MDL ASSIGNMENT 2 REPORT

Submitted by: Vanshita Mahajan
Roll Number: 2021101102

TASK 2.1

Linear regression is a statistical method used to model the relationship between a dependent variable (also called the response variable) and one or more independent variables (predictors or features). Linear Regression aims to find the best-fitting line or hyperplane that can predict the value of the dependent variable based on the values of the independent variables.

The LinearRegression().fit() method is a function in the scikit-learn library used for training a linear regression model on a given dataset. The method fits a linear regression model to the data and estimates the coefficients of the linear equation that best fit the data. These coefficients are estimated using the least squares technique, which minimises the sum of the squared differences between the predicted values and the actual values. The fit() method takes as input a feature matrix X and a target array y, where X contains the input variables and y contains the corresponding target values.

TASK 2.2

Gradient descent is an iterative optimization algorithm that is used to find the values of the coefficients in a linear regression model. In the case of simple linear regression, where there is one independent variable and one dependent variable, gradient descent can be used to find the slope and intercept of the linear equation that best describes the relationship between the variables.

1. Start by making a first guess at the regression model's coefficients. This is usually done randomly
2. Compute the predicted values of the dependent variable using the current values of the coefficients and the independent variable.
3. Compute the sum of squared errors between the predicted values and the actual values of the dependent variable. (usually MSE is calculated)
4. Compute the gradient of the cost function with respect to each coefficient.
5. Update each coefficient by subtracting the product of the gradient and a learning rate from its current value. The learning rate is a small positive constant that determines the step size in each iteration. The formula for updating the coefficients is:
   new_coefficient = old_coefficient - learning_rate * gradient
6. Repeat steps 2 to 5 until the sum of squared errors is minimised or ***until some stopping criterion is me***t. This could be in the form of a maximum number of iterations or a threshold for the improvement in the cost function

Suppose we have a dataset of 10 points that represent the relationship x (independent variable) and y (dependent variable). We want to find the best-fit line that predicts the exam score as a function of the number of hours studied.

To use gradient descent to find the coefficients of the linear regression model, we need to define a cost function that measures the difference between the predicted values and the actual values of the dependent variable.
In this case we take Mean Squared Error approach

cost = 1/2m * sum((y_predicted - y_actual)^2) where m is the number of data points, y_predicted is the predicted value of the dependent variable, and y_actual is the actual value of the dependent variable.

y_predicted = $b_o + b_1$ * x
where b_o and b_1 are the coefficients of the model and x is the independent variable

To apply gradient descent, we start by choosing initial values for b_o and b_1. Let's say b_o = 0, b_1 = 1

1. Compute the predicted values of the dependent variable using the current values of theta_0 and theta_1.
   y_predicted = $b_o + b_1$ * x
2. Compute the cost function using the predicted values and the actual values of the dependent variable.

   cost = 1/2m * sum((y_predicted − y_actual)^2

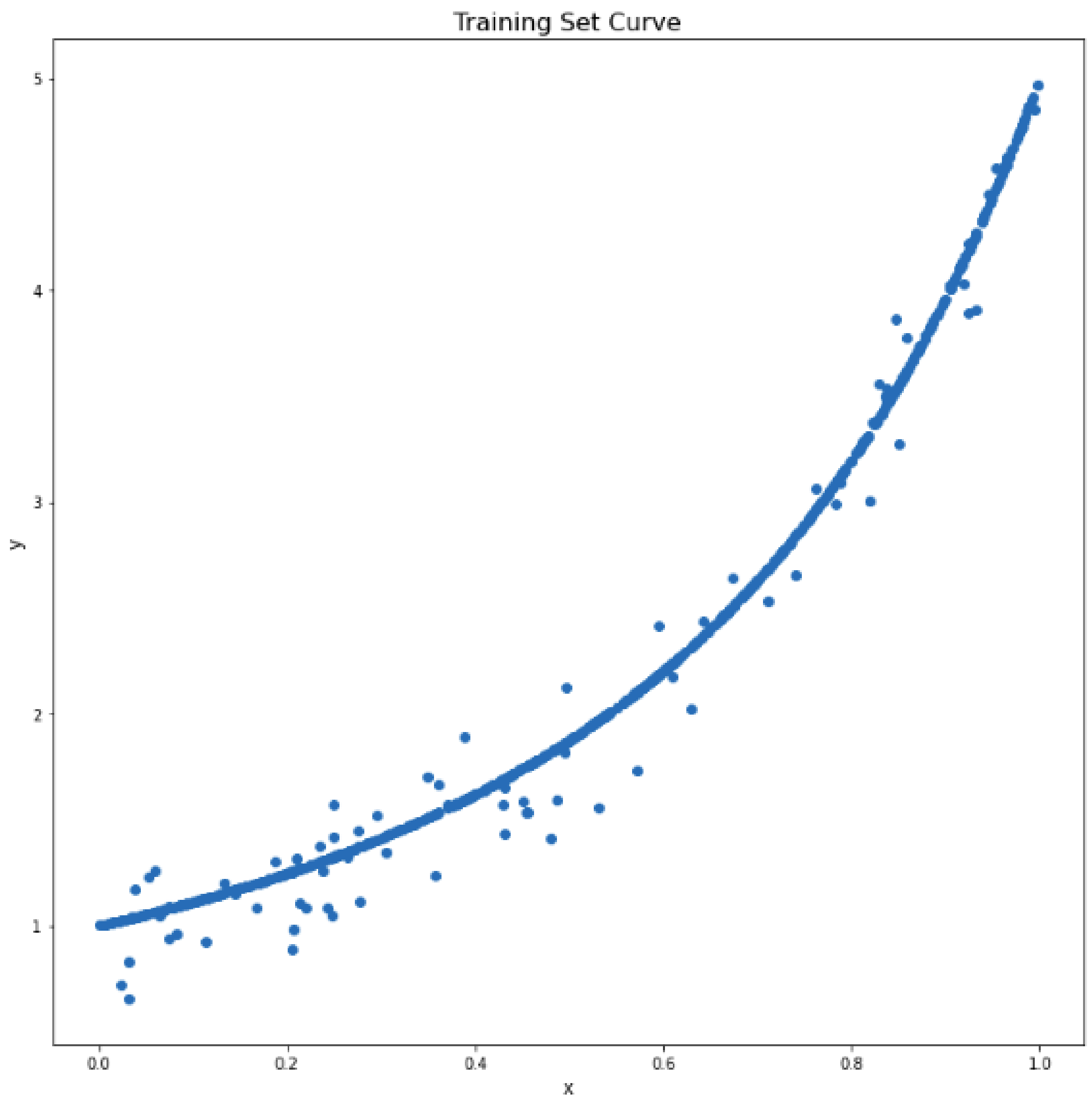
3. Compute the gradient of the cost function with respect to each coefficient.
   d_cost_d_ b_o = 1/m * sum(y_predicted − y_actual
   d_cost_ d_b_1 = 1/m * sum((y_predicted − y_actual) * x)

