```
# import packages
import numpy as np
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from sklearn.model_selection import LeaveOneOut
import pandas as pd
```

Question 1

(a) The sample size n is extremely large, and the number of predictors p is small - **Flexible statistical learning method** would be better as it would have less bias when compared to the non-flexible model and with a large amount of data, the risk of overfitting is reduced, which is a common concern with flexible models.

(b) The number of predictors p is extremely large, and the number of observations n is small - With a small number of data points, the risk of overfitting increases for a flexible model(high variance). **Non-flexible model** should be picked based on bias-variance tradeoff.

(c) The relationship between the predictors and response is highly non-linear - A non-flexible model can never correctly estimate non-linear data, implying that this model has a high bias. **Flexible model** should be picked based on bias-variance tradeoff.

Question-2

```
boston_regression = pd.read_csv("data/boston_reg.csv")
boston_regression
```

	lstat	medv
0	22.74	8.4
1	24.39	8.3
2	29.68	8.1
3	22.11	10.5
4	23.60	11.3
5	25.68	9.7
6	24.16	14.0
7	26.82	13.4
8	26.40	17.2
9	23.09	20.0

```
boston_regression.columns
```

```
    Index(['lstat', 'medv'], dtype='object')
```

```
plt.xlabel("lstat")
plt.ylabel("medv")
plt.scatter(x=boston_regression.lstat, y=boston_regression.medv)
```

```
<matplotlib.collections.PathCollection at 0x7f98ee355a20>
```

part (a) and (b)

```
# fitting KNN regression model to estimate medv for a given lstat value.
X = boston_regression.lstat.to_numpy()
X.resize((10, 1))
y = boston_regression.medv.to_numpy()
# creating two KNN regression models with number of nearest neighbours equals 1 and 5
knn_reg_1 = KNeighborsRegressor(n_neighbors=1)
knn_reg_5 = KNeighborsRegressor(n_neighbors=5)
knn_reg_1.fit(X, y)
knn_reg_5.fit(X, y)
```

```
▾ KNeighborsRegressor
KNeighborsRegressor()
```

```
# prediction for sample points 25 and 27 using the k=1 and k=5 KNN.
knn_reg_1.predict([[25]]), knn_reg_5.predict([[25]]), knn_reg_1.predict([[27]]), knn_reg_5.predict([[27]])
```

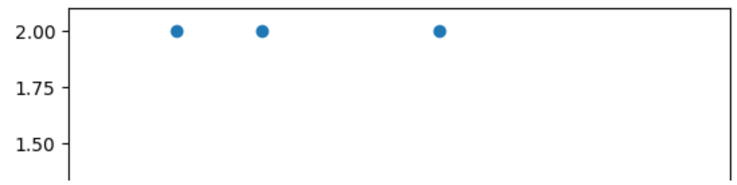
```
(array([8.3]), array([12.1]), array([13.4]), array([11.34]))
```

```
boston_classification = pd.read_csv("data/boston_class.csv")
boston_classification
```

	lstat	medv_cat
0	22.74	0
1	24.39	0
2	29.68	0
3	22.11	1
4	23.60	1
5	25.68	0
6	24.16	2
7	26.82	1
8	26.40	2
9	23.09	2

```
plt.xlabel("lstat")
plt.ylabel("medv")
plt.scatter(x=boston_classification.lstat, y=boston_classification.medv_cat)
```

```
<matplotlib.collections.PathCollection at 0x7f98ee58bee0>
```



part(c) and (d)

```
# fitting KNN regression model to estimate medv for a given lstat value.
X = boston_classification.lstat.to_numpy()
X.resize((10, 1))
y = boston_classification.medv_cat.to_numpy()
# creating two KNN regression models with number of nearest neighbours equals 1 and 5
knn_cl_1 = KNeighborsClassifier(n_neighbors=1)
knn_cl_5 = KNeighborsClassifier(n_neighbors=5)
knn_cl_1.fit(X, y)
knn_cl_5.fit(X, y)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

```
knn_cl_1.predict([[25]]), knn_cl_5.predict([[25]]), knn_cl_1.predict([[27]]), knn_cl_5.predict([[27]])
```

```
(array([0]), array([0]), array([1]), array([0]))
```

part (e)

Increases K makes a KNN mode less flexible - Since KNN is a type of instance-based or lazy learning algorithm that classifies data points based on the majority class among their K nearest neighbors, when K is small, the model considers only a few nearest neighbors, which can result in more complex and jagged decision boundaries. But as K gets larger, the model takes into account a greater number of neighbors, leading to a more generalized and smoother decision boundary.

part (f)

Bias: As K increases, bias generally increases - This happens because when K is large, the model considers a larger number of neighbors, resulting in a more generalized prediction. However, it might not capture local patterns in the data, leading to a higher bias.

Variance: As K increases, variance generally decreases - With a larger K, the model relies on more neighbors to make predictions, which results in a smoother and more stable prediction. This reduces the variance because the predictions become less sensitive to individual data points.

Training MSE: Training MSE tends to increase with larger values of K - When K is large, the model is more likely to underfit the training data. It becomes overly generalized, causing it to make predictions that are farther from the actual training data points, resulting in higher training MSE.

Test MSE: Test MSE initially decreases and then increases with K, following a U-shaped curve - Initially, as K increases, the model captures more global patterns in the data, reducing test MSE. However, when K becomes too large, the model loses the ability to capture important local patterns, leading to an increase in test MSE due to underfitting.

Irreducible Error: Irreducible error remains constant, regardless of the value of K - Irreducible error represents the inherent noise or randomness in the data that cannot be reduced by the model. Changing K does not affect this source of error.

Double-click (or enter) to edit

Question-3

part (a)

```
def f(x):
    return x ** 5 - 2 * x ** 4 + x ** 3
def get_sim_data(f, sample_size=100, std=0.01):
    x = np.random.uniform(0, 1, sample_size)
    y = f(x) + np.random.normal(0, std, sample_size)
    df = pd.DataFrame({"x": x,"y": y})
    return df


# generating sample data using f(x)
sample_data = get_sim_data(f)
sample_data
```

	x	y
0	0.162939	0.003732
1	0.126666	0.009395
2	0.148690	-0.007364
3	0.411449	0.023148
4	0.019690	-0.000145
...	...	...
95	0.092702	-0.006264
96	0.102709	-0.001527
97	0.536687	0.048213
98	0.073191	-0.001749
99	0.587828	0.030768

100 rows × 2 columns

part (b) and part (c)

```
# set of models
# step 1 - Fitting polynomial models of degree 0 to 15 with the randomly generated 100 data points.
# step 2 - Predicting the result for x_0
model_pred = []
x_0 = 0.18
for i in range(0, 16):
    model = PolynomialFeatures(degree=i)
    x_poly = model.fit_transform(sample_data[['x']])
    linear_model = LinearRegression()
    linear_model.fit(x_poly, sample_data['y'])

    coefficients = linear_model.coef_
    intercept = linear_model.intercept_
    out = intercept
    for idx, j in enumerate(coefficients):
        out += j * (x_0**idx)
    model_pred.append(out)

model_pred
```

part (d) - Repeating steps (a) - (c) 250 times

```
mul_runs = {}
x_0 = 0.18
y_0 = f(0.18) + np.random.normal(0, 0.01)
# running the above experiment 250 times
for run in range(250):
    # generating random data
    sample_data = get_sim_data(f)
    model_pred = []
    # fitting polynomial models of degree 0-15 and predicting x_0.
    for i in range(0, 16):
        model = PolynomialFeatures(degree=i)
        x_poly = model.fit_transform(sample_data[['x']])
```

```
            linear_model = LinearRegression()
            linear_model.fit(x_poly, sample_data['y'])
            model_pred.append(linear_model.predict(model.transform([[x_0]]))[0])
        mul_runs[run] = model_pred
```

```
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
    /Users/jay/opt/anaconda3/envs/ml_a1/lib/python3.10/site-packages/sklearn/base.py:465: UserWarning: X does not have valid fea
      warnings.warn(
```

```
output = pd.DataFrame(mul_runs).T
output
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	
0	0.015880	0.011083	0.009338	0.003758	0.002933	0.002485	0.002294	0.003754	0.003926	0.004062	0.003803	0.004645	0.005439	0.005
1	0.016781	0.011932	0.010478	0.005865	0.002389	0.004750	0.004813	0.005691	0.004640	0.003810	0.003641	0.003750	0.004616	0.004
2	0.017538	0.013340	0.011375	0.007859	0.006500	0.006554	0.006761	0.007456	0.008692	0.008801	0.010646	0.014858	0.015131	0.015

