Data Analysis and Machine-Learning

Chapter 2.1:

Linear Regression Model – Overview



1. Introduction

Understanding the concept of linear regression model is without doubt the most important part in terms of comprehending the framework of machine learning / deep learning algorithms. This is because, roughly speaking, all machine-learning/deep-learning algorithms are fundamentally applications and variations of a linear regression model. For example, neural network algorithm can be summarized as the process of “piling” many linear models, converting them into non-linear models via applying activation functions (e.g., sigmoid, relu) throughout multiple (hidden) layers. As we will see from now on, the irreplaceable advantages of linear regression model [(1) structural intuitiveness, (2) outstanding analytical interpretability] become the reason for its wide applicability.

Enough with words, let’s get right into the world of linear models with codes!

1. Generating Sample Data and Visualization

Import compulsory libraries

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

Generate samples for linear regression: x and y

x = np.array([30, 31, 32, 33, 34])

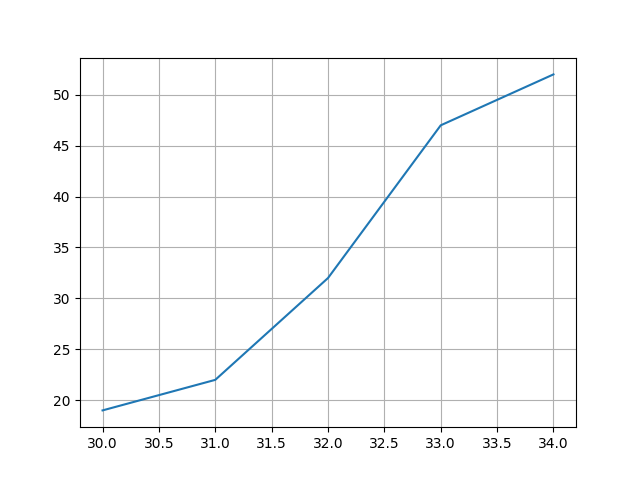
y = np.array([19, 22, 32, 47, 52])

Drawing line plots using matplotlib

plt.plot(x, y)

plt.grid()

plt.show()

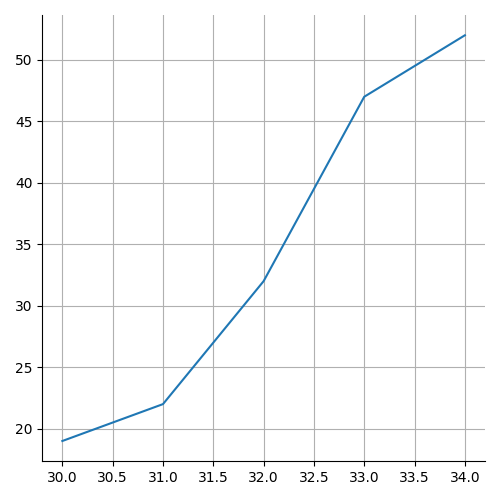


Drawing line plots using seaborn

sns.relplot(x=x, y=y, kind='line')

plt.grid()

plt.show()

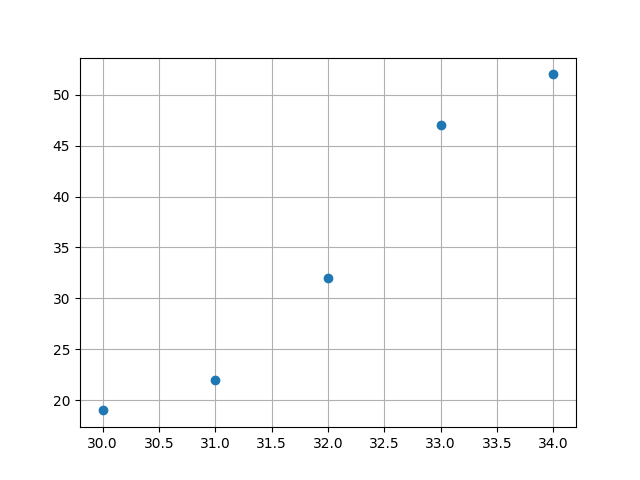


Drawing scatter plots using matplotlib (1)

plt.plot(x, y, 'o')

plt.grid()

plt.show()