4. Update each coefficient by subtracting the product of the gradient and a learning rate from its current value.
   b_o = b_o − alpha * d_cost_d_b_o
   b_1 = b_1 − alpha * d_cost_d_b_1
   Here, alpha is the learning rate that determines the step size in each iteration. Let's assume that we choose alpha = 0.01
5. Repeat procedure until the sum of squared errors is minimised or until some stopping criterion is met.Using this algorithm, we can compute the new values of theta_0 and theta_1 in each iteration until convergence.

TASK 2.3

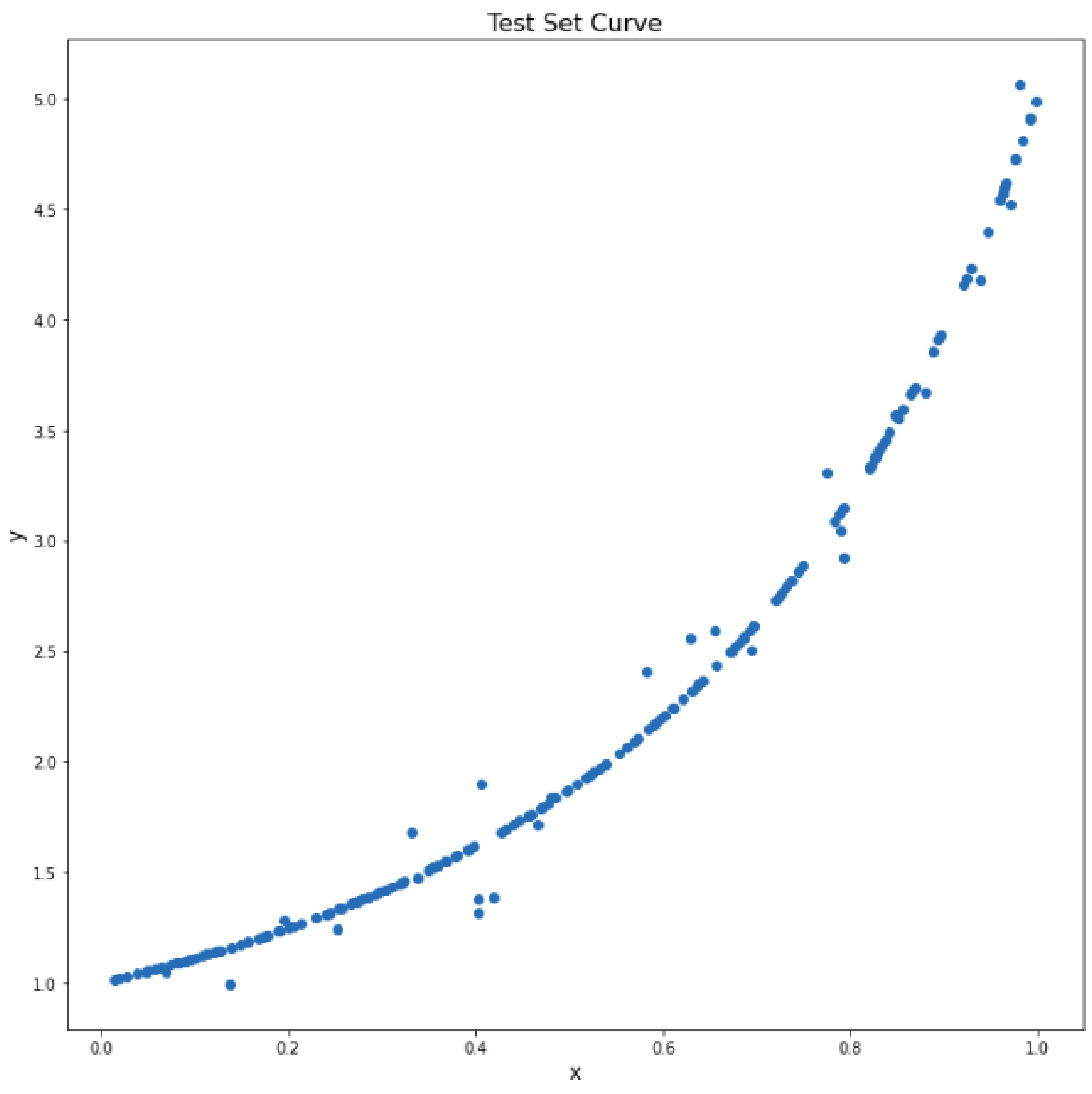
```
In [38]: # Plotting the Training set

         plt.figure(figsize=(12,12))
         plt.scatter(train['X'], train['Y'])
         plt.xlabel('x', fontsize=12)
         plt.ylabel('y', fontsize=12)
         plt.title('Training Set Curve', fontsize=16)
         plt.show()
```



```
In [39]: # Plotting the test set

         plt.figure(figsize=(12,12))
         plt.scatter(test['X'], test['Y'])
         plt.xlabel('x', fontsize=14)
         plt.ylabel('y', fontsize=14)
         plt.title('Test Set Curve', fontsize=16)
         plt.show()
```

