In [ ]:

```
#importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
```

In [ ]:

```
#combination function(required for finding coefficients of Legendre's polynomial)
def comb(n, k):
  return math.factorial(n)/(math.factorial(k)*math.factorial(n-k))
```

In [ ]:

```
#Generating coefficients of required Legendre's polynomial and normalizing it
def generate_coeff(order):
  coeff = np.zeros((order+1,1)) #coefficients
  for i in range(int(order/2)+1):
    coeff[order - 2*i] = (-1)**i*comb(order, i)*comb(2*order - 2*i, order)/2**order
    coeff = coeff/np.linalg.norm(coeff)
    #coeff[2] = coeff[2] + 2
  return coeff
```

In [ ]:

```
#function generating data points given sample size, coefficients of polynomial, order of po
def generate_data(size, coeff, order, err_sig2):
  X = np.random.uniform(-7, 7, size)                                              #uniformly
  X = np.array( sorted( X ))
  #X = X/max(X)
  X = np.array([X**i for i in range(0,order+1)]).T                                #generate n

  coeff = coeff.reshape(-1,1)

  y = X @ coeff                                                                   #generating

  if(err_sig2 != 0): y = y + np.random.normal(0,err_sig2, size).reshape(-1,1)   #adding noi

  return X[:,1], y
```

In [ ]:

```
#Function to compute lms loss
def error(a, b):
  return (1/2)*(np.linalg.norm(a-b))**2
```

In [ ]:

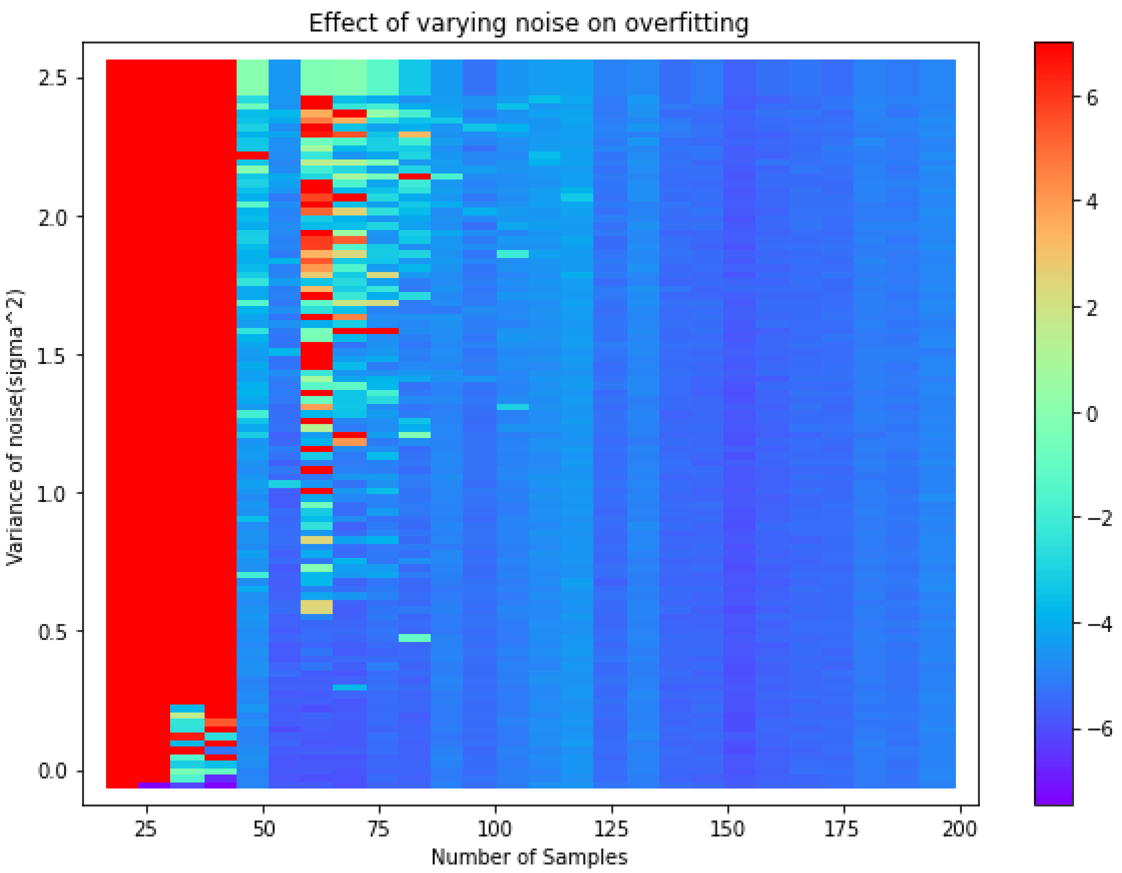
```
#defining function for fitting polynomial of given order given train and test data
def poly_fit(data, order, y, dtest, ytest):
  poly = PolynomialFeatures(degree = order)
  X_poly = poly.fit_transform(data)
  poly.fit(X_poly, y)
  lin2 = LinearRegression()                                                  #using skle
  lin2.fit(X_poly, y)
  y_pred = lin2.predict(poly.fit_transform(dtest))
  #Ein = error(y, lin2.predict(poly.fit_transform(data)))
  Eout = error(y_pred, ytest)                                                #calculatin
  return Eout
```

