CAP5768_Assignment6_Corbin_Adam

November 15, 2019

1 CAP 5768 - Data Science - Dr. Marques - Fall 2019

1.1 Assignment 6: Regression Analysis

1.2 Starter code

1.2.1 Goals

- To learn how to use perform linear regression by least squares using Python and scikit-learn.
- To appreciate that the same linear regression coefficients may be the best fit for dramatically different data distributions – as illustrated by the Anscombe's quartet.
- To practice with different types of regularization (*lasso* and *ridge*) and understand when to use them.
- To expand upon the prior experience of manipulating, summarizing, and visualizing small datasets.
- To increase our statistical analysis skills.

1.2.2 Instructions

- This assignment is structured in 4 parts.
- As usual, there will be some Python code to be written and questions to be answered.
- At the end, you should export your notebook to PDF format; it will become your report.
- Submit the report (PDF), notebook (.ipynb file), and (optionally) link to the "live" version of your solution on Google Colaboratory via Canvas.
- The total number of points is 126 (plus up to 60 bonus points).

1.2.3 Important

- It is OK to attempt the bonus points, but please **do not overdo it!**

[150]:
```
#Imports
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
import seaborn as sns; sns.set()
import scipy.stats as ss
```

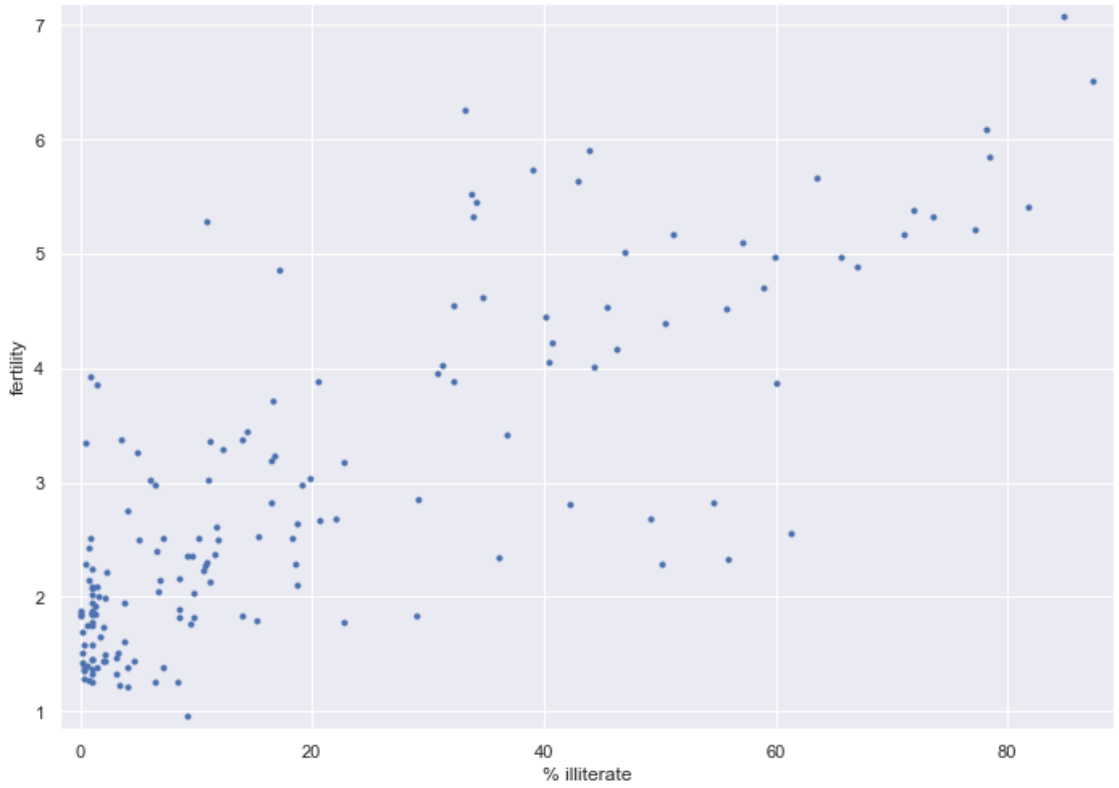
1.3 Part 1: Linear regression by least squares

In this part, we will take another look at the correlation between female literacy and fertility (defined as the average number of children born per woman) throughout the world. For ease of analysis and interpretation, we will work with the *illiteracy* rate.

The Python code below plots the fertility versus illiteracy and computes the Pearson correlation coefficient. The Numpy array `illiteracy` has the illiteracy rate among females for most of the world's nations. The array `fertility` has the corresponding fertility data.

[270]:
```
df = pd.read_csv('data/female_literacy_fertility.csv')
print(df.describe())

illiteracy = 100 - df['female literacy']

fertility = df['fertility']

def pearson_r(x, y):
    """Compute Pearson correlation coefficient between two arrays."""
    # Compute correlation matrix: corr_mat
    corr_mat = np.corrcoef(x, y)

    # Return entry [0,1]
    return corr_mat[0,1]

# Plot the illiteracy rate versus fertility
_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')

# Set the margins and label axes
plt.margins(0.02)
_ = plt.xlabel('% illiterate')
_ = plt.ylabel('fertility')

# Show the plot
plt.show()

# Show the Pearson correlation coefficient
print('Pearson correlation coefficient between illiteracy and fertility: {:.
 ↪5f}'.format(pearson_r(illiteracy, fertility)))
```

```
       female literacy   fertility
count       162.000000  162.000000
mean         80.107407    2.878673
std          23.052415    1.427597
min          12.600000    0.966000
25%          66.425000    1.823250
50%          90.000000    2.367500
75%          98.500000    3.880250
max         100.000000    7.069000
```




```
Pearson correlation coefficient between illiteracy and fertility: 0.80413
```

1.4 Your turn! (25 points)

We will assume that fertility is a linear function of the female illiteracy rate: `f=ai+b`, where `a` is the slope and `b` is the intercept.

We can think of the intercept as the minimal fertility rate, probably somewhere between one and two.

The slope tells us how the fertility rate varies with illiteracy. We can find the best fit line .

