EXPLANATION OF OUR APPROACH FOR EACH TASK

TASK 1: Data Preprocessing

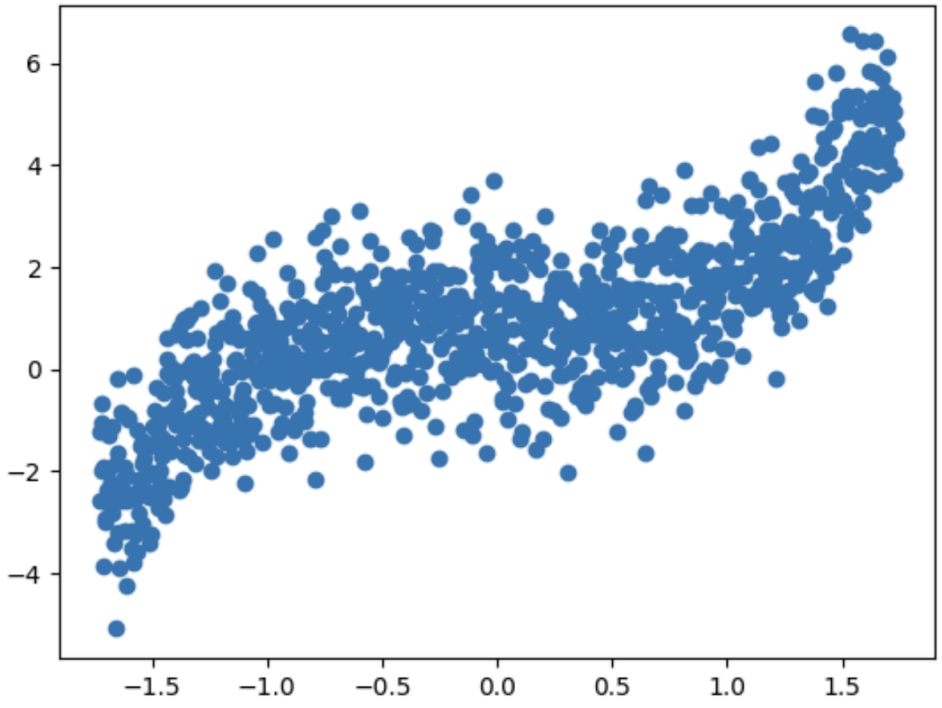
We started off with loading the dataset into a panda dataframe and scaled the feature vector by using standard scaling formula.

$X' = (X - \mu) / \sigma$

We shuffled the dataset using a fixed random state and split it in 80% training data and 20% testing data.

```
plt.scatter(X,Y)
X.shape[0]
```

```
1000
```



This is the visualization of the dataset given to us.

Task 2: Polynomial Regression and Regularization

1) Simple Polynomial Regression

We first defined the following functions to facilitate training and testing of ‘unregularised’ models:

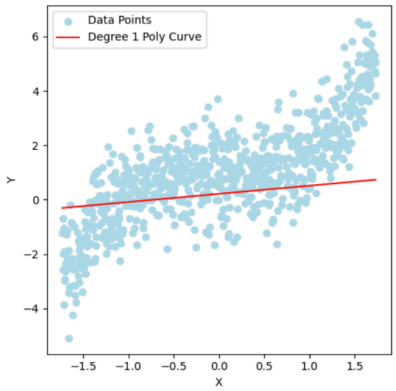
- A function to convert the univariate feature vector to a multivariate matrix with each column corresponding to a power of the feature vector.
- A function that calculates the gradient according to the formula: `gradient = (np.dot(X.T,(prediction - Y)))/m`
- A function to perform gradient descent and calculate the weights after 500 iterations.

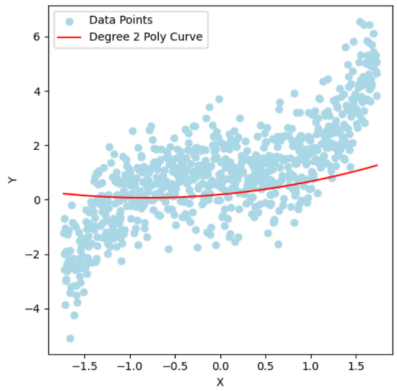
Then, we calculate the best weights for each degree polynomial model and plot the models accordingly.

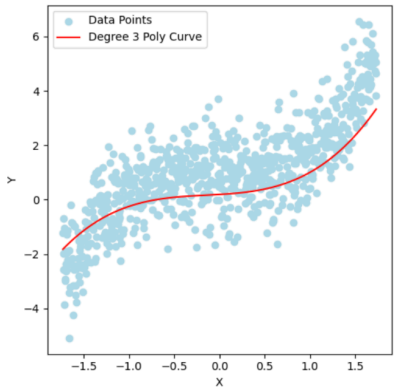
```
def polynomial_features(X,degree):

  X_polynomial = np.ones((X.shape[0], degree + 1))
  for i in range(1, degree + 1):
        X_polynomial[:, i] = np.power(X, i)
  return X_polynomial
```

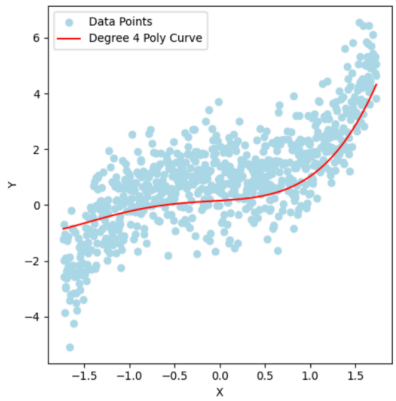
```
def compute_gradient(X,Y,weight):
  m = X.shape[0]
  prediction = np.dot(X,weight)
  gradient = (np.dot(X.T,(prediction - Y)))/m
  return gradient
```

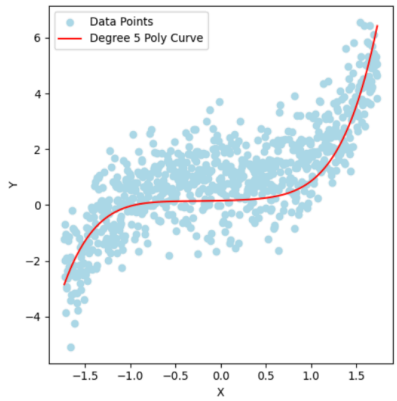
```
def gradient_descent(X,Y,learning_rate,iterations):
  num_features = X.shape[1]
  m = X.shape[0]
  weight = np.zeros(num_features)
  for i in range(1,iterations+1):
    weight = weight - learning_rate * compute_gradient(X,Y,weight)
    if i % 100 == 0:  # Print every 100 iterations to track progress
            print(f"Iteration {i}, Weights: {weight}")  # Print weights
  prediction = np.dot(X,weight)
  cost = (np.square(prediction - Y)).mean()/2
  #cost = np.sqrt(np.mean(np.square(prediction - Y)))/2
  print(f"Final Cost: {cost}")
  return weight
```

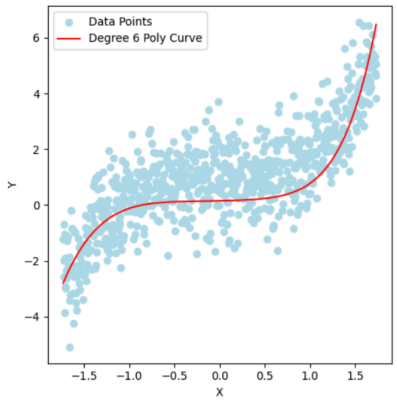

Data Points
Degree 1 Poly Curve
X
Y

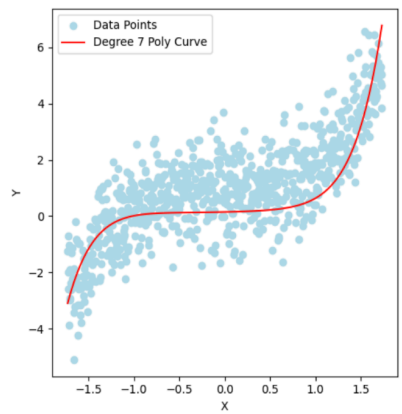

Data Points
Degree 2 Poly Curve
X
Y

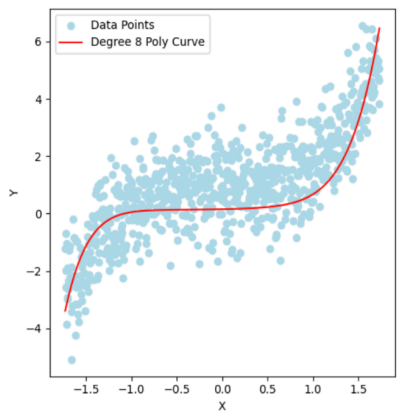

Data Points
Degree 3 Poly Curve
X
Y

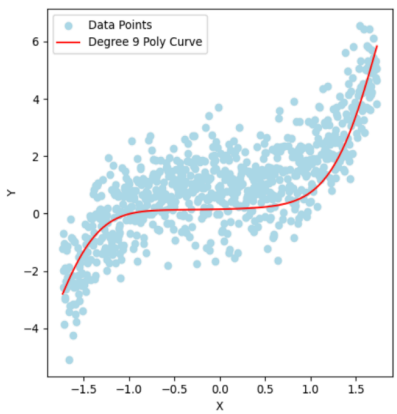

Data Points
Degree 4 Poly Curve
X
Y


Data Points
Degree 5 Poly Curve
X
Y

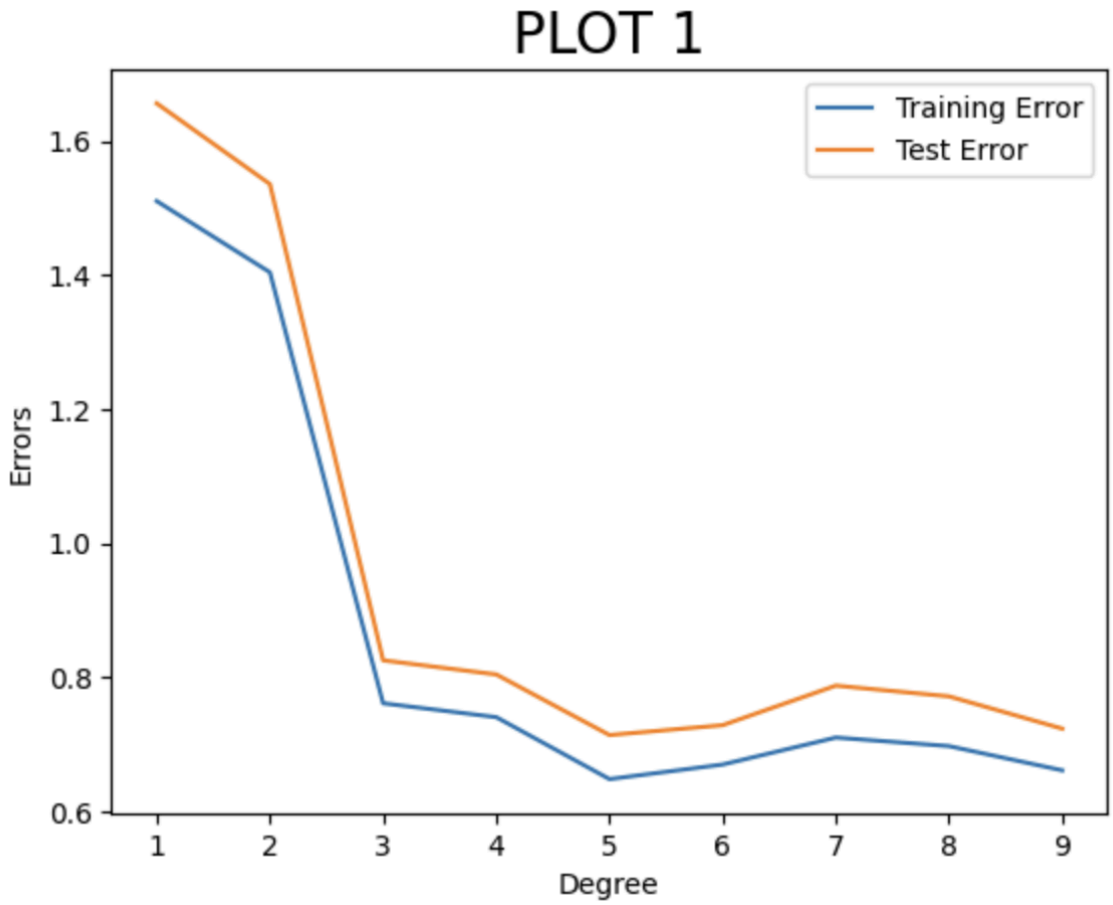

Data Points
Degree 6 Poly Curve
X
Y


Data Points
Degree 7 Poly Curve
X
Y


Data Points
Degree 8 Poly Curve
X
Y


Data Points
Degree 9 Poly Curve
X
Y

We then find the testing error for each model based on the testing dataset(20% of total data). Now, we plot training and testing error versus the degree of the model.



Here, we observe a minimum at degree 5 indicating that degree 5 polynomial fits the best to the data in case of unregularised models.

2) Regularization

Unlike the previous part, we split the dataset into 60% for training, 20% for cross validation and 20% for testing.

```
df=df.sample(frac=1,random_state=8)
train_len=int(0.6*len(df))
valid_len=int(0.8*len(df))
train_set=df[:train_len]
valid_set=df[train_len:valid_len]
test_set=df[valid_len:]
plt.scatter(x=df['X'],y=df['Y'])
```

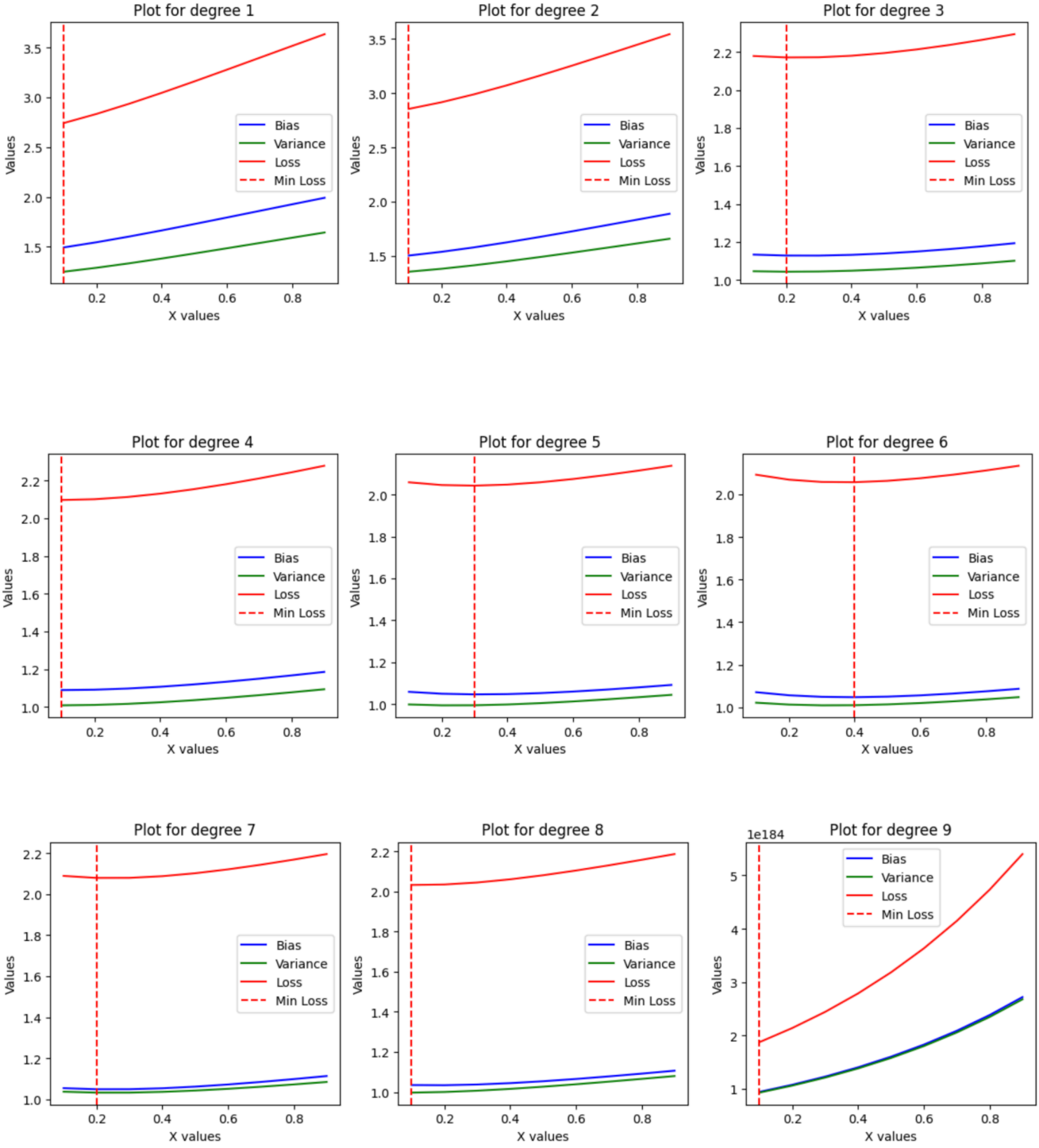
This gradient function includes the regularisation term.

```
[351] def gradient(X,Y,m,w,regularization_term):
        grad=(np.dot(X.T,np.dot(X,w)-Y))/m+(regularization_term*w)
        return grad
```

We used training error for bias^2 and cross validation error as variance according to the heuristics given.

We then find the lambda value for each degree which minimizes the bias^2 + variance value. We iterate through all the degrees and for each degree we iterate through a pre - defined set of lambda values. At the end, we will end up with vectors that contain the weights and lambda values corresponding to the best model for each degree i.e., we will have 9 models. We then plot the bias^2,variance and total error values against the pre-defined lambda values. Using axvline(), we mark the lambda value at which total error is minimized.

PLOT 2


Plot for degree 1
Plot for degree 2
Plot for degree 3
Plot for degree 4
Plot for degree 5
Plot for degree 6
Plot for degree 7
Plot for degree 8
Plot for degree 9
Bias
Variance
Loss
Min Loss
Values
X values
1e184

Now, among the 9 nine models, to decide the best model we are using the testing data(20% of the total dataset) to find the model that gives the least testing error.

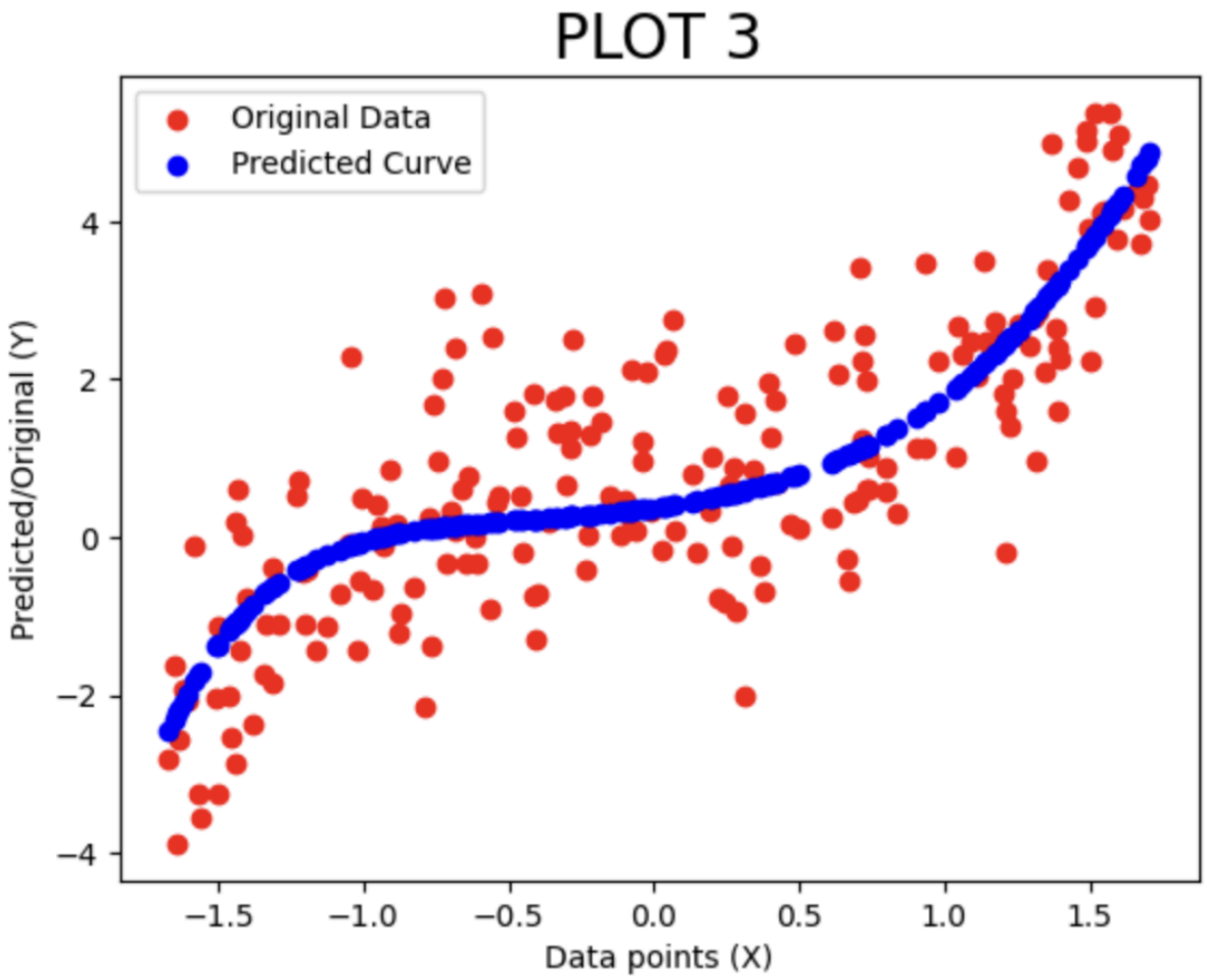
```
curr_test_error = float('inf')
best_index = 0
for i in range(1,degree+1):
  X_poly_test=polynomial_x(test_set['X'].to_numpy(),i)
  prediction = np.dot(X_poly_test,weight_of_all[i-1])
  error = np.mean(np.square(prediction - test_set['Y'].to_numpy()))/2
  if error < curr_test_error:
    curr_test_error = error
    best_index = i
  print(f"The testing error for degree {i} is {error}")
best_weight = weight_of_all[best_index]
best_degree = best_index+1
best_lambda = best_lambda_for_all[best_index]
```

```
The testing error for degree 1 is 0.790016948267199
The testing error for degree 2 is 0.803340049355254
The testing error for degree 3 is 0.6258915140888678
The testing error for degree 4 is 0.6073615441186007
The testing error for degree 5 is 0.5913864770975067
The testing error for degree 6 is 0.5948008970082822
The testing error for degree 7 is 0.6083149774823727
The testing error for degree 8 is 0.6040947246692273
The testing error for degree 9 is 3.7487241953737e+183
```

```
print(f"The best model is of degree is {best_index}")
print(f"The best model has weights {weight_of_all[best_index]}")
print(f"The best model has lambda value {best_lambda_for_all[best_index]}")
print(f"The best model has testing error {curr_test_error}")
print(best_index)
```

```
The best model is of degree is 5
The best model has weights [ 0.3767691   0.50620414  0.47817601  0.31455503  0.0483613   0.09482189
 -0.04499366]
The best model has lambda value 0.4
The best model has testing error 0.5913864770975067
5
```

This is the plot of the best model i.e, 5th degree polynomial with lambda = 0.4



**Best fit model on data points given **

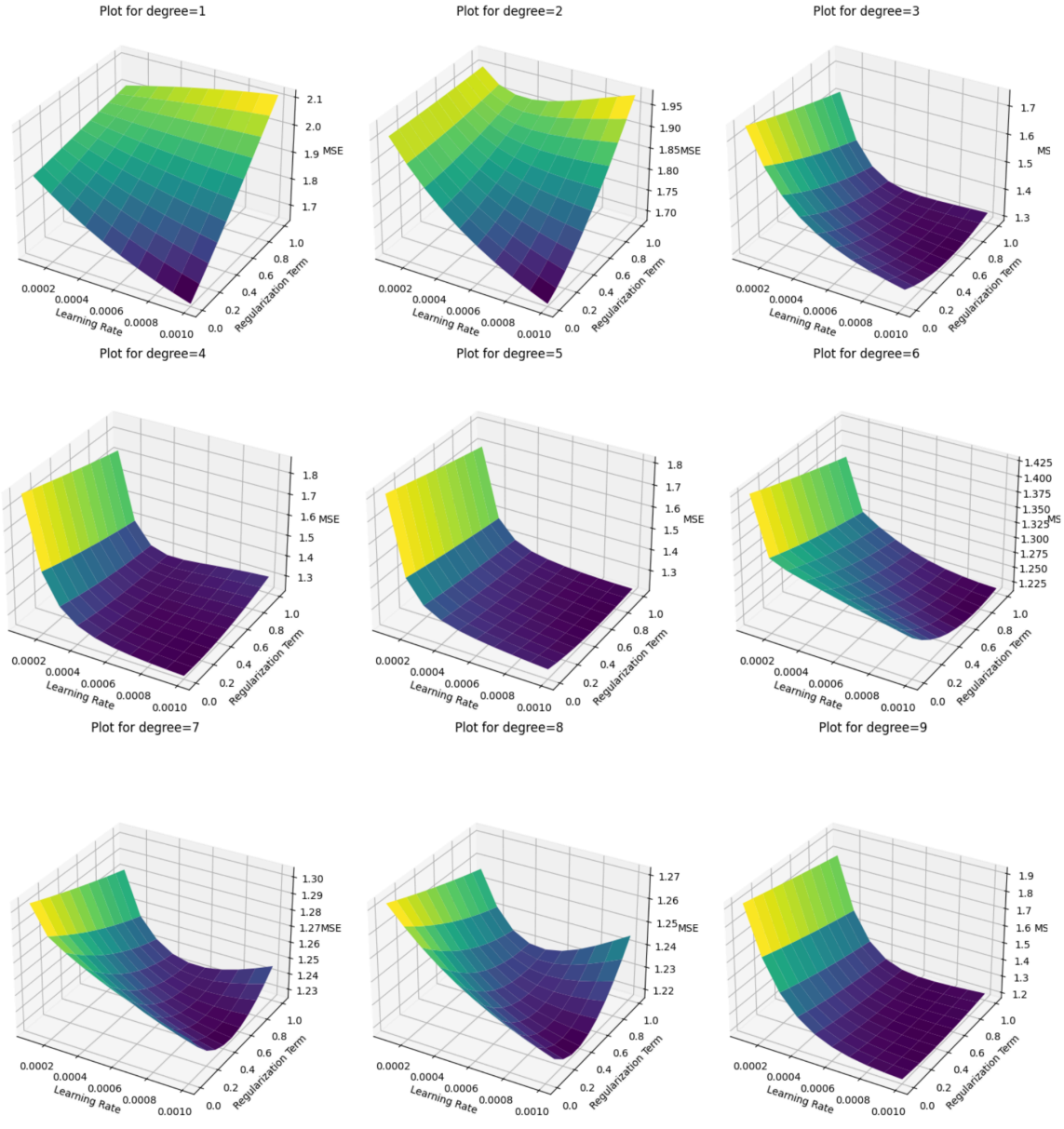
1.Best_degree=5

2.Regularization_term=0.4

3.Minimum loss (MSE)=Bias_sq+Variance= 0.59138647709750675

We now make a 3D dictionary which stores the mse values of training error for each value of degree,regularization constant and learning rate.
We then use this dictionary to make a 3D plot which shows testing error(Z-axis),learning rate(X-axis) and regularization constant(Y-axis).

PLOT 4



COMPARATIVE STUDY OF UNREGULARIZED AND REGULARIZED MODELS

Degree	Unregularised model Error	Regularized model error(with best lambda)
1	1.6563066896157128	0.790016948267199
2	1.5359090726063693	0.803340049355254
3	0.8254868715130638	0.6258915140888678
4	0.8044068053172039	0.6073615441186007
5	0.7139821318470868	0.5913864770975067
6	0.7287007028543571	0.5948008970082822
7	0.7877291644392888	0.6083149774823727
8	0.7717049842202937	0.6040947246692273
9	0.7235573836094301	3.7487241953737e+183

As we see we get lesser error for most regularized models compared to unregularized models. This happens as unregularized models are overfit and regularization ensures the w values are limited.
One model, of degree 9 on regularization, for the chosen learning rate and lambda value, is not able to control the growth of weights. It is giving too much importance to the features and hence performs very poorly on unseen data. Increasing the lambda will be able to control it but in the given constraints of lambda between 0 and 1, it is not able to control the growth of weights.

Also from the values in the table, we can see that both for regularized and unregularized cases, the 5th degree polynomial gives the best results.

Behavior of model on increasing polynomial degree and how overfitting is mitigated by regularization

Error decreases initially as the polynomial degree is increased but after a point, it increases again.

On adding regularization, the testing error is decreasing which hints that better weights have been chosen as compared to initial unregularized weights.

The initial weights were overfit as they allowed all possible weight values whereas the regularized model limits the values of w that can be chosen. That is how overfitting is mitigated by regularization.

How bias and variance change as we move from overfit to underfit models

We knew that as lambda value increases we move from overfitting to underfitting models. In the middle there is some value of lambda which will be best fit.

We observed the plot 2 graphs and see that in plots of degree 3,5,6,7 we find such optimal lambda values between 0 and 1 while for the rest the optimal lambda is not in this range.

We observed the plots 3,5,6,7 and concluded that as we move from overfit to underfit model, the bias-sq increases or decreases

slower than variance and variance decreases or increases slower than bias-sq.

How regularization constant affects the model complexity and performance

When lambda is large, the regularization term has a stronger influence on the loss function. This leads to more penalty on the coefficients, often resulting in many coefficients being pushed toward zero. This results in a simpler model with fewer features and reduced variance, but it may underfit the data.

That is why we don’t get too high lambda values as optimal in our results but rather lambda values like 0.1,0.2 and 0.3.