Hyejoon Lee & Alexis Balayre

Multiple Linear Regression

School of Aerospace, Transport and Manufacturing
Computational Software of Techniques Engineering

MSc
Academic Year: 2023 - 2024

Supervisors: Dr Stuart Barnes & Dr Peter Sherar
23th October 2023

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Linear regression is a powerful tool for predicting a single quantitative (real-valued) variable based on any number of real-valued predictors or features. A regression model that has only one independent variable is considered to be the simplest one. However, many real-world scenarios involve situations where changes in the dependent variable, denoted as 'y,' are influenced by multiple independent variables. These linear regression models, which use several independent variables to explain variations in the dependent variable, are called numerous regression models. As mentioned earlier, a multiple regression model involves two or more independent variables, as shown in the equation below:

$$y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_k x_{ik} + \varepsilon_i \tag{1.1}$$

By default, linear regression assumes that the relationship between independent and dependent variables is linear and determines the best-fitting line with the least Mean Squared Error (MSE), which measures the difference between the actual data and the predicted values. However, there are instances where the inherent pattern in the data is not sufficiently captured by a linear relationship. In such cases, polynomial regression becomes a more suitable alternative. It can be defined using the following formula:

$$y = \omega_0 + \omega_1 x + .. + \omega_n x^n \tag{1.2}$$

As the order of the polynomial increases, so does the complexity of the model. The upcoming analysis will focus on exploring datasets that pose a challenge to predict using linear regression. The aim is to design an optimal model that works well in these scenarios. The analysis will also involve an examination of the relationship between multiple independent and dependent variables, with the goal of creating a polynomial regression model.

Chapter 2

Datasets

Advertising Dataset

The dataset contains advertising and sales data seen in class. To improve sales of a particular product some data consisting of the sales of that product in 200 different markets, along with advertising budgets for the product in each of those markets for three various media: TV, radio, and newspaper.

	TV	radio	newspaper	sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	9.3
4	151.5	41.3	58.5	18.5
5	180.8	10.8	58.4	12.9

Table 2.1: Advertising Data Sample

	TV	radio	newspaper	sales
count	200.000000	200.000000	200.000000	200.000000
mean	147.042500	23.264000	30.554000	14.022500
std	85.854236	14.846809	21.778621	5.217457
min	0.700000	0.000000	0.300000	1.600000
25%	74.375000	9.975000	12.750000	10.375000
50%	149.750000	22.900000	25.750000	12.900000
75%	218.825000	36.525000	45.100000	17.400000
max	296.400000	49.600000	114.000000	27.000000

Table 2.2: Advertising Data Statistics

Auto Dataset

The auto dataset gives some information about cars, auto gas mileage, horse-power and other information. In particular, the variables are mpg, cylinders, displacement, horsepower, weight, acceleration, year origin, and name.

	mpg	cyl.	disp.	hp	wt.	accel.	yr	org	name
0	18.0	8	307.0	130.0	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	1	Plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	1	ford torino

Table 2.3: Auto Data Sample

	mpg	cyl.	disp.	hp	wt.	accel.	yr	org
count	392	392	392	392	392	392	392	392
mean	23.45	5.47	194.41	104.47	2977.58	15.54	76	1.58
std	7.81	1.71	104.64	38.49	849.40	2.76	3.68	0.81
min	9.0	3	68.0	46.0	1613	8.0	70	1
25%	17.0	4	105.0	75.0	2225.25	13.78	73	1
50%	22.75	4	151.0	93.5	2803.5	15.5	76	1
75%	29.0	8	275.75	126.0	3614.75	17.03	79	2
max	46.6	8	455.0	230.0	5140	24.8	82	3

Table 2.4: Auto Data Statistics

Chapter 3

Methodologies

3.1 Multicolinearity

Linear regression is based on four fundamental assumptions. Firstly, the relationship between the dependent variable and the independent variables should be linear. Secondly, the dependent variables should be normally distributed and have the same variance. Thirdly, the independent variables should be independent of each other.

To create an accurate model, dependent and independent variables in advertising.csv and auto.csv will be analysed in these four ways. When there is multicollinearity in the data, it becomes difficult to determine the exact impact of each variable on the dependent variable. By removing the columns with high multicollinearity from the base model using heatmaps and the VIF library, a model with less error can be created.

1. **Heatmap**

   It is a visualisation technique that shows the correlation between data using the degree of colour. If you input the table value that calculates the correlation index using the corr() function as the input of the heatmap, you can visually check the one-to-one correlation between each variable.

2. **VIF(Variance Inflation Factor)**

   It shows the performance of linear regression of one independent variable to other independent variables, and if the correlation between the dependent variable and other variables is large, the r-squared value of the regression equation is high, and the vif index is inflated.

3.1.1 Data Pre-processing

3.1.1.1 Advertising.csv

The dataset's heatmap and VIF index were analysed to detect the presence of multicollinearity. The results indicate that there was no evidence of multicollinearity between the variables as the correlation between each predictor was low, as shown in Figure 3.1 and Table 3.1.

