UNIT V NOSQL DATA MANAGEMENT FOR BIG DATA AND VISUALIZATION

NoSQL Databases : Schema-less Models: Increasing Flexibility for Data Manipulation-Key Value Stores- Document Stores - Tabular Stores - Object Data Stores - Graph Databases Hive - Sharding – Hbase – Analyzing big data with twitter - Big data for E-Commerce Big data for blogs - Review of Basic Data Analytic Methods using R.

NOSQL

The availability of a high-performance, elastic distributed data environment enables creative algorithms to exploit variant modes of data management in different ways. In fact, some algorithms will not be able to consume data in traditional RDBMS systems and will be acutely dependent on alternative means for data management. Many of these alternate data management frameworks are bundled under the term ―NoSQL databases.‖

The term ―NoSQL‖ may convey two different connotations—one implying that the data management system is not an SQL-compliant one, while the more accepted implication is that the term means ―Not only SQL,‖ suggesting environments that combine traditional SQL with alternative means of querying and access.

"Schema-Less Models": Increasing Flexibility for Data Manipulation

NoSQL data systems reduces the dependence on more formal database administration. NoSQL databases have more relaxed modeling constraints, which may benefit both the application developer and the end-user analysts. The interactive analyses are not throttled by the need to cast each query in terms of a relational table-based environment. Different NoSQL frameworks are optimized for different types of analyses.

Ex. Key-value stores, graph databases etc.

The general concepts for NoSQL include schema less modeling in which the semantics of the data are embedded within a flexible connectivity and storage model; this provides for automatic distribution of data and elasticity with respect to the use of computing, storage, and network bandwidth in ways that don't force specific binding of data to be persistently stored in particular physical locations. NoSQL databases also provide for integrated data caching that helps reduce data access latency and speed performance.

Because of the ―relaxed‖ approach to modeling and management that does not enforce shoehorning data into strictly defined relational structures, the models themselves do not necessarily impose any validity rules; this potentially introduces risks associated with ungoverned data management activities such as inadvertent inconsistent data replication, reinterpretation of semantics, and currency and timeliness issues.

APEC

KEY-VALUE STORES

Key-value store is a simple type of NoSQL data store. It is a schema-less model in which values (or sets of values) are associated with distinct character strings called keys.

Example data represented in a Key-value store

Key	Value
. . .	
"BMW"	{"1-Series", "3-Series", "5-Series", "5-Series GT", "7-Series", "X3", "X5", "X6", "Z4"}
"Buick"	{"Enclave", "LaCrosse", "Lucerne", "Regal"}
"Cadillac"	{"CTS", "DTS", "Escalade", "Escalade ESV", "Escalade EXT", "SRX", "STS"}
. . .	

The key is the name of the automobile make, while the value is a list of names of models associated with that automobile make.

The key-value store does not impose any constraints about data typing or data structure—the value associated with the key is the value, and it is up to the consuming business applications to assert expectations about the data values and their semantics and interpretation.

The core operations performed on a key-value store include:

- Get(key), which returns the value associated with the provided key.
- Put(key, value), which associates the value with the key.
- Multi-get(key1, key2,.., keyN), which returns the list of values associated with the list of keys.
- Delete(key), which removes the entry for the key from the data store.

One critical characteristic of a key-value store is uniqueness of the key—to find out any values the exact key must be used. In order to associate multiple values with a single key, consideration should be given to the representations of the objects and how they are associated with the key. Key-value stores are essentially very long, and presumably thin tables (in that there are not many columns associated with each row). The table's rows can be sorted by the key value to simplify finding the key during a query. Alternatively, the keys can be hashed using a hash function that maps the key to a particular location (sometimes called a ―bucket‖) in the table. Additional supporting data structures and algorithms (such as bit vectors and bloom filters) can be used to even determine whether the key exists in the data set at all.

The simplicity of the representation allows massive amounts of indexed data values to be appended to the same key-value table, which can then be sharded, or distributed across the storage nodes. Under the right conditions, the table is distributed in a way that is aligned with the way the keys are organized, so that the hashing function that is used to determine where any specific key exists in the table can also be used to determine which node holds that key's bucket.

APEC

While key-value pairs are very useful for both storing the results of analytical algorithms (such as phrase counts among massive numbers of documents) and for producing those results for reports, the model does pose some potential drawbacks. One is that the model will not inherently provide any kind of traditional database capabilities (such as atomicity of transactions, or consistency when multiple transactions are executed simultaneously)—those capabilities must be provided by the application itself. Another is that as the model grows, maintaining unique values as keys may become more difficult, requiring the introduction of some complexity in generating character strings that will remain unique among a myriad of keys.

DOCUMENT STORES

A document store is similar to a key-value store in that stored objects are associated (and therefore accessed via) character string keys. The difference is that the values being stored, which are referred to as ―documents,‖ provide some structure and encoding of the managed data. There are different common encodings, including XML (Extensible Markup Language), JSON (Java Script Object Notation), BSON (which is a binary encoding of JSON objects), or other means of serializing data.

Example of document store.

```
{StoreName:"Retail Store #34",
{Street:"1203 O ST", City:"Lincoln", State:"NE", ZIP:"68508"}
}

{StoreName:"Retail Store #65",
{MallLocation:"Westfield Wheaton", City:"Wheaton", State:"IL"}
}

{StoreName:"Retail Store $102",
{Latitude:" 40.748328", Longitude:" -73.985560"}
}
```

Even though the three examples all represent locations, the representative models differ. The document representation embeds the model so that the meanings of the document values can be inferred by the application. One of the differences between a key-value store and a document store is that while the former requires the use of a key to retrieve data, the latter often provides a means (either through a programming API or using a query language) for querying the data based on the contents. Because the approaches used for encoding the documents embed the object metadata, one can use methods for querying by example.

TABULAR STORES

Tabular, or table-based stores are largely descended from Google's original Bigtable design1 to manage structured data. Ex. Hadoop-related NoSQL data management system- HBase

The bigtable NoSQL model allows sparse data to be stored in a three-dimensional table that is indexed by a row key (that is used in a fashion that is similar to the key-value and document stores), a column key that indicates the specific attribute for which a data value is stored, and a timestamp that may refer to the time at which the row's column value was stored.

As an example, various attributes of a web page can be associated with the web page's URL: the HTML content of the page, URLs of other web pages that link to this web page, and the author of the content. Columns in a Bigtable model are grouped together as ‒families,‖ and the timestamps enable management of multiple versions of an object. The timestamp can be used to maintain history—each time the content changes, new column affiliations can be created with the timestamp of when the content was downloaded.

OBJECT DATA STORES

Object data stores and object databases seem to bridge the worlds of schema-less data management and the traditional relational models. On the one hand, approaches to object databases can be similar to document stores except that the document stores explicitly serializes the object so the data values are stored as strings, while object databases maintain the object structures as they are bound to object-oriented programming languages such as C11, Objective-C, Java, and Smalltalk. On the other hand, object database management systems are more likely to provide traditional ACID (atomicity, consistency, isolation, and durability) compliance—characteristics that are bound to database reliability. Object databases are not relational databases and are not queried using SQL.

GRAPH DATABASES

Graph databases provide a model of representing individual entities and numerous kinds of relationships that connect those entities. More precisely, it employs the graph abstraction for representing connectivity, consisting of a collection of vertices (which are also referred to as nodes or points) that represent the modeled entities, connected by edges (which are also referred to as links, connections, or relationships) that capture the way that two entities are related. Graph analytics performed on graph data stores are somewhat different than more frequently used querying and reporting.

CONSIDERATIONS

AEC

The decision to use a NoSQL data store instead of a relational model must be aligned with the data consumers' expectations for compliance with their expectations of relational models. As should be apparent, many NoSQL data management environments are engineered for two key criteria:

1. fast accessibility, whether that means inserting data into the model or pulling it out via some query or access method;

2. scalability for volume, so as to support the accumulation and management of massive amounts of data.

The different approaches are amenable to extensibility, scalability, and distribution, and these characteristics blend nicely with programming models (like MapReduce) with straightforward creation and execution of many parallel processing threads. Distributing a tabular data store or a key-value store allows many queries/accesses to be performed simultaneously, especially when the hashing of the keys maps to different data storage nodes. Employing different data allocation strategies will allow the tables to grow indefinitely without requiring significant rebalancing. In other words, these data organizations are designed for high performance computing for reporting and analysis. However, most NoSQL environments are not generally designed for transaction processing, and it would require some whittling down of the list of vendors to find those that support ACID transactions.

APEC

HIVE

One of the often-noted issues with MapReduce is that although it provides a methodology for developing and executing applications that use massive amounts of data, it is not more than that. The data can be managed within files using HDFS, many business applications expect representations of data in structured database tables. That was the motivation for the development of Hive.

Hive is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. Hive is specifically engineered for data warehouse querying and reporting and is not intended for use as within transaction processing systems that require real-time query execution or transaction semantics for consistency at the row level.

Hive is layered on top of the file system and execution framework for Hadoop and enables applications and users to organize data in a structured data warehouse and therefore query the data using a query language called HiveQL that is similar to SQL. The Hive system provides tools for extracting/ transforming/loading data (ETL) into a variety of different data formats. And because the data warehouse system is built on top of Hadoop, it enables native access to the MapReduce model, allowing programmers to develop custom Map and Reduce functions that can be directly integrated into HiveQL queries. Hive provides scalability and extensibility for

batch-style queries for reporting over large datasets that are typically being expanded while relying on the fault tolerant aspects of the underlying Hadoop execution model.

HBASE

HBase is an example of a non-relational data management environment that distributes massive datasets over the underlying Hadoop framework. HBase is derived from Google's BigTable and is a column-oriented data layout that, when layered on top of Hadoop, provides a fault-tolerant method for storing and manipulating large data tables. Data stored in a columnar layout is amenable to compression, which increases the amount of data that can be represented while decreasing the actual storage footprint. HBase also supports in-memory execution.

HBase is not a relational database, and it does not support SQL queries. There are some basic operations for HBase:

- Get (which access a specific row in the table),
- Put (which stores or updates a row in the table),
- Scan (which iterates over a collection of rows in the table), and
- Delete (which removes a row from the table).

Because it can be used to organize datasets, coupled with the performance provided by the aspects of the columnar orientation, HBase is a reasonable alternative as a persistent storage paradigm when running MapReduce applications.

Review of Basic Data Analytic Methods Using R

The six phases of the Data Analytics Lifecycle

- Phase 1: Discovery
- Phase 2: Data Preparation
- Phase 3: Model Planning
- Phase 4: Model Building
- Phase 5: Communicate Results
- Phase 6: Operationalize

Introduction to R

- R is a programming language and software framework for statistical analysis and graphics
- It is available for use under the GNU General Public License
- R software and installation instructions can be obtained via the Comprehensive R Archive and Network

APEC

The flow of a basic R script to address an analytical problem

- Import dataset
- Examine the contents of the dataset
- Execute the model building task

Ex.

The annual sales in U.S. dollars for 10,000 retail customers

```
# import a CSV file of the total annual sales for each customer
sales <- read.csv("c:/data/yearly_sales.csv")

# examine the imported dataset
head(sales)

summary(sales)

# plot num_of_orders vs. sales
plot(sales$num_of_orders,sales$sales_total,
        main="Number of Orders vs. Sales")

# perform a statistical analysis (fit a linear regression model)
results <- lm(sales$sales_total ~ sales$num_of_orders)
summary(results)

# perform some diagnostics on the fitted model
# plot histogram of the residuals
hist(results$residuals, breaks = 800)
```

The data file is imported using the read.csv( ) function. Once the file has been imported, it is useful to examine the contents to ensure that the data was loaded properly as well as to become familiar with the data. In the example, the head( ) function, by default, displays the first six records of sales.

```
# examine the imported dataset
head(sales)
  cust_id sales_total num_of_orders gender
1  100001      800.64             3      F
2  100002      217.53             3      F
3  100003       74.58             2      M
4  100004      498.60             3      M
5  100005      723.11             4      F
6  100006       69.43             2      F
```

The summary ( ) function provides some descriptive statistics, such as the mean and median, for each data column. Additionally, the minimum and maximum values as well as the 1st and 3rd quartiles are provided. Because the gender column contains two possible characters, an "F" (female) or "M" (male), the summary () function provides the count of each character's occurrence.

```
summary(sales)
```

```
    cust_id         sales_total       num_of_orders     gender
 Min.   :100001   Min.   :   30.02   Min.   : 1.000   F:5035
 1st Qu.:102501   1st Qu.:   80.29   1st Qu.: 2.000   M:4965
 Median :105001   Median :  151.65   Median : 2.000
 Mean   :105001   Mean   :  249.46   Mean   : 2.428
 3rd Qu.:107500   3rd Qu.:  295.50   3rd Qu.: 3.000
 Max.   :110000   Max.   :7606.09    Max.   :22.000
```

The plot( ) function generates a scatterplot of the number of orders (sales$num_of_orders) against the annual sales (sales$sales_total). The $ is used to reference a specific column in the dataset sales. The resulting plot is shown below

```
# plot num_of_orders vs. sales
plot(sales$num_of_orders,sales$sales_total,
        main="Number of Orders vs. Sales")
```



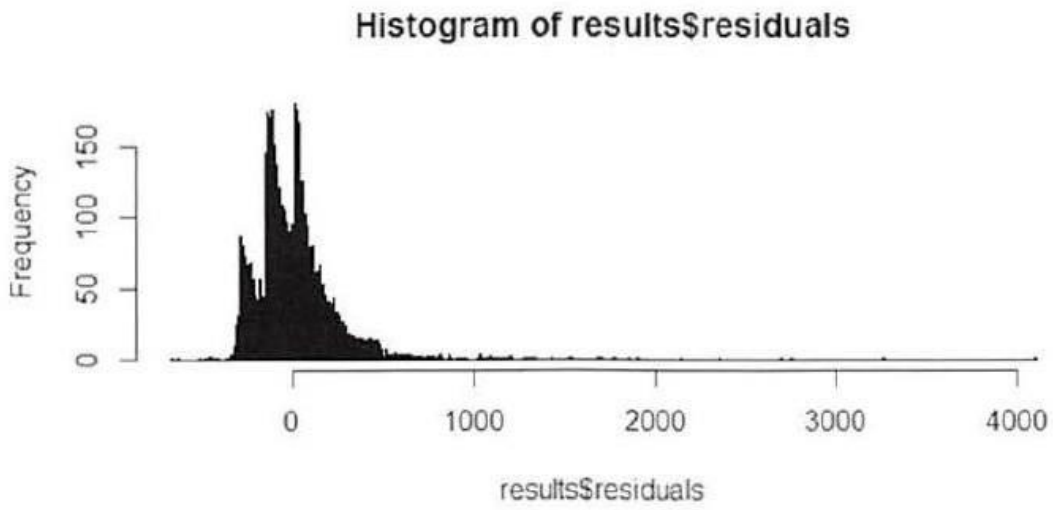
Each point corresponds to the number of orders and the total sales for each customer. The plot indicates that the annual sales are proportional to the number of orders placed. Although the observed relationship between these two variables is not purely linear, the analyst decided to apply linear regression using the lm( ) function as a first step in the modeling process.

```
results <- lm(sales$sales_total ~ sales$num_of_orders)
results
```

Results stores more information that can be examined with the summary( ) function. The summary () function is an example of a generic function. A generic function is a group of

functions sharing the same name but behaving differently depending on the number and the type of arguments they receive. The generic function hist( ) is used to generate a histogram of the residuals stored in results.

```
hist(results$residuals, breaks = 800)
```



R Graphical User Interfaces

R provides two types of basic user interfaces

1. Command line interface (CLI)
2. Graphical user interface (GUI)

To improve the ease of writing, executing, and debugging R code, several additional GUI's have been written for R. They are

- R commander
- Rattle
- RStudio

Window panes of RStudio GUI

- Script - Serves as an area to write and save R code
- Workspace - Lists the datasets and variables in the R environment
- Plot - Displays the plots generated by the R code and provides a straightforward mechanism to export the plots
- Console - Provides a history of the executed R code and the output

Obtaining help in RStudio

- The console pane can be used to obtain help information on R.
  By entering either? followed by the function name or entering help (function name) we can get the help information

Ex.

- ? lm
- help (lm)

Updating the contents of a variable

- Functions such as **edit( )** and **fix( )** allow the user to update the contents of an R variable.
- Such changes can also be implemented with RStudio by selecting the appropriate variable from the workspace pane.

Saving and loading the workspace environment

- R allows one to save the workspace environment, including variables and loaded libraries, into an . Rdata file using the **save.image( )** function. An existing . Rdata file can be loaded using the **load.image( )** function.

Data Import and Export

Importing of R dataset

The dataset was imported into R using the **read.csv( )** function

Ex. sales ← read.csv("c : /data/yearly_sales.csv")

R uses a forward slash (/) as the separator character in the directory and file paths. Windows users, who may be using a backslash (\ ) as a separator.

To simplify the import of multiple files with long path names, the **setwd( )** function can be used to set the working directory for the subsequent import and export operations, as shown in the following R code.

Ex.

setwd("c: / data/ ")

sales ← read.csv("yearly_sales.csv")

Other import functions include **read.table( )** and **read.delim( )** ,which are intended to import other common file types such as TXT.

Ex.

sales_table ← read.table ("yearly_sales.csv", header=TRUE, sep=",")

sales_delim ← read.delim ("yearly_sales.csv", sep=",")

The **read.delim( )** function expects the column separator to be a tab ("\ t"). In the event that the numerical data in a data file uses a comma for the decimal, R also provides two additional functions namely **read.csv2( )** and **read.delim2( )**

Import function details

Function	Headers	Separator	Decimal Point
read.table()	FALSE	""	"."
read.csv()	TRUE	","	"."
read.csv2()	TRUE	";"	","
read.delim()	TRUE	"\t"	"."
read.delim2()	TRUE	"\t"	","

APEC

Exporting of R datasets to an external file

The functions used are

- **write. table ( )**
- **write.csv ( )**
- **write.csv2 ( )**

Ex.

To add an additional column to the sales dataset and exports the modified dataset to an external file

```
# add a column for the average sales per order
sales$per_order <- sales$sales_total/sales$num_of_orders

# export data as tab delimited without the row names
write.table(sales,"sales_modified.txt", sep="\t", row.names=FALSE
```

Sometimes it is necessary to read data from a database management system. R packages such as **DBI** and **RODBC** are available for this purpose. These packages provide database interfaces for communication between R and DBMSs such as MySQL, Oracle, SQL Server, PostgreSQL, and Pivotal Greenplum. The following R code demonstrates how to install the RODBC package with the **install.packages( )** function. The **library( )** function loads the package into the R workspace. Finally, a connector **(conn)** is initialized for connecting to a Pivotal Greenplum database training2 via open database connectivity (ODBC) with user **user**.

install.packages ("RODBC")
library(RODBC)
conn ←odbcConnect ("training2", uid="user" , pwd= "password ")

The connector needs to be present to submit a SQL query to an ODBC database by using the **sqlQuery( )** function from the RODBC package.

Ex.

Retrieving specific columns from the housing table in which household income (hinc) is greater than $1,000,000.

housing_data ← sqlQuery(conn, "select serialno, state, persons, room from housing where hinc > 1000000")

head(housing_data) will display the first 6 results

Although plots can be saved using the RStudio GUI, plots can also be saved using R code by specifying the appropriate graphic devices. Using the **jpeg( )** function, the following R code creates a new JPEG file, adds a histogram plot to the file, and then closes the file. Such techniques are useful when automating standard reports. Other functions, such as **png( )** , **bmp( )** ,**pdf( )** ,and **postscript( )** ,are available in R to save plots in the desired format.

Ex.

```
jpeg(file="c:/data/sales_hist.jpeg") # create a new jpeg file
hist(sales$num_of_orders)            # export histogram to jpeg
dev.off()                            # shut off the graphic device
```

Attribute and Data Types

There are four categories of attributes namely

- nominal
- ordinal
- interval
- ratio

NOIR Attribute Types

	Categorical (Qualitative)		Numeric (Quantitative)	
	Nominal	**Ordinal**	**Interval**	**Ratio**
Definition	The values represent labels that distinguish one from another.	Attributes imply a sequence.	The difference between two values is meaningful.	Both the difference and the ratio of two values are meaningful.
Examples	ZIP codes, nationality, street names, gender, employee ID numbers, TRUE or FALSE	Quality of diamonds, academic grades, magnitude of earthquakes	Temperature in Celsius or Fahrenheit, calendar dates, latitudes	Age, temperature in Kelvin, counts, length, weight
Operations	$=, \neq$	$=, \neq,$ $<, \leq, >, \geq$	$=, \neq,$ $<, \leq, >, \geq,$ $+, -$	$=, \neq,$ $<, \leq, >, \geq,$ $+, -,$ $\times, \div$

There are three categories of data types namely

- numeric
- character
- logical

Ex.

```
i <- 1                      # create a numeric variable
sport <- "football"         # create a character variable
flag <- TRUE                # create a logical variable
```

To examine the characteristics of the variables **class( )** and **typeof( )** functions can be used. The class( ) function represents the abstract class of an object. The typeof( ) function determines the way an object is stored in memory.

Ex.

```
class(i)                    # returns "numeric"
typeof(i)                   # returns "double"

class(sport)                # returns "character"
typeof(sport)               # returns "character"
```

To test the variables and coerce a variable into a specific type there are functions available in R

Ex. Functions like **is.integer( )** and **as.integer( )**

```
is.integer(i)                    # returns FALSE
j <- as.integer(i)               # coerces contents of i into an integer
is.integer(j)                    # returns TRUE
```

Vectors

Vectors are a basic building block for data in R. Simple R variables are actually vectors. The tests for vectors can be conducted using the **is.vector( )** function.

Creation and manipulation of vectors

A vector can be created using the combine function, c**( )** or the colon operator **:**

Ex.

```
u <- c("red", "yellow", "blue") # create a vector "red" "yellow" "blue"
u                               # returns "red" "yellow" "blue"
u[1]                            # returns "red" (1st element in u)
v <- 1:5                        # create a vector 1 2 3 4 5
v                               # returns 1 2 3 4 5
sum(v)                          # returns 15
w <- v * 2                      # create a vector 2 4 6 8 10
w                               # returns 2 4 6 8 10
w[3]                            # returns 6 (the 3rd element of w)
z <- v + w                      # sums two vectors element by element
z                               # returns 3  6  9 12 15
z > 8                           # returns FALSE FALSE  TRUE  TRUE  TRUE
z[z > 8]                        # returns 9 12 15
z[z > 8 | z < 5]                # returns 3  9 12 15 ("|" denotes "or")
```

To initialize a vector of a specific length and then populate the content of the vector later. The **vector ( )** function is used
Ex.

```
a <- vector(length=3)           # create a logical vector of length 3
a                               # returns FALSE FALSE FALSE
b <- vector(mode="numeric", 3)  # create a numeric vector of length 3
typeof(b)                       # returns "double"
b[2] <- 3.1                     # assign 3.1 to the 2nd element
b                               # returns 0.0 3.1 0.0
c <- vector(mode="integer", 0)  # create an integer vector of length 0
c                               # returns integer(0)
length(c)                       # returns 0
```

Arrays and Matrices

The **array ( )** function can be used to restructure a vector as an array.

Ex.

To build a three-dimensional array to hold the quarterly sales for three regions over a two-year period and then assign the sales amount of $158,000 to the second region for the first quarter of the first year.

```
# the dimensions are 3 regions, 4 quarters, and 2 years
quarterly_sales <- array(0, dim=c(3,4,2))
quarterly_sales[2,1,1] <-  158000
quarterly_sales
```

```
, , 1

       [,1] [,2] [,3] [,4]
[1,]      0    0    0    0
[2,] 158000    0    0    0
[3,]      0    0    0    0

, , 2

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
```

Matrix

A two-dimensional array is known as a matrix. The parameters **nrow** and **ncol** define the number of rows and columns

Ex.

```
sales_matrix <- matrix(0, nrow = 3, ncol = 4)
sales_matrix
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
```

Operations supported

- addition
- subtraction
- multiplication

- transpose function
- inverse function

Data frames

Data frames provide a structure for storing and accessing several variables of possibly different data types. Because of their flexibility to handle many data types, data frames are the preferred input format for many of the modeling functions available in R.
The function **is.data.frame( )** function indicates, a data frame was created by the **read.csv( )** function

```
#import a CSV file of the total annual sales for each customer
sales <- read.csv("c:/data/yearly_sales.csv")
is.data.frame(sales)              # returns TRUE
```

To provide the structure of the data frame use the function **str( )**
Ex.

```
str(sales)              # display structure of the data frame object

'data.frame': 10000 obs. of 4 variables:
 $ cust_id      : int  100001 100002 100003 100004 100005 100006 ...
 $ sales_total  : num  800.6 217.5 74.6 498.6 723.1 ...
 $ num_of_orders: int  3 3 2 3 4 2 2 2 2 2 ...
 $ gender       : Factor w/ 2 levels "F","M": 1 1 2 2 1 1 2 2 1 2 ...
```

A subset of the data frame can be retrieved through subsetting operators.
Ex.

- # extract the fourth column of the sales data frame
  sales[,4]
- # extract the gender column of the sales data frame
  sales$gender
- # retrieve the first two rows of the data frame
  sales[l:2,]
- # retrieve all the records whose gender is female
  sales[sales$gender=="F",]

Lists

A list is a collection of objects that can be of various types.
Ex.
R code to create an assortment, a list of different object types
build an assorted list of a string, a numeric, a list, a vector, and a matrix

housing←**list(“own”, “rent”)**
assortment**←list(“football”, 7.5, housing,v,M)**
assortment

```
[ [ 1 ] ]
[ 1 ]  –football‖

[ [ 2 ] ]
[ 1 ] 7.5

[ [ 3 ] ]
[ [ 3 ] ] [ [ 1 ] ]
[ 1 ]  –own‖
[ [ 3 ] ] [ [ 2 ] ]
[ 1 ]  –rent‖

[ [ 4 ] ]
[ 1 ] 1  2  3  4  5

[ [5] ]

     [,1] [,2] [,3]
[1,]    1    5    3
[2,]    3    0    3
[3,]    3    4    3
```

The use of the single set of brackets only accesses an item in the list, not its content

```
# examine the fifth object, M, in the list
class(assortment[5])            # returns "list"
length(assortment[5])           # returns 1

class(assortment[[5]])          # returns "matrix"
length(assortment[[5]])         # returns 9 (for the 3x3 matrix)
```

Factor

A factor denotes a categorical variable, typically with a few finite levels such as "F" and "M" in the case of gender. Factors can be ordered or not ordered.

Ex.

```
class(sales$gender)          # returns "factor"
is.ordered(sales$gender)     # returns FALSE
head(sales$gender)           # display first six values and the levels
F F M M F F
Levels F M
```

The **cbind( )** function is used to combine variables column-wise. The **rbind( )** function is used to combine datasets row-wise.

Contingency Tables

In R, table refers to a class of objects used to store the observed counts across the factors for a given dataset. Such a table is commonly referred to as a contingency table and is the basis for performing a statistical test on the independence of the factors used to build the table.

Ex.

To build a contingency table based on the sales$gender and sales$ spender factors.

```
# build a contingency table based on the gender and spender factors
sales_table <- table(sales$gender,sales$spender)
sales_table
  small medium  big
F  1726   2746  563
M  1656   2723  586

class(sales_table)              # returns "table"
typeof(sales_table)             # returns "integer"
dim(sales_table)                # returns 2 3

# performs a chi-squared test
summary(sales_table)
Number of cases in table: 10000
Number of factors: 2
Test for independence of all factors:
  Chisq = 1.516, df = 2, p-value = 0.4686
```

Based on the observed counts in the table, the summary {) function performs a chi-squared test on the independence of the two factors. Because the reported p-value is greater than 0.05, the assumed independence of the two factors is not rejected.

Exploratory Data Analysis

Exploratory data analysis is a data analysis approach to reveal the important characteristics of a dataset, mainly through visualization.

Visualization gives a succinct, holistic view of the data that may be difficult to grasp from the numbers and summaries alone.

Ex.

Variables x and y of the data frame data can instead be visualized in a scatterplot

```
summary(data)
       x                   y
 Min.   :-1.90483   Min.   :-2.16545
 1st Qu.:-0.66321   1st Qu.:-0.71451
 Median : 0.09367   Median :-0.03797
 Mean   : 0.02522   Mean   :-0.02153
 3rd Qu.: 0.65414   3rd Qu.: 0.55738
 Max.   : 2.18471   Max.   : 1.70199
```

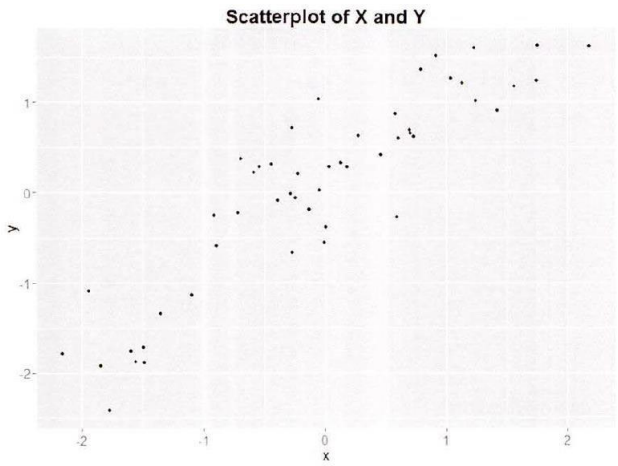
The code to generate data as well as scatterplot

```
x <- rnorm(50)
y <- x + rnorm(50, mean=0, sd=0.5)

data <- as.data.frame(cbind(x, y))

summary(data)

library(ggplot2)
ggplot(data, aes(x=x, y=y)) +
  geom_point(size=2) +
  ggtitle("Scatterplot of X and Y") +
  theme(axis.text=element_text(size=12),
        axis.title = element_text(size=14),
        plot.title = element_text(size=20, face="bold"))
```



Statistical Properties of Anscombe's Quartet

- Mean of x - 9
- Variance of x - 11
- Mean of y - 7.50
- Variance of y - 4.12 or 4.13
- Correlation between x and y - 0.816
- Linear regression line - y=3.00+0.50x

Visualizing a Single Variable

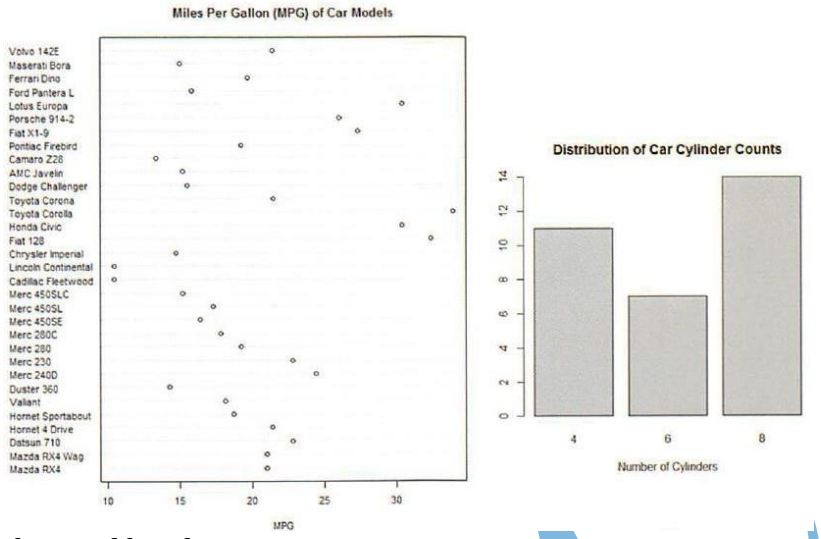
R has many functions available to examine a single variable. Some of these functions are listed below

Function	Purpose
`plot(data)`	Scatterplot where x is the index and y is the value; suitable for low-volume data
`barplot(data)`	Barplot with vertical or horizontal bars
`dotchart(data)`	Cleveland dot plot [12]
`hist(data)`	Histogram
`plot(density(data))`	Density plot (a continuous histogram)
`stem(data)`	Stem-and-leaf plot
`rug(data)`	Add a rug representation (1-d plot) of the data to an existing plot

Dotchart and Barplot

Dotchart and barplot portray continuous values with labels from a discrete variable. A dotchart can be created in R with the function **dotchart (x, label= ...)**, where x is a numeric vector and label is a vector of categorical labels for x. A barplot can be created with the **barplot (height)** function, where height represents a vector or matrix.

(a) Dotchart on the miles per gallon of cars and (b) Barplot on the distribution of car cylinder counts



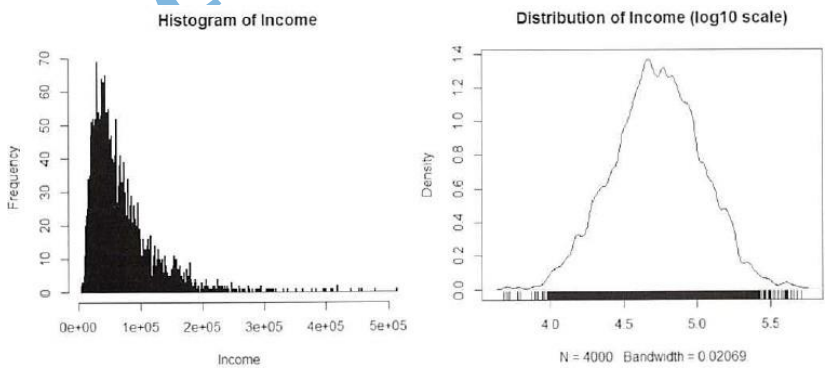
Code for the dotchart and barplot

data (mtcars)
dotchart (mtcars$mpg, labels=row .names (mtcars), cex=.7, main= "Miles Per Gallon (MPG) of Car Models", xlab= "MPG")
barplot (table (mtcars$cyl), main="Distribution of Car Cylinder Counts", xlab= "Number of Cylinders")

Histogram and Density Plot
To represent the household income

The histogram shows a clear concentration of low household incomes on the left and the long tail of the higher incomes on the right.

The density plot of represents the logarithm of household income values, which emphasizes the distribution

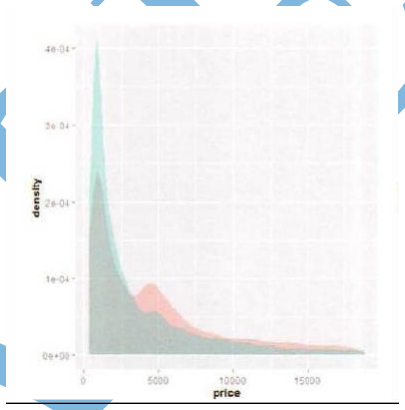


The code to generate the two plots

```
# randomly generate 4000 observations from the log normal distribution
income <- rlnorm(4000, meanlog = 4, sdlog = 0.7)
summary(income)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.301  33.720  54.970  70.320  88.800 659.800
income <- 1000*income
summary(income)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4301   33720   54970   70320   88800  659800
# plot the histogram
hist(income, breaks=500, xlab="Income", main="Histogram of Income")
# density plot
plot(density(log10(income), adjust=0.5),
     main="Distribution of Income (log10 scale)")
# add rug to the density plot
rug(log10(income))
```

The **rug( )** function creates a one-dimensional density plot on the bottom of the graph to emphasize the distribution of the observation.

Density plot



library ("ggplot2")
data (diamonds) # load the diamonds dataset from ggplot2
Only keep the premium and ideal cuts of diamonds

niceDiamonds ← diamonds [diamonds$cut=="Premium"| diamonds$cut==" Ideal ,]
plot density plot of diamond prices
ggplot (niceDiamonds, aes(x=price , fill =cut)) + geom_density(alpha = .3, color=NA)

Examining Multiple Variables

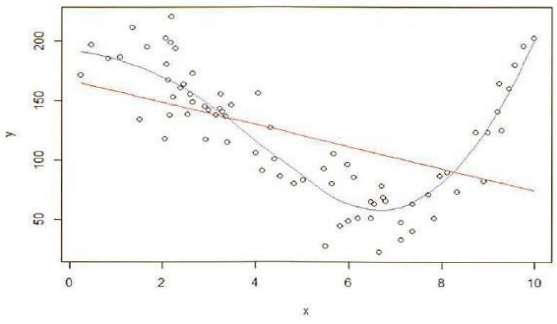
Scatterplot

A scatterplot is a simple and widely used visualization for finding the relationship among multiple variables. A scatterplot can represent data with up to five variables using x-axis, y-axis,

size, color, and shape. But usually only two to four variables are portrayed in a scatterplot to minimize confusion.

If the functional relationship between the variables is somewhat pronounced, the data may roughly lie along a straight line, a parabola, or an exponential curve. If variable y is related exponentially to x, then the plot of x versus log (y) is approximately linear. If the plot looks more like a cluster without a pattern, the corresponding variables may have a weak relationship.

Examining two variables with regression




```
# 75 numbers between 0 and 10 of uniform distribution
x <- runif(75, 0, 10)

x <- sort(x)
y <- 200 + x^3 - 10 * x^2 + x + rnorm(75, 0, 20)


lr <- lm(y ~ x)        # linear regression
poly <- loess(y ~ x)   # LOESS


fit <- predict(poly)  # fit a nonlinear line

plot(x,y)

# draw the fitted line for the linear regression
points(x, lr$coefficients[1] + lr$coefficients[2] * x,
       type = "l", col = 2)

# draw the fitted line with LOESS
points(x, fit, type = "l", col = 4)
```

Dotchart and Barplot

Dotchart and barplot can visualize multiple variables. Both of them use color as an additional dimension for visualizing the data.

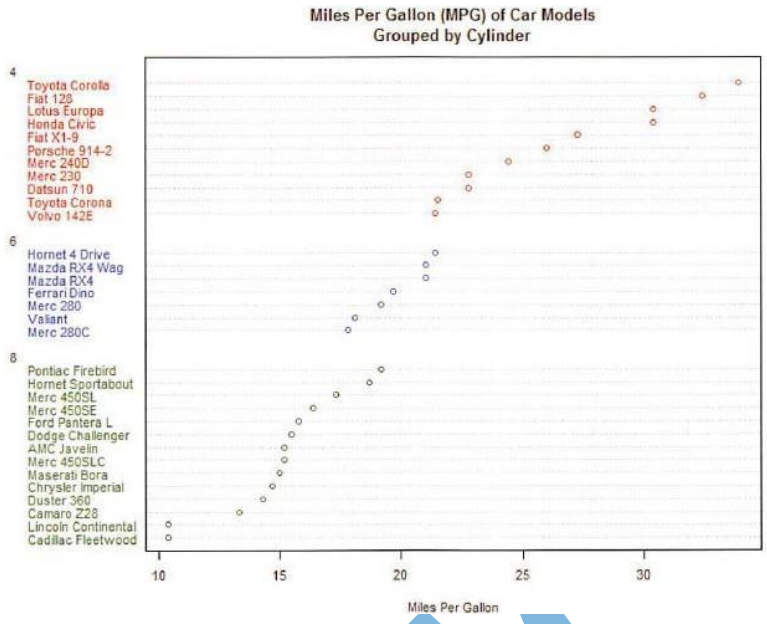
Dotchart to represent multiple variables

```
# sort by mpg
cars <- mtcars[order(mtcars$mpg),]

# grouping variable must be a factor
cars$cyl <- factor(cars$cyl)

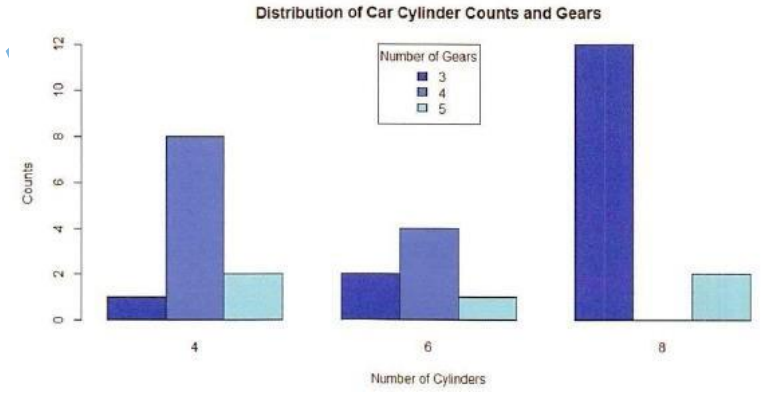
cars$color[cars$cyl==4] <- "red"
cars$color[cars$cyl==6] <- "blue"
cars$color[cars$cyl==8] <- "darkgreen"

dotchart(cars$mpg, labels=row.names(cars), cex=.7, groups= cars$cyl,
       main="Miles Per Gallon (MPG) of Car Models\nGrouped by Cylinder",
       xlab="Miles Per Gallon", color=cars$color, gcolor="black")
```



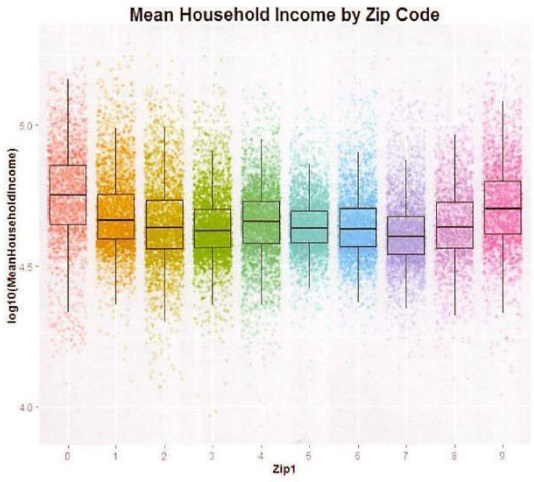
Barplot to visualize multiple variables

```
counts <- table(mtcars$gear, mtcars$cyl)
barplot(counts, main="Distribution of Car Cylinder Counts and Gears",
        xlab="Number of Cylinders", ylab="Counts",
        col=c("#0000FFFF", "#0080FFFF", "#00FFFFFF"),
        legend = rownames(counts), beside=TRUE,
        args.legend = list(x="top", title = "Number of Gears"))
```



Box-and-Whisker Plot

Box-and-whisker plots show the distribution of a continuous variable for each value of a discrete variable. The box-and-whisker plot in Figure visualizes mean household incomes as a function of region in the United States.

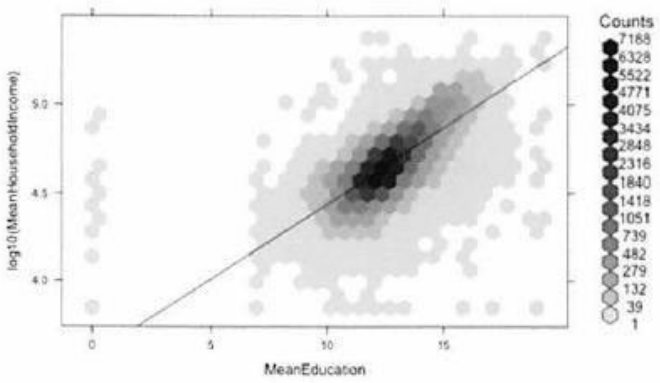



```
library(ggplot2)
# plot the jittered scatterplot w/ boxplot
# color-code points with zip codes
# the outlier.size=0 prevents the boxplot from plotting the outlier
ggplot(data=DF, aes(x=as.factor(Zip1), y=log10(MeanHouseholdIncome))) +
  geom_point(aes(color=factor(Zip1)), alpha=0.2, position="jitter") +
  geom_boxplot(outlier.size=0, alpha=0.1) +
  guides(colour=FALSE) +
  ggtitle ("Mean Household Income by Zip Code")
```

Hexbinplot for Large Data sets

A hexbinplot combines the ideas of scatterplot and histogram. Similar to a scatterplot, a hexbinplot visualizes data in the x-axis and y-axis. Data is placed into hex bins, and the third dimension uses shading to represent the concentration of data in each hexbin.

Hexbinplot of household income against years of education




```
library(hexbin)
# "g" adds the grid, "r" adds the regression line
# sqrt transform on the count gives more dynamic range to the shading
# inv provides the inverse transformation function of trans
hexbinplot(log10(MeanHouseholdIncome) ~ MeanEducation,
  data=zcta, trans = sqrt, inv = function(x) x^2, type=c("g", "r"))
```

Statistical Methods for Evaluation

Hypothesis Testing

When comparing populations, such as testing or evaluating the difference of the means from two samples of data a common technique to assess the difference or the significance of the difference is hypothesis testing.

The basic concept of hypothesis testing is to form an assertion and test it with data. When performing hypothesis tests, the common assumption is that there is no difference between two samples. This assumption is used as the default position for building the test or conducting a scientific experiment. Statisticians refer to this as the null hypothesis (H_0). The alternative hypothesis (H_A) is that there is a difference between two samples.

For example, if the task is to identify the effect of drug A compared to drug B on patients, the null hypothesis and alternative hypothesis would be this

- H_0: Drug A and drug B have the same effect on patients.
- H_A: Drug A has a greater effect than drug Bon patients.

Difference of Means

The difference in means can be tested using Student's t-test or the Welch's t-test.

Student's t-test

Student 's t-test assumes that distributions of the two populations have equal but unknown variances. Suppose n_1 and n_2 samples are randomly and independently selected from two populations, pop1 and pop2, respectively. If each population is normally distributed with the same mean ($\mu_1 = \mu_2$) and with the same variance, then T (the t-statistic), given in Equation, follows a t -distribution with $n_1 + n_2 - 2$ degrees of freedom (df).

$$T = \frac{\bar{X}_1 - \bar{X}_2}{S_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

where

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}$$

Ex. R code

```
# generate random observations from the two populations
x <- rnorm(10, mean=100, sd=5)     # normal distribution centered at 100
y <- rnorm(20, mean=105, sd=5)     # normal distribution centered at 105

t.test(x, y, var.equal=TRUE)       # run the Student's t-test
Two Sample t-test

data:  x and y
t = -1.7828, df = 28, p-value = 0.08547
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -6.1611557  0.4271893
sample estimates:
  mean of x mean of y
102.2136  105.0806
```

Welch 's t-test

When the equal population variance assumption is not justified in performing Student's t-test for the difference of means, Welch's t-test can be used based on T expressed in Equation

$$T_{welch} = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

where $\bar{X}_i$, S_i^2, and n_i correspond to the i-th sample mean, sample variance, and sample size. Notice that Welch's t-test uses the sample variance (S_i^2) for each population instead of the pooled sample variance.

Ex. R code

```
t.test(x, y, var.equal=FALSE)       # run the Welch's t-test

Welch Two Sample t-test

data:  x and y
t = -1.6596, df = 15.118, p-value = 0.1176
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -6.546629  0.812663
sample estimates:
  mean of x mean of y
102.2136  105.0806
```

The degrees of freedom for Welch's t-test is defined in Equation

$$df = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{\left(\frac{S_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{S_2^2}{n_2}\right)^2}{n_2 - 1}}$$

Wilcoxon Rank-Sum Test

The Wilcoxon rank-sum test is a nonparametric hypothesis test that checks whether two populations are identically distributed

Ex.

```
wilcox.test(x, y, conf.int = TRUE)
```

Type I and Type II Errors

A hypothesis test may result in two types of errors, depending on whether the test accepts or rejects the null hypothesis. These two errors are known as type I and type II errors.

- A type I error is the rejection of the null hypothesis when the null hypothesis is TRUE. The probability of the type I error is denoted by the Greek letter α
- A type II error is the acceptance of a null hypothesis when the null hypothesis is FALSE. The probability of the type II error is denoted by the Greek letter β

APEC

ANOVA (Analysis of Variance)

ANOVA is a generalization of the hypothesis testing of the difference of two population means. ANOVA tests if any of the population means differ from the other population means. The null hypothesis of ANOVA is that all the population means are equal. The alternative hypothesis is that at least one pair of the population means is not equal.

- $H_0: \mu_1 = \mu_2 = \ldots = \mu_n$
- $H_A: \mu_i \neq \mu_j$ for at least one pair of i, j

The **aov ( )** function performs the ANOVA

Ex.

Consider an example that every customer who visits a retail website gets one of two promotional offers or gets no promotion at all. The goal is to see if making the promotional offers makes a difference. ANOVA could be used, and the null hypothesis is that neither promotion makes a difference. The code that follows randomly generates a total of 500 observations of purchase sizes on three different offer options.

```
offers <- sample(c("offer1", "offer2", "nopromo"), size=500, replace=T)

# Simulated 500 observations of purchase sizes on the 3 offer options
purchasesize <- ifelse(offers=="offer1", rnorm(500, mean=80, sd=30),
                ifelse(offers=="offer2", rnorm(500, mean=85, sd=30),
                      rnorm(500, mean=40, sd=30)))

# create a data frame of offer option and purchase size
offertest <- data.frame(offer=as.factor(offers),
                        purchase_amt=purchasesize)
```

The summary of the offertest data frame shows that 170 offerl, 161 offer2, and 169 nopromo (no promotion) offers have been made. It also shows the range of purchase size (purchase_amt) for each of the three offer options.

```
# display a summary of offertest where offer="offer1"
summary(offertest[offertest$offer=="offer1",])
     offer      purchase_amt
 nopromo:  0   Min.   :  4.521
 offer1 :170   1st Qu.: 58.158
 offer2 :  0   Median : 76.944
               Mean   : 81.936
               3rd Qu.:104.959
               Max.   :180.507

# display a summary of offertest where offer="offer2"
summary(offertest[offertest$offer=="offer2",])
```

```
     offer       purchase_amt
 nopromo:  0   Min.   : 14.04
 offer1 :  0   1st Qu.: 69.46
 offer2 :161   Median : 90.20
               Mean   : 89.09
               3rd Qu.:107.48
               Max.   :154.33

# display a summary of offertest where offer="nopromo"
summary(offertest[offertest$offer=="nopromo",])
     offer       purchase_amt
 nopromo:169   Min.   :-27.00
 offer1 :  0   1st Qu.: 20.22
 offer2 :  0   Median : 42.44
               Mean   : 40.97
               3rd Qu.: 58.96
               Max.   :164.04
```

The aov ( ) function performs the ANOVA on purchase size and offer options.

```
# fit ANOVA test
model <- aov(purchase_amt ~ offers, data=offertest)
```

summary (model)

The **summary ( )** function shows a summary of the model

APEC