Simple examples to ilustrate the concepts of underfitten and overfitting

In [1]:

```
import numpy as np
import random

from sklearn import datasets, linear_model
from sklearn.preprocessing import PolynomialFeatures
from matplotlib import pyplot as plt
%matplotlib inline

import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

np.random.seed(1)
```

The samples described with the coordinates (X, Y) are generated using the following equation:

$$Y = g(X) + \eta$$

$g(X)$ is the function to generate the samples, η is a noise function defined with normal distribution.

To model the correlations between the vairable X and Y, we will consider a family of functions $F_\alpha(W_\alpha, X)$ that depend on the parameters $W_\alpha = (w_{\alpha 1}, w_{\alpha 2}, w_{\alpha 3}, \dots)$.

Getting samples

Modeling the samples

In [2]:

```
# Number of samples
N_samples=16

#N_samples equally spaced values are generated from 0.05 to 0.95
x=np.linspace(0.05,0.95,N_samples)

# Gaussian uncorrelated noise
# We generate N_samples values at random wint a normal distribution

sigma_samples=0.3    #Maximal noise amplitud
s = sigma_samples*np.random.randn(N_samples)

# polynomial of order 10
y= 2*x - 5*x**5 + 15*x**10 + s

samples = []
for i in range(N_samples) :
    samples.append((x[i], y[i]))
```

We consider a function to generate the samples

In [3]:

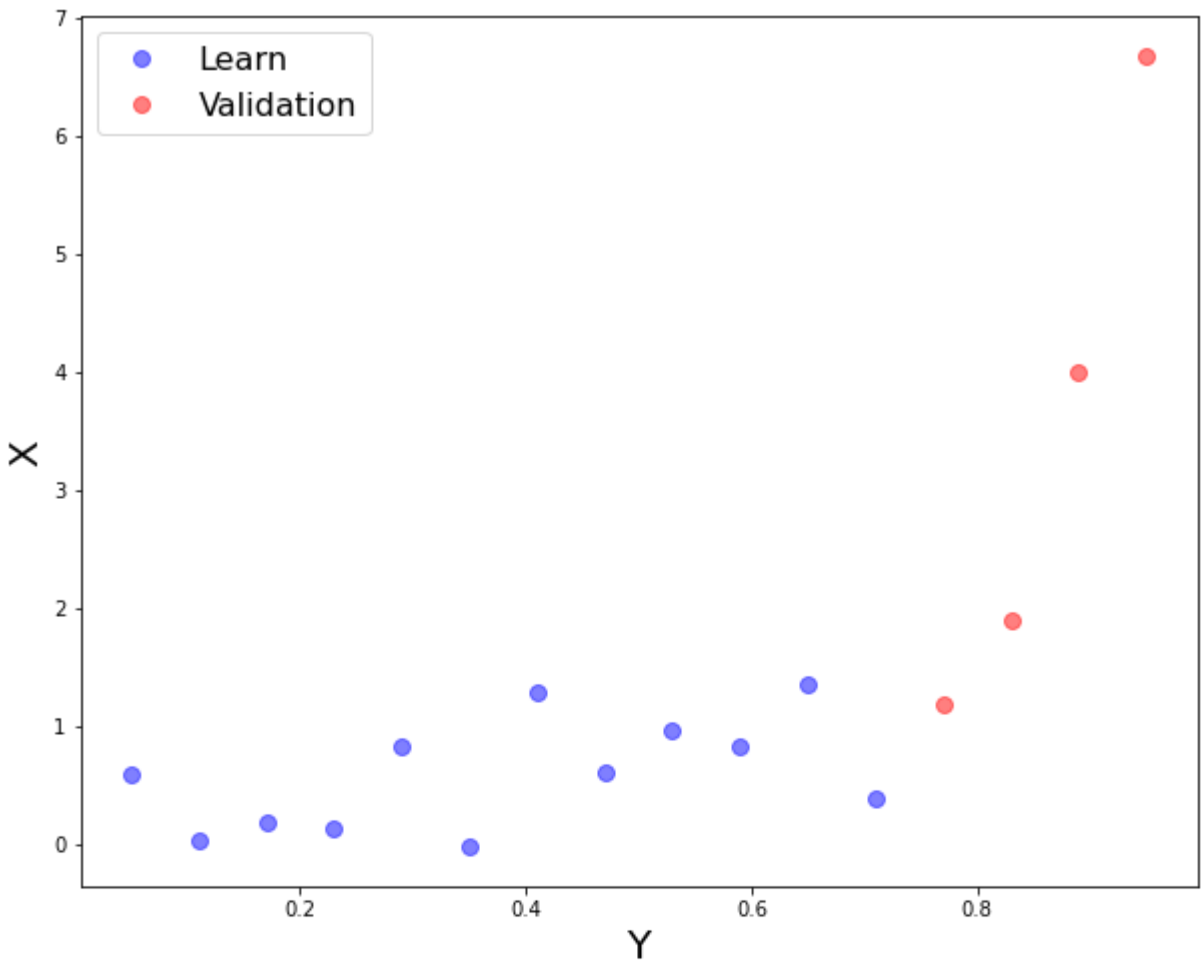
```
def split_samples(samples, val_ratio=0.2, shuffle=True):

    if shuffle==True:
        random.shuffle(samples)

    learn_ratio = int((1.0-val_ratio)*len(samples))
    learn = samples[0:learn_ratio]
    val = samples[learn_ratio:]

    learn_x=[]
    learn_y=[]
    for i in range(len(learn)):
        learn_x.append(learn[i][0])
        learn_y.append(learn[i][1])

    x_learn = np.array(learn_x)
    y_learn = np.array(learn_y)

    val_x=[]
    val_y=[]
    for i in range(len(val)):
        val_x.append(val[i][0])
        val_y.append(val[i][1])

    x_val = np.array(val_x)
    y_val = np.array(val_y)

    return x_learn, y_learn, x_val, y_val
```

