VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF APPLIED MATHEMATICS

PROBABILITY AND STATISTICS (MT2013)

Assignment (Semester 232)

Investigating the Utilization of Multiple Linear Regression for Predicting Thermal Design Power in Central Processing Units

Advisor: Dr. Nguyen Tien Dung
Students: Hoang Cong Minh – 2252474
Nguyen Van Tien – 2252811
Bui Tran Huy Vu – 2252915
Nguyen Hoang Dung – 2252109
Pham Ngoc Vuong – 2252923

Ho Chi Minh City - May 2, 2024

Contents

Abstract

This research examines the employment of *Multiple Linear Regression* (MLR) as a statistical modeling technique for analyzing and forecasting the *Thermal Design Power* (TDP) of *Central Processing Units* (CPUs). The primary objective is to deepen comprehension of MLR models and their utility in dataset examination. The approach entails developing an MLR model using chosen CPU feature parameters. The findings reveal a robust model fit, as the predicted TDP values closely correspond to the actual data. Overall, this research emphasizes the importance of MLR in statistical modeling and its applicability in predictive analysis for evaluating TDP.

1 Introduction

In computer hardware optimization, understanding and predicting *Thermal Design Power* (TDP) is crucial for efficient system design and performance management. This study leverages MLR to predict TDP using selected input parameters derived from CPU components' characteristics. TDP, representing a processor's thermal dissipation and power consumption capabilities, serves as a fundamental metric in system engineering and thermal management. By analyzing the relationships between various CPU attributes such as the maximum amount of power, the processor cores, memory controller, integrated graphics, and other components, this research aims to develop a robust predictive model that effectively navigates the challenges of multicollinearity within the dataset. Accurate TDP predictions facilitated by MLR not only optimize system efficiency but also inform decisions regarding cooling solutions and power allocation, thus contributing to enhanced performance and sustainability in CPU computing environments.

The given dataset [1] include two files as *All_GPUs.csv* and *Intel_CPUs.csv*. We select the second file because of the correlation between TDP and other independent variables. Therefore, it is possible for efficiency boost via the process of estimation of TDP relies on the given dataset.

[1] https://www.kaggle.com/datasets/iliassekkaf/computerparts

2 Theory

2.1 Multiple Linear Regression

Multiple regression is the statistical procedure to predict the values of a response (dependent) variable from a collection of predictor (independent) variable values (Sinharay, 2010). It is sometimes known simply as multiple regression, and it is an extension of linear regression.

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_p x_p + \varepsilon \tag{1}$$

Where:

- $\hat{y}$ is the predicted dependent variable (often referred to as the response or outcome variable).
- b_0 is the y-intercept (constant term).
- $b_1, b_2, \ldots, b_p$ are the coefficients for the independent variables $x_1, x_2, \ldots, x_p$ respectively.
- $x_1, x_2, \ldots, x_p$ represent the independent variables.
- ε is the error term.

Multiple regression is a type of regression where the dependent variable shows a linear relationship with two or more independent variables. It can also be non-linear, where the dependent and independent variables do not follow a straight line. Multiple linear regression is based on the following assumptions:

- ***A linear relationship between the dependent and independent variables***: The best way to check the linear relationships is to create scatter plots and then visually inspect the scatter plots for linearity. If the relationship displayed in the scatter plot is not linear, then the analyst will need to run a non-linear regression or transform the data using statistical software, such as SPSS[2].
- ***The independent variables are not highly correlated with each other***: The data should not show multicollinearity, which occurs when the independent variables (explanatory variables) are highly correlated. When independent variables show multicollinearity, there will be problems figuring out the specific variable that contributes to the variance in the dependent variable. The best method to test for the assumption is the *Variance Inflation Factor* (VIF) method. (More detail in section 2.4.
- ***The variance of the residuals is constant***: MLR assumes that the amount of error in the residuals is similar at each point of the linear model. This scenario is known as homoscedasticity. When analyzing the data, the analyst should plot the standardized residuals against the predicted values to determine if the points are distributed fairly across all the values of independent variables. To test the assumption, the data can be plotted on a scatter plot or by using statistical software to produce a scatter plot that includes the entire model.
- ***Independence of observation***: The model assumes that the observations should be independent of one another. Simply put, the model assumes that the values of residuals are independent. To test for this assumption, we use the Durbin-Watson statistic (Bartels & Goodhew, 1981). The test will show values from 0 to 4, where a value of 0 to 2 shows positive autocorrelation and values from 2 to 4 show negative autocorrelation. The mid-point, i.e., a value of 2, shows that there is no autocorrelation.
- *Multivariate normality*: Multivariate normality occurs when residuals are normally distributed. To test this assumption, look at how the values of residuals are distributed. It can also be tested using two main methods, i.e., a histogram with a superimposed normal curve or the Normal Probability Plot method.

[2]https://www.ibm.com/products/spss-statistics

2.2 Estimation

There are 3 primary methods to estimate for the different regression approaches explored:

- **Least Squares Estimation:** an approach that estimates parameters by minimizing the squared discrepancies between observed data, on the one hand, and their expected values on the other. The least-squares estimates of the intercept and slope in the simple linear regression model are:

$$b = \frac{S_{xy}}{S_{xx}} \tag{2}$$

$$a = \bar{y} - b\bar{x} \tag{3}$$

Where:

$$S_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{4}$$

$$S_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{5}$$

$$S_{yy} = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{6}$$

Advantages and Disadvantages of the Least Squares Method:

Pros	**Cons**
Easy to apply and understand	Only highlights relationship between two variables
Highlights the relationship between two variables	Doesn't account for outliers
Can be used to make predictions about future performance	May be skewed if data isn't evenly distributed

- **Maximum Likelihood Estimation:** an approach that maximizes the likelihood function to derive parameter estimates and is used for linear mixed effects models to handle correlated data.
- **Generalized Linear Models:** an approach that ultimately provides us with linear, logistic, Poisson, etc. regression models, often based on maximum likelihood estimation.

2.3 Correlation

This is a statistical measure that shows how two variables are linearly related, meaning they change together at a constant rate. The correlation coefficient, represented by the symbol r, ranges from -1 to +1. A correlation coefficient close to 0 indicates little or no relationship between the two variables. A correlation coefficient close to +1 or -1 means a strong positive or negative relationship between the two variables, respectively.

The Sample Correlation Coefficient r

The sample correlation coefficient for n pairs $(x_i, y_i) \dots (x_n, y_n)$

$$r = \frac{S_{xy}}{\sqrt{\sum (x_i - \bar{x})^2} \sqrt{\sum (y_i - \bar{y})^2}} = \frac{S_{xy}}{\sqrt{S_{xx}} \sqrt{S_{yy}}} \tag{7}$$

The Population Correlation Coefficient and Inferences About Correlation

$$\hat{p} = R = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum(X_i - \bar{X})^2}\sqrt{\sum(Y_i - \bar{Y})^2}} \tag{8}$$

Testing for the Absence of Correlation
When $H_0; p = 0$ is true, the test statistic

$$T = \frac{R\sqrt{n-2}}{\sqrt{1-R^2}} \tag{9}$$

has a t distribution with $n-2$ df.

Alternative Hypothesis	Rejection Region for Level α Test
$H_1 : p \geq 0$	$t \geq t_{\alpha,n-2}$
$H_1 : p \leq 0$	$t \leq -t_{\alpha,n-2}$
$H_1 : p \neq 0$	either $t \geq t_{\alpha/2,n-2}$ or $t \leq -t_{\alpha/2,n-2}$

Other Inferences Concerning p
When $X_1, Y_1, \ldots, X_n, Y_n$ is a sample from a bivariate normal distribution, the random variable

$$V = \frac{1}{2}\ln\frac{1+R}{1-R} \tag{10}$$

has approximately a normal distribution with mean and variance

$$\mu_v = \frac{1}{2}\ln\frac{1+p}{1-p} \qquad \sigma_v^2 = \frac{1}{n-3} \tag{11}$$

The test statistics for testing $H_0 : p = p_0$ is

$$Z = \frac{V - \frac{1}{2}\ln\frac{1+p_0}{1-p_0}}{1-\sqrt{n-3}} \tag{12}$$

Alternative Hypothesis	Rejection Region for Level α Test
$H_1 : p \geq p_0$	$z \geq z_\alpha$
$H_1 : p \leq p_0$	$z \leq -z_\alpha$
$H_1 : p \neq p_0$	either $z \geq z_{\alpha/2}$ or $z \leq -z_{\alpha/2}$

2.4 Multicollinearity

Multicollinearity, also called collinearity (Goldstein, 1993) or ill-conditioning (Chatterjee & Hadi, 2006), is a statistical phenomenon that occurs when two or more independent variables in a regression model are highly correlated with each other. (Bhandari, 2024). This phenomenon is a common problem in a regression model when the estimated regression coefficients are unstable and difficult to interpret in the presence of multicollinearity. In the dataset, there could exist some problems from the construction process caused by poorly designed experiments, highly observational data, or the inability to manipulate the data. Moreover, this issue also occurs when a variable is created depending on other variables or being indicated in the dataset and also some other issue from the dataset. When collinearity exists, this can result in large and unpredictable estimated regression coefficients, leading to unreliable conclusions about how much influence the independent variables have on the outcome value assessment. So, to eliminate this issue, identifying collinearity is a critical step in constructing a multiple linear regression model.
A metric known as the **Variance Inflation Factor** (VIF) was introduced as an approach for detecting

collinearity, which is available to measure the correlation between the explanatory variables in a regression model and can be used to detect multicollinearity of the entire data frame.
The VIF is denoted as:

$$VIF = \frac{1}{1-R^2} \tag{13}$$

Where: R^2 is R-squared. (Which had been mentioned in the previous subsections.)
The validation conclusion is based on:

- $VIF = 1$: no correlation
- $1 < VIF \leq 5$: there are correlations but not too high to be eliminated
- $5 < VIF \leq 10$: high multicollinearity between this independent variable and the others

In the VIF process, we looking for the result should be between $(1,5]$ with the meaning of enough correlation between variables to ensure the modeling conclusion is acceptable.

3 Cleaning data

3.1 Import library

```
install.packages("pastecs") # tidyr library
install.packages("DescTools") # Mode function
install.packages("zoo") #function "na.locf"
install.packages("corrplot")

library(tidyverse)
library(zoo)
library(DescTools)
library(pastecs)
library(tidyr)
library(dplyr)
library(corrplot)
install.packages("corrplot")
```

Note: import necessary libraries

3.2 Data input

```
# Set the URL of the dataset you want to download
url <- "https://huggingface.co/datasets/vantien06/PSAssignment/resolve/main/Intel_CPUs.
    csv?download=true"

# Set the destination file path where you want to save the dataset
directory_name = "data"
dir.create(directory_name)

# Check if the directory was created successfully
destination <- "./data/Intel_CPUs.csv"

# Download the dataset
download.file(url, destination)
```

3.3 Clean the data

```
Intel_CPUs<-read.csv("./data/Intel_CPUs.csv", na.strings = c("","N/A"))
#view(Intel_CPUs)
head(Intel_CPUs,10)
```

Explain:

- **Data Importation:** Load the '*Intel_CPUs.csv*' file, treating empty strings and *"N/A"* as missing values to maintain data integrity.
- **Action:** Display the first 10 rows of the dataset for a preliminary examination of the data structure.

Output:

	Product_Collection	Vertical_Segment	Processor_Number	Status	Launch_Date	Lithography	Recommended_Customer_Price	nb_of_Cores	nb_of_Threads	Processor_Base_Frequency	…
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<int>	<int>	<chr>	…
1	7th Generation Intel® Core™ i7 Processors	Mobile	i7-7Y75	Launched	Q3'16	14 nm	$393.00	2	4	1.30 GHz	···
2	8th Generation Intel® Core™ i5 Processors	Mobile	i5-8250U	Launched	Q3'17	14 nm	$297.00	4	8	1.60 GHz	···
3	8th Generation Intel® Core™ i7 Processors	Mobile	i7-8550U	Launched	Q3'17	14 nm	$409.00	4	8	1.80 GHz	···
4	Intel® Core™ X-series Processors	Desktop	i7-3820	End of Life	Q1'12	32 nm	$305.00	4	8	3.60 GHz	···
5	7th Generation Intel® Core™ i5 Processors	Mobile	i5-7Y57	Launched	Q1'17	14 nm	$281.00	2	4	1.20 GHz	···
6	Intel® Celeron® Processor 3000 Series	Mobile	3205U	Launched	Q1'15	14 nm	$107.00	2	2	1.50 GHz	···
7	Intel® Celeron® Processor N Series	Mobile	N2805	Launched	Q3'13	22 nm	NA	2	2	1.46 GHz	···
8	Intel® Celeron® Processor J Series	Desktop	J1750	Launched	Q3'13	22 nm	NA	2	2	2.41 GHz	···
9	Intel® Celeron® Processor G Series	Desktop	G1610	Launched	Q1'13	22 nm	$42.00	2	2	2.60 GHz	···
10	Legacy Intel® Pentium® Processor	Mobile	518	End of Interactive Support	NA	90 nm	NA	1	NA	2.80 GHz	···

Figure 1: Display the first 10 rows of Intel_CPUs.csv

```r
#select specific column
data<-Intel_CPUs[,c(1,2,4,6,8,10,12,14,36,40)]
# Calculate the missing ratio of each data column and represent it graphically
emptycellrate <- (colMeans(is.na(Intel_CPUs)))*100
mrdf <- data.frame(Category = names(emptycellrate), percentage = emptycellrate)

# Draw a column chart
ggplot(mrdf, aes(x = Category, y = percentage)) +
geom_bar(stat = "identity", fill = "lightblue") +
theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
ggtitle("Empty-Cell Ratio of the data")
```

Explain:

- **Data Selection:** Select and retain columns 1, 2, 4, 6, 8, 10, 12, 14, 36, and 40 for focused analysis on key CPU attributes.
- **Calculate Missing Value Ratios:** Calculate the percentage of missing (NA) values in each column of the $Intel\_CPUs$ data frame. This is achieved by using the $colMeans(is.na(Intel\_CPUs))$ function, which checks for NA values, computes the mean of these checks (the proportion of NAs) across each column, and multiplies the result by 100 to convert it to a percentage.
- **Create Data Frame for Visualization:** Create a new data frame named $mrdf$ containing two columns: Category, which holds the names of the data columns from $Intel\_CPUs$, and percentage, which contains the corresponding missing value percentages computed in the previous step. This data frame organizes the missing data information in a format suitable for visualization.
- **Draw a Column Chart:** Use the $ggplot2$ package to create a column (bar) chart. The chart plots each column of the Intel_CPUs dataset on the $x-axis$ and the percentage of missing values on the $y-axis$.

Output:
The bar chart in figure 2 visualizes the percentage of missing values for each column in the Intel CPU

dataset. The purpose of this visualization is to easily identify and quantify the completeness of data across various attributes. We decided to focus only on columns with a missing data ratio of less than 25%. This threshold ensures that the data used in the analysis is sufficiently complete, thereby enhancing the reliability of the insights generated. Columns with higher missing data ratios are excluded to prevent biases or inaccuracies due to insufficient information.
In conclusion, we have 10 main variables: *Product_Collection*, *Vertical_Segment*, *Status*, *Lithography*, *nb_of_Cores*, *Processor_Base_Frequency*, *Cache*, *TDP*, *T*, *Instruction_Set*.

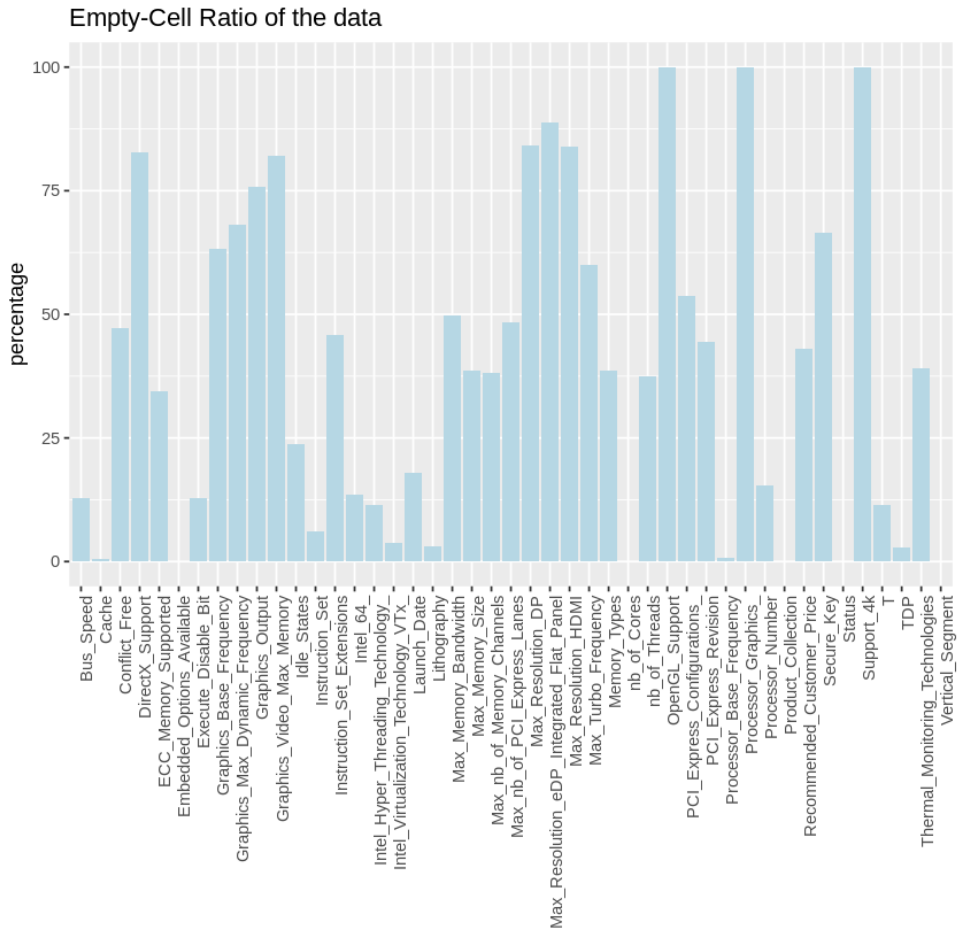

Figure 2: Percentage of missing values

In this cell, we check and count the number of missing data:

```
    # Check for missing data
print(apply(is.na(data),2,sum))
head(data)
write.csv(data, "./data/filter1.csv", row.names = TRUE)
```

Output:

```
Product_Collection        Vertical_Segment                  Status
                 0                       0                       0
       Lithography             nb_of_Cores Processor_Base_Frequency
                71                       0                      18
           Cache                       TDP                       T
                12                      67                     260
 Instruction_Set
               141
```

Figure 3: Count N/A value

```r
Cache_Size_Clean <- function(size){
  if(grepl('K',size)){
    return (as.double(gsub(" K","",size)) /1024)
  }
  else{
    return (as.double(gsub(" M","",size)))
  }
}
# separate data into 2 columns including cache type and its size
data <- separate(data,Cache,into = c("Cache_Size","Cache_Type"),sep="B")
# process the string and return it to real number type
data$Cache_Size <- sapply(data$Cache_Size,Cache_Size_Clean)
data$Cache_Size <- log(data$Cache_Size)
# because of the separate command, normal types are not included
data$Cache_Type <- ifelse(data$Cache_Type == "", "Normal", sub(" ","",data$Cache_Type))
```

Explain:

- Define Cache_Size_Clean Function: It converts cache size values from strings with 'K' (kilobytes) or 'M' (megabytes) to a uniform size in megabytes. Kilobyte values are divided by 1024 to convert them to megabytes.
- Split Cache Data: Uses the separate function from the *tidyr* package to divide the 'Cache' column into 'Cache_Size' and 'Cache_Type', where the separation is based on the 'B' character at the end of the size notation.
- Clean and Convert Cache Size: Applies the Cache_Size_Clean function to the 'Cache_Size' column using *sapply* to convert all values to numeric megabytes and then takes the logarithm of these values, likely for normalization purposes.
- Correct Cache Type: After separation, the 'Cache_Type' might have empty strings; these are replaced with "Normal". Any remaining space characters in 'Cache_Type' are removed with sub. This standardizes the cache type entries.

For example:

Cache	Cache_Size	Cache_Type
<chr>	<dbl>	<chr>
4 MB SmartCache	1.3862944	SmartCache
6 MB SmartCache	1.7917595	SmartCache
8 MB SmartCache	2.0794415	SmartCache

Figure 4: Cache before and after cleaning data

```r
data$Instruction_Set <- na.fill(data$Instruction_Set,"64")
data$Instruction_Set<- as.numeric(gsub("-bit", "", data$Instruction_Set))
data$Instruction_Set <- na.fill(data$Instruction_Set,"64")
colnames(data)[11] <- "Instruction_Set_Bit"
head(data)
```

Explain:

- **Fill Missing Values:** Replaces any missing values in the Instruction_Set column with "64", assuming a default of 64-bit.
- **Convert to Numeric:** Removes "-bit" from the *Instruction_Set* values, converting them to numeric.
- **Ensure Completeness:** Re-fills any new NA values post-conversion with "64".
- **Rename Column:** Changes the name of the column to *Instruction_Set_Bit* to reflect the data type.

For example:

Instruction_Set	Instruction_Set_Bit
<chr>	<dbl>
64-bit	64
64-bit	64

Figure 5: Instruction Set before and after cleaning data

```r
data$Lithography<- as.numeric(gsub("nm", "", data$Lithography))
colnames(data)[4] <- "Lithography_nm"
Ex<-mean(data$Lithography_nm, na.rm = TRUE)
data$Lithography_nm[is.na(data$Lithography_nm)] <- Ex
head(data)
```

Explain:

- **Convert to Numeric:** Removes the "nm" suffix from the *Lithography* values and converts the results to numeric data type.
- **Rename Column:** Changes the column name from *Lithography* to *Lithography_nm* to clearly indicate the unit of measurement.
- **Calculate Mean:** Computes the mean of the *Lithography_nm* values, ignoring missing values $na.rm = TRUE$.
- **Replace Missing Values:** Fills any missing values in *Lithography_nm* with the calculated mean.

For example:

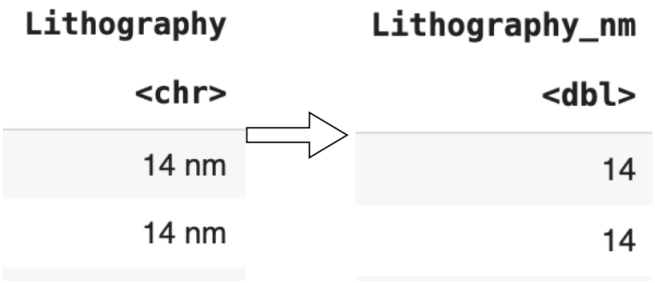

Figure 6: Lithography before and after cleaning data

```
data$TDP<- as.numeric(gsub("W", "", data$TDP))
colnames(data)[9] <- "TDP_W"
Ex <- mean(data$TDP_W, na.rm = TRUE)
data$TDP_W[is.na(data$TDP_W)] <- Ex
head(data)
```

Explain:

- **Convert to Numeric:** Strips the "W" (Watt) suffix from the *TDP* values and converts them to numeric format.
- **Rename Column:** Renames the column to *TDP_W* to emphasize that the values are in watts.
- **Calculate Mean:** Computes the average of the *TDP_W* column, excluding any missing values.
- **Replace Missing Values:** Substitutes any missing values in *TDP_W* with the calculated mean.

For example:

Figure 7: *TDP* value before and after cleaning data

```
base_frequency <-function(f){
  if (grepl(" GHz",f)) {
    return (as.double(gsub(" GHz","",f))*1000)
  }
  return (as.double(gsub(" MHz","",f)))
}
data$Processor_Base_Frequency <-as.integer( sapply(data$Processor_Base_Frequency,base_
    frequency))
```

```r
data$Processor_Base_Frequency <- as.numeric(gsub("GHz", "", data$Processor_Base_
    Frequency))
colnames(data)[6] <- "Processor_Base_Frequency_MHz"
Ex <- mean(data$Processor_Base_Frequency_MHz, na.rm = TRUE)
data$Processor_Base_Frequency_MHz[is.na(data$Processor_Base_Frequency_MHz)] <- Ex
head(data)
```

Explain:

- **Define Conversion Function:** A function (*base_frequency*) converts processor base frequency from GHz to MHz if needed, and ensures all frequency values are in MHz.
- **Apply Conversion:** Applies this function across the *Processor_Base_Frequency* column, converting all entries to integer MHz.
- **Clean Up Redundant Code:** Attempts to further strip "GHz" text and convert to numeric, which seems redundant and might be unnecessary.
- **Rename Column:** Updates the column name to *Processor_Base_Frequency_MHz* to clearly indicate the units.
- **Handle Missing Data:** Calculates the mean frequency, excluding missing values, and fills any missing values with this mean.

For example

Processor_Base_Frequency	Processor_Base_Frequency_MHz
<chr>	<dbl>
1.30 GHz	1300
1.60 GHz	1600

Figure 8: Processor Base Frequency before and after cleaning data

```r
# Convert T column to numeric after removing " C "
data$T <- as.numeric(gsub(" C ", "", data$T))

# Rename the column to "T"
colnames(data)[which(colnames(data) == "T")] <- "T_Celsius"

# Impute missing values in T_Celsius column with mean
mean_T <- mean(data$T_Celsius, na.rm = TRUE)
data$T_Celsius[is.na(data$T_Celsius)] <- mean_T

# Print count of missing values in each column
print(apply(is.na(data), 2, sum))

# Display first few rows of the dataset
head(data)
```

Explain:

- **Convert Temperature to Numeric:** The data T column's temperature values, originally with a "°C" suffix, are stripped of this text and converted to numeric format.
- **Rename Column:** Updates the column name from T to $T\_Celsius$ for clarity, indicating that the temperature values are in degrees Celsius.
- **Impute Missing Values:** Calculates the mean of the $T\_Celsius$ column (excluding missing values) and fills in any missing values with this mean to ensure a complete dataset.
- **Count Missing Values:** Executes a function to count and print the number of missing values in each column of the dataset

For example:

T	T_Celsius
<chr>	<dbl>
100°C	100.0
100°C	100.0

Figure 9: Temperature of the CPU before and after process

After these steps, we successfully solve missing data

```
Product_Collection            Vertical_Segment
                 0                           0
            Status                Lithography_nm
                 0                           0
      nb_of_Cores Processor_Base_Frequency_MHz
                 0                           0
        Cache_Size                  Cache_Type
                 0                           0
             TDP_W                    T_Celsius
                 0                           0
Instruction_Set_Bit
                 0
```

Figure 10: The number of missing values of each column after cleaning

3.4 Summary data

```
summary(data)
```

```
Product_Collection Vertical_Segment      Status          Lithography_nm
Length:2271        Length:2271        Length:2271        Min.   : 14.00
Class :character   Class :character   Class :character   1st Qu.: 22.00
Mode  :character   Mode  :character   Mode  :character   Median : 32.00
                                                         Mean   : 48.99
                                                         3rd Qu.: 65.00
                                                         Max.   :250.00
 nb_of_Cores     Processor_Base_Frequency_MHz   Cache_Size
Min.   : 1.000   Min.   :  32                 Min.   :-4.8520
1st Qu.: 2.000   1st Qu.:1660                 1st Qu.: 0.6931
Median : 2.000   Median :2260                 Median : 1.0986
Mean   : 4.083   Mean   :2233                 Mean   : 1.1699
3rd Qu.: 4.000   3rd Qu.:2800                 3rd Qu.: 2.0794
Max.   :72.000   Max.   :4300                 Max.   : 4.0943
 Cache_Type            TDP_W          T_Celsius      Instruction_Set_Bit
Length:2271        Min.   :  0.65   Min.   : 53.90   Min.   :32.00
Class :character   1st Qu.: 26.40   1st Qu.: 72.00   1st Qu.:64.00
Mode  :character   Median : 51.00   Median : 83.25   Median :64.00
                   Mean   : 60.29   Mean   : 83.25   Mean   :58.11
                   3rd Qu.: 84.50   3rd Qu.:100.00   3rd Qu.:64.00
                   Max.   :300.00   Max.   :110.00   Max.   :64.00
```

Figure 11: Summary data

The table in Figure 11 provides a detailed summary of various attributes across a dataset of *CPUs*, featuring statistical measures for each attribute such as minimum (Min.), first quartile (1st Qu.), median, mean, third quartile (3rd Qu.), and maximum (Max.). Here's a breakdown of each attribute's statistical summary:

- **Product_Collection, Vertical_Segment, Status:** These fields are categorical (character class) and cover 2271 entries each, indicating the dataset's size and the categorical nature of these attributes.
- **Lithography_nm:**
  - Range: 14 nm to 250 nm, indicating the variety of manufacturing processes used for the CPUs, from very fine (advanced) to less fine (older technologies).
  - Median of 32 nm and a mean near 49 nm, suggesting a concentration of newer technology CPUs but with a broad spread.
- **nb_of_Cores:**
  - Range: 1 to 72 cores, showing CPUs designed for a variety of uses, from basic to high-performance computing needs.
  - The mean of about 4 cores, with a median at 2, suggests that most CPUs are designed with fewer cores, likely for consumer-grade computers.
- **The remaining variables are explained in the same way**

4 Descriptive Statistic

The accompanying histograms elucidate the distribution of various factors that influence CPU performance.

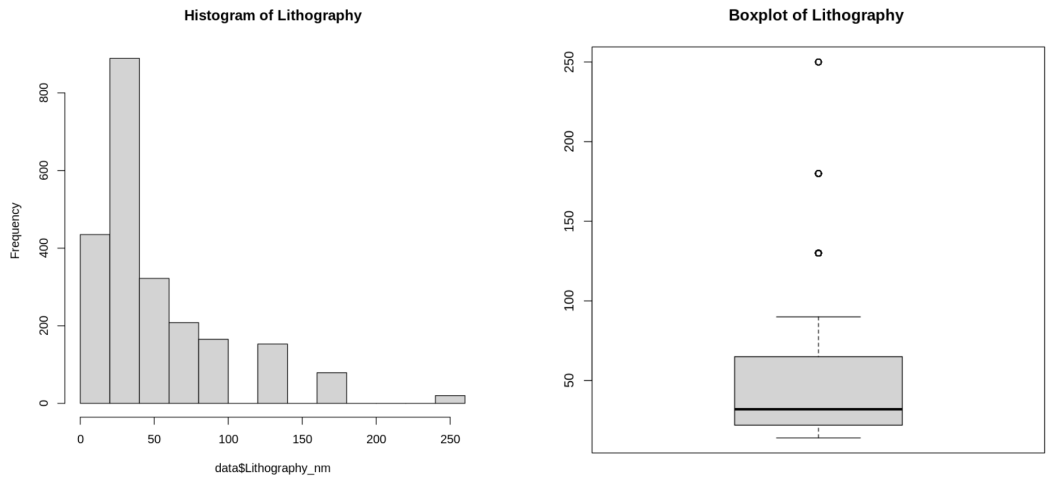

Figure 12: Diagram describe the distribution of Lithography

Lithography: figure 12 illustrates the distribution of lithography sizes in CPU manufacturing, captured in both a histogram and a boxplot. The histogram shows a dominant peak at 25 nm, signifying that this lithography size is the most frequently used, underscoring a strong industry preference for highly efficient and advanced manufacturing technology. Most CPUs are manufactured with lithography sizes below 100 nm, with the highest frequency observed at 25 nm. The boxplot supports this, placing the median around 50 nm and highlighting outliers above 150 nm, which suggests occasional applications of older, larger technologies. Together, these visualizations emphasize the industry shift towards smaller lithography sizes to optimize CPU performance and efficiency.

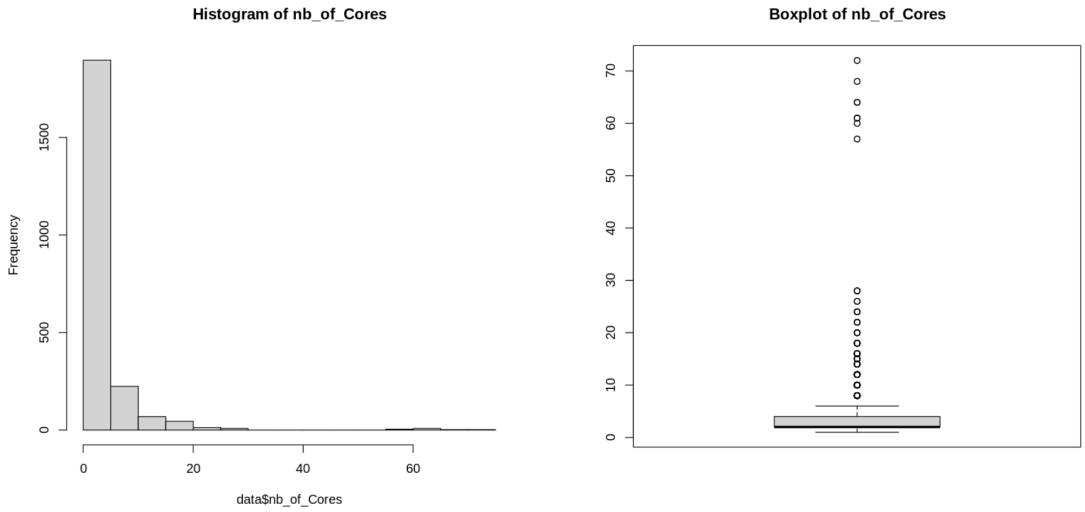

Figure 13: Diagram describe the distribution of the Number of Cores

Number of cores: The histogram and boxplot in figure 13 illustrate the distribution of CPU core counts in your dataset. Most CPUs have fewer than 10 cores, as shown by the histogram's peak at lower counts, indicating commonality in consumer-grade processors. The boxplot reinforces this with a low median and interquartile range but also highlights numerous high-core outliers up to 70 cores, reflecting specialized high-performance CPUs for tasks like server management or intensive data processing. Together, these visuals underscore a market with diverse CPU designs catering to both general and advanced computing needs.

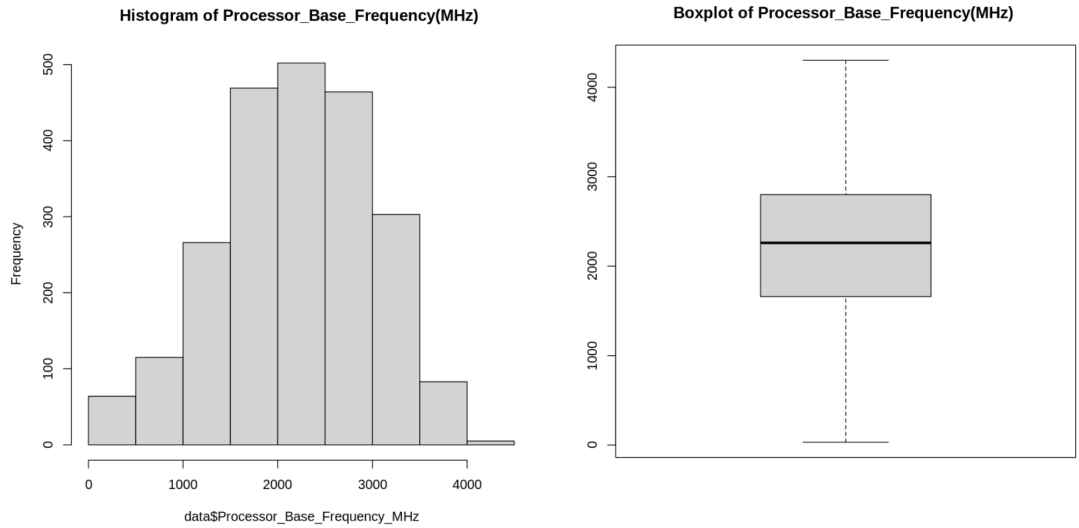

Figure 14: Diagram describe the distribution of Processor Base Frequency

Processor Base Frequency: figure 14 shows the distribution of processor base frequencies in MHz,

depicted through a histogram and a boxplot. The histogram peaks around 2000 MHz, indicating this is the most common base frequency for CPUs, suggesting standardization in processor speeds for general use. The boxplot confirms the central frequency range and highlights outliers above 2500 MHz, representing high-performance processors for specialized applications. This pattern underscores a trend where most CPUs are optimized for everyday tasks, while a minority caters to high-demand computing needs.

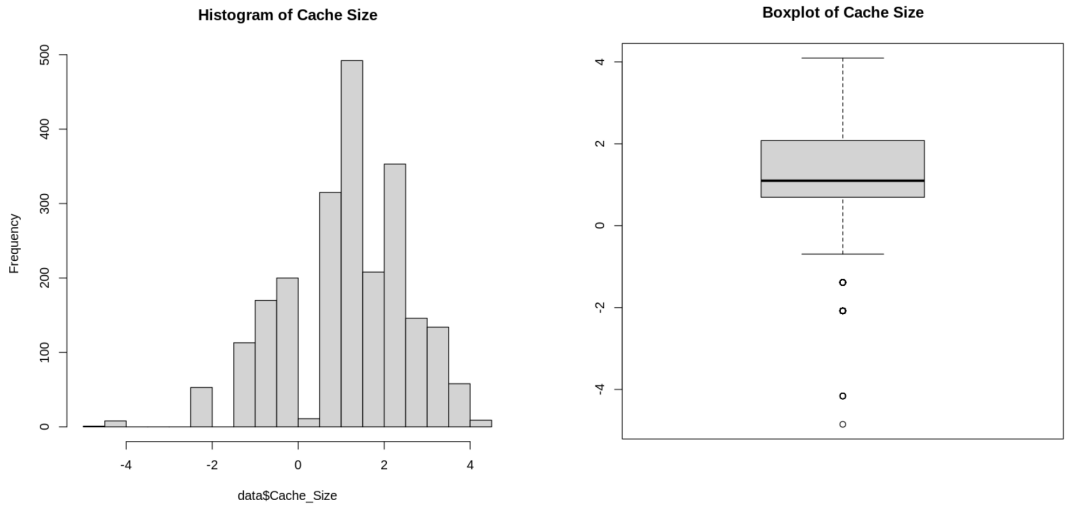

Figure 15: Diagram describe the distribution of Cache

Cache Size: Figure 15 illustrate the distribution of cache sizes in CPUs, appearing to use a standardized or transformed scale from -4 to 4. The histogram shows a peak around 1, indicating a common cache size standard among the CPUs, with frequencies declining towards the extremes, suggesting that most CPUs fall within a middle range of cache sizes. The boxplot corroborates this central clustering, showing a median near 1 and an interquartile range tightly centered around this median, highlighting consistency in cache size across most CPUs. Outliers present above 2 and below -2 in the boxplot reveal that there are exceptions with significantly smaller or larger caches, underscoring variability despite the general trend towards uniformity.

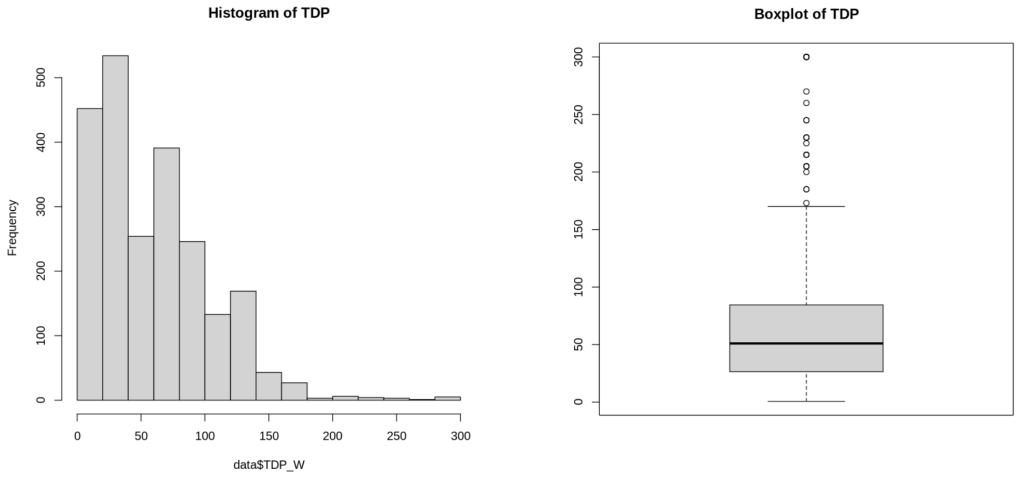

Figure 16: Diagram describe the distribution of TDP

Thermal Design Power (TDP): Figure 16 illustrates the distribution of TDP for CPUs, captured in a histogram and a boxplot. The histogram displays a distribution skewed towards lower TDP values, with a peak around 25 watts, suggesting that most CPUs are designed for moderate power consumption. The frequency declines as TDP increases, indicating fewer high-power CPUs. The boxplot supports this observation, showing a median around 50 watts and a range primarily between 25 to 100 watts, with outliers extending up to around 300 watts. These outliers represent CPUs with exceptionally high power requirements, likely tailored for high-performance or specialized tasks.

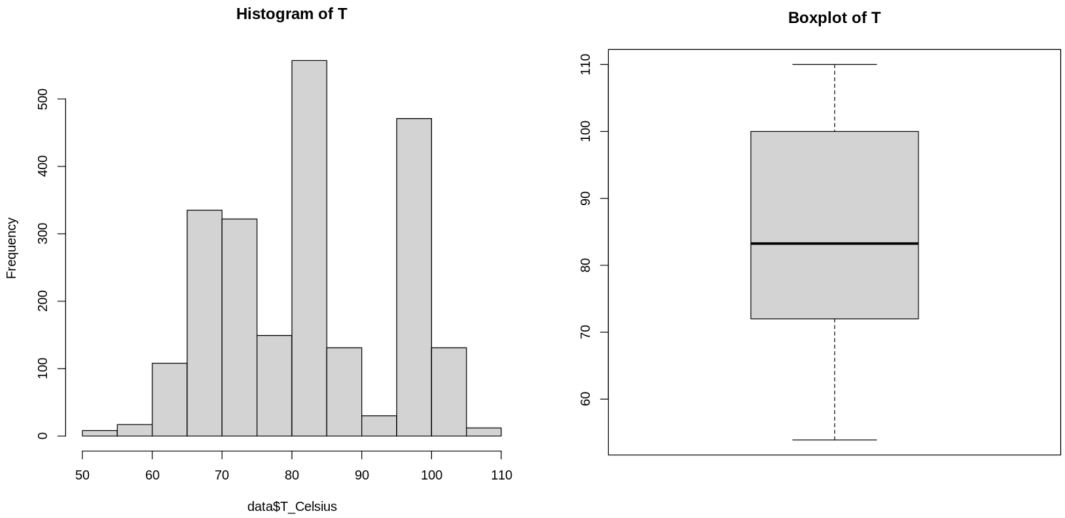

Figure 17: Diagram describe the distribution of T

Temperature: Figure 17 includes a histogram and a boxplot detailing the distribution of CPU tem-

peratures in Celsius. The histogram shows peaks at around 80°C, suggesting these are the most typical operating temperatures for CPUs in this dataset. The spread of temperatures varies from approximately 60°C to over 100°C, with the majority clustered between 80°C and 90°C. The boxplot supports this, showing a median temperature close to 85°C and most values compactly arranged around this center, indicative of consistency in CPU temperatures. Outliers above 100°C point to some CPUs operating at significantly higher temperatures. These charts together highlight that while most CPUs operate within a standard temperature range, a few reach unusually high temperatures.

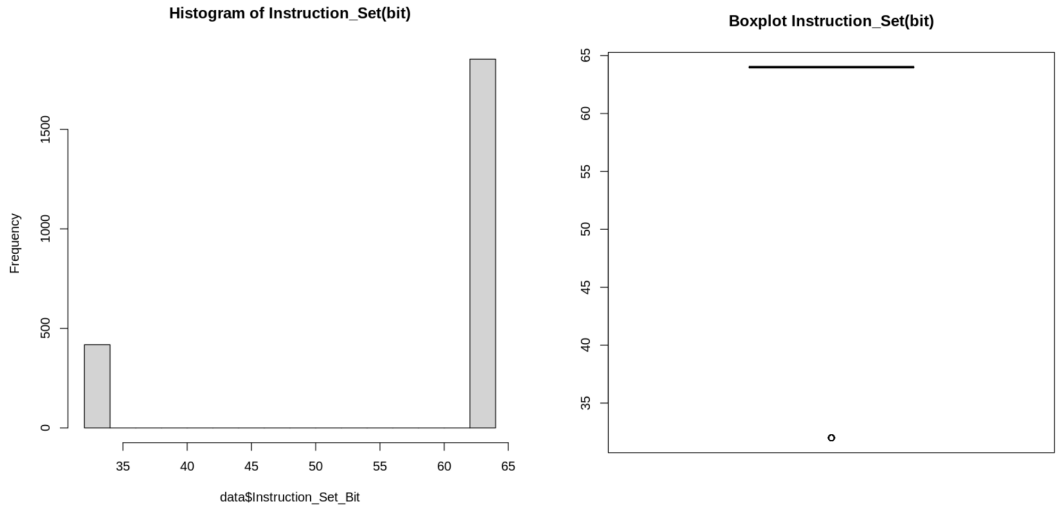

Figure 18: Diagram describe the distribution of Instruction

Instruction: Figure 18 features both a histogram and a boxplot illustrating the distribution of instruction set sizes (in bits) for CPUs. The histogram reveals a dominant presence of 64-bit instruction sets, with a negligible number of entries at 32 bits, emphasizing the industry's overwhelming preference for 64-bit architectures due to their enhanced performance and capability. The boxplot supports this, showing no spread within the interquartile range and illustrating that almost all data points are at 64 bits, with a single outlier at 32 bits. This uniformity in the histogram and boxplot underscores the standardization towards 64-bit technology in modern computing environments.

5 Modeling

5.1 Finding the most suitable outcome for model

5.1.1 Demonstrate the correlation rate between variables by using heat map

```
##finding the outcome variable by heatmap
temp=cor(data[,sapply(data,is.numeric)])
corrplot(temp,method="number")
```

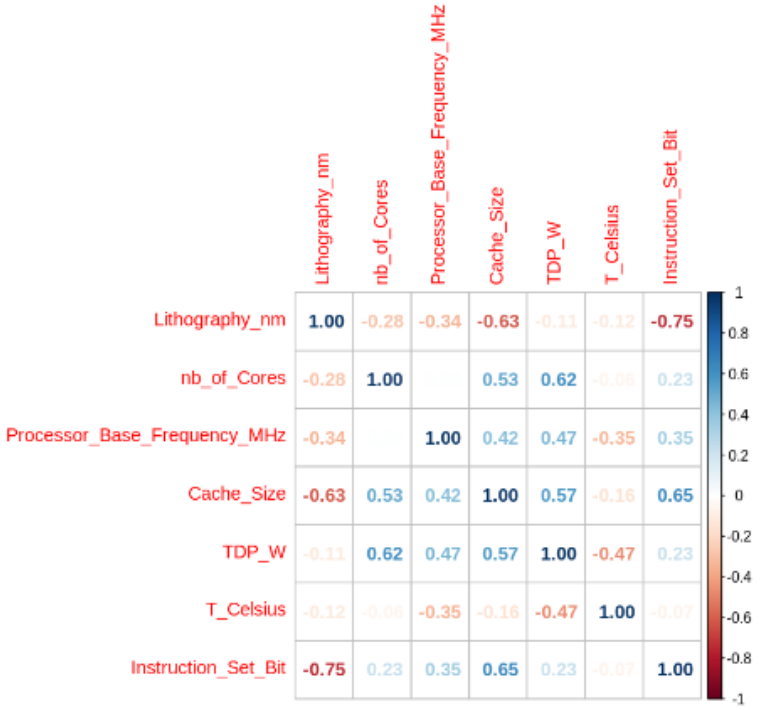

Figure 19: Heat map describes correlation rate of seven variables by the color temperature

5.1.2 Illustrate the correlation between variables by using scatter plot

```
data <- data[, c("Product_Collection", "Vertical_Segment","Status", "Cache_Type","
    Lithography_nm","nb_of_Cores","Processor_Base_Frequency_MHz", "Cache_Size", "TDP_W
    ", "T_Celsius", "Instruction_Set_Bit" )]
pairs(data[, 5:11],
      main = "Matrix Scatter Plot",
      diag.panel = NULL, # remove diagonal panels
      upper.panel = NULL, # remove upper panels
      lower.panel = panel.smooth) # add smoother to lower panels
```

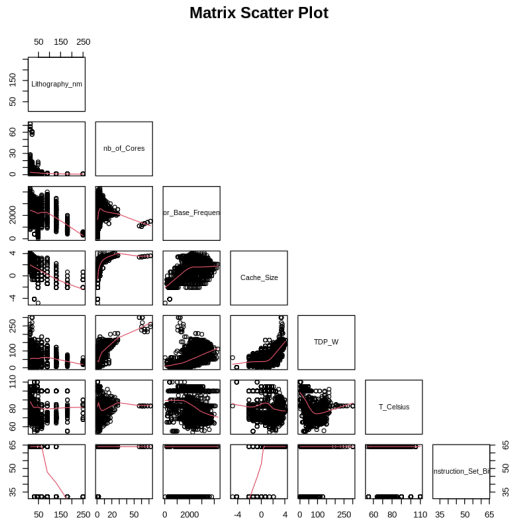

Figure 20: Scatter plot showing correlation between seven chosen variables

5.1.3 Finding the most suitable outcome for model

Through the heat map (in figure 19 and scatter plot (in figure 20), the relationship rate between chosen variables was introduced.

- **Lithography** (*Lithography_nm*) has the least correlation with six others. This is proved by the negative rate shown in the heat map (figure 19) with $-0.28, -0.34, -0.63, -0.11, -0.12, -0.75$ respectively.
- **Number of cores**, which denoted as *nb_of_Cores*, have more connection than the previous one with good correlation rates to three out of six comparing variables including *Cache_size* (0.53), TDP_W (0.62), $Instruction\_Set\_Bit$ (0.23). However, to the rest CPU' features, the number of cores could not be explained by them, proven by the gradient warm tone of the ratio color shown in figure 19.
- **Processor base frequency**($Processor\_Base\_Frequency\_MHz$)
- **Size of cache** (*Cache_Size*)
- **Thermal design power** (TDP_W) have most
- **Temperature**($T\_Celsius$)
- **Instruction set**($Instruction\_Set\_Bit$)

In conclusion, Thermal design power (TDP_W) was proven as the one having the most relation to the other names.

5.2 Building model

5.2.1 Multi Linear Regression Modeling

In R language, the linear regression model represented by:

- $mlr <- lm(\ldots)$: This line creates a linear regression model (lm) and assigns it to the variable mlr.
- $formula = TDP\_W \sim Lithography\_nm + nb\_of\_Cores + Processor\_Base\_Frequence\_MHz + Cache\_Size + T\_Celsius + Instruction\_Set\_Bit$ represents for the MLR formular as introducted in section 2.1

```
mlr<-lm(formula=TDP_W ~ Lithography_nm+nb_of_Cores+Processor_Base_Frequency_MHz
    +Cache_Size+T_Celsius+Instruction_Set_Bit,data=data)
```

5.2.2 Multicollinear detection

According to the theory in section 2.4, the R language supports the VIF() function to measure the multicollinearity of each independent variable of the MLR model. In figure 21, all VIF values are in the range $(1,5]$, and most are around 2. In other words, all independent variables have enough coefficient to each other to summarize say that all variables are available for the model

```
VIF(mlr)
```

```
                Lithography_nm                     nb_of_Cores
                      2.830818                        1.557343
Processor_Base_Frequency_MHz                    Cache_Size
                      1.545004                        2.860430
                     T_Celsius               Instruction_Set_Bit
                      1.315792                        2.697681
```

Figure 21: Result of VIF multicollinearity detection

5.2.3 Summary

```
summary(mlr)
```

This line of code generates a summary of the linear regression model *mlr* as mentioned in 5.2.1, including *coefficients*, *p-values*, *Multiple R-squared* and other statistics. For more detail, The *Adjusted R-squared value* is also provided in the red box of figure 22, which is a measure of how well the independent variables explain the variation in the dependent variable.
In this case, an *Adjusted R-squared* of 0.7521 indicates that approximately **75.21%** of the variability in *TDP_W* can be explained by the independent variables included in the model.

```
Coefficients:
                             Estimate Std. Error t value Pr(>|t|)
(Intercept)                 32.0810080  6.4251583   4.993  6.4e-07 ***
Lithography_nm               0.3363691  0.0174126  19.318  < 2e-16 ***
nb_of_Cores                  3.7739069  0.0910010  41.471  < 2e-16 ***
Processor_Base_Frequency_MHz 0.0201593  0.0007092  28.425  < 2e-16 ***
Cache_Size                   9.4615900  0.5597552  16.903  < 2e-16 ***
T_Celsius                   -0.7660741  0.0406009 -18.868  < 2e-16 ***
Instruction_Set_Bit          0.0688504  0.0612444   1.124    0.261
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 22.04 on 2264 degrees of freedom
Multiple R-squared:  0.7528,    Adjusted R-squared:  0.7521
F-statistic:  1149 on 6 and 2264 DF,  p-value: < 2.2e-16
```

Figure 22: The summary of *mlr* variable

5.3 Prediction

```
newpredictdata<-data.frame(Lithography_nm=c(30),nb_of_Cores=c(30),Processor_Base_
    Frequency_MHz=c(3), Cache_Size=c(10),T_Celsius=c(60),Instruction_Set_Bit=c(64))
TDP_prediction<-predict(mlr,newpredictdata,interval = "confidence")
TDP_prediction
```

- This first line creates a new data frame named *newpredictdata* containing the values of the independent variables, which users want to predict the dependent variable *TDP_W*.
- The second line uses the *predict* function to make predictions of the dependent variable *TDP_W* using the linear regression model *mlr* and the new data *newpredictdata*. The interval = "confidence" argument indicates that you want to compute confidence intervals for the predictions.
- The third one prints the predictions of *TDP_W* along with their associated confidence intervals. The *TDP_W* object can be observed in figure 23 contains the predicted values as well as lower and upper bounds of the confidence intervals.

A matrix: 1 × 3 of type dbl

	fit	lwr	upr
1	208.5076	199.2541	217.7612

Figure 23: The prediction of the model

In figure 23, it points out:

- The ***fit*** component of the predict function *(208.5076 W)* refers to the predicted values of the dependent variable (*TDP_W* in this case) based on the provided independent variables and the trained linear regression model *(mlr)*. These values represent the model's best estimate of the dependent variable given the input data.

- The *lower* presented by ***lwr*** in figure 23 illustrate the lower bound of these confidence intervals. Confidence intervals provide a range within which we can be reasonably confident (95% confidence by default) that the true value of the dependent variable lies, given the uncertainty in the model and the data. The lower bound of the confidence interval indicates the lower limit of this range.
- Similarly, the *upper* denoted as ***upr*** represents the upper bound of the confidence intervals calculated by the predict function. This upper bound indicates the upper limit of the range within which we can be reasonably confident that the true value of the dependent variable lies.

In conclusion, we can be reasonably confident that the true value of *TDP_W* falls within the range from ***199.2541W*** and ***217.7612W*** with the best estimate is ***208.5076W***.

References

Bartels, R., & Goodhew, J. (1981). The robustness of the durbin-watson test. *The Review of Economics and Statistics*, 136–139.

Bhandari, A. (2024). Multicollinearity — causes, effects and detection using vif (updated 2024). https://www.analyticsvidhya.com/blog/2020/03/what-is-multicollinearity/#Introduction

Chatterjee, S., & Hadi, A. (2006). *Regression analysis by example*. Wiley. https://books.google.ru/books?id=uiu5XsAA9kYC

Goldstein, R. (1993). Conditioning diagnostics: Collinearity and weak data in regression. *Technometrics*, *35*(1), 85–86. https://doi.org/10.1080/00401706.1993.10484997

Sinharay, S. (2010). An overview of statistics in education. *International Encyclopedia of Education*, 1–11. https://doi.org/10.1016/b978-0-08-044894-7.01719-x