part (e) - squared bias of fitted polynomials

... ...

```
y_0 = f(x_0) + np.random.normal(0, 0.01)
bias = {}
for i in range(0, 16):
    bias[i] = np.mean([(j-y_0)**2 for j in output[i]])
bias
```

```
{0: 0.0005214106049575326,
 1: 0.0003381072389390026,
 2: 0.00023541644518725935,
 3: 0.0001471009597579544,
 4: 8.655107570289224e-05,
 5: 0.00010599478772437649,
 6: 0.000106631182268661,
 7: 0.00010513428766612702,
 8: 0.00010883240473990711,
 9: 0.00010950126482364787,
 10: 0.00011030168461411823,
 11: 0.00011389992389668405,
 12: 0.0001158740863979604,
 13: 0.00011679858887023101,
 14: 0.00011813543773490025,
 15: 0.00012178162098545234}
```

part (f) - variance of fitted polynomials

```
var = {}
for i in range(0, 16):
    var[i] = np.var(output[i])
var
```

```
{0: 2.4812129146570038e-06,
 1: 4.965094845553191e-06,
 2: 4.154800709784682e-06,
 3: 3.690192939793598e-06,
 4: 5.0902298843421085e-06,
 5: 6.134849338567241e-06,
 6: 6.133755091479029e-06,
 7: 7.6289943155316436e-06,
 8: 1.0394915884714816e-05,
 9: 1.1158565848483304e-05,
 10: 1.1856787195447371e-05,
 11: 1.398282692702509e-05,
 12: 1.588561435344047e-05,
 13: 1.6021106602169652e-05,
 14: 1.7749000507495425e-05,
 15: 2.107655291227071e-05}
```

Double-click (or enter) to edit

part (g) - Irreducible error

```
irreducible_error = (0.01) ** 2
```

part (h) - $mse = bias^2 + variance + var(\epsilon)$

```
mse = []
for i in range(0, 16):
    mse.append(irreducible_error + bias[i] + var[i])
mse
```

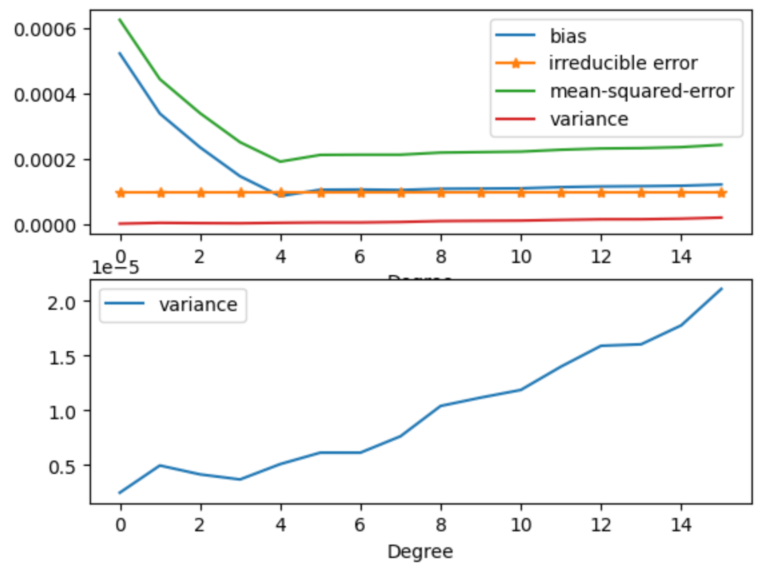
```
[0.0006238918178721896,
 0.0004430723337845558,
 0.00033957124589704404,
 0.000250791152697748,
 0.00019164130558723435,
 0.0002121296370629437,
 0.00021276493736014003,
 0.00021276328198165865,
 0.00021922732062462194,
 0.00022065983067213118,
 0.00022215847180956562,
 0.00022788275082370917,
 0.0002317597007514009,
 0.00023281969547240067,
 0.00023588443824239567,
 0.00024285817389772306]
```

```
np.argmin(mse)
```

```
4
```

```
plt.subplot(2, 1, 1)
plt.plot(list(range(0,16)), bias.values(), label = "bias")
plt.plot(list(range(0,16)), [irreducible_error]*16, marker="*", label="irreducible error")
plt.plot(list(range(0,16)), mse, label = "mean-squared-error")
plt.plot(list(range(0,16)), var.values(), label="variance")
plt.xlabel('Degree')
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(list(range(0,16)), var.values(), label="variance")
plt.xlabel('Degree')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f98d79204f0>
```



part (i)

As the degree of the polynomial model increases, there is a steady decline in squared bias and mse, indicating that the model has become more flexible. However as the model becomes more flexible, it is more sensitive to changes in data points and this is shown in the variance plot. There is a line parallel to x-axis in the graph which is the irreducible error. Irreducible error remains constant, regardless of the value of degree.