In [4]:

```
val_ratio = 0.2
x_learn, y_learn, x_val, y_val = split_samples(samples, val_ratio=val_ratio, s

fig = plt.figure(figsize=(10, 8))
plt.ylabel('X', size=20)
plt.xlabel('Y', size=20)
plt.rc('xtick', labelsize=18)
plt.rc('ytick', labelsize=18)

p1=plt.plot(x_learn, y_learn, "o", ms=8, alpha=0.5, label='Training', color='k
p1=plt.plot(x_val, y_val, "o", ms=8, alpha=0.5, label='Training', color='red')
plt.legend(['Learn', 'Validation'], loc='upper left', prop={'size': 16})
```

Out[4]:

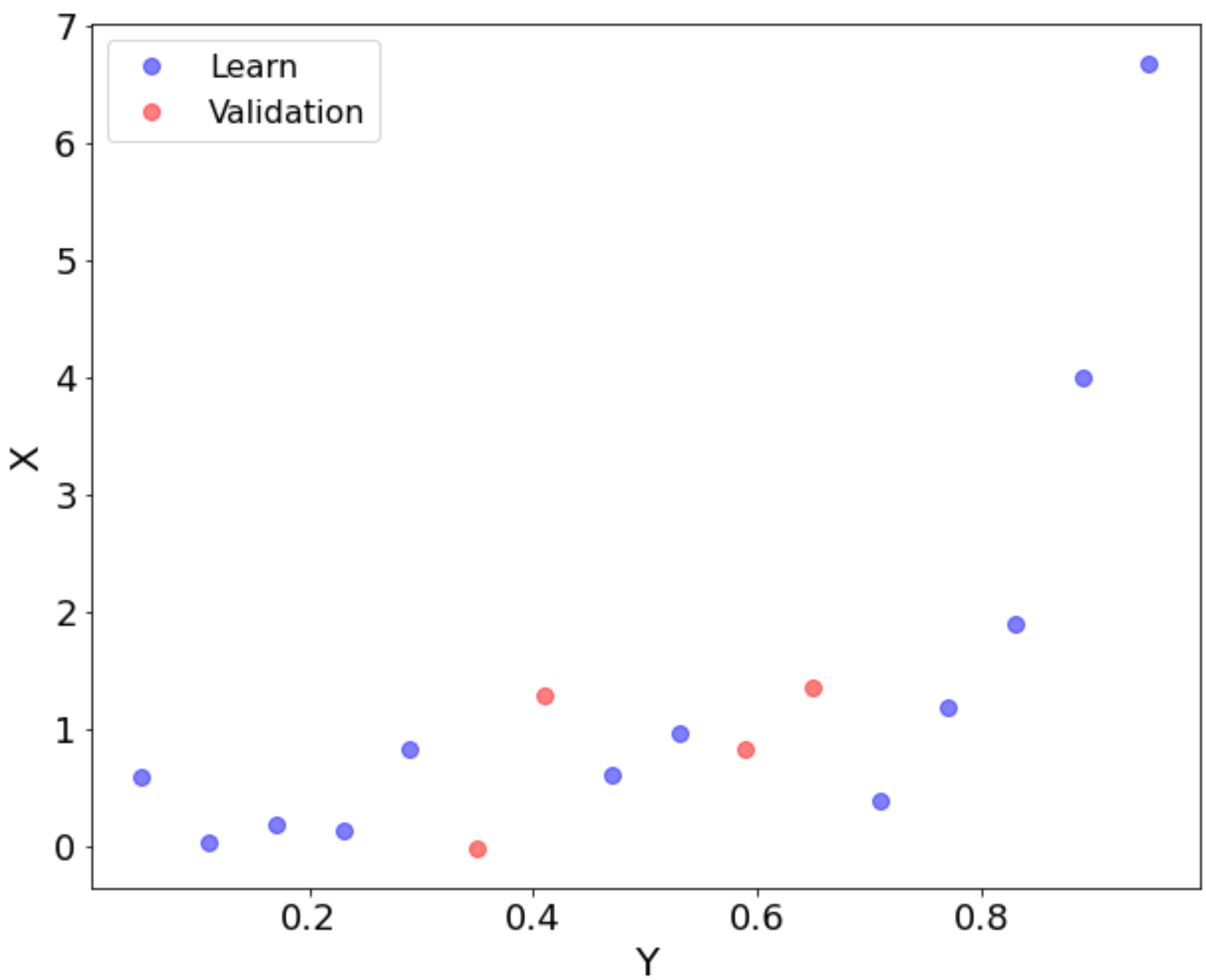
```
<matplotlib.legend.Legend at 0x7f9d6d415df0>
```



In [5]:

```
val_ratio = 0.2
x_learn, y_learn, x_val, y_val = split_samples(samples, val_ratio=val_ratio, s

fig = plt.figure(figsize=(10, 8))
plt.ylabel('X', size=20)
plt.xlabel('Y', size=20)
plt.rc('xtick', labelsize=18)
plt.rc('ytick', labelsize=18)

p1=plt.plot(x_learn, y_learn, "o", ms=8, alpha=0.5, label='Training', color='b
p1=plt.plot(x_val, y_val, "o", ms=8, alpha=0.5, label='Training', color='red')

plt.legend(['Learn', 'Validation'], loc='upper left', prop={'size': 16})
```

Out[5]: <matplotlib.legend.Legend at 0x7f9d6cb672e0>



In [6]:

```
print(x_learn.shape)
print(y_learn.shape)
```

```
(12,)
(12,)
```

n=2 poly = PolynomialFeatures(degree= n, include_bias=False, interaction_only=False)

x_learn_transf = np.expand_dims(x_learn, axis=1) x_learn_transf =

poly.fit_transform(x_learn_transf) x_val_transf = np.expand_dims(x_learn, axis=1) x_val_transf = poly.fit_transform(x_learn_transf) print(x_learn_transf.shape)