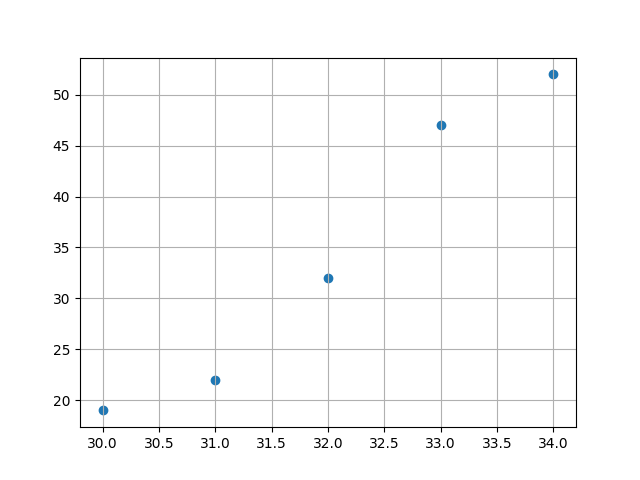


Drawing scatter plots using matplotlib (2)

plt.scatter(x, y)

plt.grid()

plt.show()

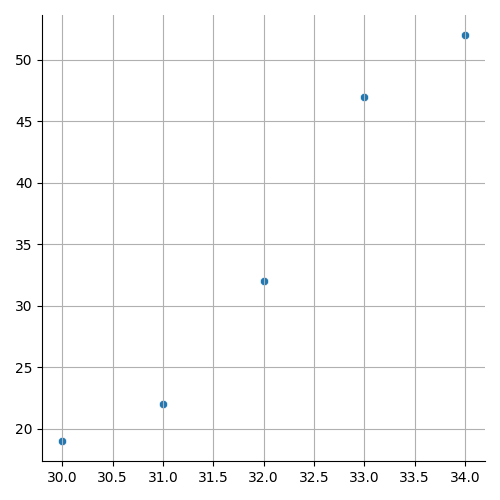


Drawing scatter plots using seaborn

sns.relplot(x=x, y=y)

plt.grid()

plt.show()



1. Goal of linear regression model

What we want to know by using linear regression is the relationship between x and y, i.e., the relationship function between the “sets” of Xs and Ys. Linear regression assumes a “linear” relationship between x and y, and the relationship function is thus defined as follows:

In such generated function, “a” becomes the coefficient and b becomes the intercept for explaining the set of Ys. Consider the following codes for simple understanding:

When samples are defined as:

x = np.array([30, 31, 32, 33, 34])

y = np.array([19, 22, 32, 47, 52])

the coefficient and intercept of this model are:

model = LinearRegression().fit(x.reshape(-1,1), y)

model.coef\_

model.intercept\_

The score of the generated model is:

model.score(x.reshape(-1,1), y)

More specifically, this score is known as “r2” score, which we will study in depth later.

Now, let us visualize once again.

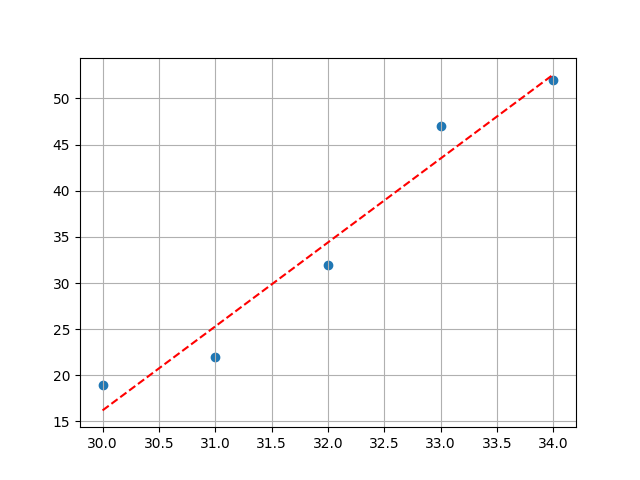
a, b = model.coef\_[0], model.intercept\_

plt.scatter(x, y)

plt.plot(x, a\*x+b, linestyle='--', color='red' )

plt.grid()

plt.show()



Now it is possible to make “predictions” using this model by designating inputs.

