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 general 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 model. Neural network algorithm, for example, can be summarized as the process of "piling" linear models, and converting them into non-linear models via applying activation functions (e.g., sigmoid, relu) throughout multiple (hidden) layers. As we will see throughout this chapter, the irreplaceable advantages of linear regression model [(1) structural intuitiveness, (2) outstanding analytical interpretability] become the reason for wide ranges of applicability. Enough with words, let's get right into the world of linear models with codes!

2. Generating Sample Data and Visualization

Import compulsory libraries (numpy for generating simple set of samples, matplotlib and seaborn for visualization)

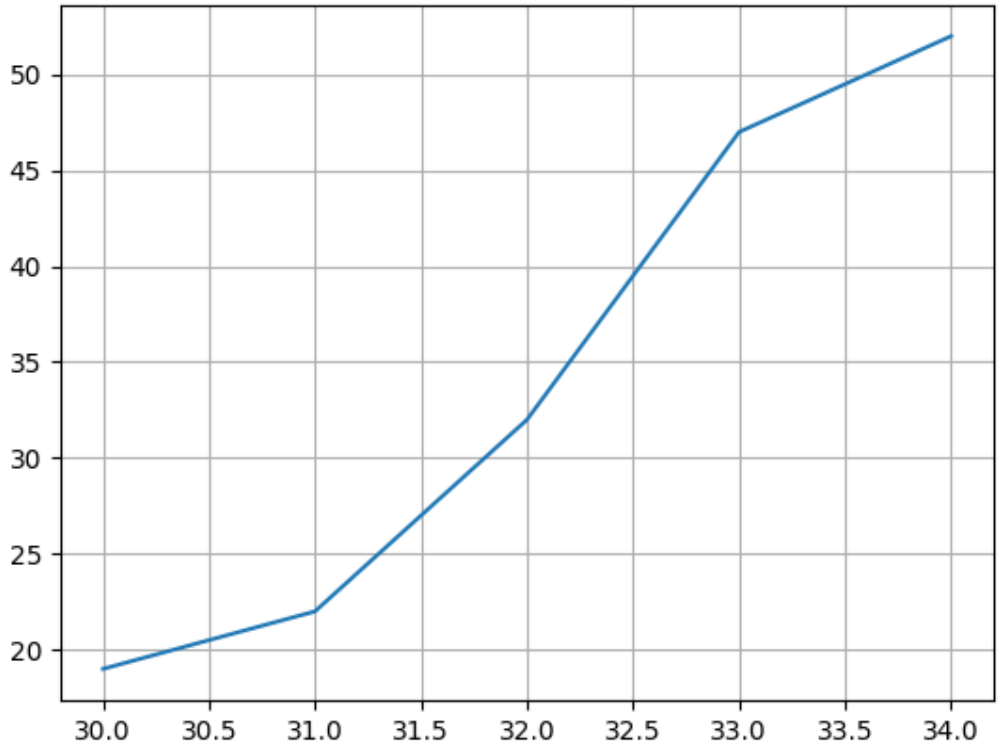
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Generate samples for linear regression, x and y, using numpy arrays

```
x = np.array([30, 31, 32, 33, 34])
y = np.array([19, 22, 32, 47, 52])
```

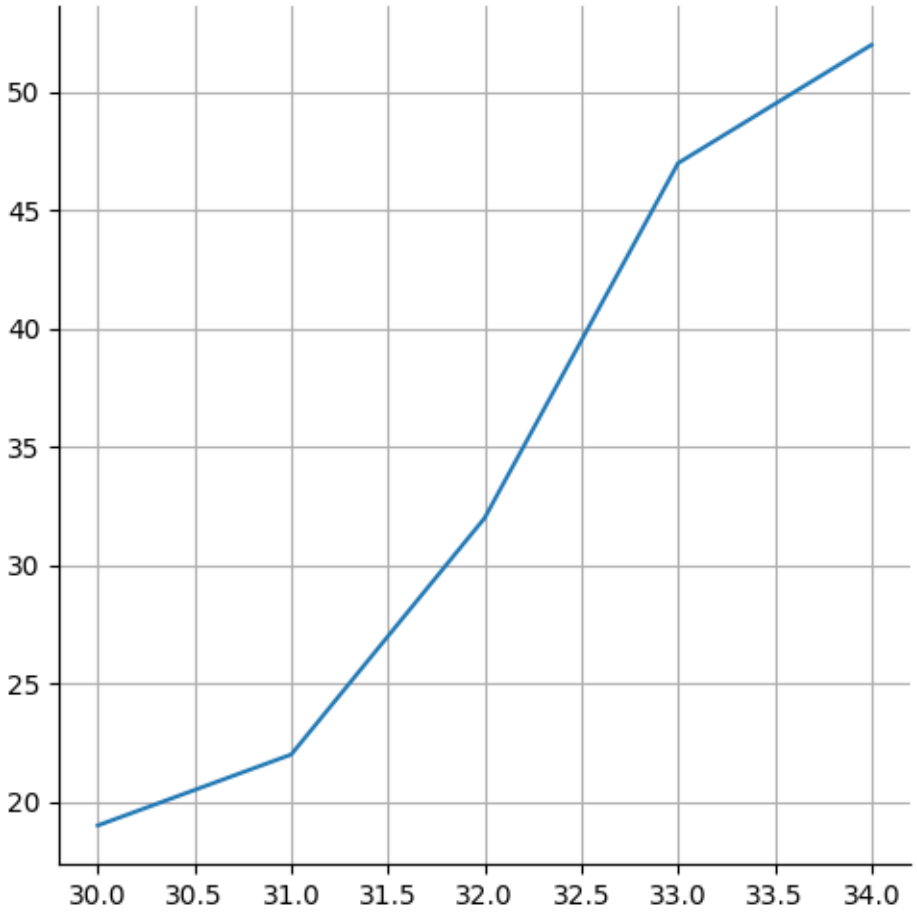
Drawing line plots (matplotlib)

```
plt.plot(x, y)
plt.grid()
plt.show()
```



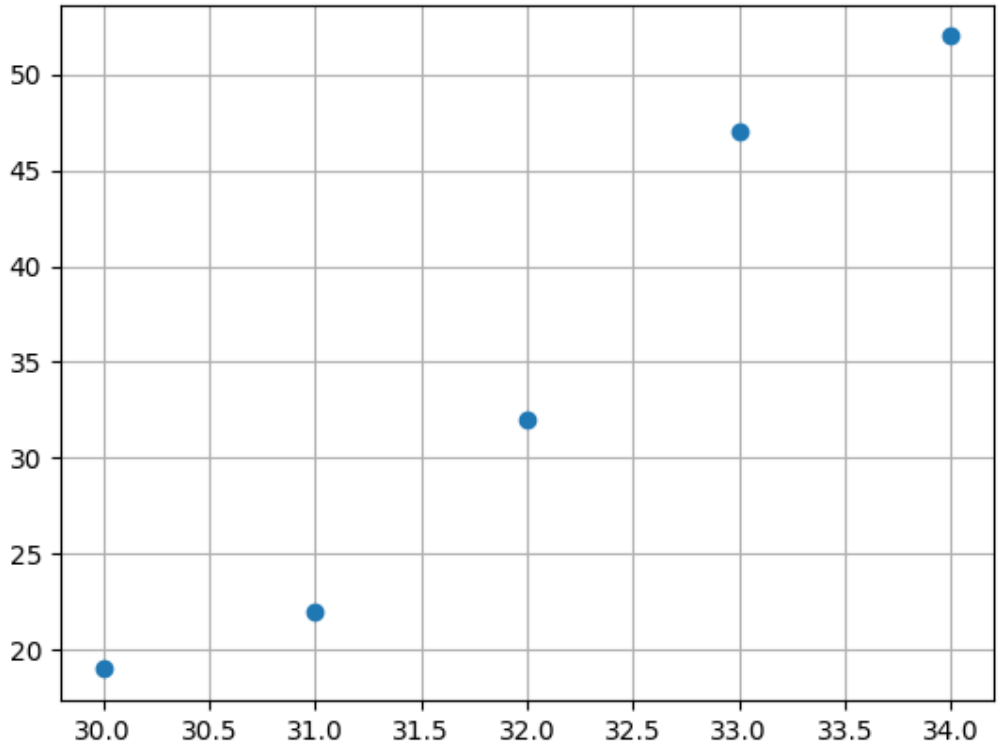
Drawing line plots (seaborn)

```
sns.relplot(x=x, y=y, kind='line')
plt.grid()
plt.show()
```



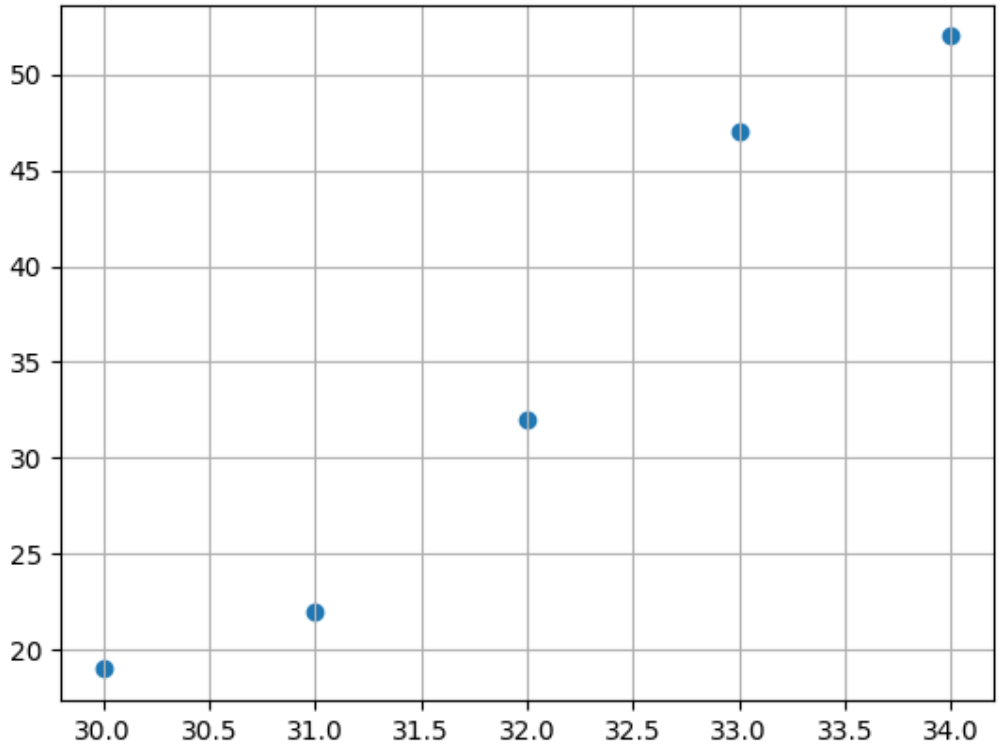
Drawing scatter plots using matplotlib (1)

```
plt.plot(x, y, 'o')
plt.grid()
plt.show()
```



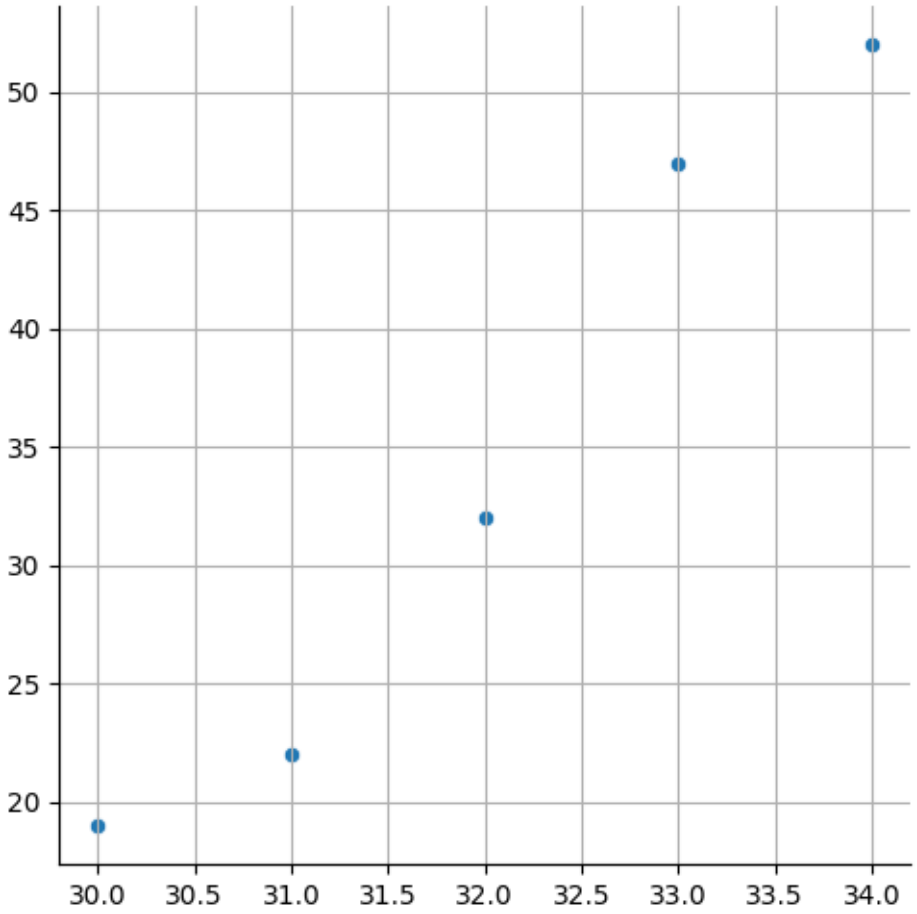
Drawing scatter plots using matplotlib (2)

```
plt.scatter(x, y)
plt.grid()
plt.show()
```



Drawing scatter plots using seaborn

```
sns.relplot(x=x, y=y)
plt.grid()
plt.show()
```



3. Goals of linear regression model

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

$$f(x) = ax + b$$

In such generated function, "a" and "b" are, respectively, referred to as the coefficient and intercept for explaining the set of Ys. Consider the following codes for better 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_
```

Output:

array([9.1])

-256.8

The score of the generated model is:

```
model.score(x.reshape(-1,1), y)
```

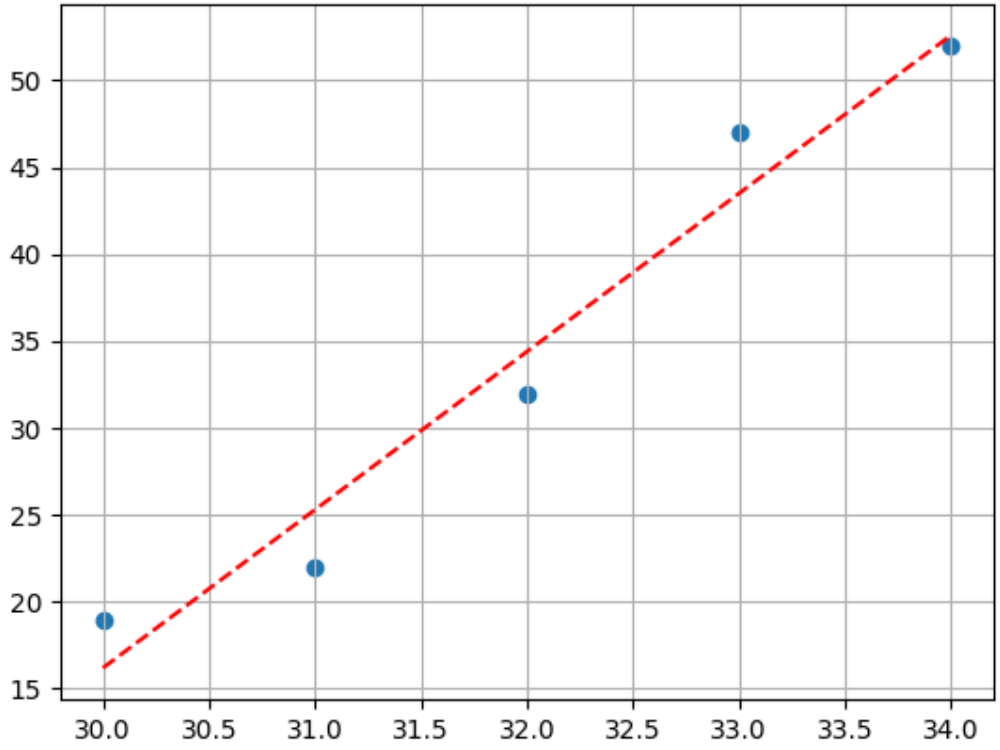
Output:

0.957119741100324

More specifically, this “score” indicates the “R2” score, which we will study in depth in the future subchapters.

For now, let us visualize once again, to comprehend the general idea.

```
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
```

Output:

289.2

or,

```
model.predict([[60]])
```

Output:

289.2

4. Multiple Regression

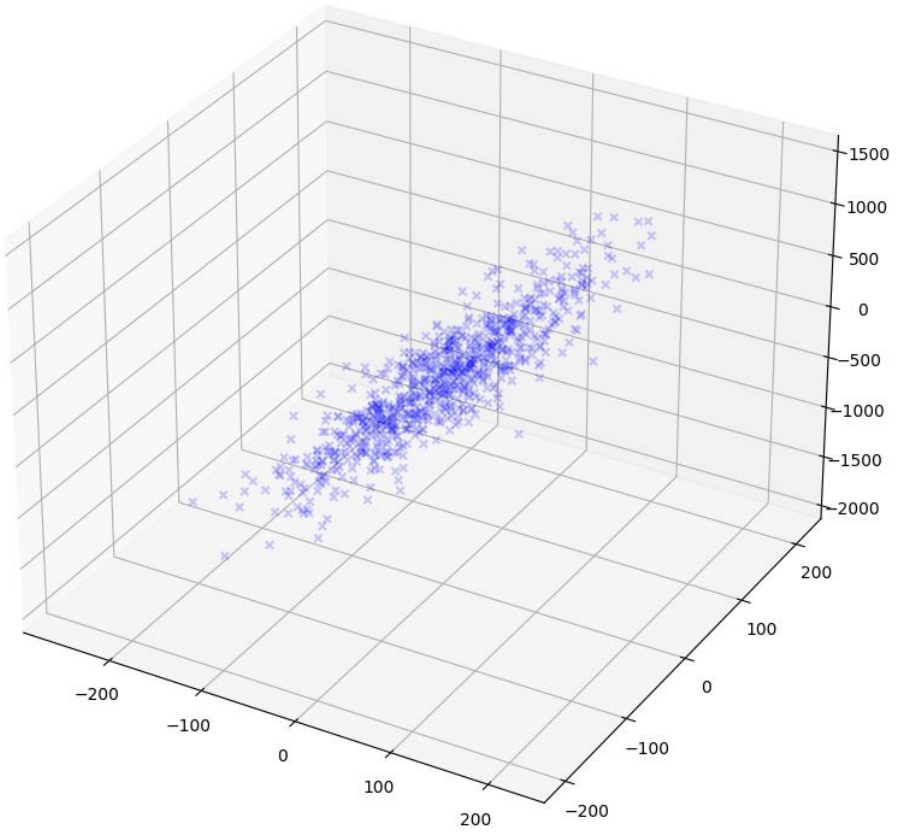
Multiple regression refers to the case where there are multiple numbers of X variables (i.e., x1, x2, x3...) to explain Y. For better understanding, consider following sample.

Sample generation (x1, x2, y):

```
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()
```



*Note that the visualization is 3-dimensional, due to the addition of another X variable.

The function for this model can be defined as follows:

$$f(x) = \beta_1 x_1 + \beta_2 x_2 + b$$

Then, let us find out the coefficients, 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:

array([6.95871615, 1.91858033]) (Coefficients)

-93.4293256574557 (Intercept)

0.9819312871833802 (Score)

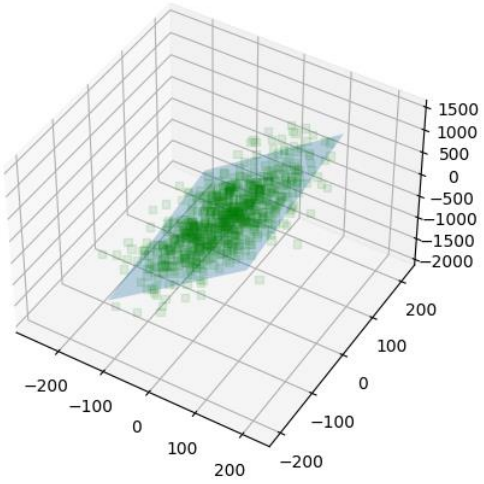
Visualization of the fitted model:

```
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()
```



*Note that the fitted model is 2-dimensional (shaded blue), in order to explain the 3-dimensional dataset.

5. Polynomial Regression

Polynomial regression modifies the X variable when the relationship of variables is assumed to be 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 include:

(1) Quadratic Model

(2) Cubic Model

(3) Exponential/Log model

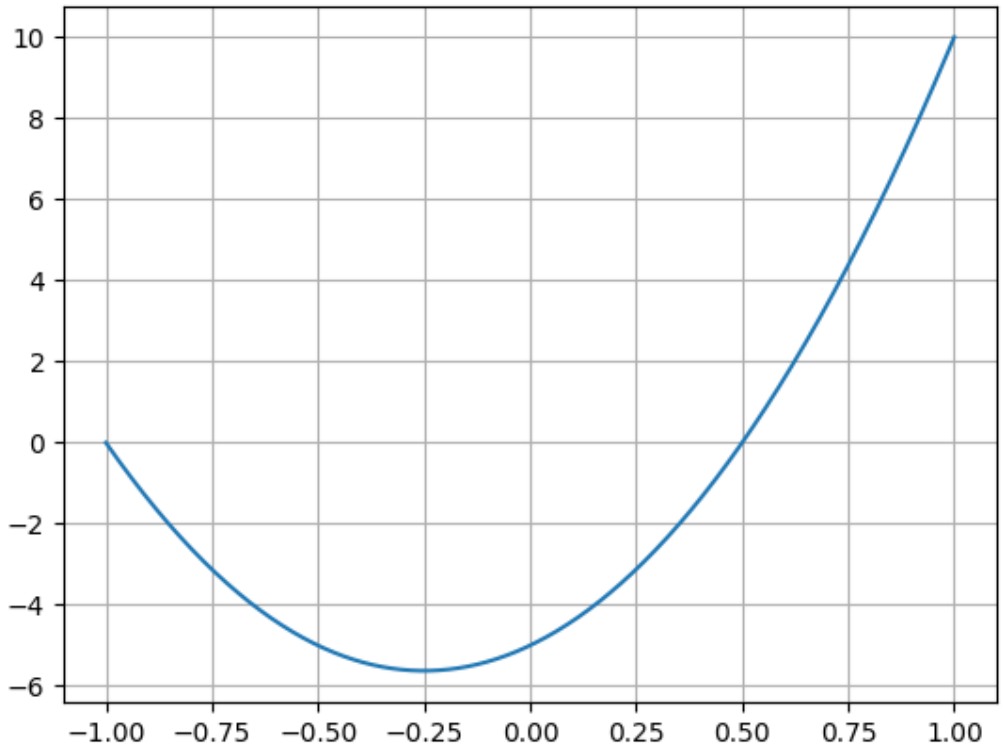
Let us explore all three models in order.

5.1. Quadratic Model

$$f(x) = \beta_1 x_1 + \beta_2 x_1^2 + b$$

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()
```

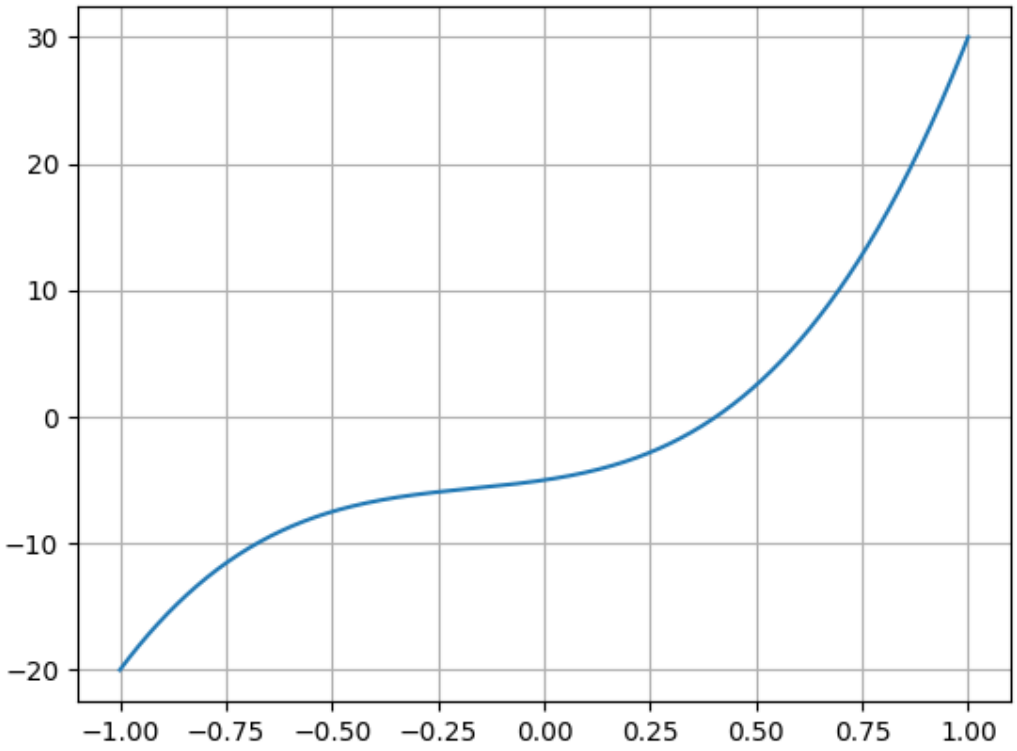


5.2. Cubic Model

$$f(x) = \beta_1 x_1 + \beta_2 x_1^2 + \beta_2 x_1^3 + b$$

Sample data generation and visualization:

```
x = np.linspace(-1, 1, 800)
y = 5*x + 10*x**2 + 20*x**3 - 5
plt.plot(x, y)
plt.grid()
plt.show()
```



5.3. Exponential Model

$$f(x) = e^{\beta_1 x_1}$$

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

$$g(x) = lnf(x) = \beta_1 x_1$$

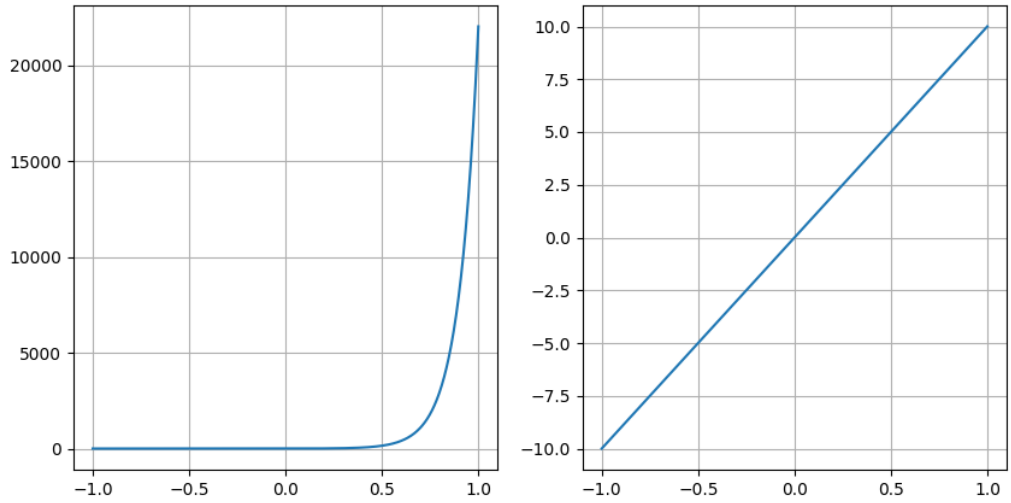
Sample data generation and visualization (original exponential model & converted model using *log function*):

```
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()
```



6. Application of the Regression Models

Now, let us apply linear regression to model actual datasets. Consider following sample for practice. Set random seed as “1234” to get the identical sample.

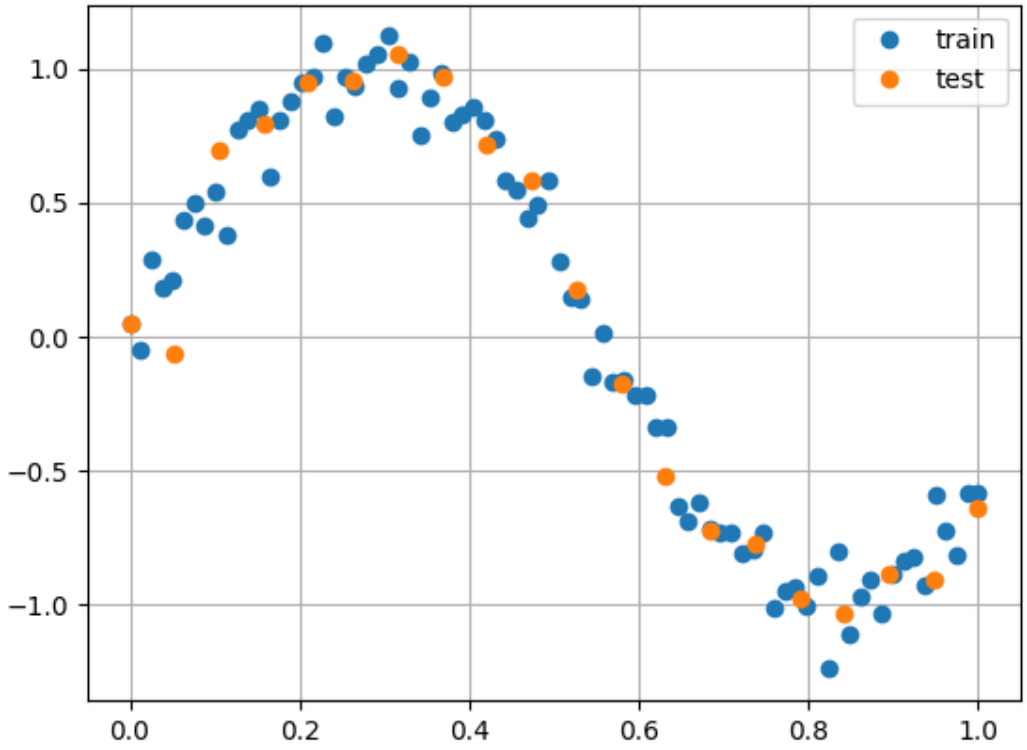
Sample data generation (train sets and test sets):

```
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)
```

6.1. 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)
```

Output:

array([-2.07536924])

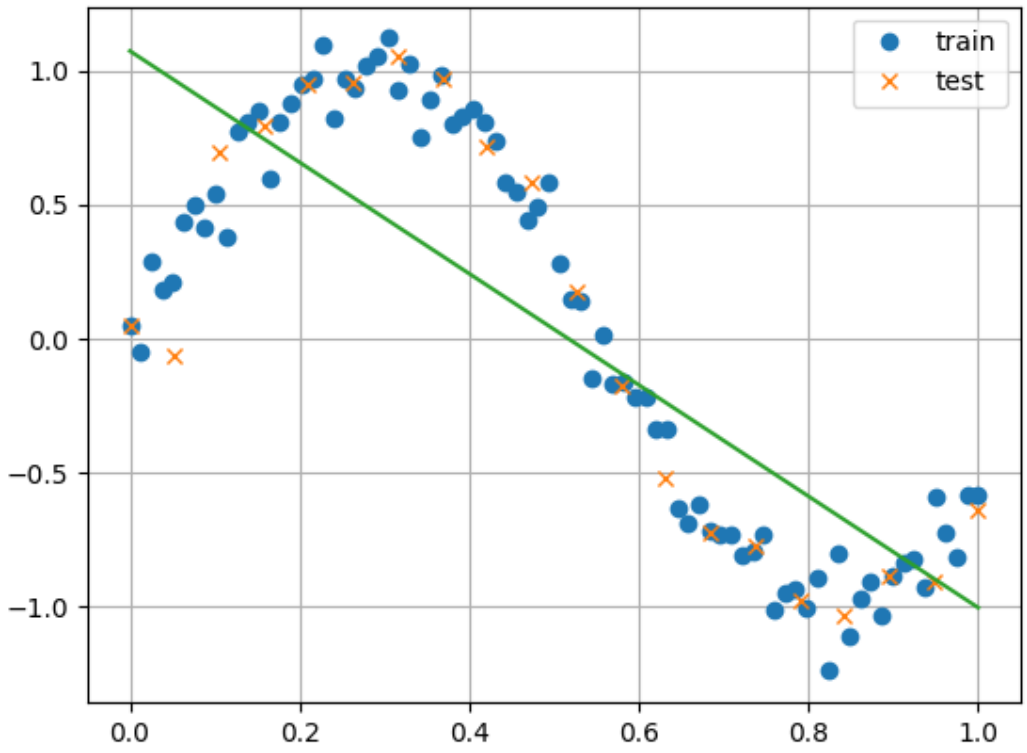
1.0730758436542476

0.670684762866159

0.5959305682064919

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()
```



6.2. 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)
```

Output:

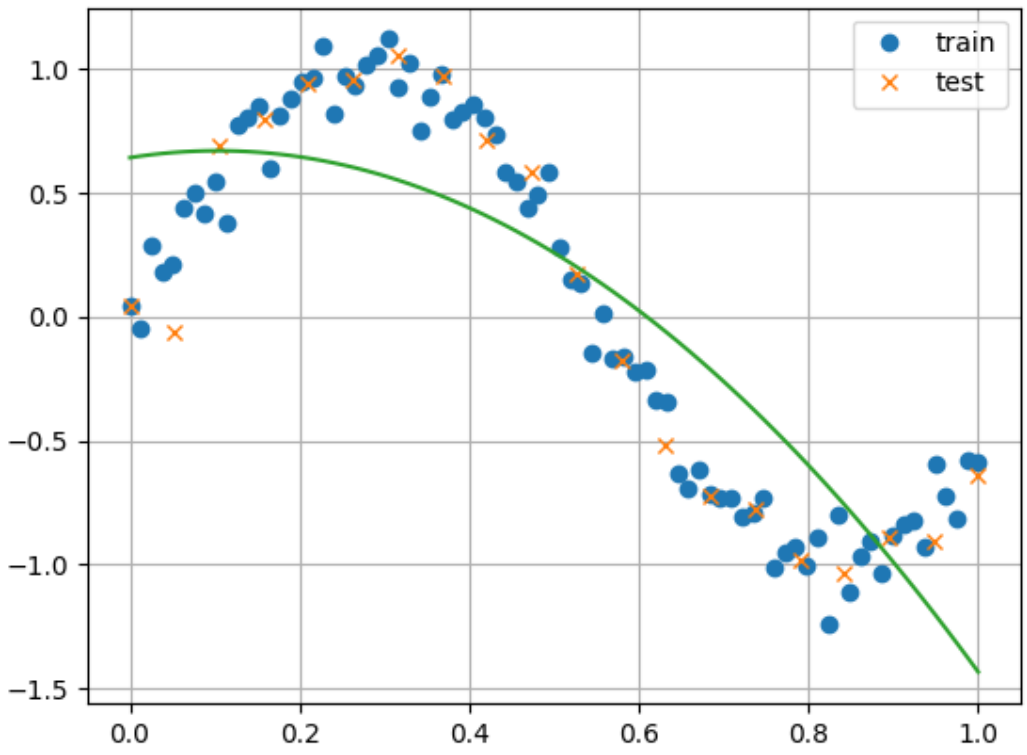
array([0.54045064, -2.61581988])

0.6426244711578968

0.7434805717228434

0.6965565807397845

Visualization:



6.3. 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)
```

Output:

array([11.3401839 , -29.7856629 , 18.11322868])

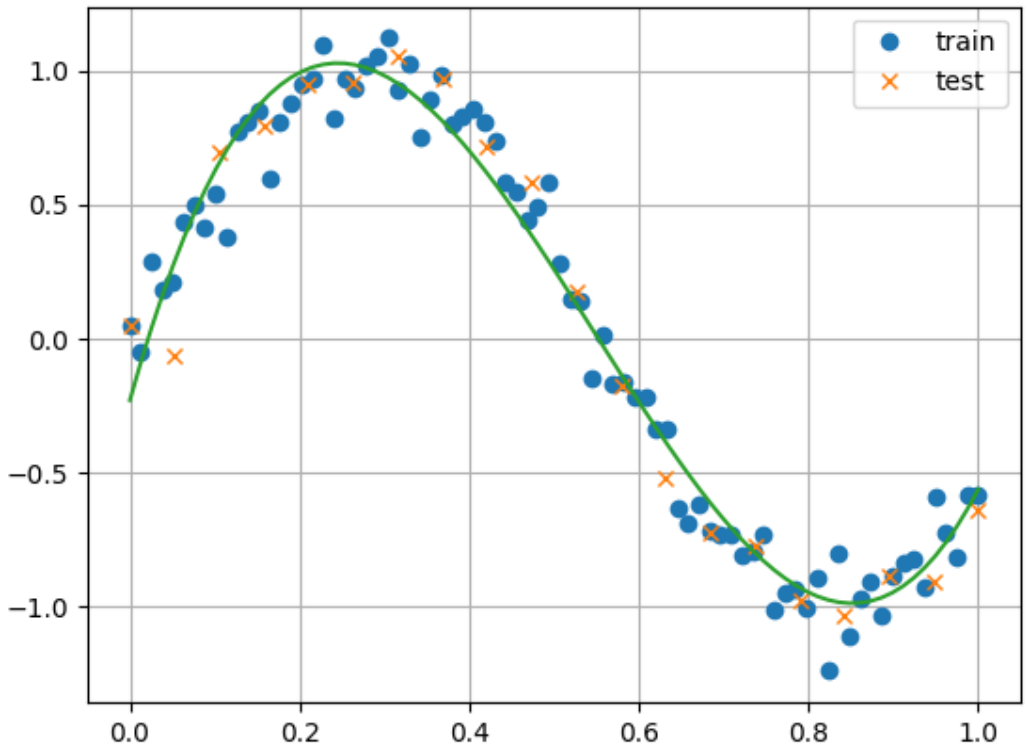
-0.2289349861270556

0.9732602162507141

0.9680301841676737

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 implied from the process of applying various regression models, it is foremost significant to understand the features and shapes of data and to accordingly decide which type of regression model to generate a 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 sample dataset; the rest are “under-fit” models, which should be avoided in terms of modelling. In addition, there is yet one more type of model to avoid: the “over-fit” models.

6.4. 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_t
est ** 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)
```

Output:

1.0

-4.829141615876579

The train-set score is 1.0 (100%), while the test-set score is -4.8. This is a typical case of over-fitting model problem, which you have to avoid in terms of modelling.

Refer to the following visualization of the over-fit model for better understanding:

```
plt.plot(x_train, y_train, 'o', label='train')
plt.plot(x_test, y_test, 'o', label='test')
plt.plot(x, polyModel.predict(x_poly))
plt.legend()
plt.grid()
plt.show()
```