Polynomial Regression

Dr Rob Collins

Version 6, 17th August 2021

Introduction

In this workshop we will extend our previous work on Regression. In this session we will review 'polynomial regression' - that is, functions of the general form:

$y =\theta_0.x_0 + \theta_1.x_1 + \theta_2.x_1^2 + \theta_3.x_1^3 + ... + \theta_n.x_1^n$

where, as previously we set: $x_0 = 1$

The above general equation is a 'polynomial' (it has squared, cubed and fourth-power terms etc.) but it is only in one dimension. Or, to put it another way, in a data-set we were using to build the model, there would only be one feature (that is, one column). To make polynomial models even more general, they need to operate across multiple dimensions. Not only that, but they need to be capable of representing 'interactions' between dimensions.

Thus, for a second-order, polynomial model in two dimensions $x_1 and x_2$ we would see:

$y =\theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \theta_3.x_1^2 + \theta_4.x_2^2 + \theta_5.x_1.x_2$

Where the final '$\theta_5.x_1.x_2$' term enables us to model any interaction between $x_1 and x_2$.

The same logic holds in higher dimensions, and thus you should expect to see a model based on not only powers of each feature, but also terms relating to all permutations of features! This may sound horribly complicated! How on earth would somebody build a model with all of those complicated terms?

Inevitably, there is an easy answer. The answer, in some ways feels like a 'trick' to me - but is actually quite obvious when you see it. Not only that, but library functions within the Machine Learning libraries enable this type of model to be built quite easily.

The data-set used in this workshop is the "Boston House Price" data. This is a popular data-set for Machine Learning tutorials. It is widely available on the Internet and also built into sklean as one of its demonstration data-sets.

Instructions for Students

This workbook includes some empty code-blocks. In those cases you are required to create the code for the block based on your previous learning on this course. This activity is indicated in the workbook below with a '**Student Task**' indicator.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

1 Load the data-set

As always we start by import the required libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Specifically, in this case we need to load the Boston Housing dataset from sklean

In [2]:
```
boston = pd.read_csv("boston.csv")
boston
```

Out[2]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	396.90	7.88	11.9

506 rows × 14 columns

2 Review the data quantitatively

It is often useful to review a brief summary of the data. The .describe() function of Pandas provides summary data for the dataframe, including:

- count : The number of items in the dataframe
- mean : The mean or 'average' value .. sum of values / number of values
- std : Standard deviation - a measure of spread in the data
- min : The smallest value
- 25%, 50%, 75% : Quartile values
- max : The largest value

In [3]:
```
boston.describe()
```

Out[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391
75%	3.677082	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396

As in previous workshops, we should review the data to see if there are any missing values. In this case, we introduce a new and useful function that reviews each column and indicates if it has any null elements:

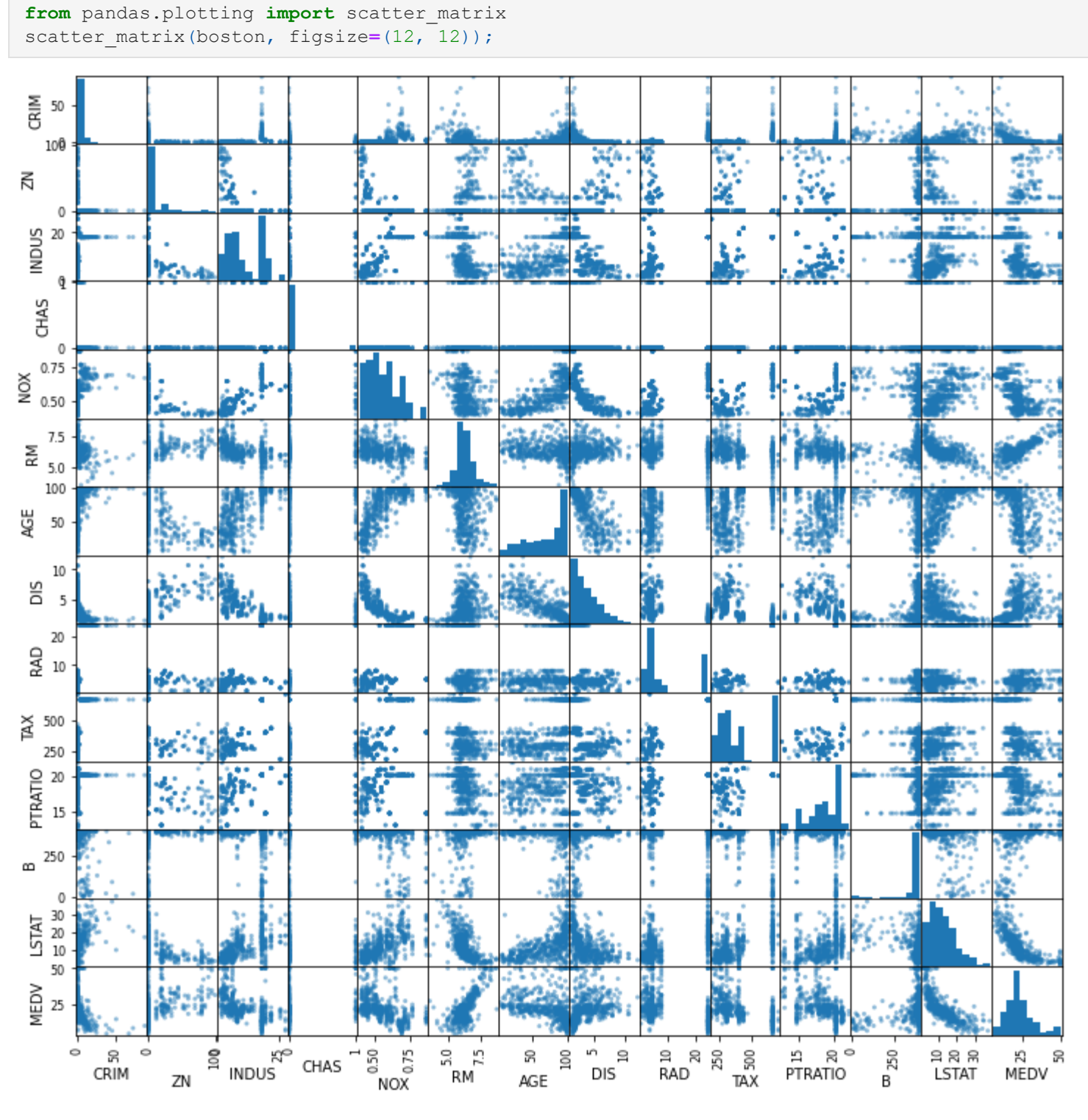
In [4]:
```
pd.isnull(boston).any()
```

Out[4]:
```
CRIM       False
ZN         False
INDUS      False
CHAS       False
NOX        False
RM         False
AGE        False
DIS        False
RAD        False
TAX        False
PTRATIO    False
B          False
LSTAT      False
MEDV       False
dtype: bool
```

3. Review the data visually

We can start with the Pandas 'scatter_matrix' function we have used before to review all of the features and the correlations between them:

In [5]:
```
from pandas.plotting import scatter_matrix
scatter_matrix(boston, figsize=(12, 12));
```