In [ ]:

```
sigma_2 = np.linspace(0, 2.5, 100)                                           #taking var
c = generate_coeff(20)                                                       #generating
Overfit_measure = []                                                         #list to st
for sigma in sigma_2:
  for n in range(20, 200, 7):                                                #sample siz
    #ein_2 = []
    eout_2 = []
    #ein_10 = []
    eout_10 = []
    for j in range(100):
      x, y = generate_data(n, c, 20, sigma)                                  #generating
      x,y = x.reshape(-1,1), y.reshape(-1,1)
      X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.50, random_stat
      Eout2 = poly_fit(X_train, 2, y_train, X_test, y_test)                  #fitting qu
      Eout10 = poly_fit(X_train, 10, y_train, X_test, y_test)                #fitting 10
      #ein_2.append(Ein2)
      eout_2.append(Eout2)
      #ein_10.append(Ein10)
      eout_10.append(Eout10)
    b = (np.mean(eout_10) - np.mean(eout_2))/n                               #taking dif
    if(b < 7 ): Overfit_measure.append(b)
    else: Overfit_measure.append(7)                                          #the overfi
    #print(n)
```

In [ ]:

```
#plotting colormaps
plt.figure( figsize=(10,7))
x, y = np.meshgrid(range(20,200, 7), sigma_2)
plt.scatter(x, y, c = Overfit_measure, cmap = 'rainbow', s = 300, marker = 's')
plt.xlabel('Number of Samples')
plt.ylabel('Variance of noise(sigma^2)')
plt.title('Effect of varying noise on overfitting')
cbar = plt.colorbar()
plt.show()
```