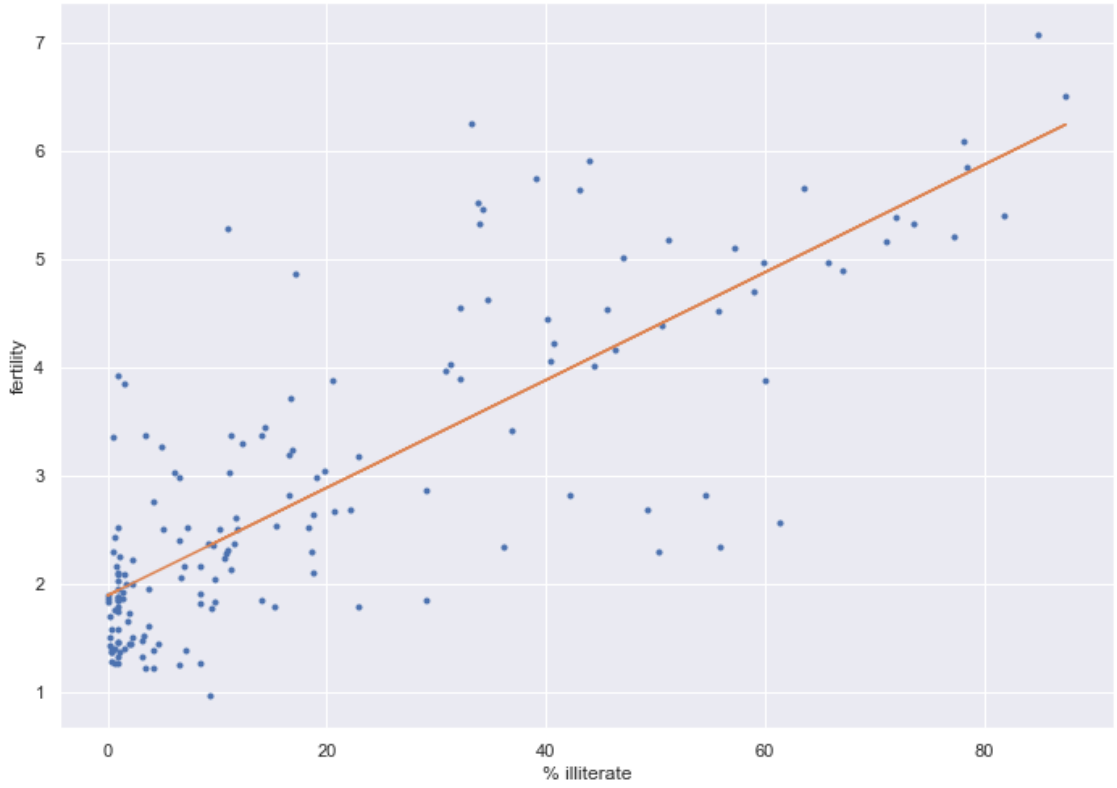
Write code to plot the data and the best fit line (using `np.polyfit()`) and print out the slope and intercept.

1.5 Solution

[271]:
```
m,b = np.polyfit(illiteracy,fertility,1)
print("Slope\t\t",m)
print("Slope intercept\t",b)
coef = np.polyfit(illiteracy,fertility,1)
poly1d_fn = np.poly1d(coef)
# poly1d_fn is now a function which takes in x and returns an estimate for y

_ = plt.xlabel('% illiterate')
```

```
_ = plt.ylabel('fertility')
_ = plt.plot(illiteracy,fertility, marker='.', linestyle='none')
_ = plt.plot(illiteracy, poly1d_fn(illiteracy))
```

```
Slope             0.04979854809063424
Slope intercept   1.8880506106365564
```



1.6 BONUS! (15 points)

The function `np.polyfit()` that you used above to get your regression parameters finds the optimal slope and intercept. It is optimizing the the *residual sum of squares (RSS)*, also known as the *sum of squared residuals (SSR)* or the *sum of squared estimate of errors (SSE)*, which can be defined as "the sum of the squares of residuals (deviations predicted from actual empirical values of data)." (see https://en.wikipedia.org/wiki/Residual_sum_of_squares)

Write code to plot the function that is being optimized, the RSS, versus the slope parameter `a`. To do this, fix the intercept (`b`) to be what you found in the optimization. Then, plot the RSS vs. the slope. Where is it minimal?

Hint: use a `for` loop to draw 100,000 permutation replicates and compute the Pearson correlation coefficient for each of them.

Your plot will probably look like this:

1.7 Solution

[289]:
```
inds = np.arange(0, len(illiteracy))

size = 1
slopes = []

bs_replicates = np.empty(len(illiteracy))
#BS is for bootstrap
for i in range(size):
    bs_inds = np.random.choice(inds,size=len(inds))

    bs_x = illiteracy[bs_inds]
#     print(bs_x)
#     print(illiteracy)
    m,b = np.polyfit(illiteracy,fertility,1)
    slopes.append(round(m,2))

slopes
#TODO- come back to this
```

[289]:
```
[0.05]
```

1.8 Part 2: Anscombe's quartet

The Anscombe's quartet is a collection of four small data sets that have nearly identical simple descriptive statistics, yet have very different distributions. Each dataset consists of 11 `(x,y)` points. The quartet was created in 1973 by the statistician Francis Anscombe to demonstrate: the importance of visualization and exploratory data analysis (EDA), the effect of outliers and other influential observations on statistical properties, and the limitations of summary statistics (*).

(*) See https://heap.io/blog/data-stories/anscombes-quartet-and-why-summary-statistics-dont-tell-the-whole-story if you're interested.

Anscombes_Quartet.png

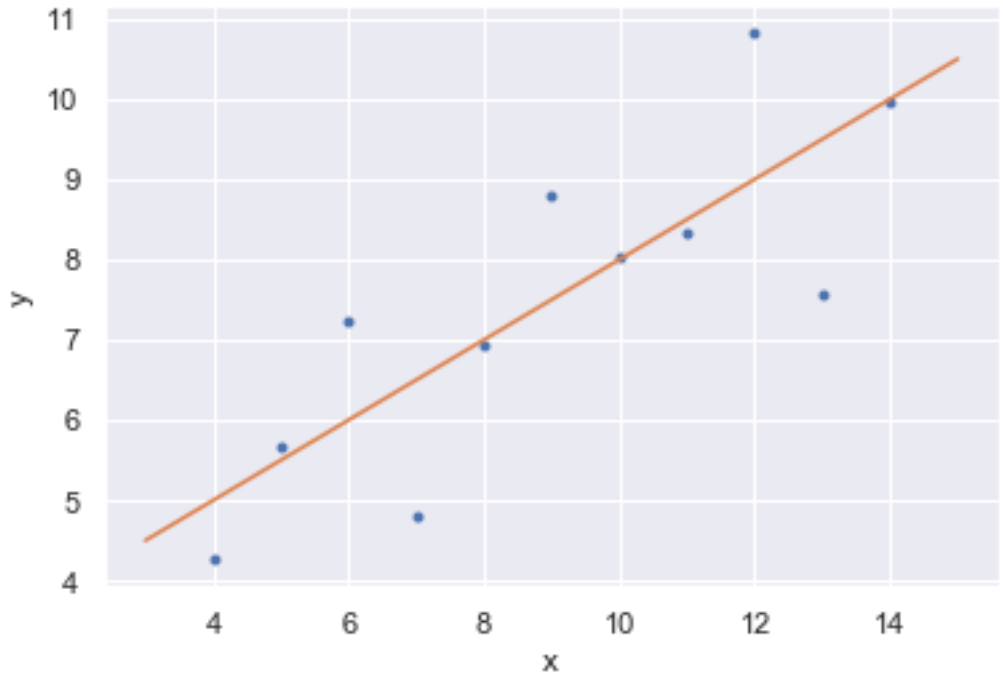
The Python code below performs a linear regression on the data set from Anscombe's quartet that is most reasonably interpreted with linear regression.

[154]:
```
x1 = [10.0, 8.0,  13.0,  9.0,  11.0, 14.0, 6.0,  4.0,  12.0,  7.0,  5.0]
y1 = [8.04, 6.95, 7.58,  8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68]

x2 = [10.0, 8.0,  13.0,  9.0,  11.0, 14.0, 6.0,  4.0,  12.0,  7.0,  5.0]
y2 = [9.14, 8.14, 8.74,  8.77, 9.26, 8.10, 6.13, 3.10, 9.13,  7.26, 4.74]

x3 = [10.0, 8.0,  13.0,  9.0,  11.0, 14.0, 6.0,  4.0,  12.0,  7.0,  5.0]
y3 = [7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15,  6.42, 5.73]
```

```
x4 = [8.0,  8.0,  8.0,   8.0,  8.0,  8.0,  8.0,  19.0,  8.0,  8.0,  8.0]
y4 = [6.58, 5.76, 7.71,  8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89]
```

[155]:
```
# Perform linear regression: a, b
a, b = np.polyfit(x1, y1, 1)

# Print the slope and intercept
print('slope =', a)
print('intercept =', b)

# Generate theoretical x and y data: x_theor, y_theor
x_theor = np.array([3, 15])
y_theor = a * x_theor + b

# Plot the Anscombe data and theoretical line
_ = plt.plot(x1, y1, marker='.', linestyle='none')
_ = plt.plot(x_theor, y_theor)

# Label the axes
plt.xlabel('x')
plt.ylabel('y')

# Show the plot
plt.show()
```

```
slope = 0.5000909090909096
intercept = 3.000090909090909
```



1.9 Your turn! (25 points)

1.9.1 Linear regression on all Anscombe data

Write code to verify that all four of the Anscombe data sets have the same slope and intercept from a linear regression, i.e. compute the slope and intercept for each set.

The data are stored in lists (`anscombe_x = [x1, x2, x3, x4]` and `anscombe_y = [y1, y2, y3, y4]`), corresponding to the x and y values for each Anscombe data set.

1.10 Solution

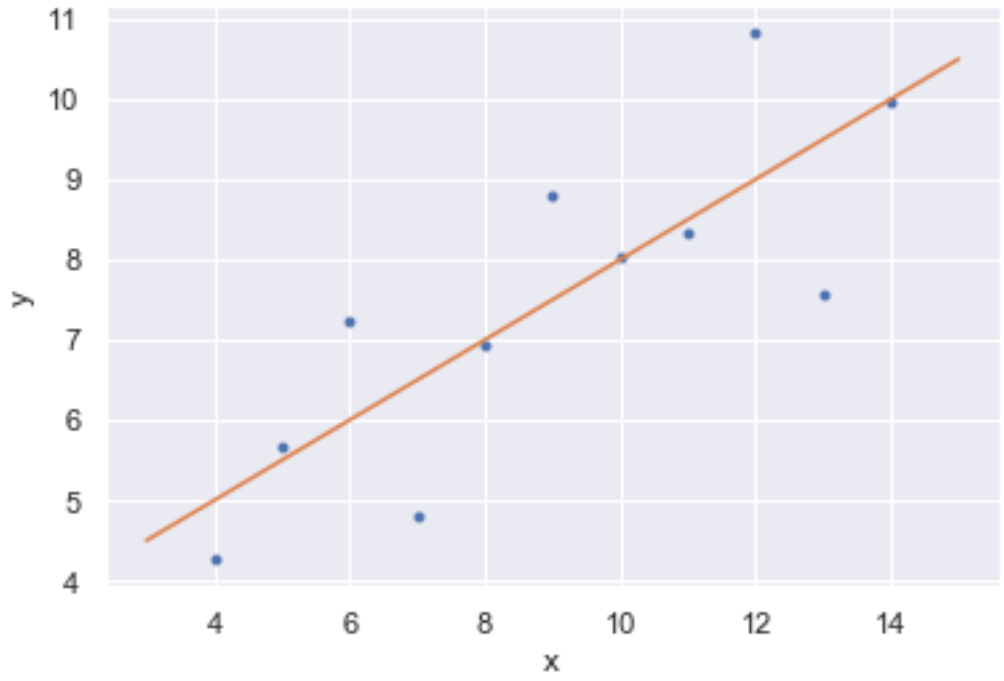
[156]:
```
anscombe_x = [x1, x2, x3, x4]
anscombe_y = [y1, y2, y3, y4]

for i in range(len(anscombe_x)):
    print("Stats for x" + str(i) + " y" + str(i) )
    a, b = np.polyfit(anscombe_x[i], anscombe_y[i], 1)

    # Print the slope and intercept
    print('slope =', round(a,3))
    print('intercept =', round(b,2))
    print("")
```

```
Stats for x0 y0
slope = 0.5
```

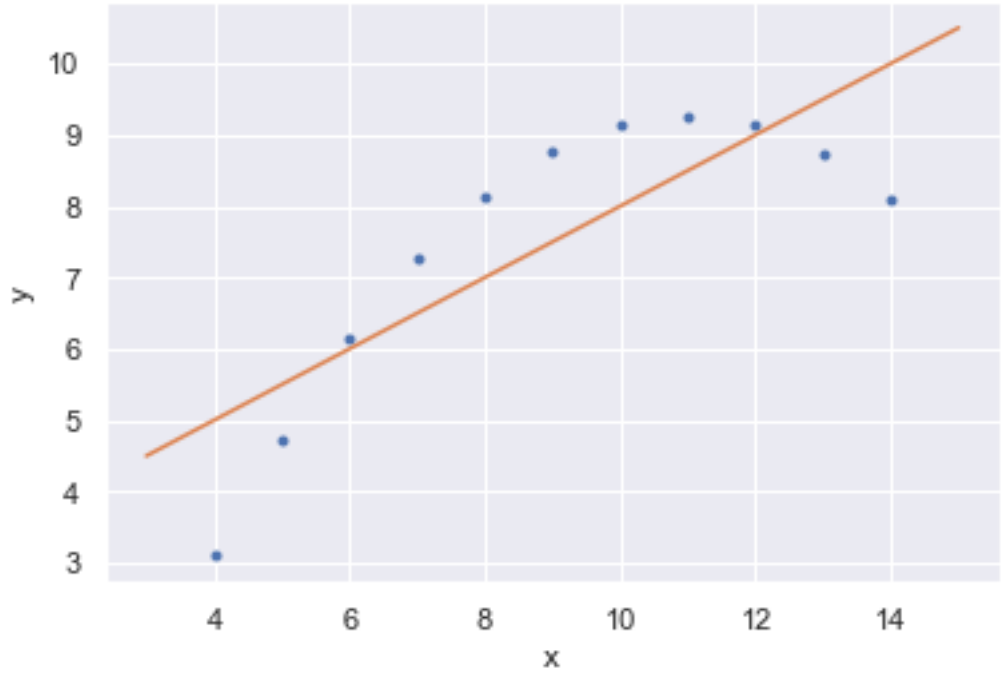
```
intercept = 3.0

Stats for x1 y1
slope = 0.5
intercept = 3.0

Stats for x2 y2
slope = 0.5
intercept = 3.0

Stats for x3 y3
slope = 0.5
intercept = 3.0
```

[157]:
```python
# Plot graphs
for i in range(len(anscombe_x)):
    print("Plot for: x" + str(i) + " y" + str(i) )
    a, b = np.polyfit(anscombe_x[i], anscombe_y[i], 1)
    # Plot the Anscombe data and theoretical line
    _ = plt.plot(anscombe_x[i], anscombe_y[i], marker='.', linestyle='none')
    _ = plt.plot(x_theor, y_theor)

    # Label the axes
    plt.xlabel('x')
    plt.ylabel('y')

    # Show the plot
    plt.show()
```

