Step 1: Import packages and classes

In [2]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
```

```
The fundamental data type of NumPy is the array type called
numpy.ndarray .  The term array to refer to instances of the type
numpy.ndarray .
The class sklearn.linear_model.LinearRegression will be used to
perform linear and polynomial regression and make
predictions accordingly.
```

Step 2: Provide data

In [3]:
```
x = np.array([5, 10, 25, 35, 45, 55]).reshape((-1, 1))
y = np.array([5, 20, 14, 32, 22, 38])
```

```
.reshape() on x because this array is required to
be two-dimensional, or to be more precise, to have one column and as
many rows as necessary. That’s exactly what the argument (-1, 1) of
.reshape() specifies.
```

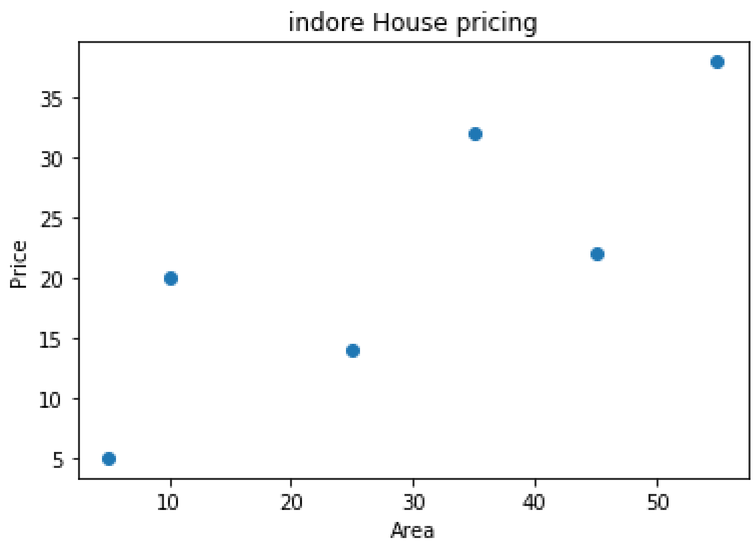
In [94]:
```
print(x)
```

```
[[ 5]
 [10]
 [25]
 [35]
 [45]
 [55]]
```

In [95]:
```
print(y)
```

```
[ 5 20 14 32 22 38]
```

In [97]:
```
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.title('indore House pricing')
plt.xlabel('Area')
plt.ylabel('Price')
plt.show()
```



In [3]:
```
print(x)
```

```
[[ 5]
 [15]
 [25]
 [35]
 [45]
 [55]]
```

In [4]:
```
print(y)
```

```
[ 5 20 14 32 22 38]
```

Step 3: Create a model and fit it

In [5]:
```
model = LinearRegression()
```

In [6]:
```
model.fit(x, y)
```

Out[6]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, norm
alize=False)
```

This statement creates the variable model as the instance of LinearRegression .
parameters to LinearRegression :
fit_intercept is a Boolean (True by default) that decides whether to calculate the intercept b0 (True) or consider it equal to zero (False).
normalize is a Boolean (False by default) that decides whether to normalize the input variables (True) or not (False).
copy_X is a Boolean (True by default) thatdecides whether to copy (True) or overwrite the input variables (False).
n_jobs is an integer or None (default) and represents the number of jobs used in parallel computation. None usually means one job and -1 to use all processors

In [7]:
```
model = LinearRegression().fit(x, y)
```

Step 4: Get results

In [8]:
```
r_sq = model.score(x, y)
```

.score() , the arguments are also the predictor x and regressor y , and the return value is R2.
The attributes of model are .intercept_ , which represents the coe icient, b0 and .coef_ , which represents b1:

In [9]:
```
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.715875613747954
```

In [10]:
```
print('intercept:', model.intercept_)
```

```
intercept: 5.633333333333329
```

In [11]:
```
print('slope:', model.coef_)
```

```
slope: [0.54]
```

In [12]:
```
new_model = LinearRegression().fit(x, y.reshape((-1, 1)))
```

In [13]:
```
print('intercept:', new_model.intercept_)
```

```
intercept: [5.63333333]
```

In [14]:
```
print('slope:', new_model.coef_)
```

```
slope: [[0.54]]
```

Step 5: Predict response

In [15]:
```
y_pred = model.predict(x)
```

```
When applying .predict() , you pass the regressor as the argument and
get the corresponding predicted response.
```

In [16]:
```
print('predicted response:', y_pred, sep='\n')
```

...

In [17]:
```
y_pred = model.intercept_ + model.coef_ * x
```

In [18]:
```
print('predicted response:', y_pred, sep='\n')
```

```
predicted response:
[[ 8.33333333]
 [13.73333333]
 [19.13333333]
 [24.53333333]
 [29.93333333]
 [35.33333333]]
```

In practice, regression models are o en applied for forecasts. This means that you can use fitted models to calculate the outputs based on some other, new inputs:

In [19]:
```
x_new = np.arange(5).reshape((-1, 1))
```

In [20]:
```
print(x_new)
```

```
[[0]
 [1]
 [2]
 [3]
 [4]]
```

In [21]:
```
y_new = model.predict(x_new)
```

In [22]:
```
print(y_new)
```

```
[5.63333333 6.17333333 6.71333333 7.25333333 7.79333333]
```

Multiple Linear Regression With scikit-learn

```
Steps 1 and 2: Import packages and classes, and provide data
```

In [23]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
x = [[0, 1], [5, 1], [15, 2], [25, 5], [35, 11], [45, 15], [55, 34],
y = [4, 5, 20, 14, 32, 22, 38, 43]
x, y = np.array(x), np.array(y)
```

In [24]:
```
print(x)
```

```
[[ 0  1]
 [ 5  1]
 [15  2]
 [25  5]
 [35 11]
 [45 15]
 [55 34]
 [60 35]]
```

```
 In multiple linear regression, x is a two-dimensional array with at
least two columns, while y is usually a one-
dimensional array. This is a simple example of multiple linear
regression, and x has exactly two columns.
```

In [4]:
```
print(y)
```

```
[ 5 20 14 32 22 38]
```

Step 3: Create a model and fit it

In [5]:
```
model = LinearRegression().fit(x, y)
```

Step 4: Get results

Step 4: Get results

In [27]:
```
r_sq = model.score(x, y)
```

In [28]:
```
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.8615939258756776
```

In [29]:
```
print('intercept:', model.intercept_)
```

```
intercept: 5.52257927519819
```

In [30]:
```
print('slope:', model.coef_)
```

```
slope: [0.44706965 0.25502548]
```

the value of R2 using .score() and the values of the estimators of regression coe icients with .intercept_and .coef_ . Again, .intercept_ holds the bias b0, while now .coef_ is an array containing b1 and b2 respectively.
In this example, the intercept is approximately 5.52, and this is the value of the predicted response when x1 = x2 = 0.
The increase of x1 by 1 yields the rise of the predicted response by 0.45. Similarly, when x2 grows by 1, the response
rises by 0.26

Step 5: Predict response

In [31]:
```
y_pred = model.predict(x)
```

In [32]:
```
print('predicted response:', y_pred, sep='\n')
```

```
predicted response:
[ 5.77760476  8.012953   12.73867497 17.9744479  23.97529728 29.466
0957
 38.78227633 41.27265006]
```

In [33]:
```
y_pred = model.intercept_ + np.sum(model.coef_ * x, axis=1)
```

In [34]:
```
print('predicted response:', y_pred, sep='\n')
```

```
predicted response:
[ 5.77760476  8.012953   12.73867497 17.9744479  23.97529728 29.466
0957
 38.78227633 41.27265006]
```

In [35]:
```
x_new = np.arange(10).reshape((-1, 2))
```

In [36]:
```
print(x_new)
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

In [37]:
```
y_new = model.predict(x_new)
```

In [38]:
```
print(y_new)
```

```
[ 5.77760476  7.18179502  8.58598528  9.99017554 11.3943658 ]
```

Polynomial Regression With scikit-learn

Implementing polynomial regression with scikit-learn is very similar to linear regression. There is only one extra step: just need to transform the array of inputs to include non-linear terms such as x2.

Step 1: Import packages and classes

In addition to numpy and sklearn.linear_model.LinearRegression , you should also import the class PolynomialFeatures from sklearn.preprocessing :

In [39]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

Step 2a: Provide data

In [40]:
```
x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))
y = np.array([15, 11, 2, 8, 25, 32])
```

Step 2b: Transform input data

This is the new step you need to implement for polynomial regression! As you've seen earlier, you need to include x2 (and perhaps other terms) as additional features when implementing polynomial regression. For that reason, you should transform the input array x to contain the additional column(s)with the values of x2 (and eventually more features).

In [41]:
```
transformer = PolynomialFeatures(degree=2, include_bias=False)
```

In [42]:
```
transformer.fit(x)
```

Out[42]:
```
PolynomialFeatures(degree=2, include_bias=False, interaction_only=F
alse,
                   order='C')
```

```
parameters to PolynomialFeatures :
degree is an integer ( 2 by default) that represents the degree of
the polynomialregression function.
interaction_only is a Boolean ( False by default) that decides
whether to include only interaction features ( True )
or all features ( False ).
include_bias is a Boolean ( True by default) that decides whether to
include the bias (intercept) column of ones
( True ) or not ( False ).
```

In [43]:
```
x_ = transformer.transform(x)
```

In [44]:
```
x_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(x
```

```
That's fitting and transforming the input array in one statement with
.fit_transform() . It also takes the input array
and e ectively does the same thing as .fit() and .transform() called
in that order. It also returns the modified array
```

In [45]:
```
print(x_)
```

```
[[   5.   25.]
 [  15.  225.]
 [  25.  625.]
 [  35. 1225.]
 [  45. 2025.]
 [  55. 3025.]]
```

Step 3: Create a model and fit it

In [46]:
```
model = LinearRegression().fit(x_, y)
```

Step 4: Get results

In [47]:
```
r_sq = model.score(x_, y)
```

In [48]:
```
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.8908516262498564
```

In [49]:
```
print('intercept:', model.intercept_)
```

```
intercept: 21.372321428571425
```

In [50]:
```
print('coefficients:', model.coef_)
```

```
coefficients: [-1.32357143  0.02839286]
```

```
Again, .score() returns R2. Its first argument is also the modified
input x_ , not x . The values of the weights are associated to
.intercept_ and .coef_ : .intercept_ represents b0, while .coef_
references the array that contains b1
and b2 respectively.
```

In [51]:
```
x_ = PolynomialFeatures(degree=2, include_bias=True).fit_transform(x)
```

In [52]:
```
print(x_)
```

```
[[1.000e+00 5.000e+00 2.500e+01]
 [1.000e+00 1.500e+01 2.250e+02]
 [1.000e+00 2.500e+01 6.250e+02]
 [1.000e+00 3.500e+01 1.225e+03]
 [1.000e+00 4.500e+01 2.025e+03]
 [1.000e+00 5.500e+01 3.025e+03]]
```

In [53]:
```
model = LinearRegression(fit_intercept=False).fit(x_, y)
```

In [54]:
```
r_sq = model.score(x_, y)
```

In [55]:
```
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.8908516262498565
```

In [56]:
```
print('intercept:', model.intercept_)
```

```
intercept: 0.0
```

In [57]:
```
print('coefficients:', model.coef_)
```

```
coefficients: [21.37232143 -1.32357143  0.02839286]
```

Step 5: Predict response

In [58]:
```
y_pred = model.predict(x_)
```

In [59]:
```
print('predicted response:', y_pred, sep='\n')
```

```
predicted response:
[15.46428571  7.90714286  6.02857143  9.82857143 19.30714286 34.464
28571]
```

```
the prediction works almost the same way as in the case of linear
regression. It just requires the modified input instead of the
original.
```

In [60]:
```
# Step 1: Import packages
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
# Step 2a: Provide data
x = [[0, 1], [5, 1], [15, 2], [25, 5], [35, 11], [45, 15], [55, 34],
y = [4, 5, 20, 14, 32, 22, 38, 43]
x, y = np.array(x), np.array(y)
# Step 2b: Transform input data
x_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(x
# Step 3: Create a model and fit it
model = LinearRegression().fit(x_, y)
# Step 4: Get results
r_sq = model.score(x_, y)
intercept, coefficients = model.intercept_, model.coef_
# Step 5: Predict
y_pred = model.predict(x_)
print('coefficient of determination:', r_sq)
```

```
coefficient of determination: 0.9453701449127822
```

In [61]:
```
print('intercept:', intercept)
```

```
intercept: 0.8430556452395734
```

In [62]:
```
print('coefficients:', coefficients, sep='\n')
```

```
coefficients:
[ 2.44828275  0.16160353 -0.15259677  0.47928683 -0.4641851 ]
```

In [63]:
```
print('predicted response:', y_pred, sep='\n')
```

```
predicted response:
[ 0.54047408 11.36340283 16.07809622 15.79139    29.73858619 23.508
34636
 39.05631386 41.92339046]
```

In this case, there are six regression coe icients (including the intercept), as shown in the estimated regression function f(x1, x2) = b0 + b1x1 + b2x2 + b3x12 + b4x1x2 + b5x22.
it can also notice that polynomial regression yielded a higher coe icient of determination than multiple linear regression for the same problem. At first, it could think that obtaining such a large R2 is an excellent result. It might be. However, in real-world situations, having a complex model and R2 very close to 1 might also be a sign of overfitting.
To check the performance of a model, you should test it with new data, that is with observations not used to fit (train) the model.

Advanced Linear Regression With statsmodels

Step 1: Import packages

In [64]:
```
import numpy as np
import statsmodels.api as sm
```

Step 2: Provide data and transform inputs

In [65]:
```
x = [[0, 1], [5, 1], [15, 2], [25, 5], [35, 11], [45, 15], [55, 34],
y = [4, 5, 20, 14, 32, 22, 38, 43]
x, y = np.array(x), np.array(y)
```

In [66]:
```
x = sm.add_constant(x)
```

In [67]:
```
print(x)
```

```
[[ 1.  0.  1.]
 [ 1.  5.  1.]
 [ 1. 15.  2.]
 [ 1. 25.  5.]
 [ 1. 35. 11.]
 [ 1. 45. 15.]
 [ 1. 55. 34.]
 [ 1. 60. 35.]]
```

In [68]:
```
print(y)
```

```
[ 4  5 20 14 32 22 38 43]
```

Step 3: Create a model and fit it

```
The regression model based on ordinary least squares is an insta
nce of the class
```

statsmodels.regression.linear_model.OLS

In [69]:
```
model = sm.OLS(y, x)
```

In [70]:
```
results = model.fit()
```

Step 4: Get results

In [71]:

```
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.862
Model:                            OLS   Adj. R-squared:                  0.806
Method:                 Least Squares   F-statistic:                     15.56
Date:                Fri, 20 Dec 2019   Prob (F-statistic):            0.00713
Time:                        20:47:32   Log-Likelihood:                -24.316
No. Observations:                   8   AIC:                             54.63
Df Residuals:                       5   BIC:                             54.87
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.5226      4.431      1.246      0.268      -5.867      16.912
x1             0.4471      0.285      1.567      0.178      -0.286       1.180
x2             0.2550      0.453      0.563      0.598      -0.910       1.420
==============================================================================
Omnibus:                        0.561   Durbin-Watson:                   3.268
Prob(Omnibus):                  0.755   Jarque-Bera (JB):                0.534
Skew:                           0.380   Prob(JB):                        0.766
Kurtosis:                       1.987   Cond. No.                         80.1
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.


/home/vikas/anaconda3/lib/python3.7/site-packages/scipy/stats/stats.py:1450: UserWarning: kurtosistest only valid for n>=20 ... continuing anyway, n=8
  "anyway, n=%i" % int(n))
```

This table is very comprehensive. You can find many statistical values associated with linear regression including R2,b0, b1, and b2. In this particular case, you might obtain the warning related to kurtosistest . This is due to the small number of observations provided.

In [72]:
```
print('coefficient of determination:', results.rsquared)
```

```
coefficient of determination: 0.8615939258756777
```

In [73]:
```
print('adjusted coefficient of determination:', results.rsquared_adj
```

```
adjusted coefficient of determination: 0.8062314962259488
```

In [74]:
```
print('regression coefficients:', results.params)
```

```
regression coefficients: [5.52257928 0.44706965 0.25502548]
```

In [75]:
```
print('predicted response:', results.fittedvalues, sep='\n')
```

```
predicted response:
[ 5.77760476  8.012953   12.73867497 17.9744479  23.97529728 29.466
0957
 38.78227633 41.27265006]
```

In [76]:
```
print('predicted response:', results.predict(x), sep='\n')
```

```
predicted response:
[ 5.77760476  8.012953   12.73867497 17.9744479  23.97529728 29.466
0957
 38.78227633 41.27265006]
```

The results of linear regression:
1. .rsquared holds R2.
2. .rsquared_adj represents adjusted R2 (R2 corrected according to the number of input features).
3. .params refers the array with b0, b1, and b2 respectively.

In [77]:
```
x_new = sm.add_constant(np.arange(10).reshape((-1, 2)))
```

Step 5: Predict response

In [78]:
```
print(x_new)
```

```
[[1. 0. 1.]
 [1. 2. 3.]
 [1. 4. 5.]
 [1. 6. 7.]
 [1. 8. 9.]]
```

In [79]:
```
y_new = results.predict(x_new)
```

In [80]:
```
print(y_new)
```

```
[ 5.77760476  7.18179502  8.58598528  9.99017554 11.3943658 ]
```

In [ ]:

```
1. Import the packages and classes you need
2. Provide data to work with and eventually do appropriate
transformations
3. Create a regression model and fit it with existing data
4. Check the results of model fitting to know whether the model is
satisfactory
5. Apply the model for predictions
```

In [ ]: