VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROBABILITY AND STATISTICS (MT2013)

Project

GRAPHIC PROCESSING UNIT

Instructor:	Mr. Nguyen Tien Dung	
Class:	CC05	
Group:	6	
Students:	Vu Ba Thai An	1852223
	Doan Mai Tuan Kiet	2052560
	Nguyen Hoang Long	1852164
	Phan Bui Tan Minh	1852578
	Tran Tien Phat	1952386

HO CHI MINH CITY, NOVEMBER 2021

Contents

Abstract

Nowadays, the emergence of digital money makes many people use their graphics cards to mine coins like Bitcoin, Ethereum, etc. Along with the COVID-19 epidemic, the demand for electronic equipment increases for many individual and organizational purposes. Those are the two main reasons for the chip supply to be severely affected, followed by the skyrocketing price. This project will study about statistics related to graphics cards as well as build some models to predict and evaluate the factors affecting the price in the future.

1 Introduction

Here are a few notes about this report:

- This is a report for the project of Probability and Statistics (MT2013) in Ho Chi Minh City University of Technology.
- Written and completed in semester HK211. For education purpose only.
- All statistical formulas and concepts in this report are referenced from [1, 2, 3].
- For calculations and comparisons as well as for building models, R is the mainly used programming language and the RStudio integrated development environment (IDE) is where the code runs. Documentation for R and RStudio is referenced from [4].
- There are some parts where we use Python instead of R for calculations because Python is very powerful for modeling and data processing. Link to Python source code here.
- All files of this project are uploaded to this Github.
- Please do not copy, modify, publish, share or transmit without permission of the author. For any further information, please feel free to contact via email address.

1.1 Dataset information

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles.

The source of the dataset we use is here [5]. This dataset has 3406 observations and 34 attributes and here is the explanation for each attribute contained in the dataset:

- `Architecture` - the name of the architecture that builds the GPU.
- `Best_Resolution` - resolution recommended by the manufacturer.
- `Boost_Clock` - the maximum clock speed that the graphics card can achieve under normal conditions before the GPU Boost is activated.
- `Core_Speed` - also known as engine clock, GPU clock speed indicates how fast the cores of a GPU are.
- `DVI_Connection` - the number of DVI (Digital Visual Interface) ports that the GPU has.
- `Dedicated` - indicates whether the GPU is separate from the CPU.

- `Direct_X` - a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms.
- `DisplayPort_Connection` - number of DP connection ports that the GPU has.
- `HDMI_Connection` - number of HDMI (High-Definition Multimedia Interface) connection ports that the GPU has.
- `Integrated` - indicates whether the GPU has integrated CPU or not.
- `L2_Cache` - like CPU, GPU also has L2 cache. The L2 cache feeds the L1 cache, which feeds the processor.
- `Manufacturer` - the name of the manufacturer of the GPU.
- `Max_Power` - the maximum amount of graphics board power that the system power supply should be able to provide to the GPU.
- `Memory` - GPU's own memory capacity.
- `Memory_Bandwidth` - the external bandwidth between a GPU and its associated system.
- `Memory_Bus` - the graphics processor is connected to the RAM (Random Access Memory) on the card via a memory bus.
- `Memory_Speed` - the memory clock is the speed of the VRAM (Video RAM) on the GPU.
- `Memory_Type` - type of graphics card memory.
- `Name` - the name of the GPU.
- `Notebook_GPU` - GPUs for laptops.
- `Open_GL` - OpenGL version supported by GPU.
- `PSU` - wattage of the power supply recommended by the manufacturer.
- `Pixel_Rate` - the number of pixels that a GPU can render onto a screen every second.
- `Power_Connector` - type of port that connects the source to the GPU.
- `Process` - GPU manufacturing process.
- `ROPs` - a hardware component in modern GPUs and one of the final steps in the rendering process of modern graphics cards.
- `Release_Date` - the date the manufacturer opens the GPU for sale.
- `Release_Price` - starting price.
- `Resolution_WxH` - maximum resolution.
- `SLI_Crossfire` - ability to support SLI(Nvidia) or Crossfire(AMD).
- `Shader` - is a type of computer program originally used for shading in 3D scenes (the production of appropriate levels of light, darkness, and color in a rendered image).
- `TMUs` - number of texture mapping units.

- `Texture_Rate` - the texture filter rate of the GPU is representative of how many pixels (specifically 'texels') the GPU can render per second.
- `VGA_Connection` - number of VGA connection ports.

1.2 R libraries

Beside that, we use a few R libraries for this project:

```
library(car)
library(dplyr)
library(ggplot2)
library(ggpubr)
library(magrittr)
library(stats)
library(VIM)
```

Details:

- `car`: This package contains mostly functions for applied regression, linear models, and generalized linear models, with an emphasis on regression diagnostics, particularly graphical diagnostic methods.
- `dplyr`: This is a grammar of data manipulation, providing a consistent set of verbs that help solve the most common data manipulation challenges.
- `ggplot2`: This is a package dedicated to data visualization.
- `ggpubr`: This is package provides some easy-to-use functions for creating and customizing `ggplot2` - based publication ready plots.
- `magrittr`: This package offers a set of operators which make code more readable.
- `stats`: This package contains functions for statistical calculations and random number generation.
- `VIM`: This package introduces new tools for the visualization of missing and/or imputed values, which can be used for exploring the data and the structure of the missing and/or imputed values.

2 Data importing

2.1 Data preprocessing

First of all, we check the "cleanliness" as well as the comparability and computable capabilities of the dataset. We check by programming with Python language.

```
import numpy as np
import pandas as pd

dataset = pd.read_csv('All_GPUs.csv')

```

```
dataset.info()
dataset.shape
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3406 entries, 0 to 3405
Data columns (total 34 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Architecture            3344 non-null   object
 1   Best_Resolution         2764 non-null   object
 2   Boost_Clock             1446 non-null   object
 3   Core_Speed              3406 non-null   object
 4   DVI_Connection          2656 non-null   float64
 5   Dedicated               3392 non-null   object
 6   Direct_X                3400 non-null   object
 7   DisplayPort_Connection  857 non-null    float64
 8   HDMI_Connection         2643 non-null   float64
 9   Integrated              3392 non-null   object
 10  L2_Cache                3406 non-null   object
 11  Manufacturer            3406 non-null   object
 12  Max_Power               2781 non-null   object
 13  Memory                  2986 non-null   object
 14  Memory_Bandwidth        3285 non-null   object
 15  Memory_Bus              3344 non-null   object
 16  Memory_Speed            3301 non-null   object
 17  Memory_Type             3350 non-null   object
 18  Name                    3406 non-null   object
 19  Notebook_GPU            3406 non-null   object
 20  Open_GL                 3366 non-null   float64
 21  PSU                     2231 non-null   object
 22  Pixel_Rate              2862 non-null   object
 23  Power_Connector         2697 non-null   object
 24  Process                 2943 non-null   object
 25  ROPs                    2868 non-null   object
 26  Release_Date            3406 non-null   object
 27  Release_Price           556 non-null    object
 28  Resolution_WxH          3211 non-null   object
 29  SLI_Crossfire           3406 non-null   object
 30  Shader                  3299 non-null   float64
 31  TMUs                    2868 non-null   float64
 32  Texture_Rate            2862 non-null   object
 33  VGA_Connection          2648 non-null   float64
dtypes: float64(7), object(27)
memory usage: 904.8+ KB
(3406, 34)
```

The dataset has 3406 observations and 34 variables. However, only 7 variables can be used for the calculation. Therefore, we decided to reprocess the dataset to make it more manipulable. Hence, the actual dataset used in this project can be viewed from here. After a few processing

steps, there are only 3365 observations left.

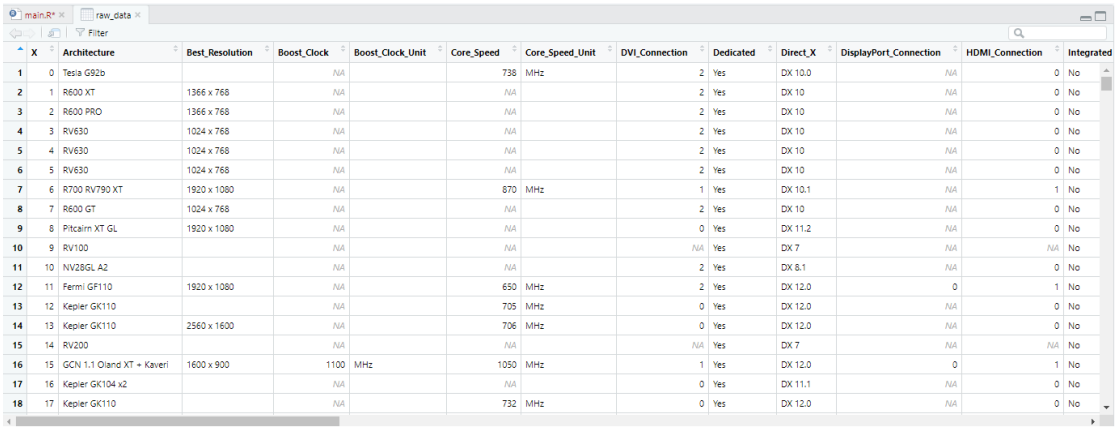
2.2 Importing to RStudio

	Architecture	Best_Resolution	Boost_Clock	Boost_Clock_Unit	Core_Speed	Core_Speed_Unit	DVI_Connection	Dedicated	...
0	Tesla G92b				738	MHz	2	Yes	...
1	R600 XT	1366 x 768					2	Yes	...
2	R600 PRO	1366 x 768					2	Yes	...
3	RV630	1024 x 768					2	Yes	...
4	RV630	1024 x 768					2	Yes	...
5	RV630	1024 x 768					2	Yes	...
...	...	...	...	...	...	...	...	...	...

File data.csv

To import data, we must put the data file in the same directory as the project directory. Then use the `read.csv()` function to read the .csv file. We use the `View()` function to check that data has been imported.

```
raw_data = read.csv('data.csv')
View(raw_data)
```

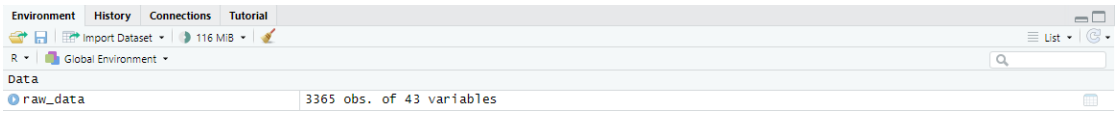


	X	Architecture	Best_Resolution	Boost_Clock	Boost_Clock_Unit	Core_Speed	Core_Speed_Unit	DVI_Connection	Dedicated	Direct_X	DisplayPort_Connection	HDMI_Connection	Integrated
1	0	Tesla G92b		NA		738	MHz	2	Yes	DX 10.0	NA	0	No
2	1	R600 XT	1366 x 768	NA		NA		2	Yes	DX 10	NA	0	No
3	2	R600 PRO	1366 x 768	NA		NA		2	Yes	DX 10	NA	0	No
4	3	RV630	1024 x 768	NA		NA		2	Yes	DX 10	NA	0	No
5	4	RV630	1024 x 768	NA		NA		2	Yes	DX 10	NA	0	No
6	5	RV630	1024 x 768	NA		NA		2	Yes	DX 10	NA	0	No
7	6	R700 RV790 XT	1920 x 1080	NA		870	MHz	1	Yes	DX 10.1	NA	1	No
8	7	R600 GT	1024 x 768	NA		NA		2	Yes	DX 10	NA	0	No
9	8	Pitcairn XT GL	1920 x 1080	NA		NA		0	Yes	DX 11.2	NA	0	No
10	9	RV100		NA		NA		NA	Yes	DX 7	NA	NA	No
11	10	NV28GL A2		NA		NA		2	Yes	DX 8.1	NA	0	No
12	11	Fermi GF110	1920 x 1080	NA		650	MHz	2	Yes	DX 12.0	0	1	No
13	12	Kepler GK110		NA		705	MHz	0	Yes	DX 12.0	NA	0	No
14	13	Kepler GK110	2560 x 1600	NA		706	MHz	0	Yes	DX 12.0	NA	0	No
15	14	RV200		NA		NA		NA	Yes	DX 7	NA	NA	No
16	15	GCN 1.1 Oland XT + Kaveri	1600 x 900	1100	MHz	1050	MHz	1	Yes	DX 12.0	0	1	No
17	16	Kepler GK104 x2		NA		NA		0	Yes	DX 11.1	NA	0	No
18	17	Kepler GK110		NA		732	MHz	0	Yes	DX 12.0	NA	0	No

3 Data cleaning

"Cleaning" refers to the process of removing invalid data points from a dataset.
Data cleaning is the process of fixing or removing incorrect, corrupted, incorrectly formatted, duplicate, or incomplete data within a dataset. When combining multiple data sources, there are many opportunities for data to be duplicated or mislabeled. If data is incorrect, outcomes and algorithms are unreliable, even though they may look correct. There is no one absolute way to prescribe the exact steps in the data cleaning process because the processes will vary from dataset to dataset.

3.1 Data extracting

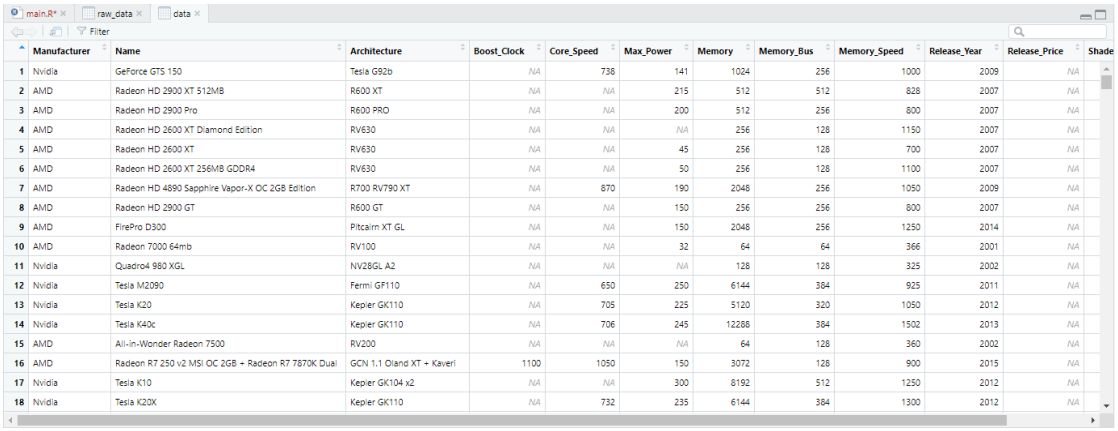


The dataset has 3365 observation of 43 variables. However, since most of the properties are in object form, we extract 13 variables:

- `Manufacturer`
- `Name`
- `Architecture`
- `Boost_Clock`
- `Core_Speed`
- `Max_Power`
- `Memory`
- `Memory_Bus`
- `Memory_Speed`
- `Release_Year`
- `Release_Price`
- `Shader`
- `TMUs`

from the data frame `raw_data`. Then we create a new data frame `data` by using `select()` function. The `select()` function in `dplyr` package helps us to select the columns based on conditions which are column names of the data frame.

```
data = raw_data %>%
select('Manufacturer','Name','Architecture','Boost_Clock','Core_Speed',
'Max_Power','Memory','Memory_Bus', 'Memory_Speed', 'Release_Year',
'Release_Price','Shader','TMUs')
View(data)
```



	Manufacturer	Name	Architecture	Boost_Clock	Core_Speed	Max_Power	Memory	Memory_Bus	Memory_Speed	Release_Year	Release_Price	Shade
1	Nvidia	GeForce GTS 150	Tesla G92b	NA	738	141	1024	256	1000	2009	NA	
2	AMD	Radeon HD 2900 XT 512MB	R600 XT	NA	NA	215	512	512	828	2007	NA	
3	AMD	Radeon HD 2900 Pro	R600 PRO	NA	NA	200	512	256	800	2007	NA	
4	AMD	Radeon HD 2600 XT Diamond Edition	RV630	NA	NA	NA	256	128	1150	2007	NA	
5	AMD	Radeon HD 2600 XT	RV630	NA	NA	45	256	128	700	2007	NA	
6	AMD	Radeon HD 2600 XT 256MB GDDR4	RV630	NA	NA	50	256	128	1100	2007	NA	
7	AMD	Radeon HD 4890 Sapphire Vapor-X OC 2GB Edition	R700 RV790 XT	NA	870	190	2048	256	1050	2009	NA	
8	AMD	Radeon HD 2900 GT	R600 GT	NA	NA	150	256	256	800	2007	NA	
9	AMD	FirePro D300	Pitcairn XT GL	NA	NA	150	2048	256	1250	2014	NA	
10	AMD	Radeon 7000 64mb	RV100	NA	NA	32	64	64	366	2001	NA	
11	Nvidia	Quadro4 980 XGL	NV28GL A2	NA	NA	NA	128	128	325	2002	NA	
12	Nvidia	Tesla M2090	Fermi GF110	NA	650	250	6144	384	925	2011	NA	
13	Nvidia	Tesla K20	Kepler GK110	NA	705	225	5120	320	1050	2012	NA	
14	Nvidia	Tesla K40c	Kepler GK110	NA	706	245	12288	384	1502	2013	NA	
15	AMD	All-in-Wonder Radeon 7500	RV200	NA	NA	NA	64	128	360	2002	NA	
16	AMD	Radeon R7 250 v2 MSI OC 2GB + Radeon R7 7870K Dual	GCN 1.1 Oland XT + Kaveri	1100	1050	150	3072	128	900	2015	NA	
17	Nvidia	Tesla K10	Kepler GK104 x2	NA	NA	300	8192	512	1250	2012	NA	
18	Nvidia	Tesla K20X	Kepler GK110	NA	732	235	6144	384	1300	2012	NA	

As shown above we can see that there are places where the value is `NA` (meaning not available). We will address this problem in the next section.

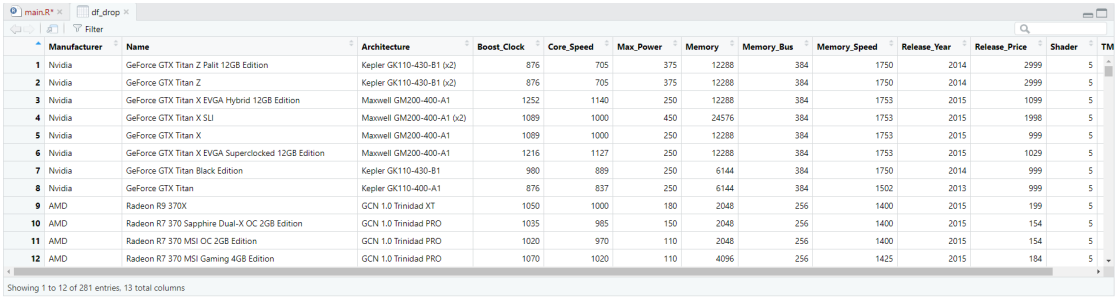
3.2 Not available data checking

As mentioned above, now we have to check `NA` data in the data frame `data`. If there are any `NA` data, find a solution for these `NA` data.
There are many ways to handle `NA` values.

3.2.1 Dropping

The first method we use here is to eliminate observations which each has at least one `NA` variable to get a new data frame `df_drop` by using `filter()` function combining with `is.na()` function. While `is.na()` fuction indicates which element is `NA` data, the `filter()` function returns rows where conditions are true. Therefore, using `OR` operator helps us eliminate `NA` data. We also use the `summary()` function to summarize.

```
1 df_drop = data %>% filter(!( is.na(Manufacturer) | is.na(Name) |
  is.na(Architecture) | is.na(Boost_Clock) | is.na(Core_Speed) |
  is.na(Max_Power) | is.na(Memory) | is.na(Memory_Bus) | is.na(Memory_Speed)
  | is.na(Release_Year) | is.na(Release_Price) | is.na(Shader) |
  is.na(TMUs)))
2 View(df_drop)
3 summary(df_drop)
```



	Manufacturer	Name	Architecture	Boost_Clock	Core_Speed	Max_Power	Memory	Memory_Bus	Memory_Speed	Release_Year	Release_Price	Shader	TM
1	Nvidia	GeForce GTX Titan Z Palit 12GB Edition	Kepler GK110-430-B1 (x2)	876	705	375	12288	384	1750	2014	2999	5	
2	Nvidia	GeForce GTX Titan Z	Kepler GK110-430-B1 (x2)	876	705	375	12288	384	1750	2014	2999	5	
3	Nvidia	GeForce GTX Titan X EVGA Hybrid 12GB Edition	Maxwell GM200-400-A1	1252	1140	250	12288	384	1753	2015	1099	5	
4	Nvidia	GeForce GTX Titan X SLI	Maxwell GM200-400-A1 (x2)	1089	1000	450	24576	384	1753	2015	1998	5	
5	Nvidia	GeForce GTX Titan X	Maxwell GM200-400-A1	1089	1000	250	12288	384	1753	2015	999	5	
6	Nvidia	GeForce GTX Titan X EVGA Superclocked 12GB Edition	Maxwell GM200-400-A1	1216	1127	250	12288	384	1753	2015	1029	5	
7	Nvidia	GeForce GTX Titan Black Edition	Kepler GK110-430-B1	980	889	250	6144	384	1750	2014	999	5	
8	Nvidia	GeForce GTX Titan	Kepler GK110-400-A1	876	837	250	6144	384	1502	2013	999	5	
9	AMD	Radeon R9 370X	GCN 1.0 Trinidad XT	1050	1000	180	2048	256	1400	2015	199	5	
10	AMD	Radeon R7 370 Sapphire Dual-X OC 2GB Edition	GCN 1.0 Trinidad PRO	1035	985	150	2048	256	1400	2015	154	5	
11	AMD	Radeon R7 370 MSI OC 2GB Edition	GCN 1.0 Trinidad PRO	1020	970	110	2048	256	1400	2015	154	5	
12	AMD	Radeon R7 370 MSI Gaming 4GB Edition	GCN 1.0 Trinidad PRO	1070	1020	110	4096	256	1425	2015	184	5	

Showing 1 to 12 of 281 entries, 13 total columns

```
> summary(df_drop)
 Manufacturer          Name           Architecture        Boost_Clock     Core_Speed     Max_Power         Memory        Memory_Bus     Memory_Speed   Release_Year
 Length:281         Length:281         Length:281         Min.   : 780   Min.   : 705   Min.   :  30.0   Min.   : 1024   Min.   :  88.0   Min.   : 900   Min.   :2012
 Class :character   Class :character   Class :character   1st Qu.:1178   1st Qu.:1020   1st Qu.: 120.0   1st Qu.: 4096   1st Qu.: 128.0   1st Qu.:1650   1st Qu.:2015
 Mode  :character   Mode  :character   Mode  :character   Median :1270   Median :1120   Median : 150.0   Median : 4096   Median : 256.0   Median :1752   Median :2016
                                                          Mean   :1342   Mean   :1207   Mean   : 162.9   Mean   : 5517   Mean   : 233.8   Mean   :1713   Mean   :2015
                                                          3rd Qu.:1518   3rd Qu.:1404   3rd Qu.: 180.0   3rd Qu.: 8192   3rd Qu.: 256.0   3rd Qu.:2000   3rd Qu.:2016
                                                          Max.   :1936   Max.   :1784   Max.   : 500.0   Max.   :24576   Max.   : 384.0   Max.   :2127   Max.   :2017

 Release_Price      Shader          TMUs
 Min.   :  69   Min.   :   5   Min.   :  20.0
 1st Qu.: 199   1st Qu.:   5   1st Qu.:  64.0
 Median : 279   Median :   5   Median : 120.0
 Mean   : 406   Mean   :  55   Mean   : 110.2
 3rd Qu.: 579   3rd Qu.:   5   3rd Qu.: 144.0
 Max.   :2999   Max.   :1000   Max.   : 240.0
> |
```

After filtering, we see that only 281 observations left, compared to the original 3365 observations. That means 3365 - 281 = 3084 observations have at least one `NA` value. And this is not good at all because the very sharp decrease in the number of observations ($281 \ll 3365$) can lead to skewed results. Thus, we will find another way to handle it.

