VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

Course:

PROBABILITY AND STATISTICS (MT2013)

Project: MEMORY SPEED PREDICTION BY USING MULTIPLE LINEAR REGRESSION

Supervisor: **Dr. Nguyen Tien Dung**

Students: Nguyen Thien Loc - 2252460

Nguyen Hong Phuc - 2252639

Ly Trieu Uy - 2252889

Le Nhan Van - 2252899

Che Thanh Vy - 2252937

Class: CC01 - Group 25

Ho Chi Minh City, April 2024

Group contribution:

Members	Contribution	Note
Nguyen Thien Loc	20%	
Nguyen Hong Phuc	20%	Leader
Ly Trieu Uy	20%	
Le Nhan Van	20%	
Che Thanh Vy	20%	

Lecturers' assessment:

Members	Grade	Assessment
Nguyen Thien Loc		
Nguyen Hong Phuc		
Ly Trieu Uy		
Le Nhan Van		
Che Thanh Vy		

Contents

1 Data Introduction

1.1 Dataset descriptions

The graphics processing unit **(GPU)** has become one of the most important types of computing technology, both for personal and business computing. Designed for parallel processing, the GPU is used in a wide range of applications, including graphics and video rendering. Although they're best known for their capabilities in gaming, GPUs are becoming more popular for use in creative production and artificial intelligence (AI).

With the rapid advancement of computer graphics and visualization technologies, GPUs have become essential components in a wide range of applications, including gaming, scientific research, artificial intelligence, and data analysis.

For that reason, this report aims to predict memory speed based on different GPUs by analyzing GPUs' attributes such as Memory_Speed, Memory, Memory_Bus, etc., in the dataset.

Data descriptions for the 13 variables are as follows:

1.2 Variables descriptions

Variables	Data type	Unit	Description
Memory_Speed	Continuous	MHz	Measure of the speed a card can perform texture mapping
Memory	Continuous	MB	Dedicated onboard memory used for storing and accessing data needed for graphics processing tasks

Memory_Bus	Continuous	bit	Type of computer bus, usually in the form of a set of wires or conductor, allow transfers of data and addresses from the main memory to the CPU or a memory controller
Core_Speed	Continuous	MHz	The speed of the graphics card.
Process	Continuous	nm	The size of the smallest features that can be created on the semiconductor chip during manufacturing.
Pixel_Rate	Continuous	GPixel/s	The number of pixels the graphics processor could write to the video memory
Texture_Rate	Continuous	GTexel/s	Measure of the speed with which a particular card can perform texture mapping
Memory_Bandwidth	Continuous	GB/sec	The amount of information that can be transferred to and from memory per unit time
TMUs	Continuous	None	specialized components responsible for applying textures to 3D objects in graphics rendering

Shader	Continuous	None	a user-defined program designed to run on some stage of a graphics processor
Resolution_WxH	Continuous	None	refers to the width (W) and height (H) dimensions of an image or display, expressed as the number of pixels in each direction
Manufacturer	Discrete	None	Manufacturer of the GPU.
Memory_Type	Discrete	None	RAM (Random Access Memory) which can read and write data, and ROM (Read Only Memory)

Table 3: Variables description

2 Background

2.1 Concept of Multiple linear regression (MLR)

Multiple linear regression is a statistical method used in various fields to analyze relationships between a dependent variable and two or more independent variables. It extends simple linear regression by incorporating multiple predictors, allowing for a more nuanced understanding of their collective influence. The model estimates coefficients for each predictor, indicating their strength and direction of effect.

2.2 The model of MLR

$$\mathbf{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \epsilon$$

where:

- y is the dependent variable
- x_i is the i_{th} independent variable
- β_0: y-intercept
- β_i: coefficient of x_i
- ϵ: is the error term

2.3 Assumptions of MLR

2.3.1 Linearity

Independent variables and dependent variable shared a linear relationship.

2.3.2 Homoscedasticity

The error term (random disturbance in the relationship between the independent variables and the dependent variable) is the same across all values of the independent

variables.

2.3.3 Independence

The observations are not influenced by each other.

2.3.4 Normality

The errors are normally distributed.

2.4 Advantages of linear regression

- **Simplicity and Interpretability:** Linear-regression models are relatively simple and provide an easy-to-interpret mathematical formula that can generate predictions. Linear regression can be applied to various areas in business and academic study.
- **Quick initial insight:** The regression coefficients quickly provide clear insights into the relative importance and effects of each independent variable on the dependent variable even if the relationship is not perfectly linear.
- **Baseline model:** Linear regression is the fundamental model for improving to a more complex model.
- **Stability:** Particularly in situations when there are few features, linear regression tends to be more stable and less prone to overfitting. Having this can be helpful when working with smaller datasets.

2.5 Concept of Random forest regression (RFR)

Random Forest is a versatile machine learning algorithm that's adept at classification and regression tasks. It constructs a multitude of decision trees during training and combines their predictions to make more accurate and robust predictions compared to individual trees. By leveraging random sampling of data and features, it mitigates overfit-

ting and generalizes well to unseen data. Additionally, it requires minimal hyperparameter tuning, making it easy to use and suitable for a wide range of applications.

2.6 Assumptions of RFR

2.6.1 Quality of data

Real values and datasets with no missing value while the input value is continuous and the target variable is discrete are provided for best predictions.

2.6.2 Independence of Trees

Each tree's predictions must have very low correlations.

2.6.3 Appropriate Hyperparameters

While Random Forests are relatively robust to hyperparameters, it is essential to tune them appropriately for optimal performance. Common hyperparameters include the number of trees in the forest, tree depth, minimum samples required for splitting nodes, and the number of features considered for each split.

2.7 Advantages of random forest regression

- **Quickly applied time:** Random forest regression requires shorter training time than other algorithms.
- **High accuracy:** Predicts output with great accuracy, and it works efficiently even on big datasets and even when significant amount of data is absent.
- **Reduced Bias:** As they combine the predictions of multiple trees, reducing the risk of bias associated with any single tree.

3 Descriptive Statistics

3.1 Data reading

We'll use R's read.csv function to import our data. Our focus will be on the following columns: Memory, Resolution, Manufacturer, Core Speed, Memory Bus, Memory Speed, Process, Pixel Rate, Texture Rate, TMUs, Shader, Memory Bandwidth. This emphasis is crucial since our primary goal is to predict Memory Speed. We'll check each column for missing values, as minimizing these is essential for accurate prediction.:

```
getwd() #get working directory
XSTK_GPU = read.csv("All_GPUs.csv",header=TRUE,na.strings=c("","\n- ","\n","\nUnknown
Release Date")) #read csv file
Main_XSTK = XSTK_GPU[,c("Memory","Resolution_WxH","Manufacturer","Core_Speed","Memory_Bus",
"Memory_Speed","Memory_Type","Process","Pixel_Rate","Texture_Rate", "TMUs", "Shader",
"Memory_Bandwidth")] #select needed features

print( apply(is.na(Main_XSTK),2,sum)) #To check for null values
```

Then we check for the missing values in the dataset

```
        Memory   Resolution_WxH     Manufacturer       Core_Speed       Memory_Bus
           420              195                0              936               62
Memory_Speed      Memory_Type          Process       Pixel_Rate     Texture_Rate
           105               56              463              544              544
          TMUs           Shader Memory_Bandwidth
           538              107              121
```

3.2 Data cleaning

The missing values comprise less than 30% of our dataset. Therefore, we'll convert relevant columns to numeric data types and replace N/A values with their corresponding median values to prepare the data for analysis

```
Convert <- function(x){
    if(is.na(x))
        {return (NA)}
    else
        {return (as.double( strsplit(x," ")[[1]][[1]] ))
    }
} #function to convert character to numeric value

Main_XSTK$Memory <- sapply(Main_XSTK$Memory, Convert)
Main_XSTK$Memory[is.na(Main_XSTK$Memory)] = median(Main_XSTK$Memory, na.rm=T)

Main_XSTK$Core_Speed <- sapply(Main_XSTK$Core_Speed, Convert)
Main_XSTK$Core_Speed[is.na(Main_XSTK$Core_Speed)] = median(Main_XSTK$Core_Speed, na.rm=T)

Main_XSTK$Memory_Bus <- sapply(Main_XSTK$Memory_Bus, Convert)
Main_XSTK$Memory_Bus[is.na(Main_XSTK$Memory_Bus)] = median(Main_XSTK$Memory_Bus, na.rm=T)

Mode <- function(x) {
  ux <- unique(x)
  ux[which.max(tabulate(match(x, ux)))]
} #Function to fill with mode
mode_value <- Mode(Main_XSTK$Memory_Type) #Calculate the mode value
Main_XSTK$Memory_Type[is.na(Main_XSTK$Memory_Type)] <- mode_value #fill Memory Type with Mode Value

mode_value <- Mode(Main_XSTK$Shader)
Main_XSTK$Shader <- ifelse(is.na(Main_XSTK$Shader), mode_value, Main_XSTK$Shader)

#Process
Main_XSTK$Process = as.double( gsub("[^0-9]","",Main_XSTK$Process) )
Main_XSTK$Process[is.na(Main_XSTK$Process)] = median(Main_XSTK$Process, na.rm=T)

#Pixel Rate
Main_XSTK$Pixel_Rate = sapply(Main_XSTK$Pixel_Rate, Convert)
Main_XSTK$Pixel_Rate[is.na(Main_XSTK$Pixel_Rate)] = median(Main_XSTK$Pixel_Rate, na.rm=T)

#Texture Rate
Main_XSTK$Texture_Rate = sapply(Main_XSTK$Texture_Rate, Convert)
Main_XSTK$Texture_Rate[is.na(Main_XSTK$Texture_Rate)] = median(Main_XSTK$Texture_Rate, na.rm=T)

#Memory Speed
Main_XSTK$Memory_Speed = sapply(Main_XSTK$Memory_Speed, Convert)
Main_XSTK$Memory_Speed[is.na(Main_XSTK$Memory_Speed)] = median(Main_XSTK$Memory_Speed, na.rm=T)
```

And for some value, we'll use the mode (the most frequent value) to fill in some missing data. This approach helps us preserve the overall shape of the data distribution, especially when dealing with categories

```
Main_XSTK$TMUs[is.na(Main_XSTK$TMUs)] = median(Main_XSTK$TMUs, na.rm=T)

mode_value <- Mode(Main_XSTK$Resolution_WxH) #Calculate the mode value
Main_XSTK$Resolution_WxH[is.na(Main_XSTK$Resolution_WxH)] <- mode_value
```

After cleaning the data set, we re-check the missing value of the data set again

```
#check for missing values again:
print( apply(is.na(Main_XSTK),2,sum)) #To check for null values
```

```
##            Memory    Resolution_WxH      Manufacturer        Core_Speed
##                 0                 0                 0                 0
##        Memory_Bus      Memory_Speed       Memory_Type           Process
##                 0                 0                 0                 0
##        Pixel_Rate      Texture_Rate              TMUs            Shader
##                 0                 0                 0                 0
## Memory_Bandwidth
##                 0
```

After checked, There's no missing value in our data set

3.3 Data visualization

3.3.1 Summary table

We summary all the columns in the data set using "describe"

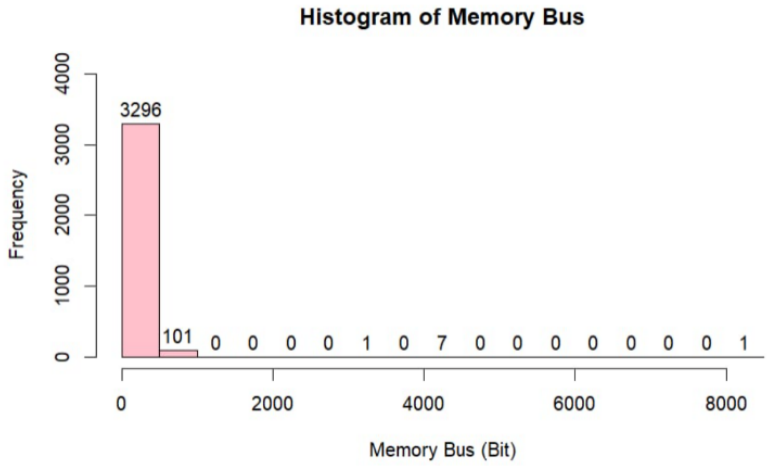
```
describe(Main_XSTK)
```

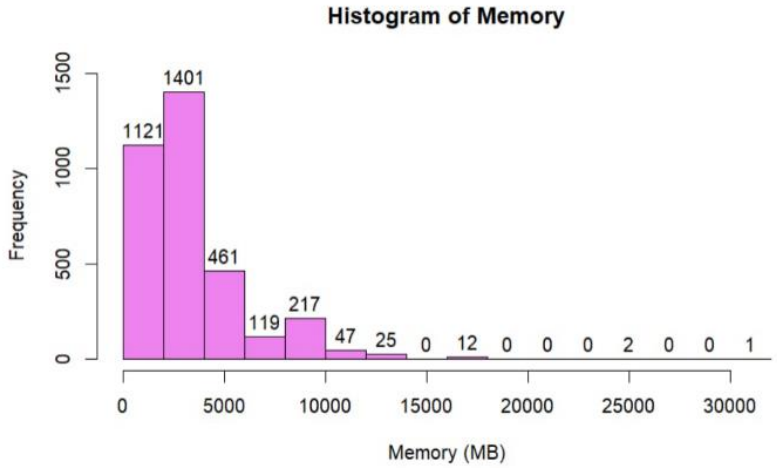
	vars	n	mean	sd	median	trimmed	mad	min
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Memory	1	3406	2771.07	2627.75	2048	2286.32	1518.18	16
Resolution_WxH*	2	3211	11.75	3.46	12	11.26	4.45	1
Manufacturer*	3	3406	2.71	1.42	4	2.76	0.00	1
Core_Speed	4	3406	955.99	234.48	980	958.61	118.61	100
Memory_Bus*	5	3344	4.57	4.30	4	3.93	4.45	1
Memory_Speed*	6	3301	94.95	54.37	91	94.93	72.65	1
Memory_Type*	7	3350	7.11	2.82	9	7.47	0.00	1
Process*	8	2943	8.12	2.05	8	8.23	0.00	1
Pixel_Rate*	9	2862	83.28	35.57	86	84.72	32.62	1
Texture_Rate*	10	2862	166.14	99.61	176	168.09	140.85	1

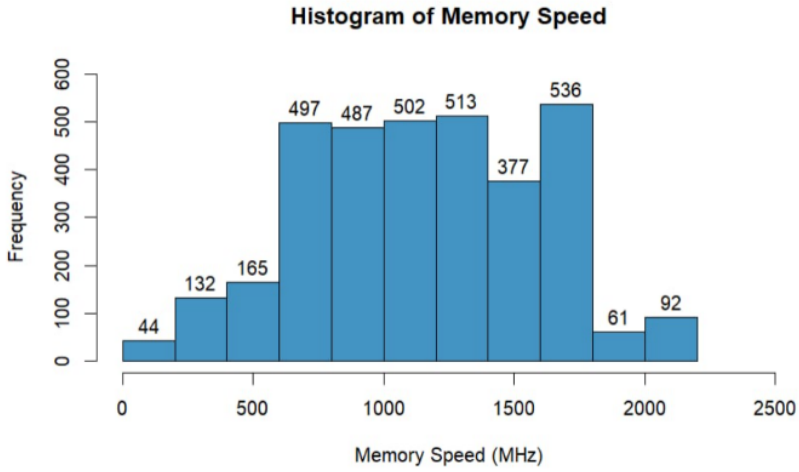
3.3.2 Histogram

Use histograms to analyze the distributions of these 12 key variables: Memory Speed, Memory, Memory Bus, Core Speed, Process, Pixel Rate, Texture Rate, Memory Bandwidth, TMUs, Shader, Resolution WxH, Manufacturer, and Memory Type. This will help us understand their spread, central tendencies, and any potential outliers

```
hist(Main_XSTK$Memory,xlab="Memory (MB)",main="Histogram of Memory", col="violet",label=T,
ylim=c(0,1500))
hist(Main_XSTK$Memory_Bus,xlab="Memory Bus (Bit)",main="Histogram of Memory Bus",
col="pink",label=T, ylim=c(0,4000))
hist(Main_XSTK$Memory_Speed,xlab="Memory Speed (MHz)",main="Histogram of Memory Speed",
col="#4393C3",label=T, ylim=c(0,600),xlim=c(0,2500))
```





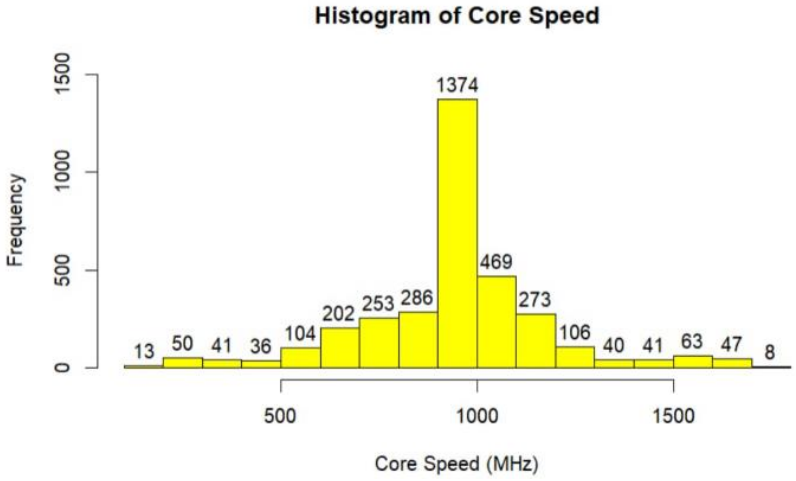


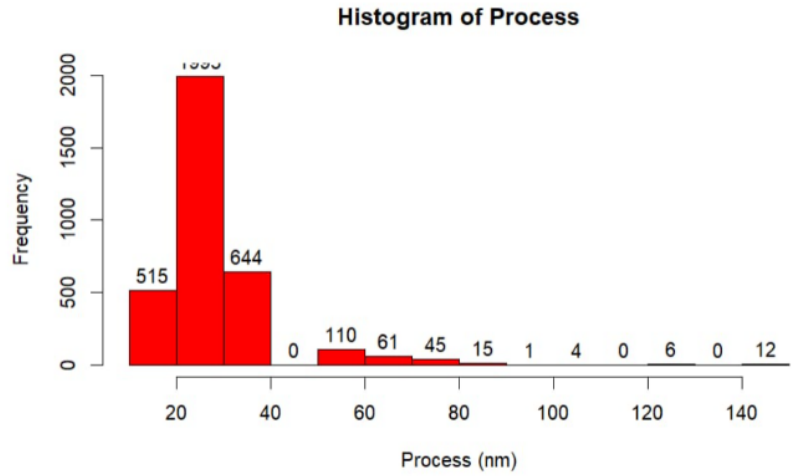
Description:

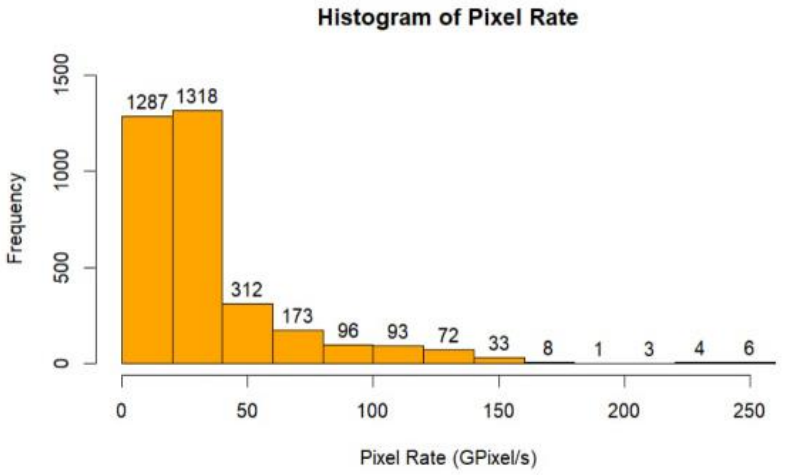
- Histogram of Memory Speed
  - The highest frequency is 536 occurs approximately between 1643 and 1786 MHz
  - The lowest frequency is 44 occurs approximately between 0 and 200 MHz.
  - This histogram tends to be central.
- Histogram of Memory
  - The highest frequency is 1401 occurs approximately between 2000 and 4000 MB
  - The lowest frequency is 0 occurs approximately between 14000 and 16000 MB, from 18000 to 24000 MB, and from 26000 to 30000 MB
  - This histogram tends to skew left.
- Histogram of Memory Bus
  - The highest frequency is 3296 occurs approximately between 0 and 5000 Bit.
  - The lowest frequency is 0 occurs approximately between 1000 and 8000 Bit except from 3000 to 3500 Bit, from 4000 to 4500 Bit and from 8000 to 8500 Bit.
  - This histogram tends to skew left.

Our next analysis will center on Core Speed, Process, and Pixel Rate.

```
hist(Main_XSTK$Core_Speed,xlab="Core Speed (MHz)",main="Histogram of Core Speed",
col="yellow",label=T, ylim=c(0,1500))
hist(Main_XSTK$Process,xlab="Process (nm)",main="Histogram of Process", col="red",label=T,
ylim=c(0,2000))
hist(Main_XSTK$Pixel_Rate,xlab="Pixel Rate (GPixel/s)",main="Histogram of Pixel Rate",
col="orange",label=T, ylim=c(0,1500))
```





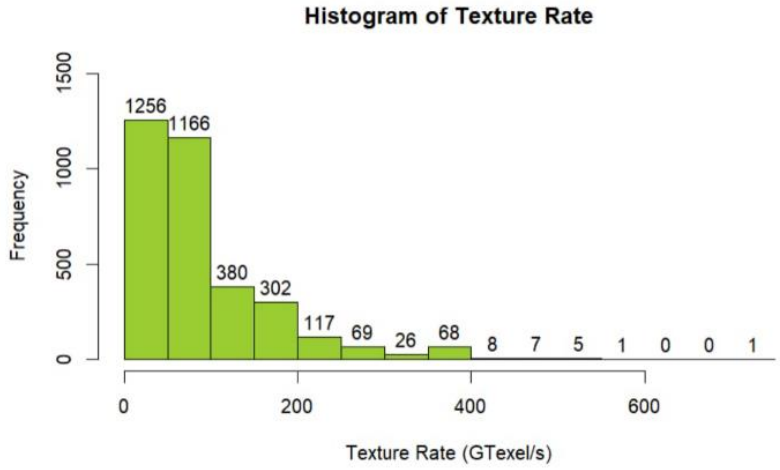


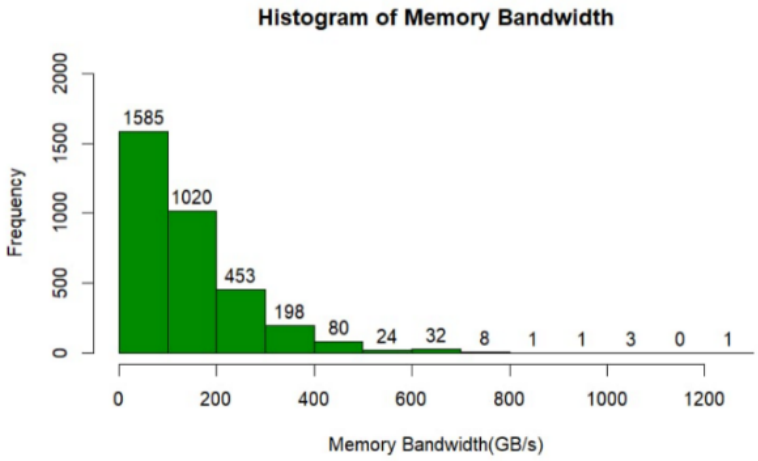
Description:

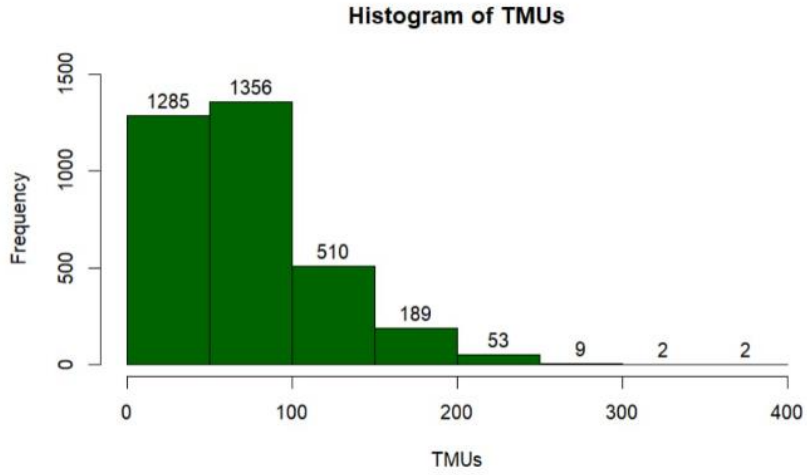
- Histogram of Core Speed
  - The highest frequency is 1374 occurs approximately between 900 and 1000 MHz.
  - The lowest frequency is 8 occurs approximately between 1700 and 1800 MHz.
  - This histogram tends to be central.
- Histogram of Process
  - The highest frequency is 1993 occurs approximately between 20 and 30nm.
  - The lowest frequency is 0 occurs approximately between 40 and 50 MB, from 110 to 120 nm, and from 130 to 140 nm.
  - This histogram tends to skew left.
- Histogram of Pixel Rate
  - The highest frequency is 1318 occurs approximately between 20 and 40 GPixel/s.
  - The lowest frequency is 1 occurs approximately between 180 and 200.
  - This histogram tends to skew left.

Now, let's shift our attention to Texture Rate, Memory Bandwith, and TMUs and explore their characteristics.

```
hist(Main_XSTK$Texture_Rate,xlab="Texture Rate (GTexel/s)",main="Histogram of Texture
Rate", col="yellowgreen",label=T, ylim=c(0,1500))
hist(Main_XSTK$Memory_Bandwidth,xlab="Memory Bandwidth(GB/s)",main="Histogram of Memory
Bandwidth", col="green4",label=T, ylim=c(0,2000))
hist(Main_XSTK$TMUs,xlab="TMUs",main="Histogram of TMUs", col="darkgreen",label=T,
ylim=c(0,1500))
```





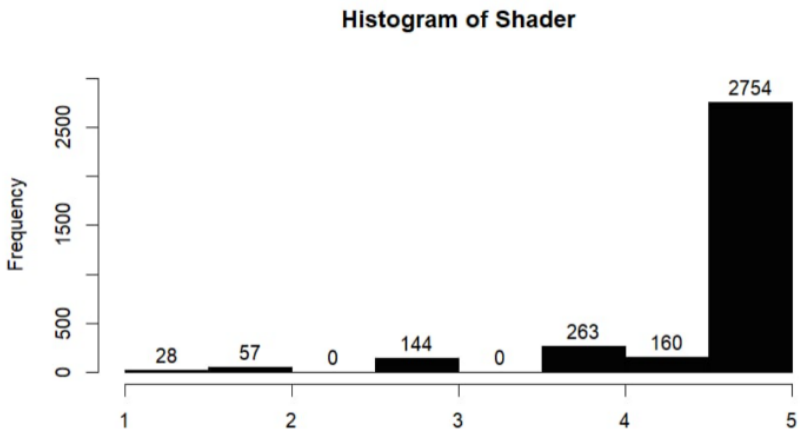


Description:

- Histogram of Texture Rate
  - The highest frequency is 1256 occurs approximately between 0 and 50 GTexel/s.
  - The lowest frequency is 0 occurs approximately between 600 and 700 GTexel/s.
  - This histogram tends to skew left.
- Histogram of Memory Bandwidth
  - The highest frequency is 1585 occurs approximately between 0 and 100 GB/s.
  - The lowest frequency is 0 occurs approximately between 1100 and 1200 GB/s.
  - This histogram tends to skew left.
- Histogram of TMUs
  - The highest frequency is 1356 occurs approximately between 50 and 100.
  - The lowest frequency is 2 occurs approximately between 300 to 400.
  - This histogram tends to skew left.

Next, we will analyze Shader, Resolution

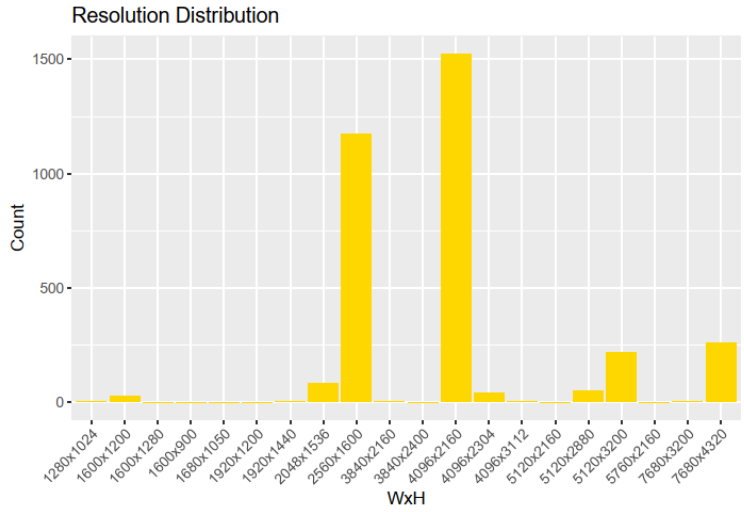
```
hist(Main_XSTK$Shader,xlab="Shader",main="Histogram of Shader", col="gray1",label=T,
ylim=c(0,3000))
```




```
count_Reso <- table(Main_XSTK$Resolution_WxH)
count_Reso
```

```
##
## 1280x1024 1600x1200 1600x1280  1600x900 1680x1050 1920x1200 1920x1440 2048x1536
##         3         28         1         2         1         2         3         85
## 2560x1600 3840x2160 3840x2400 4096x2160 4096x2304 4096x3112 5120x2160 5120x2880
##      1175         3         1      1525        41         5         1         48
## 5120x3200 5760x2160 7680x3200 7680x4320
##       217         1         6       258
```

```
data <- data.frame(
  Resolution = c("1280x1024", "1600x1200", "1600x1280", "1600x900", "1680x1050", "1920x1200", "1920x14
  Count = c(3,        28,         1,         2 ,        1  ,       2,         3 ,       85,      1175,3
)
ggplot(data, aes(x = Resolution, y = Count)) +
  geom_bar(stat = "identity", fill = "gold") +
  labs(title = "Resolution Distribution", x = "WxH", y = "Count") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



Description:

- Histogram of Shader
  - The highest frequency is 2754 occurs approximately between 4.5 and 5.
  - The lowest frequency is 0 occurs approximately between 2 and 2.5 and from 3 to 3.5.
  - This histogram tends to skew right.
- Histogram of Resolution
  - The highest frequency is 1525 occurs at 4096x2160.
  - The lowest frequency is 1 occurs at 1600x1200, 1600x1280, 1680x1050, 3840x2400,

5120x2160, and 5760x2160.

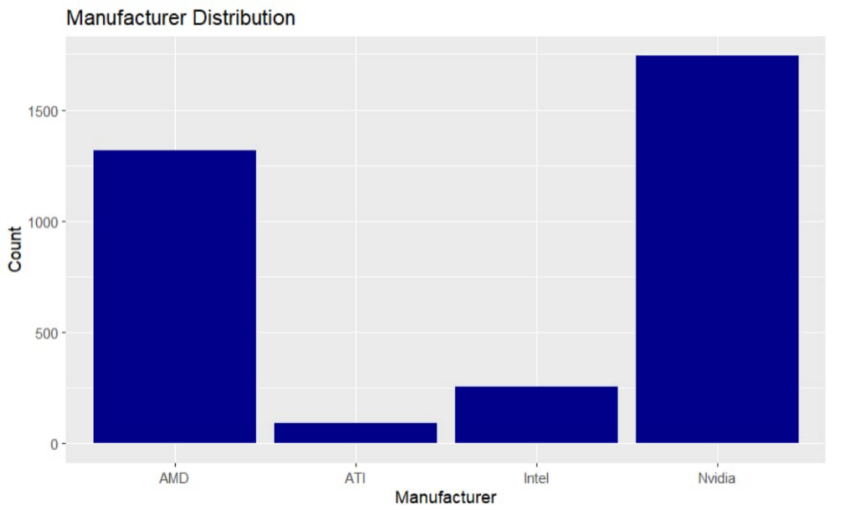
– This histogram tends to fluctuate.

Last, we are going to analyze the Memory Type and Manufacturer

```
count_Manu <- table(Main_XSTK$Manufacturer)
count_Manu
```

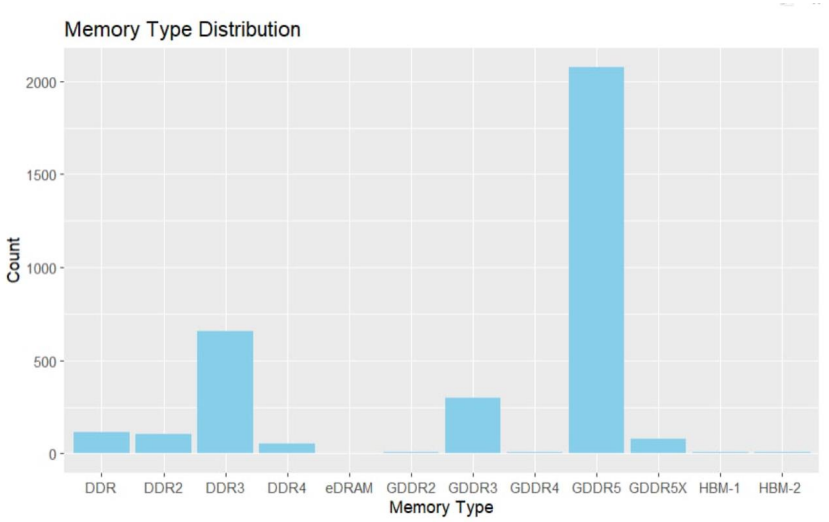
```
 AMD    ATI  Intel Nvidia
1317     92    254   1743
```

```
data <- data.frame(
  Manufacturer = c("Nvidia", "AMD", "ATI","Intel"),
  Count = c(1743, 1317, 92,254)
)
ggplot(data, aes(x = Manufacturer, y = Count)) +
  geom_bar(stat = "identity", fill = "darkblue") +
  labs(title = "Manufacturer Distribution", x = "Manufacturer", y = "Count")
```




```
count_Type <- table(Main_XSTK$Memory_Type)
count_Type
```

```
 DDR  DDR2  DDR3  DDR4 eDRAM GDDR2 GDDR3 GDDR4 GDDR5 GDDR5X HBM-1 HBM-2
 113   102   657    51     3     5   300     8  2076     79     6     6
```

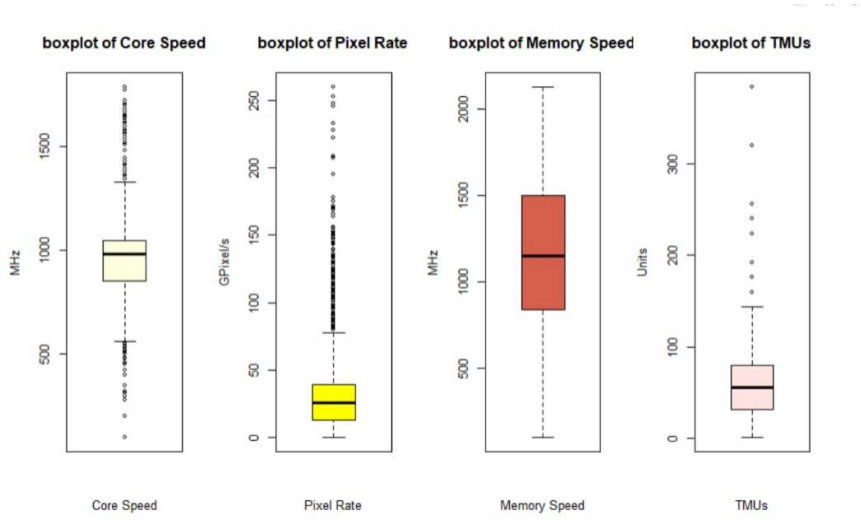


Description:

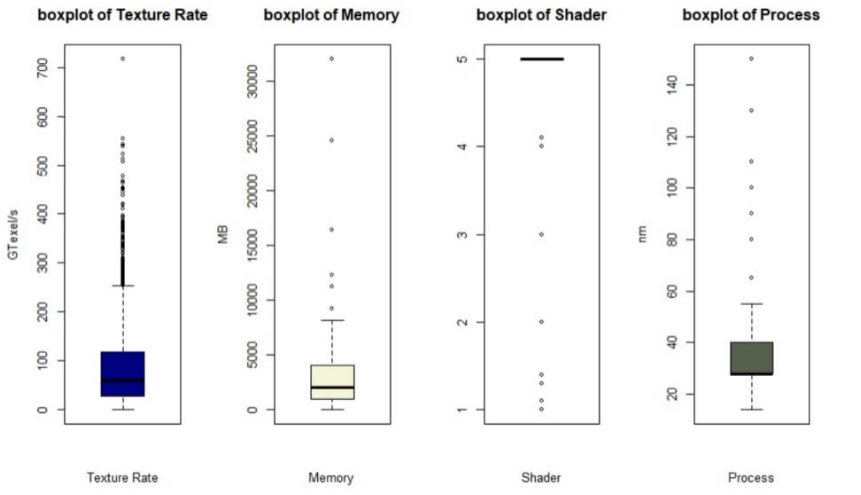
- Histogram of Memory Type
  - The highest frequency is 2754 occurs approximately between 4.5 and 5.
  - The lowest frequency is 0 occurs approximately between 2 and 2.5 and from 3 to 3.5.
  - This histogram tends to fluctuate.
- Histogram of Manufacturer
  - The highest frequency is 1743 occurs at NVIDEA.
  - The lowest frequency is 92 occurs at ATI., 5120x2160, and 5760x2160.
  - This histogram tends to fluctuate.

3.4 Boxplots

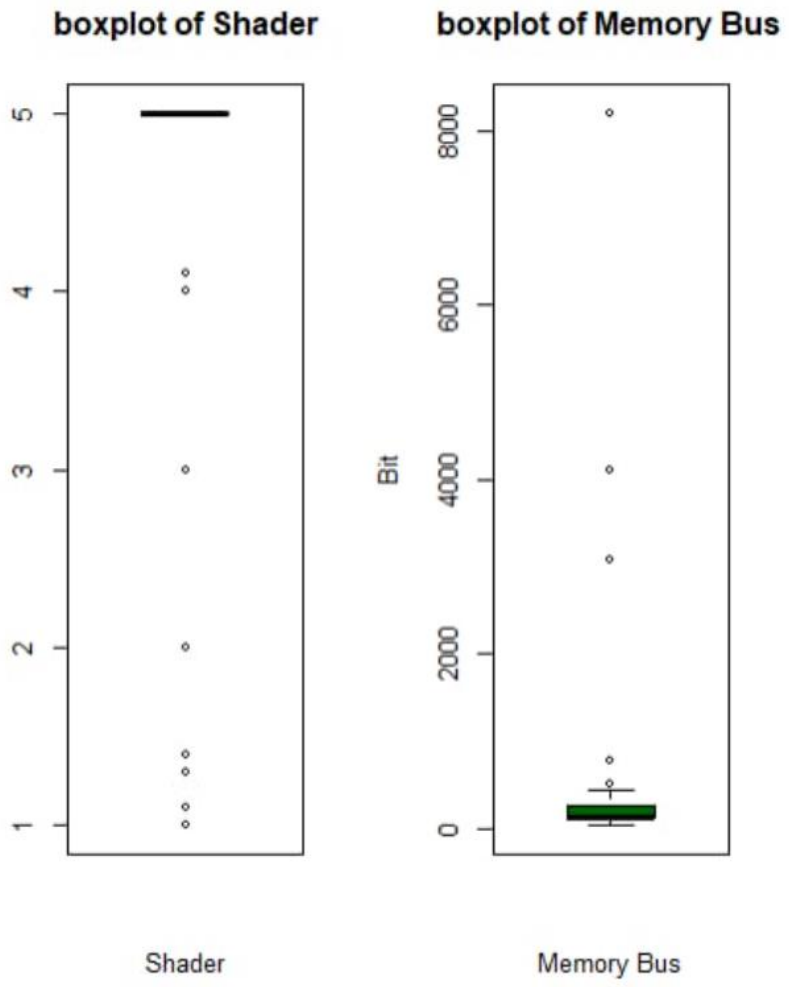
```
par(mfrow=c(1,4))
boxplot(Main_XSTK$Core_Speed,col="lightyellow", main="boxplot of Core Speed",xlab = "Core
Speed",ylab = "MHz")
boxplot(Main_XSTK$Pixel_Rate,col="yellow", main="boxplot of Pixel Rate",xlab = "Pixel
Rate",ylab = "GPixel/s")
boxplot(Main_XSTK$Memory_Speed,col="#D6604D", main="boxplot of Memory Speed",xlab = "Memory
Speed",ylab = "MHz")
boxplot(Main_XSTK$TMUs,col="mistyrose", main="boxplot of TMUs",xlab = "TMUs",ylab =
"Units")
```




```
par(mfrow=c(1,4))
boxplot(Main_XSTK$Texture_Rate,col="navy", main="boxplot of Texture Rate",xlab = "Texture
Rate",ylab = "GTexel/s")
boxplot(Main_XSTK$Memory,col="beige", main="boxplot of Memory",xlab = "Memory",ylab = "MB")
boxplot(Main_XSTK$Shader,col="orange", main="boxplot of Shader",xlab = "Shader")
boxplot(Main_XSTK$Process,col="#56604D", main="boxplot of Process",xlab = "Process",ylab =
"nm")
```




```
par(mfrow=c(1,4))
boxplot(Main_XSTK$Shader,col="orange", main="boxplot of Shader",xlab = "Shader")
boxplot(Main_XSTK$Memory_Bus,col="darkgreen", main="boxplot of Memory Bus",xlab = "Memory
Bus",ylab = "Bit")
```



Comment: We observe that each output is based on input data that is distributed and exhibits in a similar pattern except for Memory Speed with a wider value range and Shader with tiny range of value. It is not feasible to accurately predict the price by solely considering one variable

4 Inferential Statistic

We plan to construct a Multiple Linear Regression (MLR) model where Memory_Speed serves as the output or dependent variable. The independent variables, including Memory, Memory_Bandwidth, Core_Speed, Memory_Bus, Memory_Type, Process, Texture_Rate, and Pixel_Rate, will be utilized. Our aim is to predict the missing values of Memory_Speed in the original dataset using the MLR model, while also examining the relationship between Memory_Speed and the independent variables. Following the assessment of MLR assumptions, aside from normality, we proceed with splitting the dataset and constructing the model.

4.1 Sample splitting

We partitioned the dataset into training and testing subsets to ensure accurate model evaluation. Initially, the model is trained using the training data, followed by refinement using the test data. It's essential to note that evaluating the model solely on the training data is insufficient as it may not generalize well to unseen data.

For this project, we opted for an 80/20 ratio of training to testing data.

```
trainIndex <- createDataPartition(Main_XSTK$Memory_Speed, p = 0.8, list = FALSE)
train_data <- Main_XSTK[trainIndex, ] #80% for train
test_data <- Main_XSTK[-trainIndex, ] #20% for test
```

4.2 Importance of variables

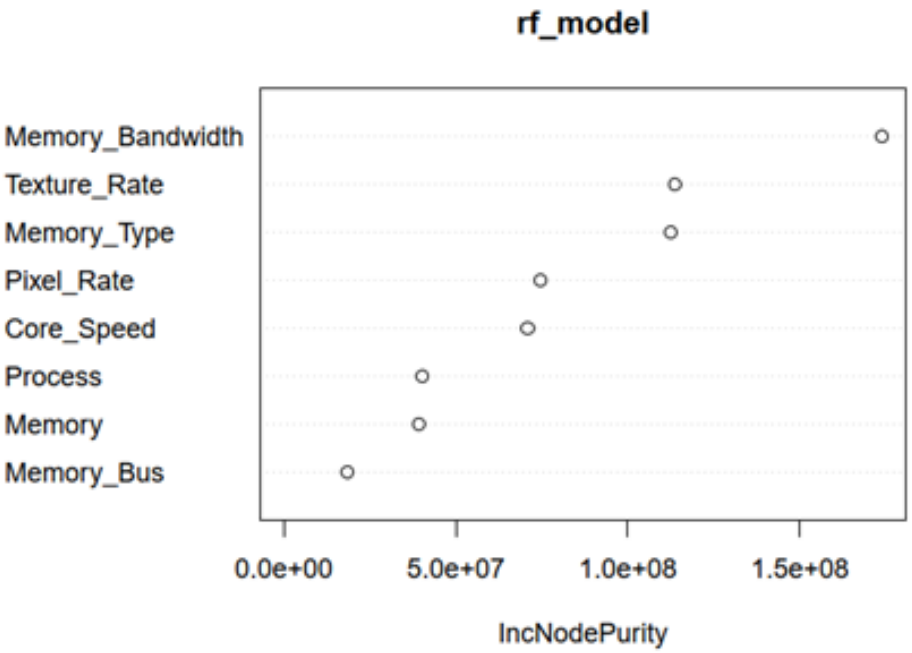
Higher the value of mean decrease accuracy or mean decrease gini score, higher the importance of the variable in the model. In the plot shown above, Memory_Bandwidth is most important variable.

```
library(randomForest) #include to use randomforest
set.seed(123)  # Set seed for reproducibility
rf_model <- randomForest(Memory_Speed ~ Memory + Core_Speed + Memory_Bus +
Memory_Type + Process + Pixel_Rate + Texture_Rate + Memory_Bandwidth ,
data = Main_XSTK) #Tao random forest model de chon variable
```

```
importance(rf_model) # Show importance of variables
```

```
##                   IncNodePurity
## Memory                 39225568
## Core_Speed             70867136
## Memory_Bus             18580634
## Memory_Type           112488450
## Process                40343779
## Pixel_Rate             74634592
## Texture_Rate          113557236
## Memory_Bandwidth      173874912
```

```
varImpPlot(rf_model)
```



4.3 Building multiple linear regression model

```
lm_model <- lm(Memory_Speed ~ Memory + Core_Speed + Memory_Bus +
Memory_Type + Process + Pixel_Rate + Texture_Rate + Memory_Bandwidth, data
= train_data) #Base on the Variables
print(summary(lm_model)) #Print the summary of the model, as we can see
```

```
## Call:
## lm(formula = Memory_Speed ~ Memory + Core_Speed + Memory_Bus +
##     Memory_Type + Process + Pixel_Rate + Texture_Rate + Memory_Bandwidth,
##     data = train_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1212.03  -145.01    -0.63   149.63   835.15
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -1.480e+02  3.857e+01  -3.838 0.000127 ***
## Memory             2.911e-02  2.974e-03   9.790  < 2e-16 ***
## Core_Speed         6.072e-01  2.645e-02  22.950  < 2e-16 ***
## Memory_Bus        -9.240e-02  3.051e-02  -3.028 0.002482 **
## Memory_TypeDDR2    3.113e+02  3.285e+01   9.475  < 2e-16 ***
## Memory_TypeDDR3    5.208e+02  2.609e+01  19.961  < 2e-16 ***
## Memory_TypeDDR4    1.002e+03  4.628e+01  21.653  < 2e-16 ***
## Memory_TypeeDRAM   7.166e+02  1.540e+02   4.654 3.41e-06 ***
## Memory_TypeGDDR2   2.843e+02  1.099e+02   2.587 0.009720 **
## Memory_TypeGDDR3   5.483e+02  2.683e+01  20.438  < 2e-16 ***
## Memory_TypeGDDR4   7.394e+02  8.426e+01   8.775  < 2e-16 ***
## Memory_TypeGDDR5   8.811e+02  2.477e+01  35.569  < 2e-16 ***
## Memory_TypeGDDR5X  5.861e+01  4.062e+01   1.443 0.149154
## Memory_TypeHBM-1  -3.915e+01  1.419e+02  -0.276 0.782572
## Memory_TypeHBM-2  -2.295e+02  1.227e+02  -1.870 0.061583 .
## Process           -3.431e+00  3.553e-01  -9.655  < 2e-16 ***
## Pixel_Rate        -1.372e+00  3.056e-01  -4.491 7.37e-06 ***
## Texture_Rate       9.904e-01  1.610e-01   6.152 8.78e-10 ***
## Memory_Bandwidth   2.604e-01  8.629e-02   3.018 0.002572 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 214.4 on 2708 degrees of freedom
## Multiple R-squared:  0.7634, Adjusted R-squared:  0.7618
## F-statistic: 485.3 on 18 and 2708 DF,  p-value: < 2.2e-16
```

The variables Memory_TypeGDDR5X, Memory_TypeHBM-1, and Memory_TypeHBM-2 are not statistically significant as their p-values are much greater than 0.05. This indicates that these variables do not make a significant contribution to the model and can be removed. Residual standard error stands at 214.4, indicating that actual GPU prices may deviate from the true regression line by roughly 214.4 USD.

Adjusted R-squared value of 0.7634 indicates that approximately 76.34% of the

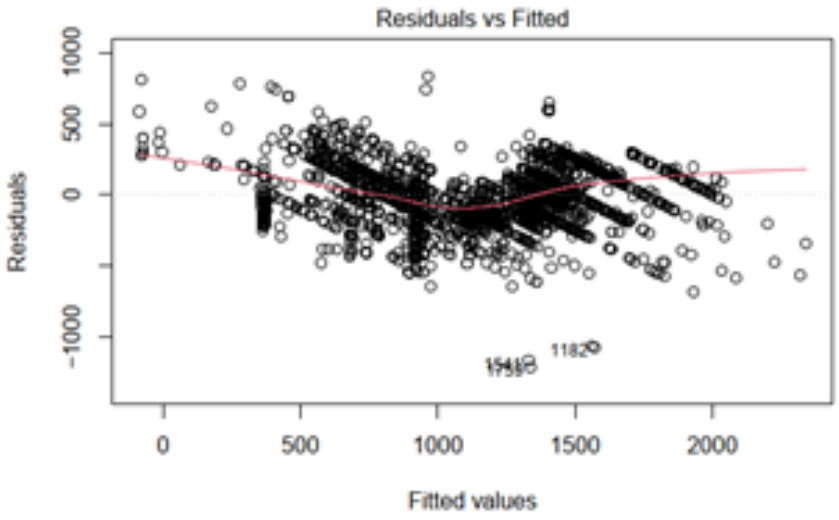
variability in the data can be explained by the model. In this case, with an F-statistic of 485.3 and a p-value below 0.05, we can confidently reject the null hypothesis. This means that there is strong evidence to suggest that at least one of the independent variables included in the model is related to the dependent variable (Memory_Speed). Therefore, we have reason to believe that there is a relationship between the independent variables and the dependent variable.

4.4 Test for assumptions

4.4.1 Assumption 1: Linearity of the Data

We can check the linearity of the data by looking at the Residual versus Fitted plot.

```
plot(lm_model, 1)
```



This red line is not too straight on the 0 line, but it's still acceptable for this model.

4.4.2 Assumption 2: Predictors (x) are Independent & Observed with Negligible Error

The easiest way to check the assumption of independence is using the Durbin-Watson test.

```
durbinWatsonTest(lm_model)
```

```
##  lag Autocorrelation D-W Statistic p-value
##    1       0.4482596      1.101474       0
##  Alternative hypothesis: rho != 0
```

The null hypothesis states that the errors are not auto-correlated with themselves (they are independent). Thus, if we achieve a $p - value > 0.05$, we would fail to reject the null hypothesis. This would give us enough evidence to state that our independence assumption is met!

4.4.3 Assumption 3: Residual Errors have a Mean Value of Zero

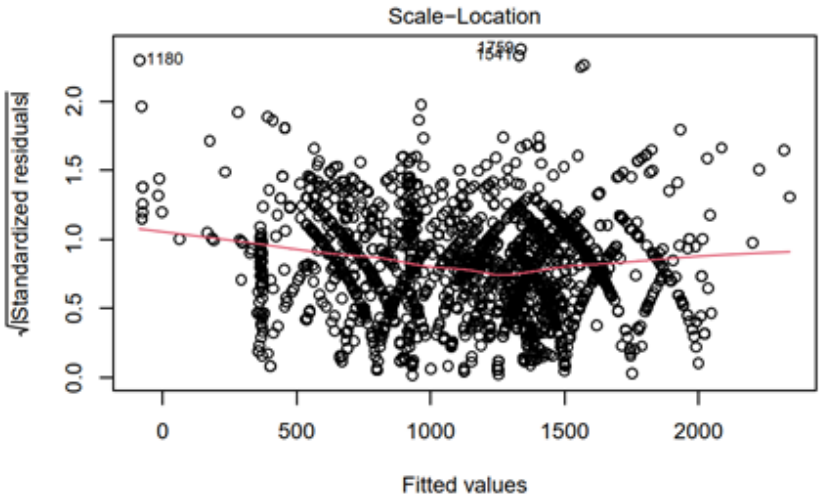
```
residual_mean <- mean(residuals(lm_model))
print(paste("Mean of Residuals:", residual_mean))
```

```
## [1] "Mean of Residuals: -2.13657576331505e-14"
```

This value is $-2.14 \cdot 10^{-14}$, approximate 0, so this assumption has been met.

4.4.4 Assumption 4: Residual Errors have Constant Variance

```
plot(lm_model, 3)
```

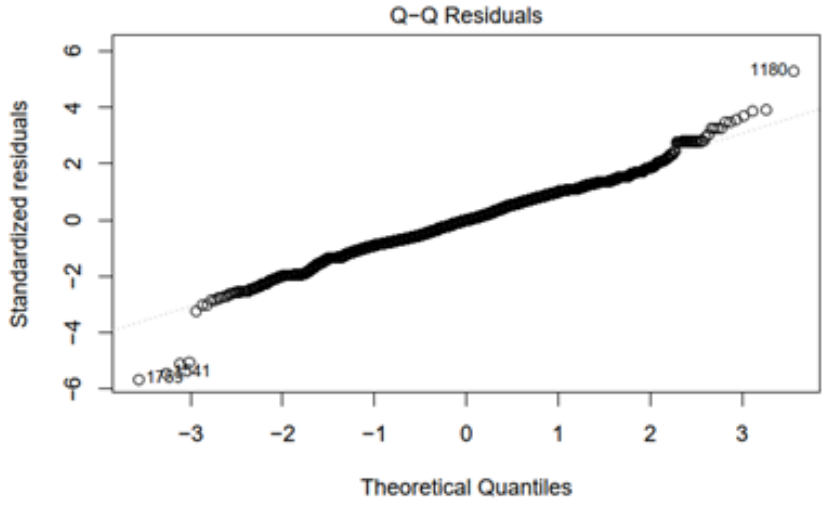


Ideally, we would want to see the residual points equally spread around the red line, which would indicate constant variance. However, it is acceptable.

4.4.5 Assumption 5: Testing for normality of the errors

To check this, we have to use $Q - Q$ plot for normality consideration. The output we expect that the residuals will mostly scatter close to the straight line to get normality hypothesis accepted.

```
plot(lm_model, 2)
```



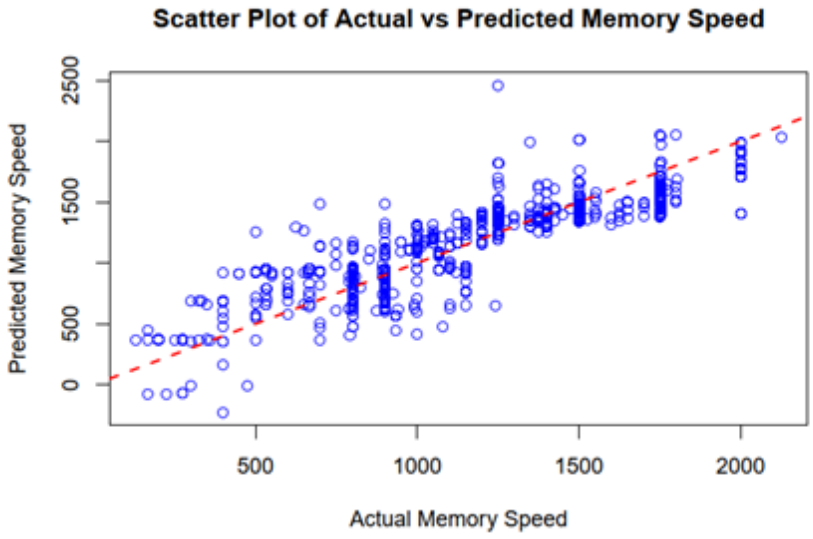
We can conclude that this model can be normally accurate, not 100% accurate. Perhaps another model will be more fitted for this data.

4.4.6 Testing for accuracy

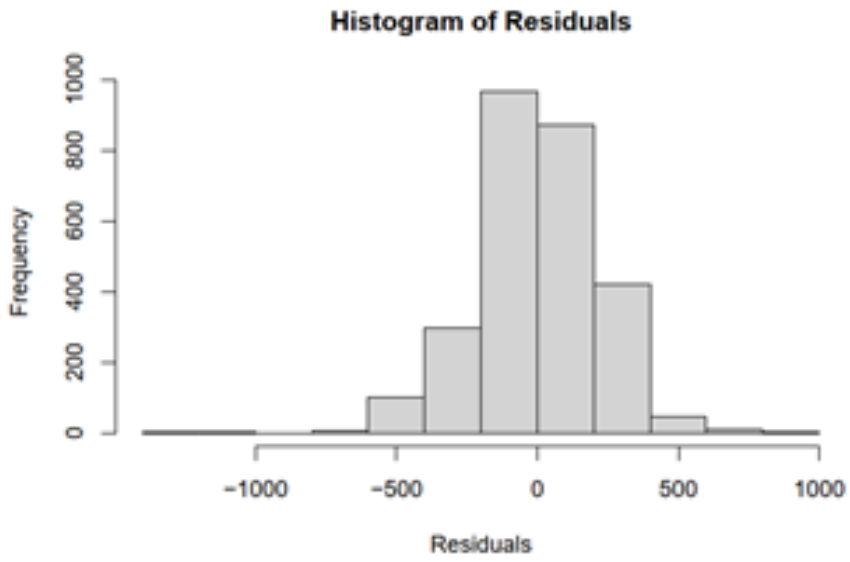
```
predictions <- predict(lm_model, newdata = test_data)
scatter_data <- data.frame(Actual = test_data$Memory_Speed, Predicted = predictions)
plot(scatter_data$Actual, scatter_data$Predicted, xlab = "Actual Memory Speed",
    ylab = "Predicted Memory Speed",
    main = "Scatter Plot of Actual vs Predicted Memory Speed",
    col = "blue")

 abline(0, 1, col = "red", lwd = 2, lty = 2)
```

In an ideal scenario, where the predicted memory speeds perfectly match the actual memory speeds, all the points would fall on a diagonal line with a slope of 1 (y = x). Deviations from this diagonal line indicate discrepancies between the actual and predicted values.




```
residuals <- residuals(lm_model)
hist(residuals, main = "Histogram of Residuals", xlab = "Residuals")
```



By examining the histogram of residuals, you can assess whether the residuals are approximately normally distributed, which is an assumption of many linear regression models. If the histogram shows a roughly bell-shaped curve, it suggests that the assumption is reasonable.

```
rmse <- sqrt(mean((predictions - test_data$Memory_Speed)^2))
print(paste("Root Mean Squared Error (RMSE):", rmse))
rsquared <- cor(predictions, test_data$Memory_Speed)^2
print(paste("R-squared:", rsquared))
```

```
## [1] "Root Mean Squared Error (RMSE): 222.083299043699"
```

```
## [1] "R-squared: 0.747390476312476"
```

This dataset is ranged from 10 to 1700 so this error can be assume as small value so it can be accepted.

```
##          pred real
## 6   970.01315 1100
## 15  366.21985  360
## 18 1313.58539 1300
## 25 1050.02824 1067
## 26  -73.97250  275
```

```
## 30  476.58035  800
## 35  -76.04407  225
## 38  -75.40553  166
## 44  568.60678  933
## 51 2053.40904 1750
```

4.5 Building random forest model

```
print(rf_model)
```

```
Call:
 randomForest(formula = Memory_Speed ~ Memory + Core_Speed + Memory_Bus +      Memory_Type +
Process + Pixel_Rate + Texture_Rate + Memory_Bandwidth,      data = Main_XSTK)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 2

          Mean of squared residuals: 5810.36
                    % Var explained: 96.99
[1] "Mean Absolute Error (MAE): 26.9471501740574"
[1] "Root Mean Squared Error (RMSE): 54.2087329273261"
[1] "R-squared: 0.985351366265067"
```

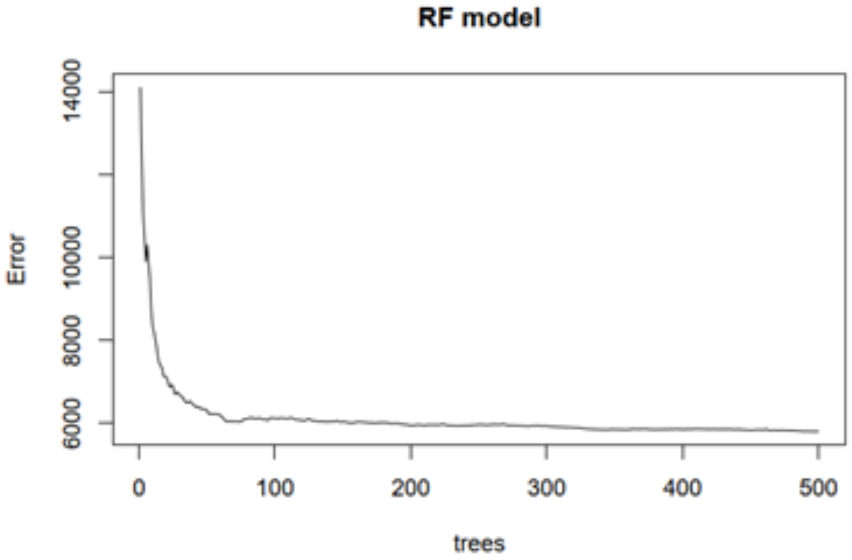
where:

- **Type of Random Forest:** Indicates whether the Random Forest model is used for regression or classification. In this case, it is a regression Random Forest, meaning it is used for predicting numeric values (Memory_Speed).
- **Number of Trees:** Specifies the number of decision trees created by the Random

Forest algorithm. Random Forest builds multiple decision trees and combines their predictions to improve the accuracy and robustness of the model. In this example, the model consists of 500 decision trees.

- **No. of Variables Tried at Each Split:** Indicates the number of predictor variables considered for each split in the decision trees. Random Forest randomly selects a subset of predictor variables at each split to create diverse trees. In this example, 4 predictor variables are tried at each split.
- **Mean of Squared Residuals:** The mean of the squared differences between the actual values and the predicted values (residuals) of the target variable (Memory_Speed). It provides a measure of the model's accuracy. A lower value indicates better performance, as it means the model's predictions are closer to the actual values. (5810.36 is a really good outcome, better than linear regression(x2)).
- **%Var:** The percentage of variance explained by the mode, the higher the %Var, the better the model.

```
plot (rf_model, main = "RF model")
```



4.5.1 Test for accuracy

```
predictions <- predict(rf_model, newdata = test_data)
mae <- mean(abs(predictions - test_data$Memory_Speed))
rmse <- sqrt(mean((predictions - test_data$Memory_Speed)^2))
rsquared <- cor(predictions, test_data$Memory_Speed)^2
rf_model
```

```
print(paste("Mean Absolute Error (MAE):", mae))
```

```
## [1] "Mean Absolute Error (MAE): 26.9471501740574"
```

```
print(paste("Root Mean Squared Error (RMSE):", rmse))
```

```
## [1] "Root Mean Squared Error (RMSE): 54.2087329273261"
```

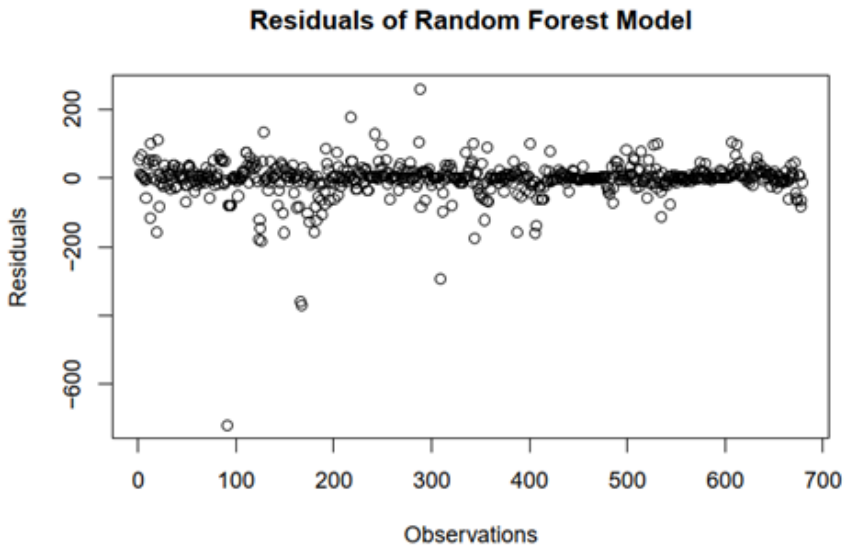
```
print(paste("R-squared:", rsquared))
```

```
## [1] "R-squared: 0.985351366265067"
```

→ This Value is really near 1, so this may be the most fit model for our dataset.

```
residuals <- test_data$Memory_Speed - predictions
plot(residuals, main = "Residuals of Random Forest Model", xlab = "Observations", ylab = "Residuals")
```


```
residuals <- test_data$Memory_Speed - predictions
plot(residuals, main = "Residuals of Random Forest Model", xlab = "Observations", ylab = "Residuals")
```



→ The ideally output is the distribution perfectly lines on the x–$axis$. Therefore, this output is acceptable.

```
results<-cbind(predictions,test_data$Memory_Speed)
colnames(results)<-c('pred','real')

results<-as.data.frame(results)
head(results, 10)
```

```
##          pred real
## 6  1045.3178 1100
## 15  348.1398  360
## 18 1233.3009 1300
## 25 1060.1806 1067
## 26  271.6730  275
## 30  800.2186  800
## 35  231.2003  225
## 38  224.1771  166
## 44  934.7422  933
## 51 1706.6015 1750
```

5 Discussion and Extension

The normality assumption is a core requirement in traditional linear regression, which assumes that the residuals adhere to a normal distribution. However, our multiple linear regression (MLR) model does not meet this assumption (Histogram of Residuals does not meet the normality assumption), raising doubts about the validity of certain inference techniques. Below, we explore the consequences and possible causes of this deviation from normality in residuals.

- **Outliers:** The presence of outliers, which are data points that significantly deviate from the rest of the dataset, can strongly affect the normality of residuals. These outliers may skew the distribution of residuals and lead to violations of the normality assumption in statistical analyses.
- **Reliability of Inference:** all Violating the normality assumption can impact the reliability of statistical inferences, such as confidence intervals and hypothesis tests for regression coefficients.
- **Prediction Interval:** While point predictions of the model may remain valid, prediction intervals could be affected, inaccurately reflecting the uncertainty of individual predictions.
- **Robustness in Larger Samples:** In larger samples, the Central Limit Theorem may mitigate the impact of non-normality, but caution is still necessary, particularly in smaller samples.
- **Potential Causes of Non-Normality:** Model misspecification or the presence of outliers may lead to non-normal residuals.
- **Data preprocessing:** Filling a large number of cells with the same repetitive value can introduce several types of errors during data preprocessing and analysis.

6 Data and code availability

Data source: data

Code source: code

References

[1] Douglas C. Montgomery, Applied Statistics and Probability for Engineers

[2] Diane Kiernan, Natural Resources Biometrics

[3] Datacamp, Linear Regression in R Tutorial

[4] R-bloggers, Random Forest in R

[5] Datadrive, A Basic Guide to Testing the Assumptions of Linear Regression in R

[6] Statistics by Jim, Multicollinearity in Regression Analysis: Problems, Detection, and Solutions

–END–