In [7]:

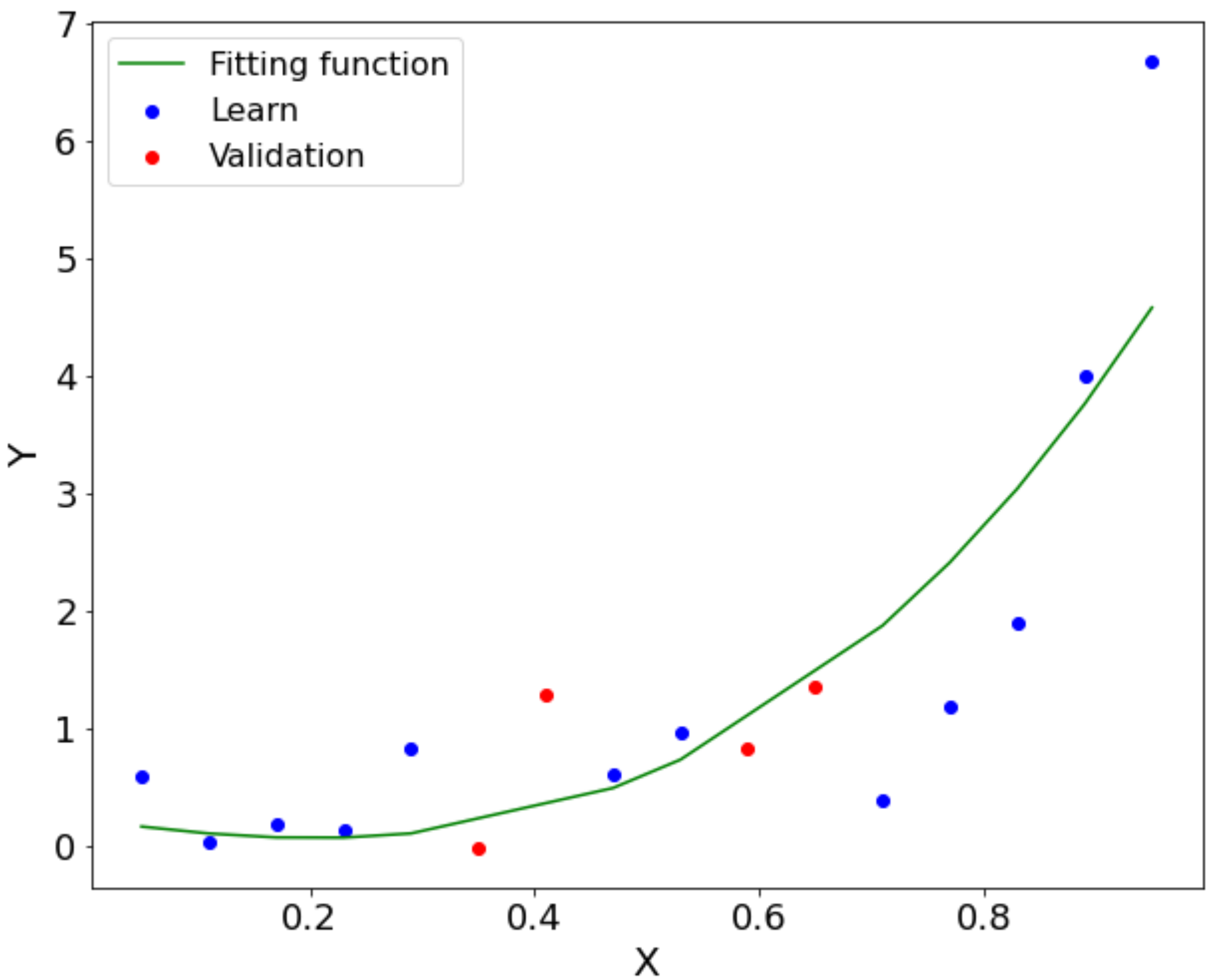
```
def transform_samples(x_learn, x_val, degree = 2):

    # The features will defined by the coeficientes of the polinomium
    # Therfore, only the X variables will be transformed
    x_learn_transf = np.expand_dims(x_learn, axis=1)
    x_val_transf = np.expand_dims(x_val, axis=1)

    # Define the number of features for the sample transformation
    # It can include or exclude a bias
    # It can include interaction between the polynomium term
    poly = PolynomialFeatures(degree=degree, include_bias=False, interaction_(

    x_learn_transf = poly.fit_transform(x_learn_transf)
    x_val_transf = poly.fit_transform(x_val_transf)

    return x_learn_transf, x_val_transf
```

In [8]:

```
def poly_fit(x_learn_pre, x_val_pre, x_learn, y_learn, x_val, y_val, epochs=5(

    inp = Input((n))
    #since one of the features is 1, we need an extra input
    out = Dense(1)(inp)
    model = Model(inputs=inp, outputs=out)
    model.compile(optimizer=Adam(lr=lr), loss="mean_squared_error")
    history=model.fit(x_learn, y_learn, epochs=epochs, validation_data=(x_val,
    y_predicted = model.predict(x_learn)
    y_predicted = np.squeeze(y_predicted,axis=1)

    x=list(x_learn_pre)
    y=list(model.predict(x_learn).squeeze())
    pairs=list(zip(x,y))
    pairs.sort(key=lambda a: a[0])
    pairs
    x_plot, y_plot = zip(*pairs)

    fig, ax = plt.subplots(figsize=(10, 8))
    plt.rc('xtick', labelsize=18)
    plt.rc('ytick', labelsize=18)
    plt.ylabel('Y', size=20)
    plt.xlabel('X', size=20)

    ax.scatter(x_learn_pre, y_learn, color='blue')
    ax.scatter(x_val_pre, y_val, color='red')
    ax.plot(x_plot, y_plot, color="green")

    plt.legend(['Fitting function', 'Learn', 'Validation'], loc='upper left',

    return history
```