3.2.2 Replacing with column mean

Another method to handle that is to replace the `NA` values present in the data set with the mean. We use the `mutate_all()` function to change these `NA` values by the corresponding mean according to the column that contains them. If a value is NA, it will be replaced with the mean. Otherwise, it will stay the same.

```
1 df_mean = data %>% mutate_all(~ifelse(is.na(.x), mean(.x, na.rm = TRUE),
  .x))
2 View(df_mean)
3 summary(df_mean)
```

main.R* × | df_mean × | df_drop ×

Filter

	Manufacturer	Name	Architecture	Boost_Clock	Core_Speed	Max_Power	Memory	Memory_Bus	Memory_Speed	Release_Year	Release_Price	Shader	TMUs
1	Nvidia	GeForce GTS 150	Tesla G92b	1201.236	738.0000	141.0000	1024.000	256	1000.000	2009	374.5521	4.000000	64.0000
2	AMD	Radeon HD 2900 XT 512MB	R600 XT	1201.236	942.3062	215.0000	512.000	512	828.000	2007	374.5521	4.000000	16.0000
3	AMD	Radeon HD 2900 Pro	R600 PRO	1201.236	942.3062	200.0000	512.000	256	800.000	2007	374.5521	4.000000	16.0000
4	AMD	Radeon HD 2600 XT Diamond Edition	RV630	1201.236	942.3062	126.0102	256.000	128	1150.000	2007	374.5521	4.000000	8.0000
5	AMD	Radeon HD 2600 XT	RV630	1201.236	942.3062	45.0000	256.000	128	700.000	2007	374.5521	4.000000	8.0000
6	AMD	Radeon HD 2600 XT 256MB GDDR4	RV630	1201.236	942.3062	50.0000	256.000	128	1100.000	2007	374.5521	4.000000	8.0000
7	AMD	Radeon HD 4890 Sapphire Vapor-X OC 2GB Edition	R700 RV790 XT	1201.236	870.0000	190.0000	2048.000	256	1050.000	2009	374.5521	4.100000	40.0000
8	AMD	Radeon HD 2900 GT	R600 GT	1201.236	942.3062	150.0000	256.000	256	800.000	2007	374.5521	4.000000	12.0000
9	AMD	FirePro D300	Pitcairn XT GL	1201.236	942.3062	150.0000	2048.000	256	1250.000	2014	374.5521	5.000000	80.0000
10	AMD	Radeon 7000 64mb	RV100	1201.236	942.3062	32.0000	64.000	64	366.000	2001	374.5521	1.000000	69.4007
11	Nvidia	Quadro4 980 XGL	NV28GL A2	1201.236	942.3062	126.0102	128.000	128	325.000	2002	374.5521	1.300000	8.0000
12	Nvidia	Tesla M2090	Fermi GF110	1201.236	650.0000	250.0000	6144.000	384	925.000	2011	374.5521	5.000000	56.0000

Showing 1 to 12 of 3,365 entries, 13 total columns

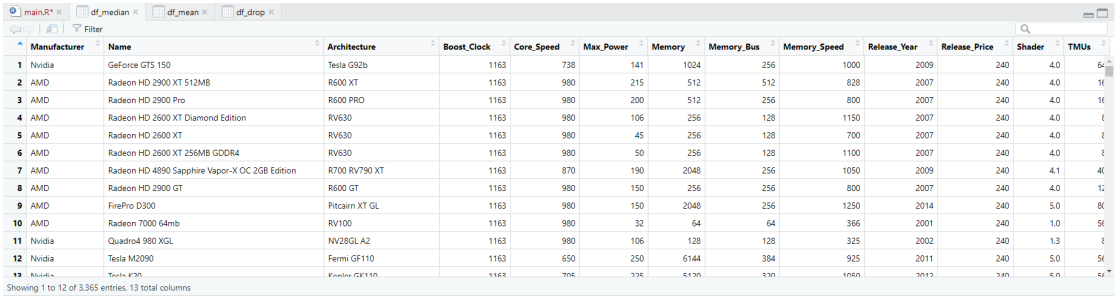
```
> summary(df_mean)
 Manufacturer          Name           Architecture        Boost_Clock     Core_Speed     Max_Power         Memory        Memory_Bus     Memory_Speed   Release_Year
 Length:3365        Length:3365        Length:3365        Min.   : 400   Min.   : 100.0   Min.   :   1   Min.   :   16   Min.   :  32.0   Min.   : 100   Min.   :1998
 Class :character   Class :character   Class :character   1st Qu.:1201   1st Qu.: 850.0   1st Qu.:  60   1st Qu.: 1024   1st Qu.: 128.0   1st Qu.: 825   1st Qu.:2010
 Mode  :character   Mode  :character   Mode  :character   Median :1201   Median : 942.3   Median : 126   Median : 2048   Median : 128.0   Median :1170   Median :2012
                                                          Mean   :1201   Mean   : 942.3   Mean   : 126   Mean   : 2865   Mean   : 205.8   Mean   :1170   Mean   :2012
                                                          3rd Qu.:1201   3rd Qu.:1040.0   3rd Qu.: 150   3rd Qu.: 4096   3rd Qu.: 256.0   3rd Qu.:1502   3rd Qu.:2014
                                                          Max.   :1936   Max.   :1784.0   Max.   : 780   Max.   :32000   Max.   :8192.0   Max.   :2127   Max.   :2017

 Release_Price        Shader           TMUs
 Min.   :  23.0   Min.   :   1.000   Min.   :  1.0
 1st Qu.: 374.6   1st Qu.:   5.000   1st Qu.: 32.0
 Median : 374.6   Median :   5.000   Median : 64.0
 Mean   : 374.6   Mean   :  14.703   Mean   : 69.4
 3rd Qu.: 374.6   3rd Qu.:   5.000   3rd Qu.: 80.0
 Max.   :14999.0   Max.   :1000.000   Max.   :384.0
> |
```

The result this time is very good because we can both keep the number of observations in the data set and deal with the `NA` values.

3.2.3 Replacing with column median

In this section we will do the same as above but change the mean to the median.

```r
df_median = data %>% mutate_all(~ifelse(is.na(.x), median(.x, na.rm = TRUE), .x))
View(df_median)
summary(df_median)
```



	Manufacturer	Name	Architecture	Boost_Clock	Core_Speed	Max_Power	Memory	Memory_Bus	Memory_Speed	Release_Year	Release_Price	Shader	TMUs
1	Nvidia	GeForce GTS 150	Tesla G92b	1163	738	141	1024	256	1000	2009	240	4.0	64
2	AMD	Radeon HD 2900 XT 512MB	R600 XT	1163	980	215	512	512	828	2007	240	4.0	16
3	AMD	Radeon HD 2900 Pro	R600 PRO	1163	980	200	512	256	800	2007	240	4.0	16
4	AMD	Radeon HD 2600 XT Diamond Edition	RV630	1163	980	106	256	128	1150	2007	240	4.0	8
5	AMD	Radeon HD 2600 XT	RV630	1163	980	45	256	128	700	2007	240	4.0	8
6	AMD	Radeon HD 2600 XT 256MB GDDR4	RV630	1163	980	50	256	128	1100	2007	240	4.0	8
7	AMD	Radeon HD 4890 Sapphire Vapor-X OC 2GB Edition	R700 RV790 XT	1163	870	190	2048	256	1050	2009	240	4.1	40
8	AMD	Radeon HD 2900 GT	R600 GT	1163	980	150	256	256	800	2007	240	4.0	12
9	AMD	FirePro D300	Pitcairn XT GL	1163	980	150	2048	256	1250	2014	240	5.0	80
10	AMD	Radeon 7000 64mb	RV100	1163	980	32	64	64	366	2001	240	1.0	56
11	Nvidia	Quadro4 980 XGL	NV28GL A2	1163	980	106	128	128	325	2002	240	1.3	8
12	Nvidia	Tesla M2090	Fermi GF110	1163	650	250	6144	384	925	2011	240	5.0	56

Showing 1 to 12 of 3,365 entries, 13 total columns

```
> summary(df_median)
 Manufacturer          Name           Architecture        Boost_Clock    Core_Speed      Max_Power         Memory         Memory_Bus     Memory_Speed   Release_Year
 Length:3365        Length:3365        Length:3365        Min.   : 400   Min.   : 100.0   Min.   :  1.0   Min.   :   16   Min.   :  32.0   Min.   : 100   Min.   :1998
 Class :character   Class :character   Class :character   1st Qu.:1163   1st Qu.: 850.0   1st Qu.: 60.0   1st Qu.: 1024   1st Qu.: 128.0   1st Qu.: 825   1st Qu.:2010
 Mode  :character   Mode  :character   Mode  :character   Median :1163   Median : 980.0   Median :106.0   Median : 2048   Median : 128.0   Median :1150   Median :2012
                                                          Mean   :1179   Mean   : 952.7   Mean   :122.3   Mean   : 2763   Mean   : 204.4   Mean   :1170   Mean   :2012
                                                          3rd Qu.:1163   3rd Qu.:1040.0   3rd Qu.:150.0   3rd Qu.: 4096   3rd Qu.: 256.0   3rd Qu.:1502   3rd Qu.:2014
                                                          Max.   :1936   Max.   :1784.0   Max.   :780.0   Max.   :32000   Max.   :8192.0   Max.   :2127   Max.   :2017

 Release_Price        Shader           TMUs
 Min.   :   23.0   Min.   :1.000   Min.   :  1.00
 1st Qu.:  240.0   1st Qu.:5.000   1st Qu.: 32.00
 Median :  240.0   Median :5.000   Median : 56.00
 Mean   :  261.9   Mean   :4.713   Mean   : 67.27
 3rd Qu.:  240.0   3rd Qu.:5.000   3rd Qu.: 80.00
 Max.   :14999.0   Max.   :5.000   Max.   :384.00
>
```

3.2.4 Replacing with k-NN

3.2.4.a Fast introduction to k-NN

A popular approach to missing data imputation is to use a model to predict the missing values.

This requires a model to be created for each input variable that has missing values. Although any one among a range of different models can be used to predict the missing values, the k-nearest neighbor (k-NN) algorithm has proven to be generally effective, often referred to as "nearest neighbor imputation". It can be used for data that are continuous, discrete, ordinal and categorical which makes it particularly useful for dealing with all kind of missing data.

The idea in k-NN methods is to identify k samples in the dataset that are similar or close in the space. Then we use these k samples to estimate the value of the missing data points. Each sample's missing values are imputed using the mean value of the k-neighbors found in the dataset. The assumption behind using k-NN for missing values is that a point value can be approximated by the values of the points that are closest to it, based on other variables.

The nearest, most similar, neighbours are found by minimising a distance function, usually the *Euclidean distance* [1], defined as [6]:

$$E(a,b) = \sqrt{\sum_{i \in D} (x_{ai} - x_{bi})^2}$$

where:

[1] There are also some other methods such as Hamming distance, Manhattan distance and Minkowski distance.

- $E(a, b)$ is the distance between the two cases a and b.
- x_{ai} and x_{bi} are the values of attribute i in cases a and b respectively.
- D is the set of attributes with non-missing values in both cases.

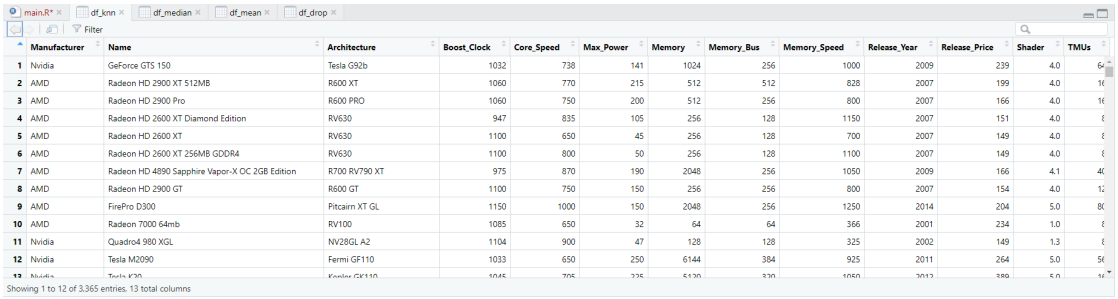
k-NN algorithm has the following basic steps:

1. Calculate distance.
2. Find closest neighbors.
3. Vote for labels.

3.2.4.b Applying to dataset

For our dataset, we choose k (number of nearest neighbours used) = square root of the number of observations present in the dataset (3365).

```
df_knn <- kNN(data, k = sqrt(nrow(data)), imp_var = FALSE)
View(df_knn)
summary(df_knn)
```



	Manufacturer	Name	Architecture	Boost_Clock	Core_Speed	Max_Power	Memory	Memory_Bus	Memory_Speed	Release_Year	Release_Price	Shader	TMUs
1	Nvidia	GeForce GTS 150	Tesla G92b	1032	738	141	1024	256	1000	2009	239	4.0	64
2	AMD	Radeon HD 2900 XT 512MB	R600 XT	1060	770	215	512	512	828	2007	199	4.0	16
3	AMD	Radeon HD 2900 Pro	R600 PRO	1060	750	200	512	256	800	2007	166	4.0	16
4	AMD	Radeon HD 2600 XT Diamond Edition	RV630	947	835	105	256	128	1150	2007	151	4.0	8
5	AMD	Radeon HD 2600 XT	RV630	1100	650	45	256	128	700	2007	149	4.0	8
6	AMD	Radeon HD 2600 XT 256MB GDDR4	RV630	1100	800	50	256	128	1100	2007	149	4.0	8
7	AMD	Radeon HD 4890 Sapphire Vapor-X OC 2GB Edition	R700 RV790 XT	975	870	190	2048	256	1050	2009	166	4.1	40
8	AMD	Radeon HD 2900 GT	R600 GT	1100	750	150	256	256	800	2007	154	4.0	12
9	AMD	FirePro D300	Pitcairn XT GL	1150	1000	150	2048	256	1250	2014	204	5.0	80
10	AMD	Radeon 7000 64mb	RV100	1085	650	32	64	64	366	2001	234	1.0	8
11	Nvidia	Quadro4 980 XGL	NV28GL A2	1104	900	47	128	128	325	2002	149	1.3	8
12	Nvidia	Tesla M2090	Fermi GF110	1033	650	250	6144	384	925	2011	264	5.0	56

Showing 1 to 12 of 3,365 entries, 13 total columns

```
> summary(df_knn)
 Manufacturer          Name           Architecture        Boost_Clock     Core_Speed      Max_Power         Memory         Memory_Bus     Memory_Speed   Release_Year
 Length:3365        Length:3365        Length:3365        Min.   : 400   Min.   : 100.0   Min.   :  1.0   Min.   :   16   Min.   :  32.0   Min.   : 100   Min.   :1998
 Class :character   Class :character   Class :character   1st Qu.:1035   1st Qu.: 700.0   1st Qu.: 45.0   1st Qu.: 1024   1st Qu.: 128.0   1st Qu.: 800   1st Qu.:2010
 Mode  :character   Mode  :character   Mode  :character   Median :1085   Median : 900.0   Median :100.0   Median : 2048   Median : 128.0   Median :1125   Median :2012
                                                          Mean   :1115   Mean   : 879.5   Mean   :120.2   Mean   : 2634   Mean   : 204.4   Mean   :1154   Mean   :2012
                                                          3rd Qu.:1150   3rd Qu.:1050.0   3rd Qu.:170.0   3rd Qu.: 4096   3rd Qu.: 256.0   3rd Qu.:1502   3rd Qu.:2014
                                                          Max.   :1936   Max.   :1784.0   Max.   :780.0   Max.   :32000   Max.   :8192.0   Max.   :2127   Max.   :2017

 Release_Price        Shader           TMUs
 Min.   :   23.0   Min.   :1.000   Min.   :  1.00
 1st Qu.:  149.0   1st Qu.:5.000   1st Qu.: 16.00
 Median :  200.0   Median :5.000   Median : 48.00
 Mean   :  239.8   Mean   :4.685   Mean   : 60.15
 3rd Qu.:  249.0   3rd Qu.:5.000   3rd Qu.: 80.00
 Max.   :14999.0   Max.   :5.000   Max.   :384.00
> |
```

In our project, we will use this data frame `df_knn`. And from this section onwards, the processing data frame is named `df`.

```
df = df_knn
```

The reason we chose k-NN is because this algorithm can compete with the most accurate models as it gives highly accurate predictions.

The k-NN algorithm is a type of lazy learning, where the computation for the generation of the predictions is deferred until classification. Although this method increases the costs of computation compared to other algorithms, k-NN is still the better choice for applications where predictions are not requested frequently but where accuracy is important.

4 Data visualization

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.

4.1 Transformation

Logarithm transformation [7] is a data transformation method in which it replaces each variable x with a $log(x)$. The choice of the logarithm base is usually left up to the analyst and it would depend on the purposes of statistical modeling.

When our original continuous data do not follow the bell curve, we can log transform this data to make it as "normal" as possible so that the statistical analysis results from this data become more valid.

Therefore, we decided to convert the data to a logarithmic equation of the form:

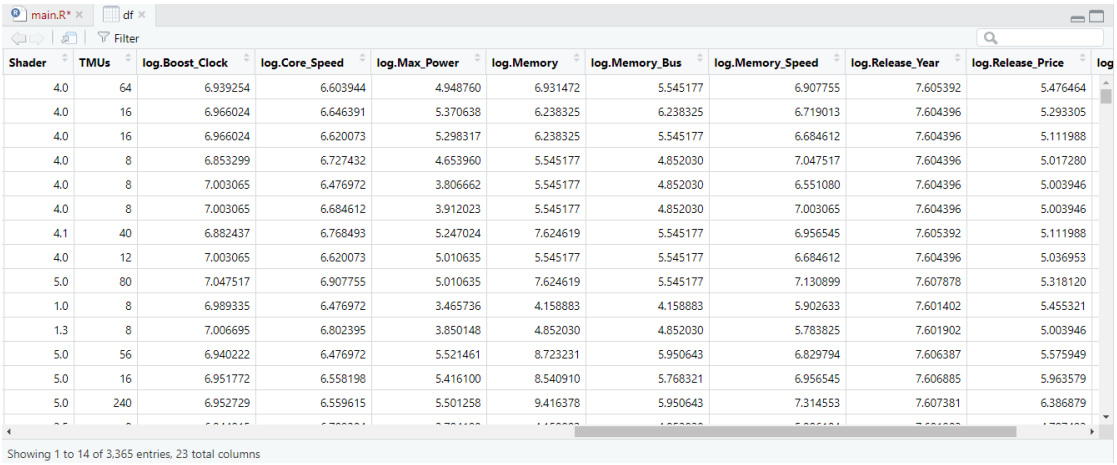
$$f(x) = log(x)$$

where:

- x: Values in our dataset.

We create new variable columns corresponding to the variable columns present in the dataset. And then add them in the data frame `df`. We use `log()` function:

```
df['log.Boost_Clock'] <- log(df['Boost_Clock'])
df['log.Core_Speed'] <- log(df['Core_Speed'])
df['log.Max_Power'] <- log(df['Max_Power'])
df['log.Memory'] <- log(df['Memory'])
df['log.Memory_Bus'] <- log(df['Memory_Bus'])
df['log.Memory_Speed'] <- log(df['Memory_Speed'])
df['log.Release_Year'] <- log(df['Release_Year'])
df['log.Release_Price'] <- log(df['Release_Price'])
df['log.Shader'] <- log(df['Shader'])
df['log.TMUs'] <- log(df['TMUs'])
View(df)
```



main.R* × df ×

Filter

Shader	TMUs	log.Boost_Clock	log.Core_Speed	log.Max_Power	log.Memory	log.Memory_Bus	log.Memory_Speed	log.Release_Year	log.Release_Price	log
4.0	64	6.939254	6.603944	4.948760	6.931472	5.545177	6.907755	7.605392	5.476464	
4.0	16	6.966024	6.646391	5.370638	6.238325	6.238325	6.719013	7.604396	5.293305	
4.0	16	6.966024	6.620073	5.298317	6.238325	5.545177	6.684612	7.604396	5.111988	
4.0	8	6.853299	6.727432	4.653960	5.545177	4.852030	7.047517	7.604396	5.017280	
4.0	8	7.003065	6.476972	3.806662	5.545177	4.852030	6.551080	7.604396	5.003946	
4.0	8	7.003065	6.684612	3.912023	5.545177	4.852030	7.003065	7.604396	5.003946	
4.1	40	6.882437	6.768493	5.247024	7.624619	5.545177	6.956545	7.605392	5.111988	
4.0	12	7.003065	6.620073	5.010635	5.545177	5.545177	6.684612	7.604396	5.036953	
5.0	80	7.047517	6.907755	5.010635	7.624619	5.545177	7.130899	7.607878	5.318120	
1.0	8	6.989335	6.476972	3.465736	4.158883	4.158883	5.902633	7.601402	5.455321	
1.3	8	7.006695	6.802395	3.850148	4.852030	4.852030	5.783825	7.601902	5.003946	
5.0	56	6.940222	6.476972	5.521461	8.723231	5.950643	6.829794	7.606387	5.575949	
5.0	16	6.951772	6.558198	5.416100	8.540910	5.768321	6.956545	7.606885	5.963579	
5.0	240	6.952729	6.559615	5.501258	9.416378	5.950643	7.314553	7.607381	6.386879	

Showing 1 to 14 of 3,365 entries, 23 total columns

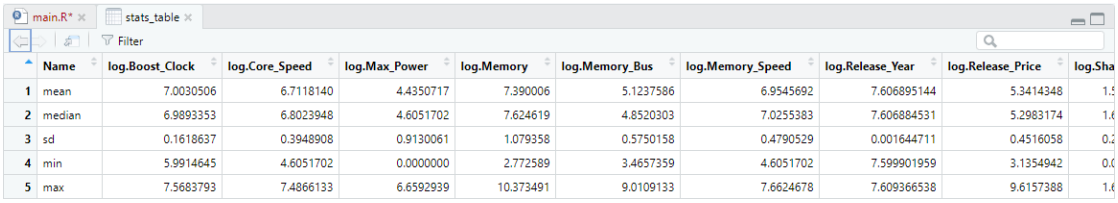
And from this step onwards, we will use logarithmic values for calculations as well as comparisons.

4.2 Descriptive statistics for each of the variables

For this section, we make a new data frame named `stats_table` to calculate mean, max, min, median and standard deviation for each variable.

```
stats_table = data.frame(
  Name = c('mean', 'median', 'sd', 'min', 'max'),
  log.Boost_Clock = c(mean(df[, 'log.Boost_Clock']), median(df[,
  'log.Boost_Clock']), sd(df[, 'log.Boost_Clock']), min(df[,
  'log.Boost_Clock']), max(df[, 'log.Boost_Clock'])),
  log.Core_Speed = c(mean(df[, 'log.Core_Speed']), median(df[,
  'log.Core_Speed']), sd(df[, 'log.Core_Speed']), min(df[,
  'log.Core_Speed']), max(df[, 'log.Core_Speed'])),
  log.Max_Power = c(mean(df[, 'log.Max_Power']), median(df[,
  'log.Max_Power']), sd(df[, 'log.Max_Power']), min(df[, 'log.Max_Power']),
  max(df[, 'log.Max_Power'])),
  log.Memory = c(mean(df[, 'log.Memory']), median(df[, 'log.Memory']),
  sd(df[, 'log.Memory']), min(df[, 'log.Memory']), max(df[,
  'log.Memory'])),
  log.Memory_Bus = c(mean(df[, 'log.Memory_Bus']), median(df[,
  'log.Memory_Bus']), sd(df[, 'log.Memory_Bus']), min(df[,
  'log.Memory_Bus']), max(df[, 'log.Memory_Bus'])),
  log.Memory_Speed = c(mean(df[, 'log.Memory_Speed']), median(df[,
  'log.Memory_Speed']), sd(df[, 'log.Memory_Speed']), min(df[,
  'log.Memory_Speed']), max(df[, 'log.Memory_Speed'])),
  log.Release_Year = c(mean(df[, 'log.Release_Year']), median(df[,
  'log.Release_Year']), sd(df[, 'log.Release_Year']), min(df[,
  'log.Release_Year']), max(df[, 'log.Release_Year'])),
  log.Release_Price = c(mean(df[, 'log.Release_Price']), median(df[,
  'log.Release_Price']), sd(df[, 'log.Release_Price']), min(df[,
  'log.Release_Price']), max(df[, 'log.Release_Price'])),
```

```
    log.Shader = c(mean(df[, 'log.Shader']), median(df[, 'log.Shader']),
    sd(df[, 'log.Shader']), min(df[, 'log.Shader']), max(df[,
    'log.Shader'])),
    log.TMUs = c(mean(df[, 'log.TMUs']), median(df[, 'log.TMUs']), sd(df[,
    'log.TMUs']), min(df[, 'log.TMUs']), max(df[, 'log.TMUs']))
)
View(stats_table)
```



	Name	log.Boost_Clock	log.Core_Speed	log.Max_Power	log.Memory	log.Memory_Bus	log.Memory_Speed	log.Release_Year	log.Release_Price	log.Sha
1	mean	7.0030506	6.7118140	4.4350717	7.390006	5.1237586	6.9545692	7.606895144	5.3414348	1.5
2	median	6.9893353	6.8023948	4.6051702	7.624619	4.8520303	7.0255383	7.606884531	5.2983174	1.6
3	sd	0.1618637	0.3948908	0.9130061	1.079358	0.5750158	0.4790529	0.001644711	0.4516058	0.2
4	min	5.9914645	4.6051702	0.0000000	2.772589	3.4657359	4.6051702	7.599901959	3.1354942	0.0
5	max	7.5683793	7.4866133	6.6592939	10.373491	9.0109133	7.6624678	7.609366538	9.6157388	1.6

We can also try to create a frequency table for `Shader` based on what we observed.

```
ftable(df[ ,'Shader'])
```

```
1   1.1   1.3   1.4     2   2.5     3     4   4.1 4.55     5
1     4    14     9    59     2   174   273   170     1  2658
```

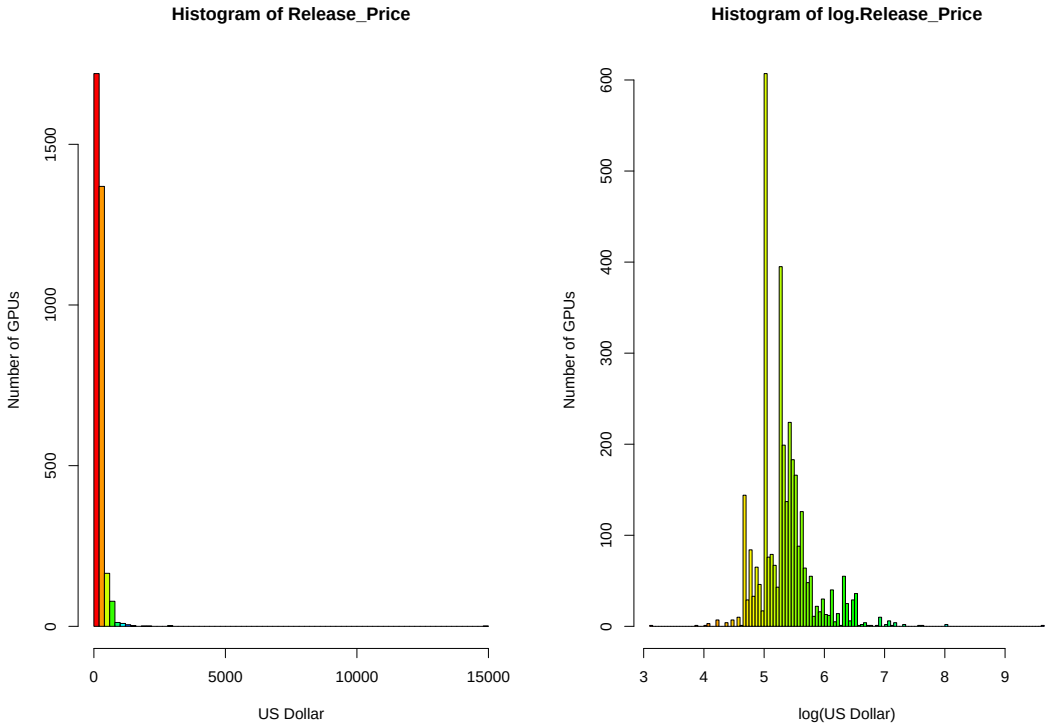
4.3 Graphs

Note that all graphs in this report are in pdf format. So, if the letters or numbers or symbols are too small, or the chart is blurry, please zoom in for a better view.

4.3.1 Hist

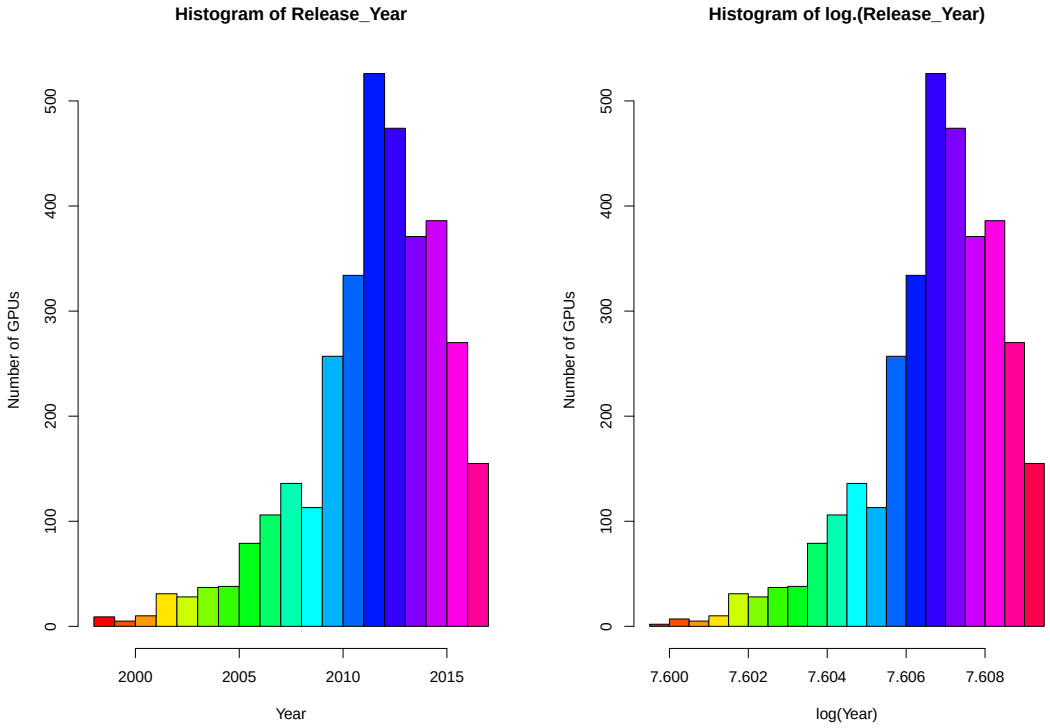
A histogram is a graphical representation that organizes a group of data points into user-specified ranges. Similar in appearance to a bar graph, the histogram condenses a data series into an easily interpreted visual by taking many data points and grouping them into logical ranges or bins.

```
hist(df$Release_Price, main="Histogram of Release_Price", xlab = "US
Dollar", ylab = "Number of GPUs", col = rainbow(10), breaks=100)
hist(df$log.Release_Price, main = "Histogram of log.Release_Price", xlab =
"log(US Dollar)", ylab = "Number of GPUs", col =
rainbow(length(unique(df$log.Release_Price))), breaks=100)
```



We can easily see the benefits of data transformation. It helps our data to focus on certain area rather than discretely distributed. Try again with `Release_Year`.

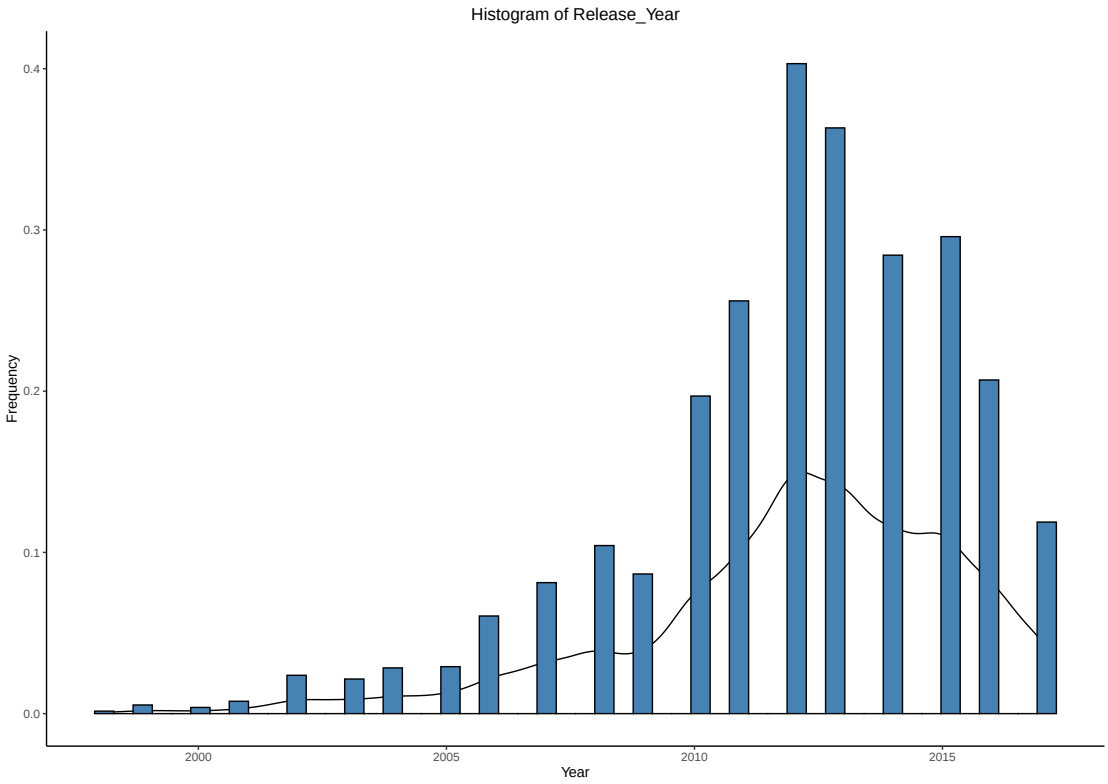
```
hist(df$Release_Year, main="Histogram of Release_Year", xlab = "Year", ylab
= "Number of GPUs", col = rainbow(20), breaks = 20, xlim=c(1998,2017))
hist(df$log.Release_Year, main = "Histogram of log.(Release_Year)", xlab =
"log(Year)", ylab = "Number of GPUs", col = rainbow(20), breaks = 20)
```



In this case we can see that both graphs are quite similar. So we can draw the conclusion that data transformation is not always the best way.

We can use the `ggplot()` function to both view the histogram and view the distribution shape of the dataset.

```
ggplot(df, aes(x = Release_Year, y=..density..)) + theme_classic() +
geom_density() +
  geom_histogram(bins = 50, fill = 'steelblue', color = 'black') +
  labs(title = 'Histogram of Release_Year', x = 'Year', y = 'Frequency') +
  theme(legend.position='bottom', plot.title = element_text(hjust = 0.5))
```

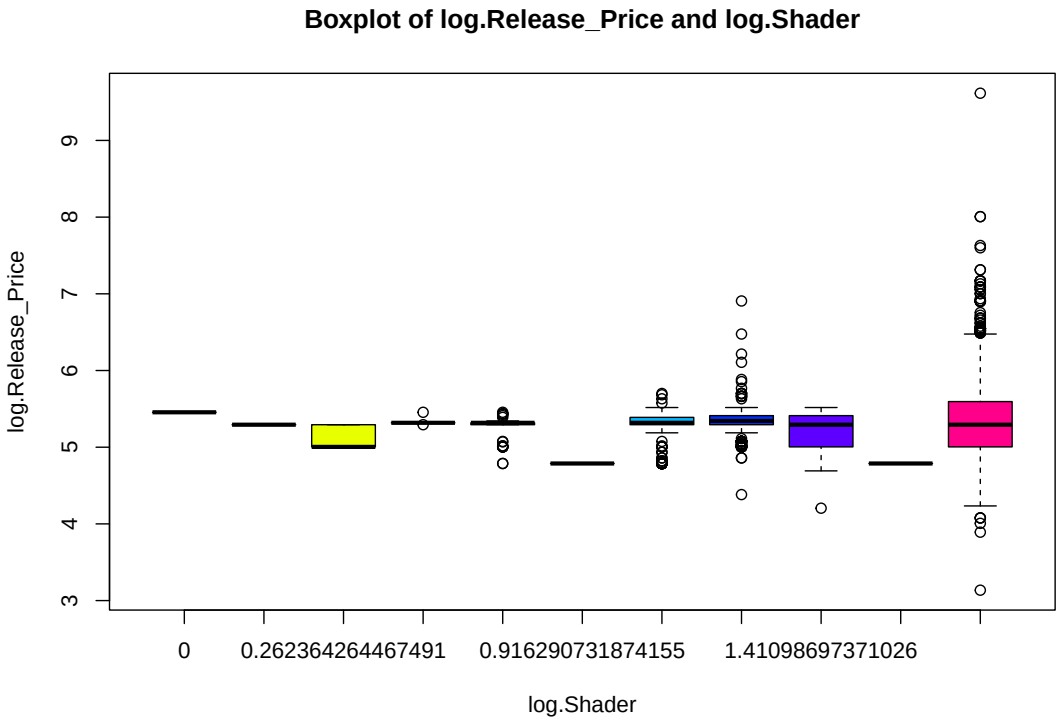


4.3.2 Boxplot

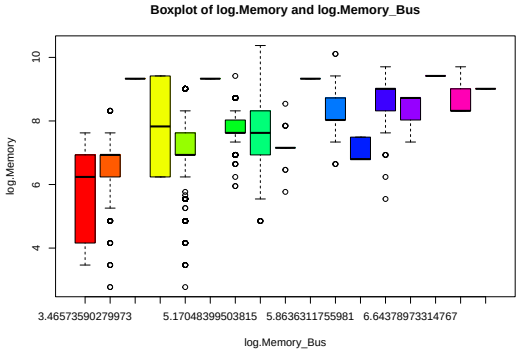
In descriptive statistics, a box plot or boxplot (also known as box and whisker plot) is a type of chart often used in explanatory data analysis. Box plots visually show the distribution of numerical data and skewness through displaying the data quartiles (or percentiles) and averages. Box plots show the seven-number summary of a set of data:

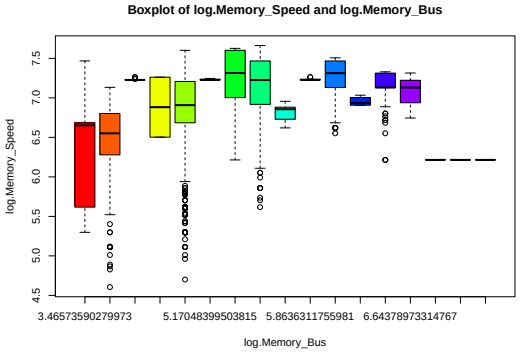
- Minimum score: The lowest score, excluding outliers (shown at the end of the left whisker).
- First (lower) quartile: 25% percent of scores fall below the lower quartile value.
- Median: The median marks the mid-point of the data and is shown by the line that divides the box into two parts (sometimes known as the second quartile). Half the scores are greater than or equal to this value and half are less.
- Third (upper) quartile: 75% of the scores fall below the upper quartile value. Thus, 25% of data are above this value.
- Maximum score: The highest score, excluding outliers (shown at the end of the right whisker).
- Whiskers: The upper and lower whiskers represent scores outside the middle 50% (i.e. the lower 25% of scores and the upper 25% of scores).
- The Interquartile Range (or IQR): This is the box plot showing the middle 50% of scores (i.e. the range between the 25th and 75th percentile).

```r
boxplot(df$log.Release_Price ~ df$log.Shader , main="Boxplot of
log.Release_Price and log.Shader", ylab = "log.Release_Price", xlab =
"log.Shader", col = rainbow (11))
boxplot(df$log.Memory ~ df$log.Memory_Bus, main = "Boxplot of log.Memory
and log.Memory_Bus", ylab = "log.Memory", xlab = "log.Memory_Bus", col =
rainbow (17))
boxplot(df$log.Memory_Speed ~ df$log.Memory_Bus, main = "Boxplot of
log.Memory_Speed and log.Memory_Bus", ylab = "log.Memory_Speed", xlab =
"log.Memory_Bus", col = rainbow (17))
```



In boxplot, the black line stands for median value, the top and bottom of the box combined make the interquartile range. The dotted lines stretching both direction from the box (whiskers) is the range, and the small circles outside the range are outliers.

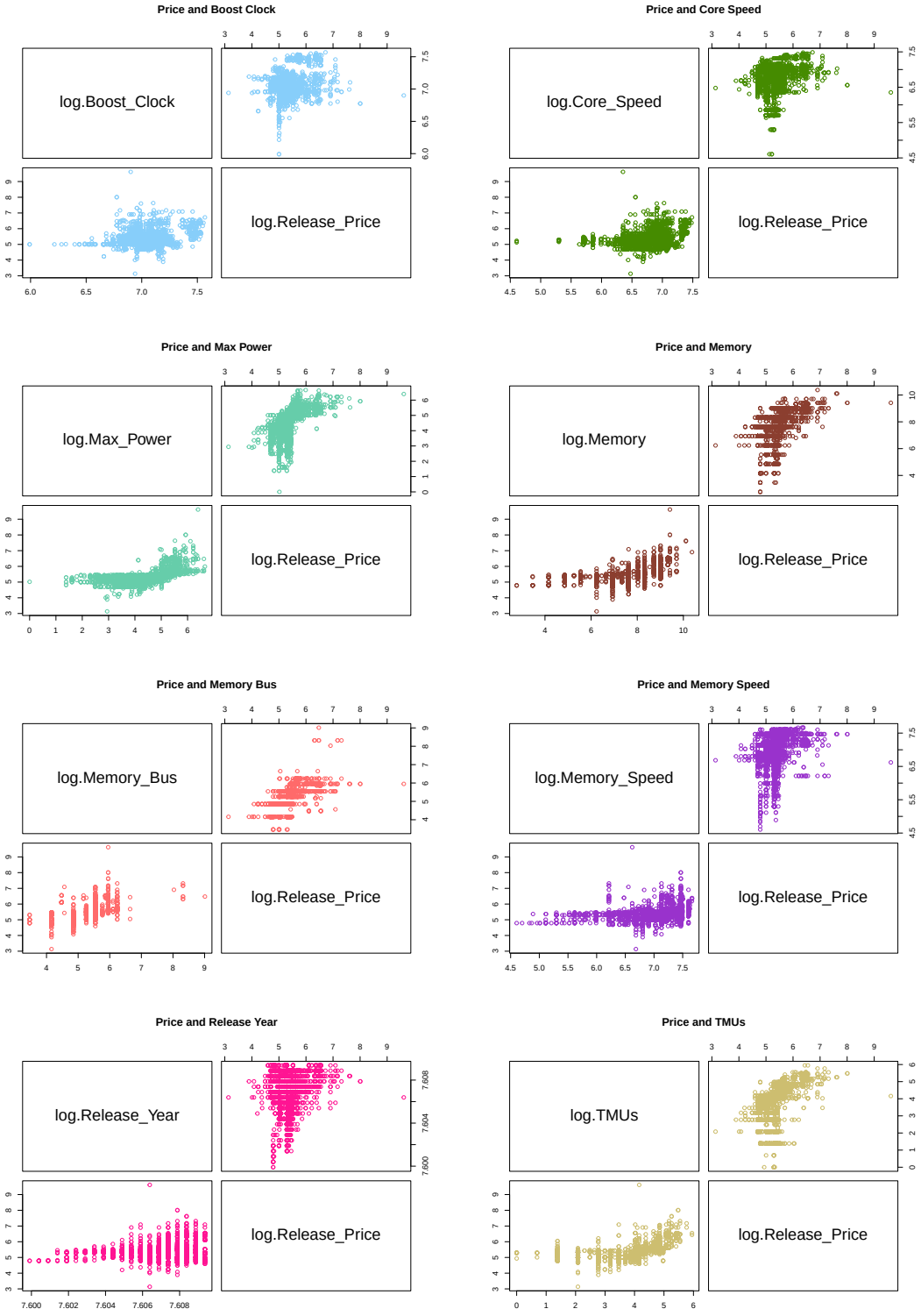




4.3.3 Pairs

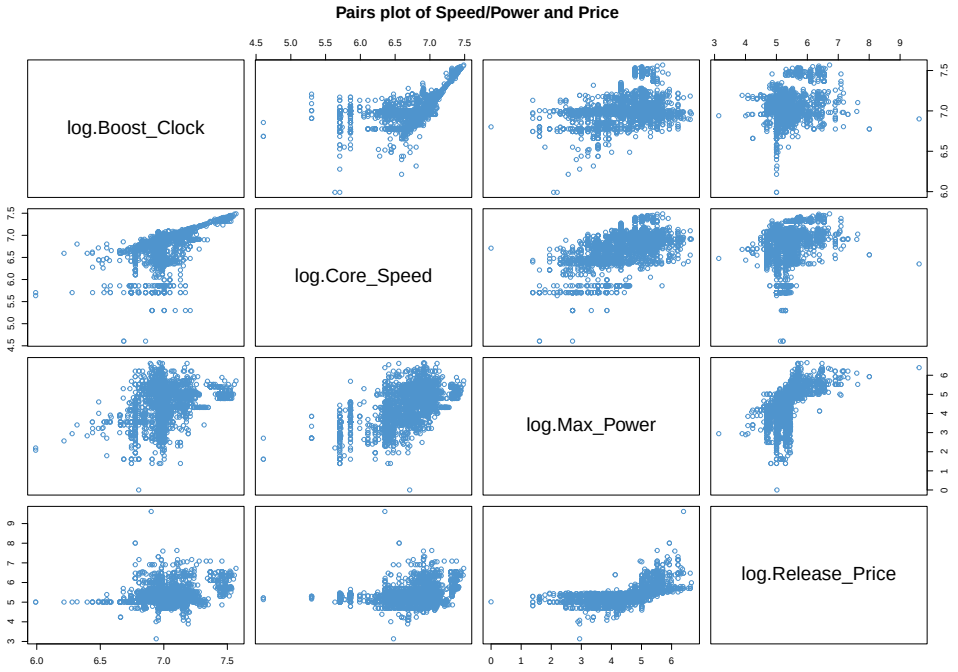
A pairs plot allows to see both distribution of single variables and relationships between two variables. It is a grid of scatterplots, showing the bivariate relationships between all pairs of variables in a multivariate dataset.

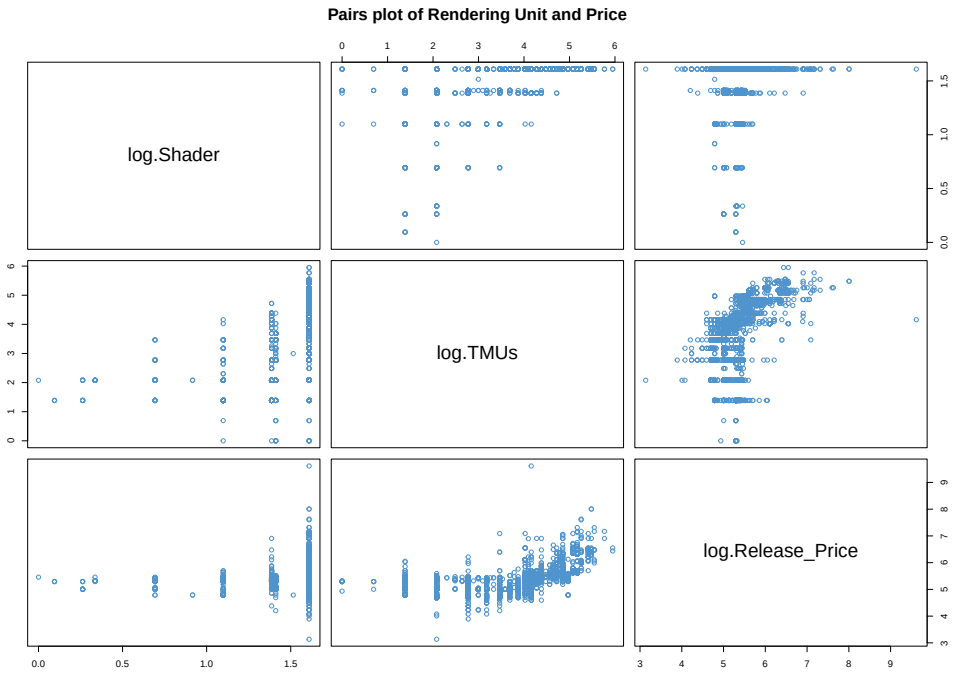
```
pairs(df[,c("log.Boost_Clock","log.Release_Price")], main = "Price and
Boost Clock", col = "lightskyblue")
pairs(df[,c("log.Core_Speed","log.Release_Price")], main = "Price and Core
Speed", col = "chartreuse4")
pairs(df[,c("log.Max_Power","log.Release_Price")], main = "Price and Max
Power", col = "aquamarine3")
pairs(df[,c("log.Memory","log.Release_Price")], main = "Price and Memory",
col = "coral4")
pairs(df[,c("log.Memory_Bus","log.Release_Price")], main = "Price and
Memory Bus", col = "indianred1")
pairs(df[,c("log.Memory_Speed","log.Release_Price")], main = "Price and
Memory Speed", col = "darkorchid")
pairs(df[,c("log.Release_Year","log.Release_Price")], main = "Price and
Release Year", col = "deeppink")
pairs(df[,c("log.TMUs","log.Release_Price")], main = "Price and TMUs", col
= "lightgoldenrod3")
```


Price and Boost Clock
log.Boost_Clock
log.Release_Price
Price and Core Speed
log.Core_Speed
log.Release_Price
Price and Max Power
log.Max_Power
log.Release_Price
Price and Memory
log.Memory
log.Release_Price
Price and Memory Bus
log.Memory_Bus
log.Release_Price
Price and Memory Speed
log.Memory_Speed
log.Release_Price
Price and Release Year
log.Release_Year
log.Release_Price
Price and TMUs
log.TMUs
log.Release_Price

Now we try combinations of 3 or more variables.

```
pairs(df[,c('log.Memory','log.Memory_Bus', 'log.Memory_Speed',
'log.Release_Price')], main = "Pairs plot of Memory and Price", col =
"steelblue3")
pairs(df[,c('log.Boost_Clock','log.Core_Speed','log.Max_Power',
'log.Release_Price')], main = "Pairs plot of Speed/Power and Price", col =
"steelblue3")
pairs(df[,c('log.Shader','log.TMUs', 'log.Release_Price')], main = "Pairs
plot of Rendering Unit and Price", col = "steelblue3")
```


Pairs plot of Speed/Power and Price
log.Boost_Clock
log.Core_Speed
log.Max_Power
log.Release_Price


Pairs plot of Rendering Unit and Price
log.Shader
log.TMUs
log.Release_Price

5 Hypothesis testing

A statistical hypothesis is a hypothesis that is testable on the basis of observed data modelled as the realised values taken by a collection of random variables. Hypothesis testing is a form of statistical inference that uses data from a sample to draw conclusions about a population parameter or a population probability distribution.

5.1 ANOVA

Analysis of variance (ANOVA) is an analysis tool used in statistics that splits an observed aggregate variability found inside a data set into two parts:

- Systematic factors.
- Random factors

The systematic factors have a statistical influence on the data set, while the random factors do not. Analysts use the ANOVA test to determine the influence that independent variables have on the dependent variable in a regression study.

5.1.1 One-way ANOVA

5.1.1.a Basic concept

A one-way ANOVA compares three or over three categorical groups to establish whether there is a difference between them. Within each group, there should be three or more observations, and the means of the samples are compared.

In this part, we want to know whether or not a significant difference in the average core speed among manufacturers.

First, we are going to use function selection to filter the data base on the core speed of each manufacturer in `df`:

```
aovCore_Speed <- select(df, Manufacturer, log.Core_Speed)
```

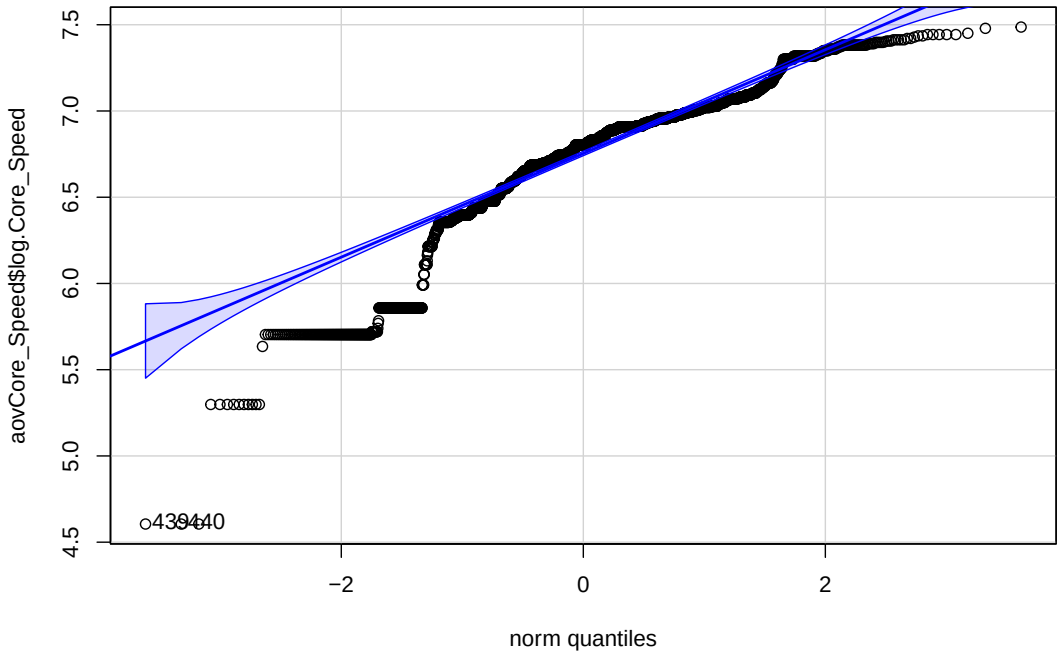
To compute one-way ANOVA test, the data in `df` must be a fixed-impact model meet the follow requirements:

- All data are independent and collected randomly.
- The population from those samples should be normally distributed.
- Homogeneity of variance: the variance among the groups should be approximately equal.

Condition 1:
The data is collected based on 4 different manufacturers, so the data satisfies the first condition.

Condition 2:
Normally, we will use the Shapiro-Wilk test, which is the most powerful normality test. However, due to the large sample size of the data, over 3300 GPUs satisfied after filtered, the Shapiro-Wilk test can not be used in R. Instead, we will use QQ-plot to see if the data is normally distributed [8]:

```r
ggqqplot(aovCore_Speed$log.Core_Speed)
```



In theory, sampled data from a normal distribution would fall along the dotted line. In reality, even data sampled from a normal distribution can exhibit some deviation from the line.

According to the plot above, the data are not normally distributed. Nonetheless the sample size is sufficiently large (> 200), the normality assumption is not needed at all as the Central Limit Theorem ensures that the distribution of disturbance term will approximate normality. Also, the one-way ANOVA is considered a robust test against the normality assumption. This means that it tolerates violations to its normality assumption rather well. As regards the normality of group data, the one-way ANOVA can tolerate data that is non-normal (skewed) with only a small effect on the Type I error rate.

We can conclude that the second condition is not necessarily satisfied.

5.1.1.b Building test

If we assume that, the data is homogeneous, which means it satisfies the one-way ANOVA assumptions. We will do the one-way ANOVA in R through these steps below:

Step 1: In order to determine if there is any significant difference in core speed among these manufacturers, we enumerate the null hypothesis H_0 and alternative hypothesis H_1.

- $H_0 : u_1 = u_2 = ... = u_i = 0$: There is a similarity in average core speed among 4 manufacturers: Nvidia, AMD, Intel and ATI.
- $H_1 : u_i \neq 0$ for at least one i .At least one of the manufacturer have a significant different in average `Core_Speed` compared to others.

We let α (significant level) $= 0.05$ by default.

Step 2: Select the appropriate test statistic.
We choose one-way ANOVA to compare 4 means in this hypothesis test.
The formula for the test statistic is:

$$F = \frac{MSTr}{MSE} = \frac{\frac{SSTr}{a-1}}{\frac{SSE}{a(n-1)}}$$

where:

- $MSTr$ is mean square for treatments.
- MSE is mean square for error.
- $SSTr$ is treatment sum of squares.
- SSE is error sum of squares.
- a is number of treatments.
- n is number of observations of each treatment (number of sample collected for each treatment).

Step 3: We will calculate the $SSTr$ with degree of freedom $a-1$ and SSE with degree of freedom $a(n-1)$ using the data from the provided table and the following formula:

$$SST = \sum_{i=1}^{I}\sum_{j=1}^{J_i} y_{ij}^2 - \frac{y^2_{..}}{n}$$
$$SSTr = \sum_{i=1}^{a} \frac{y_{i\cdot}^2}{n} - \frac{y^2_{..}}{N}$$
$$SSE = SST - SSTr$$

where:

- SST is the total sum of squares.
- i is the index of treatment.
- j is the index of observation(samples).
- y_{ij} is the data of the table at i treatments and j observations.
- $y_{i\cdot}$ is the total sum of observations(samples) for i treatment.

- $y..$ is the total sum of $y_1 + y_2 + ... + y_a$.

Step 4: Compute the test statistic.
We use RStudio to analysis with the filtered data to fulfill the request:

```
aov_one <- aov(log.Core_Speed ~ Manufacturer, data = aovCore_Speed)
aov_one
summary(aov_one)
```

```
Terms:
                Manufacturer Residuals
Sum of Squares       237.3332  287.2447
Deg. of Freedom             3      3361

Residual standard error: 0.2923424
Estimated effects may be unbalanced

               Df Sum Sq Mean Sq F value Pr(>F)
Manufacturer    3  237.3   79.11   925.7 <2e-16 ***
Residuals    3361  287.2    0.09
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

From F distribution table, we have $f_{0.05,3,\infty} = 2.2141$, therefore, $f_0 > f_{0.05,3,\infty}$, we can reject the null hypothesis H_0.

Before we go to the final conclusion, we check the third condition.

Condition 3:
As we have condition 2 is satisfied, we will use Levene-test to evaluate if the variances of two populations are equal. In statistics, Levene's test is an inferential statistic used to evaluate the equality of variances for a variable determined by two or more groups. For Levene's test the statistical hypotheses are:

- Null hypothesis H_0: All populations are equal.
- Alternative hypothesis H_1: At least two of them differ.

Thus, if the p-value is larger than the chosen α level, then the null hypothesis H_0 is not rejected and there is evidence that the variances of two populations are equal.

We use RStudio to perform Levene-test on the dataset:

```
leveneTest(log.Core_Speed ~ Manufacturer, aovCore_Speed, center = mean)
```

```
Levene's Test for Homogeneity of Variance (center = mean)
        Df F value    Pr(>F)
group    3   42.09  <2.2e-16 ***
      3361
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

From the output above, we can see that the p-value (< 2.2e-16) is much smaller compares to the alpha level 0.05. Now we reject the null hypothesis H_0 that all populations are equal, which also means there is a difference among manufacturers.

5.1.1.c Welch's test

Because of the condition 3 above is not satisfied for the one-way ANOVA, we will use the Welch's test instead. Welch's Test for Unequal Variances (also called Welch's t test, Welch's adjusted T or unequal variances t-test) is a modification of Student's t-test(more reliable when the two samples have unequal variances and/or unequal sample sizes) to see if two sample means are significantly different. The modification is to the degrees of freedom used in the test, which increases the test power for samples with unequal variance. There are two hypotheses in the Welch's test:

- Null hypothesis H_0: for the test is that the means are equal.
- Alternative hypothesis H_1: for the test is that means are not equal.

Below is the formula of Welch's test:

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{se_1^2 + se_2^2}}$$

where:

- $\overline{X_i}$ is the i^{th} sample mean.
- s_i is the i^{th} standard deviation.

Now we perform the Welch's test in R:

```
oneway.test(log.Core_Speed ~ Manufacturer, data=aovCore_Speed)
```

```
One-way analysis of means (not assuming equal variances)

data:  log.Core_Speed and Manufacturer
F = 1301.5, num df = 3.00, denom df = 488.59, p-value < 2.2e-16
```

Conclusion: According to the one-way ANOVA test and the Welch's test, which also results in rejecting the null hypothesis due to the p-value, we can officially claim that there is a difference in average core speed among manufacturers.

5.1.2 Two-way ANOVA

5.1.2.a Basic concept

A two-way ANOVA is designed to assess the interrelationship of two independent variables on a dependent variable. A two-way ANOVA is an extension of the one-way ANOVA. Use two-way ANOVA when there is one measure variable (i.e. a quantitative variable) and two nominal variables [9].

A two-way ANOVA with interaction tests three null hypotheses at the same time:

- There is no difference in group means at any level of the first independent variable.
- There is no difference in group means at any level of the second independent variable.
- The effect of one independent variable does not depend on the effect of the other independent variable (a.k.a. no interaction effect).

Note that a two-way ANOVA without interaction (a.k.a. an additive two-way ANOVA) only tests the first two of these hypotheses. And it is almost as if the dataset that we chose just allows us to do that. We assume that there are 3 price segments for GPUs (based on `Release_Price`):

- Bugdet: Under $200.
- Midrange: From $200 to $600.
- Highend: Above $600.

In this part, our hypothesis is:

- H_0: There is no difference in average core speed for any GPU manufacturer and in all 3 price segments.
- H_1: There is a difference in average core speed for any GPU manufacturer and in all 3 price segments.

We create a variable `Label` and assign a value to this variable based on the GPU price.

```
df$Label <- cut(df$log.Release_Price, c(log(0), log(200), log(600),
log(15000)), labels=c("Budget", "Midrange", "Highend"), include.lowest=T)
```

As a result, we have 2 nominal quantities and 1 quantitative quantity:

- `Manufacturer`: AMD, ATI, Intel and Nvidia.
- `Label`: Budget, Midrange and Highend.
- `log.Release_Price`: Price of GPU.

Two-way ANOVA requires a few assumptions:

1. Homogeneity of variance (a.k.a. homoscedasticity).
2. Independence of observations.
3. Normally-distributed dependent variable.

In our data set, the response variable is normally distributed, and we can check for homoscedasticity in the back.

5.1.2.b Building test

Now we will implement the two-way ANOVA model in R:

```
aov_two <- aov(log.Core_Speed ~ Manufacturer + Label, data = df)
aov_two
summary(aov_two)
```

```
Call:
   aov(formula = log.Core_Speed ~ Manufacturer + Label, data = df
  )

Terms:
                Manufacturer     Label Residuals
Sum of Squares     237.33325  16.15822 271.08648
Deg. of Freedom            3         2      3359

Residual standard error: 0.2840854

Estimated effects may be unbalanced
               Df Sum Sq Mean Sq F value Pr(>F)
Manufacturer    3 237.33   79.11   980.3 <2e-16 ***
Label           2  16.16    8.08   100.1 <2e-16 ***
Residuals    3359 271.09    0.08
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

From the result above, we can see that both manufacturer type and price segment explain a significant amount of variation in average core speed (p-value < 0.001).

5.1.2.c Post hoc test

ANOVA will tell us which parameters are significant, but not which levels are actually different from one another. To test this we can use a post-hoc test. The Tukey's Honestly-Significant-Difference (TukeyHSD) test lets us see which groups are different from one another.

```
1 TukeyHSD(aov_two)
```

```
Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = log.Core_Speed ~ Manufacturer + Label, data =
  df)

$Manufacturer
                    diff         lwr         upr     p adj
ATI-AMD        0.2734123  0.19257382  0.35425076 0.00e+00
Intel-AMD     -0.8968407 -0.94689573 -0.84678563 0.00e+00
Nvidia-AMD     0.1245477  0.09775552  0.15133986 0.00e+00
Intel-ATI     -1.1702530 -1.26095908 -1.07954687 0.00e+00
Nvidia-ATI    -0.1488646 -0.22911364 -0.06861555 1.15e-05
Nvidia-Intel   1.0213884  0.97229094  1.07048581 0.00e+00

$Label
                       diff        lwr       upr p adj
Midrange-Budget   0.1098742 0.08648174 0.1332667     0
```

```
Highend-Budget    0.2857955 0.22056322 0.3510278     0
Highend-Midrange  0.1759213 0.11044972 0.2413929     0
```

Conclusion: This output shows the pairwise differences between the four types of `Manufacturer` and between the three levels of `Label`, with the average difference (`diff`), the lower and upper bounds of the 95% confidence interval (`lwr` and `upr`) and the p-value of the difference (`p-adj`). From the post-hoc test results, we see that there are significant differences ($p < 0.05$) between all groups. That is, there is a statistically significant interaction between the effects of manufacturer and price segment on the starting price.

5.2 Z-test

One of the most common hypothesis testing method in statistics is the Z-test, used to determine whether the means of two groups are equal to each other. Note that while the T-test and Z-test have quite similar formulas, the selection of a particular test relies on sample size and the standard deviation of population. More specifically, we use the T-test when the sample size is less than 30 units, and Z-test is practically conducted when its size crosses the 30 units.

Because the dataset has a large sample size (the number of observations is more than 3000), we choose the Z-test instead of the T-test.

For the difference between Z-test and ANOVA:

- Z-test assesses whether mean of two groups are statistically different from each other or not.
- ANOVA assesses whether the average of more than two groups is statistically different.

5.2.1 One-sample Z-test

5.2.1.a Basic concept

A one-sample Z test is one of the most basic types of hypothesis test. It is used when we want to know whether the difference between the mean of a sample mean and the mean of a population is large enough to be statistically significant.

Assumptions of one-sample Z hypothesis test:

- Population data is continuous.
- Population follows a standard normal distribution.
- The mean and standard deviation of the population is known.
- Samples are independent of each other.
- The sample should be randomly selected from the population

5.2.1.b Building test

To take an example, we want to investigate the average memory among manufactures can be larger than 3000 MB or not.

```
df_z_one = df %>% select('Memory')
```

In order to run a one-sample Z test, we need to work through several steps:

Step 1: State the Null Hypothesis and the Alternative Hypothesis. The null hypothesis H_0 states that the sample mean is NOT different from the population mean. The alternative hypothesis, in the opposite, identify data that can disprove the null hypothesis. This is one of the common stumbling blocks–in order to make sense of the sample.

$$H_0 : \mu = \mu_0, \quad H_A : \mu \neq \mu_0$$

$$H_0 : \mu \leq \mu_0, \quad H_A : \mu > \mu_0$$

$$H_0 : \mu \geq \mu_0, \quad H_A : \mu < \mu_0$$

In this project, we choose our hypothesis test as:

- H_0: The average memory is equal or larger than 3000 MB.
- H_A: The average memory is less than 3000 MB.

Step 2: Specify significance level (α). Often, we choose significance levels equal to 0.01, 0.05, or 0.10; but any value between 0 and 1 can be used. In this project, we choose $\alpha = 0.05$, which means 5% chance of producing a significant result when the null hypothesis is correct.

Step 3: Compute the test statistic. Z-statistic is defined as:

$$Z = \frac{\overline{x} - \mu_0}{\sigma/\sqrt{n}}$$

We will use this function to calculate Z:

```
zStat <- (mean(df_z_one$Memory) - 3000) / (sd(df_z_one$Memory) /
sqrt(nrow(df_z_one)))
print(zStat)
```

```
-7.84571
```

Step 4: Find the critical value or compute corresponding probability value. This step of the hypothesis testing also involves the construction of the confidence interval depending upon the testing approach.

We use student's t-distribution and normal distribution to get the p-value:

```
pt(zStat, nrow(df_z_one) - 1)
```

```
2.868698e-15
```

```
pnorm(q=zStat, lower.tail=TRUE)
```

```
2.152562e-15
```

Note that we find here 2 different numbers, since those two numbers are very small, we assume they are equal.

Step 5: Construct acceptance / rejection regions and draw a conclusion. At the final step, we will reject or accept (fail to reject) the null hypothesis by making comparisons the critical value with the test statistic or P-value with significance level and make a general conclusion. So with our calculation, the p-value is very small than α, we can reject the null hypothesis. To sum up here, the average memory among manufactures can not be larger than 3000 MB.

5.2.2 Two-sample Z-test

5.2.2.a Basic concept

A two-sample Z-test is similar to one-way ANOVA in the sense that both aim to compare the means of independent groups. However, the Z-test concerns itself with only two sample groups; the null hypothesis is their two means are equal, and the alternative indicates there is a significant difference between the two groups.

In this part, we want to know whether or not there is a significant difference between the release prices of two of the GPU manufacturers, for example, Nvidia and AMD.

First, we use function selection and filter

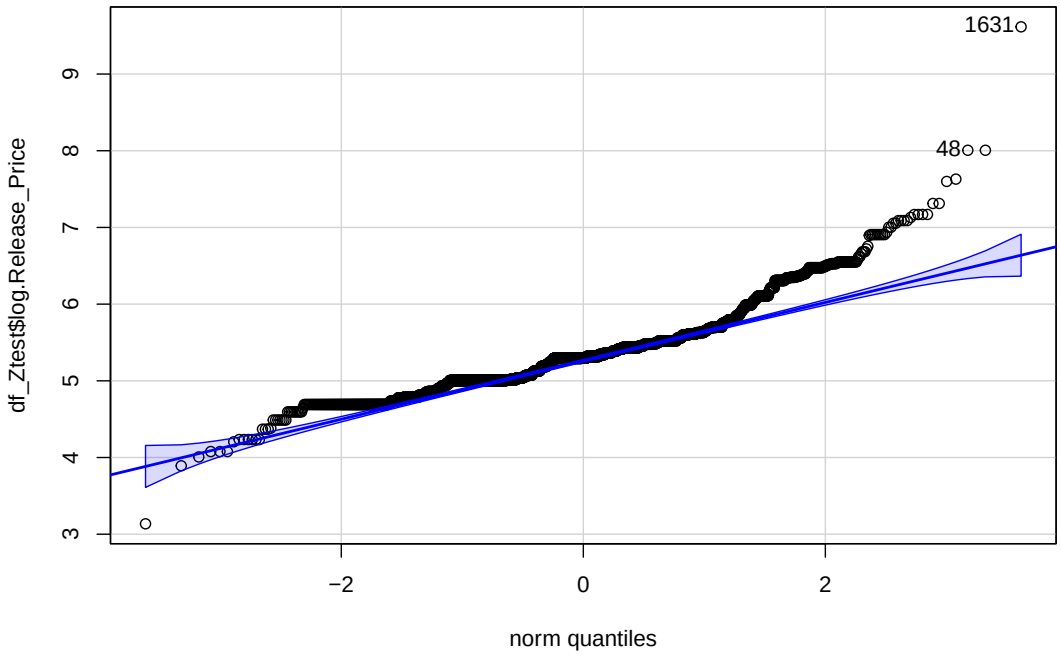
```
df_z = df %>% select('Manufacturer','log.Release_Price')
df_ztest = df_z %>% filter(dfz$Manufacturer == 'Nvidia' | dfz$Manufacturer
== 'AMD')
```

To perform the two-sample Z-test, our dataset `df-ztest` must satisfy following conditions:

- All data are independent and collected randomly.
- The test statistic is assumed to have a normal distribution.
- Homogeneity of variance: the variance among the groups should be approximately equal.

From the conditions above, we notice that the conditions of one-way Anova and of the two-sample Z-test are quite similar. Therefore, we can assume that our process of performing the two-sample Z-test should be the same as that of the one-way Anova. That includes executing the Shaprio-Wilk test or QQ-plot for condition 2, the Levene test to check and if needed, the Welch's test for condition 3.
Condition 1:
Condition is satisfied because data is collected between 2 different manufacturers Nvidia and AMD.
Condition 2:
Similar to ANOVA, because of the large sample size (3024 GPUs after filtered), we use QQ-plot to check if the data is normally distributed:

```
ggqqplot(df_ztest$log.Release_Price)
```



According to the plot above, the data are not normally distributed. Nonetheless the sample size is sufficiently large (> 200), the normality assumption is not needed at all as the Central Limit Theorem ensures that the distribution of disturbance term will approximate normality.

Condition 3:
We use RStudio to again perform Levene-test on the `df_ztest` dataset to evaluate if the variances of the two populations are equal:

```
leveneTest(log.Release_Price~Manufacturer,df_ztest, center=mean)
```

```
Levene's Test for Homogeneity of Variance (center = mean)
        Df F value    Pr(>F)
group    1  210.25 < 2.2e-16 ***
      3022
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

From the output above, we can see that the p-value `(< 2.2e-16)` is much smaller compares to the alpha level 0.05. Now we reject the null hypothesis H_0 that all populations are equal, which also means there is a difference among the two manufacturers.

5.2.2.b Welch's test

Because of the condition 3 above is not satisfied for the two-sample Z-test, we will use the Welch's test instead. There are two hypothesis in the Welch's test:

- Null hypothesis H_0: For the test is that the means are equal.
- Alternative hypothesis H_1: For the test is that means are not equal.

Recall the formula of Welch's test:

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{se_1^2 + se_2^2}}$$

where:

- $\overline{X_i}$ is the i^{th} sample mean.
- s_i is the i^{th} standard deviation.

Now we perform the Welch's test in RStudio:

```
oneway.test(log.Release_Price ~ Manufacturer, data=df_ztest)
```

```
One-way analysis of means (not assuming equal variances)

data:  log.Release_Price and Manufacturer
F = 125.23, num df = 1.0, denom df = 2869.3, p-value < 2.2e-16
```

Conclusion: According to the Welch's test, the p-value shows that we can reject the null hypothesis, and this proves the difference in average release prices between Nvidia and AMD manufacturers.

5.3 Chi-squared test

A Chi-square test is a hypothesis testing method. Tests involve checking if observed frequencies in one or more categories match expected frequencies. One of them is a test for independence compares two variables in a contingency table to see if they are related.

5.3.1 Chi-Squared Test of Independence

5.3.1.a Basic concept

The Chi-Squared Test of Independence can only compare categorical variables. It cannot make comparisons between continuous variables or between categorical and continuous variables. Additionally, the Chi-Square Test of Independence only assesses associations between categorical variables and can not provide any inferences about causation. [10, 11].

In this part, we want to find out if there is a relation between the core speed of GPUs and their prices.

To compute Chi-squared test, the data in `df` must meet the follow requirements:

- Two categorical variables.
- Two or more categories (groups) for each variable.
- Independence of observations.
  - There is no relationship between the subjects in each group.
  - The categorical variables are not "paired" in any way.
- Relatively large sample size.
  - Expected frequencies for each cell are at least 1.
  - Expected frequencies should be at least 5 for the majority (80%) of the cells.

5.3.1.b Building test

The null hypothesis (H_0) and alternative hypothesis (H_1) of the Chi-Square Test of Independence can be expressed in this way:

- H_0: Core speed is independent of price, which means knowing the value of core speed does not help to predict the value of the price.
- H_1: Core speed is associated with price.

We will use significant level $\alpha = 0.05$.
The test statistic for the Chi-squared test of Independence is denoted X^2, and is computed as:

$$X^2 = \sum_{i=1}^{R} \sum_{j=1}^{C} \frac{o_{ij} - e_{ij}}{e_{ij}}$$

where:

- o_{ij} is the observed cell count in the i^{th} row and j^{th} column of the table.
- e_{ij} is the expected cell count in the i^{th} row and j^{th} column of the table, computed as:

$$e_{ij} = \frac{\text{row}_i \text{ total} \times \text{col}_j \text{ total}}{\text{grand total}}$$

The calculated X^2 value is then compared to the critical value from the X^2 distribution table with degrees of freedom $df = (R-1)(C-1)$ and chosen confidence level. If the calculated X^2 value $>$ critical X^2 value, then we reject the null hypothesis.
Another way to find out answer is through the p-value, if it is less than a pre-determined significance level, which is 0.05 usually, then we reject the null hypothesis.

In our case, we decide to divide `Core_Speed` into 2 levels high (> 7) and low (< 7) as 7 is the mean of `log.Core_Speed` and the price into 3 ranges:

- Cheap (3-5).
- Medium (5-7).
- Expensive (>7).

```
HighCoreSpeed <- filter(df, log.Core_Speed > 7)
LowCoreSpeed <- filter(df, log.Core_Speed < 7)
```

After filter, we have this table:

log.Core_Speed	log.Release_Price		
	Cheap (3-5)	Medium (5-7)	Expensive (7-10)
High (>7)	80	530	8
Low (<7)	372	2362	12

The next step is to compute Chi-squared test in R. Chi-squared statistic can be easily computed as follow:

```
HighCoreSpeed  = c(80,530,8)
LowCoreSpeed  = c(373,2362,12)
chisq.test(data.frame(HighCoreSpeed ,LowCoreSpeed))
```

```
    Pearson's Chi-squared test

data:  data.frame(HighCoreSpeed, LowCoreSpeed)
X-squared = 6.3971, df = 2, p-value = 0.04082
```

Conclusion: We can say that since the p-value is smaller than our chosen significance level ($\alpha = 0.05$), we can reject the null hypothesis. Rather, we conclude that the two variables are in fact dependent.

6 Fitting linear regression model

Since the price of GPU is affected by many factors, we choose multiple regression model instead of simple regression model (single factor).

6.1 Fast introduction to multiple linear regression

A well-fitting regression model results in predicted values close to the observed data values. The mean model, which uses the mean for every predicted value, generally would be used if there were no informative predictor variables. Therefore, the fit of a proposed regression model will be better than the fit of the mean model.

By fitting a line to the observed data, we can describe how strong the relationship is between one or more independent variables and one dependent variable (how a dependent variable changes as the independent variable(s) change), or find the value of the dependent variable at a certain value of the independent variable(s). Given the data set we acquired, a multiple linear regression model is reasonable with the goal of observing how the dependent variable Release_Price would change under the influence of other variables.

A multiple linear regression model generally has the formula:

$$y = \beta_0 + \beta_1 X_1 + ... + \beta_n X_n + \epsilon$$

where:

- y is the predicted value of the dependent variable.
- β_0 is the y-intercept (value of y when all other parameters are set to 0).
- $\beta_1 X_1$ is the regression coefficient of the first independent variable.
- $\beta_n X_n$ is the regression coefficient of the last independent variable.
- e is the model error a.k.a. how much variation there is in our estimate of y.

To find the best-fit line for each independent variable, multiple linear regression calculates three things:

- The regression coefficients that lead to the smallest overall model error.
- The t-statistic of the overall model.
- The associated p-value (how likely it is that the t-statistic would have occurred by chance if the null hypothesis of no relationship between the independent and dependent variables was true).
- It then calculates the t-statistic and p-value for each regression coefficient in the model.

6.2 Constructing multiple linear regression model

We choose multiple linear regression model because the price of GPU depends on many factors.

6.2.1 Defining input and output

- Input:
  - `Boost_Clock`
  - `Core_Speed`
  - `Max_Power`
  - `Memory`
  - `Memory_Bus`
  - `Memory_Speed`
  - `Shader`
  - `TMUs`
  - `Release_Price`
- Output:
  - The multiple linear regression model with `Release_Price` is a dependent variable.

6.2.2 Linear regression model `lmPrice`

In this section we will implement the model and we will explain it in detail in the next section. By using the function `lm()` included in R:

```r
lmPrice = lm(log.Release_Price ~ log.Boost_Clock + log.Core_Speed +
log.Max_Power + log.Memory + log.Memory_Bus + log.Memory_Speed + log.Shader
+ log.TMUs, df)
summary(lmPrice)
```

```
Call:
lm(formula = log.Release_Price ~ log.Boost_Clock + log.Core_Speed
    +
     log.Max_Power + log.Memory + log.Memory_Bus + log.
   Memory_Speed +
     log.Shader + log.TMUs, data = df)

Residuals:
     Min       1Q   Median       3Q      Max
-1.5408  -0.1914  -0.0409   0.1517   3.4111

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)       2.159859   0.279644   7.724 1.48e-14 ***
log.Boost_Clock   0.596747   0.041774  14.285  < 2e-16 ***
log.Core_Speed   -0.307610   0.020964 -14.673  < 2e-16 ***
log.Max_Power     0.154701   0.011974  12.920  < 2e-16 ***
log.Memory        0.149222   0.009817  15.200  < 2e-16 ***
log.Memory_Bus    0.187296   0.015686  11.940  < 2e-16 ***
log.Memory_Speed -0.159942   0.022510  -7.106 1.46e-12 ***
log.Shader       -0.551948   0.041147 -13.414  < 2e-16 ***
log.TMUs          0.076648   0.011044   6.940 4.68e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3122 on 3356 degrees of freedom
Multiple R-squared:  0.5231, Adjusted R-squared:  0.522
F-statistic: 460.1 on 8 and 3356 DF,  p-value: < 2.2e-16
```

6.2.3 Explanation

6.2.3.a The `lm()` function

R linear regression uses the `lm()` function to create a regression model given some formula, in the form of

$$Y \sim X_1 + X_2 + ...$$

In our linear regression model:

- The independent X_i (predictor or random) variables:
  - `Boost_Clock`
  - `Core_Speed`
  - `Max_Power`

  - Memory
  - Memory_Bus
  - Memory_Speed
  - Shader
  - TMUs

- The dependent Y variable (the one we are trying to predict):
  - Release_Price
- Data source: The data frame df.

6.2.3.b The summary() function

The summary() is an inbuilt generic function in R used to produce result summaries of various model fitting functions. It returns:

1. Residuals: The section summarizes the residuals, the error between the prediction of the model and the actual results.
   - The min/max and quantiles Q1/Median/Q3 for the distribution of the residuals.
2. Coefficients
   - The values of the intercept (a value) and the slope (b value) for the other attributions are in the first column.
   - Estimate: The weight given to the variable. For every one unit of above attributes change, the model predicts a change of Release_Price as this equation:

$$\begin{aligned}\texttt{Release\_Price} = {} & + 0.596747 \times \texttt{Boost\_Clock} - 0.307610 \times \texttt{Core\_Speed} \\ & + 0.154701 \times \texttt{Max\_Power} + 0.149222 \times \texttt{Memory} \\ & + 0.187296 \times \texttt{Memory\_Bus} - 0.159942 \times \texttt{Memory\_Speed} \\ & - 0.551948 \times \texttt{Shader} + 0.076648 \times \texttt{TMUs} + 2.159859\end{aligned}$$

   - Std. Error: The standard error.
   - t value: A measure of how many standard deviations our coefficient estimate is far away from 0. Calculated by the formula:

$$t_{value} = \frac{Estimate}{Std.Error}$$

     A larger t-value indicates that it is less likely that the coefficient is not equal to zero purely by chance. Therefore, the higher the t-value, the better.
   - Pr(>|t|): The probability of observing any value equal or larger than t-value.
3. Signif. codes: A legend for the number of stars next to the p-value. For example, the Max_Power attribute has a p-value of <2e-16. Since this value is in the range [0, 0.001], it has a significance code of ***.

4. `Residual standard error`: A measure of the quality of a linear regression fit (the standard deviation of the residuals). Calculated by the formula:

$$SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$
$$\delta = \sqrt{\frac{SSE}{df}}$$

where:

- δ = Residual Standard Error.
- $df = n - (k+1)$ = Degrees of Freedom:
  - n = Number of observations.
  - $k+1$ = Number of coefficients including intercepts.

5. `Multiple R-squared`: A measure of how well the model is fitting the actual data. It indicates the proportion of the variance in the model that is explained by the model. Perfect is 1, none is 0.

$$SST = \sum_{i=1}^{n}(y_i - \bar{y})^2$$
$$SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$
$$SSR = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$
$$R^2 = 1 - \frac{SSE}{SST} = \frac{SSR}{SST}$$

6. `Adjusted R-squared`: The preferred measure as it adjusts for the number of variables considered.

$$MSE = \frac{SSE}{n-k}$$
$$MST = \frac{SST}{n-1}$$
$$R^2_{adj} = 1 - \frac{MSE}{MST} = 1 - \frac{(1-R^2)(n-1)}{n-k}$$

7. `F-statistic`: A indicator of whether there is a relationship between our predictor and the response variables (a test to see if a model with fewer parameters will be better).

   - `p-value`: A low value indicates that our model is probably better than a model with fewer parameters (i.e. there is little chance that the results are random).

$$MSR = \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}{k-1} = \frac{SST - SSE}{k-1}$$

- "Is the regression model containing at least one predictor useful in predicting price?". To answer the research question, we test the hypothesis testing:

$$H_0 : \beta_1 = \beta_2 = ... = \beta_n$$
$$H_1 : \exists \beta_i \neq 0$$

  - The full model
    This is the largest possible model - that is, the model containing all of the possible predictors. In this case, the full model is:

$$y_i = \alpha + (\beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n) + \varepsilon_i$$

    There are k predictors in the full model and the number of error degrees of freedom associated with the full model is: $df_R = n - (k + 1)$.
  - The reduced model
    This is the model that the null hypothesis describes:

$$y_i = \alpha + \varepsilon_i$$

    The reduced model basically suggests that none of the variation in the response y is explained by any of the predictors. Therefore, the number of error degrees of freedom associated with the reduced model is $df_R = n - 1$.
    We use the general linear `F-statistic` to decide whether or not:
    * To reject the null hypothesis H_0: The reduced model.
    * In favor of the alternative hypothesis H_1: The full model.

    In general, we reject H_0 if `F-statistic` is large - or equivalently if its associated `p-value` is small.

6.3 Considering eliminating any the independent variables due to the `Pr(>[t])`

From the result of `summary()` function, if we choose the confidence (significant) level at 0.05, we can keep all the variables:

- `Boost_Clock` ($2\times10^{-16} \ll 0.05$)
- `Core_Speed` ($2\times10^{-16} \ll 0.05$)
- `Max_Power` ($2\times10^{-16} \ll 0.05$)
- `Memory` ($2\times10^{-16} \ll 0.05$)
- `Memory_Bus` ($2\times10^{-16} \ll 0.05$)
- `Memory_Speed` ($1.46\times10^{-12} \ll 0.05$)
- `Shader` ($2\times10^{-16} \ll 0.05$)
- `TMUs` ($4.68\times10^{-12} \ll 0.05$)

It is clear that the probability of simultaneously rejecting H_0 of the above 8 variables is very high, so we have enough evidence to reject the null hypothesis H_0 that those coefficients are equal to 0.

6.4 Testing the linear regression model

`lmPrice` : Linear regression model includes all the variables.
`lmPriceNoMem` : Linear regression model without variable `Memory`.

6.4.1 Linear regression model `lmPriceNoMem`

We construct the linear regression model `lmPriceNoMem`:

```
lmPriceNoMem = lm(log.Release_Price ~ log.Boost_Clock + log.Core_Speed +
log.Max_Power + log.Memory_Bus + log.Memory_Speed + log.Shader + log.TMUs,
df)
summary(lmPriceNoMem)
```

```
Call:
lm(formula = log.Release_Price ~ log.Boost_Clock + log.Core_Speed
    +
     log.Max_Power + log.Memory_Bus + log.Memory_Speed + log.
   Shader +
     log.TMUs, data = df)

Residuals:
     Min       1Q   Median       3Q      Max
-1.6128  -0.1909  -0.0541   0.1503   3.6845

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)        0.92795    0.27666   3.354 0.000805 ***
log.Boost_Clock    0.73613    0.04213  17.473  < 2e-16 ***
log.Core_Speed    -0.30055    0.02166 -13.872  < 2e-16 ***
log.Max_Power      0.16255    0.01237  13.145  < 2e-16 ***
log.Memory_Bus     0.21411    0.01611  13.289  < 2e-16 ***
log.Memory_Speed  -0.06736    0.02240  -3.007 0.002655 **
log.Shader        -0.30204    0.03899  -7.746 1.24e-14 ***
log.TMUs           0.10841    0.01121   9.671  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3228 on 3357 degrees of freedom
Multiple R-squared:  0.4903, Adjusted R-squared:  0.4892
F-statistic: 461.3 on 7 and 3357 DF,  p-value: < 2.2e-16
```

After removing `Memory` attribute, the `R_squared` decrease, the adequacy of this model is still recognized and the `p-value` (`2.2e-16`) is much less than ($\ll$) significance value (`0.05`) which implies that the hypothesis that the intercept is zero is rejected. However, the model `lmPrice` still better than, see more in the next section.

6.4.2 Comparison

We use `anova()` function to test the linear regression model to related `Release_Price` to `Memory`.

```
lmPrice_Mem = lm(log.Release_Price ~ log.Memory, df[,
c('log.Release_Price', 'log.Memory')])
anova(lmPrice_Mem)
```

```
Analysis of Variance Table

Response: log.Release_Price
             Df Sum Sq Mean Sq F value    Pr(>F)
log.Memory    1 140.28 140.284  864.38 < 2.2e-16 ***
Residuals  3363 545.80   0.162
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Hypothesis:

H_0 : `Release_Price` does not depend on the variation of `Memory`.
H_1 : `Release_Price` depends on the variation of `Memory`.

Based on the result of the `anova()` testing, the `p-value` (`2.2e-16`) is much less than ($\ll$) significance value (`0.05`) which is a strong evidence of removing the null hypothesis H_0, admitting the dependence of `Release_Price` on `Memory`.
$\Rightarrow$ The model `lmPrice` is more accurate than the model `lmPriceNoMem`.

6.5 Model accuracy assessment

The overall quality of the model can be assessed by examining the R-squared (R^2), the Residual Standard Error (RSE) and F-Statistic.
We recall the summary of the model `lmPrice`:

```
Residual standard error: 0.312 on 3356 degrees of freedom
Multiple R-squared:  0.523,  Adjusted R-squared:  0.522
F-statistic:  460 on 8 and 3356 DF,  p-value: <2e-16
```

6.5.1 R-square

In multiple linear regression, the R^2 represents the correlation coefficient between the observed values of the outcome variable (y) and the fitted (i.e. predicted) values of y. For this reason, the value of R will always be positive and will range from zero to one.

R^2 represents the proportion of variance, in the outcome variable y, that may be predicted by knowing the value of the x variables. An R^2 value close to 1 indicates that the model explains a large portion of the variance in the outcome variable.

A problem with the R^2, is that, it will always increase when more variables are added to the model, even if those variables are only weakly associated with the response. A solution is to adjust the R^2 by taking into account the number of predictor variables.

The adjustment in the `Adjusted R Square` value in the summary output is a correction for

the number of x variables included in the prediction model.

In our multiple regression model, with 8 variables, the `Adjusted R-squared = 0.522`, meaning that "52.2% of the variance in the measure of price can be predicted by 8 variables".

6.5.2 Residual Standard Error

The RSE (or sigma σ) estimate gives a measure of error of prediction. The lower the RSE, the more accurate the model (on the data in hand).
The error rate can be estimated by dividing the RSE by the mean outcome variable:

```
sigma(lmPrice)/mean(df$log.Release_Price)
```

```
0.0585
```

In our multiple regression model, the RSE is `0.312` corresponding to 5.85% error rate.

6.5.3 F-Statistic

Recall that, the F-statistic gives the overall significance of the model. It assess whether at least one predictor variable has a non-zero coefficient.

In a simple linear regression, this test is not really interesting since it just duplicates the information given by the t-test, available in the coefficient table.

The F-statistic becomes more important once we start using multiple predictors as in multiple linear regression.

A large F-statistic will corresponds to a statistically significant p-value ($p < 0.05$). In our example, the F-statistic equal `460` producing a `p-value` of `<2e-16`, which is highly significant.

6.6 Multicollinearity checking

One of the assumptions of the Classical Linear Regression Model [12] is that there is no exact collinearity between the explanatory variables. If the explanatory variables are perfectly correlated, the following problems are faced:

- Parameters of the model become indeterminate.
- Standard errors of the estimates become infinitely large.

However, the case of perfect collinearity is very rare in practical cases. Imperfect or less than perfect multicollinearity is the more common problem and it arises when in multiple regression modelling two or more of the explanatory variables are approximately linearly related.

In multiple regression, two or more predictor variables might be correlated with each other. This situation is referred as collinearity. And so, as a result:

- Estimates for regression coefficients of the independent variables can be unreliable.
- Tests of significance for regression coefficients can be misleading.

In the presence of multicollinearity, the solution of the regression model becomes unstable. For a given predictor (p), multicollinearity can assessed by computing a score called the variance inflation factor (or VIF), which measures how much the variance of a regression coefficient is inflated due to multicollinearity in the model [13]. Here is the formula for calculating VIF:

$$VIF_i = \frac{1}{1 - R_i^2}$$

where:

- R_i^2 is the coefficient of determination of the linear regression equation,
- i is index number of variables in the linear regression model.

The smallest possible value of VIF is one (absence of multicollinearity). As a rule of thumb, a VIF value that exceeds 5 or 10 indicates a problematic amount of collinearity.

When faced to multicollinearity, the concerned variables should be removed, since the presence of multicollinearity implies that the information that this variable provides about the response is redundant in the presence of the other variables.

Now we will test our model using `vif()` function:

```
vif(lmPrice)
```

```
log.Boost_Clock   log.Core_Speed     log.Max_Power
       1.577571         2.364690         4.123739
     log.Memory   log.Memory_Bus log.Memory_Speed
       3.874111         2.807052         4.012138
     log.Shader          log.TMUs
       2.520362         5.664969
```

In our model, the VIF score for the predictor variable `TMUs` is a bit high (VIF = `5.664969`). This might be problematic. However, it seems that we can assume that the model is free from multicollinearity.

6.7 Confidence interval

We want to computes 95% confidence intervals for each variable in our model. Using `confint()` function:

```
confint(object, parm, level = 0.95, ...)
```

- `object`: A fitted model object, here we pass our `lmPrice` model.
- `parm`: A specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered. We skip this part.
- `level`: The confidence level required, default is set to 0.95.
- `...`: Additional argument(s) for methods, we skip this part.

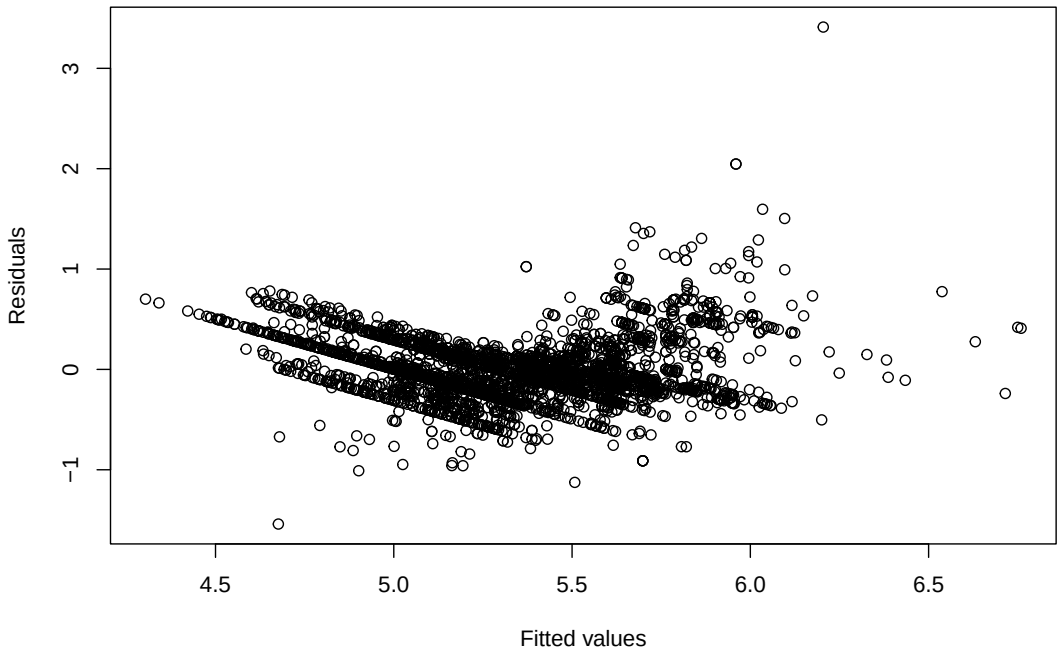
This function returns a matrix (or vector) with columns giving lower and upper confidence limits for each parameter. These will be labelled as $\frac{1-\text{level}}{2}$ and $1-\frac{1-\text{level}}{2}$ in % (by default 2.5% and 97.5%).

```
confint(lmPrice)
```

```
                   2.5 %  97.5 %
(Intercept)        1.612  2.7081
log.Boost_Clock    0.515  0.6787
log.Core_Speed    -0.349 -0.2665
log.Max_Power      0.131  0.1782
log.Memory         0.130  0.1685
log.Memory_Bus     0.157  0.2181
log.Memory_Speed  -0.204 -0.1158
log.Shader        -0.633 -0.4713
log.TMUs           0.055  0.0983
```

6.8 Graphs

```
plot(lmPrice$fitted.values, lmPrice$residuals, col = "black", xlab =
"Fitted values", ylab = "Residuals")
```



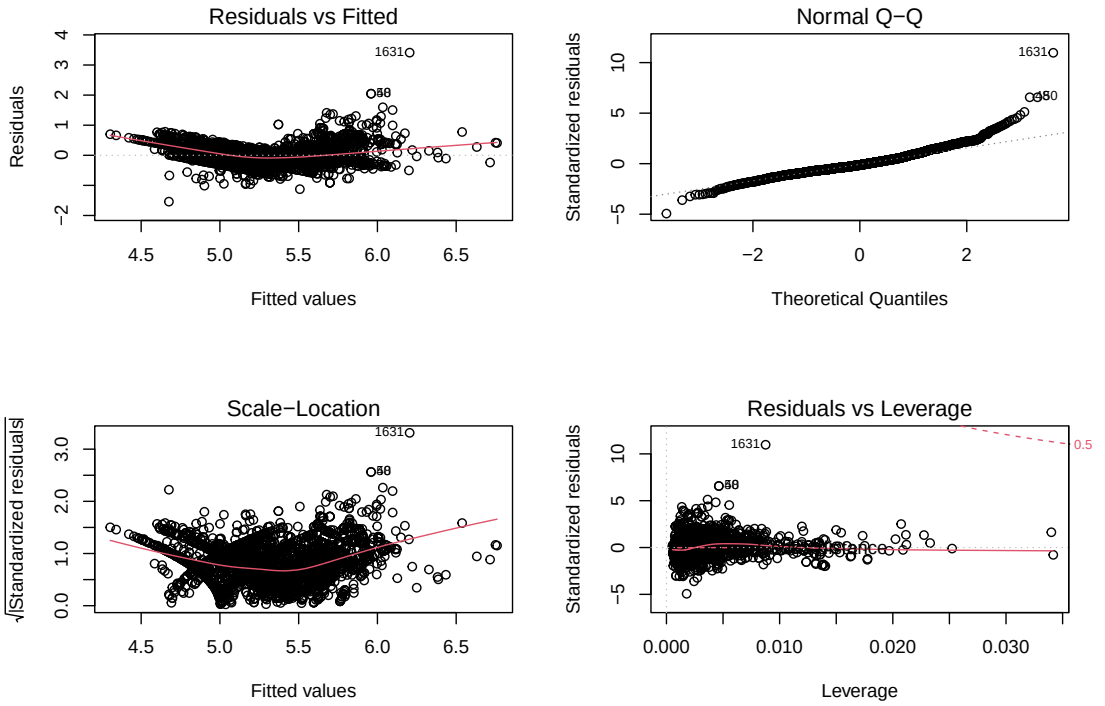
The natural logarithm of the GPU prices fluctuates mostly around $10^{4.5}$ to $10^{6.5}$ and that of the residuals is mostly around -1 to 1 which mean the predicted price and the actual price difference by between e^{-1} and e^{1}.

6.8.1 Homoscedasticity checking

Homoscedasticity, or homogeneity of variances, is an assumption of equal or similar variances in different groups being compared. The assumption of homoscedasticity (meaning "same variance") is central to linear regression models. Homoscedasticity describes a situation in which the error term (that is, the "noise" or random disturbance in the relationship between the independent variables and the dependent variable) is the same across all values of the independent variables. Heteroscedasticity (the violation of homoscedasticity) is present when the size of the error term differs across values of an independent variable. The impact of violating the assumption of homoscedasticity is a matter of degree, increasing as heteroscedasticity increases [14].

We try to plot the equation of our model to make sure that our model fit the homoscedasticity assumption of the linear model.

```
par(mfrow=c(2,2))
plot(lmPrice, col = "black")
par(mfrow=c(1,1))
```



Note that the `par(mfrow())` command will divide the Plots window into the number of rows

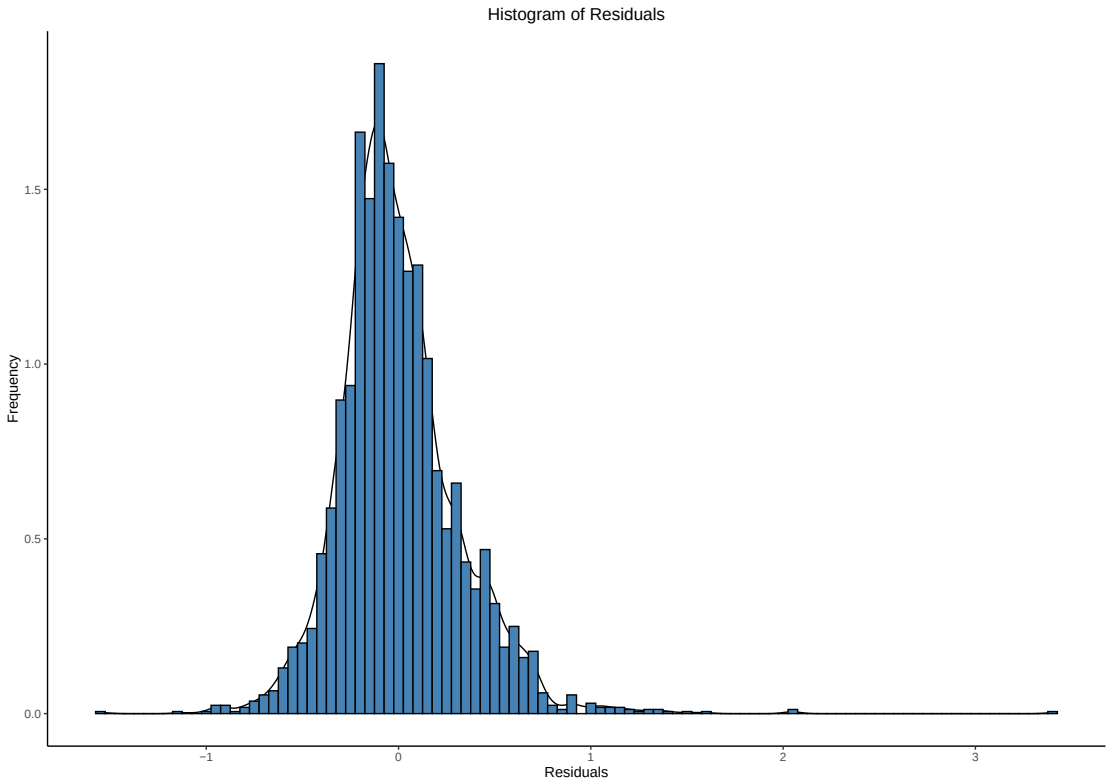
and columns specified in the brackets. So `par(mfrow=c(2,2))` divides it up into two rows and two columns. To go back to plotting one graph in the entire window, set the parameters again and replace the `(2,2)` with `(1,1)`.
Graph meaning:

- `Residuals vs Fitted`: When conducting a residual analysis, this plot is the most frequently created plot. It is a scatter plot of residuals on the y-axis and fitted values (estimated responses) on the x-axis. The plot is used to detect non-linearity, unequal error variances, and outliers.
- `Normal Q-Q`: This is a graphical technique for determining if two datasets come from populations with a common distribution.
- `Scale-Location`: This plot shows whether residuals are spread equally along the ranges of input variables (predictor).
- `Residuals vs Leverage`: This is a type of diagnostic plot that allows us to identify influential observations in a regression mode. Leverage is a measure of how far away the independent variable values of an observation are from those of the other observations

Based on the residual plots:

- Residuals are the unexplained variance. They are not exactly the same as model error, but they are calculated from it, so seeing a bias in the residuals would also indicate a bias in the error.
- The most important thing to look for is that the red lines representing the mean of the residuals are all basically horizontal and centered around zero. This means there are no outliers or biases in the data that would make a linear regression invalid.
- In the `Normal Q-Q` plot in the top right, we can see that the real residuals from our model form an almost perfectly one-to-one line with the theoretical residuals from a perfect model.
- We can check the residuals by plotting their histograms and distributions.

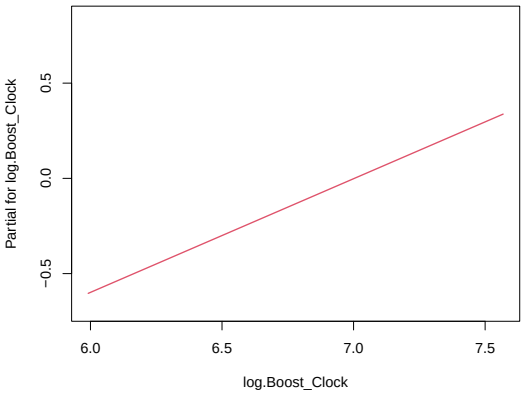
```
par(mfrow=c(1, 2))
ggplot(df, aes(x = lmPrice$residuals, y=..density..)) + theme_classic()
+  geom_density() +
  geom_histogram(bins = 100, fill = 'steelblue', color = 'black') +
  labs(title = 'Histogram of Residuals', x = 'Residuals', y =
  'Frequency') +
  theme(legend.position='bottom', plot.title = element_text(hjust =
  0.5))
```

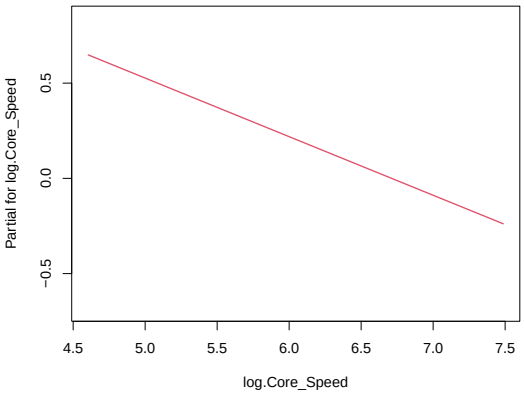


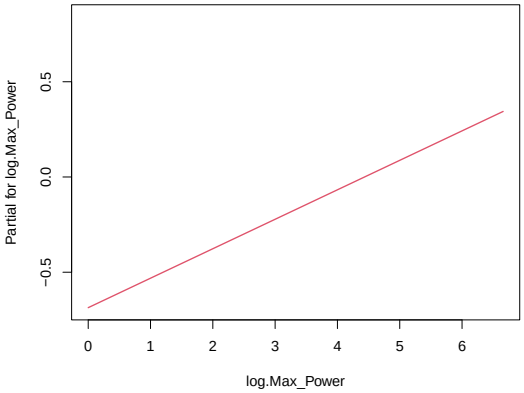
- From the above figure, we can see a "fair" normal distribution.
- Based on these residuals, we can say that our model meets the assumption of homoscedasticity.

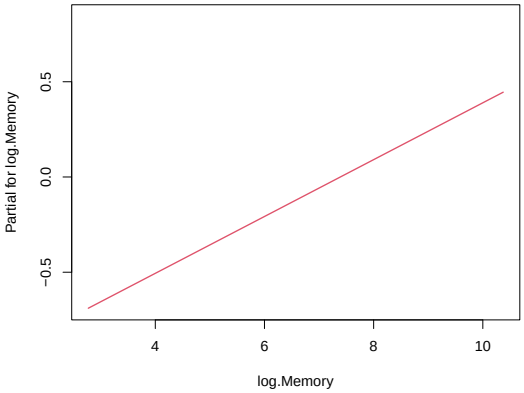
6.8.2 Linear regression graph for each variable

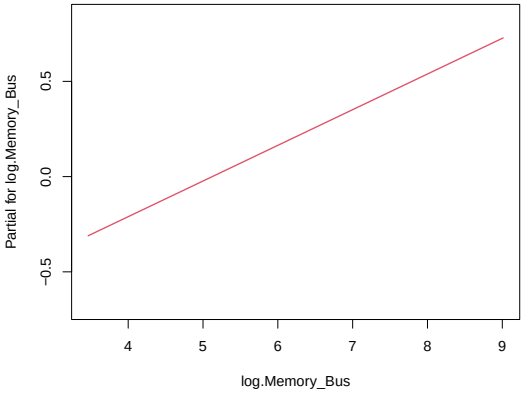
```
par(mfrow=c(1, 2))
termplot(lmPrice)
par(mfrow=c(1,1))
```

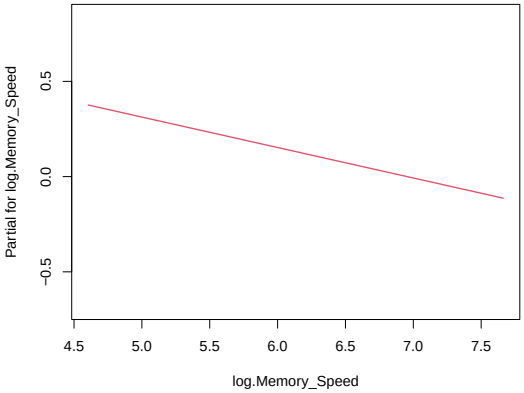

Partial for log.Boost_Clock
−0.5
0.0
0.5
6.0
6.5
7.0
7.5
log.Boost_Clock

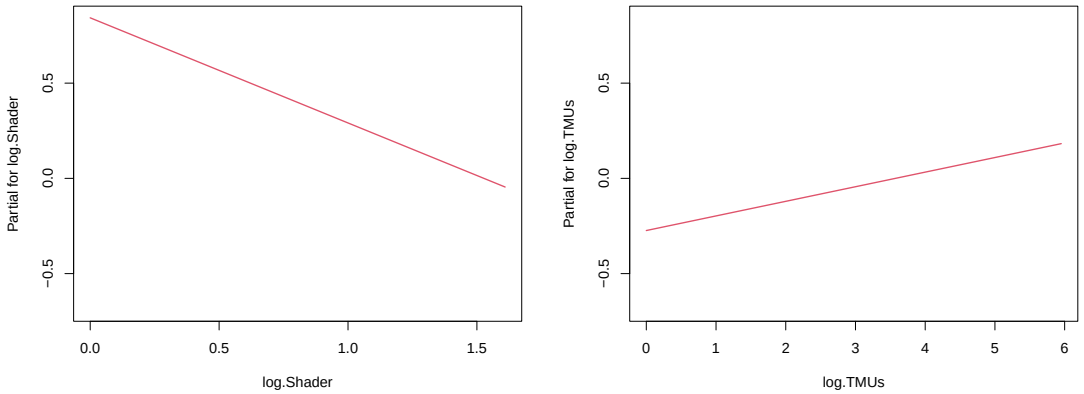

Partial for log.Core_Speed
−0.5
0.0
0.5
4.5
5.0
5.5
6.0
6.5
7.0
7.5
log.Core_Speed


Partial for log.Max_Power
−0.5
0.0
0.5
0
1
2
3
4
5
6
log.Max_Power


Partial for log.Memory
−0.5
0.0
0.5
4
6
8
10
log.Memory


Partial for log.Memory_Bus
−0.5
0.0
0.5
4
5
6
7
8
9
log.Memory_Bus

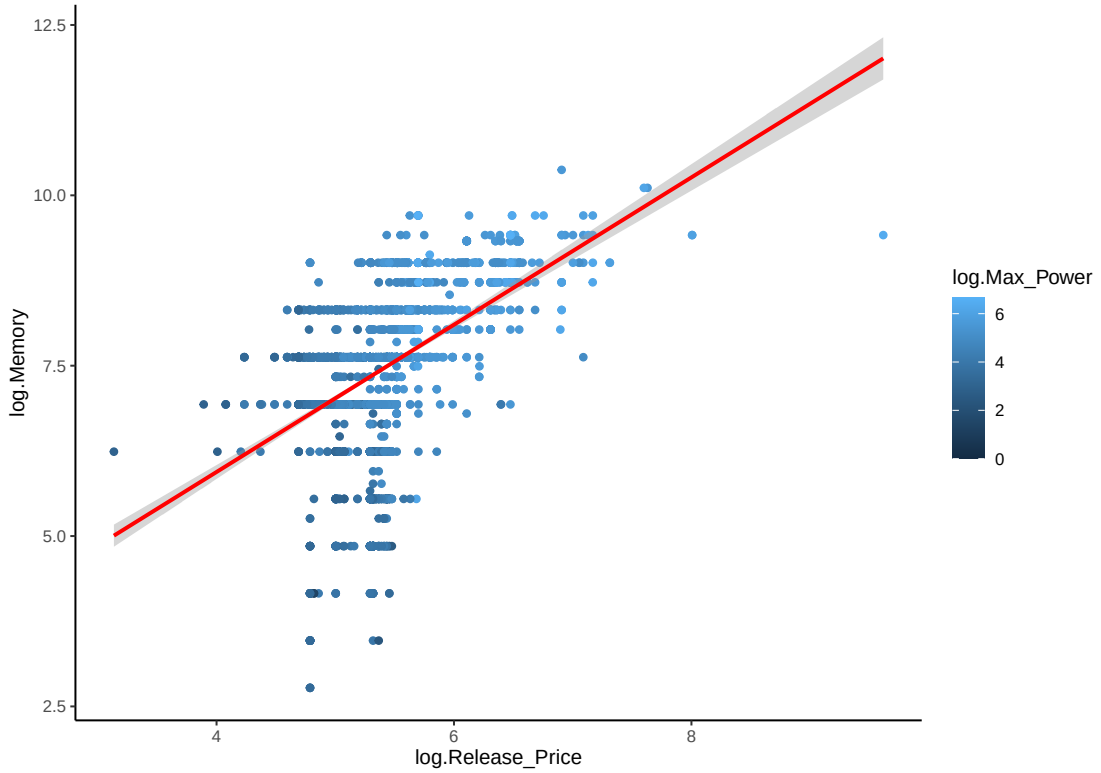

Partial for log.Memory_Speed
−0.5
0.0
0.5
4.5
5.0
5.5
6.0
6.5
7.0
7.5
log.Memory_Speed



6.8.3 Graphs involving multiple variables

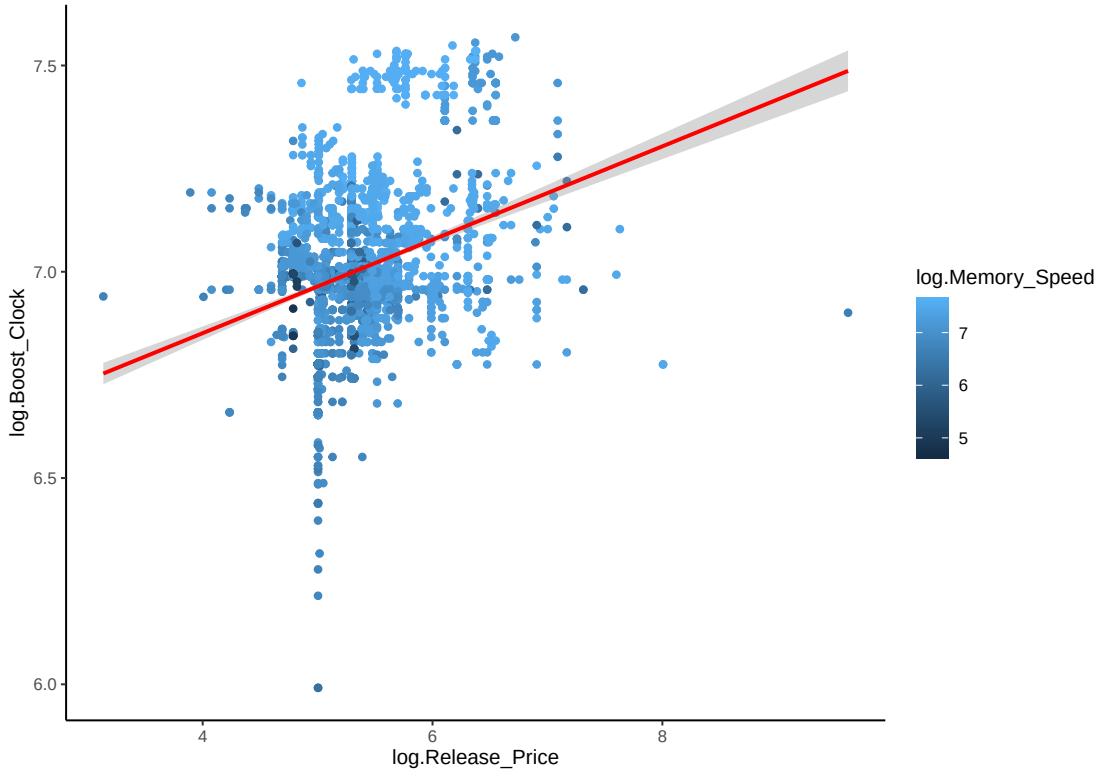
First, we try a combination of price, accessible memory capacity and maximum power.

```
price.graph <-ggplot(df, aes(x = log.Release_Price, y = log.Memory, color =
log.Max_Power)) + geom_point() + theme_classic()
price.graph <- price.graph + geom_smooth(method = "lm", col = "red")
price.graph
```



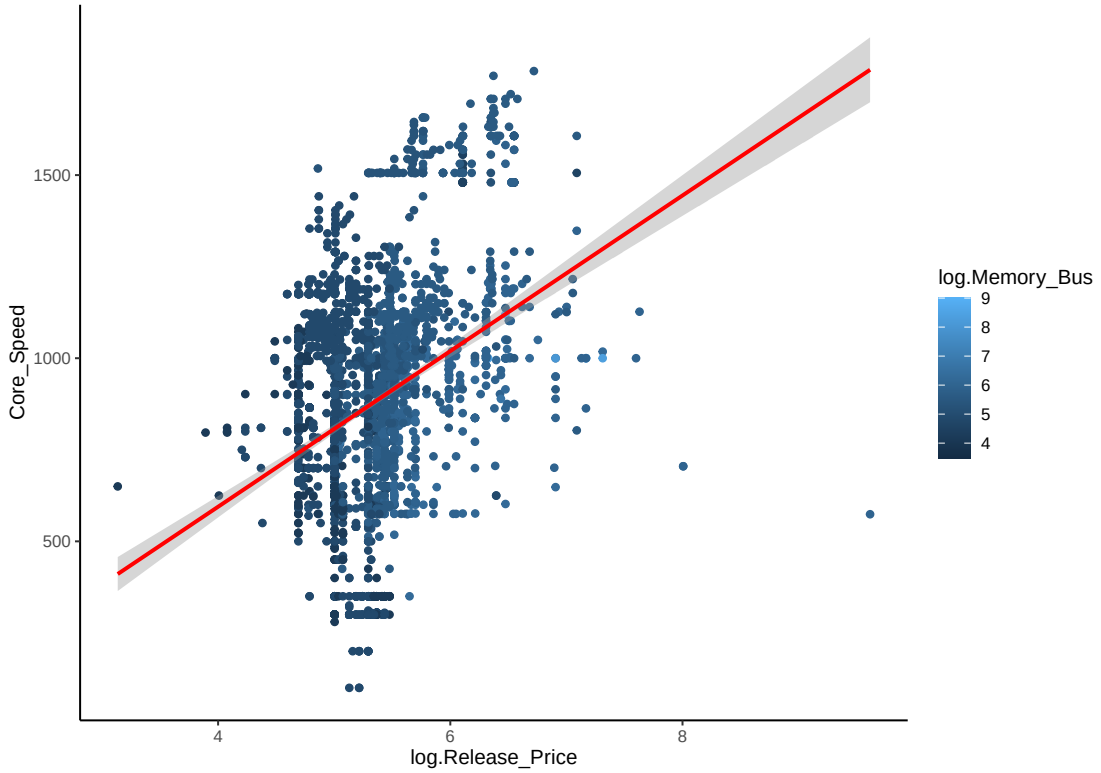
Then we try a combination of price, boost clock speed and memory speed.

```
price_2.graph <-ggplot(df, aes(x = log.Release_Price, y = log.Boost_Clock,
color = log.Memory_Speed)) + geom_point() + theme_classic()
price_2.graph <- price_2.graph + geom_smooth(method = "lm", col = "red")
price_2.graph
```



Finally, we try a combination of price, core speed and the size of the data transmission channel inside the memory.

```
price_3.graph <-ggplot(df, aes(x = log.Release_Price, y = Core_Speed, color
= log.Memory_Bus)) + geom_point() + theme_classic()
price_3.graph <- price_3.graph + geom_smooth(method = "lm", col = "red")
price_3.graph
```



6.9 Model conclusion

In the section above, we choose the linear regression model `lmPrice`. According to this model, the estimated coefficient of

- `Boost_Clock`
- `Max_Power`
- `Memory`
- `Memory_Bus`
- `TMUs`

imply that these five variables will increase the `Release_Price` of GPUs. Meanwhile,

- `Core_Speed`
- `Memory_Speed`
- `Shader`

will decrease the `Release_Price` of GPUs due to their negative coefficients.

7 Fitting extra model

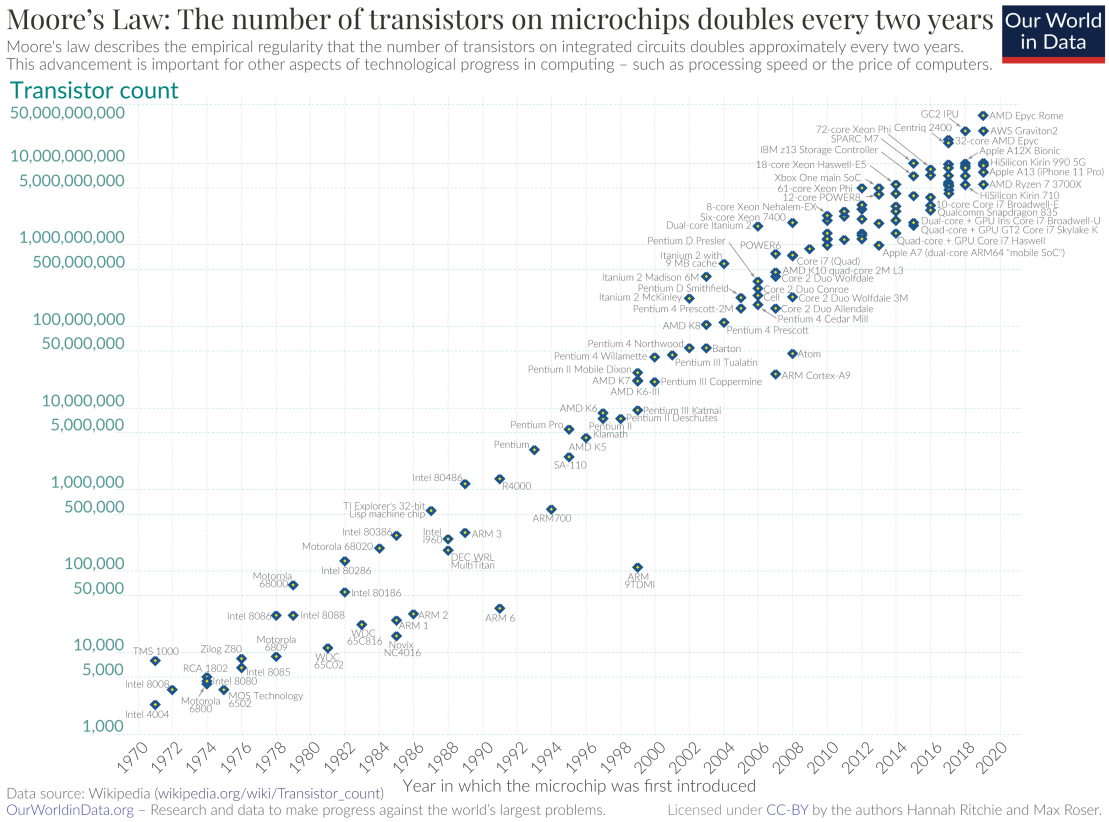
7.1 Fitting exponential model

7.1.1 Moore's law

Moore's law is a term used to refer to the observation made by Gordon Moore in 1965 that the number of transistors in a dense integrated circuit (IC) **doubles** about (approximately) every two years. This aspect of technological progress is important as the capabilities of many digital electronic devices are strongly linked to Moore's Law.

The observation is named after Gordon Moore, the co-founder of Fairchild Semiconductor and Intel (and former CEO of the latter), who in 1965 posited a doubling every year in the number of components per integrated circuit, and projected this rate of growth would continue for at least another decade. In 1975, looking forward to the next decade, he revised the forecast to doubling every two years, a compound annual growth rate (CAGR) of 41%. While Moore did not use empirical evidence in forecasting that the historical trend would continue, his prediction held since 1975 and has since become known as a "law".

As our large updated graph here shows, he was not only right about the next ten years but astonishingly the regularity he found is true for more than half a century now.



7.1.2 Theoretic model

Since our data is about the GPU, it also means about the circuit. Therefore, if the data is converted to logarithmic form, the graph, if Moore's law is correct, should be close to a straight line. In fact, the function representing the memory size calculated using Moore's law can be expressed as:

$$f(x) = y_{min} 2^{\frac{x - x_{min}}{2}}$$

where:

- y_{min}: Initial size of memory in MB.
- x_{min}: Initial year of dataset.

7.1.3 Graphs

Now we will plot a graph to see if the GPU's memory obeys Moore's law. First, we create 3 arrays:

- `year_arr`: This is an array of year values sorted in ascending order.
- `memory_arr_mean`: This is an array of mean (average) memory capacity values grouped by year and sorted in ascending order.
- `memory_arr_median`: This is an array of median memory capacity values grouped by year and sorted in ascending order.

```
year_arr <- unique(sort(strtoi(df$Release_Year)))

memory_arr_mean = data.frame(df %>% group_by(Release_Year) %>%
summarise_at(vars(Memory), list(name = mean)))
memory_arr_mean = array(memory_arr_mean$name)

memory_arr_median = data.frame(df %>% group_by(Release_Year) %>%
summarise_at(vars(Memory), list(name = median)))
memory_arr_median = array(memory_arr_median$name)
```

Next we create two functions `calculateMooresValue()` and `exponentialCurve()`:

```
calculateMooresValue <- function(x, y_trans) {
  return(memory_arr_median[1] * 2**((x-y_trans)/2))
}

exponentialCurve <- function(x, a, b, c) {
  return(a*2**((x-c)*b))
}
```

- `calculateMooresValue()`: The function $f(x)$ that we mentioned above. Used to calculate the theoretical value according to Moorse's law.
  - Input:

    - `x`: Current year.
    - `y_trans`: Start year (smallest year).
    - `memory_arr_median[1]`: Median value of start year.
  - Output:
    - The value of the year's memory capacity is entered.

- `exponentialCurve()`: Used to fit exponential curve to our dataset.
  - Input:
    - `x`: Current year.
    - `a`: Median value.
    - `b`: The coefficient corresponds to the factor $\frac{1}{2}$ in the $2^{\frac{1}{2}\cdots}$ exponent of the formula $f(x)$.
    - `c`: Start year.
  - Output:
    - The coordinate value of the y-axis of the function $f(x)$ according to our data set.

We use Python to find the triple variable `a`, `b` and `c` because this language has very strong support libraries for fitting curves. Take a look here again.

```
popt, pcov = curve_fit(f=exponentialCurve,  xdata=year_arr,
ydata=memory_arr_mean,  p0=(2, 0.5, 1998))
```

- `popt`: Optimal values for the parameters, specifically here are `a`, `b` and `c`.
- `pcov`: The estimated covariance of `popt`.
- `f`: Our function $f(x)$.
- `xdata`: x-coordinate data.
- `ydata`: y-coordinate data.
- `p0`: The initial guess for the fitting coefficients.

We check the returned values.

```
print(popt)
```

```
[1.04294249e+01 3.55525954e-01 1.99040139e+03]
```

These numbers mean:

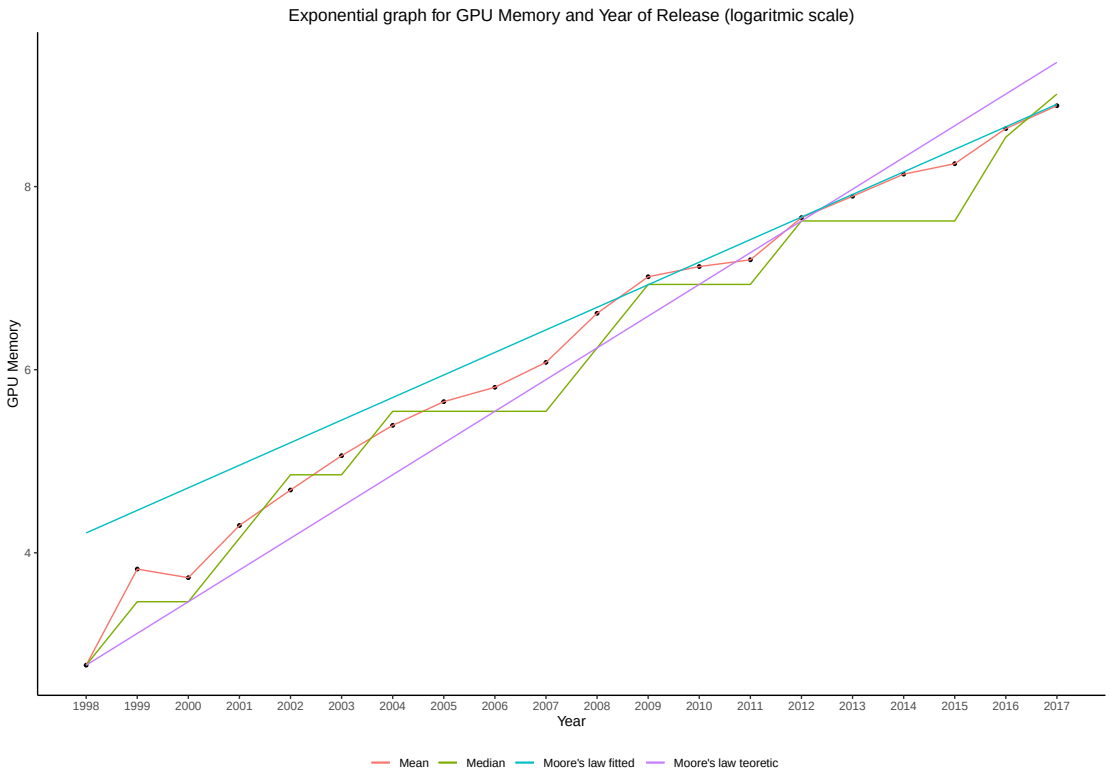
- `a = 1.04294249e+01`
- `b = 3.55525954e-01`
- `c = 1.99040139e+03`

Back to RStudio, we will build two models based on Moore's law and the above results.

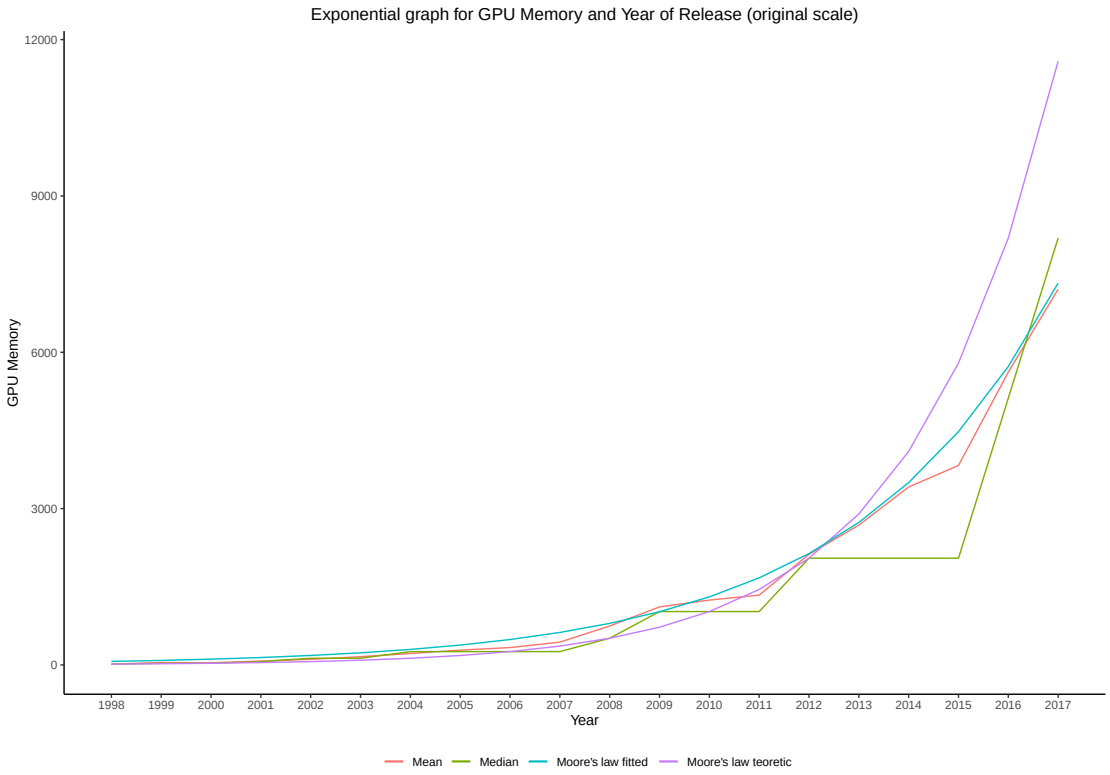
```
y_pred_moore_law_teoretic = calculateMooresValue(year_arr, year_arr[1])
popt <- c(1.04294249e+01, 3.55525954e-01, 1.99040139e+03)
y_pred_moore_law_fitted = exponentialCurve(year_arr, popt[1], popt[2],
popt[3])
```

And then we built a graph to compare. For convenient drawing and manipulation, we create a new data frame `df_extra` and store all the above values in it.

```
df_extra<-data.frame(Release_Year = year_arr, Memory_Mean =
memory_arr_mean, Memory_Median = memory_arr_median, Moore_Teoretic =
y_pred_moore_law_teoretic, Moore_Fitted = y_pred_moore_law_fitted)

ggplot(df_extra, aes(x = Release_Year)) +  theme_classic() +
  scale_x_continuous(breaks = scales::pretty_breaks(n = 20)) +
  theme(legend.position='bottom', plot.title = element_text(hjust = 0.5)) +
  xlab("Year") + ylab("GPU Memory") + labs(colour="") +
  ggtitle("GPU Memory vs Year of Release (logaritmic scale)") +
  geom_line(aes(y = log(Memory_Mean), colour = "Mean")) +
  geom_line(aes(y = log(Memory_Median), colour = "Median")) +
  geom_line(aes(y = log(Moore_Teoretic), colour = "Moore's law teoretic"))
  +
  geom_line(aes(y = log(Moore_Fitted), colour = "Moore's law fitted"))
```



As we can see, mean size [red line] of GPUs memory tends to follow teoretic Moore's law curve [purple line] but not ideally. We need to do something to better fit the dataset [cyan line]. Therefore, we will try with the polynomial model in the next section. Besides, we also draw a graph to the original scale for easy comparison with the polynomial model.



7.2 Fitting polynomial regression model

In this section, we build a polynomial regression model to compare with other models. We will focus only on the average value of the memory.

7.2.1 Constructing quadratic polynomial regression

A quadratic regression is the process of finding the equation of the parabola that best fits a set of data. As a result, we get an equation of the form:

$$y = ax^2 + bx + c$$

where:

- $a \neq 0$.
- y: Size of memory in MB.
- x: Year.

```
poly_2_mean <- lm(Memory_Mean ~ Release_Year + I(Release_Year^2),
data=df_extra)
summary(poly_2_mean)
```

```
Call:
lm(formula = log(Memory_Mean) ~ Release_Year + I(Release_Year^2),
    data = df_extra)

Residuals:
     Min       1Q   Median       3Q      Max
-0.31983 -0.11978  0.01628  0.08554  0.33658

Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
(Intercept)       -2.159e+04  4.736e+03  -4.559 0.000278 ***
Release_Year       2.122e+01  4.719e+00   4.497 0.000318 ***
I(Release_Year^2) -5.211e-03  1.175e-03  -4.434 0.000364 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1557 on 17 degrees of freedom
Multiple R-squared:  0.9932, Adjusted R-squared:  0.9924
F-statistic:  1241 on 2 and 17 DF,  p-value: < 2.2e-16
```

Let's take a look at the coefficients of the above model.

```
coef(poly_2_mean)
```

```
      (Intercept)       Release_Year I(Release_Year^2)
     1.273072e+08      -1.271265e+05       3.173648e+01
```

Looks good, now we are going to predict the amount of memory (value on the y-axis) from the model.

```
year_df <- data.frame(Release_Year = year_arr)
y_pred_lin_reg_2 <- predict(poly_2_mean, year_df, type="response")
```

And then we save these values to the data frame.

```
df_extra$Y_pred_lin_reg_2 <- y_pred_lin_reg_2
```

7.2.2 Constructing cubic polynomial regression

Like a quadratic polynomial, a cubic polynomial is a polynomial of the form:

$$y = ax^3 + bx^2 + cx + d$$

where:

- $a \neq 0$.
- y: Size of memory in MB.
- x: Year.

```
poly_3_mean <- lm(Memory_Mean ~ Release_Year + I(Release_Year^2) +
I(Release_Year^3), data=df_e)
summary(poly_3_mean)
```

```
Call:
lm(formula = Memory_Mean ~ Release_Year + I(Release_Year^2) +
    I(Release_Year^3), data = df_extra)

Residuals:
    Min      1Q  Median      3Q     Max
-670.09 -344.06   87.56  286.96 1041.90

Coefficients: (1 not defined because of singularities)
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)        1.273e+08  1.337e+07   9.524 3.15e-08 ***
Release_Year      -1.271e+05  1.332e+04  -9.546 3.05e-08 ***
I(Release_Year^2)  3.174e+01  3.317e+00   9.568 2.95e-08 ***
I(Release_Year^3)         NA         NA      NA       NA
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 439.5 on 17 degrees of freedom
Multiple R-squared:  0.9584, Adjusted R-squared:  0.9535
F-statistic:   196 on 2 and 17 DF,  p-value: 1.815e-12
```

It seems that in the 3rd degree polynomial model, there is an error that the two or more independent variables are perfectly collinear.Indeed, the error `not defined because of singularities` will occur due to strong correlation between independent variables. And if this error occurs we will not be able to find the predicted y value because one of the coefficients of the model is missing.

Therefore, to be able to run this model, we use Python to get the coefficients for the model. Check here.

```
print(lin_reg_3.intercept_)
```

```
-17128661108.085476
```

```
print(lin_reg_3.coef_)
```

```
[ 2.56615577e+07 -1.28150909e+04  2.13323868e+00]
```

After obtaining the coefficient values, we change them with the corresponding values existing in the model.

```
poly_3_mean$coefficients[1:4] <- c(-17128661108.085476, 2.56615577e+07,
-1.28150909e+04,  2.13323868e+00)
poly_3_mean$rank <- c(4)
```

Note that because in the model when the coefficients have not been changed, there is an `NA` value for the coefficient, so the rank of the model will then be missing by one unit. Therefore, when changing the coefficient, it is necessary to change the rank of the new model (plus 1 unit). We already know the rank factor was 3 before, so we just need to replace 3 by 4.

We can now guess the y-axis value based on the model.

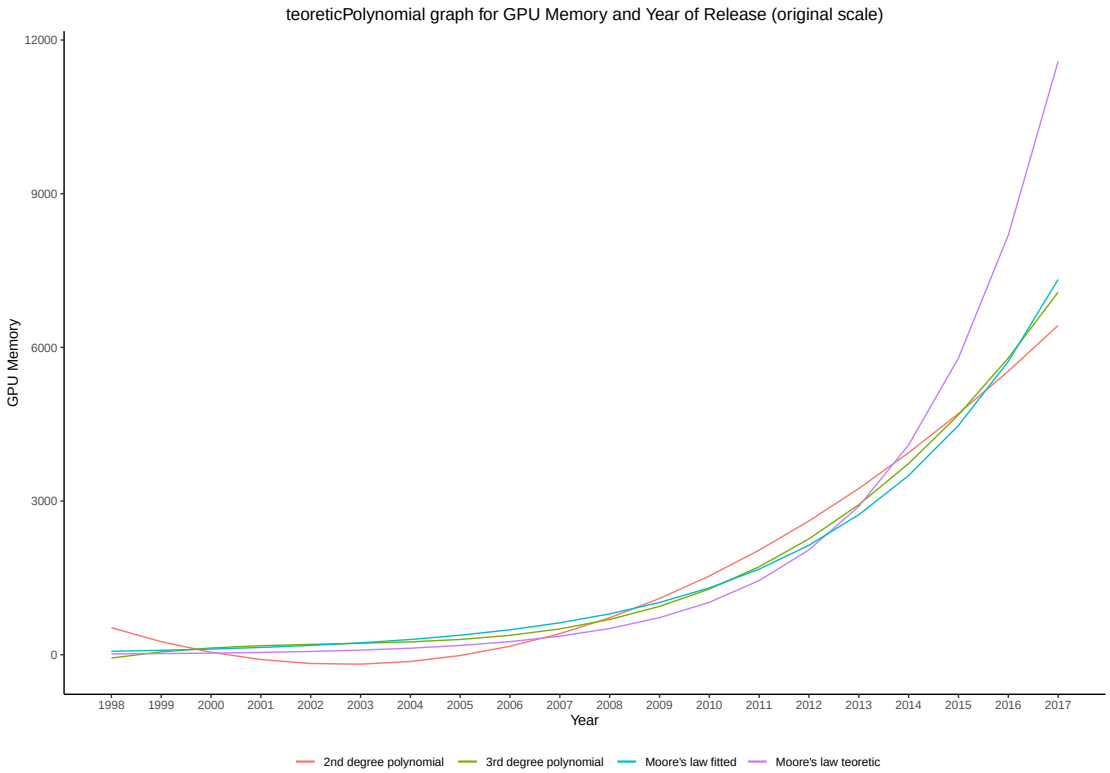
```
y_pred_lin_reg_3 <- predict(poly_3_mean, year_df, type="response")
```

Then save the predicted values to the data frame `df_extra`.

```
df_extra$Y_pred_lin_reg_3 <- y_pred_lin_reg_3
```

7.2.3 Graphs

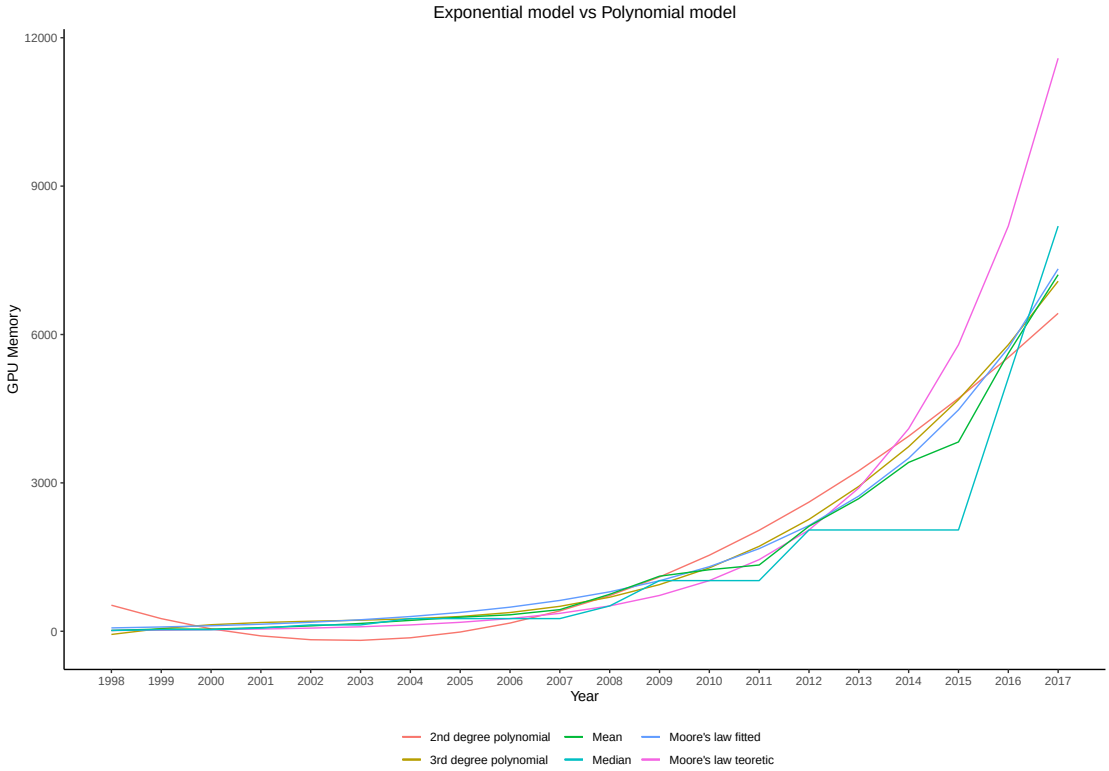
```
ggplot(df_extra, aes(x = Release_Year, y = log(Memory_Mean))) +
theme_classic() +
  scale_x_continuous(breaks = scales::pretty_breaks(n = 20)) +
  theme(legend.position='bottom', plot.title = element_text(hjust = 0.5)) +
  xlab("Year") + ylab("GPU Memory") + labs(colour="") +
  ggtitle("Polynomial graph for GPU Memory and Year of Release (logaritmic
  scale)") +
  geom_line(aes(y = y_pred_lin_reg_2, colour = "2nd degree polynomial"),
  n=1000) +
  geom_line(aes(y = y_pred_lin_reg_3, colour = "3rd degree polynomial"),
  n=1000) +
  geom_line(aes(y = Moore_Teoretic, colour = "Moore's law teoretic"),
  n=1000) +
  geom_line(aes(y = Moore_Fitted, colour = "Moore's law fitted"), n=1000)
```



The 2nd degree polynomial model [red line] and the 3rd degree polynomial model [green line] are shown in the figure above. Based on that, we can see that this model is better than the exponential model. Can be verified here or in the figure below. Note that Mean and Median are from the exponential model.

If we combine the two models above, we get the following figure.

```
ggplot(df_extra, aes(x = Release_Year, y = log(Memory_Mean))) +
theme_classic() +
  scale_x_continuous(breaks = scales::pretty_breaks(n = 20)) +
  theme(legend.position='bottom', plot.title = element_text(hjust = 0.5)) +
  xlab("Year") + ylab("GPU Memory") + labs(colour="") +
  ggtitle("Exponential model vs Polynomial model") +
  geom_line(aes(y = y_pred_lin_reg_2, colour = "2nd degree polynomial"),
  n=1000) +
  geom_line(aes(y = y_pred_lin_reg_3, colour = "3rd degree polynomial"),
  n=1000) +
  geom_line(aes(y = Moore_Teoretic, colour = "Moore's law teoretic"),
  n=1000) +
  geom_line(aes(y = Moore_Fitted, colour = "Moore's law fitted"), n=1000) +
  geom_line(aes(y = Memory_Mean, colour = "Mean"), n=1000) +
  geom_line(aes(y = Memory_Median, colour = "Median"), n=1000)
```



8 Prediction

In this section we will use our (linear regression) model to forecast the prices of several GPU models.

8.1 From dataset

- **Case 1**
  - `Boost_Clock = mean(Boost_Clock)`
  - `Core_Speed = mean(Core_Speed)`
  - `Max_Power = mean(Max_Power)`
  - `Memory = mean(Memory)`
  - `Memory_Bus = mean(Memory_Bus)`
  - `Memory_Speed = mean(Memory_Speed)`
  - `Shader = mean(Shader)`
  - `TMUs = mean(TMUs)`
- **Case 2**
  - `Boost_Clock = median(Boost_Clock)`

  - Core_Speed = median(Core_Speed)
  - Max_Power = median(Max_Power)
  - Memory = median(Memory)
  - Memory_Bus = median(Memory_Bus)
  - Memory_Speed = median(Memory_Speed)
  - Shader = median(Shader)
  - TMUs = median(TMUs)

- **Case 3**
  - Boost_Clock = max(Boost_Clock)
  - Core_Speed = max(Core_Speed)
  - Max_Power = max(Max_Power)
  - Memory = max(Memory)
  - Memory_Bus = max(Memory_Bus)
  - Memory_Speed = max(Memory_Speed)
  - Shader = max(Shader)
  - TMUs = max(TMUs)

For the purpose of predicting the GPU prices in the 3 specific cases above:

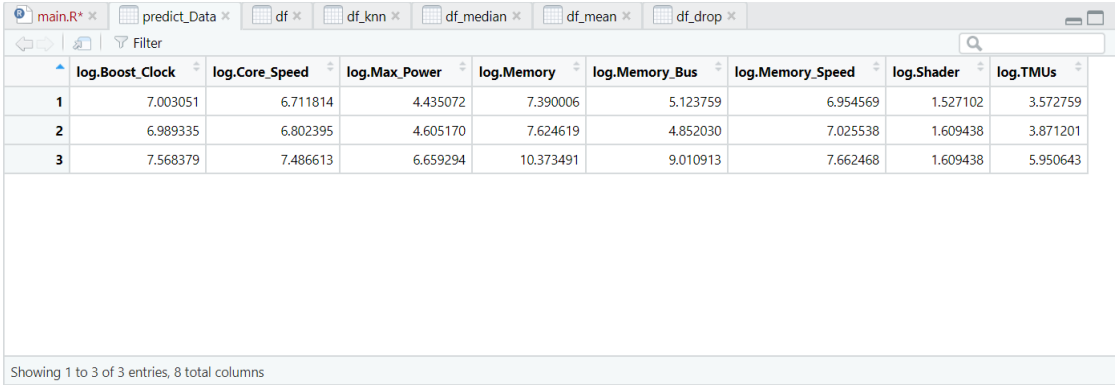
- **Step 1**: Create a data frame called `predict_Data` containing 3 observations of 8 independent variables (as listed above): `Boost_Clock`, `Core_Speed`, `Max_Power`, `Memory`, `Memory_Bus`, `Memory_Speed`, `Shader`, `TMUs`.
- **Step 2**: Call the `predict()` function with the following syntax:

  `predict(object, newdata, interval, level=0.95)`

  where:
  - `object`: A model object for which predictions are desired. We will call our `lmPrice` model.
  - `newdata`: The input data to predict the values.
  - `interval=c("none", "confidence", "prediction")`: type of interval calculation.
  - `level`: The confidence level. By default, this parameter equals to 0.95.

```
predict_Data = data.frame(
  log.Boost_Clock = c(mean(df[, 'log.Boost_Clock']), median(df[,
  'log.Boost_Clock']), max(df[, 'log.Boost_Clock'])),
  log.Core_Speed = c(mean(df[, 'log.Core_Speed']), median(df[,
  'log.Core_Speed']), max(df[, 'log.Core_Speed'])),
  log.Max_Power = c(mean(df[, 'log.Max_Power']), median(df[,
  'log.Max_Power']), max(df[, 'log.Max_Power'])),
  log.Memory = c(mean(df[, 'log.Memory']), median(df[, 'log.Memory']),
  max(df[, 'log.Memory'])),
  log.Memory_Bus = c(mean(df[, 'log.Memory_Bus']), median(df[,
  'log.Memory_Bus']), max(df[, 'log.Memory_Bus'])),
```

```
  log.Memory_Speed = c(mean(df[, 'log.Memory_Speed']), median(df[, 'log.Memory_Speed']), max(df[, 'log.Memory_Speed'])),
  log.Shader = c(mean(df[, 'log.Shader']), median(df[, 'log.Shader']), max(df[, 'log.Shader'])),
  log.TMUs = c(mean(df[, 'log.TMUs']), median(df[, 'log.TMUs']), max(df[, 'log.TMUs']))
)

predict(lmPrice, predict_Data, interval = "confidence", level = 0.95)
```



	log.Boost_Clock	log.Core_Speed	log.Max_Power	log.Memory	log.Memory_Bus	log.Memory_Speed	log.Shader	log.TMUs
1	7.003051	6.711814	4.435072	7.390006	5.123759	6.954569	1.527102	3.572759
2	6.989335	6.802395	4.605170	7.624619	4.852030	7.025538	1.609438	3.871201
3	7.568379	7.486613	6.659294	10.373491	9.010913	7.662468	1.609438	5.950643

Showing 1 to 3 of 3 entries, 8 total columns

```
       fit       lwr       upr
1 5.341435 5.330881 5.351988
2 5.281895 5.265958 5.297833
3 6.981385 6.882778 7.079993
```

We can use the `exp()` function to invert the logarithm to get the original value.

```
exp(predict(lmPrice, predict_Data, interval = "confidence", level = 0.95))
```

```
         fit       lwr        upr
1   208.8121  206.6200   211.0275
2   196.7424  193.6316   199.9032
3 1076.4082 975.3316 1187.9597
```

- **Case 1**: The GPU's price is around $208.8121 (from $206.6200 to $211.0275).
- **Case 2**: The GPU's price is around $196.7424 (from $193.6316 to $199.9032).
- **Case 3**: The GPU's price is around $1076.4082 (from $975.3316 to $1187.9597).

8.2 From real life

We try to predict the price of an Nvidia RTX 3090. Based on the specifications described here, we feed the data into the `lmPrice` model to predict the price.

```
spec_RTX_3090 = data.frame(
  log.Boost_Clock = log(1695),
  log.Core_Speed = log(1395),
  log.Max_Power = log(350),
  log.Memory = log(24576),
  log.Memory_Bus = log(384),
  log.Memory_Speed = log(1219),
  log.Shader = log(10496),
  log.TMUs =  log(328)
)

exp(predict(lmPrice, spec_RTX_3090, interval = "confidence", level = 0.95))
```

```
       fit      lwr       upr
1 8.13446 4.39761 15.04668
```

From the above results, we get a pretty bad result. Due to the fact that a new RTX 3090 has a MSRP (Manufacturer's Suggested Retail Price) price announced by Nvidia of $1499. However, our model only offers a price somewhere of $8.13446. We think that much of a difference lies in filling in the `NA` values with k-NN.
And so we re-trained the model without changing any data to see if there was any improvement.

```
df = df_drop

df['log.Boost_Clock'] <- log(df['Boost_Clock'])
df['log.Core_Speed'] <- log(df['Core_Speed'])
df['log.Max_Power'] <- log(df['Max_Power'])
df['log.Memory'] <- log(df['Memory'])
df['log.Memory_Bus'] <- log(df['Memory_Bus'])
df['log.Memory_Speed'] <- log(df['Memory_Speed'])
df['log.Release_Year'] <- log(df['Release_Year'])
df['log.Release_Price'] <- log(df['Release_Price'])
df['log.Shader'] <- log(df['Shader'])
df['log.TMUs'] <- log(df['TMUs'])

lmPrice = lm(log.Release_Price ~ log.Boost_Clock + log.Core_Speed +
log.Max_Power + log.Memory + log.Memory_Bus + log.Memory_Speed + log.Shader
+ log.TMUs, df)

exp(predict(lmPrice, spec_RTX_3090, interval = "confidence", level = 0.95))
```

```
        fit      lwr      upr
1 1565.946 1310.683 1870.922
```

Fortunately, the wind has changed direction. From the above results, the price that the model predicts is very close to the actual price ($1565.946 ≈ $1499). The spread is $66.946, which is acceptable.

9 Report goal

Through working on this project and writing this report, we have achieved:

- Learn how to handle data and deal with data loss or corruption. At the same time, we also learned graphs related to statistics and how to plot them.
- Learn how to use R in RStudio, and apply it to solve the given data.
- Learn how to transition knowledge from theory to practice as well as use appropriate testing method in research.
- Learn how to research new knowledge, applying them and implementing the problem solving process facing various obstacles on the way.
- Learn how to work as a group, to help and support each other to achieve specific goals.

In addition to that, due to the limited personal capacity and knowledge of each member, the report cannot avoid unpredictable errors. Therefore, our group is looking forward to receiving contributions and comments so that we can review and consolidate our knowledge.

10 References

References

[1] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, 7th ed. Kendallville: Wiley, 2018.

[2] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*, 9th ed. Boston, MA: Cengage Learning, 2016.

[3] T. D. Nguyen and D. H. Nguyen, *Probability – Statistics and Data Analysis*. Ho Chi Minh City: VNUHCM Press, 2020.

[4] John Verzani, *simpleR – Using R for Introductory Statistics*. [Online]. Available: https://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf.

[5] *Computer Parts (CPUs and GPUs)*, ilissek, Sep. 9, 2017. [Online]. Available: https://www.kaggle.com/iliassekkaf/computerparts.

[6] P. Jonsson and C. Wohlin, "An evaluation of k-nearest neighbour imputation using Likert data," *10th International Symposium on Software Metrics, 2004. Proceedings.*, 2004, pp. 108-118, doi: 10.1109/METRIC.2004.1357895.

[7] K. S. Htoon, *Log Transformation: Purpose and Interpretation*, Feb. 29, 2020. Accessed on: Nov. 12, 2021. [Online]. Available: https://medium.com/@kyawsawhtoon/log-transformation-purpose-and-interpretation-9444b4b049c9.

[8] A. Soetewey, *Do my data follow a normal distribution? A note on the most widely used distribution and how to test for normality in R*, Jan. 29, 2020. Accessed on: Nov. 15, 2021. [Online]. Available: https://statsandr.com/blog/do-my-data-follow-a-normal-distribution-a-note-on-the-most-widely-used-distribution-and-how-to-test-for-normality-in-r/.

[9] R. Bevans, *An introduction to the two-way ANOVA*, Mar. 20, 2020. Accessed on: Nov. 14, 2021. [Online]. Available: https://www.scribbr.com/statistics/two-way-anova/.

[10] A. Soetewey, *Chi-square test of independence in R*, Jan. 27, 2020. Accessed on: Nov. 15, 2021. [Online]. Available: https://statsandr.com/blog/chi-square-test-of-independence-in-r/.

[11] *Chi-Square Test of Independence*. [Online]. Available: https://libguides.library.kent.edu/spss/chisquare.

[12] B. Ghosh, *Multicollinearity in R*, Sep. 29, 2017. Accessed on: Nov. 12, 2021. [Online]. Available: https://datascienceplus.com/multicollinearity-in-r/.

[13] A. Kassambara, *Multicollinearity Essentials and VIF in R*, Nov. 3, 2018. Accessed on: Nov. 12, 2021. [Online]. Available: http://www.sthda.com/english/articles/39-regression-model-diagnostics/160-multicollinearity-essentials-and-vif-in-r/.

[14] K. Yang, J. Tu and T. Chen, "Homoscedasticity: an overlooked critical assumption for linear regression", 2019;32:e100148. doi: 10.1136/gpsych-2019-100148.