Question 4

```
def f(x):
    return x ** 5 - 2 * x ** 4 + x ** 3
```

part (a) - randomly generating 500 data points

```
np.random.seed(1)
x = np.random.uniform(0, 1, 500)
y = f(x) + np.random.normal(0, 0.01, 500)
df = pd.DataFrame({"x": x,"y": y})
df
```

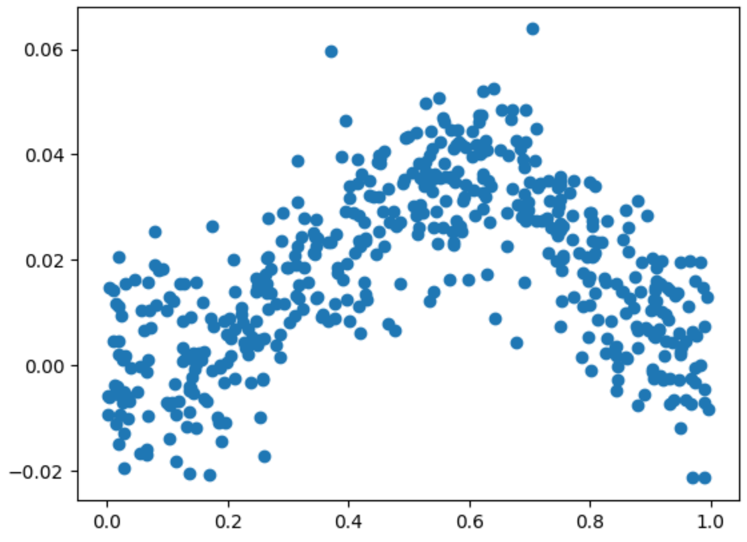
	x	y
0	0.417022	0.023488
1	0.720324	0.027480
2	0.000114	-0.009339
3	0.302333	0.008121
4	0.146756	-0.011964
...	...	...
495	0.931861	0.014270
496	0.936868	0.004227
497	0.844330	0.015394
498	0.920207	-0.002717
499	0.227900	0.003411

500 rows × 2 columns

part (b) - Scatter plot of the data

```
plt.scatter(df[["x"]], df[["y"]])
```

```
<matplotlib.collections.PathCollection at 0x7f98d82f4bb0>
```



part (c) - Computing LOOCV(Leave One Out Cross Validation) error of polynomial models of degree 1-7.

```
np.random.seed(123)
# generate data
x = np.random.uniform(0, 1, 500)
y = f(x) + np.random.normal(0, 0.01, 500)
```

```
df_4c = pd.DataFrame({"x": x,"y": y})
model_pred = []
# get the cross validation folds
loocv = LeaveOneOut()
X = df_4c[["x"]]
Y = df_4c[["y"]]
loocv_error = {}
# looping through every fold
for i, (train_idx, val_idx) in enumerate(loocv.split(X)):
    # for each fold create an empty list
    loocv_error[i] = []
    # fitting polynomial regression on train indices obtained in each fold, and then predicting the value in validation index.
    for j in range(1, 8):
        # convert features into polynomial features
        model = PolynomialFeatures(degree=j)
        x_poly = model.fit_transform(X.loc[train_idx])
        # fit a linear regression model on transformed polynomial features
        linear_model = LinearRegression()
        linear_model.fit(x_poly, Y.loc[train_idx])
        x_val = model.transform(X.loc[val_idx])
        loocv_error[i].append((Y.loc[val_idx].values[0][0] - linear_model.predict(x_val)[0][0])**2)
# computing the mean cross validation error for each model of degree 1 - 7.
overall_validation_error_c = pd.DataFrame(loocv_error).T
overall_validation_error_c = [overall_validation_error_c[i].mean() for i in overall_validation_error_c.columns]
overall_validation_error_c
```

```
[0.00023872667496951678,
 0.0001261052649907361,
 0.00011087924643876159,
 0.00010574321537933448,
 0.00010460992336824835,
 0.0001046344423490015,
 0.00010500174446146732]
```