Let us see what would be the prediction for y when x is 60.

a\*60 + b

or,

model.predict([[60]])

1. Multiple Regression

Multiple regression refers to the case where there are multiple numbers of X variables (i.e., x1, x2, x3...) to explain Y. For better understanding, consider following sample.

Sample generation:

np.random.seed(12)

x1 = np.random.randn(800) \* 70

x2 = np.random.randn(800) \* 70

y = 7\*x1 + 2\*x2 - 90 + np.random.randn(800) \* 70

Visualization of the sample data:

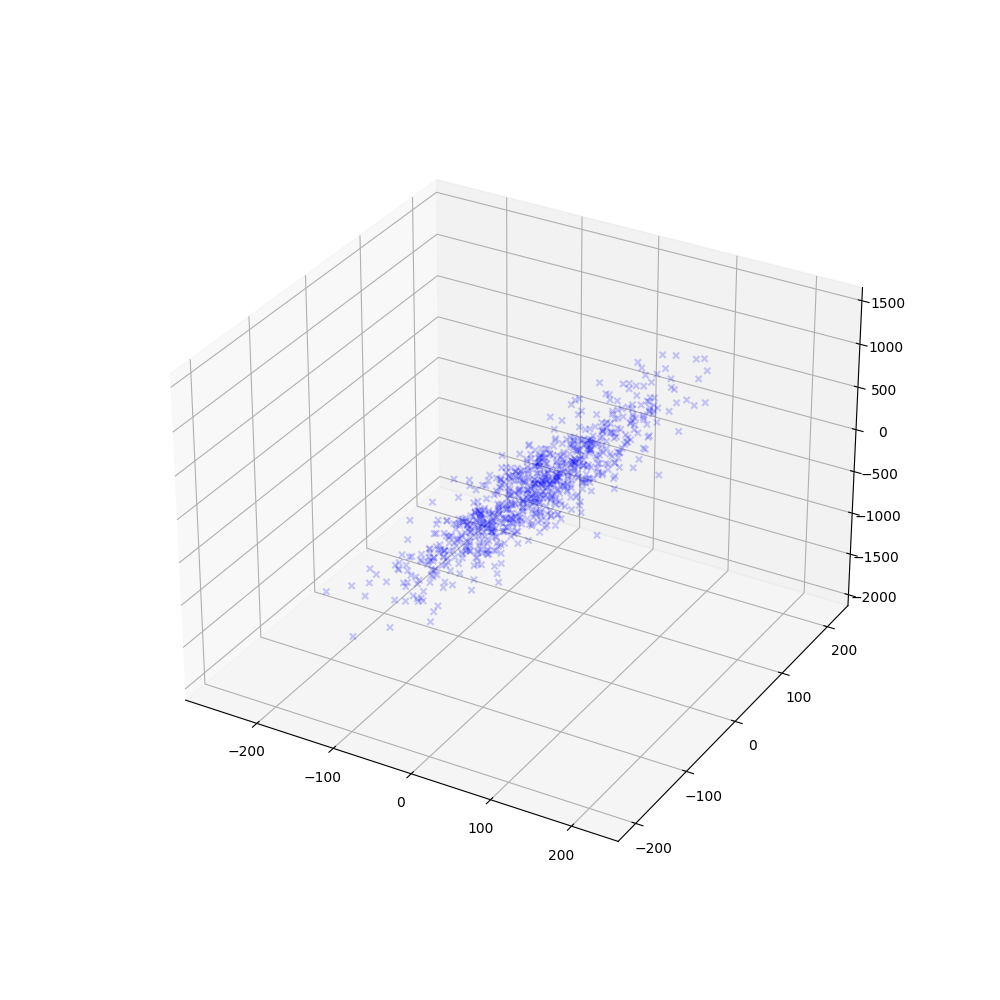
fig = plt.figure(figsize=(10, 10))

ax = fig.add\_subplot(projection='3d')

ax.scatter(x1, x2, y, marker='x', color='b', alpha=0.2)

plt.grid()

plt.show()



The function for this model can be defined as follows:

Let us find out the coefficient, intercept, and score for this model.

x = np.c\_[x1, x2]

model2d = LinearRegression().fit(x, y)

model2d.coef\_

model2d.intercept\_

model2d.score(x, y)

Output:

Coefficients: 7, 1.9

Intercept: -93.4

Score: 0.98

Visualization:

fig = plt.figure(figsize=(10,5))

ax = fig.add\_subplot(projection='3d')

ax.scatter(x1, x2, y, marker=',', color='green', alpha=0.1)

xx1 = np.tile(np.arange(-150, 150), (300,1))

xx2 = np.tile(np.arange(-150, 150), (300,1)).T

beta1 = model2d.coef\_[0]

beta2 = model2d.coef\_[1]

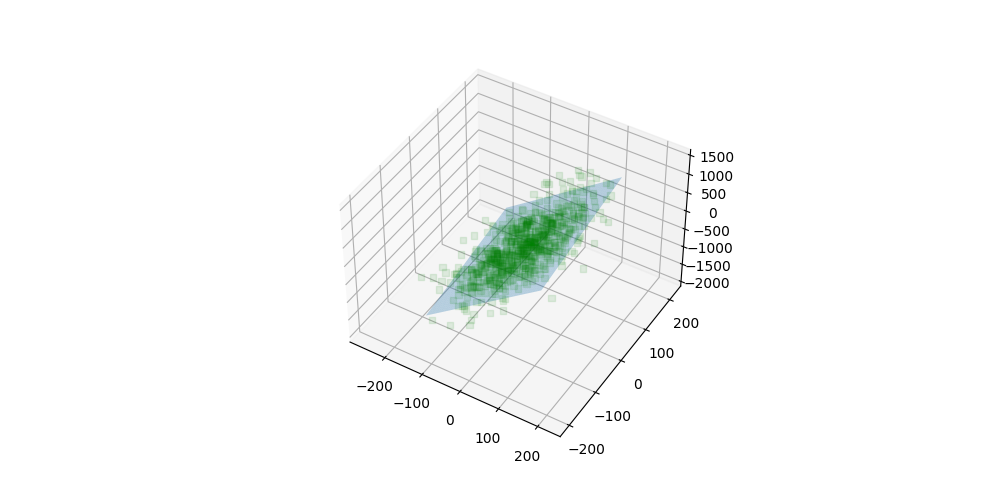
b = model2d.intercept\_

yy = beta1 \* xx1 + beta2 \* xx2 + b

ax.plot\_surface(xx1, xx2, yy, alpha=0.3)

plt.grid()

plt.show()



1. Polynomial Regression

Polynomial regression modifies the X variable when the relationship of variables is considered as non-linear. The main difference between multiple regression and polynomial regression is that polynomial regression either (1) expands the dimension of the existing X variable or (2) applies new function in the existing dataset.

The representative, well-known polynomial models are:

1. Quadratic Model
2. Cubic Model
3. Exponential/Log model

Let us explore all three models in order.

* 1. Quadratic Model

Sample data generation and visualization:

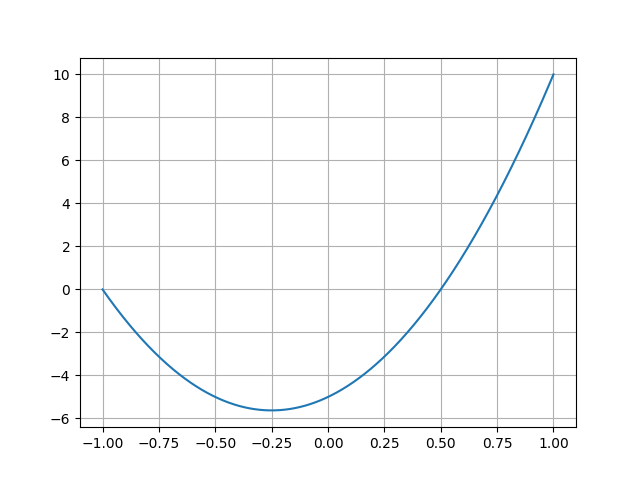
x = np.linspace(-1, 1, 800)

y = 5\*x + 10\*x\*\*2 -5

plt.plot(x, y)

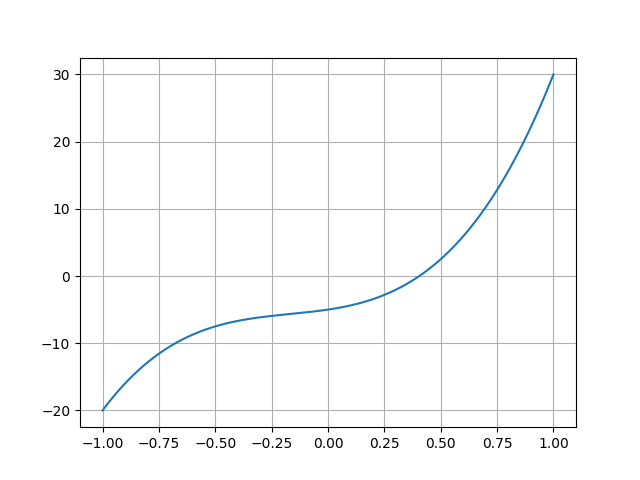
plt.grid()

plt.show()



* 1. Cubic Model

Sample data generation and visualization:



* 1. Exponential Model

Since exponential function and log function have functional/inverse-functional relationship, exponential models are usually converted into linear model in actual applications:

Sample data generation and visualization:

x = np.linspace(-1, 1, 800)

y = np.exp(10\*x)

plt.figure(figsize=(10, 5))

plt.subplot(1,2,1)

plt.plot(x,y)

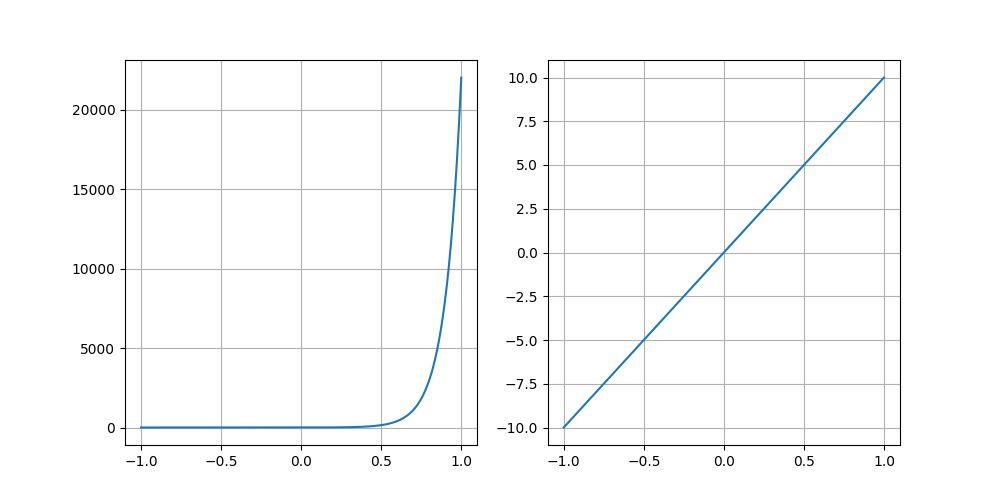
plt.grid()

plt.subplot(1,2,2)

plt.plot(x, np.log(y))

plt.grid()

plt.show()



1. Application of Regression Models

Now, let us apply regression models to explain actual datasets. Consider following sample for practice. Set random seed as “1234” to get same sample.

Sample data generation:

np.random.seed(1234)

x\_train = np.linspace(0, 1, 80)

y\_train = np.sin(1.8 \* np.pi \* x\_train) + (np.random.randn(80)/10)

x\_test = np.linspace(0, 1, 20)

y\_test = np.sin(1.8 \* np.pi \* x\_test) + (np.random.randn(20)/10)

Visualize sample data:

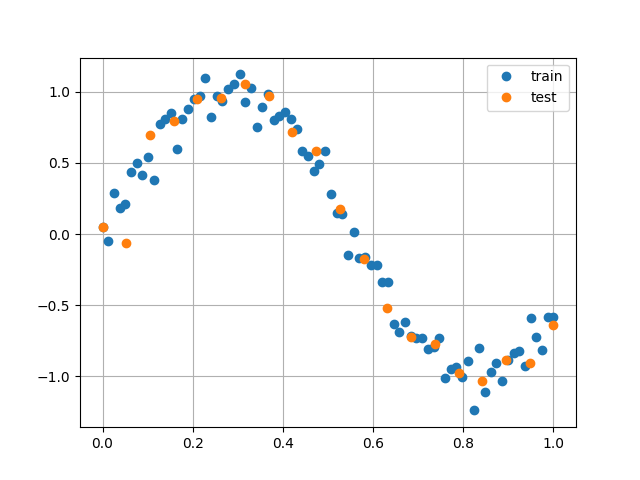
plt.plot(x\_train, y\_train, 'o', label='train')

plt.plot(x\_test, y\_test, 'o', label='test')

plt.legend()

plt.grid()

plt.show()



Reshape sample for analysis:

x\_train = x\_train.reshape(-1,1)

x\_test = x\_test.reshape(-1,1)

* 1. Application of Linear Regression Model

linearModel = LinearRegression().fit(x\_train, y\_train)

linearModel.coef\_

linearModel.intercept\_

linearModel.score(x\_train, y\_train)

linearModel.score(x\_test, y\_test)

Visualization:

plt.plot(x\_train, y\_train, 'o', label='train')

plt.plot(x\_test, y\_test, 'x', label='test')

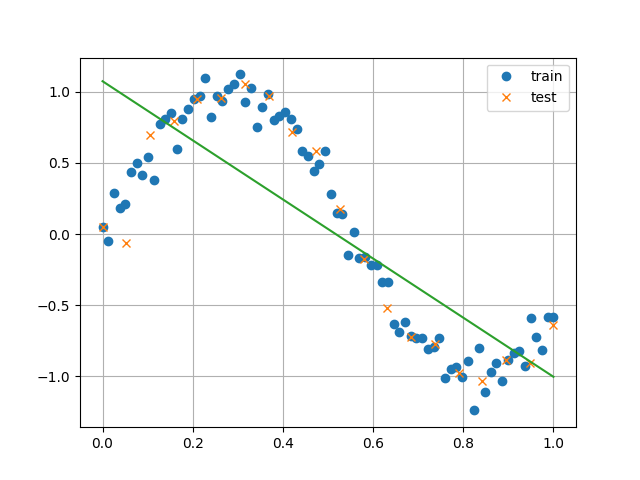
#plt.plot(x\_train, linearModel.predict(x\_train))

plt.plot(x\_train, linearModel.coef\_[0] \* x\_train + linearModel.intercept\_)

plt.legend()

plt.grid()

plt.show()



* 1. Application of Quadratic Model

x\_quadratic = np.c\_[x, x\*\*2]

x\_quadratic.shape

x\_train\_quadratic = np.c\_[x\_train, x\_train\*\*2]

x\_test\_quadratic = np.c\_[x\_test, x\_test\*\*2]

quadraticModel = LinearRegression().fit(x\_train\_quadratic, y\_train)

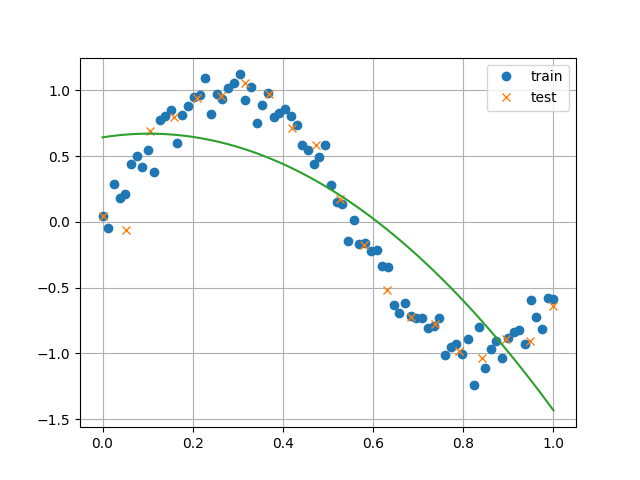
quadraticModel.coef\_

quadraticModel.intercept\_

quadraticModel.score(x\_train\_quadratic, y\_train)

quadraticModel.score(x\_test\_quadratic, y\_test)

Visualization:



* 1. Application of Cubic Model

x\_qubic = np.c\_[x, x\*\*2, x\*\*3]

x\_train\_qubic = np.c\_[x\_train, x\_train\*\*2, x\_train\*\*3]

x\_test\_qubic = np.c\_[x\_test, x\_test\*\*2, x\_test\*\*3]

qubicModel = LinearRegression().fit(x\_train\_qubic, y\_train)

qubicModel.coef\_

qubicModel.intercept\_

qubicModel.score(x\_train\_qubic, y\_train)

qubicModel.score(x\_test\_qubic, y\_test)

Visualization:

plt.plot(x\_train, y\_train, 'o', label='train')

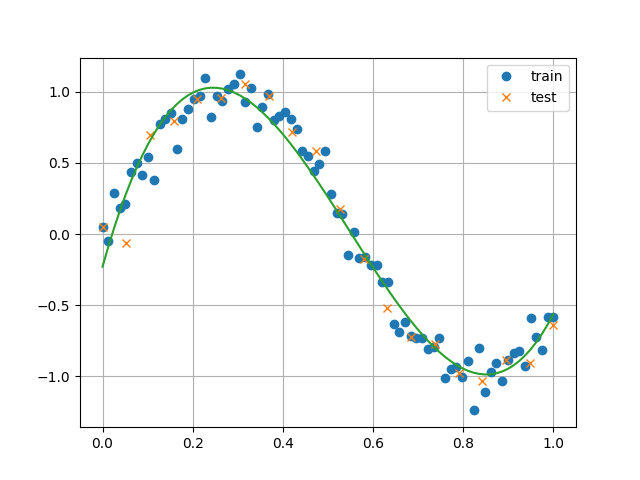
plt.plot(x\_test, y\_test, 'x', label='test')

plt.plot(x\_train, qubicModel.predict(x\_train\_qubic))

plt.grid()

plt.legend()

plt.show()



As it can be implied from the process of applying various regression models, it is foremost significant to understand the feature of data and to decide which type of regression model to find out the best-fit model. In this context, visualizing every visualizable aspect regarding the data is an important tip to remember.

As you can see, the above cubic regression model that we have generated can be defined as the “best-fit” model for the generated sample dataset; and the rest are “under-fit” models. There is yet one more model to avoid: “over-fit” models.

* 1. Over-fit models

Consider the following sample and a polynomial model for illustration.

Sample generation:

np.random.seed(1234)

x = np.linspace(0, 1, 1000)

y = np.sin(2 \* np.pi \* x) + (np.random.randn(1000)/5)

x\_train = np.linspace(0, 1, 11)

y\_train = np.sin(2 \* np.pi \* x\_train) + (np.random.randn(11)/5)

x\_test = np.linspace(0, 1, 50)

y\_test = np.sin(2 \* np.pi \* x\_test) + (np.random.randn(50)/5)

Model generation:

x\_poly = np.c\_[

  x, x \*\* 2, x \*\* 3, x \*\* 4, x \*\* 5, x \*\* 6, x \*\* 7, x \*\* 8, x \*\* 9, x\*\*10

]

x\_train\_poly = np.c\_[

  x\_train, x\_train \*\* 2, x\_train \*\* 3, x\_train \*\* 4, x\_train \*\* 5, x\_train \*\* 6, x\_train \*\* 7, x\_train \*\* 8, x\_train \*\* 9, x\_train\*\*10

]

x\_test\_poly = np.c\_[

  x\_test, x\_test \*\* 2, x\_test \*\* 3, x\_test \*\* 4, x\_test \*\* 5, x\_test \*\* 6, x\_test \*\* 7, x\_test \*\* 8, x\_test \*\* 9, x\_test\*\*10

]

Overfit model:

polyModel = LinearRegression().fit(x\_train\_poly, y\_train)

polyModel.score(x\_train\_poly, y\_train)

polyModel.score(x\_test\_poly, y\_test)

In this case, the output for trainset score is 1.0 while the output for testset score is -4.8. This is a typical case of over-fitting model problem, which you have to avoid in terms of modelling.

Consider the following visualization for better understanding:

