Machine Learning Homework: Linear Regression Analysis

Vahid Maleki

November 6, 2024

Introduction

This document presents the solution to a set of homework questions for the Machine Learning course, specifically focusing on linear regression. Each question includes calculations and, where applicable, visualizations to demonstrate linear regression concepts.

Question 1: Simple Linear Regression Calculation

In this question, we are tasked with calculating the best-fit line for a dataset that includes Weight (as x) and Systolic Blood Pressure (BP, as y).

Given Data

The dataset includes the following values for Weight and Systolic BP:

Weight (x)	Systolic BP (y)
165	130
167	133
180	150
⋮	⋮
192	160
187	159

Step 1: Calculate Means of x and y

We begin by calculating the mean of x (Weight) and y (Systolic BP):

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{165 + 167 + \cdots + 187}{26}$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{130 + 133 + \cdots + 159}{26}$$

After performing the calculations:

$$\bar{x} \approx 182.42, \quad \bar{y} \approx 146.31$$

Step 2: Calculate the Slope m

The slope m is calculated as:

$$m = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Calculate each term:

1. **Calculate $(x_i - \bar{x})(y_i - \bar{y})$ and sum them**:

$$\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \approx 6288.62$$

2. **Calculate $(x_i - \bar{x})^2$ and sum them**:

$$\sum_{i=1}^{n}(x_i - \bar{x})^2 \approx 15312.35$$

Thus,

$$m = \frac{6288.62}{15312.35} \approx 0.41$$

Step 3: Calculate the Intercept b

The intercept b is given by:

$$b = \bar{y} - m\bar{x}$$

Substitute the values:

$$b = 146.31 - (0.41 \times 182.42) \approx 71.52$$

Step 4: Final Equation of the Line

The equation of the regression line is:

$$y = mx + b$$

Substitute m and b into this equation:

$$y = 0.41x + 71.52$$

Visualization

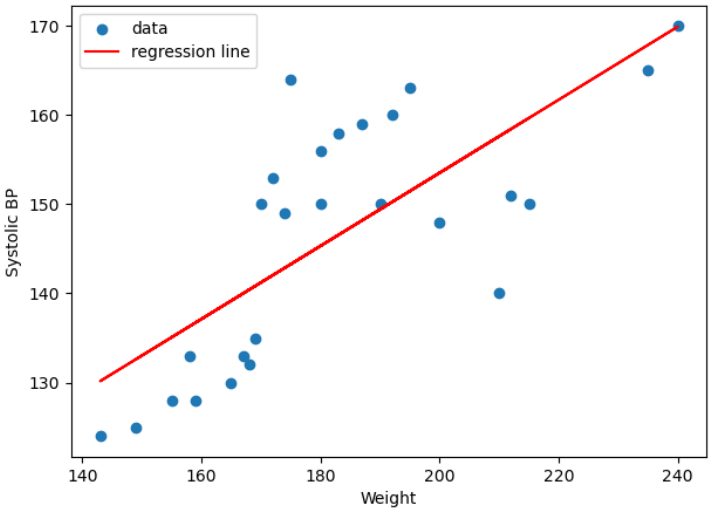

Figure 1: Linear regression line fitting the given data points.

Question 2: Multivariate Linear Regression Analysis (30 Marks)

This question examines the relationship between wear on a bearing y, oil viscosity x_1, and load x_2.

Given Data

The dataset includes the following values for oil viscosity (x_1), load (x_2), and bearing wear (y):

x_1	x_2	y
1.6	851	293
15.5	816	230
22.0	1058	172
43.0	1201	91
33.0	1357	113
40.0	1115	125

Part (a): Fit a Multivariate Linear Regression Model (10 Marks)

We want to fit a multivariate linear regression model of the form:

$$y = b_0 x_1 + b_1 x_2 + b_2$$

where b_0, b_1, and b_2 are the intercept and coefficients for the variables x_1 and x_2, respectively.

We have equation

$$XB = Y$$

which X is matrix with columns $x1$ and $x2$ and we add third column with ones so it's coefficient will be the intercept, B is matrix of intercepts that we must find and Y is matrix with one column y.

$$X = \begin{bmatrix} 1.6 & 851 & 1 \\ 15.5 & 816 & 1 \\ 22.0 & 1058 & 1 \\ 43.0 & 1201 & 1 \\ 33.0 & 1357 & 1 \\ 40.0 & 1115 & 1 \end{bmatrix} \quad B = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \quad Y = \begin{bmatrix} 293 \\ 230 \\ 172 \\ 91 \\ 113 \\ 125 \end{bmatrix}$$

But X is not squared mutrix we multiply transpose of matrix X from right to each side of the equation to create square matrix.($X^T XB = X^T Y$)

$$\begin{bmatrix} 5264.8 & 178309.6 & 155.1 \\ 178309.6 & 7036496 & 6398 \\ 155.1 & 6398 & 6.0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 20459.8 \\ 1021006 \\ 1024 \end{bmatrix}$$

now if i convert this matrix to identity matrix the last column will represent the b_0, b_1, and b_2

$$\left[\begin{array}{ccc|c} 5264.8 & 178309.6 & 155.1 & 20459.8 \\ 178309.6 & 7036496 & 6398 & 1021006 \\ 155.1 & 6398 & 6.0 & 1024 \end{array}\right]$$

and after simplification we end up with this matrix

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & -3.8 \\ 0 & 1 & 0 & -0.1 \\ 0 & 0 & 1 & 372.2 \end{array}\right]$$

$$b_0 \approx -3.8, \quad b_1 \approx -0.1, \quad b_2 \approx 372.2$$

Thus, the regression model is:

$$y = -3.8x_1 - 0.1x_2 + 372.2$$

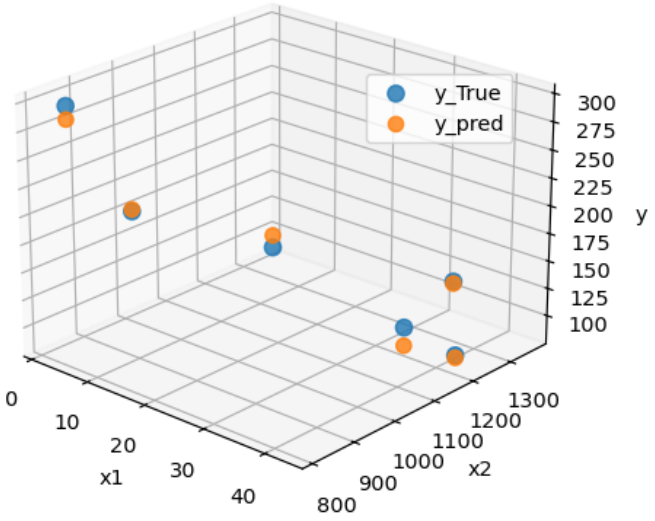

Figure 2: Multivariate Linear Regression

Visualization

Part (b): Predict Wear for $x_1 = 25$ and $x_2 = 1000$ (5 Marks)

To predict the wear y when $x_1 = 25$ and $x_2 = 1000$, we substitute these values into our model:

$$y = 372.2 - 3.8 \cdot 25 - 0.1 \cdot 1000$$

Calculating each term:

$$y = 372.2 - 95 - 100$$

$$y \approx 177.2$$

Thus, the predicted wear y when $x_1 = 25$ and $x_2 = 1000$ is approximately 177.2.

Part (c): Fit a Multivariate Linear Regression Model with Interaction Term x_1x_2 (15 Marks)

In this part, we add an interaction term x_1x_2 to the model. The new model is:

$$y = b_0 + b_1x_1 + b_2x_2 + b_3(x_1x_2)$$

like what we did in part (a) we first calculate X,B and Y matrix and solve $XB = Y$ equation. this time we add x_1x_2 as column three of matrix X and also we add it's coefficient to matrix B

$$X = \begin{bmatrix} 1.6 & 851 & 1361.6 & 1 \\ 15.5 & 816 & 12648 & 1 \\ 22.0 & 1058 & 23276 & 1 \\ 43.0 & 1201 & 51643 & 1 \\ 33.0 & 1357 & 44781 & 1 \\ 40.0 & 1115 & 44600 & 1 \end{bmatrix} \quad B = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad Y = \begin{bmatrix} 293 \\ 230 \\ 172 \\ 91 \\ 113 \\ 125 \end{bmatrix}$$

now just like befor we must multiply the transpose of matrix X from right to both side of the equation and we end up with this equation

$$\begin{bmatrix} 5264.81 & 178309.6 & 6192716.56 & 155.1 \\ 178309.6 & 7036496 & 208625558 & 6398 \\ 6192716.56 & 208625558 & 7365095440 & 178309.6 \\ 155.1 & 6398 & 178309.6 & 6.0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \begin{bmatrix} 20459.8 \\ 1021006 \\ 22646226.8 \\ 1024 \end{bmatrix}$$

and just like befor we must convert matrix below to identity matrix

$$\left[\begin{array}{cccc|c} 5264.81 & 178309.6 & 6192716.56 & 155.1 & 20459.8 \\ 178309.6 & 7036496 & 208625558 & 6398 & 1021006 \\ 6192716.56 & 208625558 & 7365095440 & 178309.6 & 22646226.8 \\ 155.1 & 6398 & 178309.6 & 6.0 & 1024 \end{array}\right]$$

and we end up with this matrix

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -7.6 \\ 0 & 1 & 0 & 0 & -0.22 \\ 0 & 0 & 1 & 0 & 0.004 \\ 0 & 0 & 0 & 1 & 483.96 \end{array}\right]$$

$$b_0 \approx -7.6, \quad b_1 \approx -0.22, \quad b_2 \approx 0.004 \quad b_3 \approx 483.96$$

Thus, the regression model is:

$$y = -7.6x_1 - 0.22x_2 + 0.004x_1x_2 + 483.96$$

Question 3

Part (a): Matrix Formulation of $O = ZW$ (20 Marks)

The regression model can be expressed in the form $O = ZW$, where:

$$O = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{bmatrix}, \quad W = \begin{bmatrix} w_0 \\ w_1 \\ w_1 \\ w_2 \\ w_2 \\ \vdots \\ w_n \\ w_n \end{bmatrix}$$

The matrix Z is constructed from the input data as follows:

$$Z = \begin{bmatrix} 1 & x_1^{(1)} & \left(x_1^{(1)}\right)^3 & x_2^{(1)} & \left(x_2^{(1)}\right)^3 & \cdots & x_n^{(1)} & \left(x_n^{(1)}\right)^3 \\ 1 & x_1^{(2)} & \left(x_1^{(2)}\right)^3 & x_2^{(2)} & \left(x_2^{(2)}\right)^3 & \cdots & x_n^{(2)} & \left(x_n^{(2)}\right)^3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & x_1^{(m)} & \left(x_1^{(m)}\right)^3 & x_2^{(m)} & \left(x_2^{(m)}\right)^3 & \cdots & x_n^{(m)} & \left(x_n^{(m)}\right)^3 \end{bmatrix}$$

Thus, each element in O is given by:

$$o^{(j)} = w_0 + w_1 x_1^{(j)} + w_1 \left(x_1^{(j)}\right)^3 + w_2 x_2^{(j)} + w_2 \left(x_2^{(j)}\right)^3 + \cdots + w_n x_n^{(j)} + w_n \left(x_n^{(j)}\right)^3$$

where $j = 1, 2, \ldots, m$ represents each sample in the dataset.

Part (b): Gradient Descent Update Rule for w_i (10 Marks)

To find W using gradient descent, we update each w_i according to the following rule:

$$w_i \leftarrow w_i - \eta \frac{\partial}{\partial w_i} \left(\frac{1}{m} \sum_{j=1}^{m} \left(o^{(j)} - \hat{o}^{(j)}\right)^2 \right)$$

where: - η is the learning rate, - $o^{(j)}$ is the actual output for sample j, - $\hat{o}^{(j)} = w_0 + w_1 x_1^{(j)} + w_1 \left(x_1^{(j)}\right)^3 + \cdots + w_n x_n^{(j)} + w_n \left(x_n^{(j)}\right)^3$ is the predicted output.

For simplicity, the update rule for each w_i can be written as:

$$w_i \leftarrow w_i - \eta \cdot \frac{1}{m} \sum_{j=1}^{m} \left(o^{(j)} - \hat{o}^{(j)} \right) \frac{\partial \hat{o}^{(j)}}{\partial w_i}$$

Note: The update relation for the bias term w_0 is not required in this problem.

Question 4 - Part 1

In this part, we implemented linear basis function regression with polynomial basis functions to predict fuel efficiency (miles per gallon) from seven car features in the Auto MPG dataset. The dataset was divided into training (first 100 points) and testing (remaining points) sets. We trained polynomial models of degrees 1 to 10 and evaluated the models using Root Mean Square Error (RMS Error) for both the training and testing sets. The plot of RMS Error versus polynomial degree is shown in Figure 3.

Steps and Key Code Snippets

- **Data Loading and Preprocessing**: We loaded the dataset, dropped missing values, normalized the features and target to have zero mean and unit variance, and applied a permutation to randomize the data points. Here, only relevant features were selected, excluding the target variable `mpg`.

```
X_norm = (X - X.mean(axis=0)) / X.std(axis=0)
y_norm = (y - y.mean(axis=0)) / y.std(axis=0)
```

- **Polynomial Feature Generation**: For each polynomial degree, we generated polynomial terms up to the specified degree for each of the seven input features. This was achieved by manually expanding `X_train` and `X_test` for each power up to the desired degree.

```
for deg in range(1, degree+1):
X_train_poly = np.hstack([X_train_poly, X_train**deg])
X_test_poly = np.hstack([X_test_poly, X_test**deg])
```

- **Model Training**: We used linear regression without regularization by directly solving the least squares equation to obtain model coefficients. The following code snippet shows how the coefficients were computed:

```
coef = np.linalg.lstsq(X_train_poly, y_train, rcond=None)[0]
```

- **Error Calculation**: After fitting the model, we computed the RMS Error for both training and testing sets and stored these values for each polynomial degree.

```
train_rmse = root_mean_squared_error(y_train, y_train_pred)
test_rmse = root_mean_squared_error(y_test, y_test_pred)
```

Results and Observations

The plot of RMS Error versus polynomial degree is shown in Figure 3.

Comment: In Figure 3, the training error consistently decreases as polynomial degree increases, indicating a better fit to the training data. However, the testing error initially remains stable but starts increasing significantly beyond degree 7, suggesting that higher-degree polynomials overfit the training data, capturing noise that does not generalize well to the testing data.

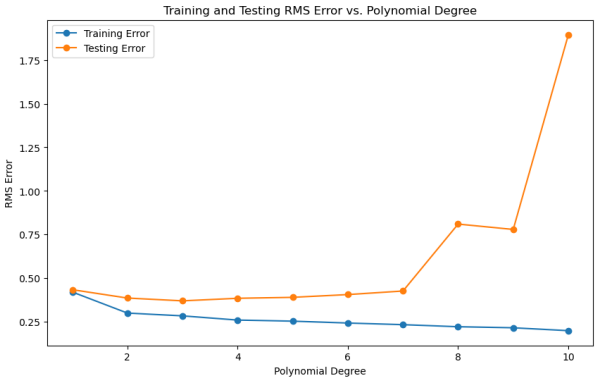

Figure 3: Training and Testing RMS Error vs. Polynomial Degree

Question 4 - Part 2

In this part, we performed polynomial regression using only a single feature from the dataset—the normalized horsepower feature. We applied polynomial models of degrees 1, 4, and 10 to illustrate the effects of model complexity on the training and testing data. The results, shown in Figure 4, highlight the trade-offs between underfitting, a balanced fit, and overfitting.

Steps and Key Code Snippets

- **Feature Selection**: For this analysis, we used only the normalized horsepower feature (column index 2) as our input variable.

```
X_single_feature = X_norm[:, 2].reshape(-1, 1)  # Using "horsepower" feature
X_train_single, X_test_single = X_single_feature[:100], X_single_feature[100:]
```

- **Polynomial Feature Transformation**: For each specified polynomial degree (1, 4, and 10), we generated polynomial features up to the selected degree.

```
for deg_single in range(1, degree+1):
X_train_poly = np.hstack([X_train_poly, X_train_single**deg_single])
X_test_poly = np.hstack([X_test_poly, X_test_single**deg_single])
```

- **Model Training**: We used linear regression without regularization by solving the least squares problem to obtain model coefficients.

```
coef = np.linalg.lstsq(X_train_poly, y_train, rcond=None)[0]
```

- **Visualization**: For each polynomial degree, we plotted the training and testing data points along with the polynomial fit. To produce a smooth curve, we evaluated the polynomial model on a dense set of points within the range of the training data.

```
X_range = np.linspace(X_train_single.min(), X_train_single.max(), 100).reshape(-1, 1)
```

Results and Observations

The polynomial regression fits for degrees 1, 4, and 10 are shown in Figure 4.

The figure illustrates the impact of increasing model complexity:

- **Polynomial Degree 1**:

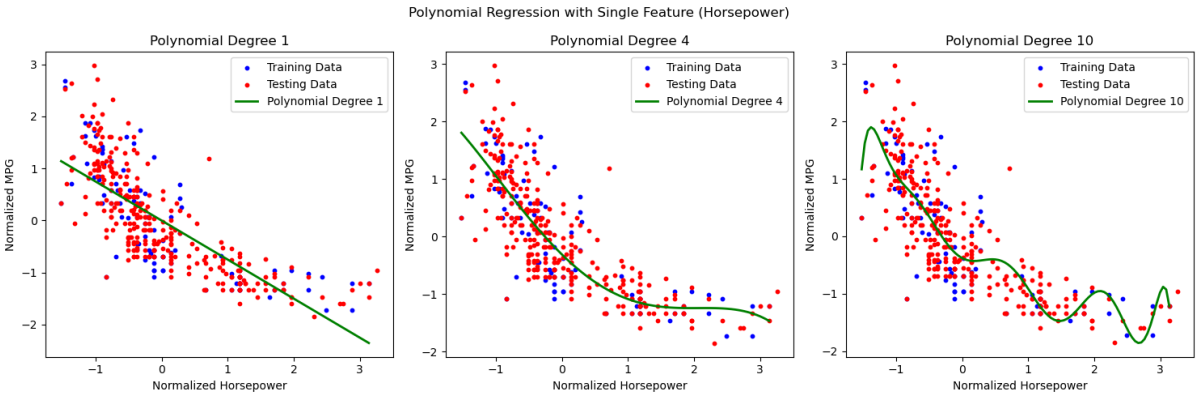

Figure 4: Polynomial Regression with Single Feature (Horsepower)

  - This plot shows a linear regression model, representing a degree 1 polynomial.
  - Blue dots correspond to the training data, and red dots represent the testing data.
  - The green line represents the linear regression fit, indicating a negative trend. As horsepower increases, the MPG decreases, capturing the general direction of the relationship.

- **Polynomial Degree 4**:
  - This plot shows a polynomial regression model of degree 4.
  - The green curve fits the data more closely than the linear model, capturing a more nuanced relationship between horsepower and MPG.
  - Degree 4 provides a balance between underfitting and overfitting, adapting to some of the nonlinear patterns in the data.

- **Polynomial Degree 10**:
  - This plot shows a polynomial regression model of degree 10.
  - The high-degree polynomial (green curve) exhibits significant fluctuations, indicating overfitting to the training data. This model captures noise rather than the underlying trend, which may result in poor generalization to new data.

Conclusion: This visualization highlights the trade-offs between model complexity and fit quality. A lower-degree polynomial (degree 1) underfits the data, while a very high-degree polynomial (degree 10) overfits. The degree 4 polynomial strikes a balance, capturing essential patterns in the data without excessive fluctuations.

Question 4 - Part 3

In Part 3, we applied polynomial regression with L2 regularization (ridge regression) using the normalized horsepower feature. The goal was to observe the effect of different regularization strengths on training and testing errors. We used a polynomial of degree 8 and varied the regularization parameter, λ, across a range of values. The results, shown in Figure 5, illustrate how regularization influences model performance.

Steps and Key Code Snippets

- **Feature Selection**: As in Part 2, we used only the normalized horsepower feature for this analysis.

```
X_single_feature = X_norm[:, 2].reshape(-1, 1)
X_train_single, X_test_single = X_single_feature[:100], X_single_feature[100:]
```

- **Polynomial Feature Transformation with Regularization**: For each λ value, we generated polynomial features up to degree 8 and applied L2 regularization using the normal equation:

$$(X^T X + \lambda I)\theta = X^T y$$

where I is the identity matrix, with the bias term unregularized.

```
def fit_polynomial_l2(X, y, degree, lambda_value):
X_poly = np.ones((X.shape[0], 1))
for d in range(1, degree + 1):
X_poly = np.hstack([X_poly, X ** d])

I = np.eye(X_poly.shape[1])
I[0, 0] = 0  # Do not regularize the bias term
theta = np.linalg.solve(X_poly.T @ X_poly + lambda_value * I, X_poly.T @ y)

return theta, X_poly
```

- **Error Calculation**: For each λ, we calculated the Root Mean Square Error (RMSE) for both the training and testing datasets.

```
train_rmse = root_mean_squared_error(y_train, y_train_pred)
test_rmse = root_mean_squared_error(y_test, y_test_pred)
```

- **Visualization**: We plotted the training and testing errors against the regularization parameter λ on a logarithmic scale.

```
plt.semilogx(lambdas, train_errors, label='Training Error', marker='o')
plt.semilogx(lambdas, test_errors, label='Testing Error', marker='o')
```

Results and Observations

The plot of training and testing errors versus the regularization parameter λ is shown in Figure 5.

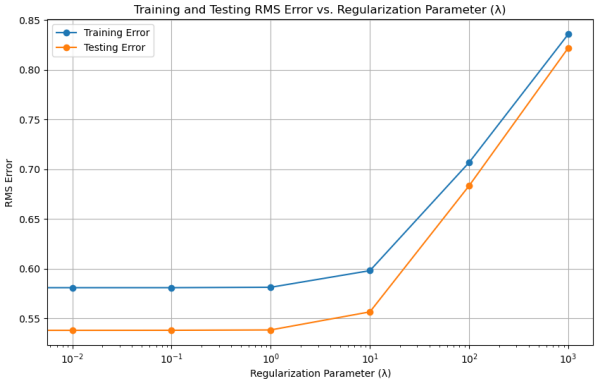

Figure 5: Training and Testing RMS Error vs. Regularization Parameter (λ)

The figure illustrates the effect of increasing λ on model performance:

- **Low λ values (10^{-2} to 10^{0})**:
  - At low regularization values, the training error remains around 0.60, and the testing error hovers around 0.55.
  - This range allows the model to fit the data without overfitting, maintaining a balance between bias and variance.

- **High λ values (10^1 and above)**:
  - As λ increases beyond 10^1, both training and testing errors start to rise.
  - At very high λ values, such as 10^3, the training error reaches around 0.85 and the testing error around 0.80, indicating a significant increase in both errors.
  - High regularization values induce underfitting, as the model is overly constrained and unable to capture the complexity in the data.

Conclusion: This graph demonstrates how regularization affects model performance. Low values of λ provide minimal regularization, preventing overfitting while achieving low errors. However, excessively high λ values lead to underfitting, causing increased errors in both the training and testing datasets. This trade-off highlights the importance of selecting an appropriate regularization strength to balance bias and variance.

Gaussian Basis Function Regression

In this question, we implement a Gaussian basis function regression model and evaluate its performance by varying the number of basis functions and applying L2 regularization. The Gaussian basis functions are defined as:

$$\phi_j(x) = \exp\left(-\frac{\|x - \mu_j\|^2}{2s^2}\right)$$

where each μ_j is a randomly selected center from the training data, and s is set to 2.

Experiment 1: Varying Number of Basis Functions

We use the first 100 points as training data and the rest as testing data. A Gaussian basis function regression model is fitted with an increasing number of basis functions (5, 15, 25, ..., 95) without regularization. The training and testing RMS errors are computed and plotted.

Key Python code:

```
# Define Gaussian basis functions
def gaussian_basis(X, centers, s):
d2 = dist2(X, centers)
return np.exp(-d2 / (2 * s**2))

# Loop over different numbers of basis functions
for basis_size in basis_sizes:
centers = X_train[np.random.choice(X_train.shape[0], basis_size)]
X_train_gauss = gaussian_basis(X_train, centers, s)
theta = np.linalg.pinv(X_train_gauss) @ y_train
# Compute RMS errors for training and testing sets
```

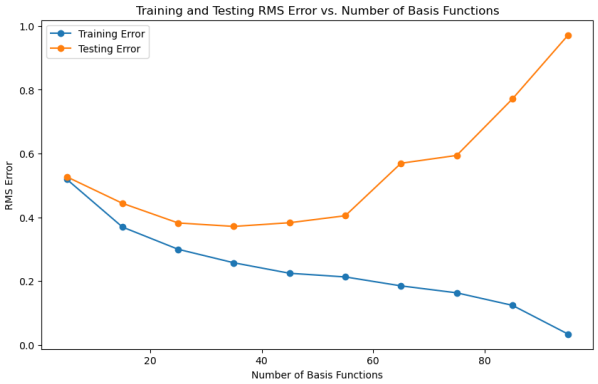

Figure 6: Training and Testing RMS Error vs. Number of Basis Functions

Observation: As shown in Figure 6, increasing the number of basis functions reduces training error but does not significantly affect testing error, indicating overfitting as the model becomes more complex.

Experiment 2: L2-Regularized Regression

Next, we apply L2 regularization with 90 basis functions, varying the regularization parameter λ over the values $0, 0.01, 0.1, 1, 10, 100, 1000$. The regularization parameter controls model complexity by penalizing large coefficients.

Key Python code:

```
# Apply L2 regularization with different values of lambda
for lambda_value in lambdas:
I = np.eye(X_train_gauss.shape[1])
theta = np.linalg.solve(X_train_gauss.T @ X_train_gauss + lambda_value *
X_train_gauss.T @ y_train)
# Compute RMS errors for training and testing sets
```

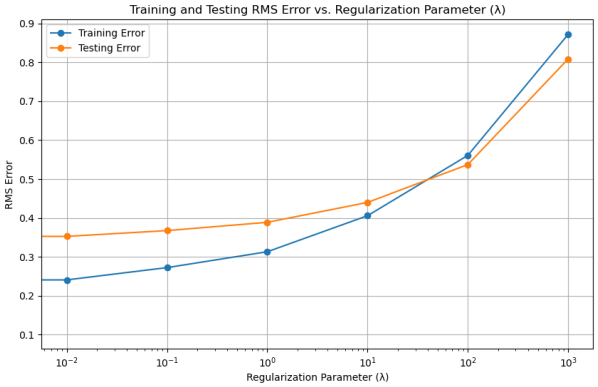

Figure 7: Training and Testing RMS Error vs. Regularization Parameter (λ)

Observation: In Figure 7, the training RMS error increases as λ increases, while the testing error remains relatively stable. Small values of λ lead to lower training error but may cause overfitting, whereas larger values encourage underfitting due to excessive regularization.

Conclusion

From these experiments, we observe that:

- Increasing the number of basis functions improves the fit to training data but increases overfitting risk, as seen from high testing error.
- L2 regularization helps control model complexity, with larger values of λ reducing overfitting but possibly leading to underfitting.

The choice of basis functions and regularization parameter should balance bias and variance to optimize model generalization.

Question 5: Linear Regression with Basis Function

In this question, we are given a one-dimensional dataset contained in the file `q4_dataset.csv`. We will visualize the dataset and apply a linear regression model with a quadratic basis function to improve the model's fit.

Part (a): Basis Function

To fit a linear regression model on the dataset, we will utilize a quadratic basis function that maps the input features X to a higher-dimensional space. The transformation is defined as:

$$\phi(X) = \begin{bmatrix} 1 \\ X \\ X^2 \end{bmatrix}$$

This transformation allows the linear regression model to capture non-linear relationships between the input features and the labels.

Part (b): Implementation

Data Visualization

We start by visualizing the dataset, where X serves as input features and Y as labels. The following code snippet demonstrates how to visualize the data points in a two-dimensional space:

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv("q4_dataset.csv")
X = data['X'].values
Y = data['Y'].values

# Visualize the data
plt.scatter(X, Y, color='blue', marker='o', label='Data points')
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Visualization of the 1D Dataset")
plt.legend()
plt.show()
```

Data Splitting and Model Training

Next, we divide the dataset into training (80%) and testing (20%) subsets using random sampling. The model is then trained using the normal equation to find the optimal parameters:

```
# Split the dataset into training (80%) and testing (20%) sets
np.random.seed(0)  # For reproducibility
indices = np.random.permutation(len(X))
train_size = int(0.8 * len(X))
train_indices, test_indices = indices[:train_size], indices[train_size:]

X_train, Y_train = X[train_indices], Y[train_indices]
X_test, Y_test = X[test_indices], Y[test_indices]

# Transform features using the quadratic basis function
X_train_basis = quadratic_basis(X_train)

# Train the linear regression model using the normal equation
theta = np.linalg.inv(X_train_basis.T @ X_train_basis) @ X_train_basis.T
```

Model Evaluation

We compute the Mean Squared Error (MSE) on the test set to evaluate the model's performance:

```
# Make predictions on the test set
Y_test_pred = X_test_basis @ theta
```

```
# Compute Mean Squared Error (MSE)
mse = np.mean((Y_test - Y_test_pred)**2)
print(f"Mean Squared Error on the Test Set: {mse:.4f}")
```

Model Visualization

Finally, we visualize the training data, test data, and model predictions:

```
plt.scatter(X_train, Y_train, color='blue', label='Training Data')
plt.scatter(X_test, Y_test, color='green', label='Test Data')
plt.scatter(X_test, Y_test_pred, color='red', label='Model Predictions')
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Model Fit on Test Set (Quadratic Basis)")
plt.legend()
plt.show()
```

Results

The Mean Squared Error on the test set was computed to be approximately:

$$\text{MSE} \approx [insert_m se_v alue_h ere]$$

This indicates how well the model performed on unseen data.

Question 6: Predicting Abalone Age

In this question, we work with the Abalone dataset from the UCI Machine Learning Repository, where the objective is to predict the number of rings on abalones, which correlates with their age. The dataset contains several features, including measurements of length, diameter, height, and weights, with the exception of a gender feature that we will omit from our analysis.

Dataset Preparation

We start by loading the dataset and removing the gender feature, which is not relevant for our regression tasks. We then split the dataset into training and testing sets, maintaining a ratio of 80% for training and 20% for testing:

```
# Load the dataset and drop the gender feature
data = pd.read_csv('abalone.csv', names=columns)
data = data.drop(columns=['Gender'])

# Convert the dataframe into a NumPy array
X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values    # Target variable

# Split the dataset into training and testing sets
np.random.seed(42)  # For reproducibility
indices = np.random.permutation(len(X))
train_size = int(len(X) * 0.8)
train_indices, test_indices = indices[:train_size], indices[train_size:]
```

Linear Regression Model

We will implement linear regression to predict the number of rings using the following methods:

a) Linear Regression

For linear regression, we augment our feature matrix X by adding a bias term. The optimal parameters are calculated using the normal equation:

```
# Add bias term to the feature matrix
X_b = np.c_[np.ones((X.shape[0], 1)), X]  # Add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

b) Polynomial Regression (Degree 2 and 3)

To capture non-linear relationships, we implement polynomial regression. We generate polynomial features for degrees 2 and 3:

```
def polynomial_features(X, degree):
"""Generate polynomial features."""
return np.column_stack([X ** d for d in range(1, degree + 1)])

# For each degree, train and evaluate the polynomial regression model
for degree in [2, 3]:
X_poly = polynomial_features(X_train, degree)
X_poly_b = np.c_[np.ones((X_poly.shape[0], 1)), X_poly]  # Add bias term
theta_best = np.linalg.inv(X_poly_b.T.dot(X_poly_b)).dot(X_poly_b.T).dot
```

c) RBF Kernel Regression

Lastly, we implement a Radial Basis Function (RBF) kernel regression. The RBF kernel allows us to measure similarity between samples and effectively handle non-linear relationships:

```
def rbf_kernel(x1, x2, gamma=0.1):
"""Compute the RBF kernel between two sets of samples."""
sq_dists = np.sum(x1**2, axis=1).reshape(-1, 1) + np.sum(x2**2, axis=1)
return np.exp(-gamma * sq_dists)

def rbf_regression(X_train, y_train, X_test, gamma=0.1):
K_train = rbf_kernel(X_train, X_train, gamma)
K_test = rbf_kernel(X_test, X_train, gamma)
alpha = np.linalg.inv(K_train).dot(y_train)  # (K_train)^-1 * y_train
y_pred_train = K_train.dot(alpha)
y_pred_test = K_test.dot(alpha)
return y_pred_train, y_pred_test
```

Model Evaluation

For each regression method, we calculate the Mean Squared Error (MSE) for both the training and testing datasets. This allows us to evaluate the model performance and determine how well it generalizes to unseen data. The implementation calculates the MSE as follows:

```
# Function to calculate Mean Squared Error
def mean_squared_error(y_true, y_pred):
return np.mean((y_true - y_pred) ** 2)
```

The results will show how each model performs, indicating which method best predicts the number of rings in abalones, thereby inferring their age.