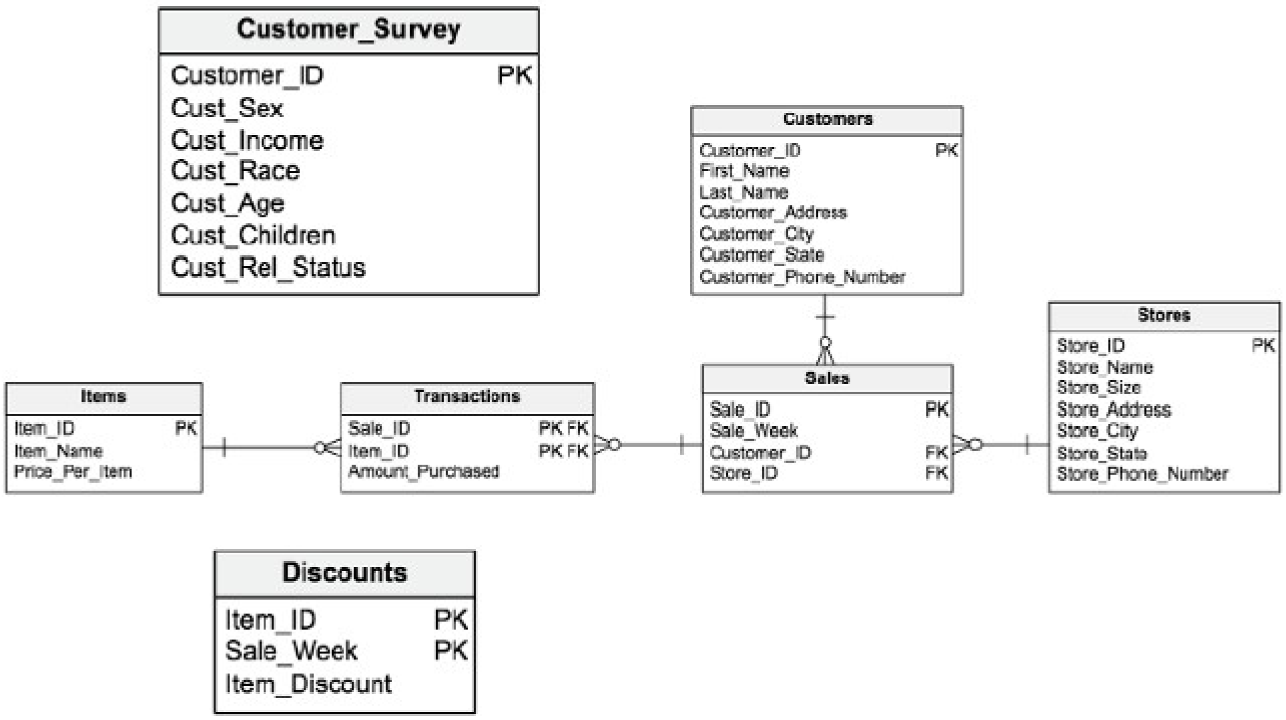
Object: For a given database for a Grocery Store Chain, several exploratory analysis will be done, and by maschine learning algorithms, it will be tried to perform decision makings.

In figure 1, there are some datas in a relational format located in MySql. Additionaly, there are two additonal tables they are Customer Surveys and Discounts tables.

Figure 1. Relational Database Structure in MySql



Connecting the MySql Database in RStudio:

First thing to do is to install the package for MySql connection. RMySql package let us connect to MySql environment and run queries through the R. With comment, we can take any data to R environement.

```
install.packages("RMySQL")
library(RMySQL)
```

Then run the library(RMySQL) to be able to use this library. After that connect to the database and the database with below code:

```
mydb = dbConnect(MySQL(), user='root', password='serdar27', dbname='transaction_database', host='127.0.0.1')
```

You can list all tables in the transaction_database with running following code:

```
dbListTables(mydb)
```

The result is:

```
[1] "clusters"       "customer_survey" "customers"       "discounts"       "items"
[6] "sales"          "stores"          "transactions"    "zipdata"
```

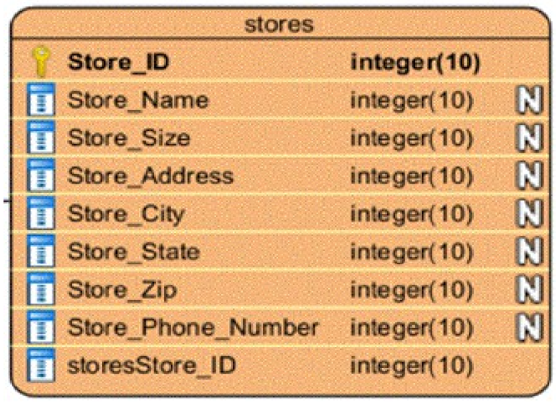
We can also list the fields of any table, Let's look what is in stores table as:

```
dbListFields(mydb, 'stores')
```

The result in RStudio Console is:

```
[1] "Store_ID"        "Store_Name"      "Store_Size"      "Store_Address"
[5] "Store_City"      "Store_State"     "Store_Zip"       "Store_Phone_Number"
```

Figure 2 Stores Table Attributes



Now we can run queris in RStudio. Let's count store amount, which is counting store id's in stores table, since every Store has unique ID and Store_ID is a primary key. We use dbSend() function for queries, and we fetch it to the RStudio environment. The result will be a dataframe, which is 1 to 1 dataframe for the query select count(*) from stores;. Then we take that one element using [1,1] index , and assign it to nb_store referring number of stores.

```
nb_store<-fetch(dbSendQuery(mydb, "select count(*) from stores;"))[1,1]
```

```
print(nb_store)
```

Result

```
[1] 12
```

So there are 12 stores in the database.We saved this in our Rstudio environment as nb_store.

Another way would be extracting all table to the RStudio environment by fetching, and count the rows of the table:

```
stores<-fetch(dbSendQuery(mydb, "select * from stores;"))
nrow(stores)
```

With the codes above, we have stores table in our RStudio environement.

In stores table, there are two important attributes: Store_City and Store_State. Let us find, how many cities has these stores. Remember that one city may have more than one store, and different states may have same city names with each other. That is why, we must select distinct city, state pairs.

```
nb_cities<-fetch(dbSendQuery(mydb, "select count(distinct store_city,store_state) from stores;"))[1,1]
print(nb_cities)
```

Result of printing nb_cities:

```
[1] 11
```

There are 12 stores in 11 cities. So it is easily understandable that one city has 2 stores.

On the other hand, we have stores table in RStudio environemnt. Let us find number of cities with R codes. There is also sqldf package of R, where you can use sql type of queries in R.

```
colnames(stores)
```

Result of the Code:

```
[1] "Store_ID"          "Store_Name"        "Store_Size"        "Store_Address"
[5] "Store_City"        "Store_State"       "Store_Zip"         "Store_Phone_Number"
```

Lets find unique pairs of Store_City and Store_State in R environemt, since we have fetched already the stores table:

```
unique(stores[c("Store_City","Store_State")])
```

The above code is extracted 2 column dataframe having unique City and State pairs. Let ' s count the rows with the code:

```
nrow(unique(stores[c("Store_City","Store_State")]))
```

Output in Console:

```
[1] 11
```

Let's find that city, that has 2 store. Define a string con that refers the whole SqlQuery as a string:

```
con<-"Select distinct store_city, store_state, count(*) from stores
group by store_city,
store_state order by count(*) desc;"
```

Now use the string con and fetch the query result to local;

```
fetch(dbSendQuery(mydb, con))
```

It will directly output the result in the RStudio console:

```
store_city        store_state    count(*)
1  Boston           MA            2
2  Springfield      MA            1
3  Pittsfield       MA            1
4  Somerville       MA            1
5  Boylston         MA            1
6  Malden           MA            1
7  Worcester        MA            1
8  Framingham       MA            1
9   Lowell          MA            1
10 Newburyport      MA            1
11 Salem            MA            1
```

So, Boston has 2 stores. And there is only one state in the stores table. To find this in R by R codes by using stores table which already in R environment, we can write following code. (we use pipe operator %>% which makes it easy. The %>% is read as "and then" and is way of listing your functions sequentially rather then nesting them.)

```
stores %>%
  distinct(Store_City,Store_State,Store_ID) %>%
  group_by(Store_City,Store_State) %>%
  summarize("Store Number"=n())
```

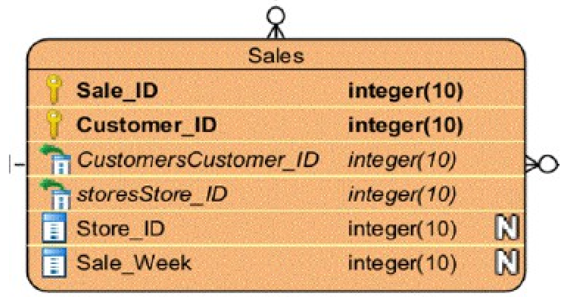
We get in the console as following in tibble class which can be converted to dataframe easily:

```
# A tibble: 11 x 3
# Groups:   Store_City [11]
   Store_City        Store_State     `Store Number`
   <chr>               <chr>                <int>
 1 Boston              MA                    2
 2 Boylston            MA                    1
 3 Framingham          MA                    1
 4 Lowell              MA                    1
 5 Malden              MA                    1
 6 Newburyport         MA                    1
 7 Pittsfield          MA                    1
 8 Salem               MA                    1
 9 Somerville          MA                    1
10 Springfield         MA                    1
11 Worcester           MA                    1
```

There are other ways to find it. R programming is very flexible.

Now fetch the sales table to the RStudio environment

Figure 3 Sales Attributes




```
sales<-fetch(dbSendQuery(mydb, "select * from sales;"),n=Inf)
total_sales<-nrow(sales)
```

And nrow(sales) gives us 31042 rows, we saved it as total_sales. To see top 5 rows, just write:

```
head(sales,5)
```

Result in console:

```
  Sale_ID Customer_ID Store_ID Sale_Week
1       1     2013154        4         1
2       2     4599139        5         1
3       3     6079324        1         1
4       4     3242746        3         1
5       5     4319355       11         1
```

The Sale_Week tells us the week of the year. There are 52 weeks in a year. You can write it in the console by Sql Query and fetching:

```
fetch(dbSendQuery(mydb, "Select count(distinct sale_week) from sales;"))
```

Result in Console:

```
  count(distinct sale_week)
1                        52
```

This means every week, stores have been opened and sold something because it equals to years week amount. If the result was 50, that would mean, stores had been 2 weeks closed or there had been no customer during these 2 weeks.
Another way to check it. We have sales table in environment, we can count distinct Sale_Week from the table. Let use n_distinct() function of dplyr library. Save it as nb_weeks which is number of weeks.

```
nb_weeks<-n_distinct(sales$Sale_Week)
nb_weeks
```

Result in Console
```
[1] 52
```

or another code to find it:

```
sapply(sales, function(Sale_Week) n_distinct(Sale_Week))['Sale_Week']
```

Result in Console

```
Sale_Week
       52
```

Now, we know that 1 year has 52 weeks, and in the database, there are datas for 52 distinct Sale Weeks. Therefore, Each week of the year, sales happened.

Let us find, for instance, for Store_ID=1, the average sales_per_week:

```
(fetch(dbSendQuery(mydb,"select count(*) sales_per_week from sales where store_id=1")))/52
```

Output in Console:

```
  sales_per_week
1          50.25
```

How would we do that with sales table in RStudio? We can use sum() function;

```
sum(sales$Store_ID == 1, na.rm=TRUE)/52
```

Output in Console:

```
[1] 50.25
```

Another way to count it:

```
nrow(sales[sales$Store_ID==1,])/52
```

Output in Console:

```
[1] 50.25
```

What is the Store Name for Store_ID=1, we can read it from Stores table:

```
stores[stores$Store_ID==1,]$Store_Name
```

Output in Console:

```
[1] "Chris's Corner"
```

Where is that Store?

```
stores[stores$Store_ID==1,]$Store_City
```

Output in Console:

[1] "Springfield"

As a result, Chris's Corner Store in city Springfield has an avaerage 50.25 sales per week.

Let's count sales per week for all stores and keep them in a dataframe called Aver_sales_per_week:

```
df<-sales %>%
  group_by(Store_ID) %>%
  summarize("Sales_per_Week"=n()/52)
```

Let's look its class:

```
class(df)
```

Output in Console which is tibble type:

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Let's convert the Tibble type to dataframe type:

```
Aver_sales_per_week = as.data.frame(df)
```

Let's look its class:

```
class(Aver_sales_per_week)
```

Output in Console:

```
[1] "data.frame"
```

Let's see the dataframe in console

```
Aver_sales_per_week
```

Output in Console:

	Store_ID	Sales_per_Week
1	1	50.25000
2	2	11.28846
3	3	32.63462
4	4	37.96154
5	5	37.86538
6	6	40.03846
7	7	21.23077
8	8	40.75000
9	9	90.76923
10	10	88.00000
11	11	68.73077
12	12	77.44231

We can add above dataframe further columns as Store Name, City and State retreiving them from Stores dataframe. Let's do it:

First we take Store_ID, Store_Name, Store_City and Store_State columns from stores dataframe and save them in a temporary dataframe df. In this way

```
rm(df)
df<-stores[c('Store_ID','Store_Name','Store_City','Store_State')]
```

And now merge it with Aver_sales_per_week dataframe with reference to Store_ID:

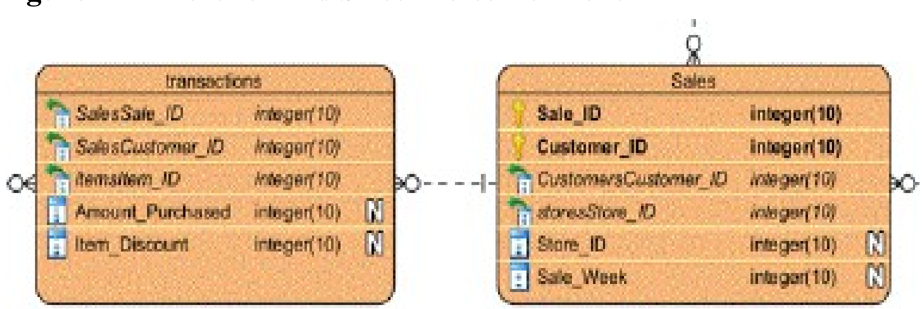
```
merge(Aver_sales_per_week,df,by='Store_ID')
```

When we run the line above, the output in the console:

	Store_ID	Sales-per_Week	Store_Name	Store_City	Store_State
1	1	50.25000	Chris's Corner	Springfield	MA
2	2	11.28846	Sheffi's Store	Pittsfield	MA
3	3	32.63462	Eva's Extravaganza	Boylston	MA
4	4	37.96154	Ahmad's Alley	Worcester	MA
5	5	37.86538	Delio's Deli	Framingham	MA
6	6	40.03846	Mary's Market	Lowell	MA
7	7	21.23077	Emma's Emporium	Newburyport	MA
8	8	40.75000	Marty's Mart	Salem	MA
9	9	90.76923	Brian's Bazaar	Boston	MA
10	10	88.00000	Sally's Shop	Somerville	MA
11	11	68.73077	Bill's Barter	Malden	MA
12	12	77.44231	Eddy's Exchange	Boston	MA

Beside dplyr package, there is another package called sqldf which allows user to code in sql query type. In short, sqldf is an R package for running SQL statements on R data frames, optimized for convenience.

Figure 4 Transaction and Sales Tables Relations



As seen in Figure above, there is one to many relation between Sales and transactions table.

Let's install sqldf package

```
install.packages("sqldf")
```

Now call the library to your editor. Just detach RMySQL package, since it blocks sql statement to be used for local tables in R. Just remember, if you want to work on MySql, you must call the RMySQL library again.

```
library(sqldf)
require(sqldf)
detach("package:RMySQL", unload=TRUE)
```

Let's try it. We have Aver_sales_per_week and stores tables in RStudio environemt.

```
sqldf('select Aver_sales_per_week.Store_ID,Sales_per_Week,
    Store_Name,Store_City,Store_State
    from Aver_sales_per_week left join stores on
    Aver_sales_per_week.Store_ID=stores.Store_ID ')
```

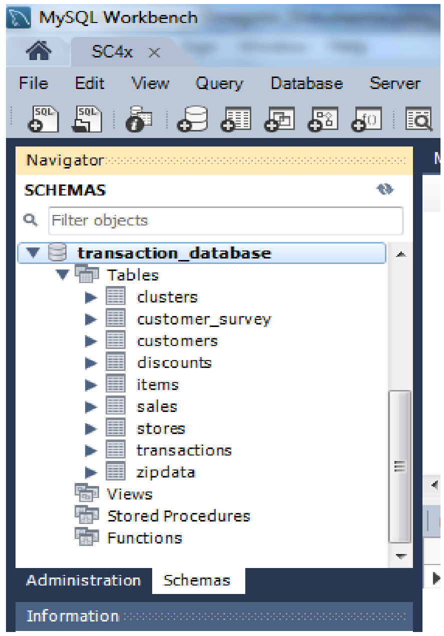
Output in Console:

	Store_ID	Sales_per_Week	Store_Name	Store_City	Store_State
1	1	50.25000	Chris's Corner	Springfield	MA
2	2	11.28846	Sheffi's Store	Pittsfield	MA
3	3	32.63462	Eva's Extravaganza	Boylston	MA
4	4	37.96154	Ahmad's Alley	Worcester	MA
5	5	37.86538	Delio's Deli	Framingham	MA
6	6	40.03846	Mary's Market	Lowell	MA
7	7	21.23077	Emma's Emporium	Newburyport	MA
8	8	40.75000	Marty's Mart	Salem	MA
9	9	90.76923	Brian's Bazaar	Boston	MA
10	10	88.00000	Sally's Shop	Somerville	MA
11	11	68.73077	Bill's Barter	Malden	MA
12	12	77.44231	Eddy's Exchange	Boston	MA

For further study in this case study, we will work on MySql server. Thats why we need to use library RMySQL and we will work on SQL server.Let's reconnect to MySql Server:

```
mydb = dbConnect(MySQL(), user='root', password='serdar27', dbname='transaction_database', host='127.0.0.1')
```

Figure 5 Transaction DB on MySQL



Send your query to MySql as following:

```
qry<-dbSendQuery(mydb," CREATE TABLE SalesData AS (SELECT s.Customer_ID, s.Sale_ID,
s.Store_ID,
SUM(t.Amount_Purchased) CountItems,
SUM(t.Amount_Purchased*i.Price_Per_Item*(1 -d.Item_Discount)) SalePrice
FROM Sales s
RIGHT JOIN Transactions t
ON t.Sale_ID= s.Sale_ID
LEFT JOIN Discounts d
ON d.Sale_Week= s.Sale_Week AND d.Item_ID= t.Item_ID
LEFT JOIN Items i
ON i.Item_ID= t.Item_ID
GROUP BY s.Sale_ID, s.Customer_ID, s.Store_ID);")
```

In the query above for columns has been created. First one is Customer_ID that tells which customer did the shopping. The second is Sale_ID, that is the id of shopping, it can be thought as receipt number given by cashier. The third attribute is Store_ID, tells in which store shopping happened. The fourth attribute is CountItems, that tells how many items is in the receipt. Finally the SalePrice, it is total receipt fee. Weekly discount ratio on price is calculated by the value taken from Discounts table.

Close pending with following code, because you should stop pending:

```
dbClearResult(qry)
```

See the head of sales data (salaesdata is not in local) :

```
head(fetch(dbSendQuery(mydb, "select * from salesdata"),n=10),5)
```

Output;

	Customer_ID	Sale_ID	Store_ID	CountItems	SalePrice
1	4599139	2	5	48	159.424
2	3242746	4	3	50	200.850
3	6143606	12	11	32	148.970
4	3111830	13	9	42	165.421
5	2439039	25	12	58	210.440

As in figure below, new created table is in the MySql database. It would better to create it as temporary table, but sending query for creating temporary tables via R code is problematic.

Figure 6 Transaction DB in MySql

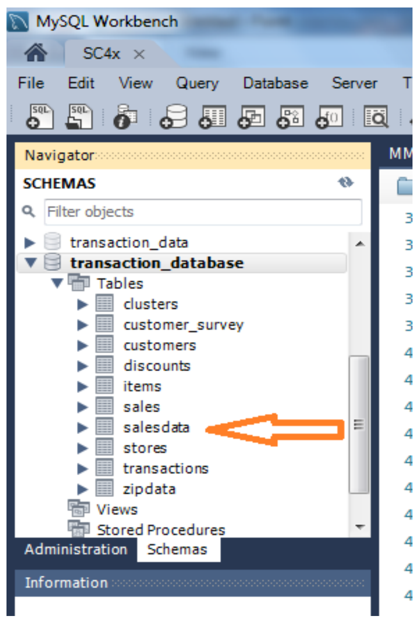
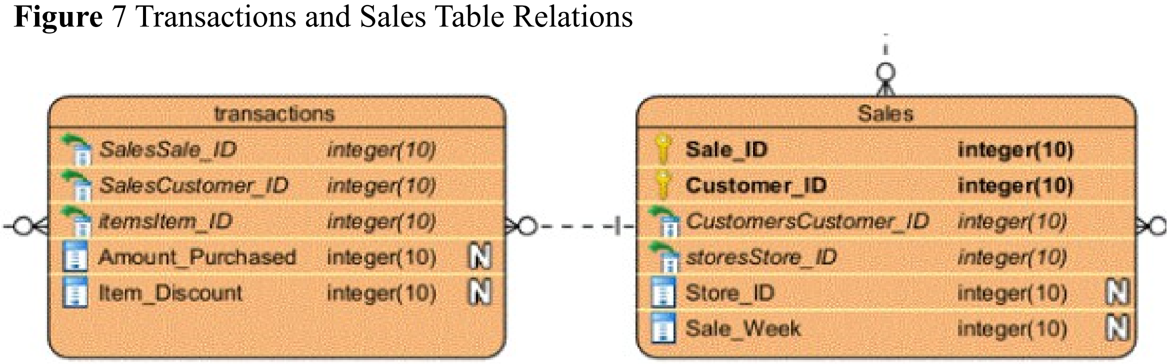


Figure 7 Transactions and Sales Table Relations



Transactions table has many to one relation to sales table. Amount_Purchased in transactions table tells the number of items sold in different levels such as customer, sale, item etc.
For Store_ID=1, which is Chris's Corner Store in Springfield, we can extract number of items sold with regarding to each sale. Let run the query and save the table in R environment.

```
result<-dbSendQuery(mydb,"select sales.sale_id,
                    sum(transactions.amount_purchased) as Num_Items
                    from sales left join transactions on
                    sales.sale_id=transactions.sale_id
                    where sales.store_id='1'
                    group by sales.sale_id order by Num_items desc;")
```

The query result is kept in result object. Let's fetch it a new table called Chris.

```
Chris<-fetch(result,n=-1)
```

The Penning must be stopped, otherwise it will hinder later queris.

```
dbClearResult(result)
```

The table Chris has 2613 observations with 2 coloumns. Let's look its head:

```
head(Chris,5)
```

The output in console:

	sale_id	Num_Items
1	27933	105
2	27265	82
3	25261	81
4	8600	81
5	19274	81

How many rows are there in Chris dataframe? This tells us the number of saler:

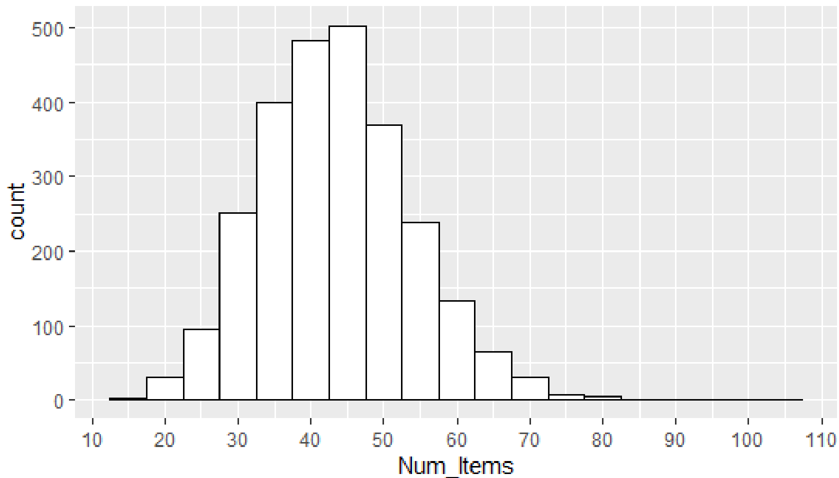
```
nrow(Chris)
```

Output : [1] 2613

Let's see distribution graph drawn by ggplot2 library:
(http://www.cookbook-r.com/Graphs/Plotting_distributions_(ggplot2)/)

```
library(ggplot2)

ggplot(Chris, aes(x=Num_Items)) + scale_x_continuous(breaks=seq(0,120,10))+
   geom_histogram(binwidth=5, colour="black", fill="white")
```

Figure 8 Distribution of Number of Items per Sales at Chris' Store

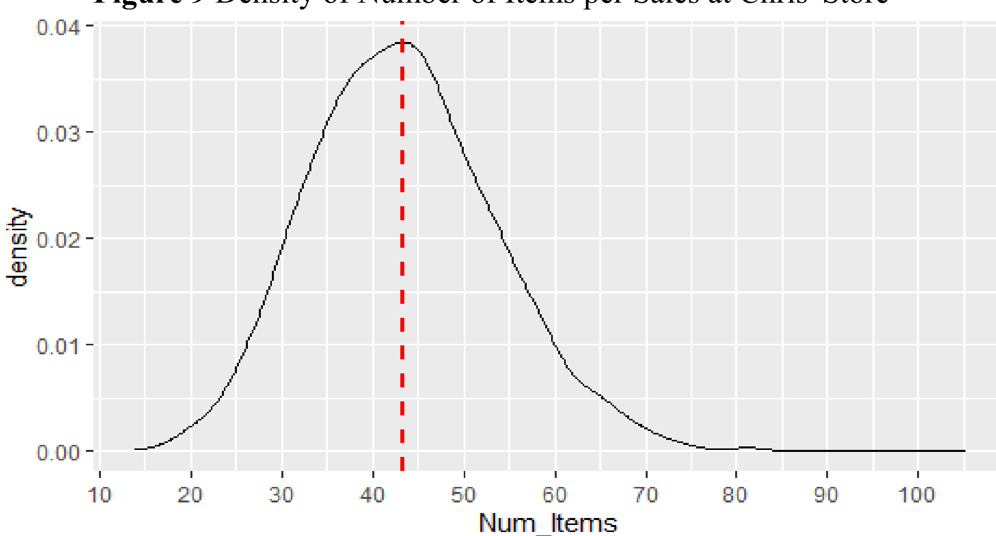


As seen in figure, about 500 sales has about 45 items sold at Chris Store.

Let's smooth that graph and draw density curve with a mean line:

```
ggplot(Chris, aes(x=Num_Items)) +
  geom_density() + scale_x_continuous(breaks=seq(0,120,10))+
  geom_vline(aes(xintercept=mean(Num_Items, na.rm=T)),
          color="red", linetype="dashed", size=1)
```

Figure 9 Density of Number of Items per Sales at Chris' Store



The mean value can be calculated as :

```
mean(Chris$Num_Items,na.rm=T)
```

Output :

```
[1] 43.37275
```

Let's find best item sold in Chris Store. To do that, we need new query in item level as following:

First save the Query as con in a string type

```
con<-('SELECT I.Item_Name, SUM(T.Amount_Purchased) TotalPurchased
FROM Transactions T LEFT JOIN Items I
ON T.Item_ID=I.Item_ID LEFT JOIN Sales S
ON S.Sale_ID=T.Sale_ID
WHERE S.Store_ID=1
GROUP BY I.Item_ID
ORDER BY TotalPurchased DESC;')
```

Let's run the query and save the resulted dataframe as Chris_items :

```
result<-dbSendQuery(mydb,con)
Chris_items<-fetch(result,n=-1)
dbClearResult(result)
```

Let"s look head of Chris_items dataframe:

```
head(Chris_items,5)
```

Output:

	Item_Name	TotalPurchased
1	Meatloaf Mix	1501
2	Radish	1404
3	Pomegranate	1391
4	Oranges	1301
5	Oatmeal	1301

And,

```
nrow( Chris_items,5)
```

Output

```
[1] 200
```

So among 200 items being sold at Chris' Store, Meatloaf Mix is the most sold product.

Let's investigate more on this material. We can this items weekly sales at Chris Store:

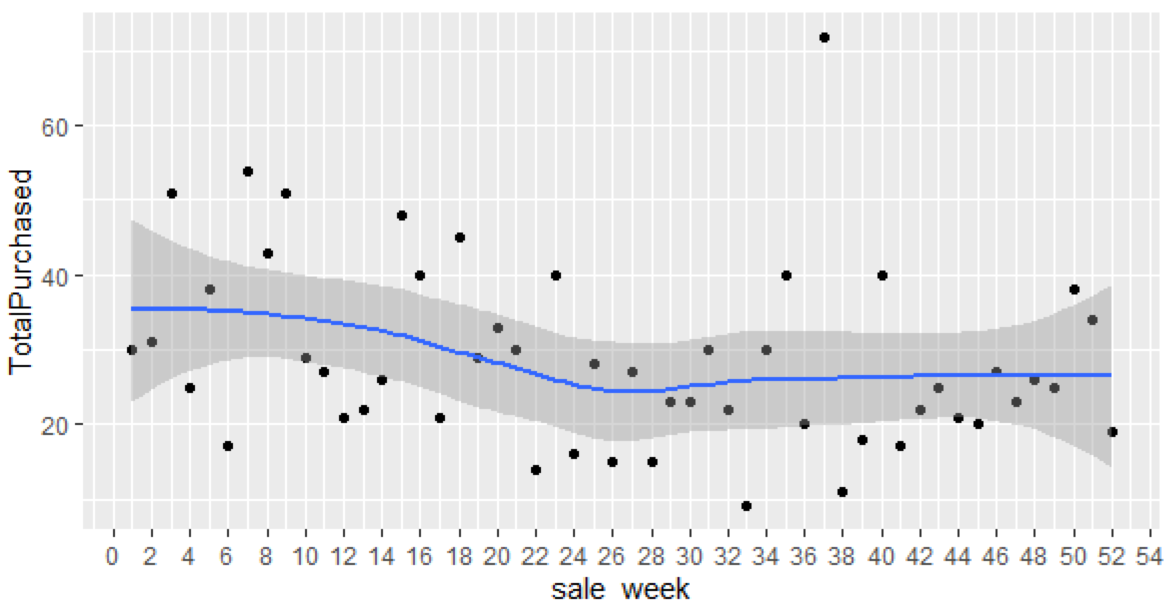
```
con<-"select s.sale_week,  SUM(T.Amount_Purchased) TotalPurchased
FROM sales s left join Transactions T
ON T.Sale_ID=S.Sale_ID left join Items I
ON T.Item_ID=I.Item_ID
WHERE S.Store_ID=1 and I.Item_name='Meatloaf Mix'
group by s.sale_week
order by s.sale_week asc;"

result<-dbSendQuery(mydb,con)
Chris_Meatloaf<-fetch(result,n=-1)
dbClearResult(result)
```

We saved the dataframe of Meatloaf in weekly bucket through the year as Chris_Meatloaf dataframe.

Let's graph it's scatter plot to see inside:

```
ggplot(Chris_Meatloaf, aes(x=sale_week,y=TotalPurchased ))+
  scale_x_continuous(breaks=seq(0,55,2))+
  geom_point()+geom_smooth()
```

Figure 10 Total Purchased Number of Meatloaf per Sales Week at Chris' Store

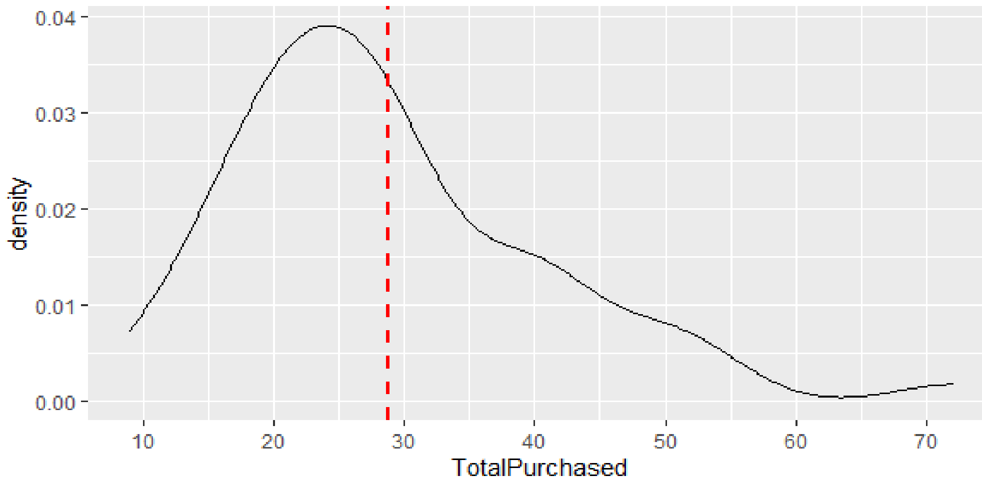


The blue line shows the trend line, As seen, from first week till 25th week, there is a decrease and then balance.

Let's draw it's density distribution graph:

```
ggplot(Chris_Meatloaf, aes(x=TotalPurchased)) +
  geom_density() + scale_x_continuous(breaks=seq(0,80,10))+
  geom_vline(aes(xintercept=mean(TotalPurchased, na.rm=T)),
           color="red", linetype="dashed", size=1)
```

Figure 11 Density Distribution of Weekly Total Purchaed Numbers for Meatloaf at Chris



It is almost normally distributed. It has positive skewness. It is indeed Gama Distribution.

Let's find Mean and Standard Deviation:

```
mean(Chris_Meatloaf$TotalPurchased)
sd(Chris_Meatloaf$TotalPurchased)
```

Outputs respectively:

```
[1] 28.86538
[1] 12.2523
```

Assuming it is normally distributed and if the store keeps 45 Meatloaf Mix weekly, then cumulative density function of the normal distribution in R:

```
pnorm(45,28.86538,12.2523)
```

Output:

```
[1] 0.906058
```

It means with 45 item Meatloaf Mix stock, with %91 percent the store fulfills the orders weekly.

Now let us create a new table called CustomerData. It will take several SalesData as seen below.

```
con<-"CREATE  TABLE CustomerData AS(
SELECT Customer_ID, AVG(CountItems) AvgCountItems,
SUM(SalePrice)/SUM(CountItems) AvgItemPrice,
AVG(SalePrice) AvgSalePrice,
COUNT(*) Trips,
SUM(SalePrice) TotalRevenue
FROM SalesData
GROUP BY Customer_ID);"


result<-dbSendQuery(mydb,con)
dbClearResult(result)
```

Let's the head of new dataframe:

```
result<-dbSendQuery(mydb, "select * from CustomerData")
head(fetch(result,n=10),5)
dbClearResult(result)
```

	Customer_ID	AvgCountItems	AvgItemPrice	AvgSalePrice	Trips	TotalRevenue
1	4599139	51.7647	3.872785	200.4736	17	3408.051
2	3242746	55.5556	4.010527	222.8071	18	4010.527
3	6143606	44.1250	3.931045	173.4574	8	1387.659
4	3111830	40.1250	3.760607	150.8944	8	1207.155
5	2439039	53.8889	3.718355	200.3780	9	1803.402

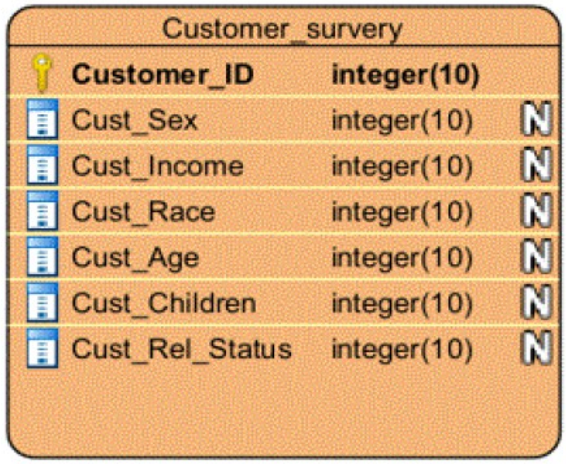
To understand the table above, lets extract head of Salesdata for customer 4599139 for 5 row.

	Customer_ID	Sale_ID	Store_ID	CountItems	SalePrice
1	4599139	2	5	48	159.424
2	4599139	1039	10	72	266.127
3	4599139	4749	7	35	129.317
4	4599139	3391	11	84	337.290
5	4599139	5104	9	49	187.323

So For CustomerData, for each cutomer, AvgCountItems is the average of CountItems from salesdata table. AvgSalePrice is the average of SalePrice attribute from Salaesdata. AvgItemPrice on the other hand is sum of SalePrice divided by sum of CountItems from Salaesdata. Trips is the count of Sale_ID's for each customer, assuming that he or she made shopping for every store visit. And finally, TotalRevenue is the sum of SalePrice in Salesdata for each customer. It could be ordered by TotalRevenue in descending way to see which customer spent best.

In the database, There is a survey table. In the survey, several datas are saved for customers which can be seen in figure below.

Figure. 12 Customer Survey Attributes in DB



A new query can be done using lately created CustomerData and this Customer_Survey table, so that these valuable informations can be merged with customer transaction datas. The new table is CustDataSurvey as below:

```
con<-"CREATE TABLE CustDataSurvey AS(SELECT cd.TotalRevenue, cd.Customer_ID,
cs.Cust_Sex, cs.Cust_Income, cs.Cust_Race, cs.Cust_Age,cs.Cust_Children,cs.Cust_Rel_Status
FROM CustomerData cd
LEFT JOIN Customer_Survey cs
ON cd.Customer_ID = cs.Customer_ID);"

result<-dbSendQuery(mydb,con)
dbClearResult(result)
```

Let's see the head of CustDataSurvey.

	TotalRevenue	Customer_ID	Cust_Sex	Cust_Income	Cust_Race	Cust_Age	Cust_Children	Cust_Rel_Status
1	3408.051	4599139	Male	70800	White	25-44	Has_Child(ren)	Married
2	4010.527	3242746	Female	85700	White	45-64	Has_no_Child(ren)	Married
3	1387.659	6143606	Female	49500	Black	45-64	Has_no_Child(ren)	Not_Married
4	1207.155	3111830	Male	45500	White	45-64	Has_no_Child(ren)	Married
5	1803.402	2439039	Male	95900	White	45-64	Has_no_Child(ren)	Married

Now by using both CustDataSurvey and CustomerData, several explatory graphs can be drawn.

Let's fetch both of them to the local RStudio environment. Both tables ordered by customer-id. Therefore, the tables row's are alligned with each other.

```
result<-dbSendQuery(mydb,"select * from customerdata order by Customer_ID")
customerdata<-fetch(result,n=-1)
dbClearResult(result)
```

```
result<-dbSendQuery(mydb,"select * from CustDataSurvey order by Customer_ID")
CustDataSurvey<-fetch(result,n=-1)
dbClearResult(result)
```

The common attribute is Customer_ID for two dataframes. Let join them on their Customer_ID.

```
Customer<-merge(customerdata,CustDataSurvey,by='Customer_ID')
```

By using Customer dataframe, Many explatory graphs can be drawn.

1) **Average Item Count by Gender**

First select Average_Item_Count by gender male as following, result will be vector:

```
a<-c(Customer[Customer$Cust_Sex=='Male',]$AvgCountItems)
```

Secondly, build a dataframe with two Columns, first column is gender and second is a vector from above:

```
dat_male <- data.frame(Gender = factor(rep(c("Male"))), AvgCountItems = c(a))
```

Now repeating same steps for Female gender:

```
b<-c(Customer[Customer$Cust_Sex=='Female',]$AvgCountItems)
```

```
dat_female<-data.frame(Gender = factor(rep(c("Female"))), AvgCountItems = c(b))
```

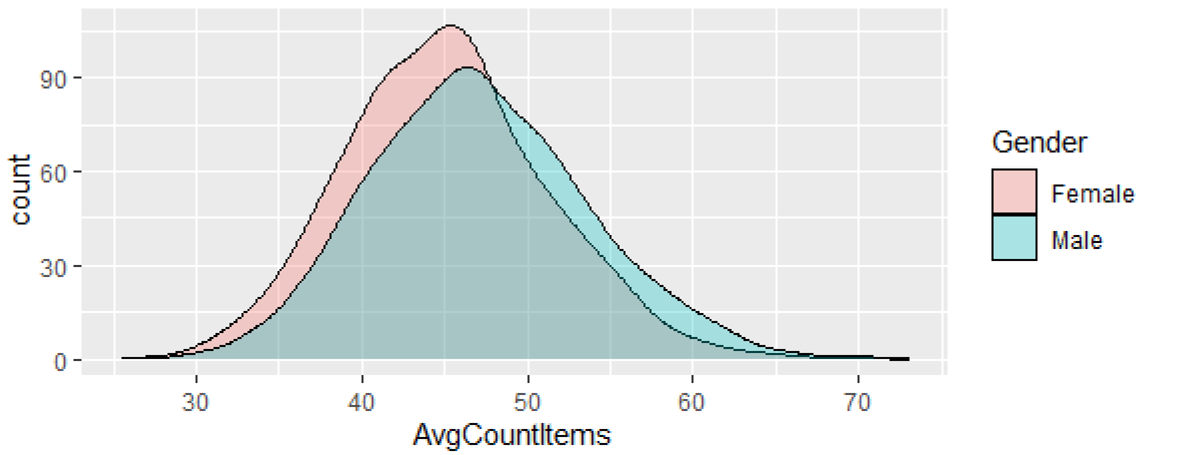
As a result, there are two dataframe where first column is gender , and second AvgCountItems. With rbind() function, they can be combined by their rows:

```
dat<-rbind(dat_female,dat_male)
```

The dat dataframe will be input to ggplot:

```
ggplot(dat, aes(x=AvgCountItems, fill=Gender))+geom_density(aes(y = ..count..), stat="density", alpha=0.3)
```

Figure 13 Average Items Number's Frequency by Genders



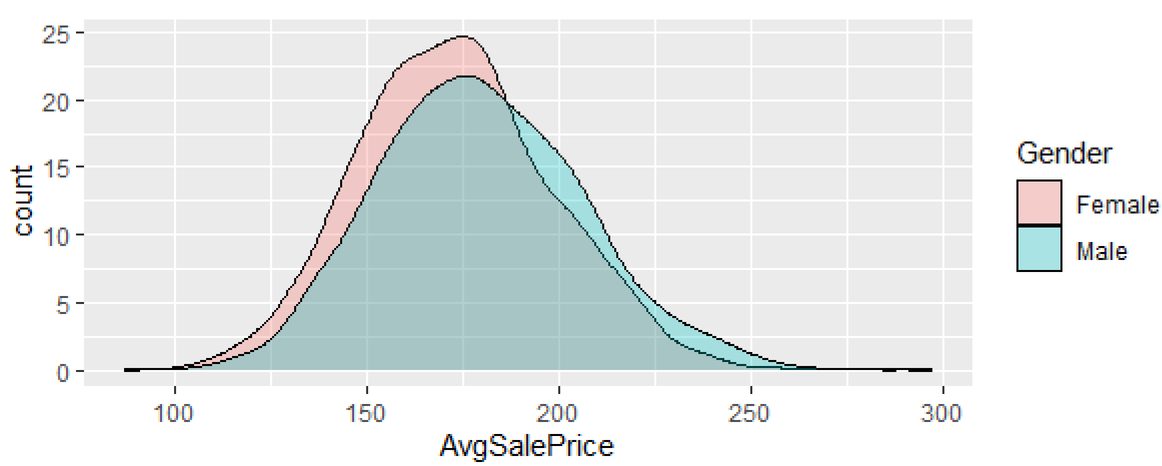
The figure above tells that till 45~48 average count items, female customers appeared more than male customers. On the other hand, the customers who shopped average 60 items during the year, are more likely male customers.

2) Average Sale Price by Gender

In same manner like previous graph, the graph for Average Sale Price by Gender can be drawn as:

```
a<-c(Customer[Customer$Cust_Sex=='Male',]$AvgSalePrice )
b<-c(Customer[Customer$Cust_Sex=='Female',]$AvgSalePrice )
dat_male <- data.frame(Gender = factor(rep(c("Male"))), AvgSalePrice  = c(a))
dat_female<-data.frame(Gender = factor(rep(c("Female"))), AvgSalePrice  = c(b))
dat<-rbind(dat_female,dat_male)
ggplot(dat,aes(x=AvgSalePrice, fill=Gender)) +geom_density(aes(y = ..count..),stat="density",
alpha=0.3)
```

Figure 14 Average Sale Price Frequency by Gender

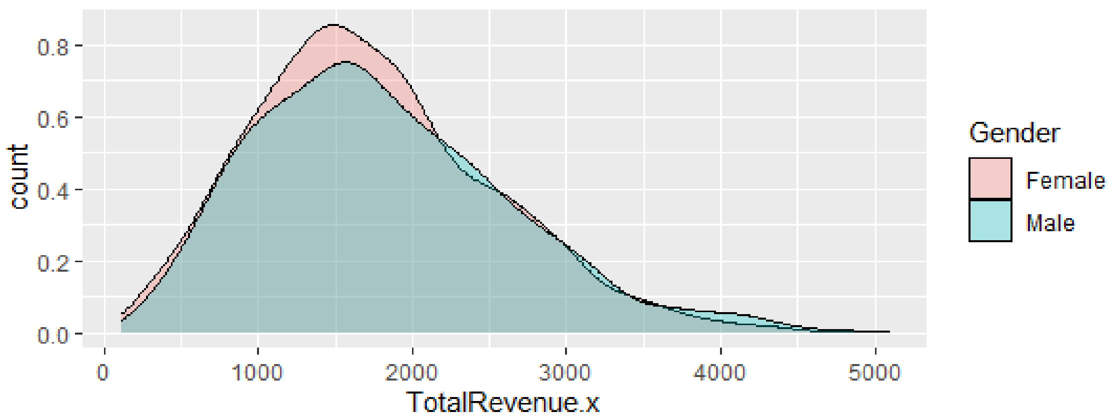


It is seen from Figure 14 above that, the number of big spenders are more male than female. For customer group who spend around 175, which is mean, there are more female customers than male customer.

3) Total Revenue by Gender

```
a<-c(Customer[Customer$Cust_Sex=='Male',]$TotalRevenue.x)
b<-c(Customer[Customer$Cust_Sex=='Female',]$TotalRevenue.x)
dat_male <- data.frame(Gender = factor(rep(c("Male"))), TotalRevenue.x = c(a))
dat_female<-data.frame(Gender = factor(rep(c("Female"))), TotalRevenue.x = c(b))
dat<-rbind(dat_female,dat_male)
ggplot(dat, aes(x=TotalRevenue.x, fill=Gender)) +  geom_density(aes(y = ..count..), stat="density",
alpha=0.3)
```

Figure 15 Total Revenue Frequency by Gender

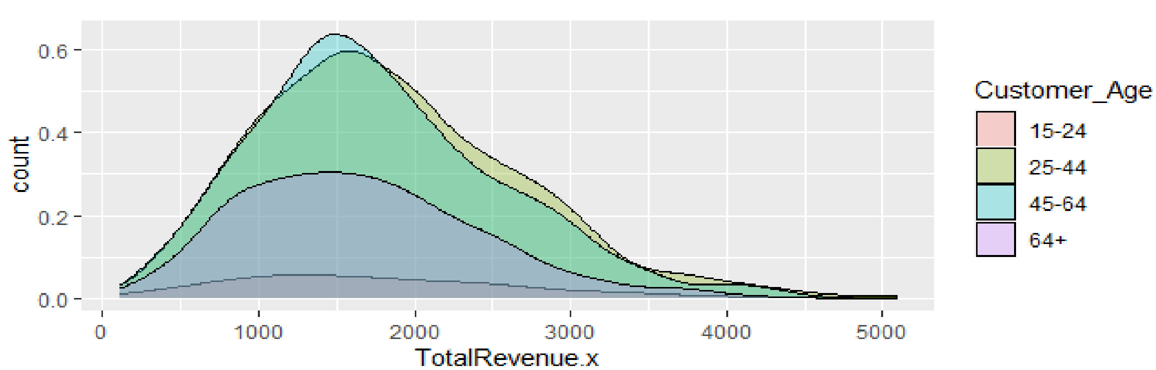


As seen in figure, the mean of total revenue, that is total spent money, is around $ 1500 and the number of female customers are more than male customers around mean.

4) Total Revenue by Customer Age Group

```
a<-c(Customer[Customer$Cust_Age=='15-24',]$TotalRevenue.x)
b<-c(Customer[Customer$Cust_Age=='25-44',]$TotalRevenue.x)
c<-c(Customer[Customer$Cust_Age=='45-64',]$TotalRevenue.x)
d<-c(Customer[Customer$Cust_Age=='64+',]$TotalRevenue.x)
dat_1524 <- data.frame(Customer_Age = factor(rep(c('15-24'))),TotalRevenue.x = c(a))
dat_2544 <- data.frame(Customer_Age = factor(rep(c('25-44'))),TotalRevenue.x = c(b))
dat_4564 <- data.frame(Customer_Age = factor(rep(c('45-64'))),TotalRevenue.x = c(c))
dat_64plus <- data.frame(Customer_Age = factor(rep(c('64+'))),TotalRevenue.x = c(d))
dat<-rbind(dat_1524,dat_2544,dat_4564,dat_64plus)
ggplot(dat, aes(x=TotalRevenue.x, fill=Customer_Age)) + geom_density(aes(y = ..count..), stat="density", alpha=0.3)
```

Figure 16 Total Revenue Frequency Graph by Age Groups



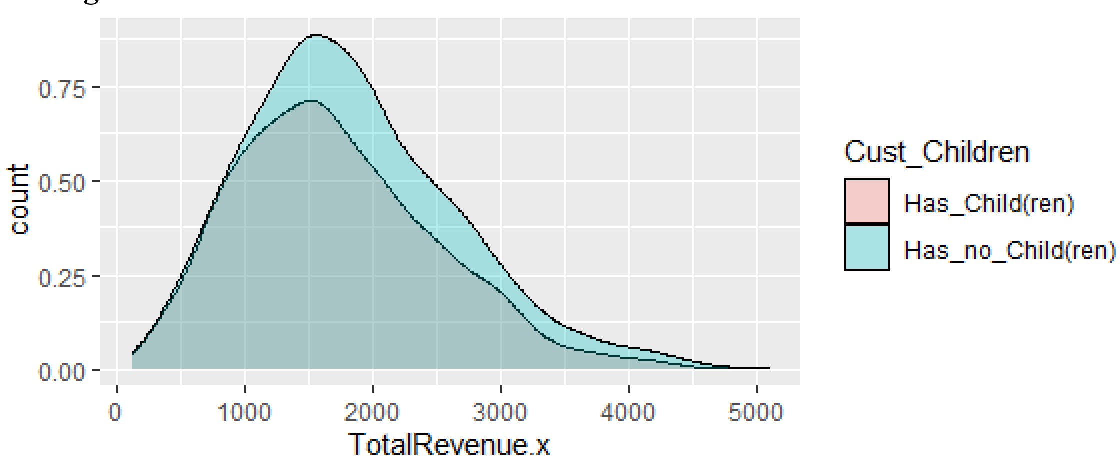
As seen in figure, among about $1500 total yearly spending customers, 45-64 age group appears more comparing to other. The four gorup can be drawn seperately to see inside as below by using patchwork library. As seen below, 15-24 age group less spenders in a year.

Figure 17 Total Revenue Frequency Graphs by Age Groups Separetly

5) Total Revenue by Customer Having Children or No Children

```
a<-c(Customer[Customer$Cust_Children=='Has_Child(ren)',]$TotalRevenue.x)
b<-c(Customer[Customer$Cust_Children=='Has_no_Child(ren)',]$TotalRevenue.x)
dat_chldr <- data.frame(Cust_Children = factor(rep(c('Has_Child(ren)'))), TotalRevenue.x = c(a))
dat_nochldr <- data.frame(Cust_Children = factor(rep(c('Has_no_Child(ren)'))),TotalRevenue.x = c(b))
dat<-rbind(dat_chldr,dat_nochldr)
ggplot(dat, aes(x=TotalRevenue.x, fill=Cust_Children)) + geom_density(aes(y = ..count..), stat="density", alpha=0.3)
```

Figure 18 Total Revenue Distribution of Customers w/ & w/o Children



6) Revenue by Store

The following sql query will extract required datas grouped by Stores.

```
con<-"SELECT s.Store_ID,s.Customer_ID, s.Sale_ID, SUM(t.Amount_Purchased)
CountItems,
SUM(t.Amount_Purchased*i.Price_Per_Item*(1 -d.Item_Discount)) Revenue
FROM Sales s
RIGHT JOIN Transactions t
ON t.Sale_ID= s.Sale_ID
LEFT JOIN Discounts d
ON d.Sale_Week= s.Sale_Week AND d.Item_ID=t.Item_ID
LEFT JOIN Items i
ON i.Item_ID= t.Item_ID
GROUP BY s.Store_ID;"

result<-dbSendQuery(mydb,con)
Store_Revenue<-fetch(result,n=-1)
dbClearResult(result)
```

Let's see the head of the Store_Revenue table :

```
head(Store_Revenue,5)
```

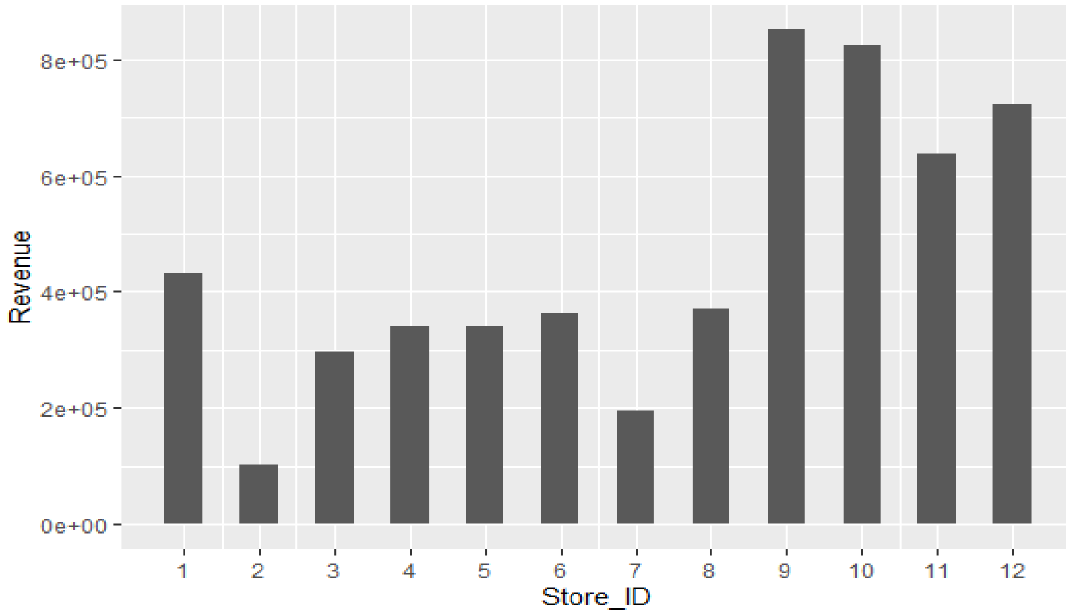
Output:

	Store_ID	Customer_ID	Sale_ID	CountItems	Revenue
1	5	4599139	2	89201	340409.1
2	3	3242746	4	77346	296243.6
3	11	6143606	12	166548	638185.3
4	9	3111830	13	221807	849936.0
5	12	2439039	25	187926	720931.5

The histogram can be drawn with following codes:

```
ggplot(Store_Revenue, aes(x=Store_ID, y=Revenue)) + geom_bar(stat = "identity", width=0.5)+
scale_x_continuous(labels=as.character(Store_Revenue$Store_ID), breaks
=Store_Revenue$Store_ID)
```

Figure 19 Histogram of Revenue through Stores



As ssen in figure above, Store 9,10,11 and 12 contributes to most revenues.

7) Top items in Stores 9,10,11 and 12

Following Sql query will extract the items datas from stores 9,10,11 and12. The table will be ranked by Total Revenue in item level descending; therefore, best items will be at top of the table in terms of their item level total revenue. By this query, the stores 9,10,11 and 12 will be considered as a one unit.

```
con<-"SELECT i.Item_ID, i.item_name, SUM(t.Amount_Purchased) TotalItems,
SUM(t.Amount_Purchased*i.Price_Per_Item*(1-d.Item_Discount)) TotalRevitems,
ROUND((SUM(t.Amount_Purchased*i.Price_Per_Item*
(1-d.Item_Discount))/SUM(t.Amount_Purchased)),2) AvgPrice FROM Sales s
RIGHT JOIN Transactions t ON t.Sale_ID= s.Sale_ID LEFT JOIN Discounts d
ON d.Sale_Week= s.Sale_Week AND d.Item_ID= t.Item_ID LEFT JOIN Items i
ON i.Item_ID= t.Item_ID where s.store_id=9 or s.store_id=10 or s.store_id=11 or s.store_id=12
GROUP BY i.item_id ORDER BY TotalRevitems desc;"

result<-dbSendQuery(mydb,con)
Items<-fetch(result,n=-1)
dbClearResult(result)
```

As a result, Items table is extracted to local.

The head of Items table is as following:

head(Items,10)

Output:

	Item_ID	item_name	TotalItems	TotalRevitems	AvgPrice
1	933312	Chickpeas	7880	50480.31	6.41
2	401557	Steak	7856	47267.58	6.02
3	156527	Pomegranate	9373	41589.61	4.44
4	626601	Bacon	7247	41374.48	5.71
5	624570	Oatmeal	8985	38793.22	4.32
6	969692	Coffee Cake	7692	37586.96	4.89
7	798039	Pie	4766	36968.16	7.76
8	876806	Chia Seeds	5003	33740.07	6.74
9	648507	Waffle Mix	6865	32352.74	4.71
10	315749	Radish	9587	31301.25	3.26

As seen the results above, Chickpeas is the most contributed item to Total Revenue.

8) Favorite stores for each customer

There are 3086 customers in the database, having key of Customer_ID. By following Sql query , the number of visits can be retreived for each customer and store pair:

```
SELECT Customer_ID, Store_ID,
COUNT(*) AS StoreCount
FROM SalesData
GROUP BY Customer_ID, Store_ID
ORDER BY StoreCount DESC;
```

The result of this query has 15311 rows. It shows for all store visits by individual customer. What needed is the one store per customer that visisted most by that customer. To do that, following query can be written in MySql:

```
SELECT Customer_ID, any_value(Store_ID), max(StoreCount) FROM (
SELECT Customer_ID, Store_ID,
COUNT(*) AS StoreCount
FROM SalesData
GROUP BY Customer_ID, Store_ID
ORDER BY StoreCount DESC) StoreCounts
GROUP BY Customer_ID;
```

Figure 20 Different Drivers for Python MySql

MySQL DB Drivers Comparison

Project	PyPi hosted	Eventlet friendly	Python 3 compatibility	Maturity and/or stability	Comment
MySQL-Python	Yes	Partial	No	Yes	Can be monkeypatched by eventlet, but only to enable thread pooling
mysqlclient	Yes	Partial	Yes	Yes	Initial testing shows that this is a promising DBAPI if eventlet requirement can be dropped
OurSQL	Yes	No	Yes, but not Pypi hosted	No	Development halted fairly early on, and has not seen commits/releases in two years
MySQL-Connector-Python	No	Yes	Yes	Yes, though the driver is still fairly new	The official Oracle-supported driver for MySQL
PyMySQL	Yes	Yes	Yes	Yes, however see notes below.	Actively maintained and popular.

(Source: https://wiki.openstack.org/wiki/PyMySQL_evaluation)

Previously, 2 dataframes created in MySql database through RStudio, they are customerdata and custdatasurvey. Their common key attribute is customer_id. Therefore, the tables can be joined through customer_id's.

First, the connection to the MySql must be managed. PyMySQL library can be used for it.

```
import pymysql
mydb =pymysql.connect(user='root', password='serdar27', db='transaction_database', host='127.0.0.1')
```

For dataframe manipulation, Panda library can be used.

```
import pandas as pd
```

Now, the joined table can retreived as pandas dataframe object as folowing:

```
cust=pd.read_sql_query("select * from customerdata cd left join custdatasurvey cs on\
cd.customer_id=cs.customer_id;",mydb)
```

```
cust.head()
```

	Customer_ID	AvgCountItems	AvgItemPrice	AvgSalePrice	Trips	TotalRevenue	TotalRevenue	Customer_ID	Cust_Sex	Cust_Income	Cust_Race	Cust_Age	Cust_Children	Cust_Rel_Status
0	4599139	51.7647	3.872785	200.473588	17	3408.051	3408.051	4599139	Male	70800	White	25-44	Has_Child(ren)	Married
1	3242746	55.5556	4.010527	222.807056	18	4010.527	4010.527	3242746	Female	85700	White	45-64	Has_no_Child(ren)	Married
2	6143606	44.125	3.931045	173.457375	8	1387.659	1387.659	6143606	Female	49500	Black	45-64	Has_no_Child(ren)	Not_Married
3	3111830	40.125	3.760607	150.894375	8	1207.155	1207.155	3111830	Male	45500	White	45-64	Has_no_Child(ren)	Married
4	2439039	53.8889	3.718355	200.378	9	1803.402	1803.402	2439039	Male	95900	White	45-64	Has_no_Child(ren)	Married

Just extract Customer_ID, AvgSalePrice, AvgItemPrice, Trips, Cust_Income and save it as cust_n

```
cust_n=cust[['Customer_ID','AvgSalePrice','AvgItemPrice','Trips','Cust_Income']]
```

cust_n.head()

```
In [96]: cust_n.head()
Out[96]:
```

	Customer_ID	Customer_ID	AvgSalePrice	AvgItemPrice	Trips	Cust_Income
0	4599139	4599139	200.473588	3.872785	17	70800
1	3242746	3242746	222.807056	4.010527	18	85700
2	6143606	6143606	173.457375	3.931045	8	49500
3	3111830	3111830	150.894375	3.760607	8	45500
4	2439039	2439039	200.378000	3.718355	9	95900

There are duplicate columns coming from MySql

cust_n.columns.duplicated()

Output:

array([False, True, False, False, False, False])

Running following code will supress one of them :

cust_n=ust_n.loc[:,~cust_n.columns.duplicated()]

```
In [76]: cust_n.loc[:,~cust_n.columns.duplicated()]
Out[76]:
```

	Customer_ID	AvgSalePrice	AvgItemPrice	Trips	Cust_Income
0	4599139	200.473588	3.872785	17	70800
1	3242746	222.807056	4.010527	18	85700
2	6143606	173.457375	3.931045	8	49500
3	3111830	150.894375	3.760607	8	45500

After this point, for KMeans Clustering can be applied for these 4 attributes (Customer ID is Metadata). First thing to is normalizations of the values.

Import following libraries :

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
%matplotlib inline
```

Let's take the value from the dataframe first, but Customer_ID is excluded, because it is only index column:

x = cust_n.values[:,1:]

```
In [102]: x = cust_n.values[:,1:]
          x

Out[102]: array([[2.00473588e+02, 3.87278523e+00, 1.70000000e+01, 7.08000000e+04],
                 [2.22807056e+02, 4.01052700e+00, 1.80000000e+01, 8.57000000e+04],
                 [1.73457375e+02, 3.93104533e+00, 8.00000000e+00, 4.95000000e+04],
                 ...,
                 [1.17340000e+02, 3.66687500e+00, 1.00000000e+00, 7.78000000e+04],
                 [1.51454000e+02, 3.60604762e+00, 1.00000000e+00, 4.14000000e+04],
                 [1.88248000e+02, 3.76496000e+00, 1.00000000e+00, 6.80000000e+04]])
```

As it is seen, all for column values are taken and saved as x which is an array.

Numpy has a function which converts NaN values to zero, if they exist. In this case there is no, but it can be run as following:

x = np.nan_to_num(x)

Now an new Dataset can be created by normalizations with Scikit StandardScaler() function as following;

Cluster_dataSet = StandardScaler().fit_transform(x)
Cluster_dataSet

All values are normalized.

```
In [111]: Cluster_dataSet = StandardScaler().fit_transform(x)
          Cluster_dataSet

Out[111]: array([[ 0.91801855,  0.37625206,  1.61417569,  0.1799738 ],
                 [ 1.76485377,  1.43172236,  1.84673154,  0.82098941],
                 [-0.10637595,  0.8226802 , -0.47882691, -0.73637737],
                 ...,
                 [-2.23422154, -1.20157089, -2.10671782,  0.48112208],
                 [-0.94069481, -1.66767128, -2.10671782, -1.08484894],
                 [ 0.45445153, -0.44997751, -2.10671782,  0.05951449]])
```

Now, the settings can be done as following for K-Means Clustering Analysis:

ClusterNum=4
k_means = KMeans(init = "k-means++", n_clusters = ClusterNum, n_init = 12)
k_means.fit(x)
labels = k_means.labels_

labels_ function creates label (cluster numbers), In this case 0,1,2 and 3 because n_clusters=4.

```
In [113]: labels
Out[113]: array([3, 1, 0, ..., 3, 0, 3])

In [114]: labels.shape
Out[114]: (3086,)

In [120]: labels[50:80]
Out[120]: array([0, 3, 3, 0, 1, 3, 3, 1, 0, 0, 1, 1, 3, 1, 1, 2, 3, 2, 3, 3, 3, 3,
                 0, 3, 3, 2, 1, 2, 3, 0])
```

Now, corresponding labels can be assigned to each rows as following. We can define new column as Clust and equalize it to label array.

cust_n['Clust']=labels

```
In [125]: cust_n.head(5)
Out[125]:
```

	Customer_ID	AvgSalePrice	AvgItemPrice	Trips	Cust_Income	Clust
0	4599139	200.473588	3.872785	17	70800	3
1	3242746	222.807056	4.010527	18	85700	1
2	6143606	173.457375	3.931045	8	49500	0
3	3111830	150.894375	3.760607	8	45500	0
4	2439039	200.378000	3.718355	9	95900	1

It is easy to find centroid values of each cluster by averaging all values grouped by Clust Coloumn:

cust_n.groupby("Clust").mean()

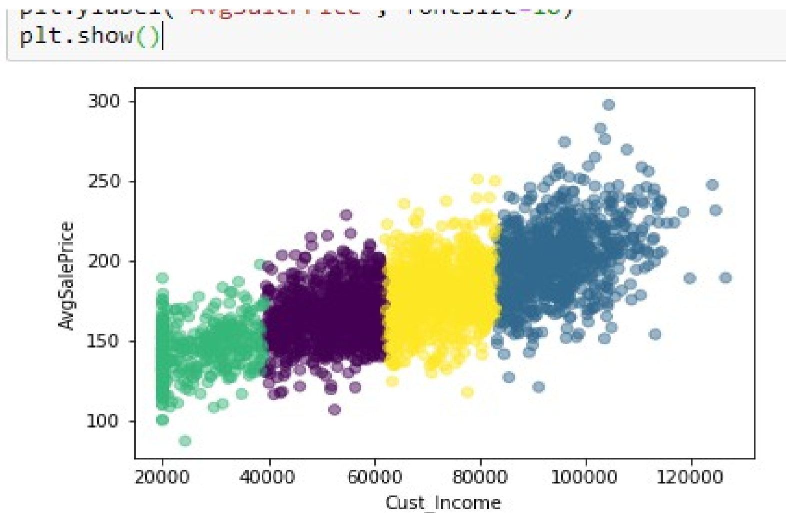
```
In [126]: cust_n.groupby("Clust").mean()
Out[126]:
```

Clust	Customer_ID	AvgSalePrice	AvgItemPrice	Trips	Cust_Income
0	3.244809e+06	165.343322	3.816076	10.335548	52133.665559
1	3.253068e+06	200.158484	3.858830	10.027972	94623.776224
2	3.240468e+06	143.985065	3.756059	9.505025	26557.286432
3	3.213011e+06	178.640762	3.827597	10.056095	72001.294498

Now, With these label values, clustering can visilualized as following:

```
plt.scatter(x[:,3], x[:, 0], c=labels.astype(np.float), alpha=0.5)
plt.xlabel('Cust_Income',fontsize=10)
plt.ylabel('AvgSalePrice', fontsize=10)
plt.show()
```

Figure 21 Clusters for Customer Income versus Average Sale Price



It seems that there is a relation between Average Sale Price and Customer Income. Let's run a corr() function to see different correlations:

```
cust_n[['AvgSalePrice','AvgItemPrice','Trips','Cust_Income']].corr()
```

```
In [142]: cust_n[['AvgSalePrice','AvgItemPrice','Trips','Cust_Income']].corr()
Out[142]:
```

	AvgSalePrice	AvgItemPrice	Trips	Cust_Income
AvgSalePrice	1.000000	0.374132	0.010996	0.716191
AvgItemPrice	0.374132	1.000000	0.032695	0.233874
Trips	0.010996	0.032695	1.000000	0.013445
Cust_Income	0.716191	0.233874	0.013445	1.000000

After correlation function, it is seen that there is a good correlation between Customer Income and Averaga Sale Price, which is already realizable in the scatter plot.

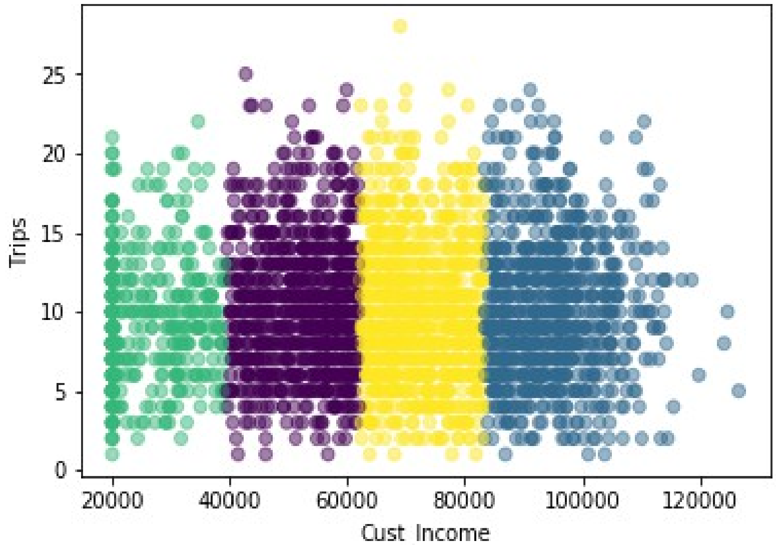
Linear Regression between Customer Income and Average Sale Price can be found by following codes:

```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(cust_n[['Cust_Income']],cust_n[['AvgSalePrice']])
```

lm.intercept_ gives array([122.13119346])
lm.coef_ gives array([[0.00081258]])

As a Result, function will be as: Average Sale Price = 0.0008 x Customer Income + 122

Figure 22 Clusters for Customer Income versus Trips



As seen in Figure Above, clusters are not telling important information. There is no correlation between Customer Income and Trips as well.

As seen in **Figure 19 (** Histogram of Revenue through Stores) , there are 4 stores they contribute mostly to overall revenue. After this point, It will be focused on their datas for analyses.

Remember, The salesdata table was created in the beginning of this document. It will created again with the condition Store_ID= 9 or 10 or 11 or 12. Let's create the table in MySql

```
CREATE TABLE SalesData_new AS (SELECT s.Customer_ID, s.Sale_ID, s.Store_ID,
SUM(t.Amount_Purchased) CountItems,
SUM(t.Amount_Purchased*i.Price_Per_Item*(1 -d.Item_Discount)) SalePrice
FROM Sales s
RIGHT JOIN Transactions t
ON t.Sale_ID= s.Sale_ID
LEFT JOIN Discounts d
ON d.Sale_Week= s.Sale_Week AND d.Item_ID= t.Item_ID
LEFT JOIN Items i
ON i.Item_ID= t.Item_ID
WHERE s.store_id=9 or s.store_id=10 or s.store_id=11 or s.store_id=12
GROUP BY s.Sale_ID, s.Customer_ID, s.Store_ID);
```

In Jupyter for Python, extract the SalesData_new as pandas dataframe:

```
df1=pd.read_sql_query("SELECT * FROM salesdata_new order by store_id desc;",mydb)
```

This time, let's define new CustomerData for only stores 9 or 10 or 11 or 12. Following query will create it in MySql Workbench.

```
DROP TABLE IF EXISTS CustomerData_new;
CREATE TABLE CustomerData_new AS(
SELECT Customer_ID, ROUND(AVG(CountItems),3) AvgCountItems,
ROUND(SUM(SalePrice)/SUM(CountItems),3) AvgItemPrice,
ROUND(AVG(SalePrice),3) AvgSalePrice,
COUNT(*) Trips,
ROUND(SUM(SalePrice),3) TotalRevenue
FROM SalesData_new
GROUP BY salesdata_new.Customer_ID);
```

Now extract to Python as customer_new dataframe with following code

```
df2=pd.read_sql_query("SELECT * FROM  CustomerData_new order by customer_id desc;",mydb)
```

Now new CustDataSurvey_new table must be created in MySql from this new CustomerData_new table which has only store_id 9,10,11 and 12.

```
CREATE TABLE CustDataSurvey_new AS(SELECT cd.TotalRevenue, cd.Customer_ID,
cs.Cust_Sex, cs.Cust_Income,
cs.Cust_Race, cs.Cust_Age,cs.Cust_Children,cs.Cust_Rel_Status
FROM CustomerData_new cd
LEFT JOIN Customer_Survey cs
ON cd.Customer_ID = cs.Customer_ID);
```

We can extract it to Python level as df3:

```
df3=pd.read_sql_query("SELECT * FROM  CustDataSurvey_new order by customer_id asc;",mydb)
```

df2 and df3 can be merged by their customer_id condition. Their row number is same.

```
main_data=pd.merge(df2[['Customer_ID','AvgItemPrice','AvgSalePrice','Trips']],df3[['Customer_ID','Cust_Income']],on='Customer_ID')
```

```
In [18]: main_data.head()
```

Out[18]:

	Customer_ID	AvgItemPrice	AvgSalePrice	Trips	Cust_Income
0	7366	3.667	161.347	3	50600
1	11947	3.971	228.724	5	82900
2	12872	4.138	213.814	9	100400
3	14392	3.666	168.627	1	64900
4	16347	3.866	183.013	3	69400

As we did Kmeans clustering before, It is seen that no useful clustering had been received. Therefore, this time “PCA: Principal Component Analaysis “ will be performed in order to see any valuable information.

First thing to do is again Standardizing the values:

```
from sklearn.preprocessing import StandardScaler
# Taking Values of main_data, without Customer_ID coloumn
# Possibly optional code will give same array
# x= main_data.loc[:,['AvgItemPrice', 'AvgSalePrice', 'Trips', 'Cust_Income']].values
x = main_data.values[:,1:]
#Converting NaN values to zero, if exists
x = np.nan_to_num(x)
#Standardazation
x_stnd = StandardScaler().fit_transform(x)
```

Now It is time to apply PCA:

```
from sklearn.decomposition import PCA
# Define PCA object with 4 dimensions
pca = PCA(n_components=4)
# Apply it to x
principalComponents = pca.fit_transform(x_stnd)
```

We have principal components as array. Lets convert it to dataframe:

```
principalDf = pd.DataFrame(data = principalComponents, columns = ['PC1', 'PC2','PC3','PC4'])
```

Let's combine the main_data and principalDf together:

```
finalDf = pd.concat([main_data, principalDf], axis = 1)
```

```
finalDf.head()
```

	Customer_ID	AvgItemPrice	AvgSalePrice	Trips	Cust_Income	PC1	PC2	PC3	PC4
0	7366	3.667	161.347	3	50600	1.313081	-0.299186	-0.504701	-0.255909
1	11947	3.971	228.724	5	82900	-1.763617	-0.767583	-0.004011	-0.575391
2	12872	4.138	213.814	9	100400	-2.544941	-0.107629	0.802192	0.415541
3	14392	3.666	168.627	1	64900	0.911670	-0.763885	-0.980540	0.003985
4	16347	3.866	183.013	3	69400	-0.098267	-0.785080	-0.078483	0.016293

Let's find eigenvectors of the pca array.

pca.components_

```
pca.components_
```

```
array([[-0.41104331, -0.63993132, -0.24493044, -0.60128229],
       [-0.34414635, -0.16030352,  0.92466909,  0.02920873],
       [ 0.81505611, -0.1947085 ,  0.28430783, -0.46576947],
       [ 0.21975038, -0.72586451, -0.0645385 ,  0.64858713]])
```

Now, create a dataframe with relevant Indexes for PCA eignevectors:

```
Components_df=pd.DataFrame(data=pca.components_,\
   columns = ['AvgItemPrice','AvgSalePrice','Trips','Cust_Income'],index=['PC1', 'PC2','PC3','PC4'])
```

```
print(Components_df)
```

```
     AvgItemPrice  AvgSalePrice     Trips  Cust_Income
PC1     -0.411043     -0.639931 -0.244930    -0.601282
PC2     -0.344146     -0.160304  0.924669     0.029209
PC3      0.815056     -0.194709  0.284308    -0.465769
PC4      0.219750     -0.725865 -0.064538     0.648587
```

This table is important and shows how PCA components and attributes realation. This relation can summarize as following:

	AvgItemPrice	AvgSalePrice	Trips	Cust_Income
PC1	-0.411043	-0.639931	-0.244930	-0.601282
PC2	-0.344146	-0.160304	0.924669	0.029209
PC3	0.815056	-0.194709	0.284308	-0.465769
PC4	0.219750	-0.725865	-0.064538	0.648587

When PC1 decreases, AvgSalePrice and Cust_Income increases. So, lowest possible PC1 relates to Big Spender Customers, They have high income and They spend more.

PC2 is highly related to Trips. They have positive realation and when PC2 increases, Trips also increases. These customers have high trips, that means they are frequent shoppers.

PC3 is related to AvgItemPrice. And PC4 is negatively related with AvgSalePrice and postitively related to Customer Income.

As a result, we have four different principal components, and we can interpret them to different meanings. Now, If we can cluster PC1, PC2, PC3 and PC4 attributes from the dataframe somehow, then these clusters can be used to cluster the customers as well.

We will try to cluster dataframe using PCA components as following:

```
# we already have principalComponents array, lets standardize the values
pc_stnd = StandardScaler().fit_transform(principalComponents)
```

Now applying Kmeans :

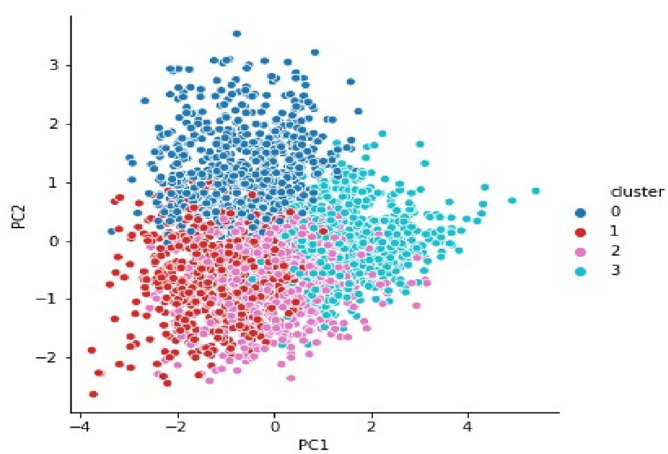
```
ClusterNum=4
k_means_pca = KMeans(init = "k-means++", n_clusters = ClusterNum, n_init = 12)
k_means_pca.fit(pc_stnd)
labels_pca = k_means_pca.labels_

# We can add the labels as coloumn to finalDf
finalDf['cluster']=labels_pca

finalDf.head(5)
```

```
finalDf.head(5)
```

	Customer_ID	AvgItemPrice	AvgSalePrice	Trips	Cust_Income	PC1	PC2	PC3	PC4	cluster
0	7366	3.667	161.347	3	50600	1.313081	-0.299186	-0.504701	-0.255909	0
1	11947	3.971	228.724	5	82900	-1.763617	-0.767583	-0.004011	-0.575391	2
2	12872	4.138	213.814	9	100400	-2.544941	-0.107629	0.802192	0.415541	1
3	14392	3.666	168.627	1	64900	0.911670	-0.763885	-0.980540	0.003985	0
4	16347	3.866	183.013	3	69400	-0.098267	-0.785080	-0.078483	0.016293	2

Now we can draw scatter plots for different attributes coloring by relevant clusters.

```
import seaborn as sns
sns.relplot(data=finalDf, x='PC1', y='PC2', hue='cluster', palette='tab10', kind='scatter')
plt.savefig('Scatter.png')
```



As seen in scatter plot, customers in cluster 0 have high PC 2 values (above 0) and they are balanced around PC1 = 0 , some minus PC1 trend. We defined previously, High PC2 means “Frequent Shoppers. On the otherhand, Cluster 1 which are red nodes, has low PC2 but and low PC1. So even they are not Frequent Shoppers, they spend much because they have low PC1.

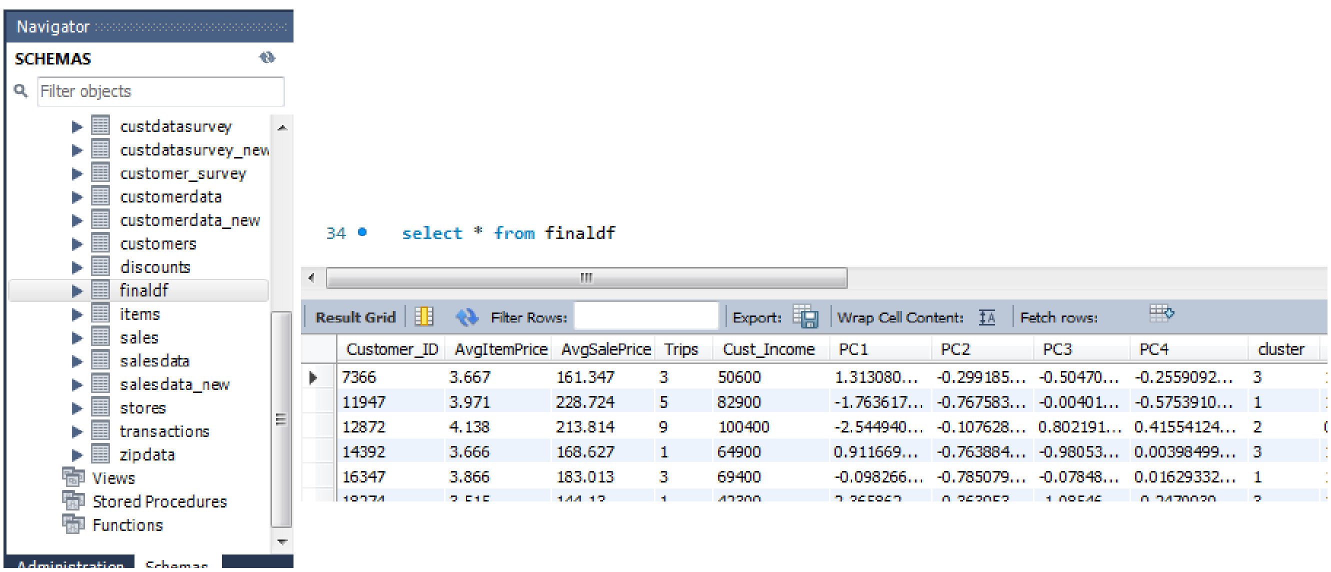
As conclusion, we can summarize as:

Cluster 0 Customers: Frequent Shoppers
Cluster 1 Customers: Big Spenders

Now we can group our customers in the dataframe as Frequent Shoppers and Big Spenders, and check their favorite items, stores etc. Remember, every time running Kmeans may result different clusters.

Let's transfer the finalDf pandas type dataframe to MySql with following codes:

```
# First import the library to create engine
from sqlalchemy import create_engine

# create the engine, adressing all infos of database in MySql
engine = create_engine("mysql+pymysql://{user}:{pw}@{host}/{db}"\
.format(host='127.0.0.1', db='transaction_database', user='root', pw='serdar27'))

# send the dataframe to MySql

finalDf.to_sql('finaldf', engine, if_exists='replace',index=False)
```



Now, we have the customers and their cluster in MySql environment. We can run any query for Frequent Shoppers and Big Spenders.

Let's create a temp table which as following:

```
CREATE TABLE temp AS (select s.customer_id, s.Sale_ID, s.sale_week,s.store_id, i.item_id,
                t.amount_purchased, d.item_discount, i.item_name,i.price_per_item
                 from sales s RIGHT JOIN Transactions t
                ON t.Sale_ID= s.Sale_ID
                 left join discounts d on d.sale_week=s.sale_week and d.item_id=t.item_id
                 left join items i on i.item_id=t.item_id
                 where s.store_id=9 or s.store_id=10 or s.store_id=11 or s.store_id=12);
```

Now, calculate saleprice with regard to customer_id and cluster group, and save it as summary table as:

```
create table summary as (select t.Customer_id, fd.cluster,t.item_id,t.item_name,t.store_id,
              SUM(t.Amount_Purchased) CountItems,
              SUM(t.Amount_Purchased*t.Price_Per_Item*(1-t.Item_Discount)) SalePrice
              FROM temp t right join finaldf fd on fd.customer_id=t.customer_id
              GROUP BY t.item_ID,fd.cluster);
```

Let's extract SalePrice in descending order and relevant item names for cluster =1 which is Big Spender customers:

```
select cluster, item_id, item_name, countitems, round(saleprice,2) Revenue from summary
                                    where cluster=1 order by Revenue desc;
```

Result Grid | Filter Rows: | Export:

cluster	item_id	item_name	countitems	Revenue
1	933312	Chickpeas	1661	10722.25
1	401557	Steak	1564	9467.52
1	626601	Bacon	1502	8651.15
1	156527	Pomegranate	1918	8561.39
1	624570	Oatmeal	1899	8249.24
1	798039	Pie	973	7553.55
1	876806	Chia Seeds	1055	7160.72
1	969692	Coffee Cake	1391	6845.86
1	648507	Waffle Mix	1451	6836.48
1	564329	Meatloaf Mix	2055	6743.12

As a result, Big Spenders have mostly spend money on the item "Chickpeas" .

If we order the table by countitems:

```
select cluster, item_id, item_name, countitems, round(saleprice,2) Revenue from summary
where cluster=1 order by countitems desc;
```

cluster	item_id	item_name	countitems	Revenue
1	564329	Meatloaf Mix	2055	6743.12
1	315749	Radish	2017	6649.28
1	106841	Grapes	2004	5357.15
1	164896	Nectarines	1964	6726.42
1	156527	Pomegranate	1918	8561.39
1	624570	Oatmeal	1899	8249.24
1	106498	Oranges	1681	6168.60
1	933312	Chickpeas	1661	10722.25
1	792879	Gum	1633	3185.66
1	760331	Pickles	1601	6095.96

As seen in result, the favorite product for Big Spenders is Meatloaf Mix.

Let's check the summary table for Cluster =0 , Frequent Shoppers.

```
select cluster, item_id, item_name, countitems, round(saleprice,2) Revenue from summary
where cluster=0 order by countitems desc;
```

cluster	item_id	item_name	countitems	Revenue
0	315749	Radish	4912	15998.05
0	156527	Pomegranate	4674	20704.35
0	624570	Oatmeal	4583	19822.35
0	564329	Meatloaf Mix	4452	14590.54
0	164896	Nectarines	4319	14740.22
0	969692	Coffee Cake	4265	20881.05
0	106841	Grapes	4228	11184.83
0	792879	Gum	4097	7991.50
0	401557	Steak	4017	24185.60
0	106498	Oranges	3993	14738.74

As seen in the result, frequent shoppers have mostly bought Radish and other items following it.

CONCLUSION:

Starting from transactional database for several stores and items, deep analytical studies are performed. Transactional datas are combined with customer surveys that include especially customer income values. By using clustering techniques of Unsupervised Machine Learning Algorithms, customers are segmented into different titles as Frequent Shoppers or Big Spenders. After that, the favorite products are extracted for relevant customer titles. These products are important in marketing and demand planning manner. Different promotions can be done on these products to increase overall revenue. Safety stock can be arranged accordingly Another approach can be done as, some high profit margin products can be set near favorite products on market shelves, so that these products can be attracted somehow for Big Spenders. Different mrketing strategies can be performed regardingly.

In the case study, R programming tool has been used for exploratory analysis. R programming is very flexible and handy to make valuable graphs. Later, Python programming is used for machine learning aim. KMeans algorithm is appropriate to find clusters but sometimes not gives any distinct clusters. In this case, PCA method is involved to help KMeans to find clear clusters. KMeans does not use categorical inputs which exist in especially in Customer Survey Table. Other Machine Learning Approaches such as KModes clustering can be used for utilization of categorical inputs.

Finally, study shows how to handle MySql DB, Python and R interactions. Further studies with supervised machine learning algorithms can be done by creating target features.