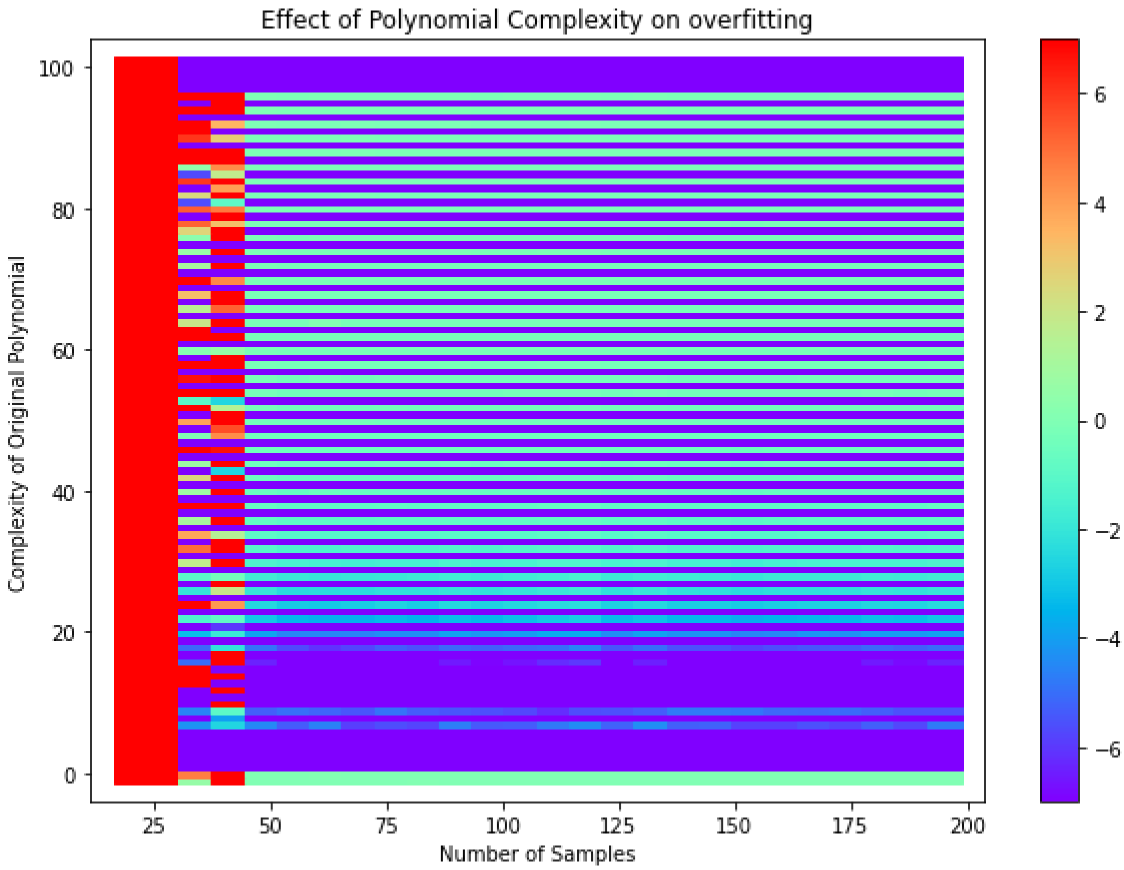


In [ ]:

```
Overfit_measure = []                                                            #list to st
for order in range(1, 100):                                                     #various de
  c = generate_coeff(order)                                                     #generating
  for n in range(20, 200, 7):                                                   #sample siz
    #ein_2 = []
    eout_2 = []
    #ein_10 = []
    eout_10 = []
    for j in range(100):
      x, y = generate_data(n, c, order, 0.1)                                    #generating
      x,y = x.reshape(-1,1), y.reshape(-1,1)
      X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.50, random_stat
      Eout2 = poly_fit(X_train, 2, y_train, X_test, y_test)                     #fitting qu
      Eout10 = poly_fit(X_train, 10, y_train, X_test, y_test)                   #fitting 10
      #ein_2.append(Ein2)
      eout_2.append(Eout2)
      #ein_10.append(Ein10)
      eout_10.append(Eout10)
    b = (np.mean(eout_10) - np.mean(eout_2))/n                                  #taking dif
    if(b > 7 ): Overfit_measure.append(7)
    elif(b < -7): Overfit_measure.append(-7)
    else: Overfit_measure.append(b)                                             #the overfi
#Overfit_measure = Overfit_measure/np.linalg.norm(Overfit_measure)
    #print(n)
```

In [ ]:

```
#plotting colormaps
plt.figure( figsize=(10,7))
x, y = np.meshgrid(range(20,200, 7), range(1,100))
plt.scatter(x, y, c = Overfit_measure, cmap = 'rainbow', s = 300, marker = 's')
plt.xlabel('Number of Samples')
plt.ylabel('Complexity of Original Polynomial')
plt.title('Effect of Polynomial Complexity on overfitting')
cbar = plt.colorbar()
plt.show()
```



Observations and Conclusions:

1. As we increase noise in data, the overfitting increases for higher complex model. But, as we go on increasing sample size, it'd give you better fit.
2. For complex target, higher order polynomial will give better fit for large number of points.

Note: As overfit measures were bounded for getting better graphs, there are borderline issues in the colormaps. But, without bounding the measure, graph was not showing color changes over space.