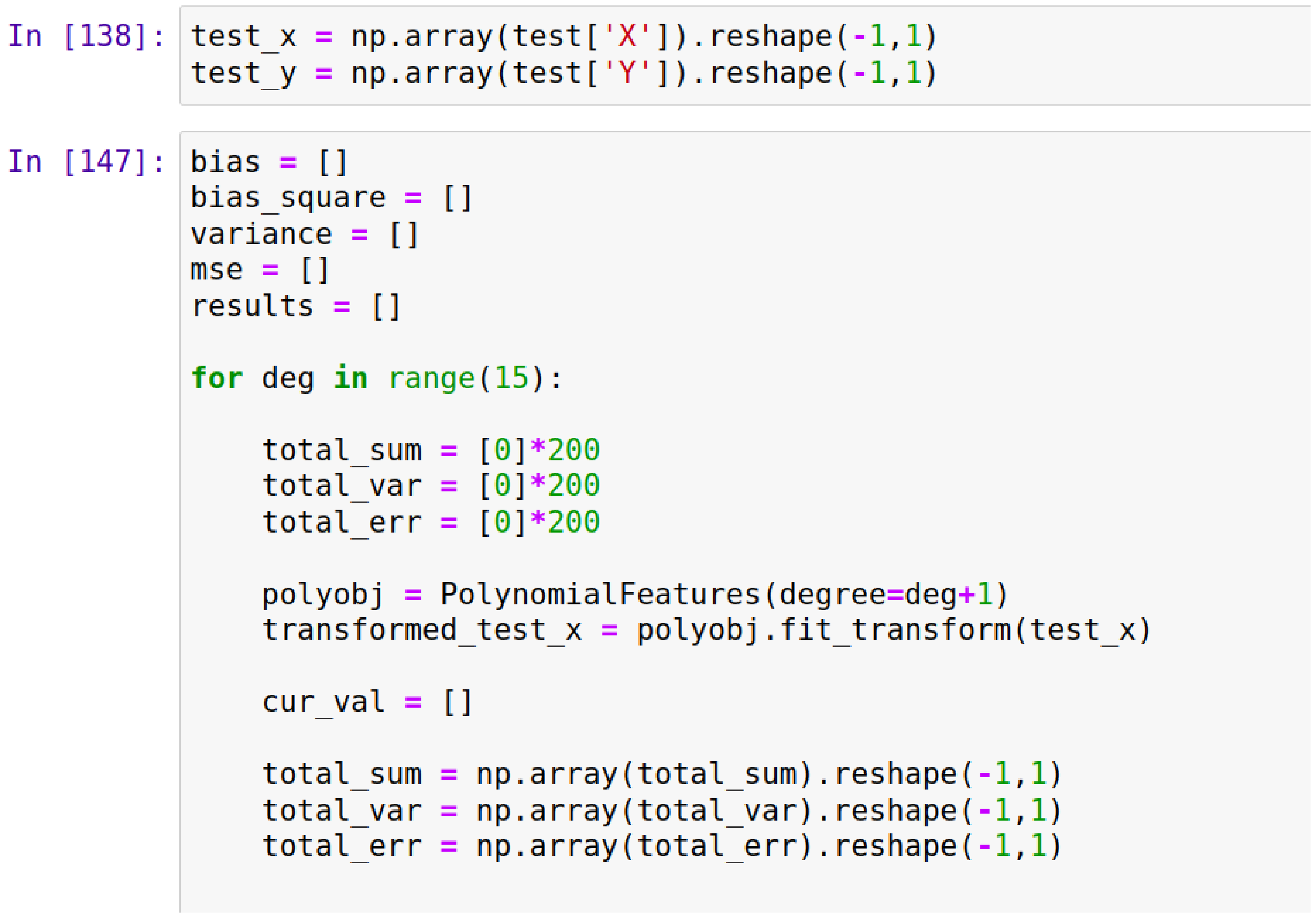


The train dataset has 800 data points while the test dataset has 200 data points. Hence the plot for the test set is spare compared to the plot for the train set.
Using the numpy.array_split function split the train array is split into 20 partitions. The resulting train₩_split variable is a list containing 20 arrays, where each array represents a partition of the original train array.

We initialise an empty list called trained_models to store the trained models for each degree of polynomial fit. Then, it loops over each degree from 1 to 15, inclusive.

Within the loop, we initialise another empty list to store the trained models for the current degree. We then create an instance of the PolynomialFeatures class in python corresponding to the current degree of polynomial fit, which will be used to transform the input features into a polynomial feature matrix.