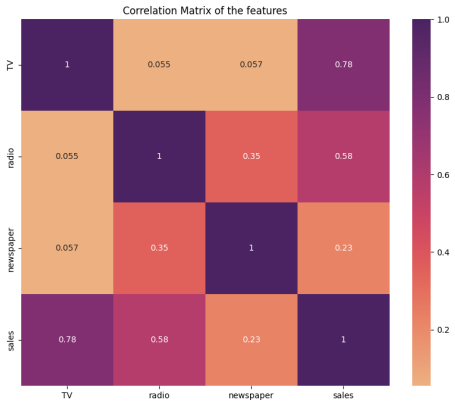


Figure 3.1: Heat map of advertising dataset

	VIF Factor	Feature
0	2.486772	TV
1	3.285462	radio
2	3.055245	newspaper

Table 3.1: VIF factor

Due to the robust correlation observed between 'sales' and the 'TV' variable and the limited insights offered by the 'newspaper' data, a comparative analysis was conducted to assess the impact of including or excluding the 'newspaper' variable. This investigation encompassed the development of two distinct multiple regression models.

The first model, referred to as a multiple linear regression model, considered all three independent variables individually, as delineated in Equations (3.1) and (3.2). In contrast, the second model employed a multiple regression model with interaction, where the product of two variables was considered as a nonlinear term to account for the interaction effects of each independent variable, as illustrated in Equation (3.3).

$$sales = \theta_0 + \theta_1 * TV + \theta_2 * radio \tag{3.1}$$

$$sales = \theta_0 + \theta_1 * TV + \theta_2 * radio + \theta_3 * newspaper \tag{3.2}$$

$$sales = \theta_0 + \theta_1 * TV + \theta_2 * radio + \theta_3 * TV * radio \tag{3.3}$$

3.1.1.2 Auto.csv

Figure 3.2 shows a correlation between horsepower and mpg of 0.78, which is generally considered a correlation above the absolute value of 0.4. The main aim of this paper is to analyse the relationship between the dependent variable mpg and the independent variable horsepower while excluding other variables from the dataset. To accomplish this, two variables were extracted from the entire dataset, horsepower and mpg, and were utilised as the independent and dependent variables, respectively. The objective of this paper is to investigate the correlation between these two variables.

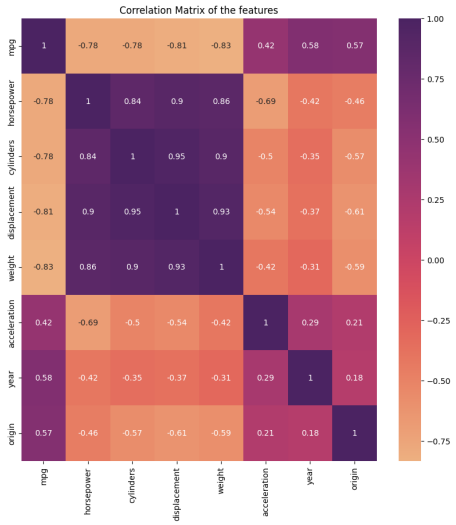


Figure 3.2: Heat map of auto dataset

3.2 Optimal splitting ratio

In the development of the model, a random sample was drawn from the dataset, with the remaining data earmarked for validation. In the context of determining the appropriate allocation of data for training and testing, the optimal train/test split ratio is calculated using the formula $\sqrt{p}$:1, where 'p' denotes the number of parameters requisite for an accurate representation of the linear regression model that encapsulates the underlying data (3.3). The calculation of the ideal training dataset size entails the substitution of the parameter count into the aforementioned formula. (1)

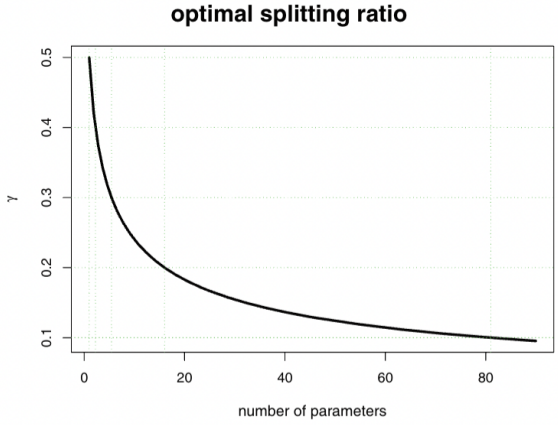


Figure 3.3: Optimal splitting ratio against the number of parameters

For example, for the two variables model, the optimal train size is 0.63, while for the three variables model, it is 0.66.

3.3 Polynomial Regression

Polynomial regression works in the same way as multiple linear regression, except that it preprocesses the data to create new combinations of variables before running the regression. However, the curve being fitted has a quadratic nature.

$$Y = \beta_0 * X^2 + \beta_1 * X + \beta_2$$
$$X_1 = X^2$$
$$X_2 = X$$
$$Y = \beta_0 * X_1 + \beta_1 * X_2 + \beta_2$$

Therefore, after transforming the training set to include polynomial terms up to a degree, the model was predicted by inheriting the previously implemented model. (See Appendix D and E)

3.4 Gradient Descent

Gradient descent is an algorithm for finding the minimum of the cost function.