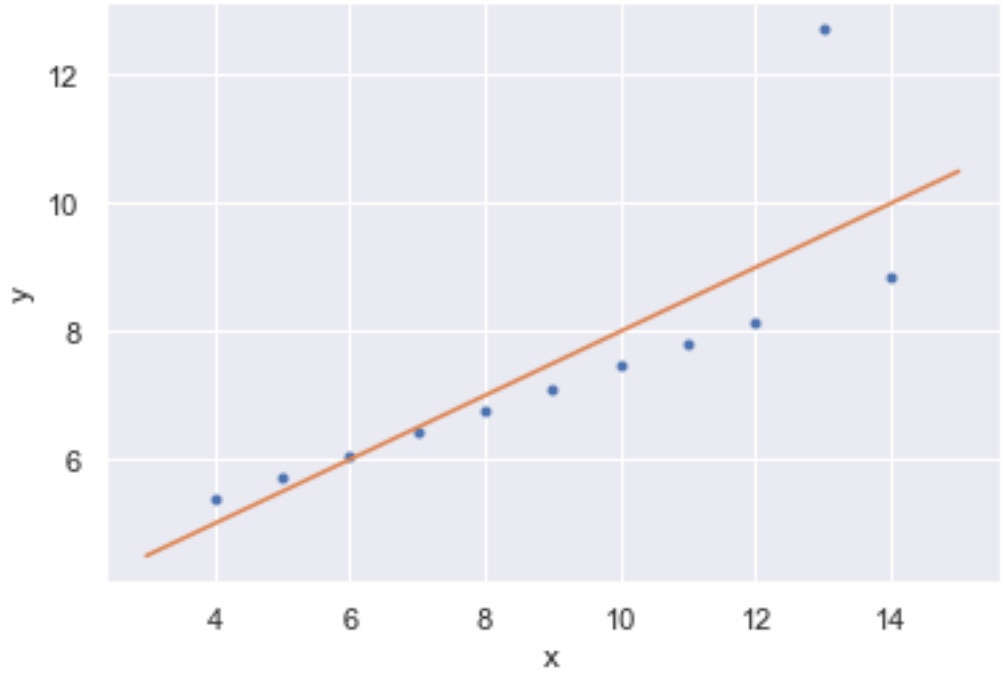
```
Plot for: x0 y0
```



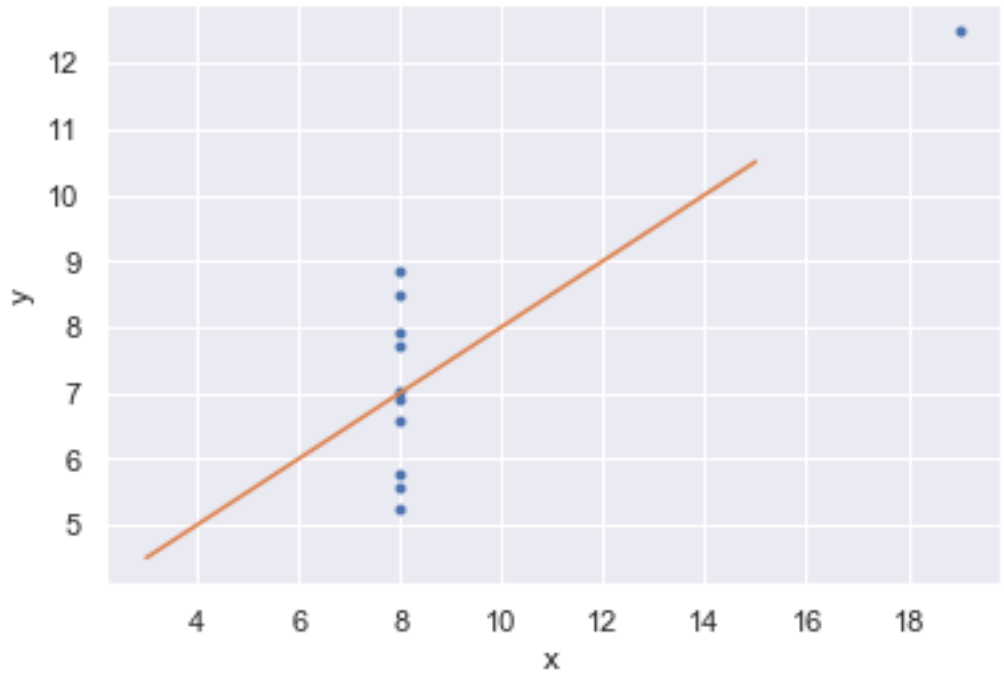
Plot for: x1 y1



Plot for: x2 y2



Plot for: x3 y3



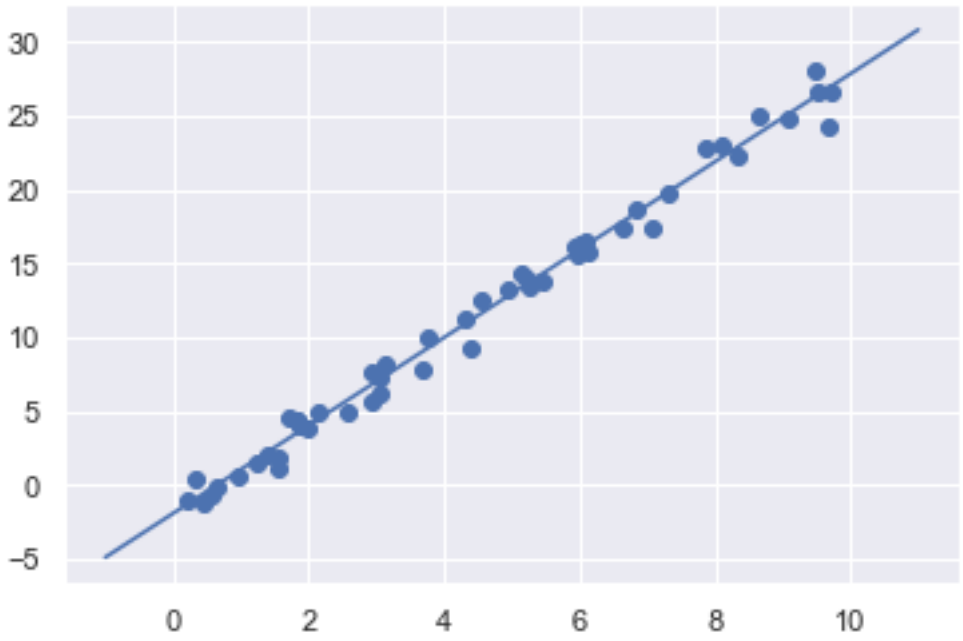
1.11 Part 3: Regression using scikit-learn

Now that we know the basics of linear regression, we will switch to scikit-learn, a powerful, workflow-oriented library for data science and machine learning.

The Python code below shows a simple linear regression example using scikit-learn. Note the use of the `fit()` and `predict()` methods.

[158]:
```
import matplotlib.pyplot as plt
import numpy as np

# Generate random data around the y = ax+b line where a=3 and b=-2
rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 3 * x - 2 + rng.randn(50)

from sklearn.linear_model import LinearRegression

# Note: If you get a "ModuleNotFoundError: No module named 'sklearn'" error␣
 ↪message, don't panic.
# It probably means you'll have to install the module by hand if you're using␣
 ↪pip.
# If you're using conda, you should not see any error message.

model = LinearRegression(fit_intercept=True)

X = x[:, np.newaxis]
X.shape

model.fit(X, y)
print(model.coef_)
print(model.intercept_)

xfit = np.linspace(-1, 11)
Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)

plt.scatter(x, y)
plt.plot(xfit, yfit);
```

```
[2.9776566]
-1.9033107255311084
```



1.12 Polynomial regression

One way to adapt linear regression to nonlinear relationships between variables is to transform the data according to *basis functions*.

The idea is to take the multidimensional linear model:

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots$$

and build the x_1, x_2, x_3, and so on, from our single-dimensional input x. That is, we let $x_n = f_n(x)$, where $f_n()$ is some function that transforms our data.