Next we vectorize the X and Y arrays of the test dataset. 5 lists to store the corresponding mean/expected values for each degree from 1 to 15 are created.



```
In [138]: test_x = np.array(test['X']).reshape(-1,1)
          test_y = np.array(test['Y']).reshape(-1,1)

In [147]: bias = []
          bias_square = []
          variance = []
          mse = []
          results = []

          for deg in range(15):

              total_sum = [0]*200
              total_var = [0]*200
              total_err = [0]*200

              polyobj = PolynomialFeatures(degree=deg+1)
              transformed_test_x = polyobj.fit_transform(test_x)

              cur_val = []

              total_sum = np.array(total_sum).reshape(-1,1)
              total_var = np.array(total_var).reshape(-1,1)
              total_err = np.array(total_err).reshape(-1,1)
```

cur_val.append(trained_models[deg][j].predict(transformed_test_x))
line predicts the output y using the trained model for the current degree and the transformed test data

```
for j in range(20):

    cur_val.append(trained_models[deg][j].predict(transformed_test_x))

    total_sum = total_sum + cur_val[j]    # calculating sum to obtain mean of all predicted values in the end
    total_var = total_var + cur_val[j]**2 # calculating summation of squares
    total_err = total_err + (cur_val[j] - test_y)**2 #calculating error using least squares approach -
                                                     #summation of squares of difference between predicted valu
                                                     #and actual value

# calculating expected values

pred = total_sum / partition_size         # this gives mean predicted value for each degree
var = total_var / partition_size - pred**2
mse_val = total_err / partition_size
bias_square_val = np.mean((pred - test_y)**2)
bias_val = np.mean(np.abs(pred - test_y)) # this gives mean bias for each degree
                                          # (since bias is a measure of error in data
                                          # we simply calculate absolute difference 7
                                          # between predicted value and given value for y)


bias.append(bias_val)
bias_square.append(bias_square_val)
variance.append(np.mean(var))
mse.append(np.mean(mse_val))
resultarray.append(pred)
```

TASK 2.4

Irreducible Error is the error that cannot be reduced, no matter how complex or good the training model is. It is the amount of noise inherently present in the data. It arises from the natural variability in the data, which is not explained by the features in the model.

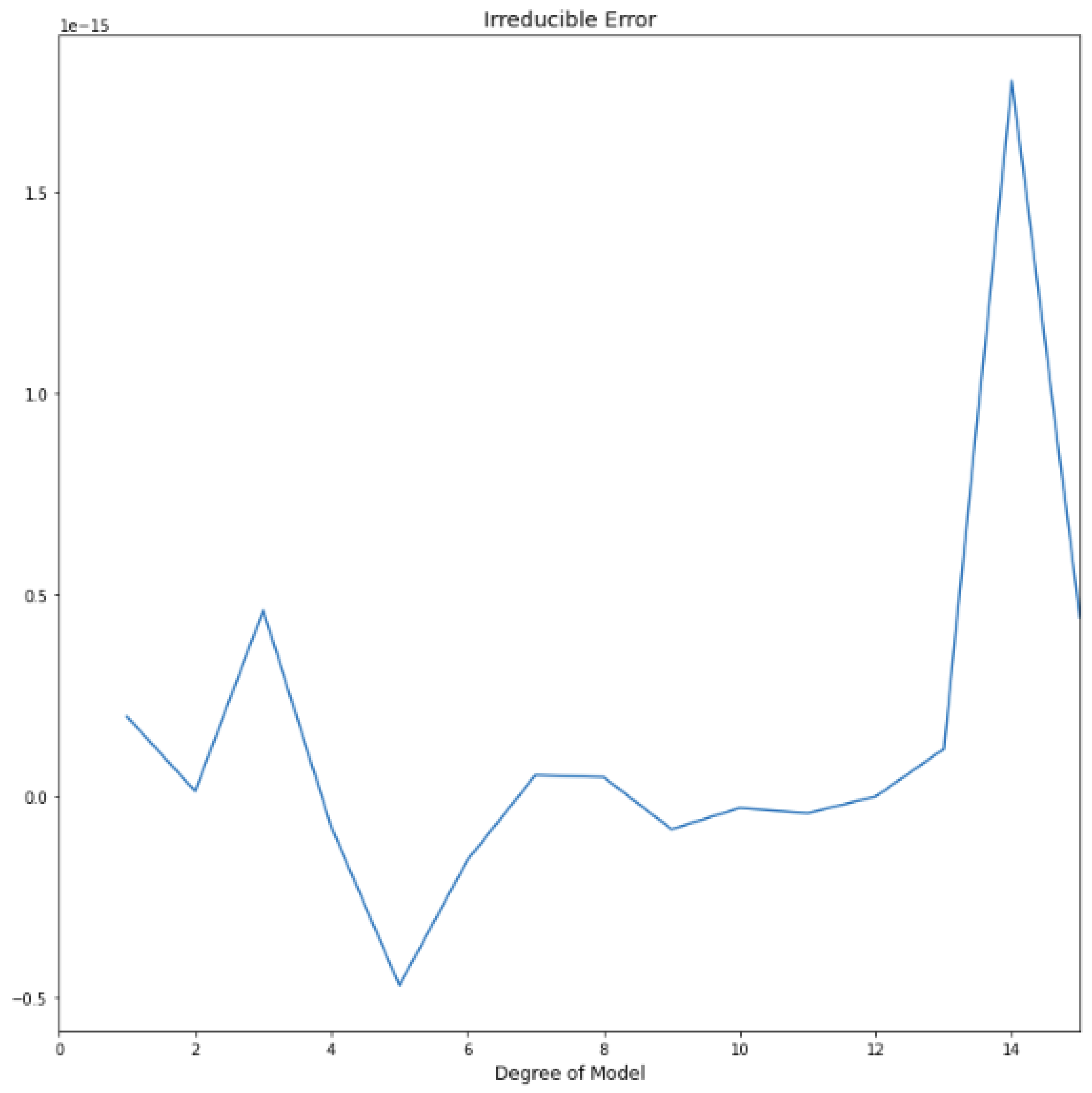
```
In [27]: irreducible_error = np.array(mse) - np.array(bias_square) - np.array(variance)

         table_irreducible_error = {
             'Degree': [deg for deg in range(1, 16)],
             'Irreducible Error' : irreducible_error
         }

         print(tabulate(table_irreducible_error, headers='keys', tablefmt='psql'))
```

```
+----------+--------------------+
|   Degree |   Irreducible Error |
|----------+--------------------|
|        1 |        1.97758e-16 |
|        2 |        1.30104e-17 |
|        3 |        4.6187e-16  |
|        4 |       -7.55147e-17 |
|        5 |       -4.68972e-16 |
|        6 |       -1.58185e-16 |
|        7 |        5.31259e-17 |
|        8 |        4.79217e-17 |
|        9 |       -8.1532e-17  |
|       10 |       -2.86229e-17 |
|       11 |       -4.16334e-17 |
|       12 |        0           |
|       13 |        1.17961e-16 |
|       14 |        1.77636e-15 |
|       15 |        4.44089e-16 |
+----------+--------------------+
```

From the table above depicting irreducible error values in data for each degree, it can be deduced that irreducible error has extremely small values for this data set, ranging from the order of 10^{-17} to 10^{-15} in most cases.We note that the values lie in the same range for all 15 degree models. This is indicative of the fact that noise is an inherent characteristic of the dataset and does not vary with the different models applied.



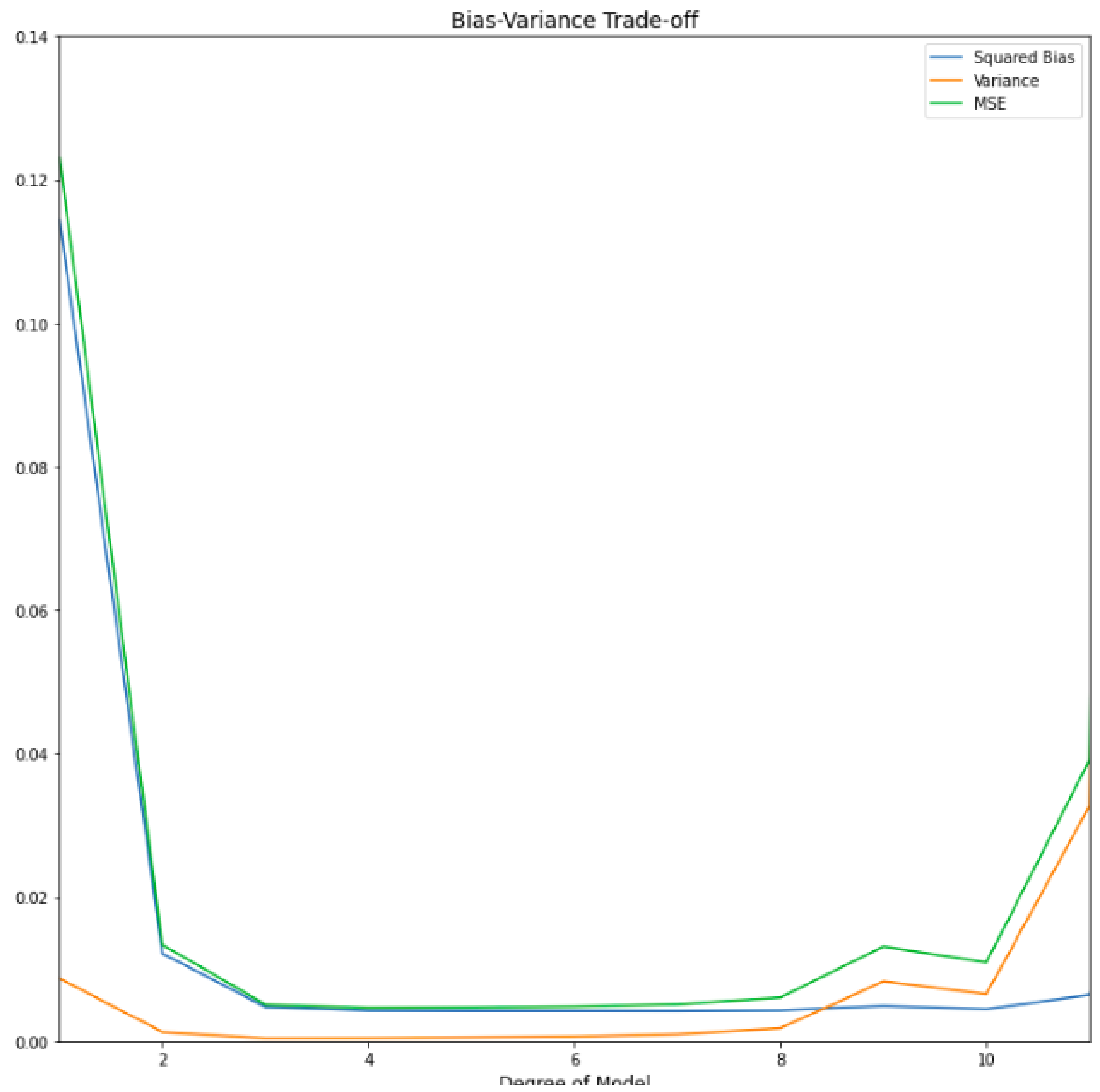
Furthermore, we see that some of the values obtained are negative. In general, however, the value of irreducible error cannot be negative because it represents the minimum error that is always present in the data, even if the model is perfectly fit. Hence it can be deduced that a portion of these errors is generated due to rounding off operations of python apart from actual noise in the dataset.

TASK 2.5

Bias and Variance both are a measure of error in the predicted values i.e., difference in values of predicted and actual data.

```
In [108]: # depicting the bias-variance tradeoff
          plt.figure(figsize=(12,12))
          plt.plot(range(1,16), bias_square)
          plt.plot(range(1,16), variance)
          plt.plot(range(1,16), mse)
          plt.xlabel('Degree of Model', fontsize=12)
          plt.title('Bias-Variance Trade-off', fontsize=14)
          plt.legend(['Squared Bias', 'Variance', 'MSE'])
          plt.xlim(1,11)
          plt.ylim(0,0.14)

Out[108]: (0.0, 0.14)
```



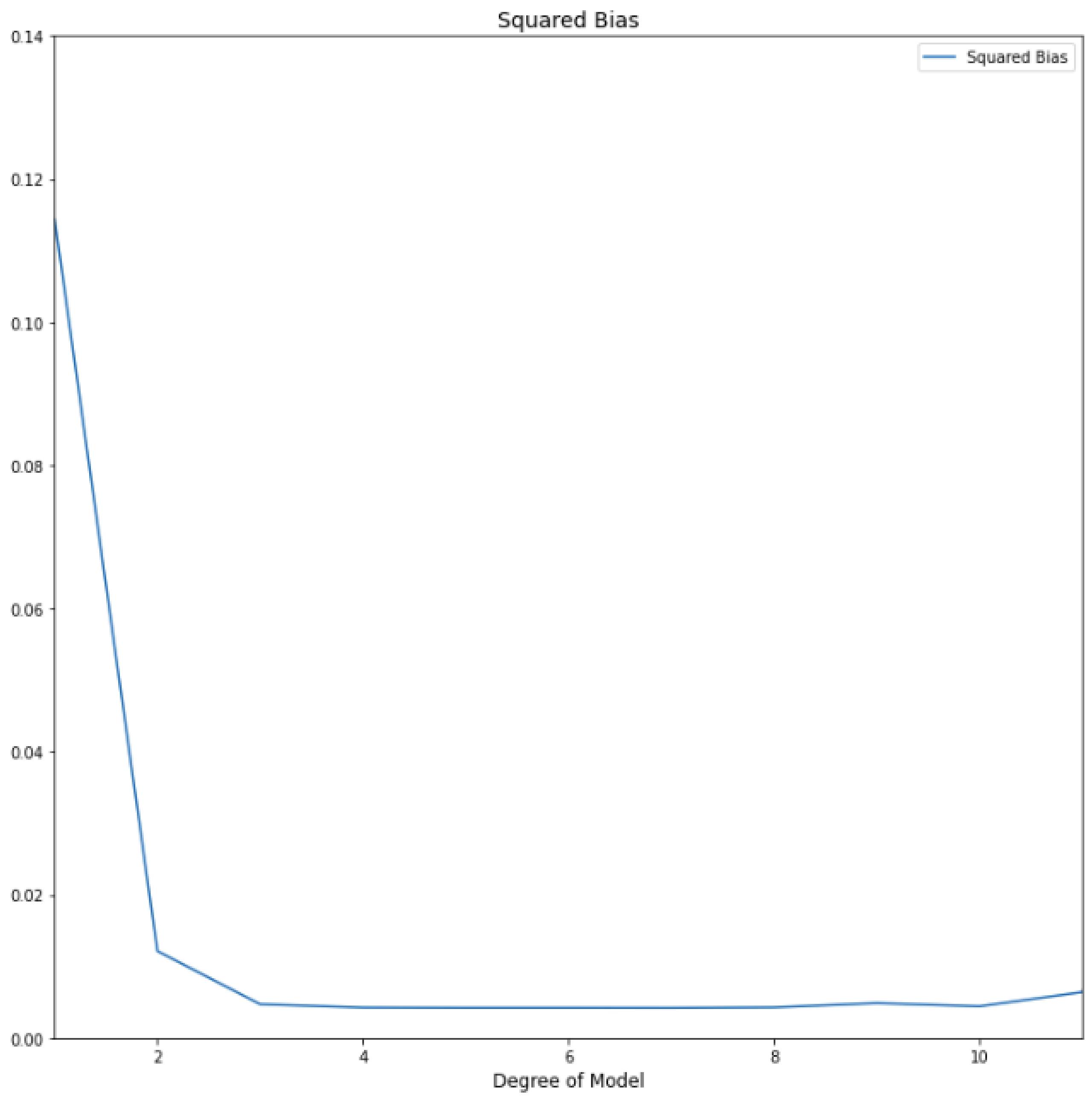
Shown above is the plot for three errors that we get for the given dataset.

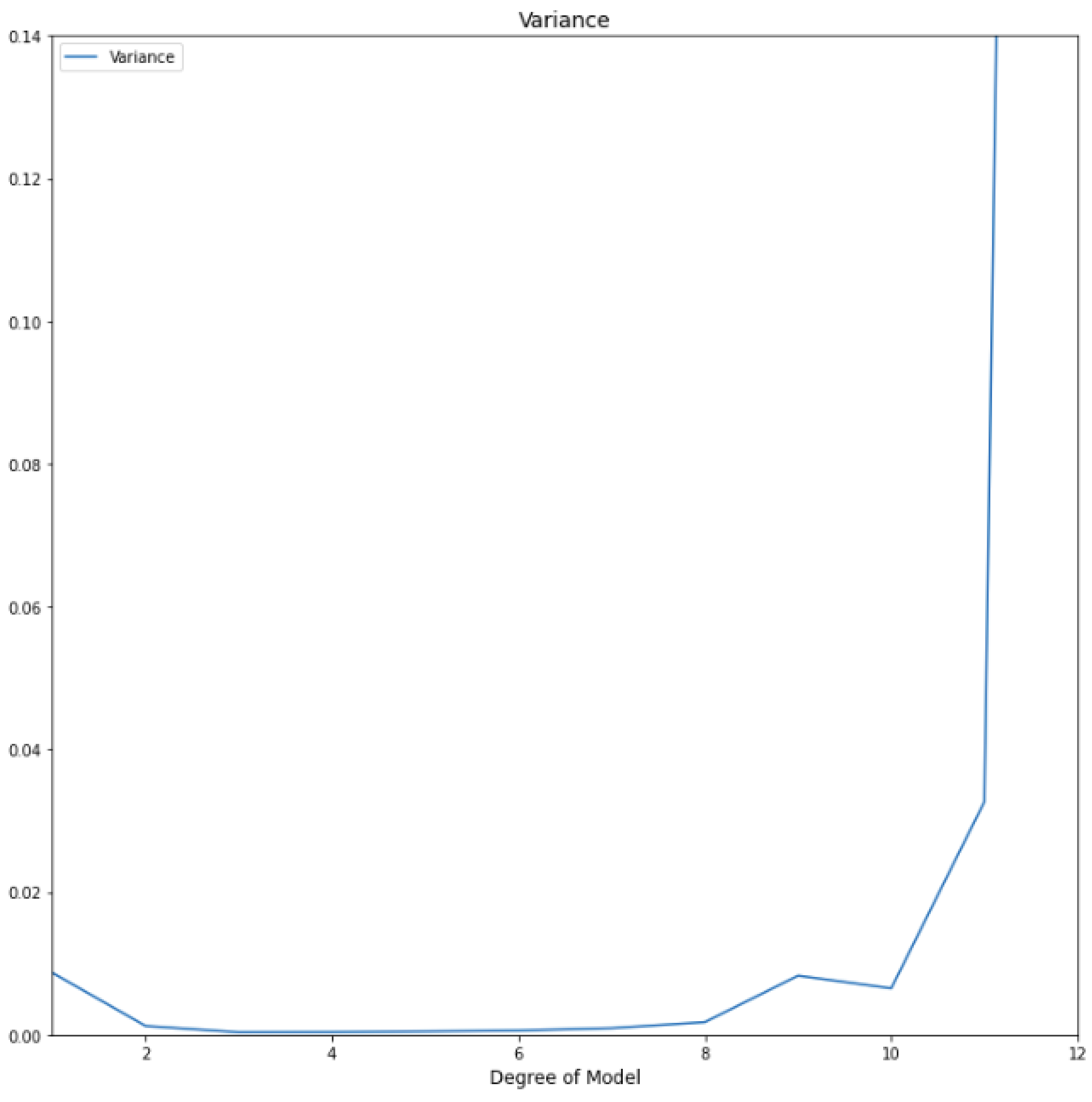
The bias variance trade−off refers to the trade−off between the ability of a model to fit the training data well and its ability to generalise to new data. As the complexity of the model increases, the bias decreases, and the variance increases. This is because a more complex model can fit the training data better, resulting in lower bias, but it may also be more sensitive to small changes in the input data, resulting in higher variance.

OBSERVATIONS:

1. We notice that the squared bias(blue curve) decreases as the degree increases (and hence, the complexity) while the variance (orange curve) increases. That is, bias has very high values at lower degrees and becomes low around degree = 3. It then stays approximately the same.

This is indicative of the fact that at lower degrees (say < 3) the model underfits to the train data, resulting in a high bias. And low bias. At higher degrees (say > 10) model overfits to train data, resulting in high variance and low bias





2. Variance almost increases with increase in degree, with a few variations.

3. It can also be seen that at degree=4, both squared bias and variance are minimised. Hence 4 is the optimal degree for performing regression on this dataset where error is minimum.

4. BONUS

```
In [150]: # Tabulating the Bias-Variance-MSE
          from tabulate import tabulate
          table_bias_variance_mse = {
              'Degree': [i for i in range(1, 16)],
              'Bias': bias,
              'Variance': variance,
              'MSE' : mse
          }

          print(tabulate(table_bias_variance_mse, headers='keys', tablefmt='psql'))
```

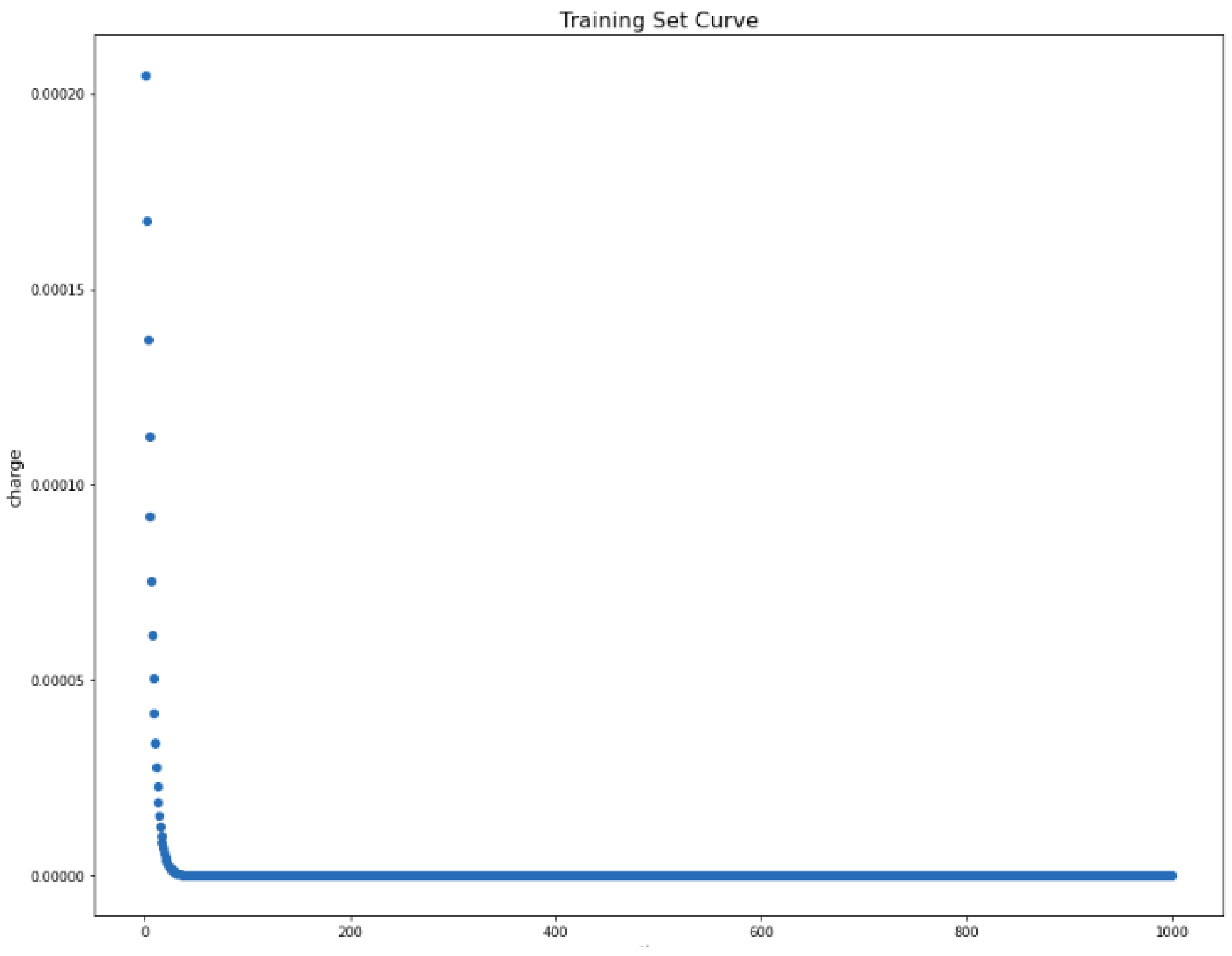
```
+----------+-----------+-------------+-----------+
|   Degree |      Bias |    Variance |       MSE |
|----------+-----------+-------------+-----------|
|        1 | 0.269398  | 0.00868095  | 0.123073  |
|        2 | 0.0862565 | 0.00122436  | 0.0133656 |
|        3 | 0.0332718 | 0.000337339 | 0.00504545 |
|        4 | 0.0242826 | 0.000366999 | 0.0046061  |
|        5 | 0.0238793 | 0.000461938 | 0.00465952 |
|        6 | 0.0239554 | 0.00058152  | 0.00477993 |
|        7 | 0.02483   | 0.000916796 | 0.00510317 |
|        8 | 0.0248874 | 0.00176092  | 0.00602264 |
|        9 | 0.0304184 | 0.00827683  | 0.0131353  |
|       10 | 0.0286639 | 0.00650085  | 0.0109153  |
|       11 | 0.0365903 | 0.0326928   | 0.0390896  |
|       12 | 0.0709172 | 0.884494    | 0.941428   |
|       13 | 0.0422491 | 0.054317    | 0.0618877  |
|       14 | 0.156428  | 8.04334     | 8.48912    |
|       15 | 0.08913   | 2.66963     | 2.74963    |
+----------+-----------+-------------+-----------+
In [2]: #loading train data set

        with open('capacitor_dataset.pickle','rb') as f:
            file_train = pickle.load(f)
            train = pd.DataFrame(file_train,columns = ['Time','Charge'])
        #     print(train)
            print(len(train))

        1000
```

```
In [3]: # Plotting the Training set

        plt.figure(figsize=(15,12))
        plt.scatter(train['Time'], train['Charge'])
        plt.xlabel('time', fontsize=13)
        plt.ylabel('charge', fontsize=13)
        plt.title('Training Set Curve', fontsize=16)
        plt.show()
```



Here we have a dataset with 1000 time–charge value pairs. The plot of the data looks like the figure above. We load the values of time in the ***train_time*** array and values of charge in the ***train_charge*** array.

Charge as a function of time is represented as
 $Q = CV_oe^{-t/RC.}$
Note that Q varies exponentially with t.
To make it a linear dependence, we take natural logarithm on both sides.
$\ln(Q) = \ln(CV_o\, e^{-t/RC})$
$\ln(Q) = \ln(CV_o) + \ln e^{-t/RC}$
$\ln(Q) = \ln(CV_o) - t/RC$

This gives us a linear dependence between ln(Q) and t. Linear Regression is now applied according to this equation. We get slope and intercept value from this equation and we calculate the values of capacitance and resistance as :

```
In [12]: train_time = np.array(train_time).reshape(-1,1)
         model = LinearRegression().fit(train_time, ln_charge)

         # Extract slope and intercept
         slope = model.coef_[0]
         intercept = model.intercept_
```

since slope = -1/RC,
RC = -1/slope
and
ln(CVo) = y-intercept
ln(c * 5) = y-intercept
c * 5 = exp(y-intercept)
c = exp(y-intercept)/5

```
In [24]: # Print results
         print("Slope m = {:.4f}".format(slope))
         print("Intercept b = {:.4f}".format(intercept))
```

```
Slope m = -0.2000
Intercept b = -8.2940
```

```
In [18]: C = np.exp(intercept) / 5.000
         R = -1.0 / (slope * C)

         # Print results
         print(f"Capacitance C = {C:.8f} Farads")
         print(f"Resistance R = {R:.8f} Ohms")
```

```
Capacitance C = 0.00005000 Farads
Resistance R = 100000.00000001 Ohms
```