The learning rate α in (3.4) and the gradient are multiplied and subtracted to keep moving the parameter towards the cost function's minimum. The learning rate is a constant, and its purpose is to control the amount of change in motion. It is usually between 0 and 1. The optimal model was determined by systematically assessing various combinations of learning rates and iterations. This was done by organising learning rate and iteration values into arrays. (see Appendix F)

The slope is proportional to the distance from the minima, meaning that if you are farther away from the minima, you can move more because the slope is steeper on the graph of the cost function, and if you are closer to the minima, you can move less because the slope is gentler. Subtract the product of the learning rate and the slope because the sign of the derivative is opposite to the direction of travel to the minima.

$$\theta_i = \theta_i - \alpha \frac{\partial J(\theta_0, \theta_i)}{\partial \theta_i} \tag{3.4}$$

The Gradient Descent algorithm is used to optimise multivariable linear regression by finding the optimal theta values that minimise the cost function. An algorithm for the Multivariate Regression Cost Function can be implemented by continuously updating the theta values after each model fits using the Gradient Descent algorithm, as shown in Equation 3.5.

$$\theta_j := \theta_j - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i \tag{3.5}$$

3.4.1 Normalised Dataset

Each of the three variables in the given Advertising dataset has a different range and density distribution. To address the issue of heavily influenced term creation by larger

values when multiplying two independent variables, standardization of the three independent and dependent variables using their mean and standard deviation was necessary. This is because the cost function graph becomes very scattered in three dimensions, and the gradient descent method takes a lot of time when dealing with variables that have a wide range.

(2)

3.4.2 Gradient Clipping

When performing gradient descent, it's crucial to prevent erratic movements of the gradient. These movements often occur when the gradient takes on substantial values. The gradient can then point in misleading directions, which can hinder convergence towards the global minimum.

A prominent solution to this challenge is the method of *gradient clipping* (3).

The core idea behind gradient clipping is to limit the magnitude of the gradient. Formally, given a gradient $\mathbf{g}$, the condition to be satisfied is:

$$||\mathbf{g}|| \leq \text{threshold} \tag{3.6}$$

The term *threshold* denotes a predefined maximum magnitude for the gradient. If the magnitude $||\mathbf{g}||$ surpasses this threshold, the gradient is scaled such that its magnitude is confined to the threshold, but its original direction is retained.

The use of gradient clipping in a gradient descent procedure effectively minimises the potential risks posed by disproportionately high gradient values, thus avoiding potential overflow scenarios.

3.5 Regression criteria

3.5.1 R squared value

In regression analysis, the value of the coefficient of determination R^2 is a measure of the magnitude of the explanatory power of the independent variables on the total variation of the dependent variable and ranges from 0 to 1. The closer the value of R^2 is to 1, the greater the explanatory power of the dependent variable by the independent variables and the better the fit of the regression equation.

It is imperative to scrutinise the adjusted R-squared value to ascertain whether it has been substantively enhanced if an additional independent variable has been integrated into the model.

$$R_a{}^2 = 1 - (n-1)\frac{(1-R^2)}{n-p-1} \tag{3.7}$$

In equation 3.7, where p means the number of independent variables, p is in the denominator, and as p increases, the R-squared value in the numerator also increases, somewhat offsetting the impact of the increase.

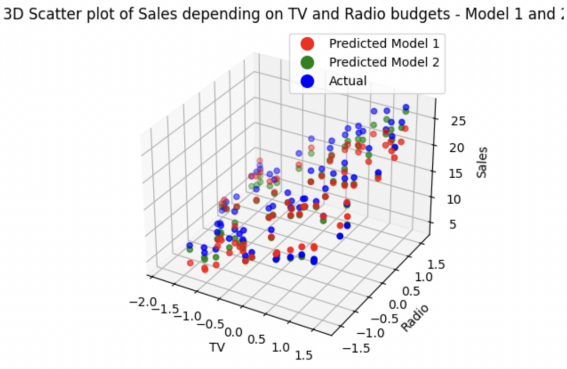
Chapter 4

Results & Discussion

4.1 Advertising dataset

4.1.1 Multivariate linear regression

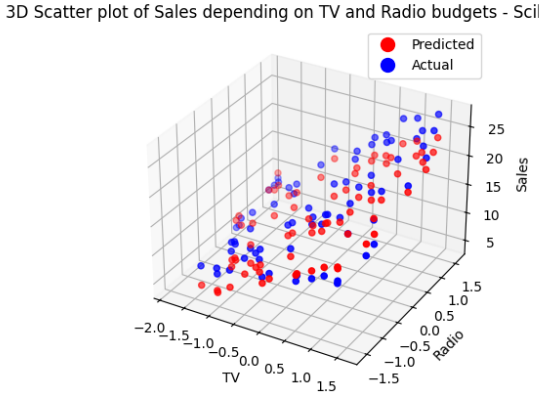
A regression model that considers the relationship between the first two independent variables and the dependent variable is shown in Model 1 (3.1). The first term on the left-hand side of the proposed model is constant, and the next two terms consider one of each independent variable. Model 2(3.3) includes an interaction term formed by multiplying the two independent variables. Considering all the interaction terms may adversely affect the efficiency of the model due to multicollinearity, the VIF and hitmap for all the terms were performed to derive the final multiple regression model function. The results are shown in Figure 4.1 The R-squared value for the multiple linear regression model without interactions is 0.816, but the R-squared value for the multiple regression model with interactions is 0.890, a significant improvement.



	Model1	Model2
R^2score	0.816	0.890
MSE	5.34	2.79
RMSE	2.31	1.67

Figure 4.1: Fitting the data with custom Model

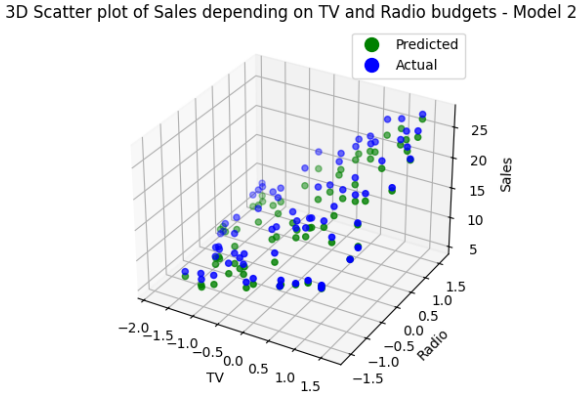
To fit a linear regression model using Sklearn, a three-step process was followed. Firstly, a linear regression class instance was created. Subsequently, the model was trained by invoking the fit() function. Finally, the predict() function was called to generate the predictions of the trained model.



	Sklearn
R^2score	0.83
MSE	4.38
RMSE	2.09

Figure 4.2: Fitting the data with sklearn Model

In Model2 equation 3.3, θ_3 represents the coefficient for this interaction term. The magnitude and sign of θ_3 will determine how the interaction between TV and radio affects the predicted values of sales. If θ_3 is positive, the interaction has a positive effect on the radio, and if it's negative, the interaction has a negative effect. The model yields a value of approximately 1.37 for θ_3, indicating the mutual influence of the two independent variables.



	Coefficient
θ_0	13.41
θ_1	3.92
θ_2	2.67
θ_3	1.37

Figure 4.3: TV+radio+TV*radio

4.1.2 TV + radio + newspaper

The model's predictive capability was extended through the introduction of an additional independent variable, 'newspaper,' into the existing framework, yielding the following regression equation:

$$sales = 13.5851 + 3.5680 * TV + 2.8118 * radio + 0.2626 * newspaper \quad (4.1)$$

It is noteworthy that the values of the coefficients in Equation (5.1) may exhibit slight variability in response to fluctuations in the training and test data. Nonetheless, the model can be deemed valid, particularly in consideration of the R-squared value. It is discernible from this model that 'TV' exerts the most substantial influence on the sales outcome, while 'newspaper' demonstrates a relatively minor effect. Despite the increment in the R-squared value, it is imperative to scrutinise the adjusted R-squared value to ascertain whether it has been substantively enhanced, as this additional independent variable has been integrated into the model.

The adjusted R-square value is 0.86, meaning that the predicted value calculated from the regression can explain 86% of the actual y-value.

To conduct a comprehensive analysis, advertising budgets for the product and data for three different media formats, namely TV, radio, and newspaper, were collected from 200 markets. These variables were utilised as independent factors in the analysis, enabling the identification of the impact of media formats on the product's performance in each market. This approach provided a deeper understanding of the relationship between advertising expenditure and media platforms. To account for the interaction between the independent variables, a multiple regression model was built that additionally considers a term consisting of the product of the two independent variables. The analysis showed that significant p-values were obtained for TV and radio, indicating that the null hypothesis about the association between these two variables and sales can be rejected. However, the null hypothesis for newspaper cannot be rejected, suggesting that there is insufficient evidence to conclude that this variable is significantly associated with sales.

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	2.9389	0.312	9.422	0.000	2.324	3.554
TV	0.0458	0.001	32.809	0.000	0.043	0.049
radio	0.1885	0.009	21.893	0.000	0.172	0.206
newspaper	−0.0010	0.006	−0.177	0.860	−0.013	0.011

Figure 4.4: P-value of variables

When it comes to advertising, all expenditures are positively correlated with sales. However, it is highly unlikely that newspaper ad expenditures have any correlation with sales. When regressing multiple variables, overlapping variances explaining sales are often not accounted for, resulting in less variance and a lower regression coefficient for newspapers.

The results of Model 3, which includes newspaper data as an independent variable in Model 1, are shown in Table 4.3.

	model	**sklearn**
RMSE	1.98	2.09
R^2	0.87	0.83

Table 4.1: TV+radio+newpaper data

4.1.3 Overall

Figure 4.5 shows the relationship between sales and ad budget, as plotted using the Spearmanr library. The relationship appears to be roughly linear, indicating that increasing the ad budget leads to a corresponding increase in sales. Furthermore, the parameter values for TV and radio are larger than those for newspapers, suggesting that increasing the budget for these two media channels is more effective in boosting sales than for newspapers.

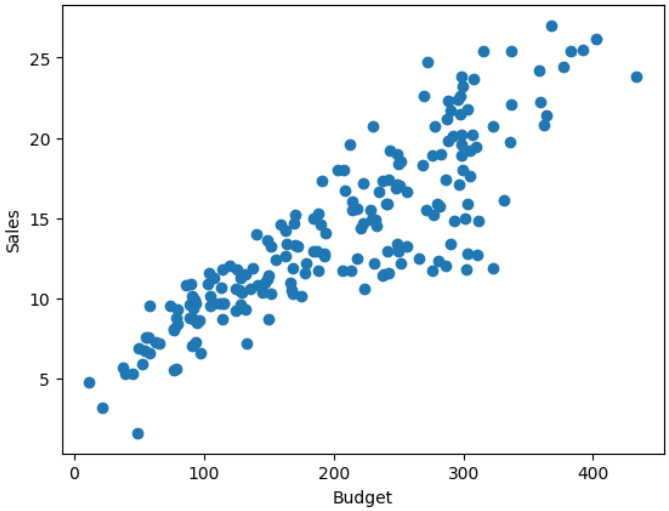
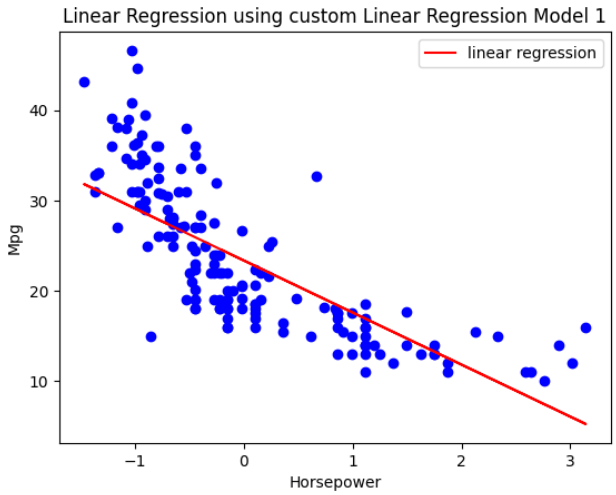


Figure 4.5: Correlation between Budget and Sales

After comparing the obtained model with the model obtained using the Sklearn library (Figure 4.2 and Table 4.1), it was found that the model produces similar results to the Sklearn function.

4.2 Auto dataset

Assuming a linear relationship between horsepower and mpg, the model was predicted: $Y = \beta_0 + \beta_1 X_1$.

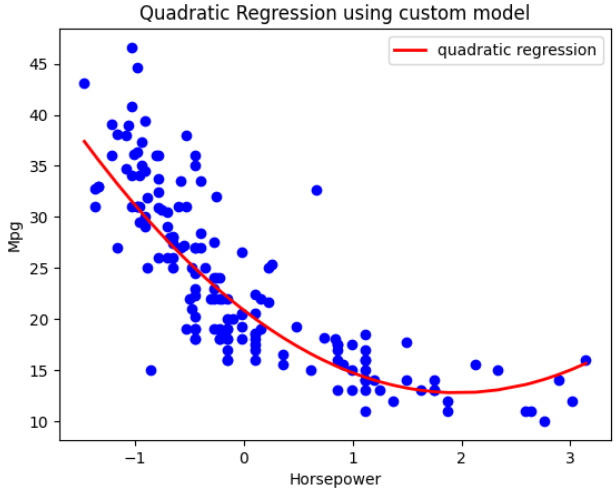


	Value
R^2score	0.59897
MSE	27.9184
RMSE	5.2837

Figure 4.6: Linear Regression

In **Fig 5.5**, the straight line is unable to capture the pattern in the data. This is categorised as an under-fitting. The R squared value is 0.59, which indicates that linear regression is inadequate to explain the model.

A higher-order equation can be generated by adding powers of the original features as new features. The linear model can then be transformed to $Y = \beta_0 X^2 + \beta_1 X + \beta_2$.

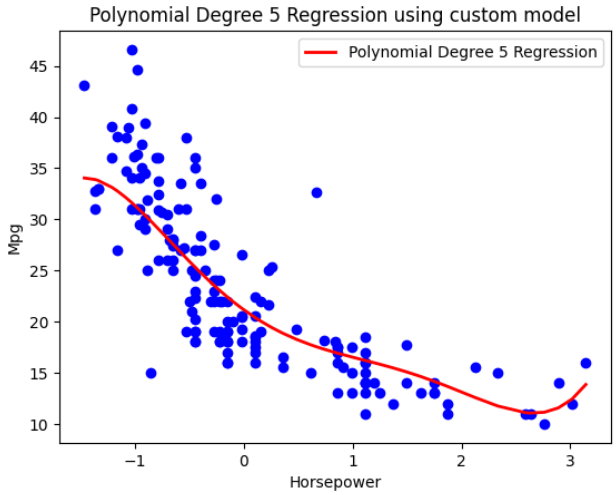


	Value
R^2score	0.71
MSE	20.17
RMSE	4.49

Figure 4.7: Quadratic Regression

Figure 5.6 illustrates the graph and numbers obtained through the quadratic regression model, which fits the data better than the linear curve. Comparing the R^2 values, an increase in the model's explained probability to 0.71 is evident. Additionally, the RMSE has decreased, and the R^2 score has increased when compared to the linear line.

When the degree value is set to 5, **Figure 5.7** shows that the model passes through more points and finds a better fit. However, it tends to overfit and performs worse than polynomial regression when data other than the training data is included.



	Value
R^2score	0.71
MSE	20
RMSE	4.47

Figure 4.8: Degree 5 Regression

Table 5.1 and **Table 5.2** present the results obtained from the implemented model and the sklearn library, respectively. It appears that the models produced using the library are generally better than those produced using the implemented model.

	linear	**quadratic**	**degree 5**
RMSE	5.28	4.49	4.47
R^2	0.59	0.71	0.71

Table 4.2: Polynomial regression results for custom model

	linear	**quadratic**	**degree 5**
RMSE	5.28	4.58	4.5
R^2	0.59	0.69	0.70

Table 4.3: Polynomial regression results for sklearn library

4.3 Conclusion

When working with complex data sets in data science, it can be challenging to pinpoint the most crucial variables. While machine learning techniques can leverage multiple variables to improve predictions, this can result in overfitting. As a result, it is often more effective to focus on the most significant variables. For instance, Multicollinearity was identified in the sales dataset, and only the essential variables were selected for modelling.

Furthermore, we utilised a basic gradient descent method to construct the model, which can be implemented in various ways, including mini-batch and stochastic. It's worth noting that initialising values to zero can significantly impact training time and results since the derivative is a crucial algorithm component. Once the slope value is determined, the algorithm adjusts the point's position by adding or subtracting it to move closer to the minimum.

References

1. Joseph VR. Optimal ratio for data splitting. Statistical Analysis and Data Mining: The ASA Data Science Journal. 2022;15(4). Available at: https://onlinelibrary.wiley.com/doi/full/10.1002/sam.11583. © 2022 The Author. Statistical Analysis and Data Mining published by Wiley Periodicals LLC.

2. Watt J, et al. Machine Learning Refined: Foundations, Algorithms, and Applications 2nd Edition. Cambridge University Press; March 12, 2020.

3. Bajaj A. Understanding Gradient Clipping (and How It Can Fix Exploding Gradients Problem); 2023. Available at: https://neptune.ai/blog/understanding-gradient-clipping-and-how-it-can-fix-exploding-gradients-problem. Accessed: 11 October 2023.

Appendix A

Documentation

1. Project tree

```
data/
    - Advertising.csv
    - Auto.csv
library/
    - LinearRegressionCustomModel1.py
    - LinearRegressionCustomModel2.py
    - PolynomialDegree5RegressionCustomModel.py
    - QuadraticRegressionCustomModel.py
    - utils.py
assessment_advertising-dataset.ipynb
assessment_auto-dataset.ipynb
requirements.txt
```

2. Getting Started

To run the program, follow these steps:

1. Create a virtual environment using `python3 -m venv venv`.
2. Activate the virtual environment using `source venv/bin/activate`.
3. Install the required dependencies using `pip3 install -r requirements.txt`.
4. Use the jupyter notebook `assessment_advertising-dataset.ipynb` to run the program on the advertising dataset.
5. Use the jupyter notebook `assessment_auto-dataset.ipynb` to run the program on the auto dataset.

3. Detailed Features of Classes and Functions

Classes

`LinearRegressionCustomModel1`: Class to perform linear regression with the gradient descent algorithm, using the model:

$$y = \theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \ldots + \theta_n \times x_n$$

`LinearRegressionCustomModel2`: Class to perform linear regression with the gradient descent algorithm, using the model:

$$y = \theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \theta_3 \times (x_1 \times x_2)$$

`QuadraticRegressionCustomModel`: Extends the LinearRegressionCustomModel1 Class to perform quadratic regression.

`PolynomialDegree5RegressionCustomModel`: Extends the LinearRegressionCustomModel1 Class to perform polynomial regression of degree 5.

Each class has the following methods:

- `fit(self, x_train, y_train)`: Method to fit the model to the training data.
- `predict(self, x_test)`: Method to predict the output for a given input.
- `metrics(self, x_test, y_test)`: Method to compute the model's metrics.

Functions in utils.py

`computeOptimalTrainSize(r)`: Function to compute the optimal training dataset size based on the number r of predictors.

`computeOptimalParameters(x_train, x_test, y_train, y_test, model_type)`: Function to compute the best hyperparameters (learning rate and number of steps).

`prepare_data(x, y, train_size)`: Function to prepare the data (split the data into training and testing sets, and standardise the data).

Appendix B

Linear Regression Custom Model 1 Source Code

```
import numpy as np
from sklearn import metrics


# Class to perform linear regression with gradient descent algorithm - Model 1
# y =  _0  +  _1  * x1 +  _2  * x2 + ... +  _n  * xn
class LinearRegressionCustomModel1:
    def __init__(self, learning_rate, iterations_nb):
        self.learning_rate = learning_rate  # Learning rate
        self.iterations_nb = iterations_nb  # Number of iterations
        self.theta = None

    # Method to fit the model to the training data
    def fit(self, x_train, y_train):
        # Add column of ones to x_train - Bias / Intercept
        x_train = np.column_stack((np.ones(x_train.shape[0]), x_train))

        # Number of training samples and number of features
        n, r = x_train.shape

        # Initialise theta with a vector of zeros
        self.theta = np.zeros(r)

        # Initialise variables for convergence check
        prev_cost = float("inf")  # Set initial previous cost to infinity
        tolerance = 1e-6  # Convergence tolerance
        gradient_threshold = 1e5  # Gradient clipping threshold

        # Gradient descent algorithm
        for _ in range(self.iterations_nb):
            # Compute predictions (y_pred = x_train * theta)
            y_pred = x_train.dot(self.theta)

            # Compute error (difference between predictions and actual values)
```

```
            error = y_pred - y_train

            # Compute gradient
            gradient = (2 / n) * error.dot(x_train)

            # Gradient clipping to prevent divergence (if gradient is too large)
            gradient = np.clip(gradient, -gradient_threshold, gradient_threshold)

            # Update parameters
            self.theta -= self.learning_rate * gradient

            # Compute cost (mean squared error)
            cost = (1 / n) * np.sum((y_pred - y_train) ** 2)

            # Convergence check
            if abs(prev_cost - cost) < tolerance:
                break

            # Update previous cost
            prev_cost = cost

        # Return None if training was not successful
        if np.isnan(cost):
            return None, float("inf")  # Return None for theta and infinity for cost

        return self.theta, cost

    # Method to predict the output for a given input
    def predict(self, x_test):
        # Add column of ones to x_test - Bias / Intercept
        x_test = np.column_stack((np.ones(x_test.shape[0]), x_test))

        # Return predictions with interaction term
        y_pred = x_test.dot(self.theta)  # Compute predictions
        assert (
            y_pred.shape[0] == x_test.shape[0]
        ), "Mismatch in prediction shape"  # Check if the shape of the prediction is correct

        return y_pred

    # Method to compute the metrics of the model
    def metrics(self, x_test, y_test):
        # Compute predictions
        y_pred = self.predict(x_test)

        # Compute metrics
        rSquare = metrics.r2_score(y_test, y_pred)  # R squared
        meanAbErr = metrics.mean_absolute_error(y_test, y_pred)  # Mean absolute error
        meanSqErr = metrics.mean_squared_error(y_test, y_pred)  # Mean squared error
```

```
84          rootMeanSqErr = np.sqrt(
85              metrics.mean_squared_error(y_test, y_pred)
86          )  # Root mean squared error
87
88          return rSquare, meanAbErr, meanSqErr, rootMeanSqErr
```

Appendix C

Linear Regression Custom Model 2 Source Code

```
import numpy as np
from sklearn import metrics


# Class to perform linear regression with gradient descent
    algorithm - Model 2
# y =  _0  +  _1  * x1 +  _2  * x2 +  _3  * (x1 * x2)
class LinearRegressionCustomModel2:
    def __init__(self, learning_rate, iterations_nb):
        self.learning_rate = learning_rate  # Learning rate
        self.iterations_nb = iterations_nb  # Number of iterations
        self.theta = None

    # Method to fit the model to the training data
    def fit(self, x_train, y_train):
        # Convert DataFrame to numpy array
        x_train = x_train.values

        # Add column of ones to x_train - Bias / Intercept
        interaction_term = (
            x_train[:, 0] * x_train[:, 1]
        )  # Interaction term between x1 and x2
        x_train = np.column_stack(
            (np.ones(x_train.shape[0]), x_train, interaction_term)
        )

        # Number of training samples and number of features
        n, r = x_train.shape

        # Initialise theta with a vector of zeros
        self.theta = np.zeros(r)

        # Gradient descent algorithm
        for _ in range(self.iterations_nb):
            # Compute predictions
            y_pred = x_train.dot(self.theta)

            # Compute error (difference between predictions and
```

```
    actual values)
38              error = y_pred - y_train
39
40              # Compute gradient
41              gradient = (2 / n) * error.dot(x_train)
42
43              # Update parameters
44              self.theta -= self.learning_rate * gradient
45
46              # Compute cost (mean squared error)
47              cost = (1 / n) * np.sum((y_pred - y_train) ** 2)
48
49          return self.theta, cost
50
51      # Method to predict the output for a given input
52      def predict(self, x_test):
53          # Add column of ones to x_test - Bias / Intercept
54          x_test = np.column_stack((np.ones(x_test.shape[0]), x_test)
    )
55
56          # Return predictions with interaction term
57          y_pred = (
58              self.theta[0]
59              + self.theta[1] * x_test[:, 1]
60              + self.theta[2] * x_test[:, 2]
61              + self.theta[3] * (x_test[:, 1] * x_test[:, 2])
62          )
63          assert (
64              y_pred.shape[0] == x_test.shape[0]
65          ), "Mismatch in prediction shape"  # Check if the shape of
    the prediction is correct
66
67          return y_pred
68
69      # Method to compute the metrics of the model
70      def metrics(self, x_test, y_test):
71          # Compute predictions
72          y_pred = self.predict(x_test)
73
74          # Compute metrics
75          rSquare = metrics.r2_score(y_test, y_pred)  # R squared
76          meanAbErr = metrics.mean_absolute_error(y_test, y_pred)  #
    Mean absolute error
77          meanSqErr = metrics.mean_squared_error(y_test, y_pred)  #
    Mean squared error
78          rootMeanSqErr = np.sqrt(
79              metrics.mean_squared_error(y_test, y_pred)
80          )  # Root mean squared error
81
82          return rSquare, meanAbErr, meanSqErr, rootMeanSqErr
```

Appendix D

Quadratic Regression Custom Model Source Code

```
from library.LinearRegressionCustomModel1 import
    LinearRegressionCustomModel1
import numpy as np


# Extend the LinearRegressionCustomModel1 class to implement a
    quadratic regression model
class QuadraticRegressionCustomModel(LinearRegressionCustomModel1):
    # Function to add quadratic terms to the dataset
    def __add_quadratic_terms(self, X):
        # Square each feature
        X_squared = np.square(X)

        # Concatenate the original features and their squared
    values
        return np.hstack((X, X_squared))

    def fit(self, x_train, y_train):
        # Transform x_train to include quadratic terms
        x_train_transformed = self.__add_quadratic_terms(x_train)

        # Call the fit method of the parent class (
    LinearRegressionCustomModel1)
        theta, cost = super().fit(x_train_transformed, y_train)

        return theta, cost

    def predict(self, x_test):
        # Transform x_test to include quadratic terms
        x_test_transformed = self.__add_quadratic_terms(x_test)

        # Call the predict method of the parent class
        return super().predict(x_test_transformed)
```

Appendix E

Degree 5 Regression Custom Model Source Code

```
from library.LinearRegressionCustomModel1 import
    LinearRegressionCustomModel1
import numpy as np


# Extend the LinearRegressionCustomModel1 class to implement a
    polynomial regression model of degree 5
class PolynomialDegree5RegressionCustomModel(
    LinearRegressionCustomModel1):
     # Add polynomial terms up to degree 5 to the dataset
     def __add_degree_5_terms(self, X):
         # Square each feature
         X_deg_2 = np.power(X, 2)
         X_deg_3 = np.power(X, 3)
         X_deg_4 = np.power(X, 4)
         X_deg_5 = np.power(X, 5)

         # Concatenate the original features and their polynomial
    values
         return np.hstack((X, X_deg_2, X_deg_3, X_deg_4, X_deg_5))

     def fit(self, x_train, y_train):
         # Transform x_train to include polynomial terms up to
    degree 5
         x_train_transformed = self.__add_degree_5_terms(x_train)

         # Call the fit method of the parent class (
    LinearRegressionCustomModel1)
         theta, cost = super().fit(x_train_transformed, y_train)

         return theta, cost

     def predict(self, x_test):
         # Transform x_test to include polynomial terms up to degree
     5
         x_test_transformed = self.__add_degree_5_terms(x_test)

         # Call the predict method of the parent class
```

```
32          return super().predict(x_test_transformed)
```

Appendix F

Utility Functions Source Code

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

from library.LinearRegressionCustomModel1 import
    LinearRegressionCustomModel1
from library.LinearRegressionCustomModel2 import
    LinearRegressionCustomModel2
from library.PolynomialDegree5RegressionCustomModel import (
    PolynomialDegree5RegressionCustomModel,
)
from library.QuadraticRegressionCustomModel import
    QuadraticRegressionCustomModel


# Function to compute the optimal train size
def computeOptimalTrainSize(r):
    # Compute optimal train size
    optimalTestSize = 1 / (np.sqrt(r) + 1)
    optimalTrainSize = 1 - optimalTestSize

    return optimalTrainSize


# Function to compute the optimal parameters depending on the model
    type
def computeOptimalParameters(x_train, x_test, y_train, y_test,
    model_type):
    # Set hyperparameters
    iterations_numbers = [
        200,
        300,
        400,
        500,
        1000,
        2000,
        3000,
        5000,
        7000,
        10000,
```

```
        20000,
        50000,
    ]  # Number of iterations
    learning_rates = [0.1, 0.01, 0.05, 0.001, 0.005, 0.0001,
0.0005]  # Learning rate

    # Initialise an empty array to store the results
    results = []

    # Set seed for reproducibility
    np.random.seed(0)

    # Computes linear regression for each hyperparameters
    for iterations_nb in iterations_numbers:
        for learning_rate in learning_rates:
            # Initialise model
            if model_type == "linear_2":
                # Linear Model 2
                model = LinearRegressionCustomModel2(learning_rate,
 iterations_nb)
            elif model_type == "quadratic":
                # Quadratic model
                model = QuadraticRegressionCustomModel(
learning_rate, iterations_nb)
            elif model_type == "degree_5":
                # Polynomial degree 5 model
                model = PolynomialDegree5RegressionCustomModel(
                    learning_rate, iterations_nb
                )
            else:
                # Model 1
                model = LinearRegressionCustomModel1(learning_rate,
 iterations_nb)
            # Fit model to training data
            theta, cost = model.fit(x_train, y_train)
            # Skip if training was not successful
            if theta is None:
                continue
            # Compute metrics
            rSquare, meanAbErr, meanSqErr, rootMeanSqErr = model.
metrics(x_test, y_test)
            # Save results
            results.append(
                {
                    "iterations_nb": iterations_nb,
                    "learning_rate": learning_rate,
                    "cost": cost,
                    "rSquare": rSquare,
                    "meanAbErr": meanAbErr,
                    "meanSqErr": meanSqErr,
                    "rootMeanSqErr": rootMeanSqErr,
                }
            )

    # Convert results to dataframe
    results = pd.DataFrame(results)
```

```
87
88      # Sort results by MSE
89      results = results.sort_values("meanSqErr", ascending=True)
90
91      # Return best hyperparameters
92      return (
93          int(results.iloc[0]["iterations_nb"]),
94          results.iloc[0]["learning_rate"],
95          results,
96      )
97
98
99  # Function to prepare data (split data into train and test sets,
        and standardise data)
100 def prepare_data(x, y, train_size):
101     # Split data into train and test sets
102     x_train, x_test, y_train, y_test = train_test_split(x, y,
        train_size=train_size)
103
104     # standardise data
105     x_train = (x_train - x_train.mean()) / x_train.std()  #
        standardise train set
106     x_test = (x_test - x_test.mean()) / x_test.std()  # standardise
         test set
107
108     #  Return train and test sets
109     return x_train, x_test, y_train, y_test
```