For example, if $f_n(x) = x^n$, our model becomes a polynomial regression:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

Notice that this is *still a linear model*—the linearity refers to the fact that the coefficients a_n never multiply or divide each other. What we have effectively done is taken our one-dimensional x values and projected them into a higher dimension, so that a linear fit can fit more complicated relationships between x and y.

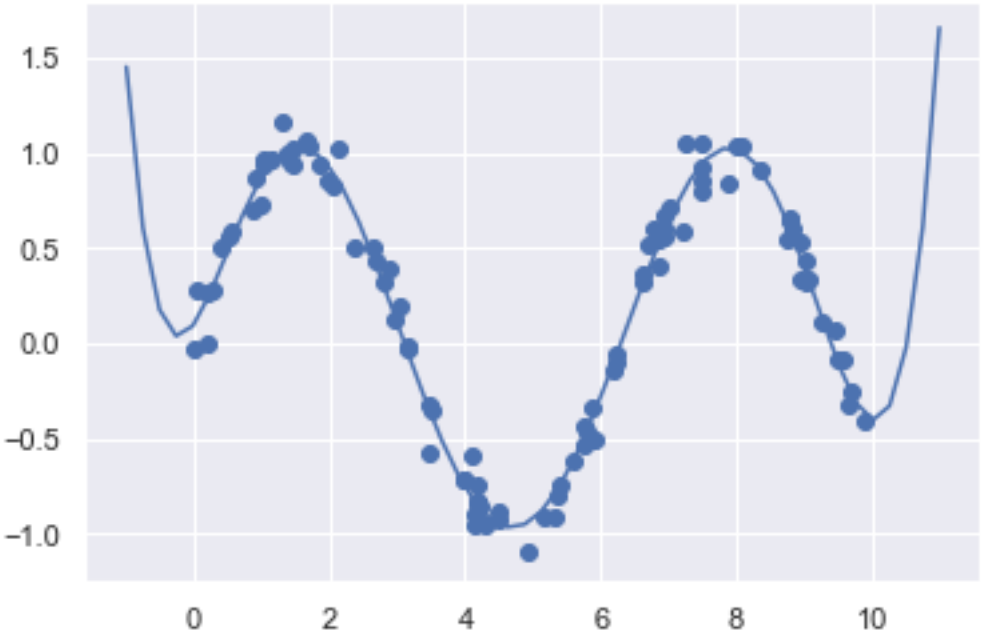
The code below shows a simple example of polynomial regression using the `PolynomialFeatures` transformer in scikit-learn. Concretely, it shows how we can use polynomial features with a polynomial of degree seven, i.e.

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_7 x^7$$

It also introduces the notion of a *pipeline* in scikit-learn. "The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters." (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)

[159]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())

rng = np.random.RandomState(1)
x = 10 * rng.rand(100)
y = np.sin(x) + 0.1 * rng.randn(100)

poly_model.fit(x[:, np.newaxis], y)
yfit = poly_model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit);

print('The R^2 score for the fit is: ', poly_model.score(x[:, np.newaxis], y))
```

```
The R^2 score for the fit is:  0.9806993128749468
```



Our linear model, through the use of 7th-order polynomial basis functions, can provide an excellent fit to this non-linear data!

1.13 Questions 1-3 (12 points, i.e. 4 pts each)

1. Which mathematical function was used to simulate the data points (with a bit of random noise around them)?
2. Which degree/order was used for the polynomial basis functions?
3. How good was the linear model fit to the non-linear data?

1.14 Solution

1. The numpy sin function was used to generate this data set along with using RandomState.rand which gives some random values from a normal distribution.
2. The 7th degree was used for the polynomial basis function
3. They both fit well based on their data sets. Now if you swapped the models based on their data sets they both would have performed poorly.

1.15 Your turn (18 points)

Write code to find the best degree/order for the polynomial basis functions (between 1 and 15) by computing the quality of the fit using a suitable metric, in this case the R^2 coefficient (which can be computer using the `score()` function).

Remember that **the best possible score is 1.0**. The score can be negative (because the model can be arbitrarily worse). A score of 0 suggests a constant model that always predicts the expected value of y, disregarding the input features.

Hint: If you plot the score against the degree/order of the polynomial, you should see something like this:

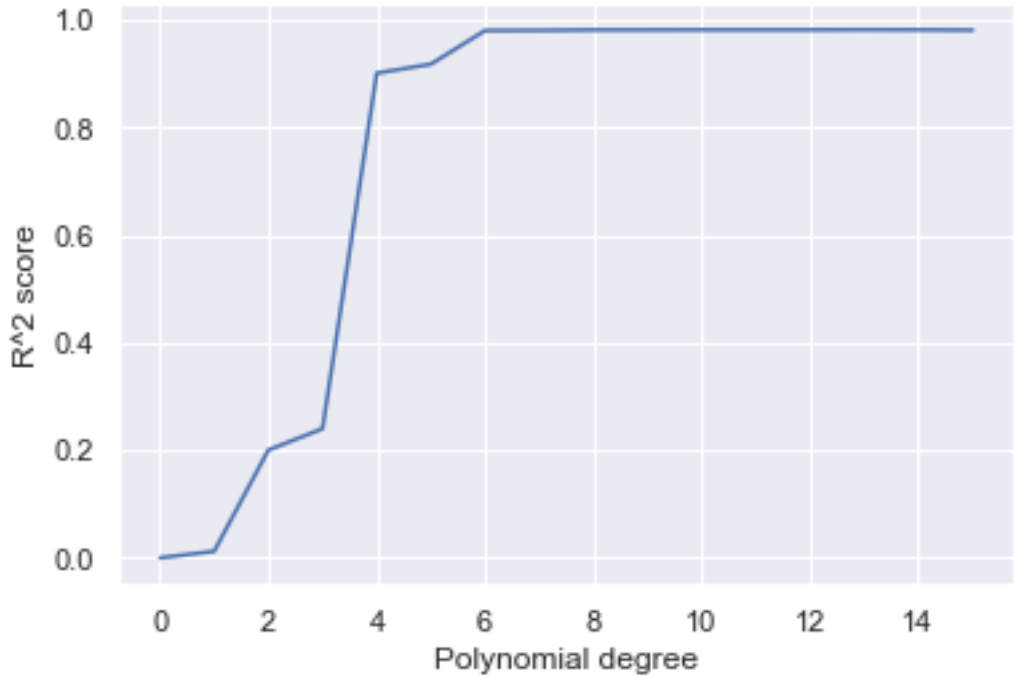
download.png

1.16 Solution

[160]:
```
def compute_r_baised_on_poly_degree(poly_degree):
    poly_model = make_pipeline(PolynomialFeatures(poly_degree),
                               LinearRegression())

    rng = np.random.RandomState(1)
    x = 10 * rng.rand(100)
    y = np.sin(x) + 0.1 * rng.randn(100)
    poly_model.fit(x[:, np.newaxis], y)
    return poly_model.score(x[:, np.newaxis], y)
```

[161]:
```
r_values = []
for degree in range(16):
    r_values.append(compute_r_baised_on_poly_degree(degree))

_ = plt.xlabel('Polynomial degree')
_ = plt.ylabel('R^2 score')
plt.plot(r_values)
```

[161]:
```
[<matplotlib.lines.Line2D at 0x26166344308>]
```



[162]:
```
print("Highest polynomial", r_values.index(max(r_values)))
```

```
Highest polynomial 13
```

1.17 Questions 4-6 (12 points, i.e. 4 pts each)

4. Which degree/order polynomial produced the best fit (i.e., highest R^2 score)?
5. Would you consider using the resulting polynomial as your model? Why (not)?
6. If you answered 'no' to question 5 (as you should!), which degree would you choose for your polynomial regression model?

1.18 Solution

4. Polynomial 13 was the highest R^2 score as I computed right after the graph
5. No we should not pick this as a model
6. This will produce an over fitting model that potentially would do worse when applied to new data or a larger dataset. It also makes our model more complex that it really needs to be. A way to combat this would be to use regularization which we will do in the next Part.

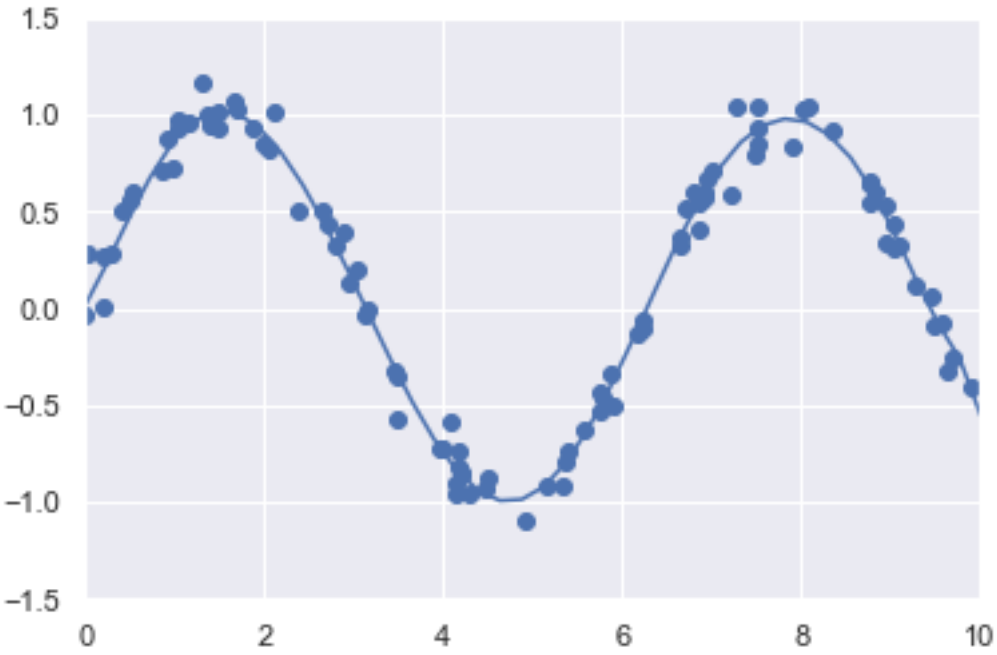
1.19 Part 4: Regularization

The use of polynomial regression with high-order polynomials can very quickly lead to overfitting. In this part, we will look into the use of regularization to address potential overfitting.

The code below shows an attempt to fit a 15th degree polynomial to a sinusoidal shaped data. The fit is excellent ($R^2 > 0.98$), but might raise suspicions that it will lead to overfitting.

[163]:
```
model = make_pipeline(PolynomialFeatures(15),
                      LinearRegression())
model.fit(x[:, np.newaxis], y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);

score = poly_model.score(x[:, np.newaxis], y)
print(score)
```

```
0.9806993128749468
```



1.20 Your turn! (10 points)

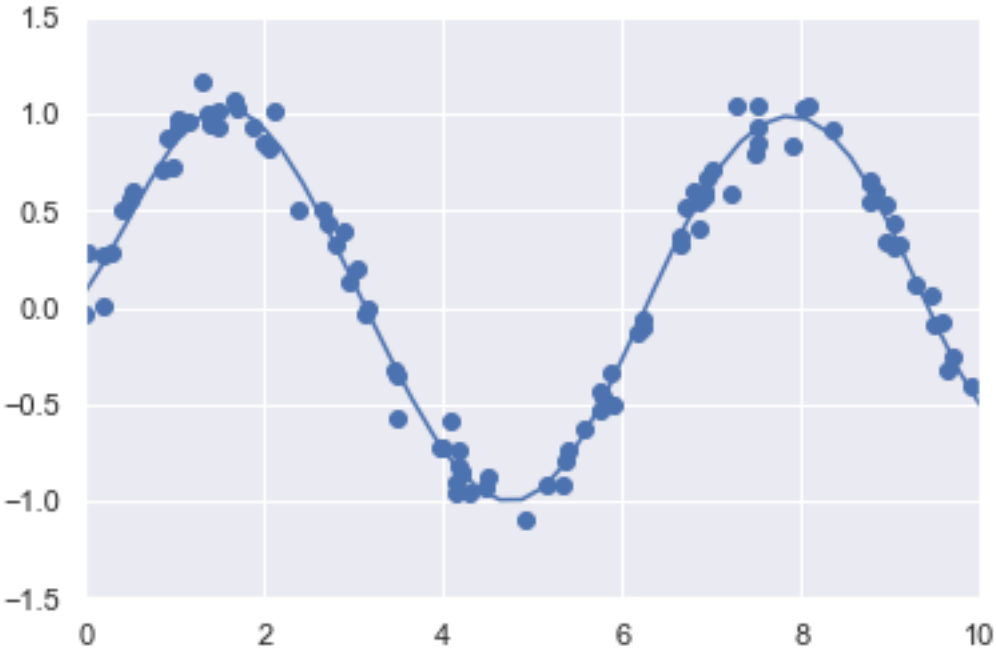
Write Python code to perform Ridge regression (L_2 Regularization), plot the resulting fit, and compute the R^2 score.

Hints: 1. This type of penalized model is built into Scikit-Learn with the `Ridge` estimator. 2. In the beginning, use all default values for its parameters. 3. After you get your code to work, spend some time trying to fine-tune the model, i.e., experimenting with the regularization parameters.

1.21 Solution

[164]:
```
from sklearn.linear_model import Ridge
model = make_pipeline(PolynomialFeatures(15),
                      Ridge(alpha=.1,
                            tol=0.001,
                            solver='auto'))

model.fit(x[:, np.newaxis], y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);

score = poly_model.score(x[:, np.newaxis], y)
print(score)
```

```
0.9806993128749468
```



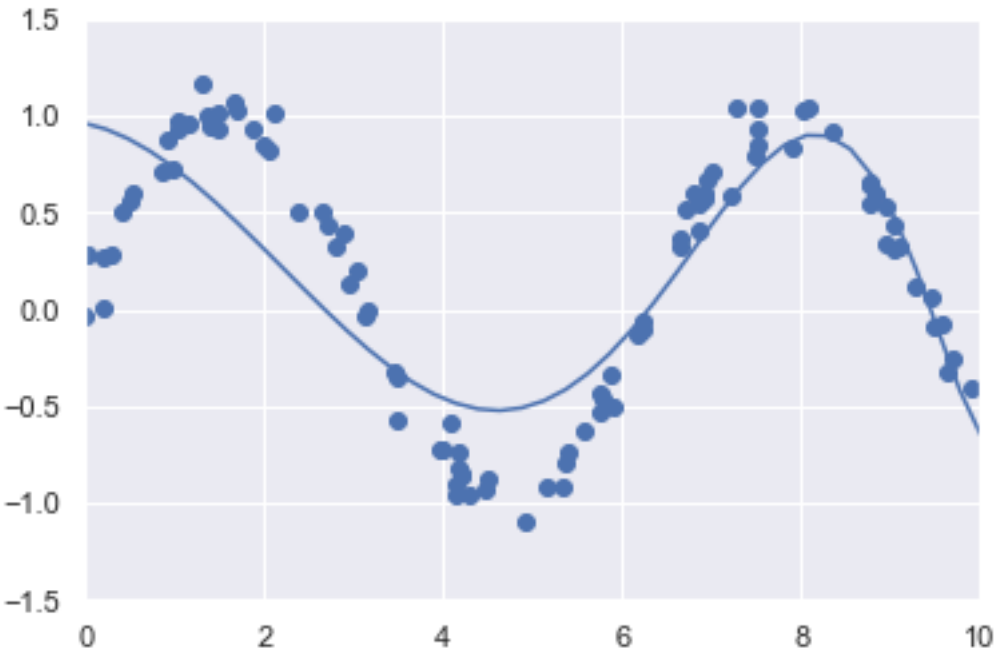
Your turn! (10 points) Write Python code to perform Lasso regression (L_1 Regularization), plot the resulting fit, and compute the R^2 score.

Hints: 1. This type of penalized model is built into Scikit-Learn with the `Lasso` estimator. 2. In the beginning, use `Lasso(alpha=0.1, tol=0.2)` 3. After you get your code to work, spend some time trying to fine-tune the model, i.e., experimenting with the regularization parameters.

1.22 Solution

[175]:
```
from sklearn.linear_model import Lasso

model = make_pipeline(PolynomialFeatures(15),
                      Lasso(alpha=0.01, tol=0.15,precompute=True))
# Lasso()
model.fit(x[:, np.newaxis], y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);

score = poly_model.score(x[:, np.newaxis], y)
print(score)
```

```
0.9806993128749468
```



1.23 Question 7 (4 points)

7. Which of the two regularization approaches produced a better fit (and was also the easiest to fine-tune)?

1.24 Solution

I found that the Ridge was very close to the model with the default values. Even messing with the values didnt result much change in the Ridge graph until I changed the solver which completely changed the model.

The Lasso model was not as close to the data points. I found that increasing the alpha greater than 1 negatively impacted the graph. Making alpha smaller and the tolerance smaller seemed to help little but not by much from the human eye. I am not sure why but the R^2 score didnt really change when I tweaked those values. I did find that having tol=.1 caused a warning and when I increased it to .15 that helped. This was the error/warning: "ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations"

Overall the Ridge seamed to be easier to model to fit for this dataset.

1.25 Bonus! (30 points)

The Boston housing dataset is a classic dataset used in linear regression examples. (See https://scikit-learn.org/stable/datasets/index.html#boston-dataset for more)

The Python code below: - Loads the Boston dataset (using scikit-learn's `load_boston()`) and converts it into a Pandas dataframe - Selects two features to be used for fitting a model that will then be used to make predictions: LSTAT (% lower status of the population) and RM (average number of rooms per dwelling) (*) - Splits the data into train and test sets

(*) See https://towardsdatascience.com/linear-regression-on-boston-housing-dataset-f409b7e4a155 for details.

[176]:
```
from sklearn.datasets import load_boston
boston_dataset = load_boston()
boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston.head()
```

[176]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0

	PTRATIO	B	LSTAT
0	15.3	396.90	4.98
1	17.8	396.90	9.14
2	17.8	392.83	4.03
3	18.7	394.63	2.94
4	18.7	396.90	5.33

[177]:
```
boston.describe()
```

[177]:

	CRIM	ZN	INDUS	CHAS	NOX	RM
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000

```
25%      0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
50%      0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
75%      3.677083   12.500000   18.100000    0.000000    0.624000    6.623500
max     88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

              AGE         DIS         RAD         TAX     PTRATIO           B  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean    68.574901    3.795043    9.549407  408.237154   18.455534  356.674032
std     28.148861    2.105710    8.707259  168.537116    2.164946   91.294864
min      2.900000    1.129600    1.000000  187.000000   12.600000    0.320000
25%     45.025000    2.100175    4.000000  279.000000   17.400000  375.377500
50%     77.500000    3.207450    5.000000  330.000000   19.050000  391.440000
75%     94.075000    5.188425   24.000000  666.000000   20.200000  396.225000
max    100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

            LSTAT
count  506.000000
mean    12.653063
std      7.141062
min      1.730000
25%      6.950000
50%     11.360000
75%     16.955000
max     37.970000
```

[201]:
```
boston_dataset.target
```

[201]:
```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15. ,
       18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
       15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
       13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
       21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
       35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
       19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
       20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
       23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
       33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
       21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
       20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
       23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
       15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
       17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
       25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
       23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
       32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
       34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
       20. , 21.7, 19.3, 22.4, 28.1, 23.7, 25. , 23.3, 28.7, 21.5, 23. ,
       26.7, 21.7, 27.5, 30.1, 44.8, 50. , 37.6, 31.6, 46.7, 31.5, 24.3,
```

```
       31.7, 41.7, 48.3, 29. , 24. , 25.1, 31.5, 23.7, 23.3, 22. , 20.1,
       22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
       42.8, 21.9, 20.9, 44. , 50. , 36. , 30.1, 33.8, 43.1, 48.8, 31. ,
       36.5, 22.8, 30.7, 50. , 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
       32. , 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46. , 50. , 32.2, 22. ,
       20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
       20.3, 22.5, 29. , 24.8, 22. , 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
       22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
       21. , 23.8, 23.1, 20.4, 18.5, 25. , 24.6, 23. , 22.2, 19.3, 22.6,
       19.8, 17.1, 19.4, 22.2, 20.7, 21.1, 19.5, 18.5, 20.6, 19. , 18.7,
       32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
       18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25. , 19.9, 20.8,
       16.8, 21.9, 27.5, 21.9, 23.1, 50. , 50. , 50. , 50. , 50. , 13.8,
       13.8, 15. , 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3,  8.8,
        7.2, 10.5,  7.4, 10.2, 11.5, 15.1, 23.2,  9.7, 13.8, 12.7, 13.1,
       12.5,  8.5,  5. ,  6.3,  5.6,  7.2, 12.1,  8.3,  8.5,  5. , 11.9,
       27.9, 17.2, 27.5, 15. , 17.2, 17.9, 16.3,  7. ,  7.2,  7.5, 10.4,
        8.8,  8.4, 16.7, 14.2, 20.8, 13.4, 11.7,  8.3, 10.2, 10.9, 11. ,
        9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4,  9.6,  8.7,  8.4, 12.8,
       10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13. , 13.4,
       15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20. , 16.4, 17.7,
       19.5, 20.2, 21.4, 19.9, 19. , 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
       29.8, 13.8, 13.3, 16.7, 12. , 14.6, 21.4, 23. , 23.7, 25. , 21.8,
       20.6, 21.2, 19.1, 20.6, 15.2,  7. ,  8.1, 13.6, 20.1, 21.8, 24.5,
       23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22. , 11.9])
```

[178]:
```
boston['MEDV'] = boston_dataset.target
X = pd.DataFrame(np.c_[boston['LSTAT'], boston['RM']], columns = ['LSTAT','RM'])
y = boston['MEDV']

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state=5)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(404, 2)
(102, 2)
(404,)
(102,)
```

Write Python code to:

1. Fit a linear model to the data.
2. Compute and print the RMSE and R^2 score for both train and test datasets.

3. Fit a polynomial model (of degree 4) to the data.
4. Compute and print the RMSE and R^2 score for both train and test datasets.
5. Apply Ridge regression to the polynomial model.
6. Compute and print the RMSE and R^2 score for both train and test datasets.

1.26 Solution

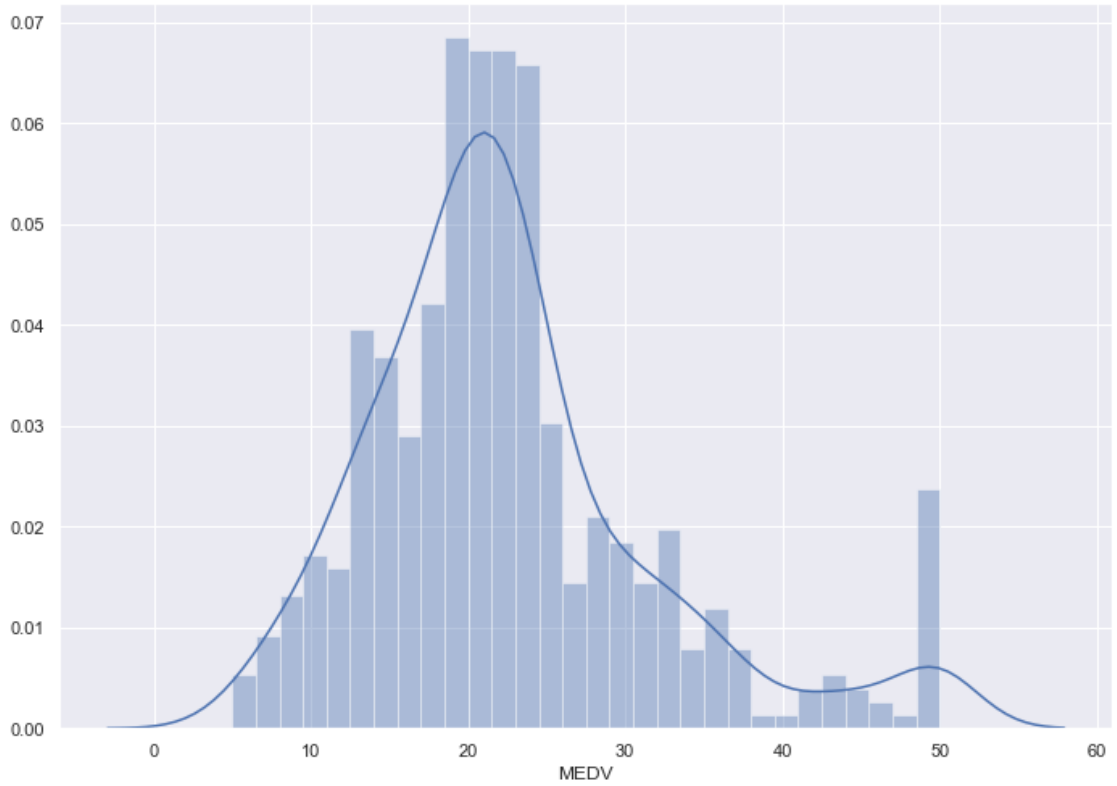
[186]:
```
X
```

[186]:
```
     LSTAT     RM
33   18.35  5.701
283   3.16  7.923
418  20.62  5.957
502   9.08  6.120
402  20.31  6.404
..     ...    ...
486  14.98  6.114
189   5.39  7.185
495  17.60  5.670
206  10.97  6.326
355   5.57  5.936

[404 rows x 2 columns]
```

[192]:
```
y
```

[192]:
```
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: MEDV, Length: 506, dtype: float64
```

[228]:
```
sns.set(rc={'figure.figsize':(11.7,8.27)})
sns.distplot(boston['MEDV'], bins=30)
plt.show()
```



[ ]:

[259]:
```
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt

def test_model(model, X_train, y_train, X_test, y_test):
    model.fit(X_train, y_train)

    y_train_prediction = model.predict(X_train)
    rmse = (np.sqrt(mean_squared_error(y_train, y_train_prediction)))
    r_2 = r2_score(y_train,y_train_prediction)
    print("Training data")
    print("The RMSE: ", rmse)
    print('The R^2 score: ', r_2)

    # Test data
    y_test_prediction = model.predict(X_test)
    rmse = (np.sqrt(mean_squared_error(y_test, y_test_prediction)))
    r_2 = r2_score(y_test,y_test_prediction)

    print("\nPrediction Data")
    print("The RMSE: ", rmse)
    print('The R^2 score: ', r_2)
```

[267]:
```
print("1 & 2")
# 1. Fit a linear model to the data.
# 2. Compute and print the RMSE and $R^2$ score for both train and test␣
 ↪datasets.

model = LinearRegression(fit_intercept=True)
test_model(model, X_train, y_train, X_test, y_test)
```

```
1 & 2
Training data
The RMSE:  5.6371293350711955
The R^2 score:  0.6300745149331701

Prediction Data
The RMSE:  5.137400784702911
The R^2 score:  0.6628996975186953
```

[264]:
```
print("3 & 4")
# 3. Fit a polynomial model (of degree 4) to the data.
# 4. Compute and print the RMSE and $R^2$ score for both train and test␣
 ↪datasets.

poly_model = make_pipeline(PolynomialFeatures(4),
                           LinearRegression())
test_model(poly_model, X_train, y_train, X_test, y_test)
```

```
3 & 4
Training data
The RMSE:  4.2760812120742235
The R^2 score:  0.787141918608568

Prediction Data
The RMSE:  3.888594602753813
The R^2 score:  0.8068665115825728
```

[266]:
```
print("5 & 6")
# 5. Apply Ridge regression to the polynomial model.
# 4. Compute and print the RMSE and $R^2$ score for both train and test␣
 ↪datasets.
poly_ridge_model = make_pipeline(PolynomialFeatures(4),
                                 Ridge(alpha=.1,
                                       tol=0.001,
                                       solver='auto'))
test_model(poly_ridge_model, X_train, y_train, X_test, y_test)
```

```
5 & 6
Training data
The RMSE:  4.407971809087175
The R^2 score:  0.773808713678973

Prediction Data
The RMSE:  3.8921858701614847
The R^2 score:  0.8065096143723436
```

1.27 Conclusions (10 points)

Write your conclusions and make sure to address the issues below: 1. What have you learned from this assignment? 2. Which parts were the most fun, time-consuming, enlightening, tedious? 3. What would you do if you had an additional week to work on this?

1.28 Solution

1. I a learned about computing the linear regression model and plotting that against the dataset. Thats a very simple best fit straight line for a dataset. I learned that it was interesting with the 4 different Anscombe's quartet datasets have the same statistical computations BUT they are drastically different. Gives a good perspective that we really need to look at different angles when trying to view the data just to be sure we are not viewing the data in a blind spot. Learned about build different linear regression models with different number of polynomials to try to fit to some data sets. Also that its important that we try to reduce the number of polynomials so the model doesnt over fit the data. Learned that regularization adds information which can help reduce the very large coefficients in the models to prevent over fitting

2. I like the tweaking of the models but it was kinda time consuming. I didnt really know what I was changing but kinda shooting in the dark to see what happens. I know over time & practice I would get the feel for what each param of the models does. I got tripped up big time on the first bonus problem

3. I would love to figure out how to graph the Boston bonus on each model. I get a little confused on how the model will look since we are taking in consideration of 2 attributes.