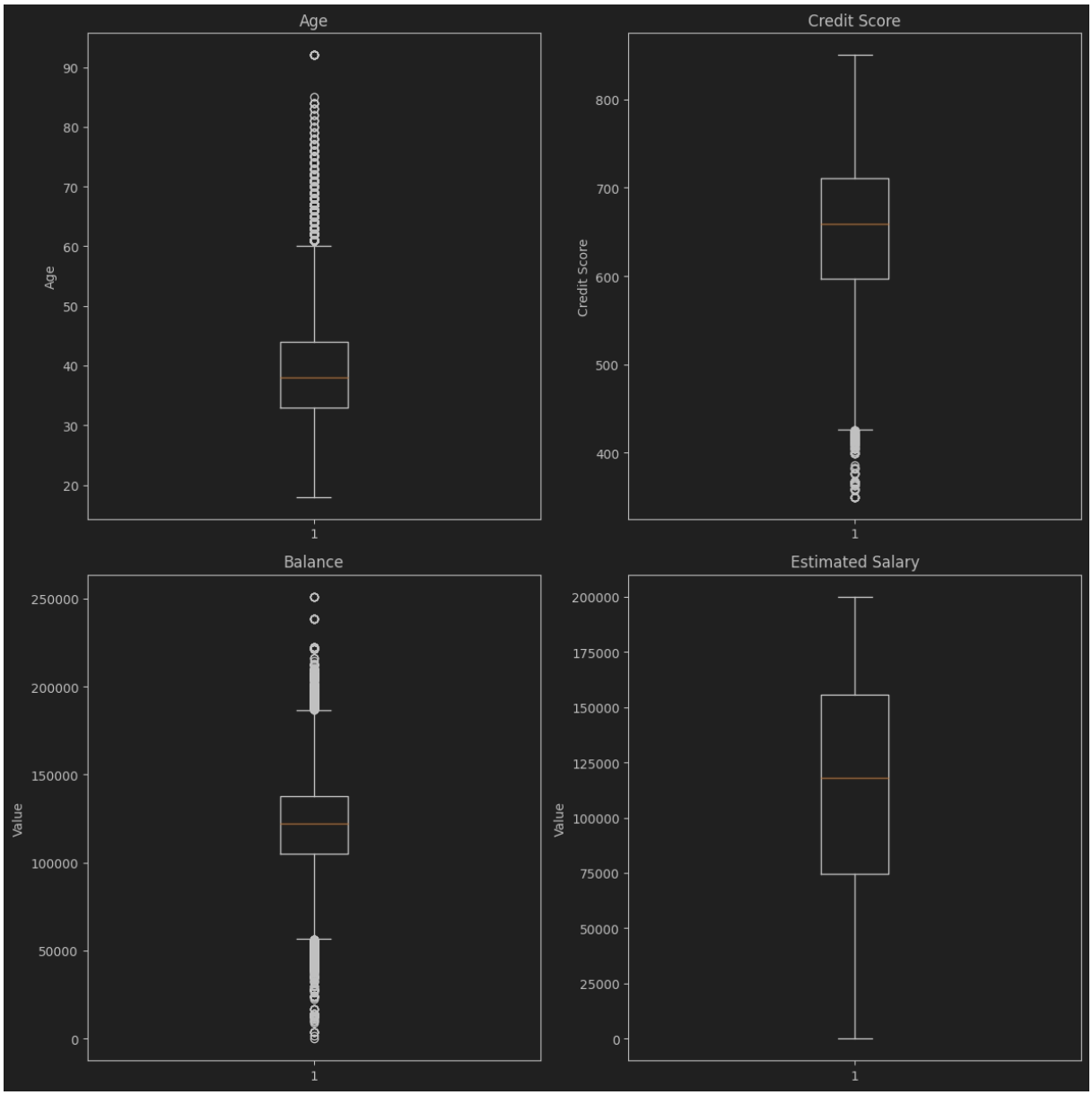
This looks quite good.

Its obvious that there are some outliers in the data from the scatter plots, lets visualzie them using boxplots.

```
plt.subplot(2, 2, 1)
plt.boxplot(data['Age'])
plt.title('Age')
plt.ylabel('Age')

plt.subplot(2, 2, 2)
plt.boxplot(data['CreditScore'])
plt.title('Credit Score')
plt.ylabel('Credit Score')

plt.subplot(2, 2, 3)
plt.boxplot(data['Balance'])
plt.title('Balance')
```

```
plt.ylabel('Value')

plt.subplot(2, 2, 4)
plt.boxplot(data['EstimatedSalary'])
plt.title('Estimated Salary')
plt.ylabel('Value')
```

output:



This makes it quite clear that we have a lot of outliers in both age, credit score and balance. Our data is also quite heavily focused around certain areas like the age group 35-45. This should be ok, but we will have to use a scalar that accounts for this. I have previously had

great success with sklearns robust scalar that is build to be robust against outliers. This ensures that we capture the data from the outliers and that it does not get ignored during training.

This concludes the data analysis for now, as our date isnt horribly complicated, and already quite clean, not much needed to be done with the data to get good results. How well the NN trains on the data however, is always hard to predict.

3 Pipelines

Opgaverne:

3.0.1 Qa) Create a Min/max scaler for the MLP

Manually scaling the data would mean dividing the X values given, by the largest value in X. This would give a range from [0;1], with the lower boundary 0 not being completely representative of the lowest value in x.

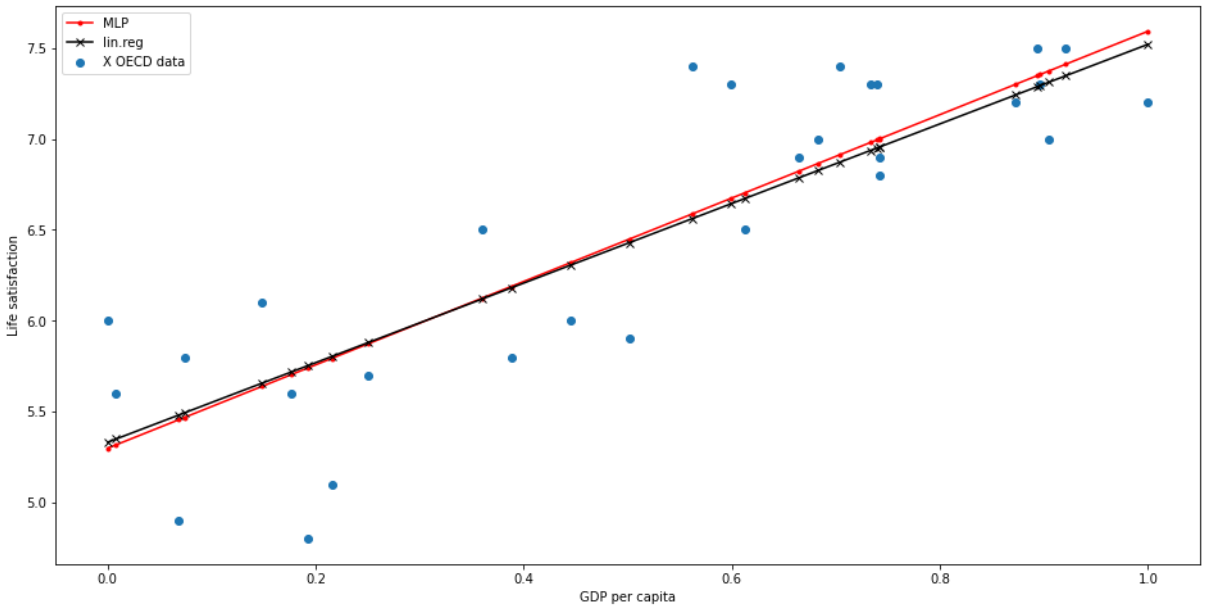
Thats what the MinMaxScaler from sklearn.preprocessing does, which is what is presented below. When using that scaler, the smallest value in X becomes 0, and the largest 1, in [0;1].

Code:

```
    from sklearn.preprocessing import MinMaxScaler, StandardScaler
import numpy as np

# X transformation
min_max_scaler = MinMaxScaler([0,1])
X_new = min_max_scaler.fit_transform(X)

mlp = MLPRegressor( hidden_layer_sizes=(10,), solver='adam', activation='relu',
    ↪ tol=1E-5, max_iter=100000, verbose=True)
mlp.fit(X_new, y.ravel())

# create an test matrix M, with the same dimensionality as X, and in the range
    ↪ [0;1]
# and a step size of your choice
m=np.linspace(0, 1, 1000)
M=np.empty([m.shape[0],1])
M[:,0]=m

# Prediction
pred = [[22000]]
mlp_pred = mlp.predict(pred)

# New linear regression
linregLMMS = LinearRegression()

linregLMMS.fit(X_new, y)

PlotModels(linregLMMS, mlp, X_new, y, "MLP","lin.reg")
```

Output:

```
    Iteration 1031, loss = 0.09114793
Training loss did not improve more than tol=0.000010 for 10 consecutive epochs.
    ↪ Stopping.
    MLP.score(X, y)=0.73
lin.reg.score(X, y)=0.73
```



Figur 3.1: MLP and LinReg with scaling

3.0.2 Qb) Scikit-learn Pipelines

Since the MinMaxScaler already has been used in Qa, the pipelines functionality was used immediately.

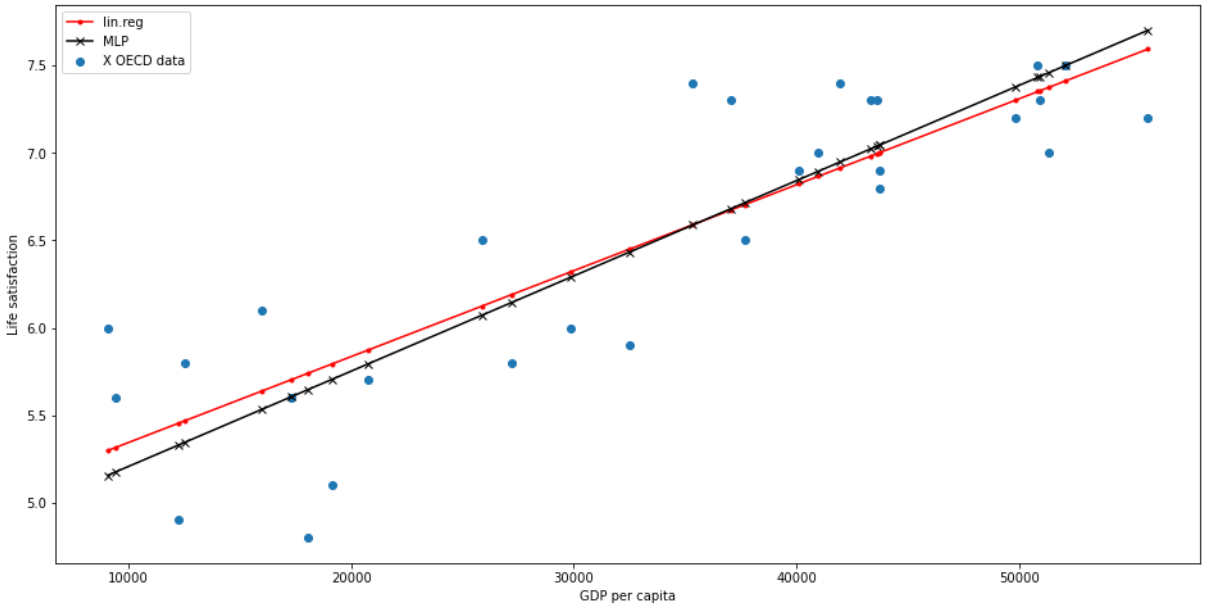
Code:

```

# MinMaxScaler imported earlier

from sklearn.pipeline import Pipeline

pipeLIN = Pipeline([
    ('scaler',MinMaxScaler()),
    ('linreg', LinearRegression()),
])


mlp = MLPRegressor( hidden_layer_sizes=(10,), solver='adam', activation='relu',
    ↪ tol=1E-5, max_iter=100000, verbose=True)

pipeMLP = Pipeline([
    ('scaler',MinMaxScaler()),
    ('linreg', mlp),
])

pipeLIN.fit(X,y)
pipeMLP.fit(X,y)


PlotModels(pipeLIN, pipeMLP, X, y, "lin.reg", "MLP")
```

Output:

The output is, not surprisingly, the same as in Qa, since the same scaling is used, but the pipelines functionality makes it easy to keep an overlook of what goes into a regressor. Linear and MLP in this case.

```
Iteration 2186, loss = 0.09341684
Training loss did not improve more than tol=0.000010 for 10 consecutive epochs.
    ↪ Stopping.
lin.reg.score(X, y)=0.73
    MLP.score(X, y)=0.73
```



Figur 3.2: MLP and LinReg with MinMaxScaling using pipelines

3.0.3 Qc) Outliers and the Min-max Scaler vs. the Standard Scaler

(Written BEFORE making a standardscaler)

It might, since it removes the mean, and only scales the data to its variance. This means that after the scaling, the empirical mean and deviation wont represent the "true"mean and deviation. This might make it unable to balance its scales correctly, or it could make it more accurate within the range.

(Written AFTER making a standardscaler)

It seems as if the MLP model with the standardscaler does handle the outliers in the data well, since it enables the MLP to more accurately fit the data (reflected in the higher R^2 score). Since the StandardScaler scales to unit variance, we see that the fit becomes non-linear in the lower GDP section, since there is a large "clump"of data with relative higher variance that the "clump"the in higher GDP range.

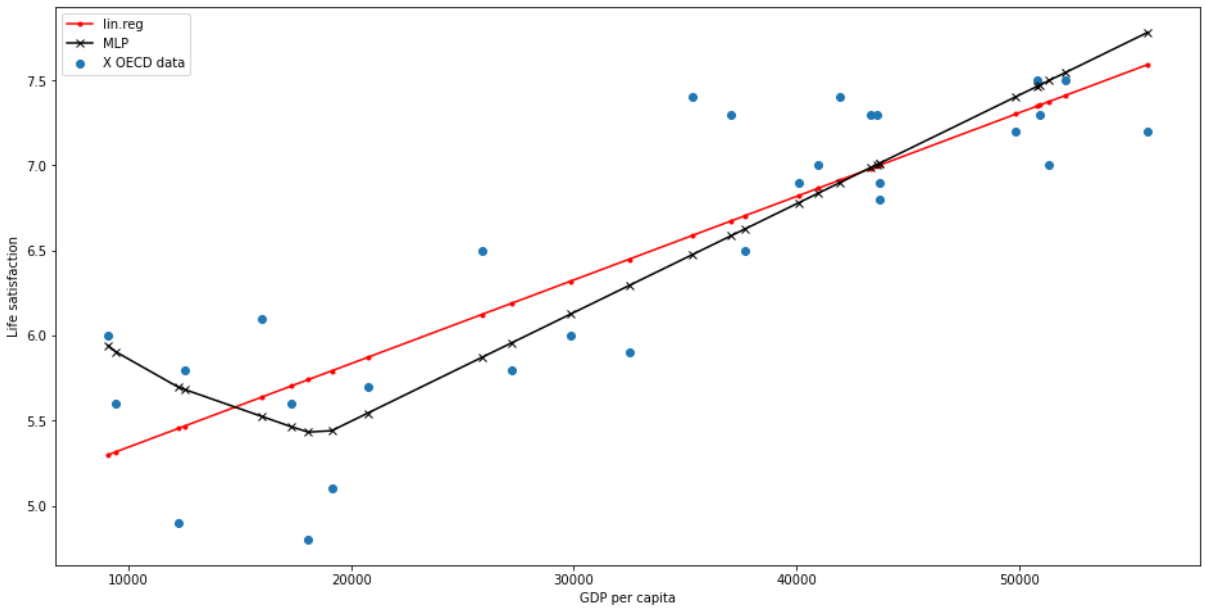
Code:

```
pipeLINSS = Pipeline([
('scaler',StandardScaler()),
('linreg', LinearRegression()),
])

```

```
6
7  mlpSS = MLPRegressor( hidden_layer_sizes=(10,), solver='adam',
       ↪ activation='relu', tol=1E-5, max_iter=100000, verbose=True)
8
9  pipeMLPSS = Pipeline([
10     ('scaler',StandardScaler()),
11     ('linreg', mlpSS),
12 ])
13
14 pipeLINSS.fit(X,y)
15 pipeMLPSS.fit(X,y)
16
17
18 PlotModels(pipeLINSS, pipeMLPSS, X, y, "lin.reg", "MLP")
```

Output:

```
1  Iteration 2801, loss = 0.08282683
2  Training loss did not improve more than tol=0.000010 for 10 consecutive epochs.
       ↪ Stopping.
3  lin.reg.score(X, y)=0.73
4  MLP.score(X, y)=0.76
```



Figur 3.3: MLP and LinReg with standardscaler

3.0.4 Qd) Modify the MLP Hyperparameters

Firstly, the MLPRegressor, with early-stopping = false, has been tested for a number of neurons, and the R^2 has been plotted as a function of neurons.

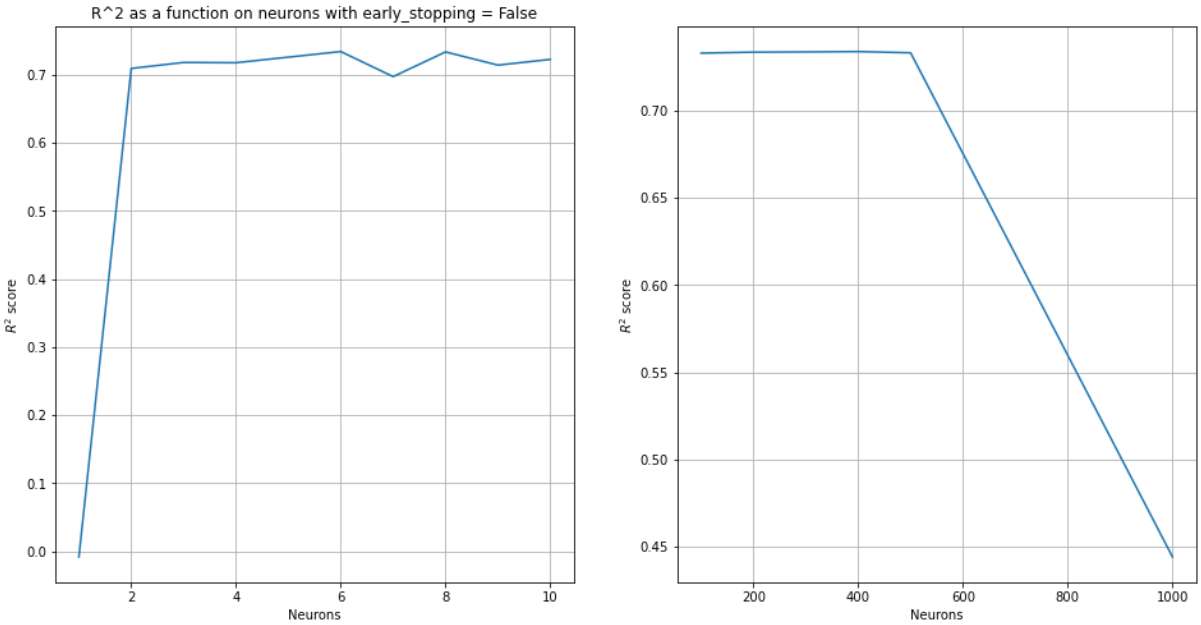
Code:

```
# Stolen function
def CalcPredAndScore(model, X: np.ndarray, y: np.ndarray,):
        assert isNumpyData(X, 2) and isNumpyData(y, 1) and X.shape[0]==y.shape[0]
        y_pred_model = model.predict(X)
        score_model = r2_score(y, y_pred_model) # call r2
        return score_model

Neurons = [1,2,3,4,5,6,7,8,9,10,100,200,300,400,500,1000]

MLPScores = []

for Neuron in Neurons:
    mlp = MLPRegressor( hidden_layer_sizes=(Neuron,), solver='adam',
        ↪ activation='relu', tol=1E-5, max_iter=100000, verbose=False,
        ↪ early_stopping=False)

    pipeMLP = Pipeline([
        ('scaler', MinMaxScaler()),
        ('linreg', mlp),
    ])


    pipeMLP.fit(X,y)

    MLPScores.append(CalcPredAndScore(pipeMLP, X, y))


plt.figure(figsize=(16,8))


plt.subplot(1,2,1)
plt.plot(Neurons[:10], MLPScores[:10])
plt.title("R^2 as a function on neurons with early_stopping = False")
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
plt.grid()
plt.subplot(1,2,2)
plt.plot(Neurons[10:], MLPScores[10:])
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
```

```
39 plt.grid()
40 np.max(MLPScores)
```

Output:

```
1 Max R^2 = 0.733
```



Figur 3.4: MLP with different values for number of neurons

Which shows that already at 2 neurons the model has almost reached its R^2 "potential".

The same model, but with early-stopping = true. Early stopping means that the MLP stops training when the validation score is not improving. From the documentation:

"Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside validation-fraction of training data as validation and terminate training when validation score is not improving by at least tol for n-iter-no-change consecutive epochs. Only effective when solver='sgd' or 'adam'."

Which makes good sense. Theres no reason to continue training, unless the score is improving. Here, validation-fraction and n-iter-no-change are set to their default values, 0.1 and 10.

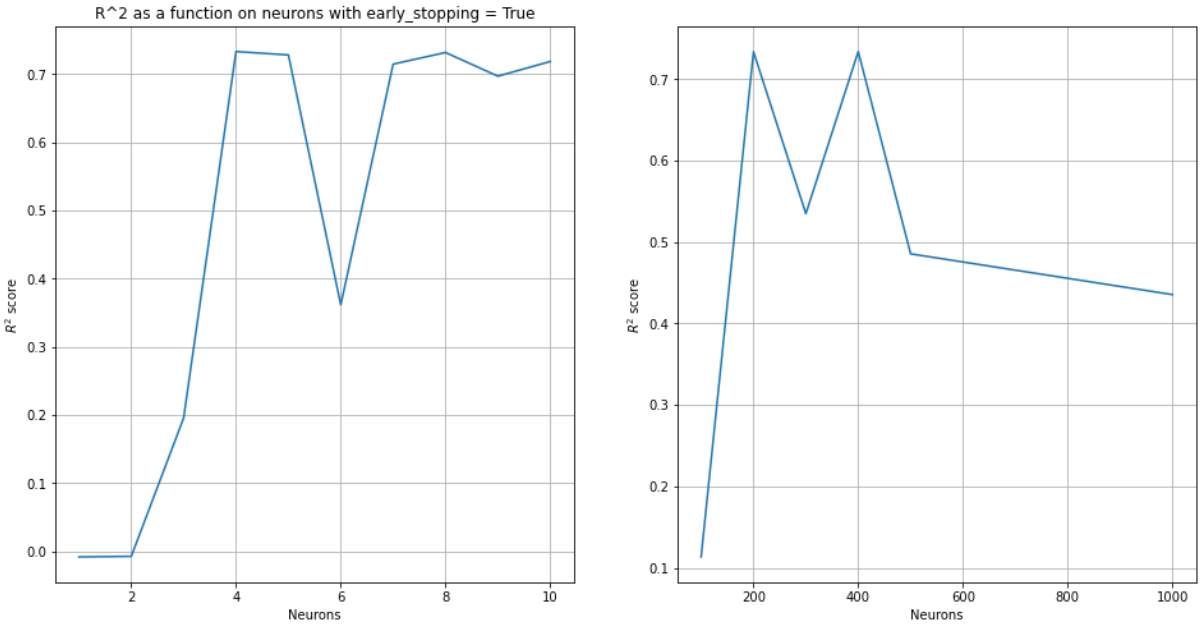
Code:

```
Neurons = [1,2,3,4,5,6,7,8,9,10,100,200,300,400,500,1000]

MLPScores = []
BVS = []

for Neuron in Neurons:
    mlp = MLPRegressor( hidden_layer_sizes=(Neuron,), solver='adam',
        ↪ activation='relu', tol=1E-5, max_iter=100000, verbose=False,
        ↪ early_stopping=True)

    pipeMLP = Pipeline([
        ('scaler', MinMaxScaler()),
        ('linreg', mlp),
    ])


    pipeMLP.fit(X,y)

    MLPScores.append(CalcPredAndScore(pipeMLP, X, y))


plt.figure(figsize=(16,8))


plt.subplot(1,2,1)
plt.plot(Neurons[:10], MLPScores[:10])
plt.title("R^2 as a function on neurons with early_stopping = True")
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
plt.grid()
plt.subplot(1,2,2)
plt.plot(Neurons[10:], MLPScores[10:])
plt.xlabel("Neurons")
```

```
32 plt.ylabel("$R^2$ score")
33 plt.grid()
34 np.max(MLPScores)
```

Output:

```
1 Max R^2 = 0.734
```



Figur 3.5: MLP with early stopping = true

Here, the effect of early-stopping is clear, since the graph is more jagged. For example, with 6 neurons, the early-stopping = False reached an $R^2 \approx 0\ 7$, whereas with early-stopping = True, there was a period of 10 iterations where the score did not improve by tol, resulting in an $R^2 \approx 0\ 38$.

However, already at 4 neurons the R^2 value reached close to its maximum value, so running "adam"with early-stopping = True, n-iter-no-change and tol has to be carefully adjusted.

The same exact scenario has also been tested with the sigmoid function.

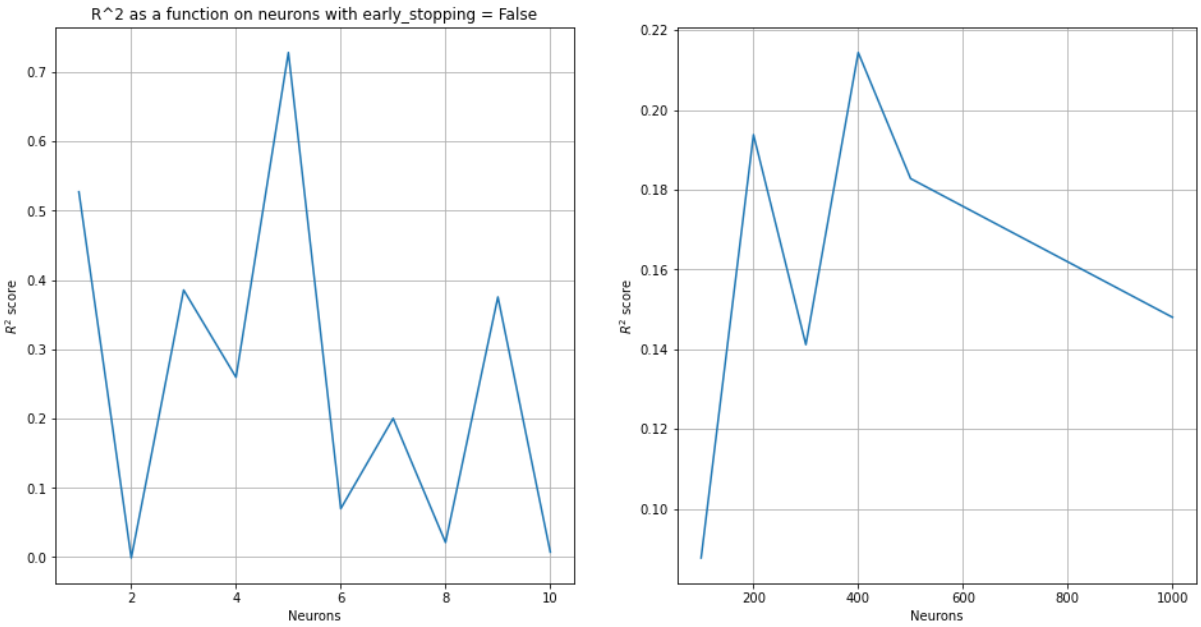
Code:

```
Neurons = [1,2,3,4,5,6,7,8,9,10,100,200,300,400,500,1000]

MLPScoresSGD = []

for Neuron in Neurons:
    mlp = MLPRegressor( hidden_layer_sizes=(Neuron,), solver='sgd',
        ↪ activation='relu', tol=1E-5, max_iter=100000, verbose=False,
        ↪ early_stopping=False)

    pipeMLP = Pipeline([
        ('scaler', MinMaxScaler()),
        ('linreg', mlp),
    ])


    pipeMLP.fit(X,y)

    MLPScoresSGD.append(CalcPredAndScore(pipeMLP, X, y))

plt.figure(figsize=(16,8))


plt.subplot(1,2,1)
plt.plot(Neurons[:10], MLPScoresSGD[:10])
plt.title("R^2 as a function on neurons with early_stopping = False")
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
plt.grid()
plt.subplot(1,2,2)
plt.plot(Neurons[10:], MLPScoresSGD[10:])
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
plt.grid()
np.max(MLPScores)

np.max(MLPScoresSGD)

Neurons = [1,2,3,4,5,6,7,8,9,10,100,200,300,400,500,1000]

MLPScoresSGD = []

for Neuron in Neurons:
    mlp = MLPRegressor( hidden_layer_sizes=(Neuron,), solver='sgd',
        ↪ activation='relu', tol=1E-5, max_iter=100000, verbose=False,
```

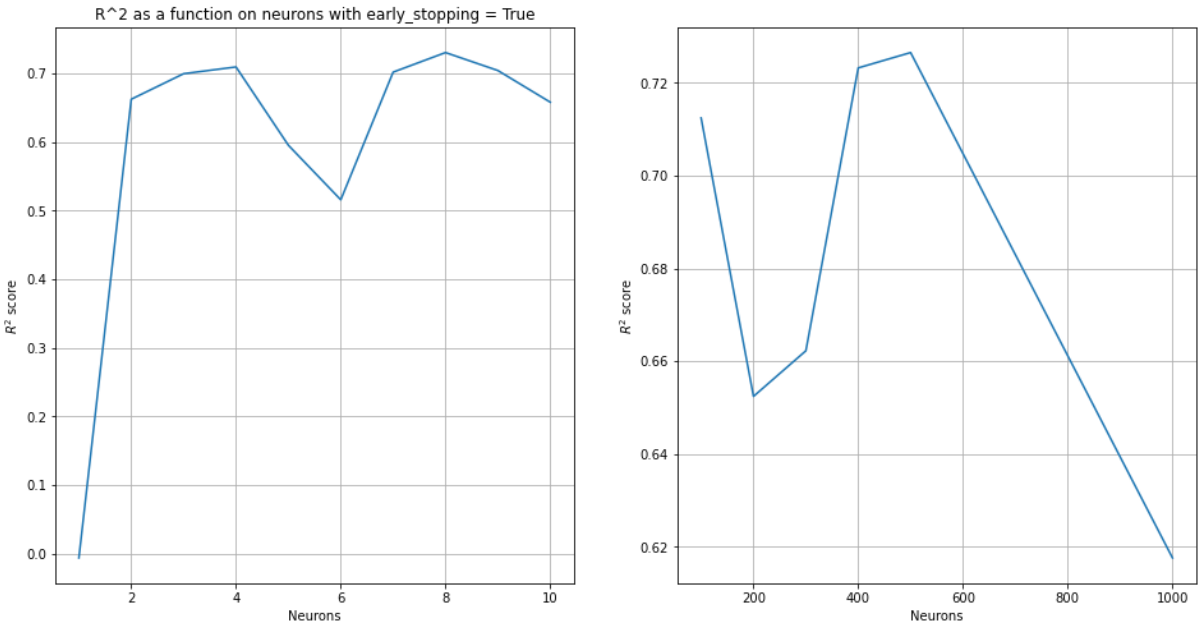
```
        ↪ early_stopping=True)

    pipeMLP = Pipeline([
        ('scaler', MinMaxScaler()),
        ('linreg', mlp),
    ])


    pipeMLP.fit(X,y)

    MLPScoresSGD.append(CalcPredAndScore(pipeMLP, X, y))

plt.figure(figsize=(16,8))


plt.subplot(1,2,1)
plt.plot(Neurons[:10], MLPScoresSGD[:10])
plt.title("R^2 as a function on neurons with early_stopping = True")
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
plt.grid()
plt.subplot(1,2,2)
plt.plot(Neurons[10:], MLPScoresSGD[10:])
plt.xlabel("Neurons")
plt.ylabel("$R^2$ score")
plt.grid()
np.max(MLPScores)

np.max(MLPScoresSGD)
```

Output:

```
Max R^2 False early stopping = 0.728
Max R^2 True early stopping = 0.730
```



Figur 3.6: MLP With sigmoid function and early stopping = False



Figur 3.7: MLP With sigmoid function and early stopping = True

With the "sgd"(Stochastic Gradient Descent), we see that it benefits more from early-stopping than "adam". This might be because "adam"can automatically adjust the amount to update the parameters, based on adaptive estimates of lower-order moments Source for description.

This adaptive behaviour of "adam"may also be the reason the runtime of the "adam"tests

took 15-25 seconds to run, whereas the "sgd"only took 1.7-3 seconds to run. Quite useful to know.

4 L05 Our own Linear Regressor

Our Linear Regressor shown below is created by Chat-GPT and reviewed by us for the given assignment. Changes to this code will later be introduced, as the journal is written. The promt given to Chat-GPT for the creation of our regressor was the following:
——— PROMT begin ———
(In the following you need not present your journal in the Qa+b+c+ etc. order. You could just present the final code with test and comments.)

Training Your Own Linear Regressesor

Create a linear regressor, with a Scikit-learn compatible fit-predict interface. You should implement every detail of the linear regressor in Python, using whatever library you want (except a linear regressor itself).

You must investigate and describe all major details for a linear regressor, and implement at least the following concepts (MUST):

Qa: Concepts and Implementations MUSTS

* the fit-predict interface, and a R^2 score function, * one-dimensional output only, * loss function based on (R)MSE, * setting of the number of iterations and learning rate (η) via parameters in the constructor, the signature of your __init__ must include the named parameters max_iter and eta0, * the batch-gradient decent algorithm (GD), * constant or adaptive learning rate, * learning graphs, * stochastic gradient descent (SGD), * epochs vs iteations, * compare the numerical optimization with the Closed-form solution.

Qb: [OPTIONAL] Additional Concepts and Implementations

And perhaps you could include (SHOULD/COULD):

* (stochastic) mini-bach gradient decent, * early stopping, * interface to your bias and weights via intercept_ and coef_ attributes on your linear regressor class, * get/set functionality of your regressor, such that is fully compatible with other Scikit-learn algorithms, try it out in say a cross_val_score() call from Scikit-learn, * test in via the smoke tests at the end of this Notebook, * testing it on MNIST data,

With the following no-nos (WONT):

* no multi-linear regression, * no reuse of the Scikit-learn regressor * no C/C++ optimized implementaion with a thin Python interface (nifty, but to much work for now), * no copy-paste of code from other sources WITHOUT a clear cite/reference for your source."
——— PROMT end ———
This created the following Linear Regressor:

```python
import numpy as np
import matplotlib.pyplot as plt

class MyLinReg:
    def __init__(self, max_iter=1000, eta0=0.01):
        self.max_iter = max_iter
        self.eta0 = eta0
        self.intercept_ = None
        self.coef_ = None

    def _initialize_weights(self, n_features):
        self.intercept_ = 0.0
        self.coef_ = np.zeros(n_features)

    def _compute_loss(self, X, y):
        n_samples = X.shape[0]
        predictions = self.intercept_ + np.dot(X, self.coef_)
        loss = np.sqrt(np.mean((predictions - y) ** 2))
        return loss

    def _compute_gradients(self, X, y):
        n_samples = X.shape[0]
        predictions = self.intercept_ + np.dot(X, self.coef_)
        error = predictions - y
        grad_intercept = np.mean(error)
        grad_coef = np.dot(X.T, error) / n_samples
        return grad_intercept, grad_coef

    def fit(self, X, y):
        self._initialize_weights(X.shape[1])
        for i in range(self.max_iter):
            grad_intercept, grad_coef = self._compute_gradients(X, y)
            self.intercept_ -= self.eta0 * grad_intercept
            self.coef_ -= self.eta0 * grad_coef

    def predict(self, X):
        return self.intercept_ + np.dot(X, self.coef_)

    def score(self, X, y):
        predictions = self.predict(X)
        u = ((y - predictions) ** 2).sum()
        v = ((y - y.mean()) ** 2).sum()
        return 1 - u / v
```

By default, given by the initialization function, our Linear Regressor has 1000 iterations and a learning rate of 0.01, unless parameters passed in the creation of a variable are

changed. For example, could we want a 100 iterations instead of the prefixed 1000, or a different learning rate per iteration, per epochs. The above code does one epoch per call. This means if we had a loop, where the Linear Regressor was called x-times; here the amount of times would equal the amount of epochs. From the initialization both coefficient and interception are created referring to the function call, and our feature size from the fit function:

```
def fit(self, X, y):
    self._initialize_weights(X.shape[1])
    for i in range(self.max_iter):
        grad_intercept, grad_coef = self._compute_gradients(X, y)
        self.intercept_ -= self.eta0 * grad_intercept
        self.coef_ -= self.eta0 * grad_coef
```

The fit function, once initialized, creates the weight (giving the coefficient and interception values), form here, depending on the amount of iterations provided to the Linear Regressor, the iterative process begins. It starts by creating the Gradient of both the coefficient and interception through the function _compute_gradients(self, X, y):

```
def _compute_gradients(self, X, y):
    n_samples = X.shape[0]
    predictions = self.intercept_ + np.dot(X, self.coef_)
    error = predictions - y
    grad_intercept = 2 * np.mean(error)
    grad_coef = 2 * np.dot(X.T, error) / n_samples
    return grad_intercept, grad_coef
```

However, looking at the Chat-GPT generated code, and the code snippet above you will notice that a very important numerical multiplier is missing. The numerical value 2 multiplied to the summed expression in both gradient of the coefficient and intercept. The corrected math (as shown in the code snippet above) behind the returned values are the following:

$$MSE \quad = \frac{1}{n} \cdot \Sigma_{i=0}^{n}((a \cdot x_i + b) - y_i)^2 \tag{4.1}$$

$$\nabla_a MSE = \frac{\partial MSE}{\partial a} \quad = \frac{1}{n} \cdot 2 \cdot \Sigma_{i=0}^{n}\ x \cdot ((a \cdot x_i + b) - y_i) \tag{4.2}$$

$$\nabla_b MSE = \frac{\partial MSE}{\partial b} \quad = \frac{1}{n} \cdot 2 \cdot \Sigma_{i=0}^{n}\ ((a \cdot x_i + b) - y_i) \tag{4.3}$$

Here x_i is a matrices of n times m size that is multiplied by the weight, which equals our predicted value. By subtracting the actual value from the predicted value the error is found, which is done in the following steps of the function:

```
predictions = self.intercept_ + np.dot(X, self.coef_)
error = predictions - y
```

Moving on, we would like for our function to minimize the error function (MSE), so we get the lowest possible MSE value. The only way to influence this, is through our coefficient and intercept values. By taking the partial derivative of our MSE with respect to our coefficient and intercept we can find the steepest ascend and then find the steepest descend afterwards. The gradients are found through the following bit of code:

```
    grad_intercept = 2 * np.mean(error)
    grad_coef = 2 * np.dot(X.T, error) / n_samples
```

If we were to look at the code snippet and compare it to the mathematical expression 3.2 & 3.3 we will find them as being equal. These values are then returned from the function.

Returning to our fit-function iteration, our gradients are now initialized and the iterative process to minimize the error continues. Our gradients have the value that is connected to the steepest **ASCEND** and since we want to minimize, we would like for the opposite. So, what we do is creating a new coefficient and interception for the next iteration like so:

$$Coefficient = Coefficient - \eta_0 \cdot \nabla_a \ MSE \tag{4.4}$$

$$Interception = Interception - \eta_0 \cdot \nabla_b \ MSE \tag{4.5}$$

Looking at the code again:

```
def fit(self, X, y):
    self._initialize_weights(X.shape[1])
    for i in range(self.max_iter):
        grad_intercept, grad_coef = self._compute_gradients(X, y)
        self.intercept_ -= self.eta0 * grad_intercept
        self.coef_ -= self.eta0 * grad_coef
```

These new values for our coefficient and intercept is passed on through the hole function and defines our gradient descend. So, when the gradient function is called, new values of gradients will be outputted, so on and so forth for each iteration. With some print statements added to our functions for ease and overview:

```
import numpy as np
import matplotlib.pyplot as plt

class MyLinReg:
```

```
5
6  # __init__ is the computerpower of this class
7
8      def __init__(self, max_iter=1000, eta0=0.01):
9          self.max_iter = max_iter
10         self.eta0 = eta0
11         self.intercept_ = None
12         self.coef_ = None
13
14 # Our weights is our manipulated data through each iteration.
15 # However, they need a call before they can be initialize. Hence this function.
16     def _initialize_weights(self, n_features):
17         self.intercept_ = 0.0
18         self.coef_ = np.zeros(n_features)
19
20     def _compute_gradients(self, X, y):
21         n_samples = X.shape[0]
22         predictions = self.intercept_ + np.dot(X, self.coef_)
23         error = y - predictions
24         grad_intercept = -2 * np.mean(error)
25         grad_coef = -2 * np.dot(X.T, error) / n_samples
26
27         print(f"This is our interception: {self.intercept_}")
28         print(f"This is our coefficient: {self.coef_}")
29         print("-----")
30         print(f"This is the amount of samples we have in our data: {n_samples} &
               ↪ {y}")
31         print(f"This is our predictions, our linear line: {predictions} &
               ↪ {np.dot(X, self.coef_)} ")
32         print(f"This is the error between our predicted value and the y-data
               ↪ points: {error}")
33         print(f"This is the gradient of our constant: {grad_intercept}")
34         print(f"This is the gradient of our coefficient: {grad_coef}")
35         print("")
36         return grad_intercept, grad_coef
37
38
39 # When we fit our data we run through every iteration.
40 # First time we run through it our intercept and coefficient is o.
41 # However for each iteration our last gradient intercept and coefficient is our
   ↪ new intercept and coefficient that is surpassed to all functions using
   ↪ the variable.
42
43     def fit(self, X, y):
44         self._initialize_weights(X.shape[1])
45         for i in range(self.max_iter): # iterates over the entered amount
               ↪ entered itertions.
```

```
            print(f"This is interation {i+1}")
            grad_intercept, grad_coef = self._compute_gradients(X, y)
            self.intercept_ -= self.eta0 * grad_intercept # Learning rate
                ↪ multiplied by the gradient of our intercept [a + x * b] =>
                ↪ gradient(a) -> Creating a new intercept
            self.coef_ -= self.eta0 * grad_coef # Learning rate multiplied by
                ↪ the gradient of our coefficient [a + x * b] => gradient(b) ->
                ↪ Creating a new coefficient

# Our prediction is our y-point data from our guess: y_pred = a + x * c
# Each call will contain new intercepts and new coefficients
    def predict(self, X):
        return self.intercept_ + np.dot(X, self.coef_)

    def score(self, X, y):
        predictions = self.predict(X)
        u = ((y - predictions) ** 2).sum()
        v = ((y - y.mean()) ** 2).sum()
        return [1 - u / v, u, v]
```

The iterations can be shown. Calling the function with code taking from the assignment:

```
import matplotlib.pyplot as plt
def GenerateData():
    X =
        ↪ np.array([[8.34044009e-01],[1.44064899e+00],[2.28749635e-04],[6.04665145e-01]])
    y = np.array([5.97396028, 7.24897834, 4.86609388, 3.51245674])
    return X, y

X, y = GenerateData()

model = MyLinReg(4,0.4)
model.fit(X,y)
model.predict(X)
model.score(X,y)
display(model.score(X,y)[0])
plt.scatter(X,y)
plt.plot(list(range(0,3)), [model.score(X,y)[1] + x * model.score(X,y)[2] for x
    ↪ in range(0,3)])
```

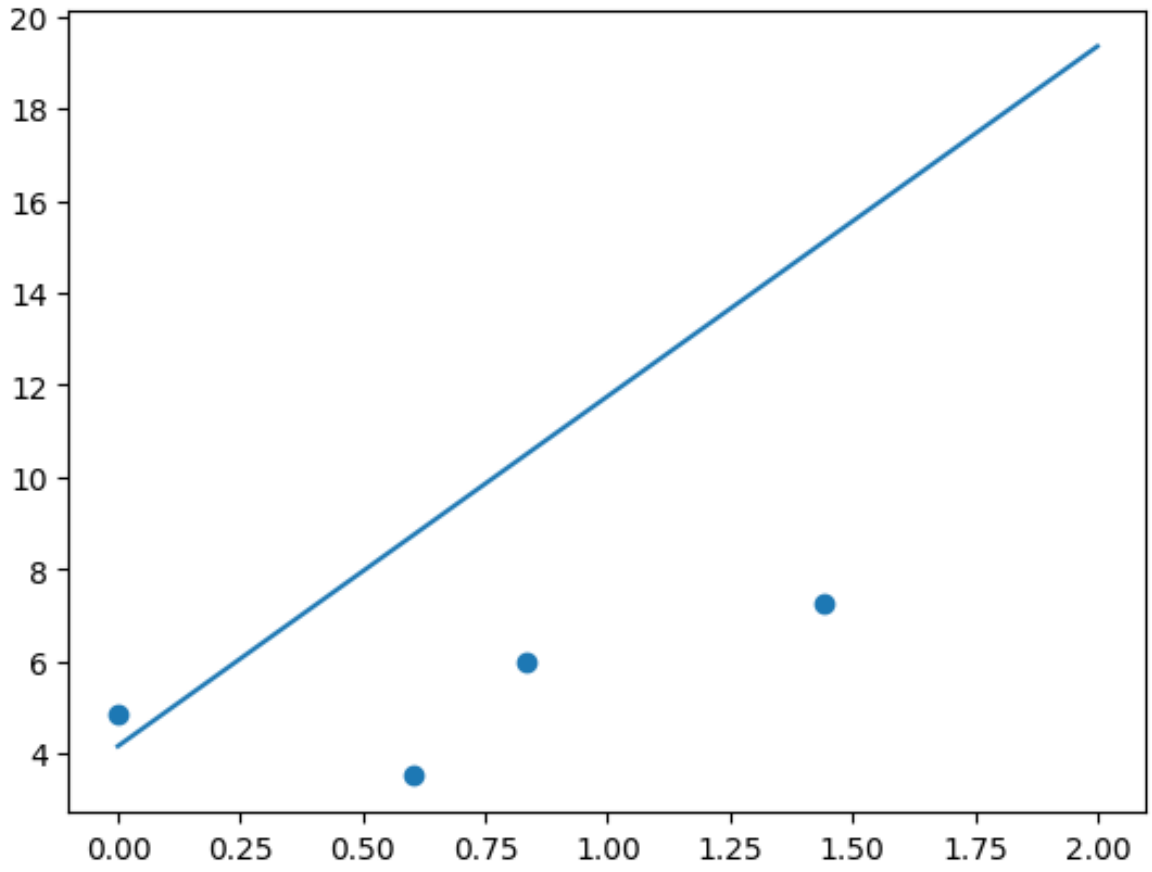
Some of the output looks like the following (shown are the two first prints from the print statements):

```
This is iteration 1
This is our interception: 0.0
This is our coefficient: [0.]
-----
```

```
This is the amount of samples we have in our data: 4 & [5.97396028 7.24897834 4.86609388
    ↪ 3.51245674]
This is our predictions, our linear line: [0. 0. 0. 0.] & [0. 0. 0. 0.]
This is the error between our predicted value and the y-data points: [5.97396028 7.24897834
    ↪ 4.86609388 3.51245674]
This is the gradient of our constant: -10.800744620000001
This is the gradient of our coefficient: [-8.77537619]

This is iteration 2
This is our interception: 4.320297848000001
This is our coefficient: [3.51015048]
-----
This is the amount of samples we have in our data: 4 & [5.97396028 7.24897834 4.86609388
    ↪ 3.51245674]
This is our predictions, our linear line: [7.24791782 9.37719259 4.32110079 6.4427635 ] &
    ↪ [2.92761998e+00 5.05689474e+00 8.02945640e-04 2.12246565e+00]
This is the error between our predicted value and the y-data points: [-1.27395754 -2.12821425
    ↪ 0.54499309 -2.93030676]
This is the gradient of our constant: 2.8937427306428196
This is the gradient of our coefficient: [2.95013803]
```

The scatter plot and the predicted score are as follows:

```
0.45260799807744334
```



Figur 4.1: Scatter plot of selected points and the prediceted best fit linear regression line

Which ends the 'model.score(X,y)' call.

4.1 QF Smoke testing our Linear Regressor

From testing the supplied code we found that a 'eps' of 1E-5, in line 95, was needed before we would be able to run the code. With the 'esp' adjusted we got the following results from our Linear Regressor:

```
y_pred = [5.61498304 6.75547481 4.04730809 5.18372265]
SCORE = 0.49500564295554395
bias = 4.046878010107266
coefficients = [1.88012265]
   w =[4.046878010107266, array([1.880122650194])]
```

```
   w_expected=[4.046879011698 1.880121487278]
OK
```

4.2 QH A conclusion

We have been introduced to the working of a Linear Regressor class with some of the things it might contain. Since our regressor i some what simple, features such as epochs iterations could have been added to make the is consider either the double amount of iterations or half the learning for optimization. We have been introduced to the workings of the gradient descend, and how one might handle it, when the output of the gradient focuses on (maximizing) ascend. A valuable lesson was also found in the small mistakes one might face, when asking a transformer model (Chat-GPT) for an answer to a assignment, where it might have been quicker to sit down and read the book and gotten inspiration from there.

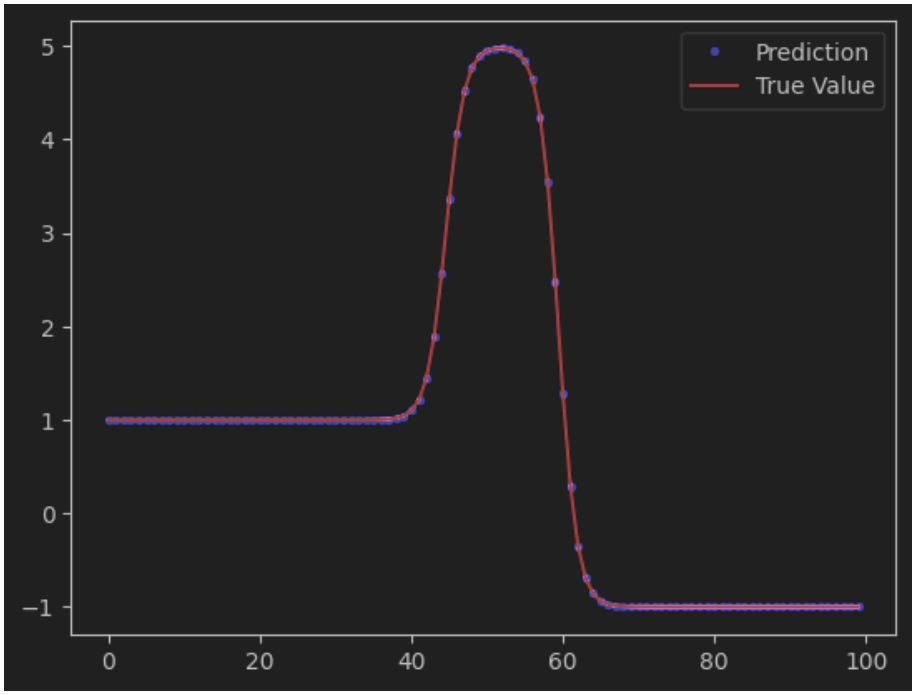
5 L06 ANN

5.1 Qa: plot ANN

We were simply asked to plot the prediction of the ANN to demonstrate.

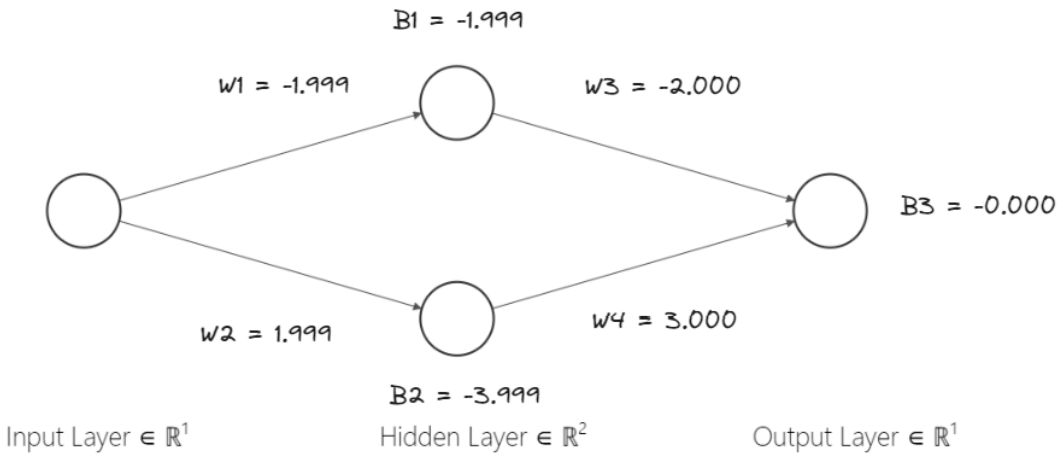
using matplotlib:

```
plt.plot(y_pred, "b.", label='Prediction') #Plot blue dots
plt.plot(y_true,"r" ,label='True Value') #Plot red line
plt.legend()
plt.show()
```



Figur 5.1: Looks very good, prediction is accurate

5.2 Qb: Drawing



5.3 Qc: Create a mathematical formula for the network ala

we insert into the formula.

$$y_{math} = -2\ 000 * tanh(-1\ 999 * x - 1\ 999) - 3\ 000 * tanh(1\ 999 * x - 3\ 999) + 0\ 000 \tag{5.1}$$

here are values in order: -2.000 = weight 1 from hidden to output -1.999 = weight 1 from input to hidden -1.999 = bias 1 for hidden layer -3.000 = weight 2 from hidden to output 1.999 = weight 2 from input to hidden -3.999 = bias 2 for hidden layer 0.000 = bias for output layer.

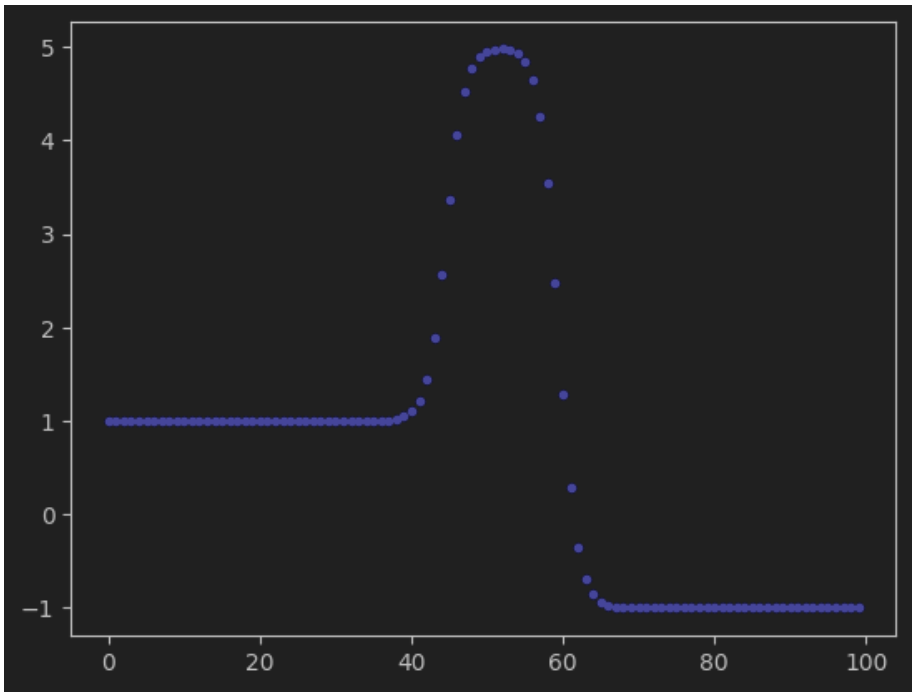
in code:

```
1 y_math = -2.000*math.tanh(-1.999*x-1.999)-3.000*math.tanh(1.999*x-3.999)+0.000
```

5.4 Qd: Plot the mathematical formula

we simply create an empty list an itterate over X to get x, then run use the formula on these x values.

```
y_math_pred = []
for x in X:
    y_math =
        ↪ -2.000*math.tanh(-1.999*x-1.999)-3.000*math.tanh(1.999*x-3.999)+0.000
    y_math_pred.append(y_math)

plt.plot(y_math_pred, "b.", label='Prediction')
```



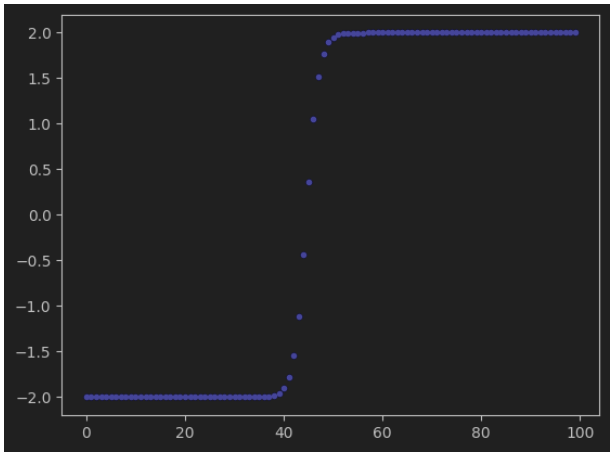
Figur 5.2: Looks perfect!

5.5 Qe: Plot the function piece wise

we use essentally the same code as before. First section:

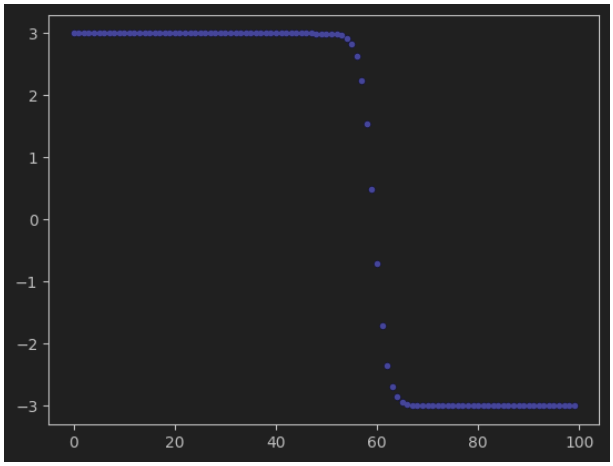
```
for x in X:
    y_math =
        ↪ -2.000*math.tanh(-1.999*x-1.999)-3.000*math.tanh(1.999*x-3.999)+0.000
    y_math_pred.append(y_math)
```

```
4 plt.plot(y_math_pred, "b.", label='Prediction')
```



second section:'

```
1 for x in X:
2     y_math = -3.000*math.tanh(1.999*x-3.999)
3     y_math_pred.append(y_math)
4 plt.plot(y_math_pred, "b.", label='Prediction')
5 plt.show()
```

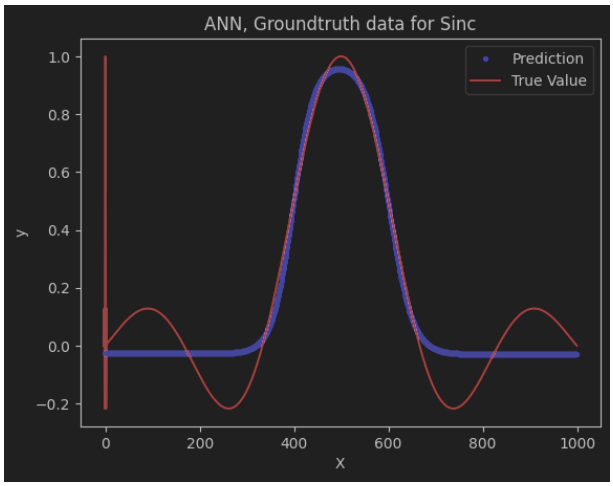


We were also asked if the parts look like tanh functions. While the second one is inverted they are still very good fits.

5.6 Qf: Use sinc data instead

We want to see if we can trian an ann on a more advanced dataset, this will require more layers and neurons.

using similar code as earlier, we just do a test run to see how our two neuron 1 layer ann handles the case:



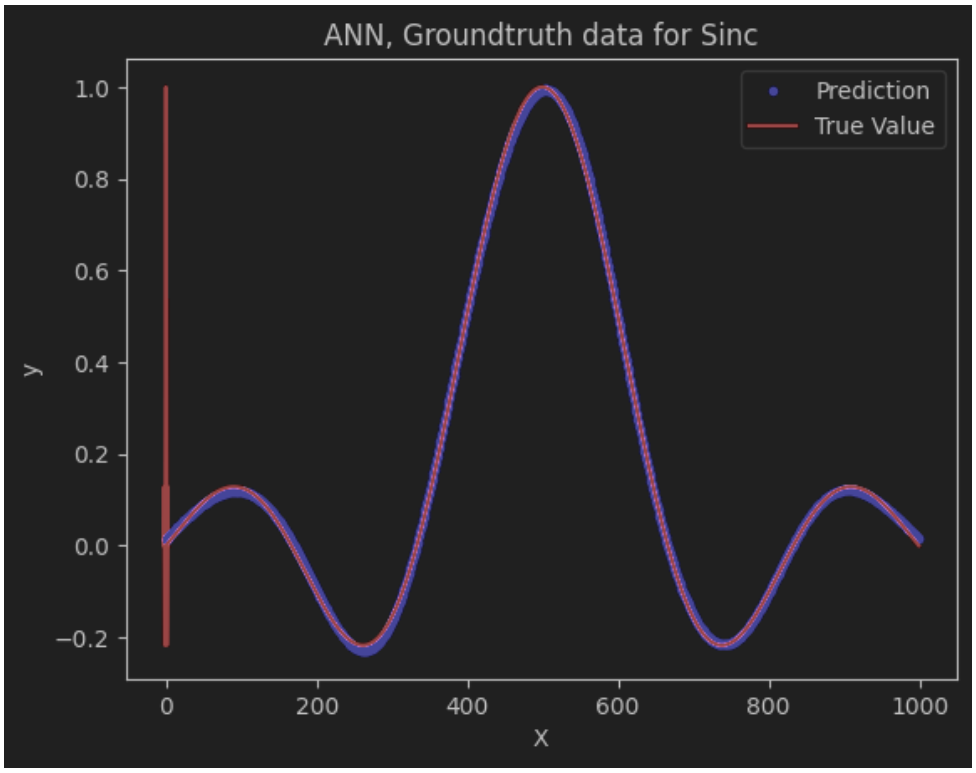
Its quite clear that it fits poorly as the ann is not able to handle the complexity of the data.

we increase our neuron and layer count.

through testing, we get a good prediction with 6 layers with 6 neurons. Here is the model in code:

```
mlp = MLPRegressor(activation='tanh', # activation function
                   hidden_layer_sizes=[6], # layes and neurons in layers: one
                       ↪ hidden layer with two neurons
                   alpha=1e-5, # regularization parameter
                   solver='lbfgs', # quasi-Newton solver
                   max_iter=10000,
                   verbose=True)
```

this gives us the folliowng result:



This gives us a quite good result other then the strange noise at the start of the data. It should probably be cut out in any case. The model fits well with 6 layers and 6 neurons, however it took a few runs for it to get a good fit due to the randomness of an ann. Lower layer and neuron count was possible, i got OK results at 4 layers and 4 neurons but it was unreliable and mostly made poor fits, so 6 was chosen as a baseline.