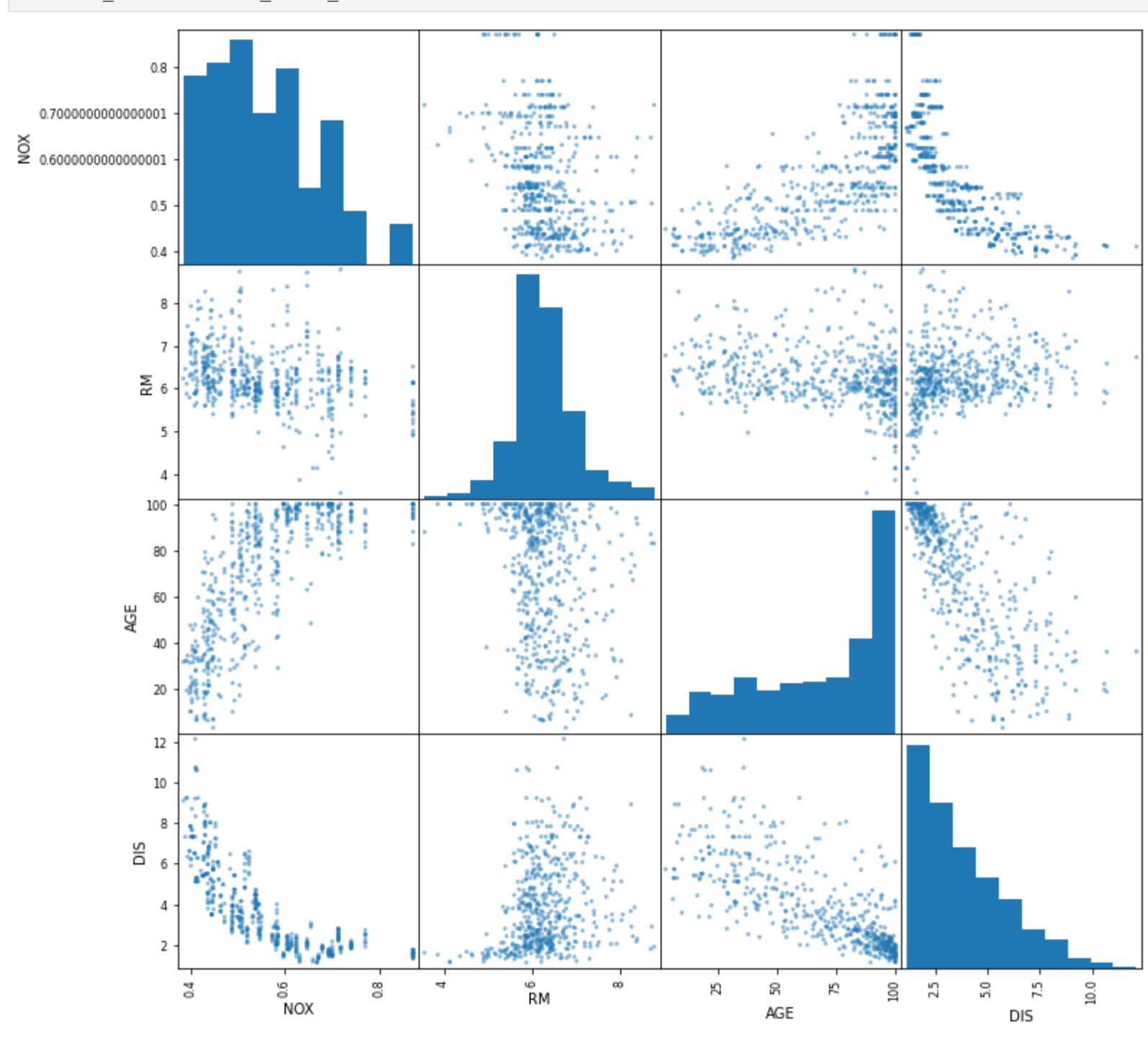


This chart is rather large and complex to read .. therefore it may be better to review only part the data-set.

Student task : Experiment with the code-block below to plot different parts of the data-set.