part (d) - sampling data with different seed.

```
np.random.seed(12345)
# generate data
x = np.random.uniform(0, 1, 500)
y = f(x) + np.random.normal(0, 0.01, 500)
df_4d = pd.DataFrame({"x": x,"y": y})
model_pred = []
# get the cross validation folds
loocv = LeaveOneOut()
X = df_4d[["x"]]
Y = df_4d[["y"]]
loocv_error = {}
# looping through every fold
for i, (train_idx, val_idx) in enumerate(loocv.split(X)):
    # for each fold create an empty list
    loocv_error[i] = []
    # fitting polynomial regression on train indices obtained in each fold, and then predicting the value in validation index.
    for j in range(1, 8):
        # convert features into polynomial features
        model = PolynomialFeatures(degree=j)
        x_poly = model.fit_transform(X.loc[train_idx])
        # fit a linear regression model on transformed polynomial features
        linear_model = LinearRegression()
        linear_model.fit(x_poly, Y.loc[train_idx])
        x_val = model.transform(X.loc[val_idx])
        loocv_error[i].append((Y.loc[val_idx].values[0][0] - linear_model.predict(x_val)[0][0])**2)
# computing the mean cross validation error for each model of degree 1 - 7.
overall_validation_error = pd.DataFrame(loocv_error).T
overall_validation_error = [overall_validation_error[i].mean() for i in overall_validation_error.columns]
overall_validation_error
```

```
[0.00022114215496333624,
 0.00011371176055646526,
 0.00010057119012530983,
 9.740695636902114e-05,
 9.26890929957335e-05,
 9.300730908595378e-05,
 9.335757987940532e-05]
```

part (e) - min overall validation error

Polynomial with degree 5 has the smallest LOOCV error. Yes, since the model used to generate the synthetic data is a polynomial model of degree 5.

```
np.argmin(overall_validation_error)
```

```
4
```

part (f) - Fit $f_5(x)$ using least squares

const (Intercept): Coefficient Estimate: -0.0053 - Statistical Significance: The coefficient for the intercept is statistically significant with a p-value of 0.034. This indicates that the intercept is not likely to be zero, meaning there is a statistically significant constant term in the model.

x1: Coefficient Estimate: 0.1436 - Statistical Significance: The coefficient for x1 is statistically significant with a low p-value of 0.005. This suggests that changes in x1 have a significant effect on the dependent variable, and the estimated coefficient is unlikely to be zero by chance.

x2: Coefficient Estimate: -0.8814 - Statistical Significance: The coefficient for x2 is statistically significant with a p-value of 0.006. This indicates that x2 has a significant negative impact on the dependent variable, and this relationship is unlikely to be due to random chance.

x3: Coefficient Estimate: 3.1448 - Statistical Significance: The coefficient for x3 is highly statistically significant with a p-value of 0.000. This suggests a strong positive relationship between x3 and the dependent variable, and the estimated coefficient is very unlikely to be zero by chance.

x4: Coefficient Estimate: -4.2798 - Statistical Significance: The coefficient for x4 is highly statistically significant with a p-value of 0.000. This indicates a strong negative relationship between x4 and the dependent variable, and the estimated coefficient is very unlikely to be zero randomly.

x5: Coefficient Estimate: 1.8827 - Statistical Significance: The coefficient for x5 is highly statistically significant with a p-value of 0.000. This implies a strong positive association between x5 and the dependent variable, and the estimated coefficient is very unlikely to be zero by chance.

In summary, all the coefficients in the model have estimated values that are different from zero. The statistical significance of the coefficients is assessed using p-values, and all of them have p-values well below the common significance threshold of 0.05, indicating their statistical significance.

```
import statsmodels.api as sm
model = PolynomialFeatures(degree=5)
x_poly = model.fit_transform(df_4d[["x"]])
ols_model = sm.OLS(df_4d[["y"]], x_poly)
results = ols_model.fit()
```

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

```
                            OLS Regression Results                            
==============================================================================
Dep. Variable:                      y   R-squared:                       0.615
Model:                            OLS   Adj. R-squared:                  0.611
Method:                 Least Squares   F-statistic:                     157.7
Date:                Tue, 26 Sep 2023   Prob (F-statistic):          6.80e-100
Time:                        16:09:07   Log-Likelihood:                 1618.0
No. Observations:                 500   AIC:                            -3224.
Df Residuals:                     494   BIC:                            -3199.
Df Model:                           5                                         
Covariance Type:            nonrobust                                         
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0053      0.002     -2.130      0.034      -0.010      -0.000
x1             0.1436      0.051      2.813      0.005       0.043       0.244
x2            -0.8814      0.321     -2.749      0.006      -1.511      -0.252
x3             3.1448      0.819      3.838      0.000       1.535       4.754
x4            -4.2798      0.909     -4.708      0.000      -6.066      -2.494
x5             1.8827      0.364      5.172      0.000       1.167       2.598
==============================================================================
Omnibus:                        1.498   Durbin-Watson:                   2.076
Prob(Omnibus):                  0.473   Jarque-Bera (JB):                1.536
Skew:                           0.131   Prob(JB):                        0.464
Kurtosis:                       2.932   Cond. No.                     3.91e+03
==============================================================================

Notes:
```

```
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.91e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```