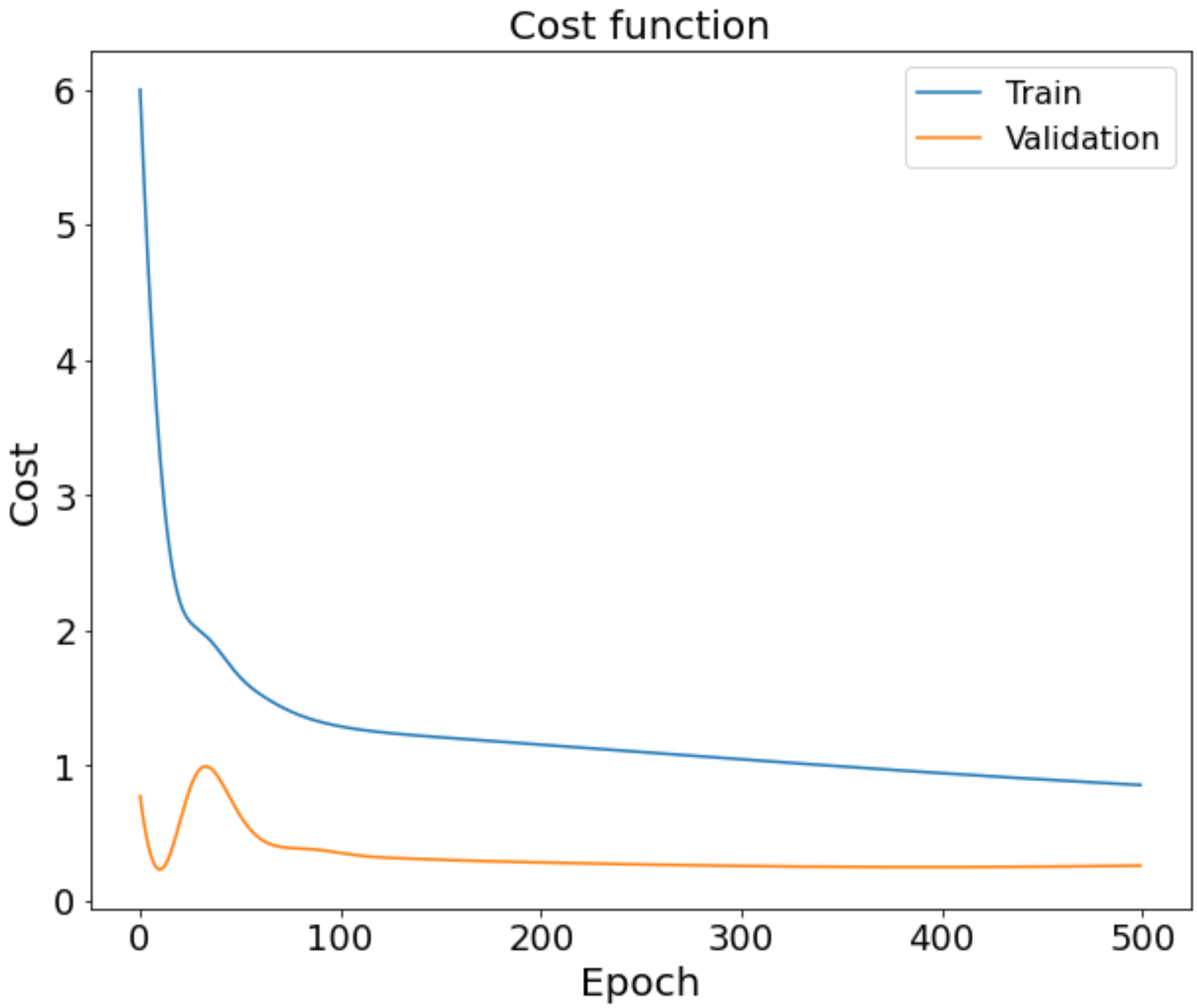
In [9]:

```
n=3
x_learn_transf, x_val_transf = transform_samples(x_learn, x_val, degree=n)

learning_rate = 0.04
epochs =500

history=poly_fit(x_learn, x_val, x_learn_transf, y_learn, x_val_transf, y_val,
```



In [10]:

```
fig = plt.figure(figsize=(10, 8))
plt.rc('xtick', labelsize=18)
plt.rc('ytick', labelsize=18)

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Cost function', size=20)
plt.ylabel('Cost', size=20)
plt.xlabel('Epoch', size=20)
plt.legend(['Train', 'Validation'], loc='upper right', prop={'size': 16})
plt.show()
```


Cost function
Train
Validation
Cost
Epoch
0
1
2
3
4
5
6
0
100
200
300
400
500

In [ ]:

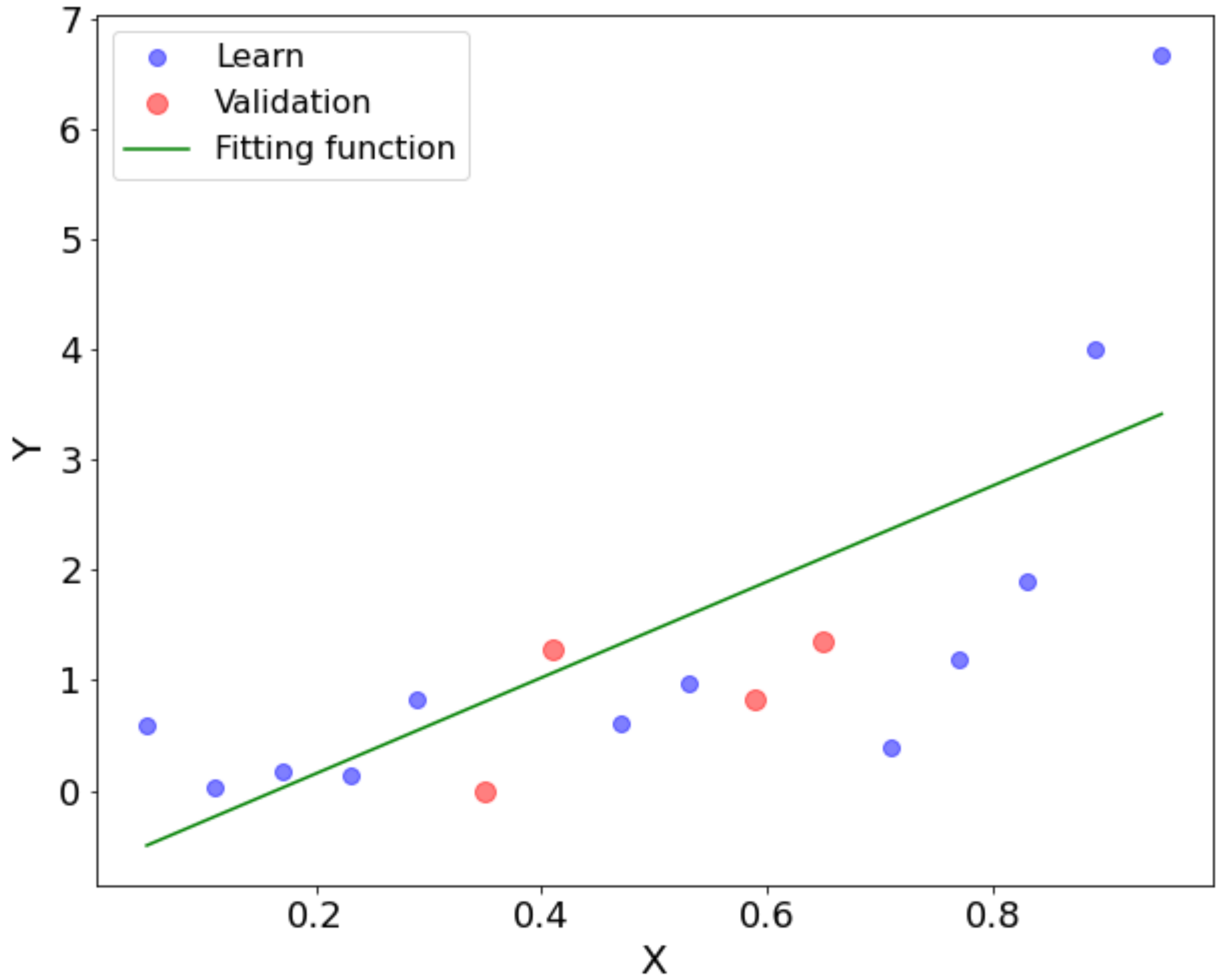
In [11]:

```
def fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree):

    fig = plt.figure(figsize=(10, 8))
    plt.ylabel('Y', size=20)
    plt.xlabel('X', size=20)
    plt.rc('xtick', labelsize=18)
    plt.rc('ytick', labelsize=18)


    # plot teh samples:
    p1=plt.plot(x_learn, y_learn, "o", ms=8, alpha=0.5, label='Training', colo
    p1=plt.plot(x_val, y_val, 'o', ms=10, alpha=0.5, label='test data', color=

    # Polynomial Regression
    poly = PolynomialFeatures(degree=poly_degree)

    # Construct polynomial features
    X = poly.fit_transform(x_learn[:,np.newaxis])
    clf = linear_model.LinearRegression()
    clf.fit(X,y_learn)

    Xplot=poly.fit_transform(xplot[:,np.newaxis])
    poly_plot=plt.plot(xplot, clf.predict(Xplot), label='Poly', color='green')

    plt.legend(['Learn', 'Validation', 'Fitting function'], loc='upper left',
```

Fit with a Linear Regression

In [12]:

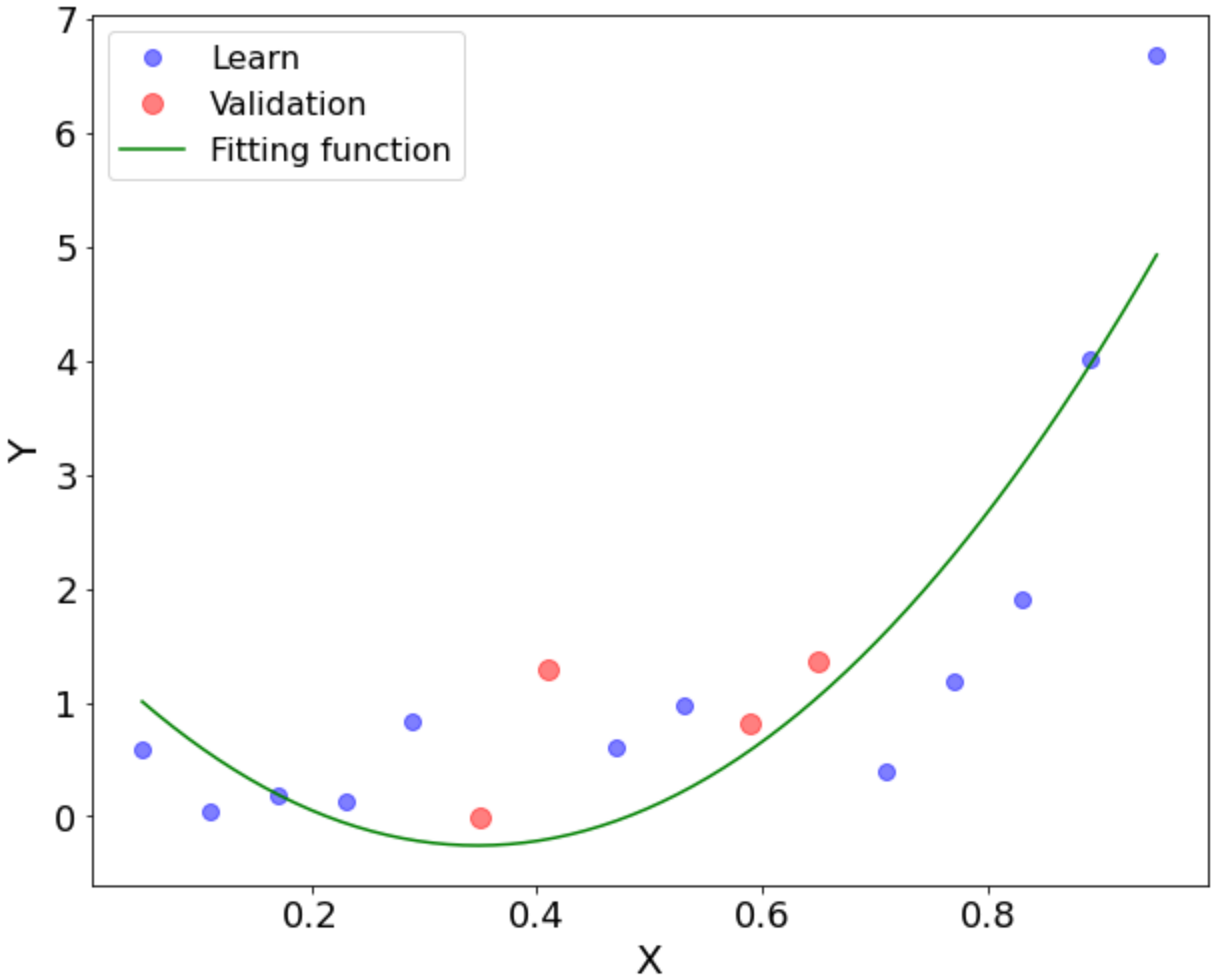
```
xplot=np.linspace(0.05,0.95,200)

fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree=1)
```