In [6]:
```
boston_fewer_columns = boston.iloc[:, 4:8:1]
          # 'iloc' indexes dataframes with integer numbers
          # The first part .. [:, ... means "Every Row"
          # The second part .. 4:8:1 .. means
          #  "Every column between columns 4 and less than 8 in steps of 1"

scatter_matrix(boston_fewer_columns, figsize=(12, 12));
```

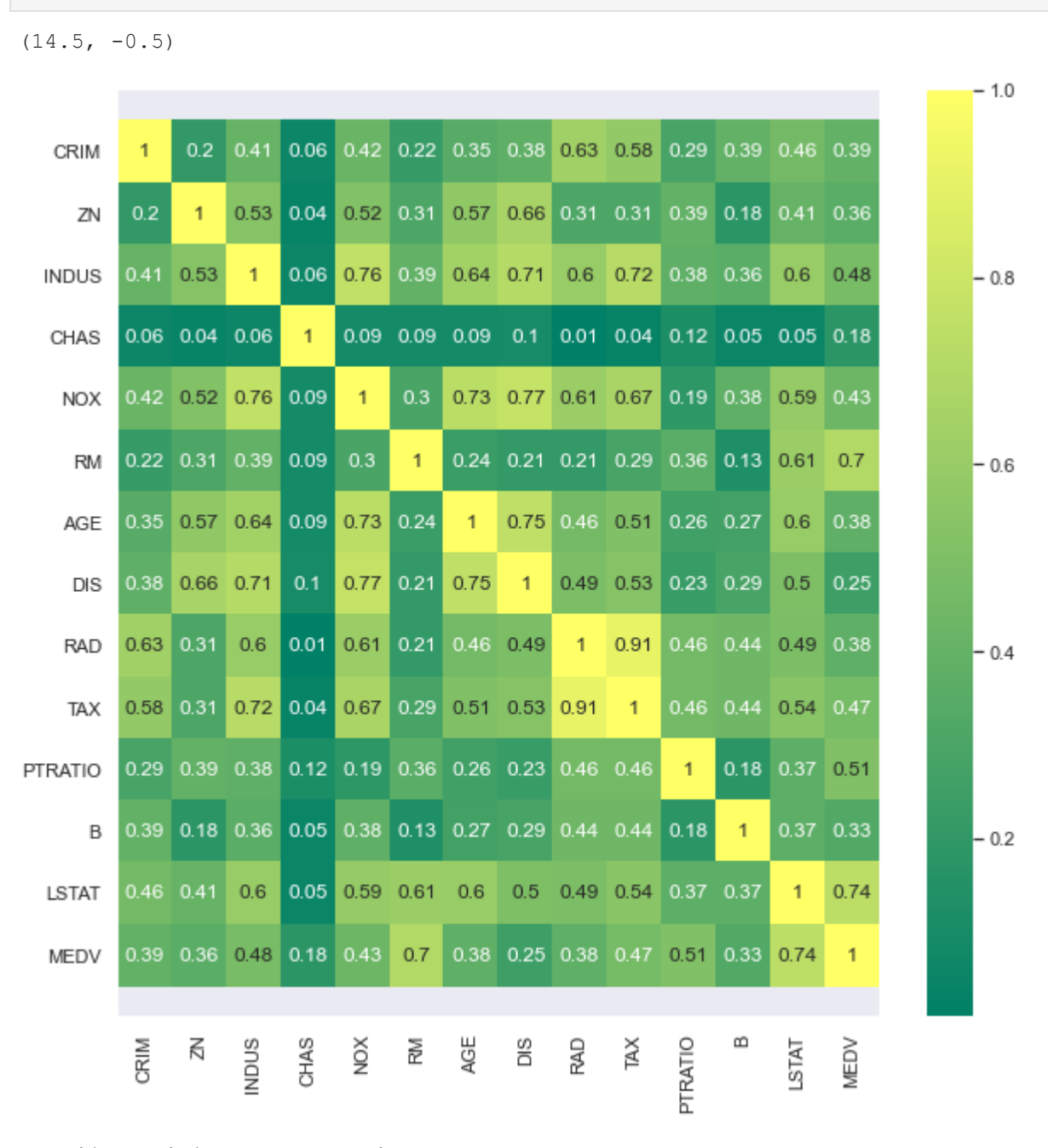


This workshop is focussed on regression. We will want to work out which of the features can be used to predict house prices. A useful clue is thus the correlation between the variables. In particular, which of the variables is correlated with the MEDV label?

Student task : Just for fun, you can experiment with the way that the following heat map is displayed. The 'cmap' option in the 'heatmap' function defines the colours that are used for the plot. Many different sets of colours are available and these are documented here: Other colour map options at https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html. Experiment with some of the other possible cmap pallets and decide which one you prefer.

In [7]:
```
%matplotlib inline
correlation_matrix = np.absolute(boston.corr().round(2))
sns.set(rc={'figure.figsize':(10,10)})
ax = sns.heatmap(correlation_matrix, annot=True, cmap='summer')
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
```

Out[7]: (14.5, -0.5)



From this correlation map, we see that:

- MEDV is strongly correlated with LSTAT and RM
- RAD and TAX are correlatated with each other Therefore, LSTAT and RM and good candidates for building our model, whearas we should not use both RAD and TAX since we will be 'double counting' their impact on the result

It is interesting to review those two key features (LSTAT and RM) in a 3D plot. Here, the vertical dimension represents MEDV. I have also colour-coded the points based on MEDV.

Student Task 1 : Rotate the 3D view and look at the 'shape' of the data. This is the thing we are going to try to model. It is clearly multi-dimensional and also appears to be non-linear. It will thus require a polynomial model.

Student Task 2 : Again, just for fun, experiment with the cmap values to create a visualisation that you prefer.

In [8]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Uncomment next  line if you want the graph to appear in the page
# %matplotlib inline

# Uncomment next line if you want a 3D pop-out
%matplotlib auto

fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(boston['RM'],boston['LSTAT'],boston['MEDV'],c=boston['MEDV'], cmap ='gist_heat' )
```

```
Using matplotlib backend: Qt5Agg
```

Out[8]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1cd4da848b0>

4 Selecting the data we wish to use for modelling

So we can build our final data-frame based on the key features we have selected:

In [9]:
```
X = pd.DataFrame(np.c_[boston['LSTAT'], boston['RM']], columns = ['LSTAT','RM'])
Y = boston['MEDV']
X
```

Out[9]:

	LSTAT	RM
0	4.98	6.575
1	9.14	6.421
2	4.03	7.185
3	2.94	6.998
4	5.33	7.147
...	...	...
501	9.67	6.593
502	9.08	6.120
503	5.64	6.976
504	6.48	6.794
505	7.88	6.030

506 rows × 2 columns

5 Splitting data for training and testing

We want to be able to test how good our model is for prediction. In particular, we need to be assured that is is not over-fitting the data. So we split the model into training and a testing sub-sets

In [11]:
```
from sklearn.model_selection import train_test_split
```

In [12]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2)
print(X_train[0:5])
```

```
     LSTAT     RM
299   4.74  7.041
341   5.49  7.241
239   7.37  6.606
477  24.91  5.304
184  13.98  5.604
```

6. First model - linear regression

To give us a baseline to judge our polynomial regression model we first generate a linear model and score it for model quality

6.1 Build the linear regression model

In [13]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

lin_model = LinearRegression()
lin_model.fit(X_train, Y_train)
```

Out[13]: LinearRegression()

6.2 Test the linear regression model

The sklearn library provides a range of functions for scoring different types of models: https://scikit-learn.org/stable/modules/model_evaluation.html

Two useful and common quality measure for regression are:

- The R^2 score or 'Coefficient of Determination' (See https://en.wikipedia.org/wiki/Coefficient_of_determination)
- The 'root mean squared error' or RMSE

Both of these can easily be computed using sklearn functions as follows.

It is interesting to compare how well the model fitted to the training data as compared with the performance of the model on test data (which the model has not 'seen'). So we will compute these scores first for the training data and then for the test data:

In [14]:
```
y_train_predict = lin_model.predict(X_train)
r2_train = r2_score(Y_train, y_train_predict)
rmse_train = (np.sqrt(mean_squared_error(Y_train, y_train_predict)))
```

Now the same thing for the test data:

In [15]:
```
y_test_predict = lin_model.predict(X_test)
r2_test = r2_score(Y_test, y_test_predict)
rmse_test = (np.sqrt(mean_squared_error(Y_test, y_test_predict)))
```

And compare the results:

In [16]:
```
print("R2:")
print("  Train = ", r2_train)
print("  Test  = ", r2_test)
print("RMSE:")
print("  Train = ", rmse_train)
print("  Test  = ", rmse_test)
```

```
R2:
  Train =  0.6251061599561654
  Test  =  0.6849168837212041
RMSE:
  Train =  5.550444454223435
  Test  =  5.419760456591725
```

As we might expect, the model fits the training data somewhat better than it performs when seeing new data. That is, for the training set the R2 value is closer to 1 (better) and the RMSE is lower (better).

7 Second model - Polynomial regression

Now comes the 'trick' that enables us to easily generate a polynomial model for given data.

sklearn provides a function that generates an expanded data-set. It creates new features that are combinations of the other features. So that if we have a feature 'x' and we use the function to generate Polynomial Features of degree 3, then the function will add new data columns (features) for x^2 and x^3.

Even better, if we have two features .. say 'x' and 'z', and we use the function to generate Polynomial Features, for example, order 2, then it will generate new features for x^2, z^23 and $x.z$ (the last of these being the interaction term described in the first section above.

So let's experiment with the sklearn.PolynomialFeatures function and generate an expanded data-set for our housing price variables in X_train.

We first generate the features and take a look at the extended set of feature names:

In [17]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=5)
X_train_poly = poly_features.fit_transform(X_train)
print(poly_features.get_feature_names(["LSTAT","RM"]))
```

```
['1', 'LSTAT', 'RM', 'LSTAT^2', 'LSTAT RM', 'RM^2', 'LSTAT^3', 'LSTAT^2 RM', 'LSTAT RM^2', 'RM^3', 'LSTAT^4',
'LSTAT^3 RM', 'LSTAT^2 RM^2', 'LSTAT RM^3', 'RM^4', 'LSTAT^5', 'LSTAT^4 RM', 'LSTAT^3 RM^2', 'LSTAT^2 RM^3', 'L
STAT RM^4', 'RM^5']
```

```
C:\Anaconda3\envs\ML_Course_2023\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: Function get
_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please use get
_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

We can also look at the actual data that has been produced:

In [18]:
```
X_train_poly[0:5]
```

Out[18]:
```
array([[1.00000000e+00, 4.74000000e+00, 7.04100000e+00, 2.24676000e+01,
        3.33743400e+01, 4.95756810e+01, 1.06496424e+02, 1.58194372e+02,
        2.34988728e+02, 3.49062370e+02, 5.04793050e+02, 7.49841321e+02,
        1.11384657e+03, 1.65455563e+03, 2.45774815e+03, 2.39271906e+03,
        3.55424786e+03, 5.27963274e+03, 7.84259370e+03, 1.16497262e+04,
        1.73050047e+04],
       [1.00000000e+00, 5.49000000e+00, 7.24100000e+00, 3.01401000e+01,
        3.97530900e+01, 5.24320810e+01, 1.65469149e+02, 2.18244464e+02,
        2.87852125e+02, 3.79660699e+02, 9.08425628e+02, 1.19816211e+03,
        1.58030816e+03, 2.08433723e+03, 2.74912312e+03, 4.98725670e+03,
        6.57790997e+03, 8.67589182e+03, 1.14430114e+04, 1.50926859e+04,
        1.99064005e+04],
       [1.00000000e+00, 7.37000000e+00, 6.60600000e+00, 5.43169000e+01,
        4.86862200e+01, 4.36392360e+01, 4.00315553e+02, 3.58817441e+02,
        3.21621169e+02, 2.88280793e+02, 2.95032563e+03, 2.64448454e+03,
        2.37034802e+03, 2.12462944e+03, 1.90438292e+03, 2.17438999e+04,
        1.94898511e+04, 1.74694649e+04, 1.56585190e+04, 1.40353021e+04,
        1.25803536e+04],
       [1.00000000e+00, 2.49100000e+01, 5.30400000e+00, 6.20508100e+02,
        1.32122640e+02, 2.81324160e+01, 1.54568568e+04, 3.29117496e+03,
        7.00778483e+02, 1.49214334e+02, 3.85030302e+05, 8.19831683e+04,
        1.74563920e+04, 3.71692907e+03, 7.91432830e+02, 9.59110483e+06,
        2.04220072e+06, 4.34838725e+05, 9.25887032e+04, 1.97145918e+04,
        4.19775973e+03],
       [1.00000000e+00, 1.39800000e+01, 5.60400000e+00, 1.95440400e+02,
        7.83439200e+01, 3.14048160e+01, 2.73225679e+03, 1.09524800e+03,
        4.39039328e+02, 1.75992589e+02, 3.81969500e+04, 1.53115671e+04,
        6.13776980e+03, 2.46037639e+03, 9.86262468e+02, 5.33993360e+05,
        2.14055708e+05, 8.58060218e+04, 3.43960620e+04, 1.37879493e+04,
        5.52701487e+03]])
```

Notice that, rather conveniently, the PolynomialFeatures function has also added a column of '1's to the data. This follows the pattern introduced in the last session for dealing with the constant term.

Having generated an extended set of polynomial features, we can now simply apply a linear regression model to the new extended set! This effectively gives us terms representing polynomial versions of the features and hence is a polynomial model.

The beauty of this is that we can simply re-use our linear model fitting tools.

In [19]:
```
polynomial_model = LinearRegression()
polynomial_model.fit(X_train_poly, Y_train)
print("Model coefficients    = " , polynomial_model.coef_)
print("Constant term (bias) = ", polynomial_model.intercept_)
```

```
Model coefficients    =  [ 0.00000000e+00  2.05167097e+02  3.22343302e+03  6.30140080e+00
 -1.97110837e+02 -9.95387396e+02 -2.90487151e-01 -4.63452054e-01
  5.61747192e+01  1.42985082e+02  4.61513962e-03  4.24963759e-02
 -1.45200906e-01 -6.48105821e+00 -9.60295658e+00 -2.85063011e-05
 -3.33985144e-04 -1.47902040e-03  1.42858517e-02  2.66142832e-01
  2.39876666e-01]
Constant term (bias) =  -3760.603906194849
```

As previously, we can apply this model to both the original training data and to the test data:

In [20]:
```
y_train_predicted = polynomial_model.predict(X_train_poly)
y_test_predict = polynomial_model.predict(poly_features.fit_transform(X_test))
```

Then measure the model quality for each of these cases:

In [21]:
```
rmse_train = np.sqrt(mean_squared_error(Y_train, y_train_predicted))
r2_train = r2_score(Y_train, y_train_predicted)

rmse_test = np.sqrt(mean_squared_error(Y_test, y_test_predict))
r2_test = r2_score(Y_test, y_test_predict)

print("R2:")
print("  Train = ", r2_train)
print("  Test  = ", r2_test)
print("RMSE:")
print("  Train = ", rmse_train)
print("  Test  = ", rmse_test)
```

```
R2:
  Train =  0.821327548704215
  Test  =  0.7124062736909332
RMSE:
  Train =  3.831796235808422
  Test  =  5.177942577150979
```

Notice that both the R2 and RMSE values are better in the case of the polynomial model than they were above in the linear model.

But there is also some evidence here that the model is moving towards over-fitting the data. The two scores are very much improved for the training data (they indicate a good fit) but only improved by a small amount for the testing data.

In other words, the second-order polynomial model is better than the original model - but not as much better as we might think without careful testing with new data!

Student Task : Experiment by generating models of higher-degree polynomials. That is, in the first code block of section 7 above try substituting different values for the 'degree' in the line:

```
poly_features = PolynomialFeatures(degree=2)
```

Observe the behaviour and determine at which degree the model starts to over-fit the data. That is, the polynomial degree at which the model actually starts to perform less well with new test data.

In [ ]:

In [ ]: