Machine Learning - Assignment 2

Frederik Predut - StudieNr: 201502305

Magnus Hauge Kyneb - StudieNr: 201807279

Sebastian Laczek Nielsen - StudieNr: 201806678

Gruppe 22 - Oktober 2020

Table of Contents

1 L04 **3**
1.1 Exercise a 3
1.2 Exercise b 5
1.3 Exercise c 5
1.4 Exercise d 6
1.5 Exercise e 7
1.6 Exercise f 7
1.7 Exercise g 8
1.8 Exercise 2a 9
1.9 Exercise 2b 10
1.10 Exercise 2c 11

2 L05 **12**
2.1 Exercise a 12
2.2 Exercise b 13
2.3 Exercise c 14
2.4 Exercise d 14
2.5 Exercise e 15

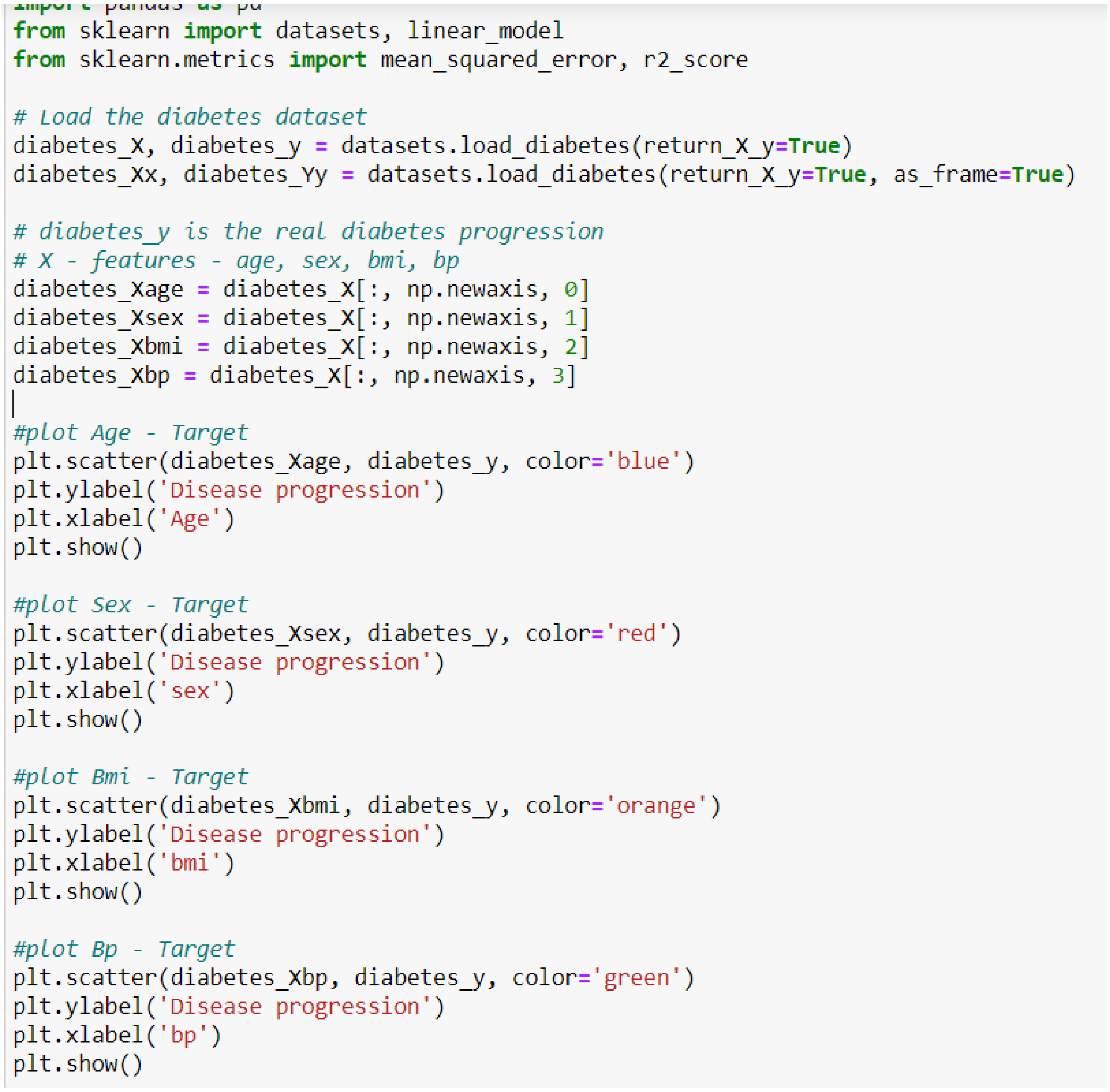
3 L06 **16**
3.1 Exercise a 16
3.2 Exercise b 17
3.3 Exercise c 18
3.4 Exercise d 19
3.5 Exercise e 19
3.6 Exercise f g 20

4 Data analysis of own project **22**
4.1 22

1 L04

1.1 Exercise a

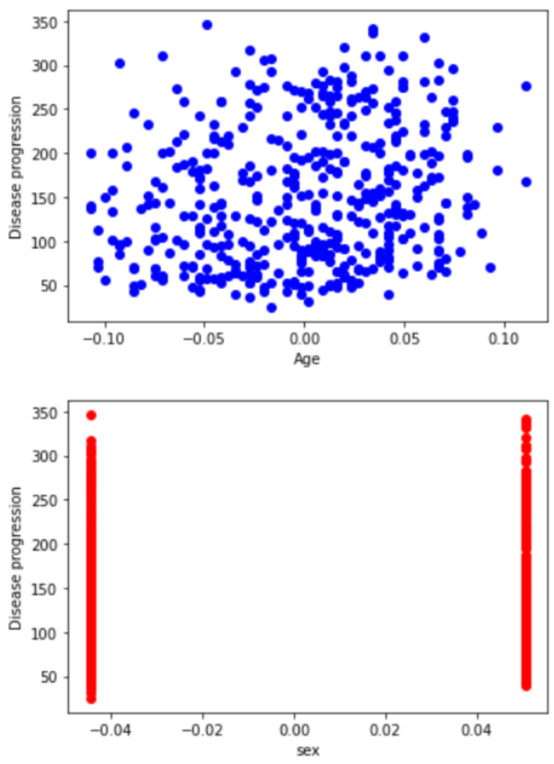
In this exercise we plot the "disease progerssion" i relation to the four features age, sex, bmi and blood pressure. First the dataset is loaded and each feature is extracted. Each feature is then compared to Y-value (true disease progression) of the dataset.



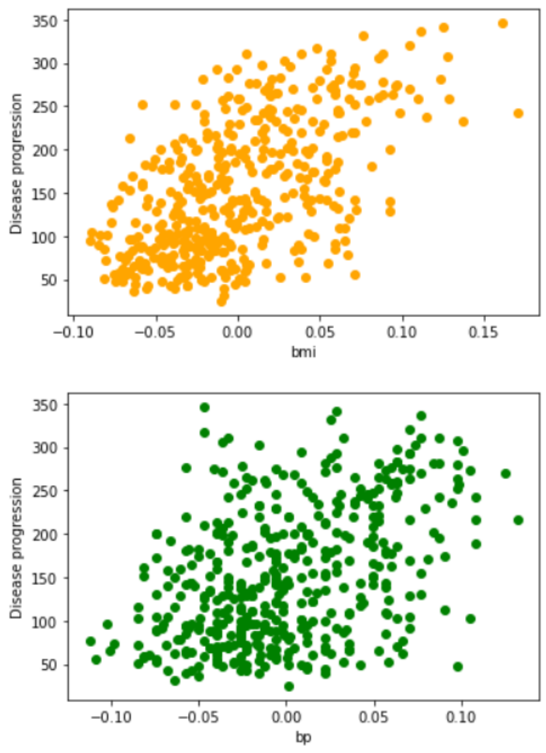
```
import pandas as pd
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

# Load the diabetes dataset
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True)
diabetes_Xx, diabetes_Yy = datasets.load_diabetes(return_X_y=True, as_frame=True)

# diabetes_y is the real diabetes progression
# X - features - age, sex, bmi, bp
diabetes_Xage = diabetes_X[:, np.newaxis, 0]
diabetes_Xsex = diabetes_X[:, np.newaxis, 1]
diabetes_Xbmi = diabetes_X[:, np.newaxis, 2]
diabetes_Xbp = diabetes_X[:, np.newaxis, 3]

#plot Age - Target
plt.scatter(diabetes_Xage, diabetes_y, color='blue')
plt.ylabel('Disease progression')
plt.xlabel('Age')
plt.show()

#plot Sex - Target
plt.scatter(diabetes_Xsex, diabetes_y, color='red')
plt.ylabel('Disease progression')
plt.xlabel('sex')
plt.show()

#plot Bmi - Target
plt.scatter(diabetes_Xbmi, diabetes_y, color='orange')
plt.ylabel('Disease progression')
plt.xlabel('bmi')
plt.show()

#plot Bp - Target
plt.scatter(diabetes_Xbp, diabetes_y, color='green')
plt.ylabel('Disease progression')
plt.xlabel('bp')
plt.show()
```

Figur 1: Code snippet for question a

As can be seen on the plots on figure 2 and 3, some features seem to show more of a correlation than others. While the correlation is difficult to spot in the age and sex plots, blood pressure and especially bmi show a clear trend. It seems like there's a positive correlation between bmi and disease progression.



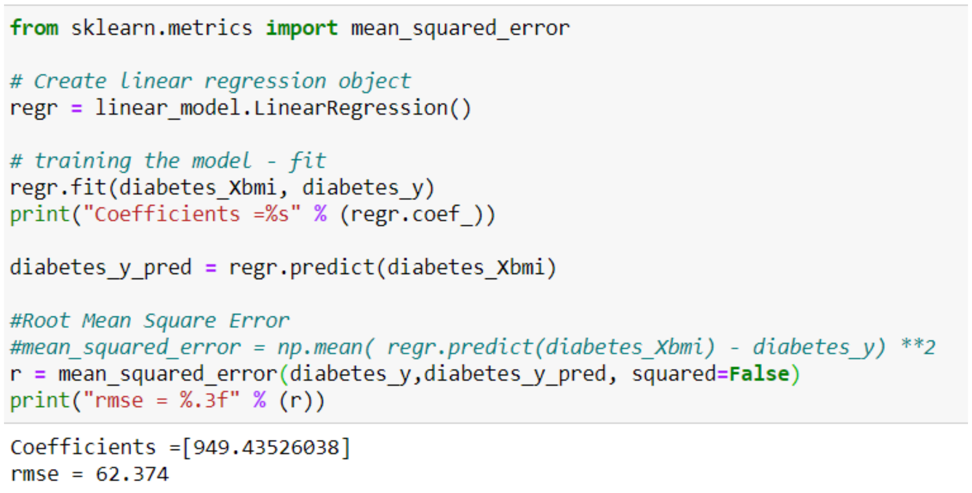
Figur 2: Plot of age and sex feature



Figur 3: Plot of bmi and bp

1.2 Exercise b

In this exercise, we use the feature that seemed to show the most potential, to fit a linear regression model. After this, we find the root-mean-square-error of the model.



```
from sklearn.metrics import mean_squared_error

# Create linear regression object
regr = linear_model.LinearRegression()

# training the model - fit
regr.fit(diabetes_Xbmi, diabetes_y)
print("Coefficients =%s" % (regr.coef_))

diabetes_y_pred = regr.predict(diabetes_Xbmi)

#Root Mean Square Error
#mean_squared_error = np.mean( regr.predict(diabetes_Xbmi) - diabetes_y) **2
r = mean_squared_error(diabetes_y,diabetes_y_pred, squared=False)
print("rmse = %.3f" % (r))
```

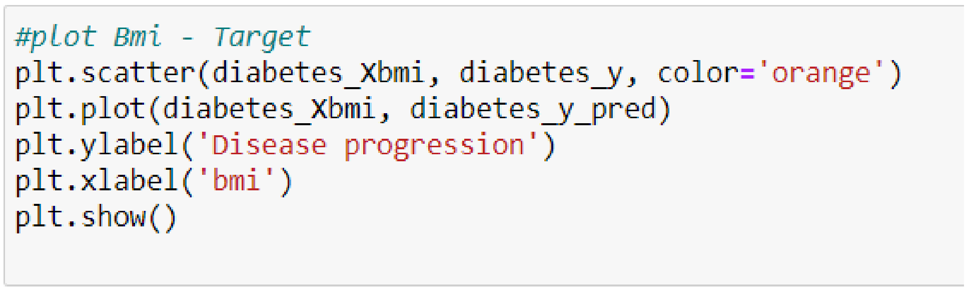
```
Coefficients =[949.43526038]
rmse = 62.374
```

Figur 4: Code snippet for question b

We get a RMSE of 62.374. This is a baseline we will use to compare future models to.

1.3 Exercise c

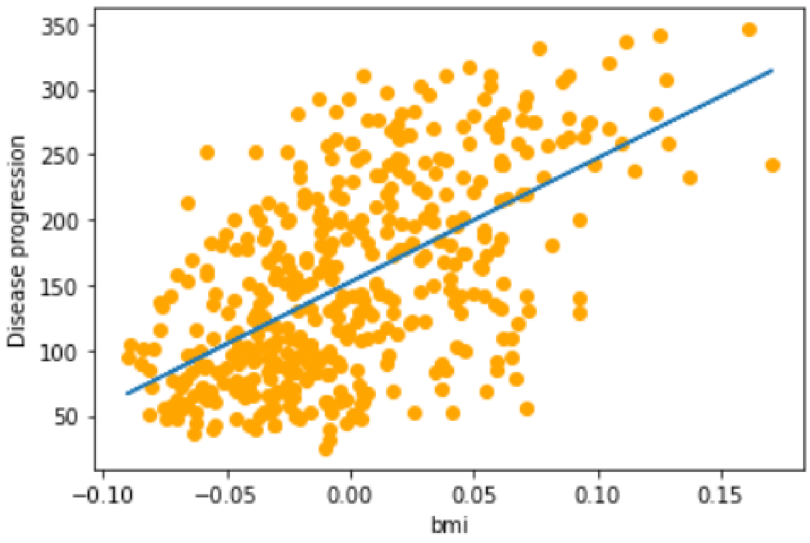
In this exericise, we plot the linear model on top of the bmi and disease progerssion correlation data. This can give us on idea of how on track the model is.



```
#plot Bmi - Target
plt.scatter(diabetes_Xbmi, diabetes_y, color='orange')
plt.plot(diabetes_Xbmi, diabetes_y_pred)
plt.ylabel('Disease progression')
plt.xlabel('bmi')
plt.show()
```

Figur 5: Code snippet for question c

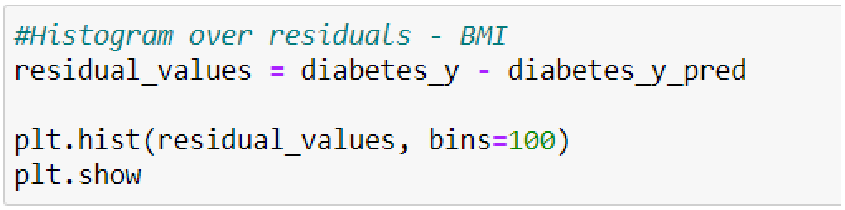
From the plot on figure: 6, we can see the model is followig the expected trendline in a productive fashion.



Figur 6: Plot of bmi with regression line

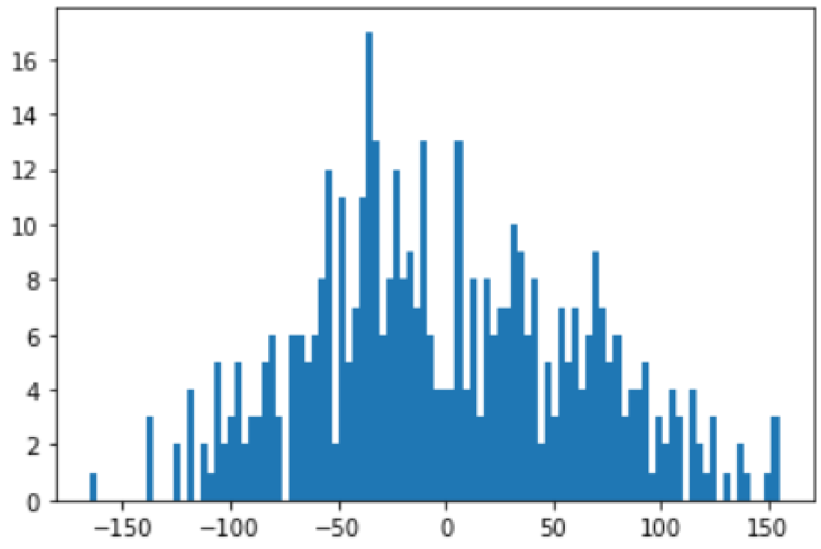
1.4 Exercise d

In this exercise, a histogram of the residuals is shown. To calculate the residuals, we look at the difference between each prediction and the true disease progression value figure: 7.



```python
#Histogram over residuals - BMI
residual_values = diabetes_y - diabetes_y_pred

plt.hist(residual_values, bins=100)
plt.show
```

Figur 7: Code snippet for question d

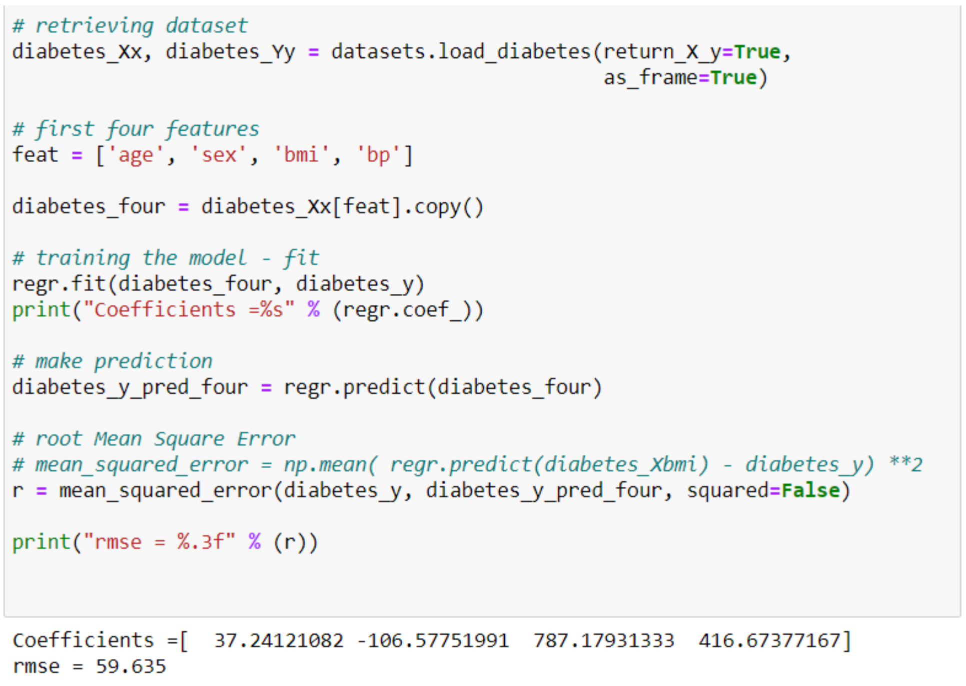


Figur 8: Plot of residuals in a histogram

In the histogram on figure: 8, we hope to see as many values around 0 as possible. The resulting histogram drawn follows a normal distribution around the value 0. On top of that, we have a relatively limited amount of outliers. There's 4 values with a positive difference of around 150. These are outliers, but our assessment is that it's not too many or too far out to be a problem that might drag the model in a wrong direction.

1.5 Exercise e

In this exercise we create a model based 4 features rather than just the 1. The 4 features are age, sex, bmi and blood pressure. The linear regression model is trained on the 4 features using the fit function and another prediction is made and RMSE is calculated see figure: 9.



```
# retrieving dataset
diabetes_Xx, diabetes_Yy = datasets.load_diabetes(return_X_y=True,
                                                   as_frame=True)

# first four features
feat = ['age', 'sex', 'bmi', 'bp']

diabetes_four = diabetes_Xx[feat].copy()

# training the model - fit
regr.fit(diabetes_four, diabetes_y)
print("Coefficients =%s" % (regr.coef_))

# make prediction
diabetes_y_pred_four = regr.predict(diabetes_four)

# root Mean Square Error
# mean_squared_error = np.mean( regr.predict(diabetes_Xbmi) - diabetes_y) **2
r = mean_squared_error(diabetes_y, diabetes_y_pred_four, squared=False)

print("rmse = %.3f" % (r))
```

```
Coefficients =[  37.24121082 -106.57751991  787.17931333  416.67377167]
rmse = 59.635
```

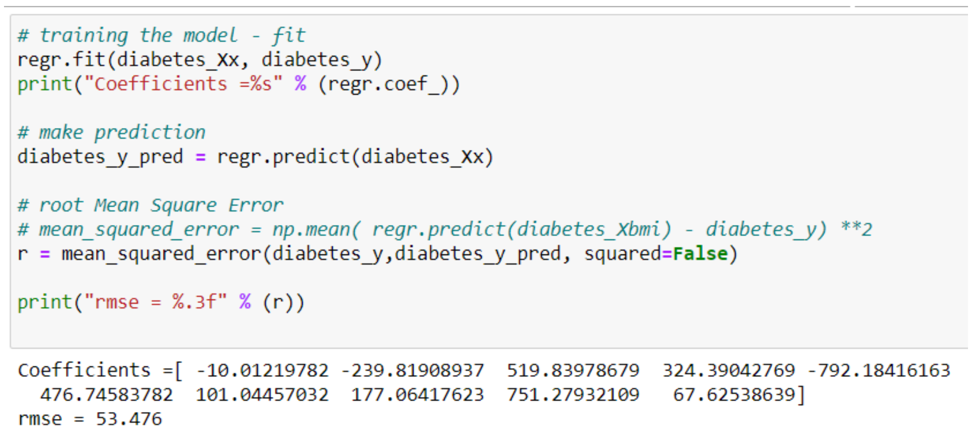
Figur 9: Code snippet for question e

The resulting RMSE is 59.635. This is a lower error than the one we got from fitting with just the BMI-data (RMSE: 62.374). While the human eye might not have been able to spot correlations in the sex-plot, age-plot and blood-pressure-plot, the machine learning method is able to extract some additional correlation.

1.6 Exercise f

In this exercise, all the features of the diabetes dataset is used to fit the model. The RMSE is then calculated.

The resulting RMSE is 53.476. This is a lower error than both the one we got from fitting with just the BMI-data (RMSE: 62.374) as well as the one from using 4 features (RMSE: 59.635). Again, the machine learning method is able to extract some additional correlation from the added information.



```
# training the model - fit
regr.fit(diabetes_Xx, diabetes_y)
print("Coefficients =%s" % (regr.coef_))

# make prediction
diabetes_y_pred = regr.predict(diabetes_Xx)

# root Mean Square Error
# mean_squared_error = np.mean( regr.predict(diabetes_Xbmi) - diabetes_y) **2
r = mean_squared_error(diabetes_y,diabetes_y_pred, squared=False)

print("rmse = %.3f" % (r))
```

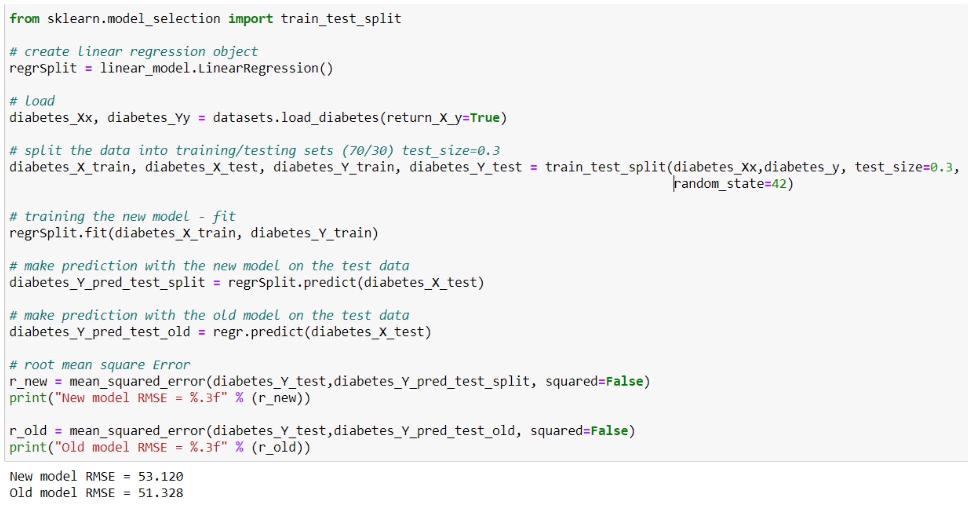
```
Coefficients =[ -10.01219782 -239.81908937  519.83978679  324.39042769 -792.18416163
  476.74583782  101.04457032  177.06417623  751.27932109   67.62538639]
rmse = 53.476
```

Figur 10: Code snippet for question f

1.7 Exercise g

In this exercise, we split the diabetes dataset into train and test set see figure: 11. The split is 70-train/30-test. Then a new linear model is trained by fitting it the training part of the set. It is then asked to predict the test part of the set that it has never seen before. It's RMSE on the test data is then calculated.

To compare this new model to the old model trained on the entire data set, the old model is asked to predict the same 30-test values that the new model was asked to predict. Knowing that the old model has already seen the "answers" and used these in the training process, it is expected that the old model will perform better.



```
from sklearn.model_selection import train_test_split

# create linear regression object
regrSplit = linear_model.LinearRegression()

# load
diabetes_Xx, diabetes_Yy = datasets.load_diabetes(return_X_y=True)

# split the data into training/testing sets (70/30) test_size=0.3
diabetes_X_train, diabetes_X_test, diabetes_Y_train, diabetes_Y_test = train_test_split(diabetes_Xx,diabetes_y, test_size=0.3,
                                                                                        random_state=42)

# training the new model - fit
regrSplit.fit(diabetes_X_train, diabetes_Y_train)

# make prediction with the new model on the test data
diabetes_Y_pred_test_split = regrSplit.predict(diabetes_X_test)

# make prediction with the old model on the test data
diabetes_Y_pred_test_old = regr.predict(diabetes_X_test)

# root mean square Error
r_new = mean_squared_error(diabetes_Y_test,diabetes_Y_pred_test_split, squared=False)
print("New model RMSE = %.3f" % (r_new))

r_old = mean_squared_error(diabetes_Y_test,diabetes_Y_pred_test_old, squared=False)
print("Old model RMSE = %.3f" % (r_old))
```

```
New model RMSE = 53.120
Old model RMSE = 51.328
```

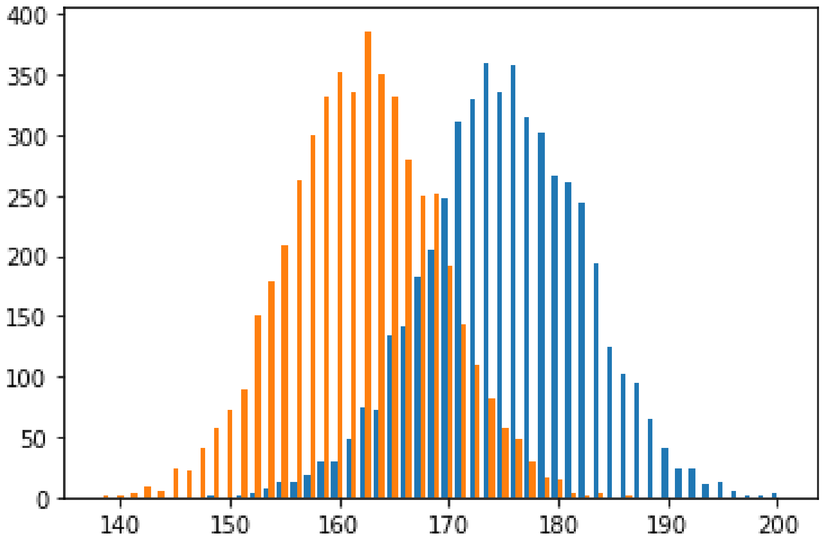
Figur 11: Code snippet for question g

As expected, the old model performs better with an RMSE of 51.328 compared to the new model with an RMSE of 53.120.

This might indicate that the old method was overfitted slightly. However, with a linear model, it's limited how overfitted the model can become. If the two models, old and new, had performed the exact same, it might have indicated that we had already reached a somewhat generalized and more "correct" model in using the old model. The difference in RMSE shows sign of the opposite, that the linear models are not as accurate as a linear model could be.This, in return, could be a sign that our dataset is too small and in this case is looking at too many irrelevant parameters that is adding noise to the limited amount of data we have.

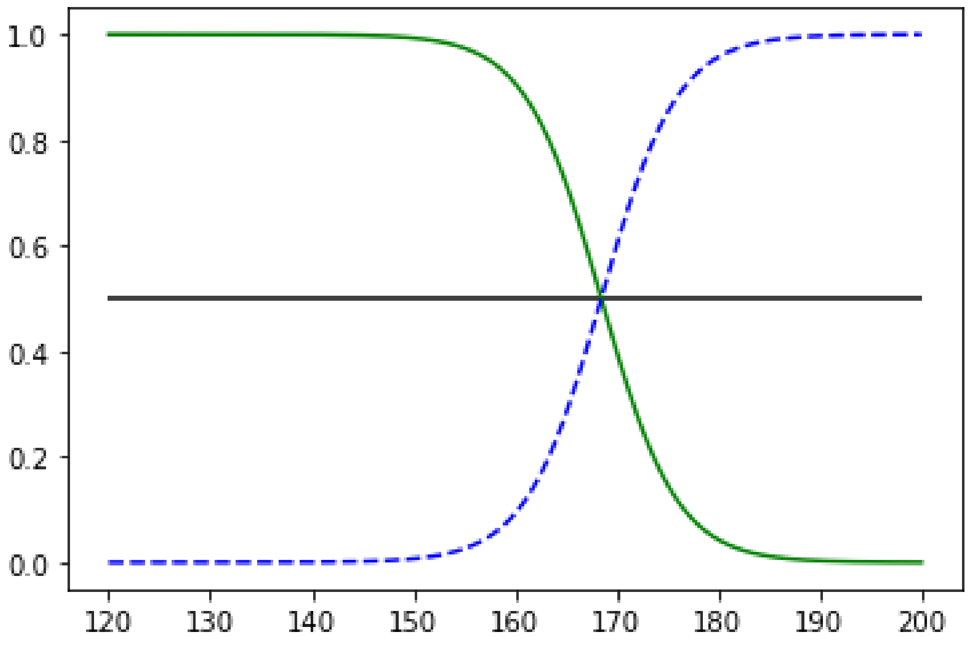
1.8 Exercise 2a

Logistic regression for height and weight data set will be focused on in the following exercises. Firstly the height data set is plotted as a histogram below on figure 12. after this a logistic regression is fitted



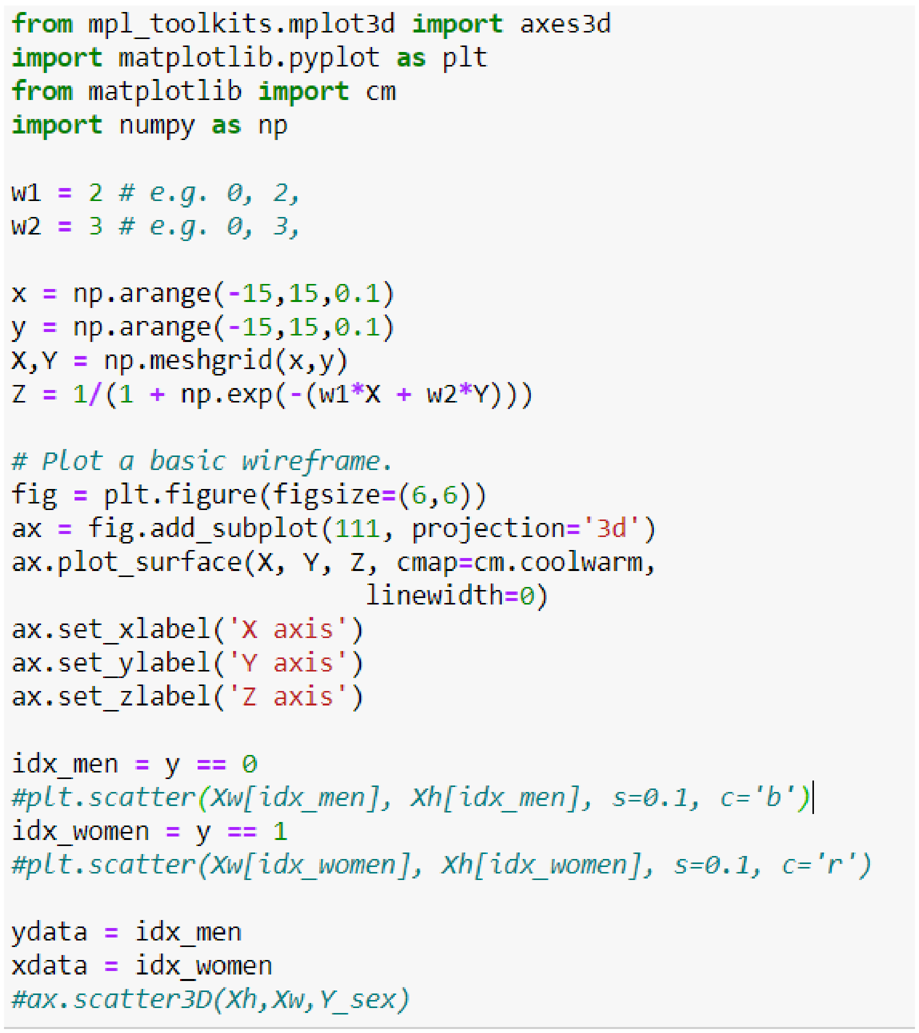
Figur 12: Histogram of height for both male and female

on the data as seen on figure 13



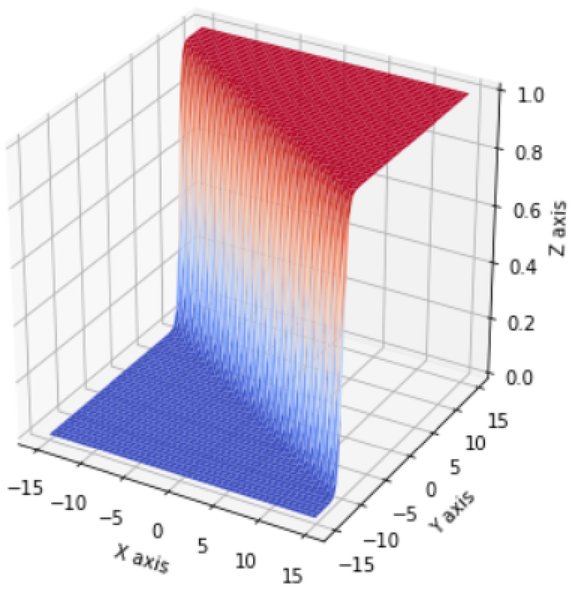
Figur 13: Logistic regression for the height

1.9 Exercise 2b



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

w1 = 2 # e.g. 0, 2,
w2 = 3 # e.g. 0, 3,

x = np.arange(-15,15,0.1)
y = np.arange(-15,15,0.1)
X,Y = np.meshgrid(x,y)
Z = 1/(1 + np.exp(-(w1*X + w2*Y)))

# Plot a basic wireframe.
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

idx_men = y == 0
#plt.scatter(Xw[idx_men], Xh[idx_men], s=0.1, c='b')
idx_women = y == 1
#plt.scatter(Xw[idx_women], Xh[idx_women], s=0.1, c='r')

ydata = idx_men
xdata = idx_women
#ax.scatter3D(Xh,Xw,Y_sex)
```

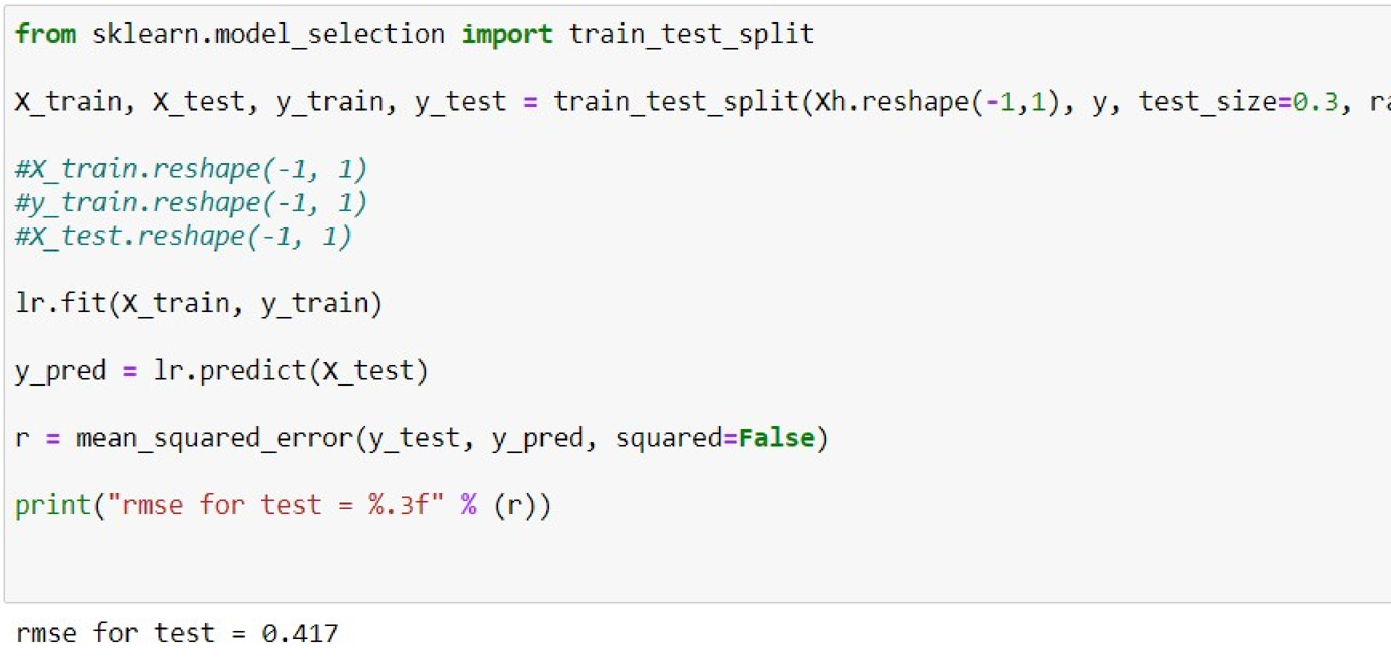
Figur 14: Plot Code



Figur 15: 3D plot of the logistic function

1.10 Exercise 2c

Lastly to train with a train/test set, the following code is made, as seen on figure16.



```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(Xh.reshape(-1,1), y, test_size=0.3, r

#X_train.reshape(-1, 1)
#y_train.reshape(-1, 1)
#X_test.reshape(-1, 1)

lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)

r = mean_squared_error(y_test, y_pred, squared=False)

print("rmse for test = %.3f" % (r))
```

```
rmse for test = 0.417
```

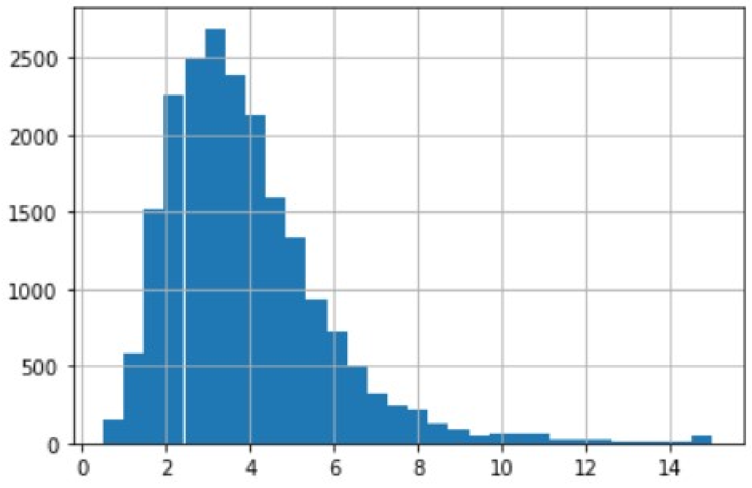
Figur 16: Code for getting the rmse value for the new train test set

2 L05

This section is gonna be an analysis of the california housing prices from kaggle[1].

2.1 Exercise a

Below on figure 17 is shown median_income.



Figur 17: Plot of median income

To calculate the median, the function from the pandas library median() is called on the datafile, this

[1]https://www.kaggle.com/camnugent/california-housing-prices

just returns the median, however it could also be gained by taking the middle most value in the datafile assuming it's sorted. The mean is calculated taking the sum and then average it. Lastly the range is calculated by taking the difference between the max and minimum values, so we know the range the dataset is operating on. Thus the values are as follows:

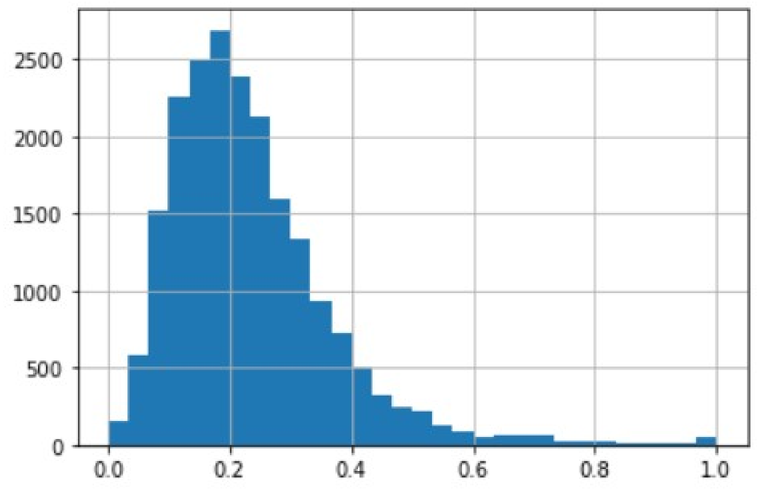
Median: 3.535
Mean: 3.871
Range: 15.500

2.2 Exercise b

There is a slight difference between median and mean values, the one that best describes a 'normal family's income', would be the median, as the mean could easily be skewed by a few high income families, e.g. Millionares or Bilionares. However in this dataset it does not seem to impact much.

2.3 Exercise c

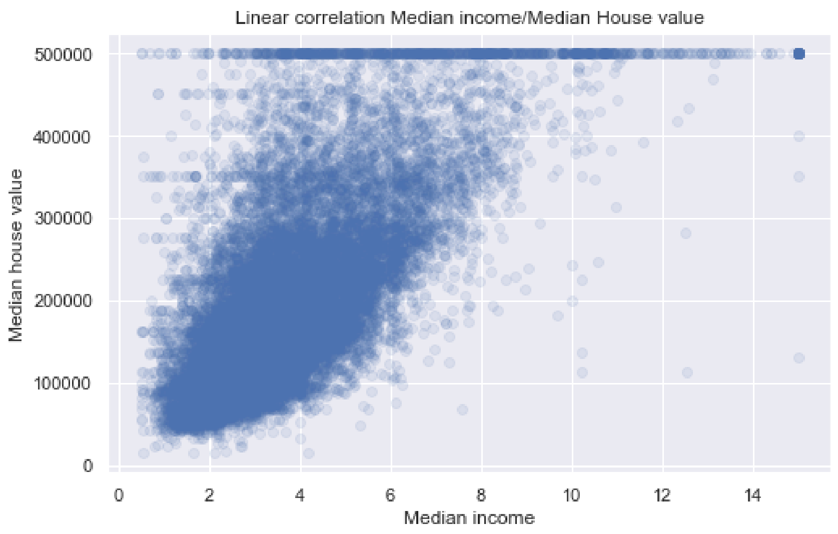
Here the data is normalized and plotted on figure 18. Here it shows the two plots still match each other.



Figur 18: Normalized data plot

2.4 Exercise d

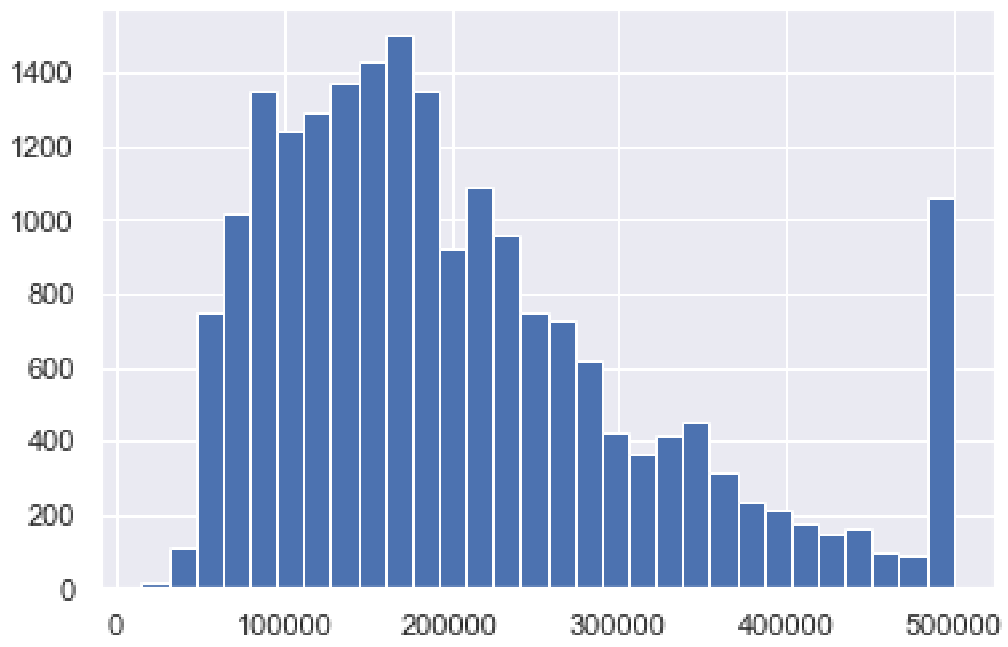
On figure 19 is show the correlation between median income and median house value.As can be seen on the graph there is a tendency for lower income owns lower value houses, and the higher the income the higher the housing value. However when the house value rises to above 300k, there is a trend that does not seem to correlate as strongly with the median income. And the 500k median house value seems be spread out on most middle incomes, which realisticly seems odd.



Figur 19: Linear correlation of both median income and median house value

2.5 Exercise e

To calculate the 5% and the 95% value for median house value, is to take the datafram size and times it by 0.05 and 0.95 respectively. This gives the index in a sorted dataframe, that returns; 95%: 489.800, and 5% 66.200. Below on figure 20 is seen the how the 95% and 5% fit well with the plot.



Figur 20: The median house value plotted

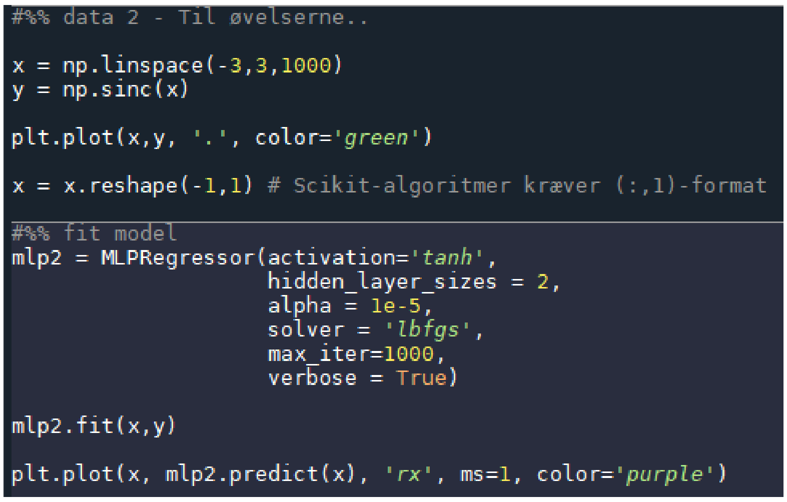
However even though the percentage values fit with the plot, does not necessarily make them realistic. It is not very realistic for the max values to all be exactly equal, which might also skew the 95% value. The amount max values are also a steep increase from the amount of second highest value, meaning the 95% value is going to be closer to the max value, which inflates the perception of housing values. More realistic data probably shows a very different curve of the current max value. It almost seem the data has a max value for housing values, meaning a house value going above 500k, seems to be rounded down to 500k. Meaning realisticly these values would probably be spread out far between.

3 L06

This section is going to be about our experiments with Scikit-learns version of supervised neural network. More specifically, the 2-layer Multi Layer Perceptron.

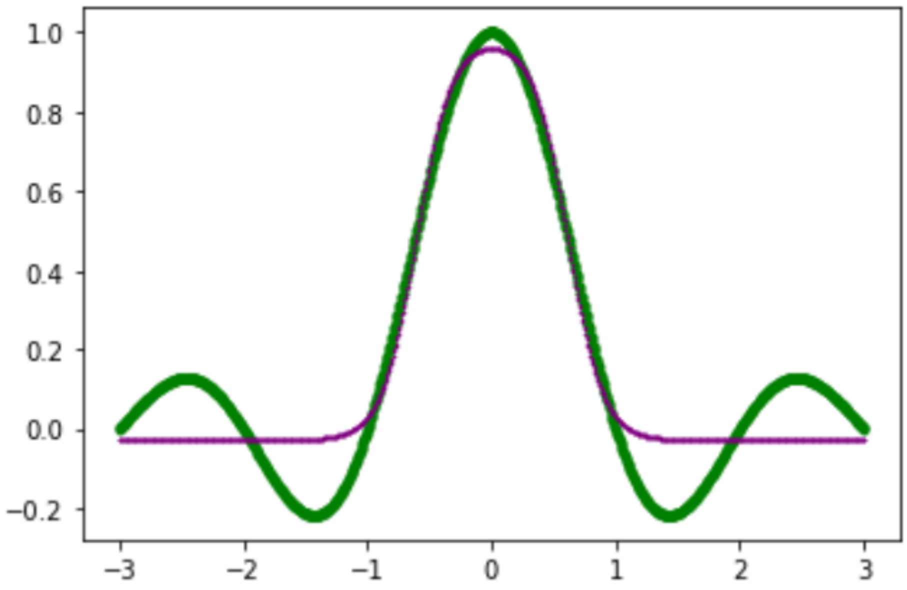
3.1 Exercise a

Fit the MLP-model to the sinc-function data. Here we are using the hidden layer sizes parameter to modify the amount of nodes in the hidden layer.



```
#%% data 2 - Til øvelserne..

x = np.linspace(-3,3,1000)
y = np.sinc(x)

plt.plot(x,y, '.', color='green')

x = x.reshape(-1,1) # Scikit-algoritmer kræver (:,1)-format

#%% fit model
mlp2 = MLPRegressor(activation='tanh',
                    hidden_layer_sizes = 2,
                    alpha = 1e-5,
                    solver = 'lbfgs',
                    max_iter=1000,
                    verbose = True)

mlp2.fit(x,y)

plt.plot(x, mlp2.predict(x), 'rx', ms=1, color='purple')
```

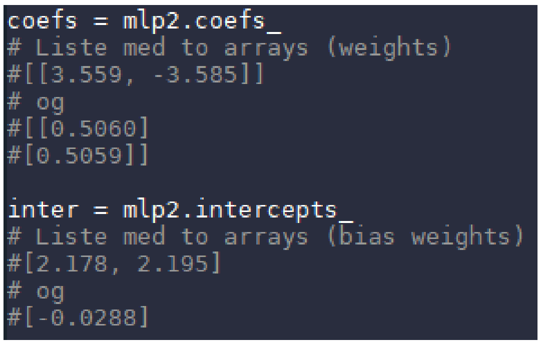
Figur 21: Generating the MLP-model, fitting it to the data. Plotting its prediction on the same data.



Figur 22:
MLPRegressor prediction on top of true values.
X-axis is input value, Y-axis is output value.
In green: The true output of the sinc(x)-function.
In purple: The fitted model prediction on the same input.

3.2 Exercise b

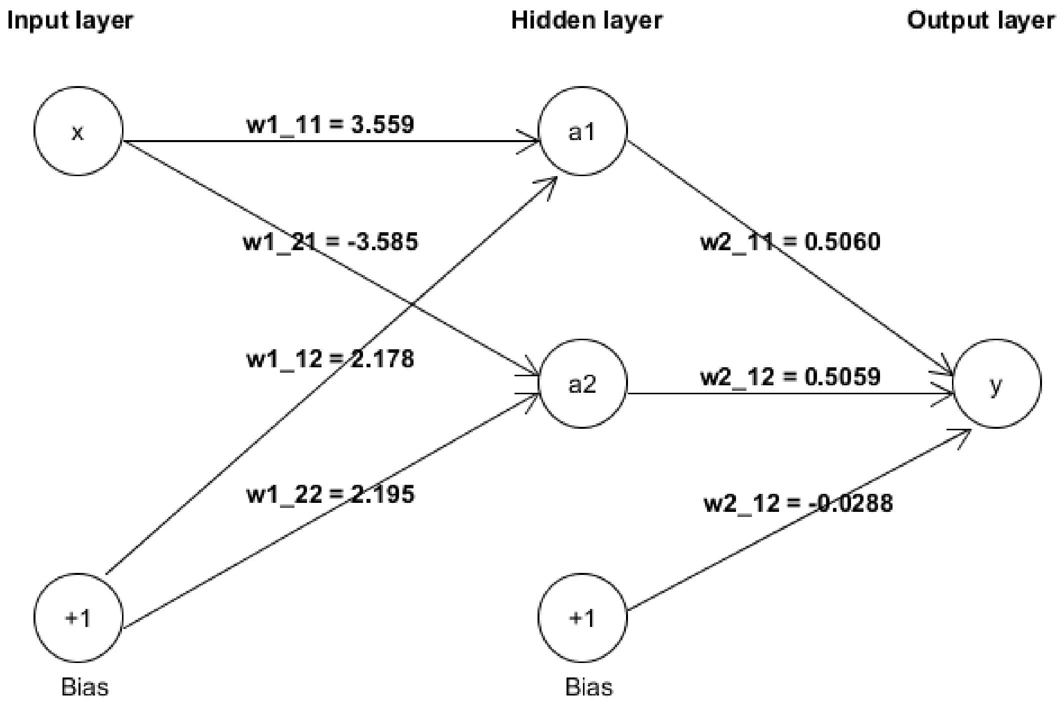
Here, we need to draw a graphical model of the neural network. To access the weights and bias of each element, the .coefs and .intercepts functions are called. This gives lists of the weights and bias of the calculation. These values have been noted as comments in the code below as well as shown as part of the graphic representation of the network.



```
coefs = mlp2.coefs_
# Liste med to arrays (weights)
#[[3.559, -3.585]]
# og
#[[0.5060]
#[0.5059]]

inter = mlp2.intercepts_
# Liste med to arrays (bias weights)
#[2.178, 2.195]
# og
#[-0.0288]
```

Figur 23: Using the .coefs and .intercepts functions to access weight and bias. The results are shown in the IDE, but have been copied into the comment below each function.

Weights are given as list of arrays, in the order that they appear, one layer at a time. Here we only have one hidden layer, therefore two arrays of weights. This becomes obvious when represented graphically. There's one bias per layer of calculations, which results in another list of which ultimately boils down to two values. Below these values are added to the various connections in the graph.

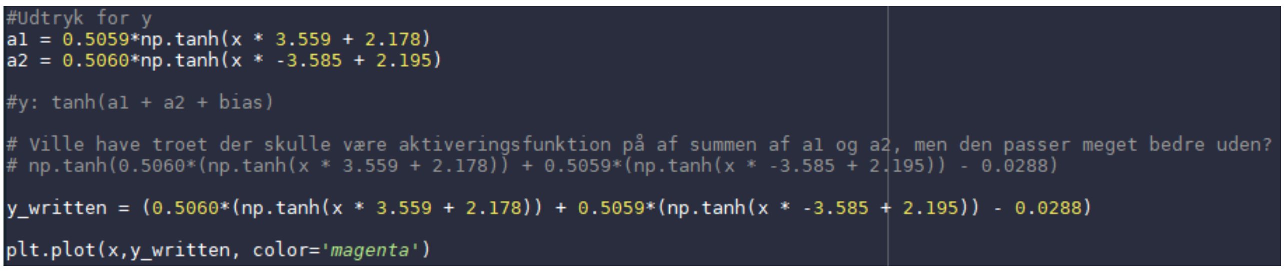


Figur 24: Graphical representation of the neural network. Weights and bias from the .coefs and .intercepts have been added to the connections between the nodes.

3.3 Exercise c

In this part of the exercise, we're writing out the expression for the output y, generated by the MLPRegressor. We can reverse engineer this formula based on the weights and biases.

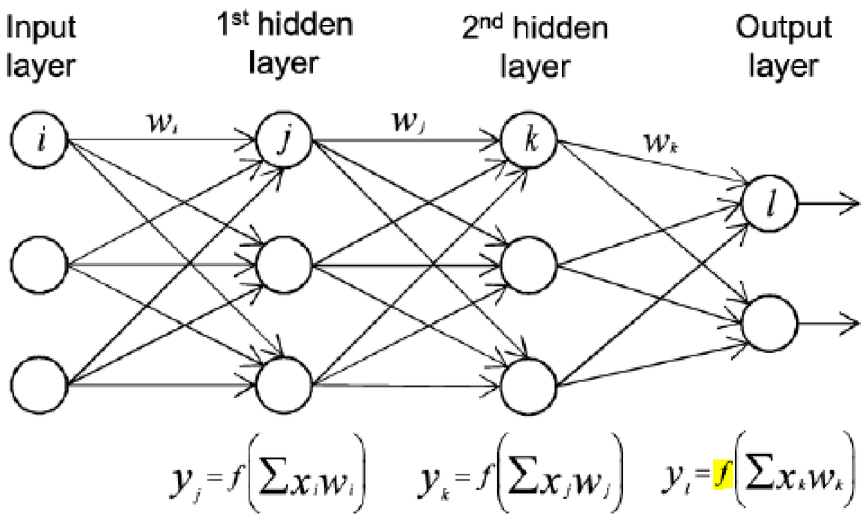
To simplify, we started out by writing down the calculations for the value of the nodes on the hidden layer, a1 and a2. These we're summed up and the bias was applied. In our original function that can be seen in the comments, the activation-function was also used from the hidden-layer to the output-layer. However, that expression did not match the MLP-output.



```
#Udtryk for y
a1 = 0.5059*np.tanh(x * 3.559 + 2.178)
a2 = 0.5060*np.tanh(x * -3.585 + 2.195)

#y: tanh(a1 + a2 + bias)

# Ville have troet der skulle være aktiveringsfunktion på af summen af a1 og a2, men den passer meget bedre uden?
# np.tanh(0.5060*(np.tanh(x * 3.559 + 2.178)) + 0.5059*(np.tanh(x * -3.585 + 2.195)) - 0.0288)

y_written = (0.5060*(np.tanh(x * 3.559 + 2.178)) + 0.5059*(np.tanh(x * -3.585 + 2.195)) - 0.0288)

plt.plot(x,y_written, color='magenta')
```

Figur 25: Reverse engineering the expression for prediction values of y based on x.

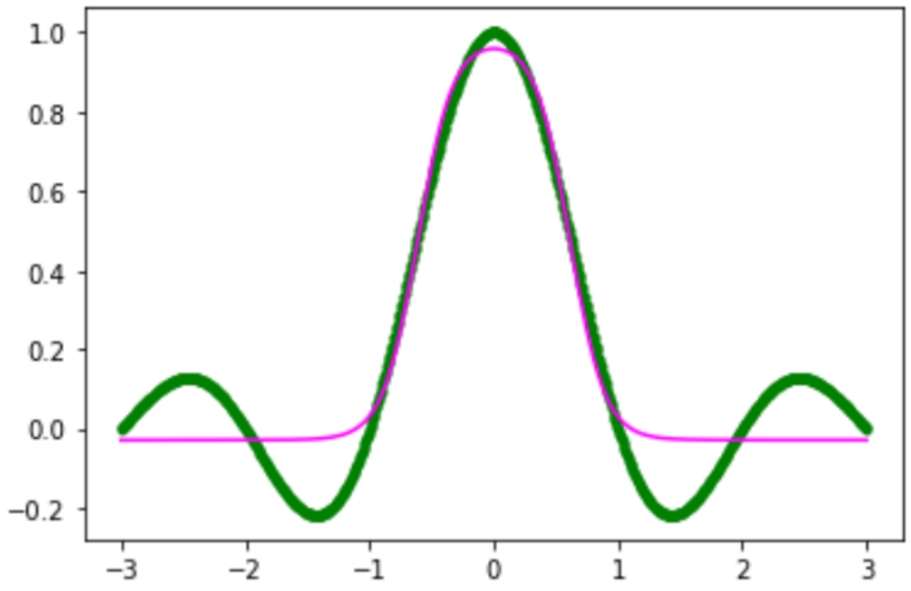
With some experimentation, we found that removing the activation-function on the final summation made the outputs match. This is still a bit of a mystery to us, as this does not agree with the following information on neural networks we were provided, as seen below.


Input layer
1st hidden layer
2nd hidden layer
Output layer
i
j
k
l
w_i
w_j
w_k
$y_j = f\left(\sum x_i w_i\right)$
$y_k = f\left(\sum x_j w_j\right)$
$y_l = f\left(\sum x_k w_k\right)$

Figur 26: "Linear and logistic regression.pdf", information provided as part of the lecture. Highlighted is the activation function used on the summation of the nodes in the final hidden layer, which does not match our final written expression. The expression that correctly follows the expressions shown in this figure can be seen on figure 25 as a comment. However, as previously mentioned, this did not fit the output of the MLP.

3.4 Exercise d

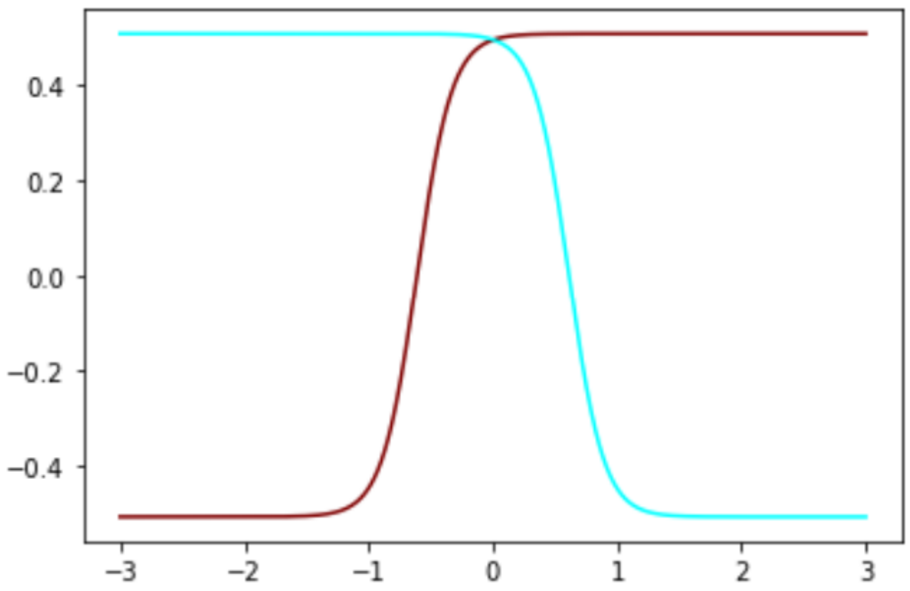
The expression from exercise c is now plotted. As can be seen it, it matches the MLP-output on figure 22.



Figur 27: The reverse engineered expression plotted. X-axis is input value, Y-axis is output value. Green is the true output, while pink is the reverse engineered function.

3.5 Exercise e

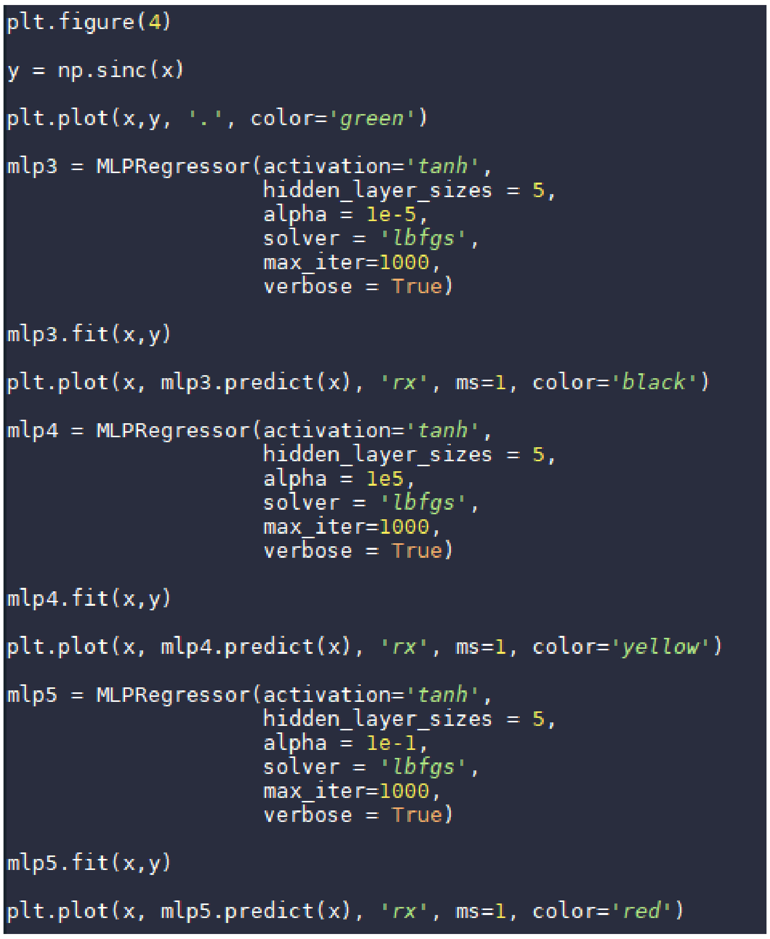
The expressions for the first parts of the function is now plotted. These are a1 and a2 that is also seen on figure 25. It's obvious that the sum of these result in the final function. Notice that the y-axis goes from about -0.5 to 0.5, meaning they alost cancel eachother out as you move away from x = 0, while resulting in 1 at around x = 0.



Figur 28: The reverse engineered expression plotted. X-axis is input value, Y-axis is output value. a1 is maroon, a2 is aqua-blue.

3.6 Exercise f g

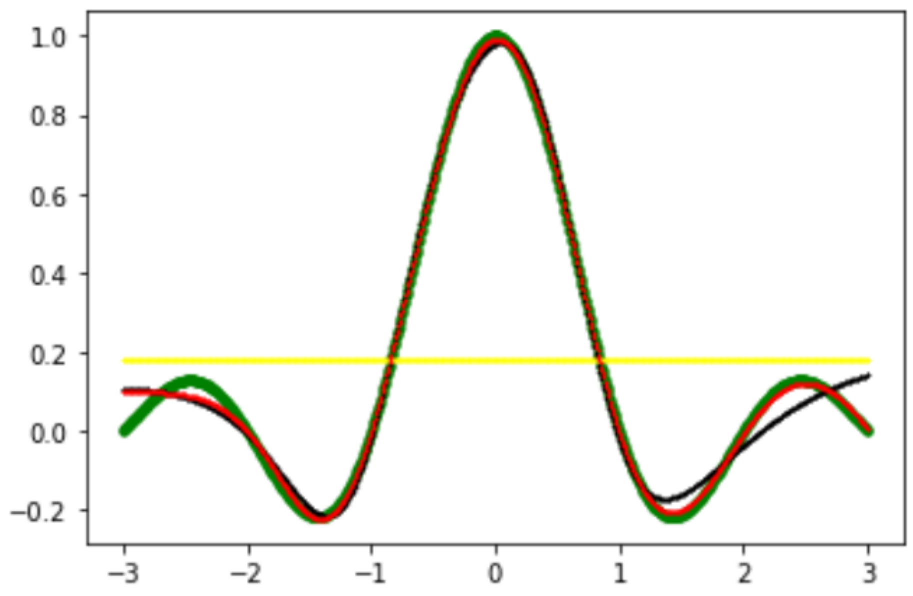
In this exercise, the effect of a different amount of nodes in the hidden layer is observed as well as different alpha values. On figure 29 the different MLPRegressor parameter choices can be seen.



```python
plt.figure(4)

y = np.sinc(x)

plt.plot(x,y, '.', color='green')

mlp3 = MLPRegressor(activation='tanh',
                    hidden_layer_sizes = 5,
                    alpha = 1e-5,
                    solver = 'lbfgs',
                    max_iter=1000,
                    verbose = True)

mlp3.fit(x,y)

plt.plot(x, mlp3.predict(x), 'rx', ms=1, color='black')

mlp4 = MLPRegressor(activation='tanh',
                    hidden_layer_sizes = 5,
                    alpha = 1e5,
                    solver = 'lbfgs',
                    max_iter=1000,
                    verbose = True)

mlp4.fit(x,y)

plt.plot(x, mlp4.predict(x), 'rx', ms=1, color='yellow')

mlp5 = MLPRegressor(activation='tanh',
                    hidden_layer_sizes = 5,
                    alpha = 1e-1,
                    solver = 'lbfgs',
                    max_iter=1000,
                    verbose = True)

mlp5.fit(x,y)

plt.plot(x, mlp5.predict(x), 'rx', ms=1, color='red')
```

Figur 29: Code showing the various MLPRegressor-setups.

The output from the various models fitted on the same input data can be seen on figure 30.



Figur 30: Green is the true output of sinc-function. Black is the same MLPRegressor setup as in exercise a to e, but with a layer size of 5. Yellow is the same as black, but with a much higher alpha value, as recommended in the exercise. Red having an alpha value in between black and yellow. X-axis is the input, while Y-axis is the output.

The black function shows that a larger hidden layer with more neurons allows the function to have additional curvature at the edges, as more function can be added together for a more complex result.

The red function with a higher alpha value seems to fit even better. This was not the expected result. The alpha value determines the regularization of the regressor, smoothing the function out to avoid overfitting. We expected a higher alpha value to create a less volatile function, not the opposite.

The yellow function, however, looks as expected. The extremely high alpha value, means the function is extremely regularized. So regularized in fact, it's nothing but a straight line.

4 Data analysis of own project

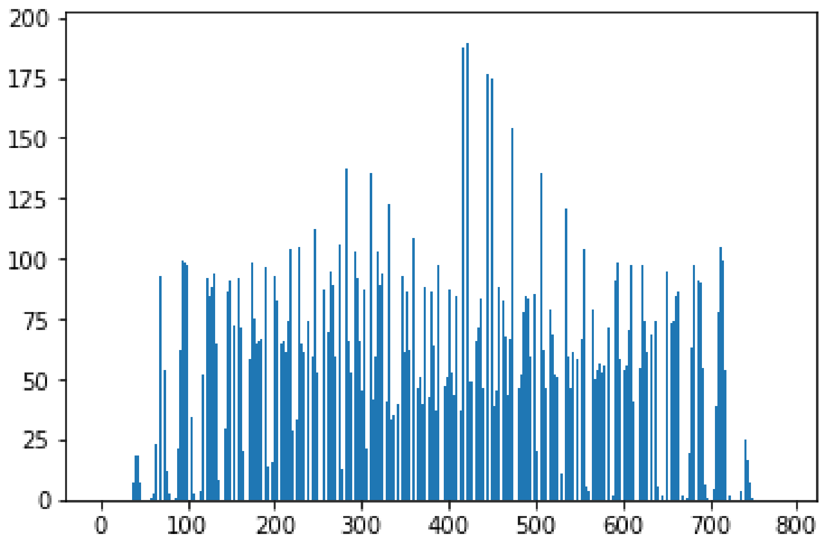
This section is about analysing grayscale image files, and how it should be approached.

4.1

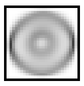
Firstly load the data set seen in the code below.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
%matplotlib inline

# read data file
path = os.path.join("./data")
try:
    donuts = np.load(path + "/donut.npy")
    houses = np.load(path + "/house.npy")
except Exception as e:
    print("error")
    raise e
```

Next, to find out how to approach looking at the data, the first way to look at the data was ultimately scraped, as it did not gather any meaningful data. The first approach was to look at each indivual image, and looking at the different pixel values. When doing this medians of each picture would always be 0, as the image was always mostly white. Instead, looking at each individual pixel for all images, and finding the mean value, would yield what each pixels average value would be. Looking at figure 31 does not really provide much information, however converting this to an image shows how the average donut image looks. Which means that we can now compare a single donut image to how the average donut looks like.

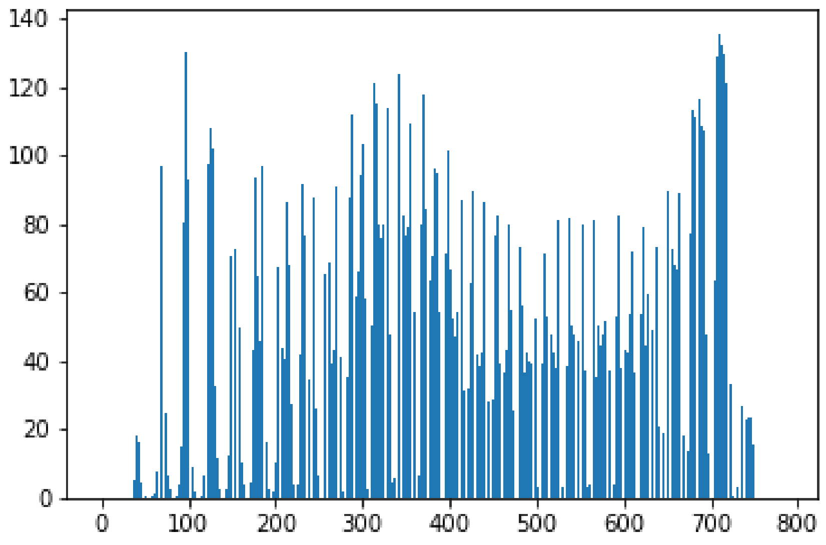


Figur 31: Plot of the average pixel value for all donuts



Figur 32: Average donut image

However this data would not be meaningful unless we ensure that it is different from the other data set of houses. Looking at the average house graph on figure 33 shows us that it does look different from the average donut. This can help classify that a house is not a donut and vice versa.



Figur 33: Average pixel value for a house drawing