Polynomial Regression: seconth order

In [13]:

```
fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree=2)
```

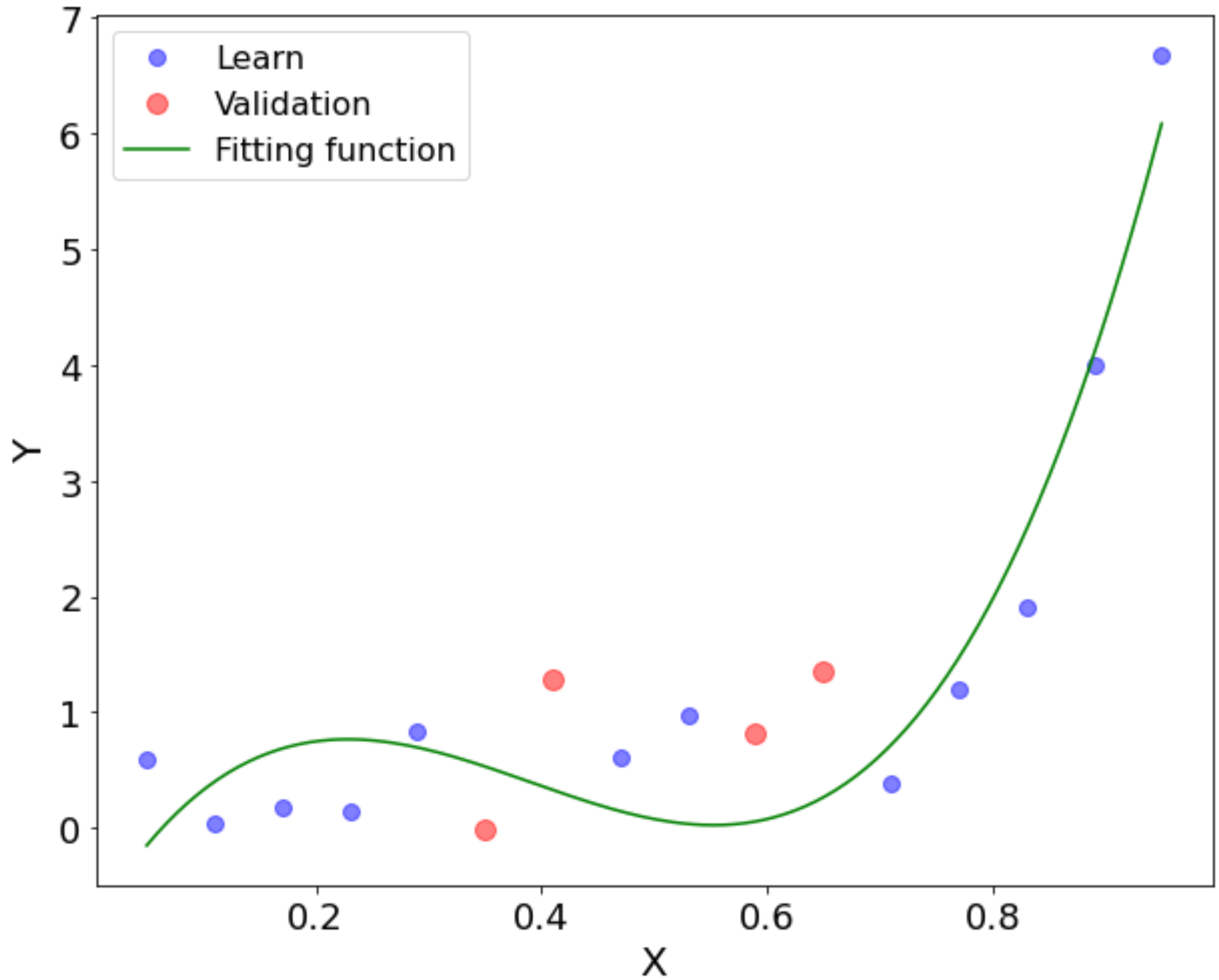


In [ ]:

Polynomial Regression: third order

In [14]:

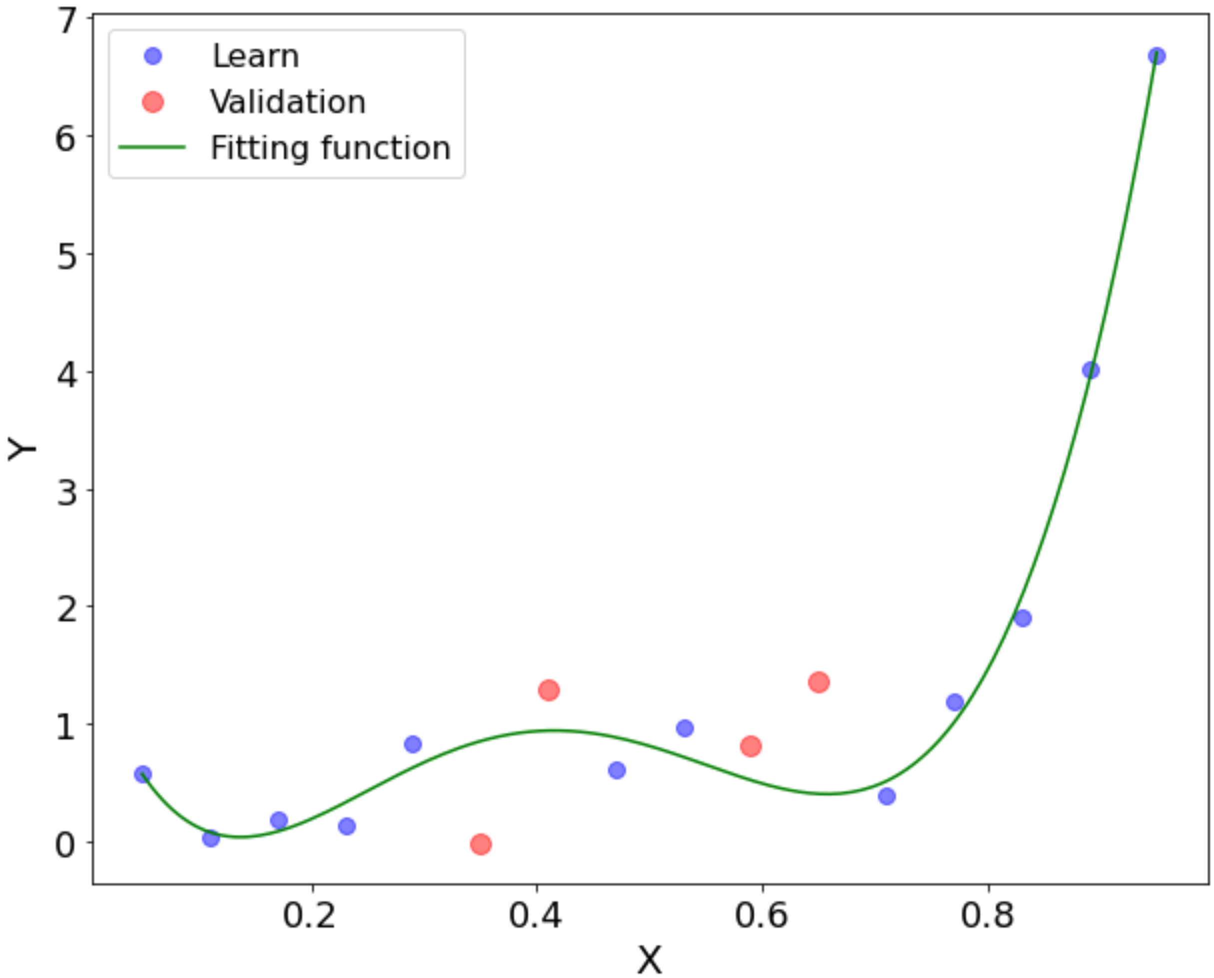
```
fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree=3)
```



Polynomial Regression: fifth order

In [15]:

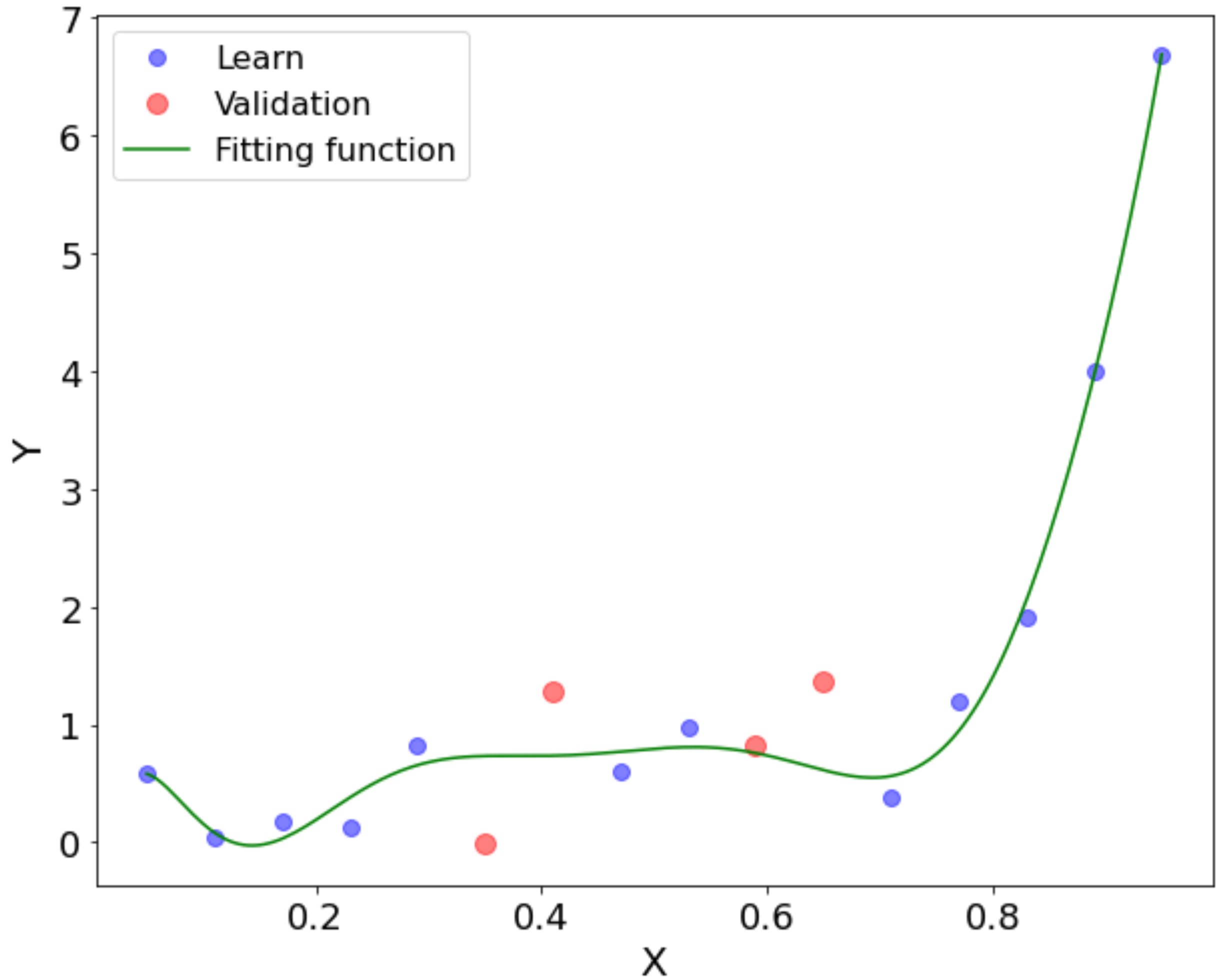
```
fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree=5)
```



Polynomial Regression: tenth order

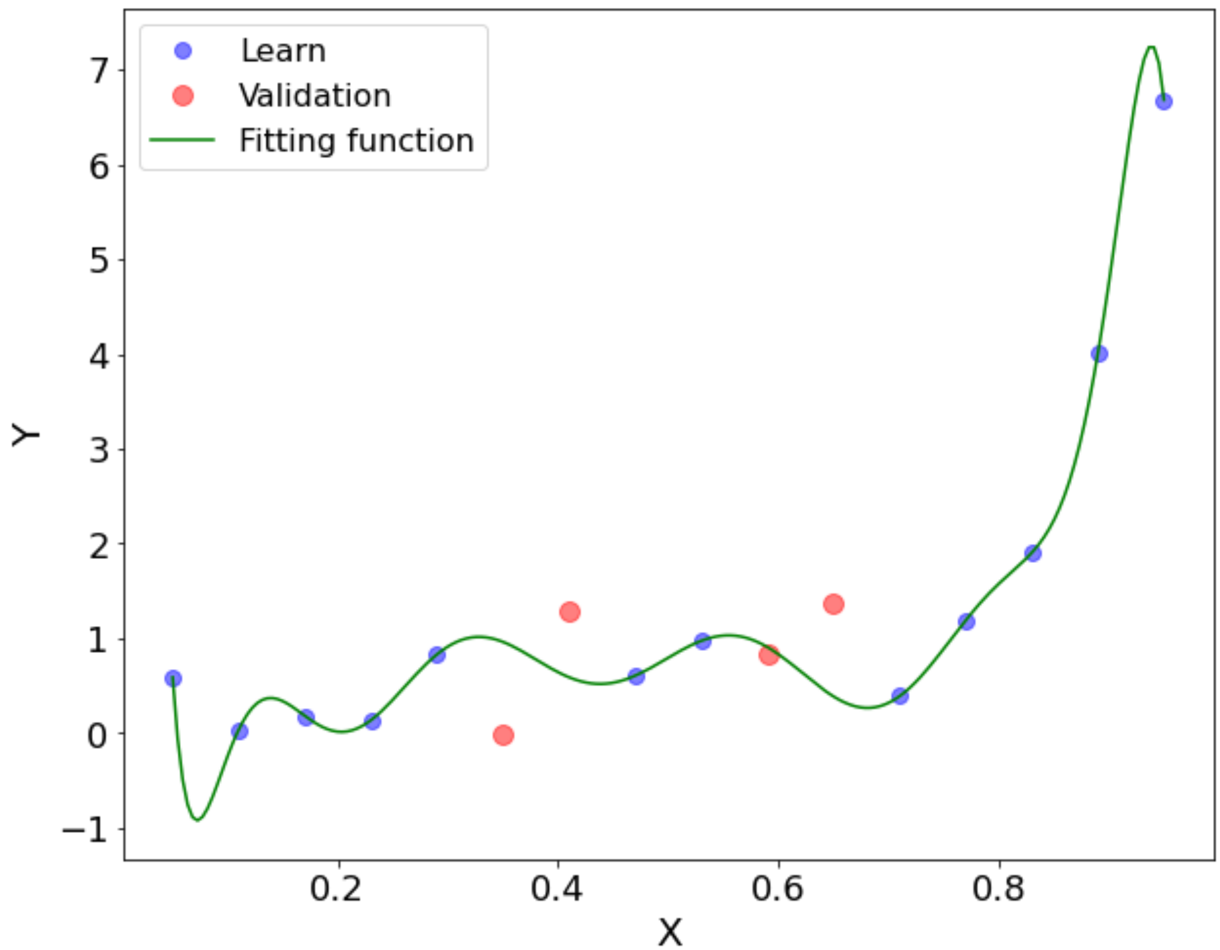
In [16]:

```
fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree=10)
```



Polynomial Regression: eleventh order

In [17]:
```
fit_poly(xplot, x_learn, y_learn, x_val, y_val, poly_degree=11)
```



In [ ]: