DEPARTMENT OF MINING ENGINEERING

Mine Data Analytics

(MNC306)

Mine Data Analytics Laboratory

Indian Institute of Technology

(Indian School of Mines)

DHANBAD-826004

List of Experiments

EXPERIMENT - 1

INTRODUCTION TO R PROGRAMMING AND INSTALLING R Studio.

AIM:

Introduction to R programming and its installation.

THEORY:

In R, the fundamental unit of share-able code is the package. A package bundles together code, data, documentation, and tests and provides an easy method to share with others1. As of May 2017 there were over 10,000 packages available on CRAN. This huge variety of packages is one of the reasons that R is so successful: chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package.

INSTALLING PACKAGES:

The most common place to get packages from is CRAN. To install packages from CRAN you use install.packages("packagename"). For instance, if you want to install the ggplot2 package, which is a very popular visualization package you would type the following in the console:

install package from CRAN

#install.packages("ggplot2")

Loading Packages

Once the package is downloaded to your computer you can access the functions and resources provided by the package in two different ways:

load the package to use in the current R session

library(packagename)

Getting Help on Packages

For more direct help on packages that are installed on your computer you can use the help and vignette functions.

Here we can get help on the ggplot2 package with the following:

help(package = "ggplot2") # provides details regarding contents of a package

vignette(package = "ggplot2") # list vignettes available for a specific package

vignette("ggplot2-specs") # view specific vignette

vignette() # view all vignettes on your computer

Assignment

Installing R and RStudio

https://rstudio-education.github.io/hopr/starting.html

x <- 3

Interestingly, R actually allows for five assignment operators:

leftward assignment

x <- value

x = value

x <<- value

rightward assignment

value -> x

value ->> x

The original assignment operator in R was <- and has continued to be the preferred among R users. The = assignment operator was added in 2001 primarily because it is the accepted assignment operator in many other languages and beginners to R coming from other languages were so prone to use it.

The operators <<- is normally only used in functions which we will not get into the details.

Evaluation

We can then evaluate the variable by simply typing x at the command line which will return the value of x. Note that prior to the value returned you'll see ## [1] in the command line. This simply implies that the output returned is the first output. Note that you can type any comments in your code by preceding the comment with the hash tag (#) symbol. Any values, symbols, and texts following # will not be evaluated. # evaluation

x ## [1] 3

Case Sensitivity

Lastly, note that R is a case sensitive programming language. Meaning all variables, functions, and objects must be called by their exact spelling:

x <- 1

y <- 3

z <- 4 x * y * z

[1] 12 x * Y * z

Error in eval(expr, envir, enclos): object 'Y' not found

Basic Arithmetic

At its most basic function R can be used as a calculator. When applying basic arithmetic, the PEMDAS order of operations applies: parentheses first followed by exponentiation, multiplication and division, and final addition and subtraction.

```
8 + 9 / 5 ^ 2
```

```
## [1] 8.36 8 + 9 / (5 ^ 2)
```

```
## [1] 8.36 8 + (9 / 5) ^ 2
```

```
## [1] 11.24 (8 + 9) / 5 ^ 2
```

```
## [1] 0.68
```

By default R will display seven digits but this can be changed using options() as previously outlined.

```
1 / 7
```

```
## [1] 0.1428571
```

```
options(digits = 3)
```

```
1 / 7
```

```
## [1] 0.143
```

```
pi
```

```
## [1] 3.141592654
```

```
options(digits = 22)
```

```
pi
```

```
## [1] 3.1415926535897931159
```

We can also perform integer divide (%/%) and modulo (%%) functions. The integer divide function will give the integer part of a fraction while the modulo will provide the remainder.

```
42 / 4
```

```
# regular division
```

```
## [1] 10.5
```

```
42 %/% 4 # integer division
```

```
## [1] 10
```

```
42 %% 4 # modulo (remainder)
```

```
## [1] 2
```

Miscellaneous Mathematical Functions

There are many built-in functions to be aware of. These include but are not limited to the following. Go ahead and run this code in your console.

x <- 10 abs(x) # absolute value

sqrt(x) # square root

exp(x) # exponential transformation

log(x) # logarithmic transformation

cos(x) # cosine and other trigonometric functions

Infinite, and NaN Numbers:

When performing undefined calculations, R will produce Inf (infinity) and NaN (not a number) outputs.

1 / 0 # infinity

[1] Inf

Inf - Inf # infinity minus infinity

[1] NaN

The workspace environment will also list your user defined objects such as vectors, matrices, data frames, lists, and functions. For example, if you type the following in your console:

x <- 2

y <- 3

You will now see x and y listed in your workspace environment. To identify or remove the objects (i.e. vectors, data frames, user defined functions, etc.) in your current R environment:

list all objects

ls()

identify if an R object with a given name is present

exists("x")

remove defined object from the environment

rm(x)

you can remove multiple objects

rm(x, y)

basically removes everything in the working environment -- use with caution!

rm(list = ls())

CONCLUSION:

In this way we had understand the basics of r programming.

TASK:

1. Install following packages for future use by running codes given below:

```
# install packages
install.packages("dplyr")
install.packages("ggplot2")
install.packages("stringr")
install.packages("boot")
install.packages("DescTools")
install.packages("ggpubr")
install.packages("flextable")
# installklippy for copy-to-clipboard button in code chunks
install.packages("remotes")
remotes::install_github("rlesur/klippy")
```

2. Now activate them by running codes given below:

```
# set options
options(stringsAsFactors = F)         # no automatic data transformation
options("scipen" = 100, "digits" = 12) # suppress math annotation
# activate packages
library(boot)
library(DescTools)
library(dplyr)
library(stringr)
library(ggplot2)
library(flextable)
library(ggpubr)
# activateklippy for copy-to-clipboard button
klippy::klippy()
```

EXPERIMENT - 2

DESCRIPTIVE STATISTISTICS WITH R

AIM:

To describe and showcase how data can be summarized using R.

THEORY:

Measures of central tendency:

In linguistics three measures of centrality or measures of central tendency are of particular relevance: the *mean*, the *median* and the *mode* (Gaddis and Gaddis 1990). In addition, there are two more measures of central tendency, the geometric and the harmonic mean which we will only briefly discuss as they are not that relevant for language research. What measure is appropriate depends on the type of variable scaling, the distribution of the data, and what is the intended aim of the data summary

Means	Use
(Arithmetic) mean (average)	Description of normally distributed numeric variables (most common measure of central tendency)
Median (middle value)	Description of non-normal numeric variables or ordinal variables (skewed data or influential outliers)
Mode (most frequent value)	Description of nominal and categorical variables
Geometric mean (average factor)	Description of dynamic processes such as growth rates
Harmonic mean (average rate)	Description of dynamic processes such as velocities

Fig1 : measures of central tendency and their use

In the following we will see how to calculate them in R.

MEAN:

Mean is defined as sum of the items divided by the number of items.

Calculate the mean of given numbers using R

3, 40, 15, 87

In R mean is calculated as follows:

```
# create numeric vector
frequencies <- c(3, 40, 15, 87)
```

```
# calculate mean
mean(frequencies)
```

Practice:

1. Calculate the arithmetic mean: 1, 2, 3, 4, 5, 6
2. Calculate the arithmetic mean for the following values using the mean function: 4, 3, 6, 2, 1, 5, 6, 8
3. Create a vector out of the following values and calculate the arithmetic mean using the mean function: 1, 5, 5, 9

MEDIAN:

The median is the central value in a de- or increasing ordering of values in a vector. In other words, 50 percent of values are above and 50 percent of values are below the median in a given vector.

If the vector contains an even number of elements, then the two central values are summed up and divided by 2. If the vector contains an uneven number of elements, the median represents the central value.

Example:

Number of speakers across age groups in the private dialogue section of the Irish component of the *International Corpus of English* (ICE).

S.NO.	Age	Counts
1	0-18	9
2	19-25	160
3	26-33	70
4	34-41	15
5	42-49	9
6	50+	57

In R, the *median* of above data is calculated as shown below:

```
# create a vector consisting out of ranks
ranks <- c(rep(1, 9), rep(2, 160), rep(3, 70), rep(4, 15), rep(5, 9), rep(6, 57))
# calculate median
```

```
median(ranks)
```

Practice:

1. Calculate the median: 1, 2, 3, 4, 5, 6
2. Calculate the median for the following values using the median function: 4, 3, 6, 2, 1, 5, 6, 8
3. Create a vector out of the following values and calculate the median using the median function: 1, 5, 5, 9

MODE:

The mode is typically used when dealing with categorical variables and it reports which level of a factor or a categorical variable is the most frequent.

Example data:

Number of speakers across counties of current residency in the private dialogue section of the Irish component of the *International Corpus of English* (ICE).

CurrentResidence	**Speakers**
Belfast	98
Down	20
Dublin (city)	110
Limerick	13
Tipperary	19

In R the *mode* is calculated as shown below:

```
# create a factor with the current residence of speakers
CurrentResidence<- c(rep("Belfast", 98),        # repeat "Belfast" 98 times
rep("Down", 20),           # repeat "Down" 20 times
rep("Dublin (city)", 110),  # repeat "Dublin (city)" 110 times
rep("Limerick", 13),        # repeat "Limerick" 13 times
rep("Tipperary", 19))       # repeat "Tipperary" 19 times
# calculate mode
names(which.max(table(CurrentResidence)))        # extract which level occurs most fr
equently
```

MEASURES OF VARIABILITY:

Measures of variability provide information about the distribution of values such as whether the data are distributed evenly and do not differ substantially or whether the data are rather heterogeneous and are distributed very unevenly (Thompson 2009). In the following, we will have a look at the *variance* and the *standard deviation*.

Example data:

Average temperature in Hamburg and Moscow by month.

Month	**Moscow**	**Hamburg**
January	-5.00	7.00
February	-12.00	7.00
March	5.00	8.00
April	12.00	9.00
May	15.00	10.00
June	18.00	13.00
July	22.00	15.00
August	23.00	15.00
September	20.00	13.00
October	16.00	11.00
November	8.00	8.00
December	1.00	7.00
Mean	10.25	10.25

Range:

The range is the simplest measure of variability and reports the lowest and highest value of a distribution. That is, the range provides minimum and maximum of a vector to show the span of values within a distribution.

In R, the *range* is extracted as shown below.

```
# create a numeric vector
Moscow <- c(-5, -12, 5, 12, 15, 18, 22, 23, 20, 16, 8, 1)
min(Moscow); max(Moscow) # extract range
```

Interquartile range (IQR)

The interquartile range (IQR) denotes the range that encompasses the central 50 percent of data points and thus informs about how values are distributed. This means that the IQR spans from the first quartile that encompasses 25 percent of the data to the third quartile that encompasses 75 percent of the data.

The easiest way to extract the IQR in R is to apply the summary function to a vector as shown below and then subtract the value of the 1st quartile from the value of the 3rd quartile.

```
summary(Moscow) # extract IQR
```

Standard deviation:

In R, the *standard deviation* is calculated as shown below.

```
# calculate standard deviation
sd(Moscow)
```

Variance:

In R variance is calculated as shown below

```
sd(Moscow)^2
```

Conclusion:

By this experiment we learned basics of descriptive statistics and their calculation in R programming.

TASK:

1. Calculate the mean, median, and mode as well as the standard deviation for the following two vectors
   A: 1, 3, 6, 2, 1, 1, 6, 8, 4, 2, 3, 5, 0, 0, 2, 1, 2, 1, 0
   B: 3, 2, 5, 1, 1, 4, 0, 0, 2, 3, 0, 3, 0, 5, 4, 5, 3, 3, 4

EXPERIMENT - 3

CORRELATION TEST BETWEEN TWO VARIABLES

AIM:

To describe different correlation methods and to provide practical examples using R

THEORY:

There are different methods to perform **correlation analysis**:

- **Pearson correlation (r)**, which measures a linear dependence between two variables (x and y). It's also known as a **parametric correlation** test because it depends to the distribution of the data. It can be used only when x and y are from normal distribution. The plot of y = f(x) is named the **linear regression** curve.
- **Kendall tau** and **Spearman rho**, which are rank-based correlation coefficients (non-parametric)
- **T**he most commonly used method is the pearson correlation method

Compute correlation in R

R functions

Correlation coefficient can be computed using the functions **cor()** or **cor.test()**:

- **cor()** computes the **correlation coefficient**
- **cor.test()** test for association/correlation between paired samples. It returns both the **correlation coefficient** and the **significance level**(or p-value) of the correlation .

The simplified formats are:

```
cor(x, y, method=c("pearson", "kendall", "spearman"))
cor.test(x, y, method=c("pearson", "kendall", "spearman"))
```

- **x, y**: numeric vectors with the same length
- **method**: correlation method

```
cor(x, y,  method="pearson", use="complete.obs")
```

Import your data into R

1. **Prepare your data**
2. **Save your data** in an external .txt tab or .csv files
3. **Import your data into R** as follow:

```
# If .txt tab file, use this
my_data<-read.delim(file.choose())
# Or, if .csv file, use this
my_data<-read.csv(file.choose())
```

Here, we'll use the built-in R data set *mtcars* as an example.
The R code below computes the correlation between mpg and wt variables in mtcars data set:

```
my_data<-mtcars
head(my_data, 6)
```

We want to compute the correlation between mpg and wt variables

Visualize your data using scatter plots

To use R base graphs, click this link: scatter plot - R base graphs. Here, we'll use the **ggpubr** R package.

```
library("ggpubr")
ggscatter(my_data, x="mpg", y="wt",
add="reg.line", conf.int=TRUE,
cor.coef=TRUE, cor.method="pearson",
xlab="Miles/(US) gallon", ylab="Weight (1000 lbs)")
```

Preleminary test to check the test assumptions

1. **Is the covariation linear**? Yes, form the plot above, the relationship is linear. In the situation where the scatter plots show curved patterns, we are dealing with nonlinear association between the two variables.
2. **Are the data from each of the 2 variables (x, y) follow a normal distribution**?
    - Use Shapiro-Wilk normality test –> R function: **shapiro.test**()
    - and look at the normality plot —> R function: **ggpubr::ggqqplot**()

- **Shapiro-Wilk test** can be performed as follow:
    - Null hypothesis: the data are normally distributed
    - Alternative hypothesis: the data are not normally distributed

```
# Shapiro-Wilk normality test for mpg
shapiro.test(my_data$mpg)# => p = 0.1229
```

```
# Shapiro-Wilk normality test for wt
shapiro.test(my_data$wt)# => p = 0.09
```

- **Visual inspection** of the data normality using **Q-Q plots** (quantile-quantile plots). Q-Q plot draws the correlation between a given sample and the normal distribution.

```
library("ggpubr")
# mpg
ggqqplot(my_data$mpg, ylab="MPG")
# wt
ggqqplot(my_data$wt, ylab="WT")
```

Pearson correlation test

Correlation test between mpg and wt variables:

```
res<-cor.test(my_data$wt, my_data$mpg,
method="pearson")
res
```

Interpretation of the result:

Access to the values returned by cor.test() function

The function **cor.test**() returns a list containing the following components:

- **p.value**: the p-value of the test
- **estimate**: the correlation coefficient

```
# Extract the p.value
res$p.value
```

```
# Extract the correlation coefficient
res$estimate
cor
```

Kendall rank correlation test

The **Kendall rank correlation coefficient** or **Kendall's tau** statistic is used to estimate a rank-based measure of association. This test may be used if the data do not necessarily come from a bivariate normal distribution.

```
res2<-cor.test(my_data$wt, my_data$mpg,  method="kendall")
res2
```

Spearman rank correlation coefficient

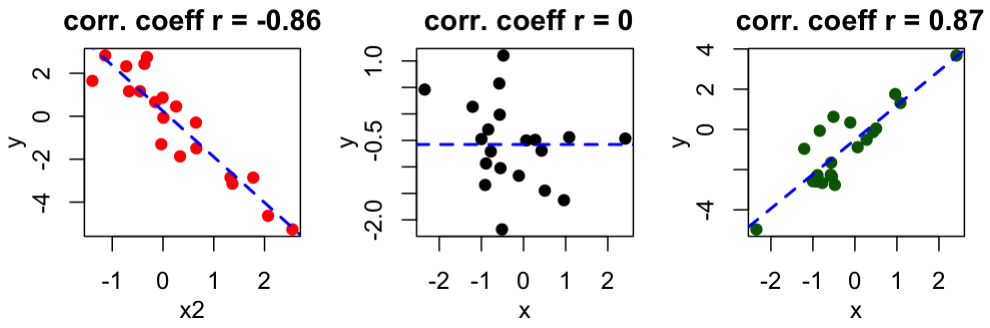
Spearman's **rho** statistic is also used to estimate a rank-based measure of association. This test may be used if the data do not come from a bivariate normal distribution.

```
res2<-cor.test(my_data$wt, my_data$mpg,  method="spearman")
res2
```

Interpret correlation coefficient

Correlation coefficient is comprised between **-1** and **1**:

- **-1** indicates a strong **negative correlation** : this means that every time **x increases**, **y decreases** (left panel figure)
- **0** means that there is no **association** between the two variables (x and y) (middle panel figure)
- **1** indicates a strong **positive correlation** : this means that **y increases** with **x** (right panel figure)



CONCLUSION:

- Use the function **cor.test**(x,y) to analyze the correlation coefficient between two variables and to get significance level of the correlation.
- Three possible correlation methods using the function **cor.test**(x,y): pearson, kendall, spearman

EXPERIMENT - 4

HYPOTHESIS TESTING IN R AND EXCEL

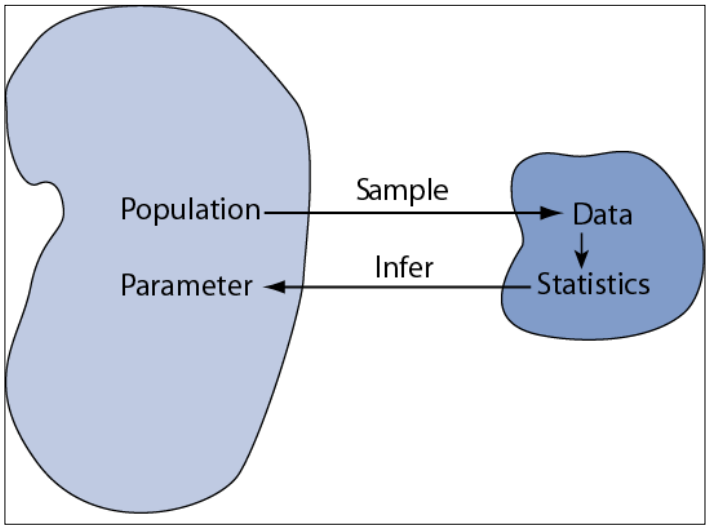
AIM:

Hypothesis testing (also called significance testing) uses a quasi-deductive procedure to judge claims about parameters. Before testing a statistical hypothesis it is important to clearly state the nature of the claim to be tested.

- ✓ Given a claim, identify the null hypothesis and the alternative hypothesis, and express them both in symbolic form.
- ✓ Given a claim and sample data, calculate the value of the test statistic.
- ✓ Given a significance level, identify the critical value(s).
- ✓ Given a value of the test statistic, identify the *P*-value.
- ✓ State the conclusion of a hypothesis test in simple, non-technical terms**.**

THEORY:

In statistics, a hypothesis is a claim or statement about a property of a population. A hypothesis test (or test of significance) is a standard procedure for testing a claim about a property of a population.



- Population ≡ all possible values
- Sample ≡ a portion of the population
- Statistical inference ≡ generalizing from a sample to a population with calculated degree of certainty
- Two forms of statistical inference
  - Hypothesis testing
  - Estimation
- Parameter ≡ a characteristic of population, e.g., population mean μ
- Statistic ≡ calculated from data in the sample, e.g., sample mean ( )

Components of a Formal Hypothesis Test

1. Convert the research question to null and alternative hypotheses
2. The null hypothesis (H_0) is a claim of "no difference in the population"
3. The alternative hypothesis (H_a) claims "H_0 is false"
4. Collect data and seek evidence against H_0 as a way of bolstering H_a (deduction)

Null Hypothesis *H*0

- ✓ The null hypothesis (denoted by H0) is a statement that the value of a population parameter (such as proportion, mean, or standard deviation) is equal to some claimed value.
- ✓ We test the null hypothesis directly.
- ✓ Either reject H0 or fail to reject H0.

Alternative Hypothesis: H1

- ✓ The alternative hypothesis (denoted by H1 or Ha or HA) is the statement that the parameter has a value that somehow differs from the null hypothesis.
- ✓ The symbolic form of the alternative hypothesis must use one of these symbols: ≠, <, >.

Illustrative Example: "Body Weight"

- **The problem:** In the 1970s, 20–29 year old men in the U.S. had a mean μ body weight of 170 pounds. Standard deviation σ was 40 pounds. We test whether mean body weight in the population now differs.
- **Null hypothesis** $H_{0:}$ μ = 170 ("no difference")
- The **alternative hypothesis** can be either $H_{a:}$ μ > 170 (**one-sided test**) or $H_{a:}$ μ < 170 (**one-sided test**) or
  $H_{a:}$ μ ≠ 170 (**two-sided test**)

Test Statistic

The test statistic is a value used in making a decision about the null hypothesis, and is found by converting the sample statistic to a score with the assumption that the null hypothesis is true.

Two-Sided Tests

- Suppose a pharmaceutical company manufactures ibuprofen pills. They need to perform some quality assurance to ensure they have the correct dosage, which is supposed to be 500 milligrams. This is a two-sided test because if the company's pills are deviating significantly in either direction, meaning there are more than 500 milligrams or less than 500 milligrams, this will indicate a problem.

$$H_0: \mu = 500\ mg$$

$$H_A: \mu \neq 500\ mg$$

Z score calculation

- In a random sample of 125 pills, there is an average dose of 499.3 milligrams with a standard deviation of 6 milligrams. Because this is quantitative data, 500 mg is the population mean. We can use the following formula to calculate the z-score:

$$z = \frac{\overline{x} - \mu}{\frac{\sigma}{\sqrt{n}}}$$

$$\overline{x} = sample\ mean = 499.3\ mg$$

$$\mu = population\ mean = 500\ mg$$

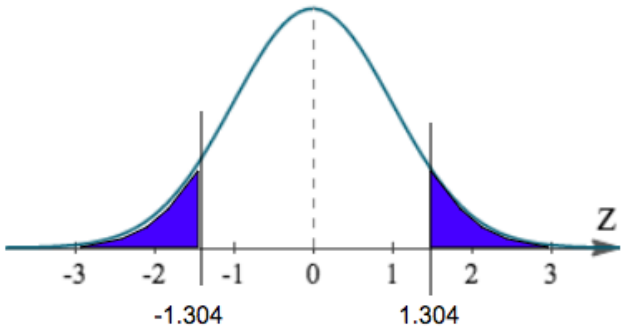
$$\sigma = sample\ standard\ deviation = 6\ mg$$

$$n = sample\ size = 123$$

$$z = \frac{\overline{x} - \mu}{\frac{\sigma}{\sqrt{n}}} = \frac{499.3 - 500}{\frac{6}{\sqrt{125}}} = \frac{-0.7}{\frac{6}{\sqrt{125}}} = \frac{-0.7}{\frac{6}{11.18}} = \frac{-0.7}{0.53667} = -1.304$$

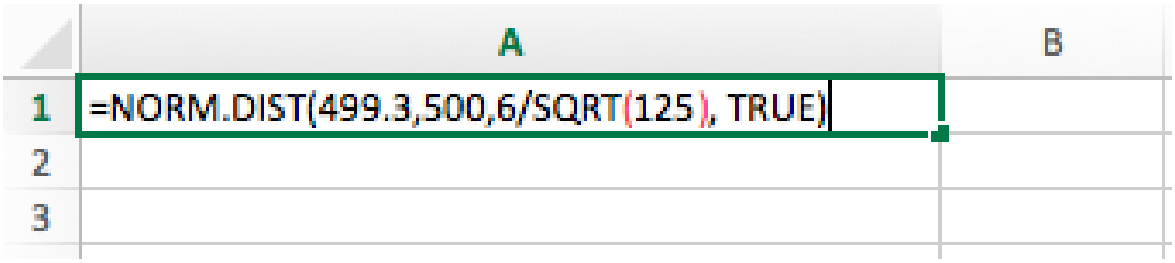
Z score interpretation

- We get a z-score of negative 1.304. Because this is a two-sided test, it is not enough to just look at the left tail. We also have to look at the equivalent of the right tail, or a positive 1.304.
- Now that we have the z-score, we can use a variety of methods to find the probability, or p-value.



Z-score to p-value

- This results in a p-value of 0.0968, or 9.68%, for a z-score of negative 1.304. We also need to take the positive 1.304 into account, which is the upper right tail.
- To calculate the true p-value, we just need to multiply 0.0968 by two, or 0.1936. This would be a p-value of 19.36%.



Interpretation

- *P-value answer the question: What is the probability of the observed test statistic … when H0 is true?*
- *Thus, smaller and smaller P-values provide stronger and stronger evidence against H0*
- *Small P-value ⇒ strong evidence*

Conventions*

- $P > 0.10 \Rightarrow$ non-significant evidence against H_0
- $0.05 < P \leq 0.10 \Rightarrow$ marginally significant evidence
- $0.01 < P \leq 0.05 \Rightarrow$ significant evidence against H_0
- $P \leq 0.01 \Rightarrow$ highly significant evidence against H_0
- Examples
- $P = .27 \Rightarrow$ non-significant evidence against H_0

- P =.01 $\Rightarrow$ highly significant evidence against H_0

α-Level (Used in some situations)

- Let α ≡ probability of erroneously rejecting H0
- Set α threshold (e.g., let α = .10, .05, or whatever)
- Reject H0 when P ≤ α
- Retain H0 when P > α
- Example: Set α = .10. Find P = 0.27 $\Rightarrow$ retain H0
- Example: Set α = .01. Find P = .001 $\Rightarrow$ reject H0

Practice Problems

1) A z-test statistic = 2.97 for a two-sided test. What is the p-value?

2) A z-test statistic = -0.95 for a left-tailed test. What is the p-value?

3) A z-test statistic = 1.81 for a right-tailed test. What is the p-value?

Question #:	Correct Answer
1	p-value = 0.003 (0.3%)
2	p-value = 0.171 (17.1%)
3	p-value = 0.035 (3.5%)

Exercise

We have a sample of 106 body temperatures having a mean of 98.20°F. Assume that the sample is a simple random sample and that the population standard deviation σ is known to be 0.62°F. Use a 0.05 significance level to test the common belief that the mean body temperature of healthy adults is equal to 98.6°F. Use the *P*-value method

EXPERIMENT – 5

LINEAR REGRESSION IN R

AIM:

To describe different regression methods and to provide practical examples using R

THEORY:

Linear regression is a regression model that uses a straight line to describe the relationship between variables. It finds the line of best fit through your data by searching for the value of the regression coefficient(s) that minimizes the total error of the model.

There are two main types of linear regression:

- **Simple linear regression** uses only one independent variable
- **Multiple linear regression** uses two or more independent variables

In this class we will learn step-by-step guideonsimple linear regression in R.

Sample dataset:

The dataset contains observations about income (in a range of $15k to 75k) and happiness (rated on a scale of 1 to 10) in an imaginary sample of 500 people. The income values are divided by 10000 to make the income data match the scale of happiness scores (so a value of $2 represents $20,000, $3 is $30,000, etc.)

Download dataset from:

https://cdn.scribbr.com/wp-content/uploads//2020/02/income.data_.zip

STEPS FOR SIMPLE LINEAR REGRESSION IN R

Step1: Load data into R

Follow these four steps for each dataset:

1. In RStudio, go to **File > Import dataset > From Text (base)**.
2. Choose the data file you have downloaded (income.data or heart.data), and an **Import Dataset** window pops up.
3. In the **Data Frame** window, you should see an **X** (index) column and columns listing the data for each of the variables (income and happiness or biking, smoking, and heart.disease).
4. Click on the **Import** button and the file should appear in your **Environment** tab on the upper right side of the RStudio screen.

After you've loaded the data, check that it has been read in correctly using summary().

```
summary(income.data)
```

when we run this function, we see a table in our console with a numeric summary of the data.

Step 2: Make sure the data meet the assumptions

1. **Independence of observations:**

   Because we only have one independent variable and one dependent variable, we don't need to test for any hidden relationships among variables.

2. **Normality:**

   To check whether the dependent variable follows a normal distribution, use the hist() function.

   ```
   hist(income.data$happiness)
   ```

3. **Linearity:**

   The relationship between the independent and dependent variable must be linear. We can test this visually with a scatter plot to see if the distribution of data points could be described with a straight line.

   ```
   plot(happiness ~ income, data = income.data)
   ```

The relationship looks roughly linear, so we can proceed with the linear model.

4. **Homoscedasticity**

   This means that the prediction error doesn't change significantly over the range of prediction of the model. We can test this assumption later, after fitting the linear model.

Step 3: perform the linear regression analysis

Now that you've determined your data meet the assumptions, you can perform a linear regression analysis to evaluate the relationship between the independent and dependent variables.

To perform a simple linear regression analysis and check the results, you need to run two lines of code. The first line of code makes the linear model, and the second line prints out the summary of the model:

```
income.happiness.lm<- lm(happiness ~ income, data = income.data)

summary(income.happiness.lm)
```

Step 4: check for Homoscedasticity

We can run plot(income.happiness.lm) to check whether the observed data meets our model assumptions:

```
par(mfrow=c(2,2))
plot(income.happiness.lm)
par(mfrow=c(1,1))
```

Note that the par(mfrow()) command will divide the **Plots** window into the number of rows and columns specified in the brackets. So par(mfrow=c(2,2)) divides it up into two rows and two columns. To go back to plotting one graph in the entire window, set the parameters again and replace the (2,2) with (1,1).:

The most important thing to look for is that the red lines representing the mean of the residuals are all basically horizontal and centered around zero. This means there are no outliers or biases in the data that would make a linear regression invalid.

Step 5: Visualize the results with a graph

Follow 4 steps to visualize the results of your simple linear regression.

1. **Plot the data points on graph**

```
income.graph<-ggplot(income.data, aes(x=income, y=happiness))+
geom_point()
income.graph
```

2. **Add the linear regression line to the plotted data**

Add the regression line using geom_smooth() and typing in lm as your method for creating the line. This will add the line of the linear regression as well as the standard error of the estimate (in this case +/- 0.01) as a light grey stripe surrounding the line:

```
income.graph<- income.graph + geom_smooth(method="lm", col="black")

income.graph
```

3. **Add the equation for the regression line:**

```
income.graph<- income.graph +
stat_regline_equation(label.x = 3, label.y = 7)
```

income.graph

4. **Make the graph ready for publication**

We can add some style parameters using theme_bw() and making custom labels using labs().

```
income.graph +

theme_bw() +
labs(title = "Reported happiness as a function of income",
x = "Income (x$10,000)",
y = "Happiness score (0 to 10)")
```

This produces the finished graph that you can include in your papers:

Step 6: Reporting results

In addition to the graph, include a brief statement explaining the results of the regression model.

CONLUSION:

By this experiment we learned basics of linear regression and its calculation calculation in R programming.

EXPERIMENT - 6

MULTIPLE LINEAR REGRESSION IN R

AIM:

To describe different regression methods and to provide practical examples using R

THEORY:

Linear regression is a regression model that uses a straight line to describe the relationship between variables. It finds the line of best fit through your data by searching for the value of the regression coefficient(s) that minimizes the total error of the model.

There are two main types of linear regression:

- **Simple linear regression** uses only one independent variable
- **Multiple linear regression** uses two or more independent variables

In this class we will learn step-by-step guide on Multiple linear regression in R.

Sample dataset:

The data set contains observations on the percentage of people biking to work each day, the percentage of people smoking, and the percentage of people with heart disease in an imaginary sample of 500 towns.

Download the dataset from

https://cdn.scribbr.com/wp-content/uploads//2020/02/heart.data_.zip

STEPS FOR MULTIPLE LINEAR REGRESSION IN R

Follow these four steps for each dataset:

1. In RStudio, go to **File > Import dataset > From Text (base)**.
2. Choose the data file you have downloaded (income.data or heart.data), and an **Import Dataset** window pops up.
3. In the **Data Frame** window, you should see an **X** (index) column and columns listing the data for each of the variables (income and happiness or biking, smoking, and heart.disease).
4. Click on the **Import** button and the file should appear in your **Environment** tab on the upper right side of the RStudio screen.

After you've loaded the data, check that it has been read in correctly using summary().

```
summary(heart.data)
```

running the code produces a numeric summary of the data for the independent variables (smoking and biking) and the dependent variable (heart disease):

Step 2: **Make sure the data meet the assumptions**

1. **Independence of observations**

Use the cor() function to test the relationship between your independent variables and make sure they aren't too highly correlated.

```
cor(heart.data$biking, heart.data$smoking)
```

2. **Normality**

Use the hist() function to test whether your dependent variable follows a normal distribution.

```
hist(heart.data$heart.disease)
```

The distribution of observations is roughly bell-shaped, so we can proceed with the linear regression.

3. **Linearity**

We can check this using two scatterplots: one for biking and heart disease, and one for smoking and heart disease.

```
plot(heart.disease ~ biking, data=heart.data)
```

```
plot(heart.disease ~ smoking, data=heart.data)
```

Although the relationship between smoking and heart disease is a bit less clear, it still appears linear. We can proceed with linear regression.

4. **Homoscedasticity**

We will check this after we make the model.

Step 3: Perform the linear regression analysis

To test the relationship, we first fit a linear model with heart disease as the dependent variable and biking and smoking as the independent variables. Run these two lines of code:

```
heart.disease.lm<-lm(heart.disease ~ biking + smoking, data = heart.data)
```

```
summary(heart.disease.lm)
```

Step 4: Check for homoscedasticity

Again, we should check that our model is actually a good fit for the data, and that we don't have large variation in the model error, by running this code:

```
par(mfrow=c(2,2))
plot(heart.disease.lm)
par(mfrow=c(1,1))
```

As with our simple regression, the residuals show no bias, so we can say our model fits the assumption of homoscedasticity.

Step 5: Visualize the results with a graph

plotting the relationship between biking and heart disease at different levels of smoking. In this example, smoking will be treated as a factor with three levels, just for the purposes of displaying the relationships in our data.

There are 7 steps to follow.

1. **Create a new dataframe with the information needed to plot the model**

Use the function expand.grid() to create a dataframe with the parameters you supply. Within this function we will:

- Create a sequence from the lowest to the highest value of your observed biking data;
- Choose the minimum, mean, and maximum values of smoking, in order to make 3 levels of smoking over which to predict rates of heart disease.

```
plotting.data<-expand.grid(
biking = seq(min(heart.data$biking), max(heart.data$biking), length.out=30),
smoking=c(min(heart.data$smoking), mean(heart.data$smoking),
max(heart.data$smoking)))
```

This will not create anything new in your console, but you should see a new data frame appear in the **Environment** tab. Click on it to view it.

2. **Predict the values of heart disease based on your linear model**

Next we will save our 'predicted y' values as a new column in the dataset we just created.

```
plotting.data$predicted.y<- predict.lm(heart.disease.lm, newdata=plotting.data)
```

3. **Round the smoking numbers to two decimals**

This will make the legend easier to read later on.

```
plotting.data$smoking<- round(plotting.data$smoking, digits = 2)
```

4. **Change the 'smoking' variable into a factor**

This allows us to plot the interaction between biking and heart disease at each of the three levels of smoking we chose.

```
plotting.data$smoking<- as.factor(plotting.data$smoking)
```

5. **Plot the original data**

```
heart.plot<- ggplot(heart.data, aes(x=biking, y=heart.disease)) +
geom_point()

heart.plot
```

6. **Add the regression lines**

```
heart.plot<- heart.plot +
geom_line(data=plotting.data, aes(x=biking, y=predicted.y, color=smoking), size=1.25)

heart.plot
```

7. **Make the graph ready for publication**

```
heart.plot<-
heart.plot +
theme_bw() +
labs(title = "Rates of heart disease (% of population) \n as a function of biking to work and smoking",
x = "Biking to work (% of population)",
y = "Heart disease (% of population)",
color = "Smoking \n (% of population)")

heart.plot
```

Step 6: reporting the results

In our survey of 500 towns, we found significant relationships between the frequency of biking to work and the frequency of hear disease and the frequency of smoking and frequency of heart disease ($p < 0$ and $p < 0.001$, respectively).

EXPERIMENT – 7

INTRODUCTION TO PYTHON AND ITS INSTALLATION

AIM:

Introduction to python and its Installation.

THEORY:

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and Sample Code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

COMPARING PYTHON TO OTHER LANGUAGES:

Java: Python programs are generally expected to run slower than Java programs, but they also take much less time to develop. Python programs are typically 3-5 times shorter than equivalent Java programs. This difference can be attributed to Python's built-in high-level data types and its dynamic typing.

JavaScript: Python's "object-based" subset is roughly equivalent to JavaScript. Like JavaScript (and unlike Java), Python supports a programming style that uses simple functions and variables without engaging in class definitions. However, for JavaScript, that's all there is. Python, on the other hand, supports writing much larger programs and better Sample Code reuse through a true object-oriented programming style, where classes and inheritance play an important role.

Pearl: Python and Perl come from a similar background (Unix scripting, which both have long outgrown), and sport many similar features, but have a different philosophy. Perl emphasizes support for common application-oriented tasks, e.g. by having built-in regular expressions, file scanning and report generating features. Python emphasizes support for common programming methodologies such as data structure design and object-oriented programming, and encourages programmers to write readable (and thus maintainable) Sample Code by providing an elegant but not overly cryptic notation. As a consequence, Python comes close to Perl but rarely beats it in its original application domain; however, Python has an applicability well beyond Perl's niche.

C++: Almost everything said for Java also applies for C++, just more so: where Python Sample Code is typically 3-5 times shorter than equivalent Java Sample Code, it is often 5-10 times shorter than equivalent C++ Sample Code! Anecdotal evidence suggests that one Python programmer can finish in two months what two C++ programmers can't

complete in a year. Python shines as a glue language, used to combine components written in C++.

STEP TO INSTALL PYTHON USING ANACONDA ENVIRONMENT:

For this class, the easiest way to setup Python (Links to an external site.) and get all the packages you need will be to install Anaconda (Links to an external site.) on your machine. Anaconda provides a virtual environment for you with preconfigured packages. This guarantees that everything will work smoothly for you.

You should be using the latest version of the language (Python 3.7), but we've tried to make sure the code we give you is backwards compatible.

TO GET ANACONDA ON YOUR MACHINE:

1. Go to https://www.anaconda.com/download (Links to an external site.) and click Download for the Python 3.7 version
2. Run the install executable or package. In Windows, we recommend checking the" Add to Path" option in the installer.
3. Run the Windows command line or Unix terminal.
4. Run the following lines of code:

5. $ conda create --name cs355env
6. $ activate cs355env (or source activate cs355env on Unix)
7. $ conda install numpy
8. $ conda install scipy
9. $ conda install matplotlib
10. $ conda install pil (or pillow)
11. $ conda install jupyter
12. $ pip install pygame
13. $ pip install sounddevice

Note: if you've previously installed Python on your machine, it's always a good idea to update conda and pip before using them to install new stuff.

14. On mac or linux machines, Run

15. $ conda install pyopengl

```
16. $ conda install pyopengl-accelerate
```

SAMPLE CODE:

Open Jupyter/ PyCharm, and write the below Sample Code

print (“Hello world”)

Tasks:

1. Install the python on your personal computer.

2. Install the required library using mentioned command.

3. Record.

EXPERIMENT – 8

LOGISTIC REGRESSION IN PYTHON

IN THIS CLASS YOU'LL SEE THE FOLLOWING:

- **A summary of Python packages** for logistic regression (NumPy, scikit-learn, StatsModels, and Matplotlib)
- **One illustrative example** of logistic regression solved with scikit-learn
- **One conceptual example** solved with StatsModels

Let's start implementing logistic regression in Python!

LOGISTIC REGRESSION PYTHON PACKAGES:

There are several packages you'll need for logistic regression in Python.

NumPy: it is a fundamental package for scientific and numerical computing in Python. NumPy is useful and popular because it enables high-performance operations on single- and multi-dimensional arrays.

scikit-learn: This is one of the most popular data science and machine learning libraries.

StatsModels: useful. It's a powerful Python library for statistical analysis.

Matplotlib: to visualize the results of your classification. This is a Python library that's comprehensive and widely used for high-quality plotting.

For additional information, you can check the official website and user guide.

Logistic Regression in Python With scikit-learn: Example 1
The first example is related to a single-variate binary classification problem. This is the most straightforward kind of classification problem. There are several general steps you'll take when you're preparing your classification models:

1. **Import** packages, functions, and classes
2. **Get** data to work with and, if appropriate, transform it
3. **Create** a classification model and train (or fit) it with your existing data
4. **Evaluate** your model to see if its performance is satisfactory

A sufficiently good model that you define can be used to make further predictions related to new, unseen data. The above procedure is the same for classification and regression.

Step 1: Import Packages, Functions, and Classes
First, you have to import Matplotlib for visualization and NumPy for array operations. You'll also need LogisticRegression, classification_report(), and confusion_matrix() from scikit-learn:

```
importmatplotlib.pyplotasplt
importnumpyasnp
```

```
fromsklearn.linear_modelimportLogisticRegression

fromsklearn.metricsimportclassification_report,confusion_matrix
```

Now you've imported everything you need for logistic regression in Python with scikit-learn!

Step 2: Get Data

In practice, you'll usually have some data to work with. For the purpose of this example, let's just create arrays for the input (□) and output (□) values:

```
x=np.arange(10).reshape(-1,1)

y=np.array([0,0,0,0,1,1,1,1,1,1])
```

The input and output should be NumPy arrays (instances of the class numpy.ndarray) or similar objects. numpy.arange() creates an array of consecutive, equally-spaced values within a given range. For more information on this function, check the official documentation or NumPy arange(): How to Use np.arange().

The array x is required to be **two-dimensional**. It should have one column for each input, and the number of rows should be equal to the number of observations. To make x two-dimensional, you apply .reshape() with the arguments -1 to get as many rows as needed and 1 to get one column. For more information on .reshape(), you can check out the official documentation. Here's how x and y look now:

```
>>>x
```

```
>>>y
```

x has two dimensions:

1. **One column** for a single input
2. **Ten rows**, each corresponding to one observation

y is one-dimensional with ten items. Again, each item corresponds to one observation. It contains only zeros and ones since this is a binary classification problem.

Step 3: Create a Model and Train It

Once you have the input and output prepared, you can create and define your classification model. You're going to represent it with an instance of the class LogisticRegression:

```
model=LogisticRegression(solver='liblinear',random_state=0)
```

The above statement creates an instance of LogisticRegression and binds its references to the variable model.

You should carefully match the solver and regularization method for several reasons:

- 'liblinear' solver doesn't work without regularization.
- 'newton-cg', 'sag', 'saga', and 'lbfgs' don't support L1 regularization.

- 'saga' is the only solver that supports elastic-net regularization.

Once the model is created, you need to fit (or train) it. Model fitting is the process of determining the coefficients b_0, b_1, …, b_r that correspond to the best value of the cost function. You fit the model with .fit():

```
model.fit(x,y)
```

.fit() takes x, y, and possibly observation-related weights. Then it fits the model and returns the model instance itself:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
intercept_scaling=1, l1_ratio=None, max_iter=100,
multi_class='warn', n_jobs=None, penalty='l2',
random_state=0, solver='liblinear', tol=0.0001, verbose=0,
warm_start=False)
```

This is the obtained string representation of the fitted model.

You can use the fact that .fit() returns the model instance and chain the last two statements. They are equivalent to the following line of code:

```
model=LogisticRegression(solver='liblinear',random_state=0).fit(x,y)
```

At this point, you have the classification model defined.

You can quickly get the attributes of your model. For example, the attribute .classes_ represents the array of distinct values that y takes:

```
>>>model.classes_
```

This is the example of binary classification, and y can be 0 or 1, as indicated above.

You can also get the value of the slope b_1 and the intercept b_0 of the linear function f like so:

```
>>>model.intercept_
```

```
>>>model.coef_
```

As you can see, b_0 is given inside a one-dimensional array, while b_1 is inside a two-dimensional array. You use the attributes .intercept_ and .coef_ to get these results.

Step 4: Evaluate the Model

Once a model is defined, you can check its performance with .predict_proba(), which returns the matrix of probabilities that the predicted output is equal to zero or one:

```
>>>model.predict_proba(x)
```

You can get the actual predictions, based on the probability matrix and the values of (□), with .predict():

```
>>>model.predict(x)
```

This function returns the predicted output values as a one-dimensional array.

.

When you have nine out of ten observations classified correctly, the accuracy of your model is equal to 9/10=0.9, which you can obtain with .score():

>>>python3.7

```
>>>model.score(x,y)
```

.

score() takes the input and output as arguments and returns the ratio of the number of correct predictions to the number of observations.

You can get a more comprehensive report on the classification with classification_report():

```
>>>print(classification_report(y,model.predict(x)))
```

This function also takes the actual and predicted outputs as arguments. It returns a report on the classification as a dictionary if you provide output_dict=True or a string otherwise.

Logistic Regression in Python WithStatsModels:

Example

The procedure is similar to that of scikit-learn.

Step 1: Import Packages

All you need to import is NumPy and statsmodels.api:

```
importnumpyasnp

importstatsmodels.apiassm
```

Now you have the packages you need.

Step 2: Get Data
You can get the inputs and output the same way as you did with scikit-learn. However, StatsModels doesn't take the intercept β_0 into account, and you need to include the additional column of ones in x. You do that with add_constant():

```
x=np.arange(10).reshape(-1,1)
y=np.array([0,1,0,0,1,1,1,1,1,1])
x=sm.add_constant(x)
```

add_constant() takes the array x as the argument and returns a new array with the additional column of ones. This is how x and y look:

```
>>>x
```

```
>>>y
```

This is your data. The first column of x corresponds to the intercept β_0. The second column contains the original values of x.

Step 3: Create a Model and Train It

Your logistic regression model is going to be an instance of the class statsmodels.discrete.discrete_model.Logit. This is how you can create one:

```
>>>model=sm.Logit(y,x)
```

Note that the first argument here is y, followed by x.

Now, you've created your model and you should fit it with the existing data. You do that with .fit() or, if you want to apply L1 regularization, with .fit_regularized():

```
>>>result=model.fit(method='newton')
```

The model is now ready, and the variable result holds useful data. For example, you can obtain the values of β_0 and β_1 with .params:

```
>>>result.params
```

The first element of the obtained array is the intercept β_0, while the second is the slope β_1. For more information, you can look at the official documentation on Logit, as well as .fit() and .fit_regularized().

Step 4: Evaluate the Model
You can use results to obtain the probabilities of the predicted outputs being equal to one:

>>>

```
>>>result.predict(x)
```

These probabilities are calculated with .predict(). You can use their values to get the actual predicted outputs:

```
>>>(result.predict(x)>=0.5).astype(int)
```

.

You can obtain the confusion matrix with .pred_table():

```
>>>result.pred_table()
```

This example is the same as when you used scikit-learn because the predicted ouptuts are equal. The confusion matrices you obtained with StatsModels and scikit-learn differ in the types of their elements (floating-point numbers and integers).

.summary() and .summary2() get output data that you might find useful in some circumstances:

```
>>>result.summary()
<class 'statsmodels.iolib.summary.Summary'>
================================================================================
=====
>>>result.summary2()
<class 'statsmodels.iolib.summary2.Summary'>
```

.

CONCLUSION:

You now know what **logistic regression** is and how you can implement it for **classification** with Python. You've used many open-source packages, including NumPy, to work with arrays and Matplotlib to visualize the results. You also used both scikit-learn and StatsModels to create, fit, evaluate, and apply models.

EXPERIMENT – 9

DATA VISUALIZATION- EXPLORATORY DATA ANALYSIS AND DATA CLEANING REGRESSION

AIM: To perform data analysis and Data cleaning regression.

INTRODUCTION:

Exploratory Data Analysis or EDA is used to take insights from the data. Data Scientists and Analysts try to find different patterns, relations, and anomalies in the data using some statistical graphs and other visualization techniques. Following things are part of EDA :

1. Get maximum insights from a data set
2. Uncover underlying structure
3. Extract important variables from the dataset
4. Detect outliers and anomalies(if any)
5. Test underlying assumptions
6. Determine the optimal factor settings

The main purpose of EDA is to detect any errors, outliers as well as to understand different patterns in the data. It allows Analysts to understand the data better before making any assumptions. The outcomes of EDA helps businesses to know their customers, expand their business and take decisions accordingly.

OVERVIEW OF THE DATA:

The Ames Housing Dataset is publicly available on Kaggle and is a great dataset to develop your skills. The data set also includes a data dictionary that provides a high-level description of each column.

https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

Exploratory Data Analysis (EDA)

An important step in any analysis is to define our target, or the variable we want to predict, in this case, the “sale price” column is the feature that I want to predict using the other features provided in the dataset.

EDA can reveal insights about our data that can be used to build a model as well as help the data scientist identify necessary data cleaning steps. As a first step, I imported the following libraries, which will be used to explore and visualize the data:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt%matplotlib inline
```

Upon importing inspecting a few rows, I can see that there are 81 columns of data that include numerical and object type columns and from looking at the values for some observations, categorical data as well.

```
# inspecting train file
train.head(3)
```

	Id	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Screen Porch	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold
0	109	533352170	60	RL	NaN	13517	Pave	NaN	IR1	Lvl	...	0	0	NaN	NaN	NaN	0	3	2010
1	544	531379050	60	RL	43.0	11492	Pave	NaN	IR1	Lvl	...	0	0	NaN	NaN	NaN	0	4	2009
2	153	535304180	20	RL	68.0	7922	Pave	NaN	Reg	Lvl	...	0	0	NaN	NaN	NaN	0	1	2010

3 rows × 81 columns

Modifying the column names:

I noted that column names are capitalized and contain spaces. I find it easier to analyze and perform subsequent modeling if feature names follow a certain format — lower cased and have spaces represented by an underscore.

Using the pandas rename method is a quick and simple way approach to do this, for example:

```
train.rename(columns={
    'Id':                                   'id',
    'PID':                                  'pid',
    'MS                 SubClass':          'ms_subclass',
}, inplace=True)
```

However, with so many columns in this data set, this method would be slow, inefficient and error prone. Fortunately, Python offers an efficient tool to fix this — the dictionary comprehension. For each column in my dataframe, the code below will replace spaces with an underscore and return lower case text.

```
train.columns = [i.replace(' ', '_').lower() for i in train.columns]
```

I can then inspect my columns and confirm that the change is as expected:

```
train.columns
```

```
Index(['id', 'pid', 'ms_subclass', 'ms_zoning', 'lot_frontage', 'lot_area',
       'street', 'alley', 'lot_shape', 'land_contour', 'utilities',
       'lot_config', 'land_slope', 'neighborhood', 'condition_1',
       'condition_2', 'bldg_type', 'house_style', 'overall_qual',
       'overall_cond', 'year_built', 'year_remod/add', 'roof_style',
       'roof_matl', 'exterior_1st', 'exterior_2nd', 'mas_vnr_type',
       'mas_vnr_area', 'exter_qual', 'exter_cond', 'foundation', 'bsmt_qual',
       'bsmt_cond', 'bsmt_exposure', 'bsmtfin_type_1', 'bsmtfin_sf_1',
       'bsmtfin_type_2', 'bsmtfin_sf_2', 'bsmt_unf_sf', 'total_bsmt_sf',
       'heating', 'heating_qc', 'central_air', 'electrical', '1st_flr_sf',
       '2nd_flr_sf', 'low_qual_fin_sf', 'gr_liv_area', 'bsmt_full_bath',
       'bsmt_half_bath', 'full_bath', 'half_bath', 'bedroom_abvgr',
       'kitchen_abvgr', 'kitchen_qual', 'totrms_abvgrd', 'functional',
       'fireplaces', 'fireplace_qu', 'garage_type', 'garage_yr_blt',
       'garage_finish', 'garage_cars', 'garage_area', 'garage_qual',
       'garage_cond', 'paved_drive', 'wood_deck_sf', 'open_porch_sf',
       'enclosed_porch', '3ssn_porch', 'screen_porch', 'pool_area', 'pool_qc',
       'fence', 'misc_feature', 'misc_val', 'mo_sold', 'yr_sold', 'sale_type',
       'saleprice'],
      dtype='object')
```

Efficient way to perform initial EDA

Next, I want to examine my data to inspect data types, identify null values, and inspect the summary statistics for each column available.

To achieve this, I will define a function, that takes my data as an input, and returns a data frame where each feature in my data set is now a row and the summary statistics are columns. The function will take a data frame as an input and calculate summary statistics to reveal insights about the data.

```
def ames_eda(df):
eda_df = {}
eda_df['null_sum'] = df.isnull().sum()
eda_df['null_pct'] = df.isnull().mean()
eda_df['dtypes'] = df.dtypes
eda_df['count'] = df.count()
eda_df['mean'] = df.mean()
eda_df['median'] = df.median()
eda_df['min'] = df.min()
eda_df['max'] = df.max()

  return pd.DataFrame(eda_df)ames_eda(train)
```

By passing my dataframe('train') into the function, I can get a quick and clean visual summary of my data, revealing summary statistics, data types and whether certain features have missing data and if so, what percentage of total data those values represent.

	null_sum	null_pct	dtypes	count	mean	median	min	max
1st_flr_sf	0	0.000000	int64	2051	1164.488055	1093.0	334	5095
2nd_flr_sf	0	0.000000	int64	2051	329.329108	0.0	0	1862
3ssn_porch	0	0.000000	int64	2051	2.591419	0.0	0	508
alley	1911	0.931741	object	140	NaN	NaN	NaN	NaN
bedroom_abvgr	0	0.000000	int64	2051	2.843491	3.0	0	8
...	...	...	...	...	...	...	...	...
utilities	0	0.000000	object	2051	NaN	NaN	AllPub	NoSewr
wood_deck_sf	0	0.000000	int64	2051	93.833740	0.0	0	1424
year_built	0	0.000000	int64	2051	1971.708922	1974.0	1872	2010
year_remod/add	0	0.000000	int64	2051	1984.190151	1993.0	1950	2010
yr_sold	0	0.000000	int64	2051	2007.775719	2008.0	2006	2010

81 rows × 8 columns

From the dataframe above, I can see that there are object and integer columns. I want to inspect what all my column types and evaluate if there are any implications for EDA:

```
train.dtypes.value_counts()
```

```
object     42
int64      28
float64    11
dtype: int64
```

This reveals that there are 42 object columns, 28 integer columns and 11 float columns in my dataset. Importantly, more than half my data consists of string or text values that will need to be explored and cleaned before modeling. As we will see below, just because the data does not have a numeric type out of the box does not mean it won't be important or useful to understand when modeling.

To see all the different object columns, I use the following code to return a list and inspect the data dictionary for a high level explanation of what these columns represent.

```
train.select_dtypes(include=['object']).columns
```

```
Index(['ms_zoning', 'street', 'alley', 'lot_shape', 'land_contour',
       'utilities', 'lot_config', 'land_slope', 'neighborhood', 'condition_1',
       'condition_2', 'bldg_type', 'house_style', 'roof_style', 'roof_matl',
       'exterior_1st', 'exterior_2nd', 'mas_vnr_type', 'exter_qual',
       'exter_cond', 'foundation', 'bsmt_qual', 'bsmt_cond', 'bsmt_exposure',
       'bsmtfin_type_1', 'bsmtfin_type_2', 'heating', 'heating_qc',
       'central_air', 'electrical', 'kitchen_qual', 'functional',
       'fireplace_qu', 'garage_type', 'garage_finish', 'garage_qual',
       'garage_cond', 'paved_drive', 'pool_qc', 'fence', 'misc_feature',
       'sale_type'],
      dtype='object')
```

Exploring the Object Columns:

By inspecting the data dictionary provided, I can see that many of these object columns, such as 'central_air' and 'heating_qc' are categorical or ordinal features that:

1. can be converted to numeric values through data cleaning, and
2. are intuitively related to the price of a house — a house with central air would logically have a higher sale price than one without, holding all else constant.

Exploring Relationships with our target:

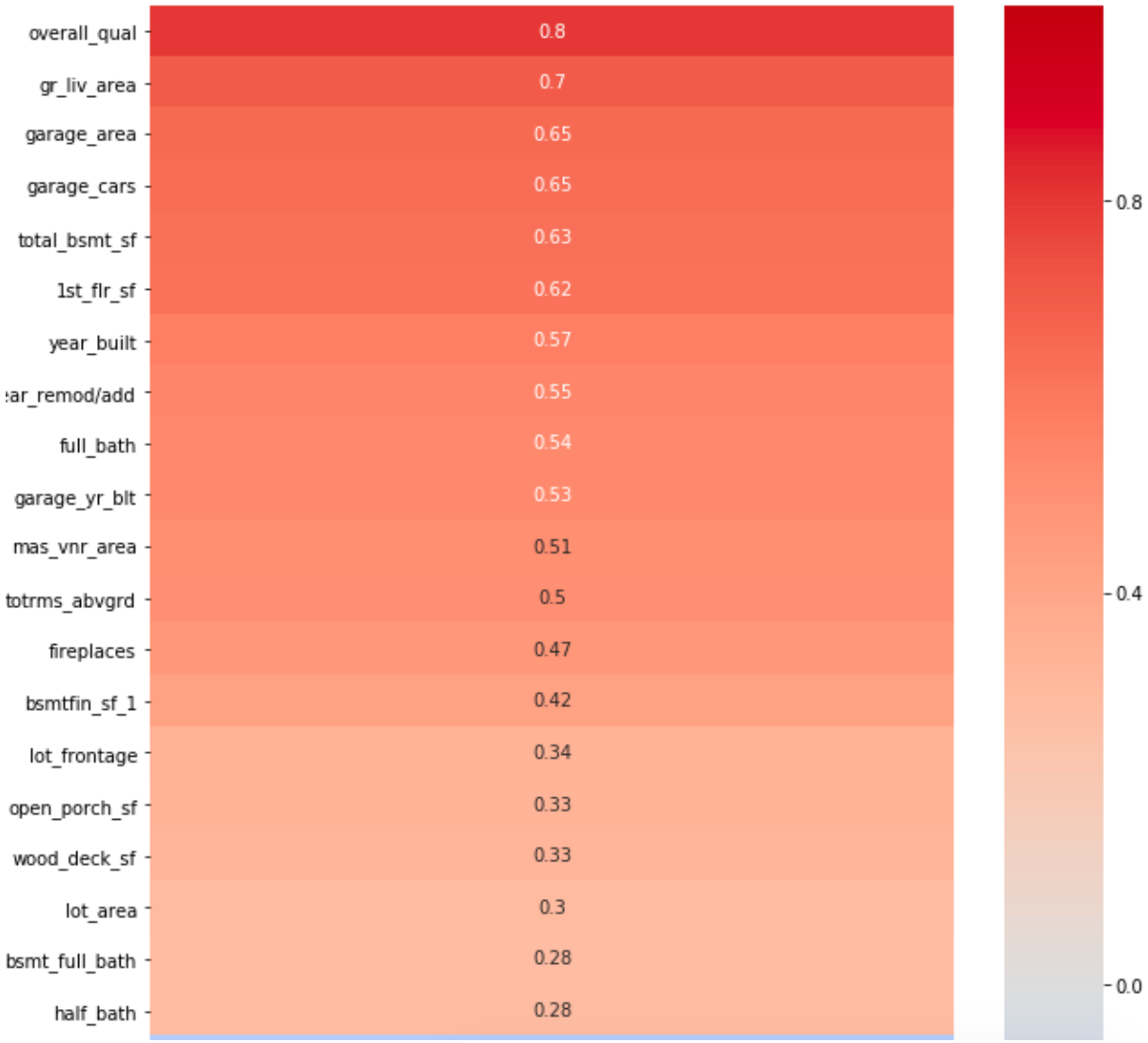
A key step in our EDA investment is to explore whether there is a relationship between our potential feature columns and our target, the home's sale price.

Given the different data types of our features, we need to take a different approach to visually exploring the relationships.

For example, using our numeric columns, we can calculate the correlations between potential features and our target, but correlations won't be calculated for non-numeric columns. This will be addressed further below.

Below, I've calculated the pairwise correlation between all of the numeric variables in the data frame and our target, sale price. Pandas' corrwith() method will return a pair-wise correlation for each numeric variable with the target and ignore non-numeric columns.

```
correlations = train.corrwith(train['saleprice']).iloc[:-1].to_frame()
correlations['abs'] = correlations[0].abs()
sorted_correlations = correlations.sort_values('abs', ascending=False)[0]fig, ax = plt.subplots(figsize=(10,20))
sns.heatmap(sorted_correlations.to_frame(), cmap='coolwarm', annot=True, vmin=-1, vmax=1, ax=ax);
```



For ease of analysis, this visualization sorts the pair-wise correlations by absolute value.

Using seaborn, we can visualize these pair-wise correlations.

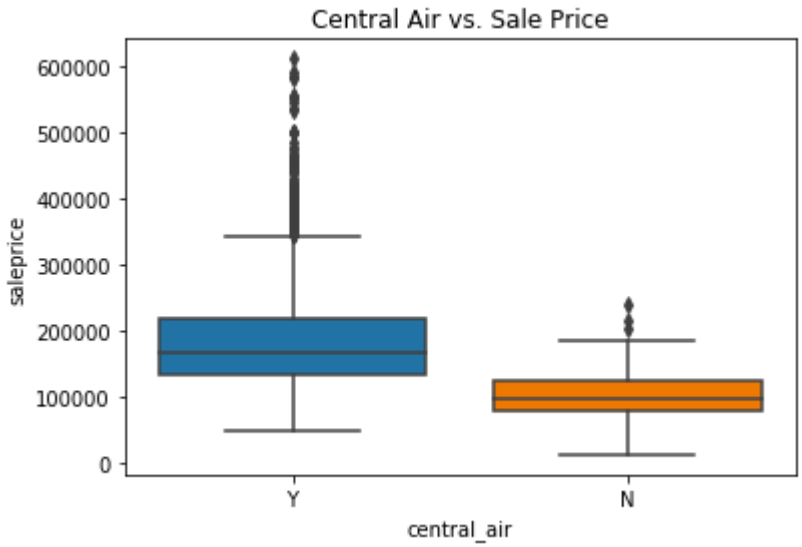
Unsurprisingly, features such as the overall quality of the home and the size of the living area have a strong relationship with the sale price.

But what about our non-numeric columns? Ordinal and categorical variables such as ‘exterior condition’ and ‘central air’ intuitively would have a relationship to sale price. It is critical that we visualize this!

A great way to achieve this is to generate a box plot that compares the values of an ordinal/categorical and visualize a relationship with sale price.

As we can see from the seaborn box plots below, there is a clear relationship between non-numeric values such the presence of central air or a home with “excellent” kitchen quality with sale price — relationships we want to capture in our model.

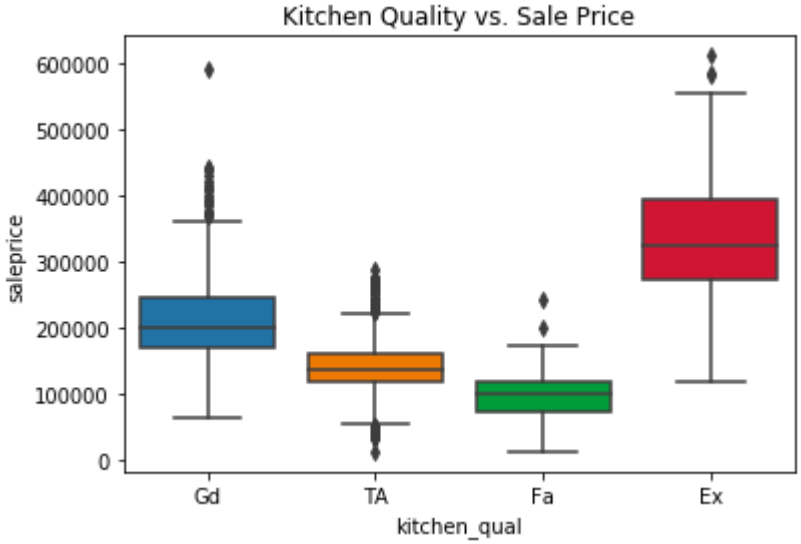
```
sns.boxplot(train['central_air'],
        train['saleprice']).set_title('Central Air vs. Sale Price');
```



Based on this visualization, we will need to convert these columns to represent numeric values when we clean our data before modeling.

We can also use the box plots to look at features that have categorical values. For example, kitchen quality is ranked on a scale from poor to excellent and again we can visualize a relationship to price:

```
sns.boxplot(train['kitchen_qual'],
            train['saleprice']).set_title('Kitchen Quality vs. Sale Price');
```



So we want to use the string columns, but how do we clean them?

One option to clean our categorical data is to define a function and apply it to our data such as in the example below to convert the garage quality from its categorical labels to numeric.

First, we want to identify the range of values that a certain feature may contain. Based on the values identified, we can create a function to overwrite each value with numerical values.

```
train['garage_qual'].value_counts()
```

```
TA    1832
Fa      82
Gd      18
Ex       3
Po       2
Name: garage_qual, dtype: int64
```

```
def garage_qual_cleaner(cell):
    if cell == 'Ex':
        return 5
    elif cell == 'Gd':
        return 4
    elif cell == 'TA':
        return 3
    elif cell == 'Fa':
        return 2
    elif cell == 'Po':
        return 1
    else:
        return 0
```

The function above would take a pandas series as an input and convert the string to a numeric value. We can then apply the function to the series of interest.

Alternatively, we could map a dictionary to overwrite values without creating a function.

```
train['kitchen_qual'].value_counts()
```

```
TA    1047
Gd     806
Ex     151
Fa      47
Name: kitchen_qual, dtype: int64
```

For example, given the values for kitchen quality we could use map a dictionary using the below code to convert the string values to a numeric data type:

```
train['kitchen_qual'].map({'Ex': 5, 'Gd': 4, 'TA': 3, 'Fa': 2, 'Po': 1})
```

However, given that there are 42 object columns that need cleaning, writing different sets of code to handle all these situations individually is inefficient.

Instead we can invest in defining one master function that can clean and organize the data:

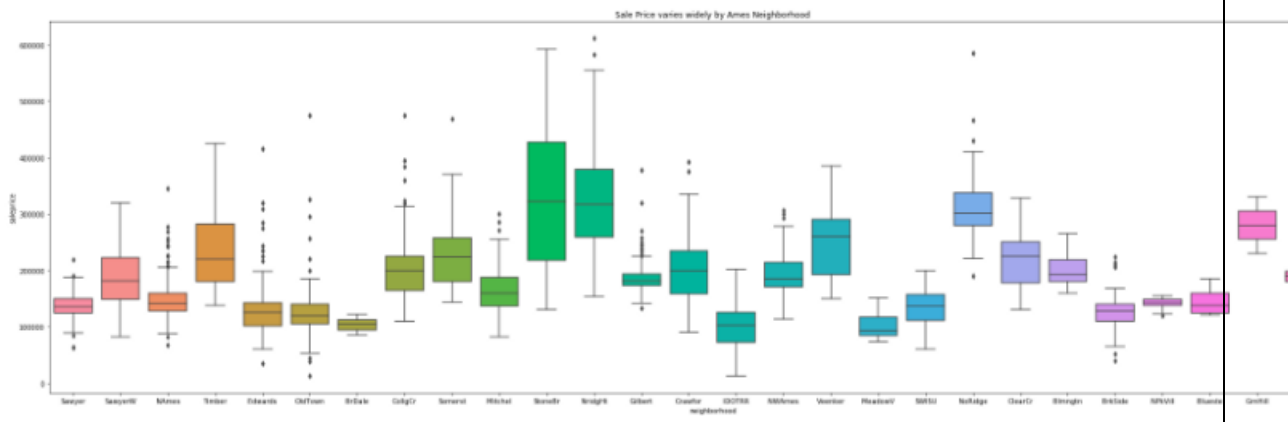
```
def data_cleaner(df):
  # map numeric values onto all the quality columns using a quality dictionary
qual_dict = {'Po':0,'Fa':1,'TA':2,'Gd':3,'Ex':4}
  # create a list of ordinal column names
ordinal_col_names = [col for col in df.columns if (col[-4:] in ['qual', 'cond']) and col[:3] != 'ove'] # last section ignores "overall quality columns which will be addressed below
  # creating a new feature called age
df['age'] = df.apply(lambda row: row['yr_sold'] - max(row['year_built'], row['year_remod/add']), axis=1)
  # dummify the date sold column
df['date_sold'] = df.apply(lambda row: str(row['mo_sold'])+ '-' + str(row['yr_sold']), axis=1)
df.loc[:,df.dtypes!= 'object'] = df.loc[:, df.dtypes != 'object'].apply(lambda col: col.fillna(col.mean()))

  # transforming columns
df[ordinal_col_names] = df[ordinal_col_names].applymap(lambda cell: 2 if pd.isnull(cell) else qual_dict[cell])

  return df# applying the function to train data
train = clean_data(train)
```

But what about variables without a clear, ordered relationship?

For example, the ‘neighborhood’ column contains string values detailing what neighborhood the home is located in. Without knowing the intimate details of the Ames real estate market, it is difficult to assign numeric values to the variable. We can’t simply assign random numbers or number them alphabetically as python will read neighborhood B (if we assign it as ‘2’) as being more valuable than neighborhood A (if we assign it as ‘1’), but less than neighborhood C — but this may not be the actual case and doing so would skew our model! So what do we do?

First, we want to visualize the relationship with our target:

```
plt.figure(figsize=(35,10)) # adjust the fig size to see everything
sns.boxplot(train['neighborhood'], train['saleprice']).set_title('Sale Price varies widely by Ames Neighborhood');
```



Some neighborhoods clearly have higher sale prices have than others — a relationship that we want to capture in our model.

Even string variables that do not have take on ordinal values such as neighborhood can be easily converted to a numerical amount by dummifying. Pandas provides a method to get dummify the variables — for each value (in this case neighborhood) a new feature will be created and the row will have a value of 0 or 1 for that column — a 1 signifying that in the original string column, a row contained the value that is now in the column name.

```
# dummifying neighborhood in the train dataset
pd.get_dummies(train, columns=['neighborhood'], drop_first=True)
```

	id	pid	ms_subclass	ms_zoning	lot_frontage	lot_area	street	alley	lot_shape	land_contour	...	neighborhood_NoRidge	neighborhood_Nridgh
0	109	533352170	60	RL	NaN	13517	Pave	NaN	IR1	Lvl	...	0	
1	544	531379050	60	RL	43.0	11492	Pave	NaN	IR1	Lvl	...	0	
2	153	535304180	20	RL	68.0	7922	Pave	NaN	Reg	Lvl	...	0	
3	318	916386060	60	RL	73.0	9802	Pave	NaN	Reg	Lvl	...	0	
4	255	906425045	50	RL	82.0	14235	Pave	NaN	IR1	Lvl	...	0	

As shown above, will use the pandas get dummies method to convert these to numeric values. It is important that we call the 'drop_first' argument and set it as 'True.' This will dummify all variables after dropping the first one. We do this because it is important that a categorical variable of K categories, or levels, usually enters a regression as a sequence of K-1 dummy variables and the dropped variable will serve as our reference category. If a row has a value of 0 for all categories, we know that that observation belonged to the dropped column.

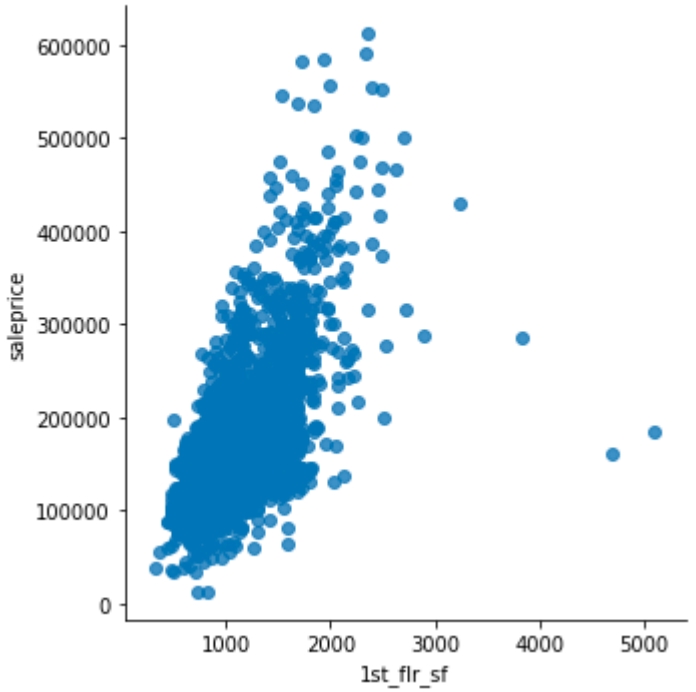
Now that we have cleaned and explored our data, we want to run a final check for outliers:

A quick way to check for outliers is to build a scatter plot between the target and a variable we would expect to be linearly related to our target. For example, it is not

unreasonable to assume a linear relationship between sale price and square feet (as the size of the home increases, it logically follows that the sale price would as well).

To inspect this, I created a seaborn scatter plot between the target and the first floor square footage in a home:

```
1 sns.lmplot(x='1st_flr_sf', y='saleprice', data=train, fit_reg=False);
```



We would expect to see a linear relationship between a variable such as first floor square footage and sale price (the larger the house, the higher the price). However, based on the scatterplot above, there are some outliers — large homes that have 4000 square feet on just one floor, but are relatively lower priced. We need to determine how best to handle these outliers to train our model.

Using pandas, I filtered the dataset below to inspect the outliers in greater detail:

```
1 train.loc[train['1st_flr_sf'] > 3800]
```

	id	pid	ms_subclass	ms_zoning	lot_frontage	lot_area	street	alley	lot_shape	land_contour	...	pool_qc	fence	misc_feature	misc_val
616	1498	908154080	20	RL	123.0	47007	Pave	NaN	IR1	Lvl	...	NaN	NaN	NaN	0
960	1499	908154235	60	RL	313.0	63887	Pave	NaN	IR3	Bnk	...	Gd	NaN	NaN	0
1885	2181	908154195	20	RL	128.0	39290	Pave	NaN	IR1	Bnk	...	NaN	NaN	Elev	17000

3 rows × 83 columns

Based on the filtered data, rows 616, 960 and 1885 are our outlier columns — I will drop these from the dataset so that they do not skew the model. Using the for loop

below, we can drop the outlier observations and then by re-examining our scatter plot, our data now appears less influenced by outliers.

```
rows_to_drop = [616, 960, 1885]for row in rows_to_drop: train.drop(row, inplace=True)
```

CONCLUSION:

As shown in this post, conducting exploratory data analysis and performing cleaning steps on the data can be an in depth process. But rather than viewing it as time consuming, it is important that we view this an important investment in the data science process. By investing our time to explore and clean our data, we have identified important outliers and extracted valuable information from string columns that will be critical to building a powerful predictive model.

APPLICATION:

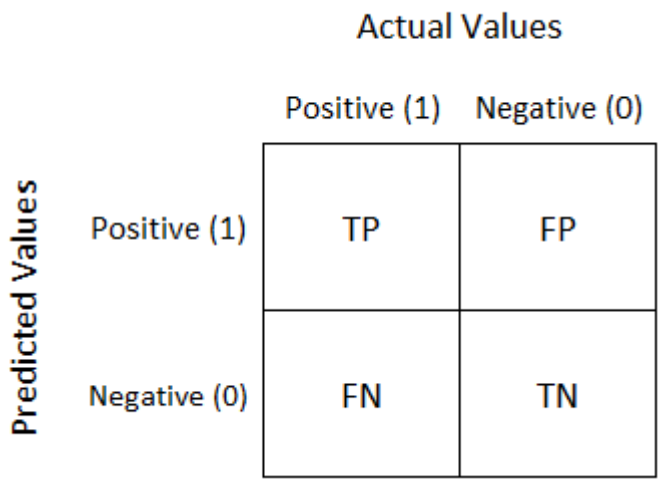
Exploratory spatial data analysis (ESDA) is a branch of EDA that is concerned specifically with geographical data. Those with training in this field can perform a variety of geographical tasks, such as visualizing spatial distributions, spotting physical outliers, and uncovering spatial clusters or patterns.

EXPERIMENT –10

CONFUSION MATRIX

AIM: Measuring a performance using confusion matrix.

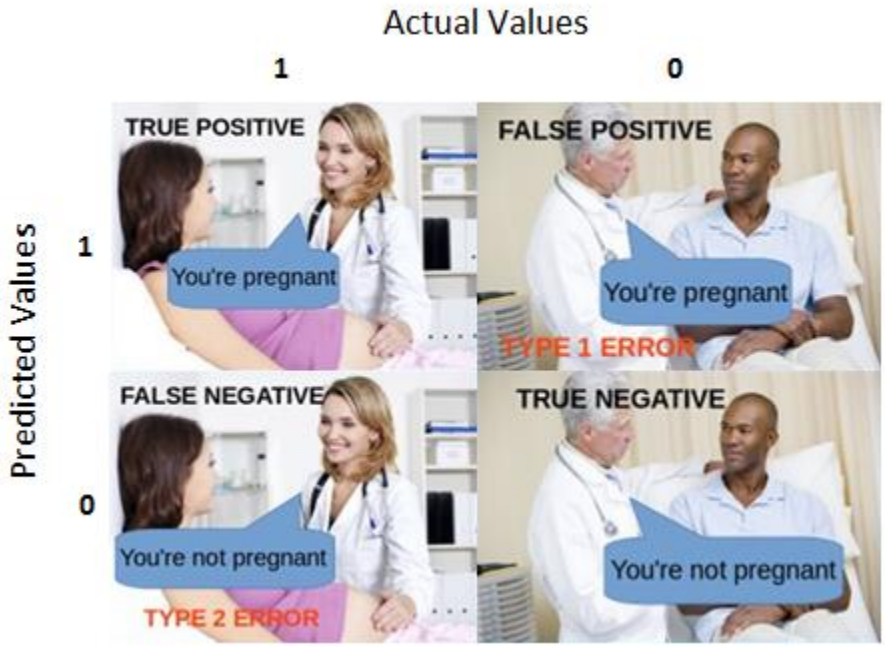
INTRODUCTION: It is a performance measurement for machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values.



Confusion Matrix

It is extremely useful for measuring Recall, Precision, Specificity, Accuracy, and most importantly AUC-ROC curves.

Let’s understand TP, FP, FN, TN in terms of pregnancy analogy.



Confusion Matrix

True Positive:

Interpretation: You predicted positive and it's true.

You predicted that a woman is pregnant and she actually is.

True Negative:

Interpretation: You predicted negative and it's true.

You predicted that a man is not pregnant and he actually is not.

False Positive: (Type 1 Error)

Interpretation: You predicted positive and it's false.

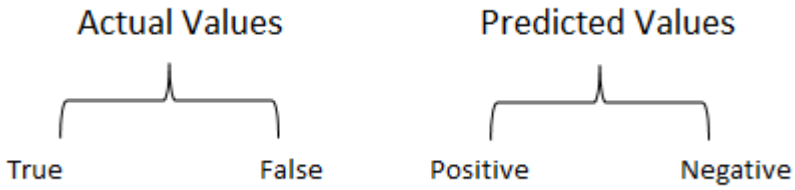
You predicted that a man is pregnant but he actually is not.

False Negative: (Type 2 Error)

Interpretation: You predicted negative and it's false.

You predicted that a woman is not pregnant but she actually is.

Just Remember, We describe predicted values as Positive and Negative and actual values as True and False.

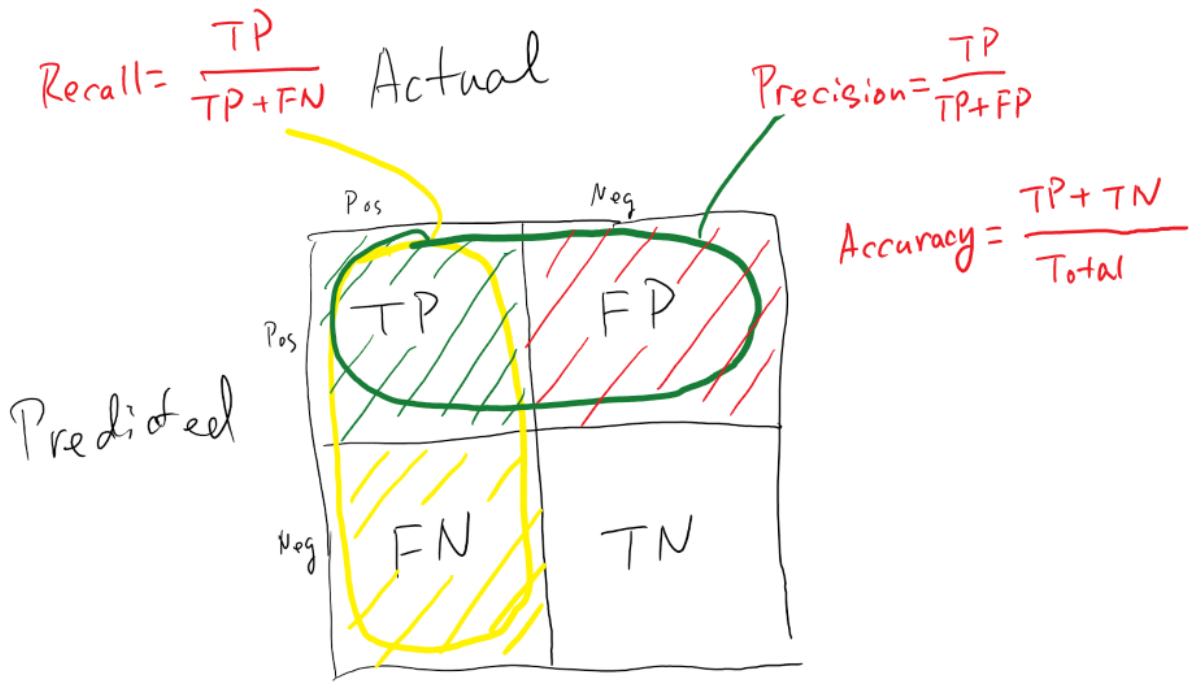


Actual vs Predicted values

How to Calculate Confusion Matrix for a 2-class classification problem?

Let's understand the confusion matrix through math.

y	y pred	output for threshold 0.6	Recall	Precision	Accuracy
0	0.5	0			
1	0.9	1			
0	0.7	1			
1	0.7	1	**1/2**	**2/3**	**4/7**
1	0.3	0			
0	0.4	0			
1	0.5	0			



Recall

$$\textbf{Recall} = \frac{TP}{TP + FN}$$

Recall [Image 7] (Image courtesy: My Photoshopped Collection)

The above equation can be explained by saying, from all the positive classes, how many we predicted correctly.

Recall should be high as possible.

Precision

$$\textbf{Precision} = \frac{TP}{TP + FP}$$

Precision [Image 8] (Image courtesy: My Photoshopped Collection)

The above equation can be explained by saying, from all the classes we have predicted as positive, how many are actually positive.

Precision should be high as possible.

and

Accuracy

From all the classes (positive and negative), how many of them we have predicted correctly. In this case, it will be 4/7.

Accuracy should be high as possible.

F-measure

$$F\text{ - measure} = \frac{2*\text{Recall}*\text{Precision}}{\text{Recall} + \text{Precision}}$$

F1 Score

It is difficult to compare two models with low precision and high recall or vice versa. So to make them comparable, we use F-Score. F-score helps to measure Recall and Precision at the same time. It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more.

EXAMPLE:

Data:

datasets.load_iris()

Code:

importnumpyasnp
importmatplotlib.pyplotasplt

fromsklearnimportsvm,datasets
fromsklearn.model_selectionimporttrain_test_split
fromsklearn.metricsimportConfusionMatrixDisplay

 # import some data to play with

```
iris=datasets.load_iris()
X=iris.data
y=iris.target
class_names=iris.target_names
```

 # Split the data into a training set and a test set

```
X_train,X_test,y_train,y_test=train_test_split(X,y,random_state=0)
```

 # Run classifier, using a model that is too regularized (C too low) to see

 # the impact on the results

```
classifier=svm.SVC(kernel="linear",C=0.01).fit(X_train,y_train)

np.set_printoptions(precision=2)
```

 # Plot non-normalized confusion matrix

```
titles_options=[
("Confusion matrix, without normalization",None),
("Normalized confusion matrix","true"),
]
fortitle,normalizeintitles_options:
disp=ConfusionMatrixDisplay.from_estimator(
classifier,
X_test,
y_test,
display_labels=class_names,
cmap=plt.cm.Blues,
normalize=normalize,
)
disp.ax_.set_title(title)

print(title)
print(disp.confusion_matrix)

plt.show()
```

CONCLUSION:

A confusion matrix is a remarkable approach for evaluating a classification model. It provides accurate insight into how correctly the model has classified the classes depending upon the data fed or how the classes are misclassified.

Talking about the measuring parameters, among precision, recall, accuracy and f-measure, it can be seen that precision and recall are immensely deployed parameters since their tradeoff relationship is a pragmatic measure for the achievement of

prediction. Though the necessary model is presumed to have high precision and high recall, applicable in an ideally separable data.

EXPERIMENT -11

DECISION TREE

AIM: Decision Tree Classification Using Python.

INTRODUCTION:

A Decision Tree is a supervised Machine learning algorithm. It is used in both classification and regression algorithms. The decision tree is like a tree with nodes. The branches depend on a number of factors. It splits data into branches like these till it achieves a threshold value. A decision tree consists of the root nodes, children nodes, and leaf nodes.

How Does the Decision Tree Algorithm works?

The basic idea behind any decision tree algorithm is as follows:

1. Select the best Feature using Attribute Selection Measures(ASM) to split the records.

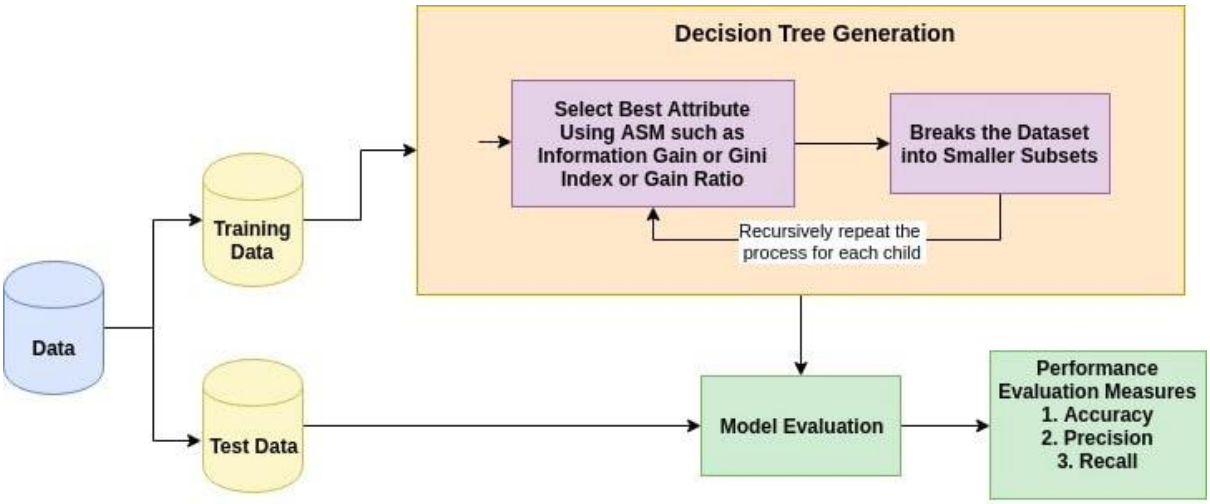
2. Make that attribute/feature a decision node and break the dataset into smaller subsets.

3 Start the tree-building process by repeating this process recursively for each child until one of the following condition is being achieved :

a) All tuples belonging to the same attribute value.

b) There are no more of the attributes remaining.

c) There are no more instances remaining.



HOW TO BUILD DECISION TREES FROM SCRATCH?

While building a Decision tree, the main thing is to select the best attribute from the total features list of the dataset for the root node as well as for sub-nodes. The selection of best attributes is being achieved with the help of a technique known as the Attribute selection measure (ASM).

With the help of ASM, we can easily select the best features for the respective nodes of the decision tree.

There are two techniques for ASM:

a) Information Gain

b) Gini Index

a) Information Gain:

1. Information gain is the measurement of changes in entropy value after the splitting/segmentation of the dataset based on an attribute.

2. It tells how much information a feature/attribute provides us.

3. Following the value of the information gain, splitting of the node and decision tree building is being done.

4. decision tree always tries to maximize the value of the information gain, and a node/attribute having the highest value of the information gain is being split first. Information gain can be calculated using the below formula:

Information Gain= Entropy(S)- [(Weighted Avg) *Entropy(each feature)

Entropy: Entropy signifies the randomness in the dataset. It is being defined as a metric to measure impurity. Entropy can be calculated as:

Entropy(s)= -P(yes)log2 P(yes)- P(no) log2 P(no)

Where,

S= Total number of samples

P(yes)= probability of yes

P(no)= probability of no.

b) Gini Index:

Gini index is also being defined as a measure of impurity/ purity used while creating a decision tree in the CART(known as Classification and Regression Tree) algorithm.

An attribute having a low Gini index value should be preferred in contrast to the high Gini index value.

It only creates binary splits, and the CART algorithm uses the Gini index to create binary splits.

Gini index can be calculated using the below formula:

Gini Index= 1- ∑jPj2

Where pj stands for the probability.

PYTHON CODE IMPLEMENTATION:

#Numerical computing libraries

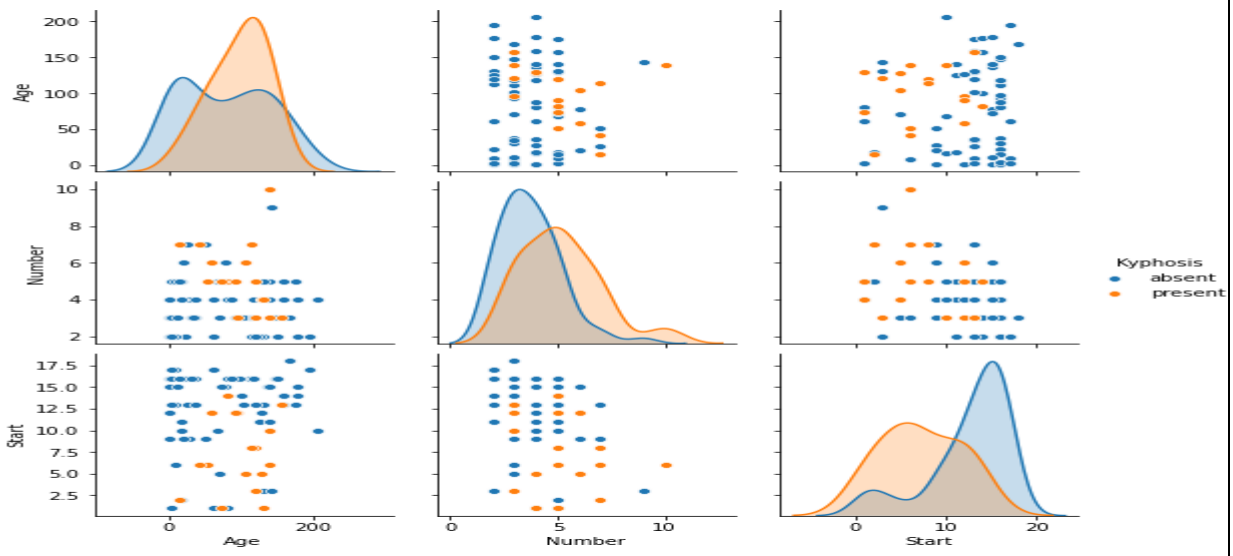
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

#Loading Data

```
raw_data = pd.read_csv('kyphosis.csv')
raw_data.columns
Index(['Kyphosis', 'Age', 'Number', 'Start'], dtype='object')
```

#Exploratory data analysis

```
raw_data.info()
sns.pairplot(raw_data, hue = 'Kyphosis')
```



#Split the data set into training data and test data

```
from sklearn.model_selection import train_test_split
x = raw_data.drop('Kyphosis', axis = 1)
y = raw_data['Kyphosis']
x_training_data, x_test_data, y_training_data, y_test_data = train_test_split(x, y, test_size = 0.3)
```

#Train the decision tree model

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(x_training_data, y_training_data)
predictions = model.predict(x_test_data)
```

#Split the data set into training data and test data

```
from sklearn.model_selection import train_test_split
x = raw_data.drop('Kyphosis', axis = 1)
y = raw_data['Kyphosis']
x_training_data, x_test_data, y_training_data, y_test_data = train_test_split(x, y, test_size = 0.3)
```

#Train the decision tree model

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(x_training_data, y_training_data)
predictions = model.predict(x_test_data)
```

APLLICATION AND CONCLUSION:

The application of Decision Tree mining tool on a real-world safety records perfectly reveals useful information that are subsumed in the volume and nature of the data. The concept of Entropy and Information Gain theory are used in building the decision tree model and an accuracy of 71.4% arrived at, indicating a good performance of decision tree induction on the dataset.

#TASK:

Use pima-indians-diabetes dataset.

It is well described in the following address : https://www.kaggle.com/kumargh/pimaindiansdiabetescsv

1. Import the necessary python modules :numpy, pandas and matplotlib

2. Load the dataset. It is available at the address :https://www.labri.fr/~zemmari/datasets/pima-indians-diabetes.csv

3. In the sequel, we will not consider entire the dataset. Write instructions to extract the columns pregnant, insulin, bmi, age, glucose, bp, and pedigree.

4. Split the dataset into two subsets : one for training and the other for testing. Save 30% of the dataset for test.

5. Using the python module sklearn.tree, train a model using DecisionTreeClassifier.
6. Give the confusion matrix and evaluate the model.

7. You can visualise the trained tree. For this purpose, you can execute the following instructions :

```
from sklearn.tree import export_graphviz

from sklearn.externals.six import StringIO

from IPython.display import Image

import pydotplus

dot_data = StringIO()

export_graphviz(dt, out_file=dot_data,

                filled=True, rounded=True,

special_characters=True,feature_names = feature_cols,class_names=['0','1']) graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
```

```
graph.write_png('diabetes.png')

Image(graph.create_png())
```

CONCLUSION:

In this lab, you covered details about Decision Tree; It's working, attribute selection measures such as Information Gain, and Gini Index, decision tree model building, visualization and evaluation.

EXPERIMENT-12

TIME SERIES

AIM: Time Series Analysis in Python.

INTRODUCTION: Time series is a sequence of observations recorded at regular time intervals. Depending on the frequency of observations, a time series may typically be hourly, daily, weekly, monthly, quarterly and annual. Sometimes, you might have seconds and minute-wise time series as well, like, number of clicks and user visits every minute etc

1. It is time dependent. So the basic assumption of a linear regression model that the observations are independent doesn't hold in this case.
2. Along with an increasing or decreasing trend, most TS have some form of seasonality trends, i.e. variations specific to a particular time frame. For example, if you see the sales of a woolen jacket over time, you will invariably find higher sales in winter seasons.

The sources of time series data are periodic measurements or observations. Just to give a few examples:

- Stock prices over time
- Daily, weekly, monthly sales
- Periodic measurements in a process
- Power or gas consumption rates over time

EXAMPLE:

We will be using pandas for data analysis and manipulation and matplotlib to create visualizations. Let's start by importing these libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

There are various ways to obtain stock price data. A simple one is the pandas datareader module. It creates a dataframe that contains 6 different pieces of information about a stock during a given period.

The following syntax creates two dataframes that contain the stock price data of Google and Apple from January, 2019 to December, 2020

```
from pandas_datareader import dataapple = data.DataReader("AAPL", start='2019-1-1', end='2020-12-1',
```

```
data_source='yahoo')google = data.DataReader("GOOG", start='2019-1-1', end='2020-12-1', data_source='yahoo')
```

We just need to pass the stock name, start and end dates, and the data source to the data function. Here are the first 5 rows of the returned dataframe.

```
apple.head()
```

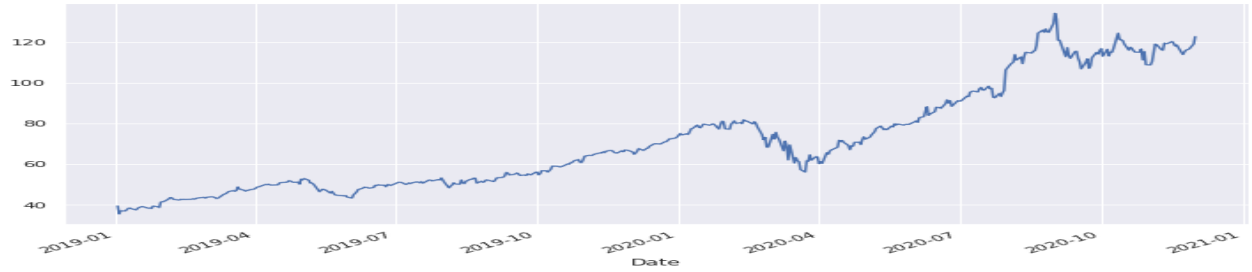
Date	High	Low	Open	Close	Volume	Adj Close
2019-01-02	39.712502	38.557499	38.722500	39.480000	148158800.0	38.249401
2019-01-03	36.430000	35.500000	35.994999	35.547501	365248800.0	34.439476
2019-01-04	37.137501	35.950001	36.132500	37.064999	234428400.0	35.909672
2019-01-07	37.207500	36.474998	37.174999	36.982498	219111200.0	35.829746
2019-01-08	37.955002	37.130001	37.389999	37.687500	164101200.0	36.512772

The most fundamental tool used in time series analysis is line plot. It shows how prices change over time.

Note: When working with time series, it is convenient to keep the dates or times as index. It makes both the analysis and creating plots easier.

Let’s create a simple line plot of the closing price of Apple stock.

```
plt.figure(figsize=(12,6))
apple['Close'].plot()
```

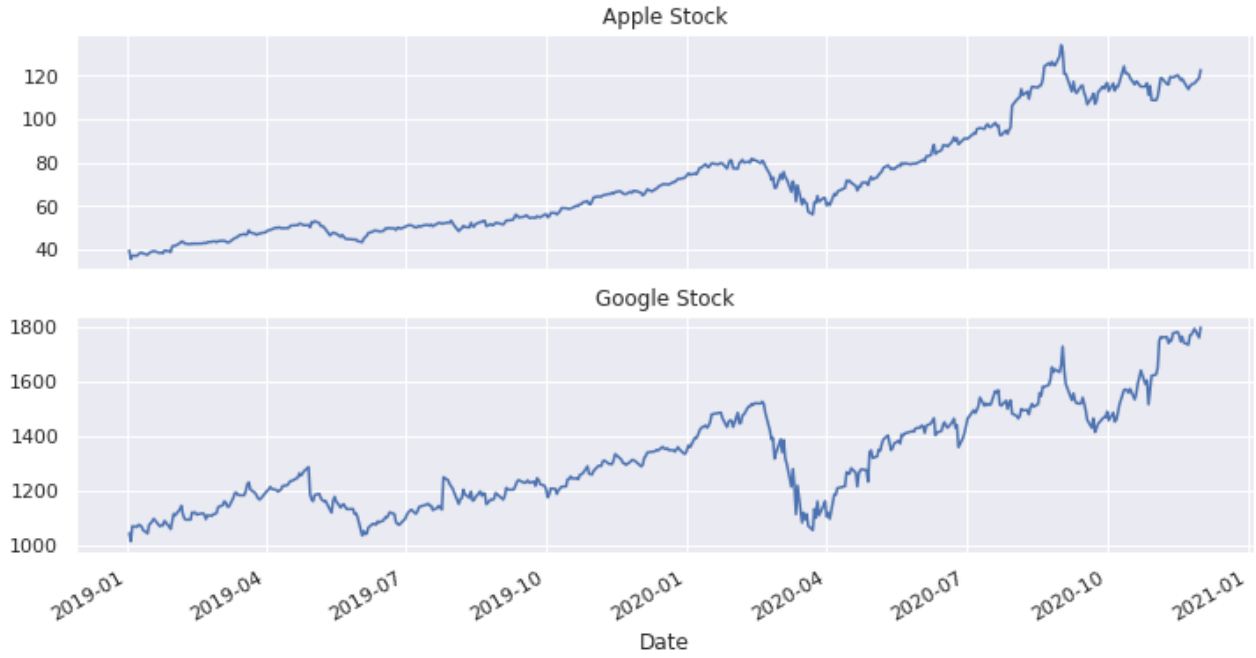


We observe an increasing trend with a few exceptions. The biggest downward movement occurs around April 2020 which is probably due to the global corona virus pandemic.

We can compare the stock prices of Google and Apple by plotting them on the same figure.

The subplots function of matplotlib can be used as follows:

```
fig, ax = plt.subplots(nrow=2, sharex=True, figsize=(12,6))
apple['Close'].plot(ax=ax[0], title="Apple Stock", legend=False)
google['Close'].plot(ax=ax[1], title="Google Stock", legend=False)
```



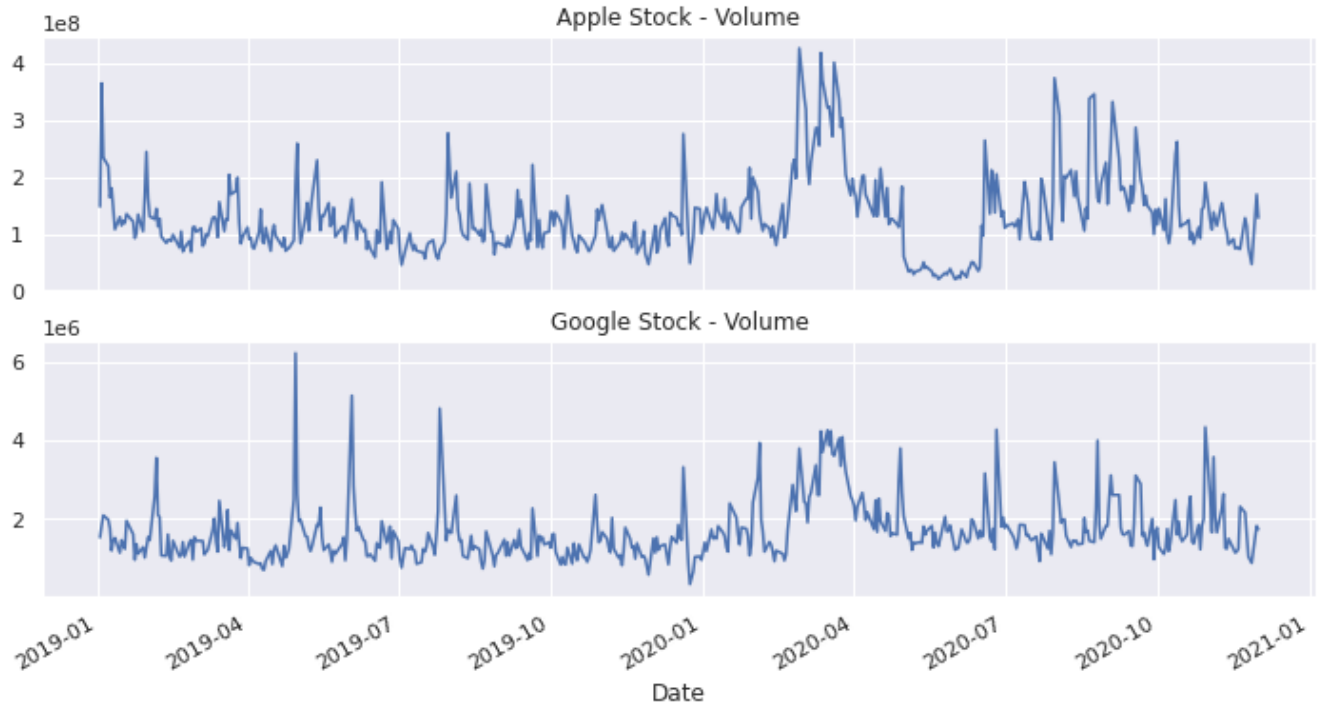
The subplot function creates a grid of axes objects on a figure. The number of axes objects and their alignment are determined by the nrows and ncols parameters.

After creating the figure and axes, we specify the position of each plot in the grid by using the ax parameter.

From the visualization above, we can conclude that the stock prices of Google and Apple follow similar trends during the given period.

Let's also compare these two stocks in terms of the daily volumes. All we need to do is to change the column name to volume.

```
fig, ax = plt.subplots(nrows=2, sharex=True, figsize=(12,6))
apple['Volume'].plot(ax=ax[0], title="Apple Stock - Volume", legend=False)
google['Volume'].plot(ax=ax[1], title="Google Stock - Volume", legend=False)
```

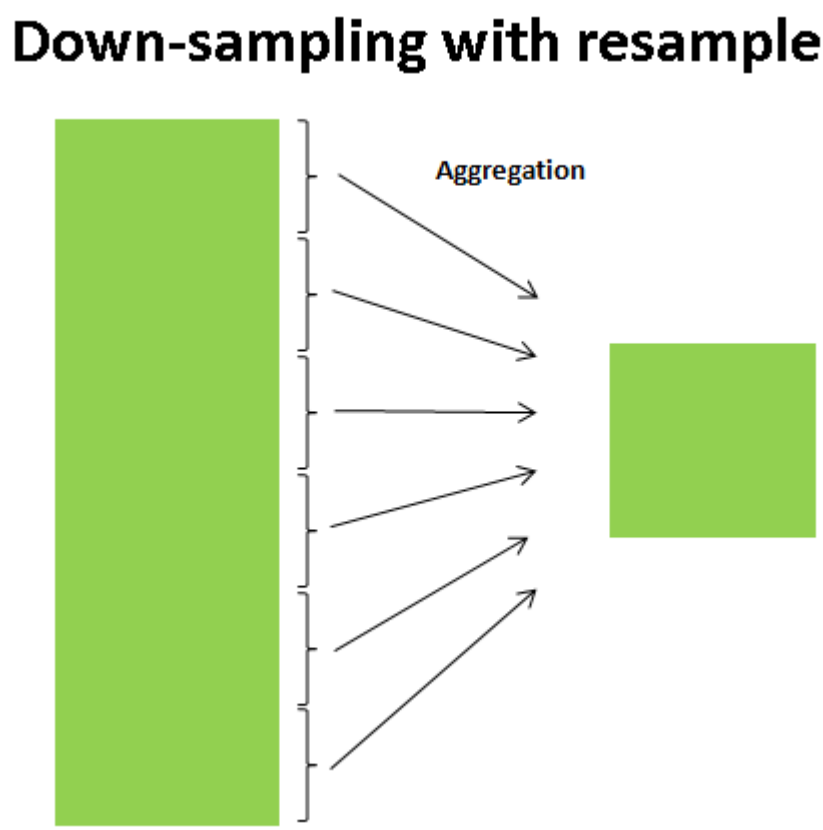


It is hard to compare the volumes based on daily frequencies. In this case, we can resample the time series data.

Resampling basically means representing the data with a different frequency. If we increase the frequency, it is called up-sampling. The opposite is down-sampling which means decreasing the frequency.

In our case, we will down-sample the data to get a better overview for comparison. Pandas provides two methods for resampling which are the resample and asfreq functions.

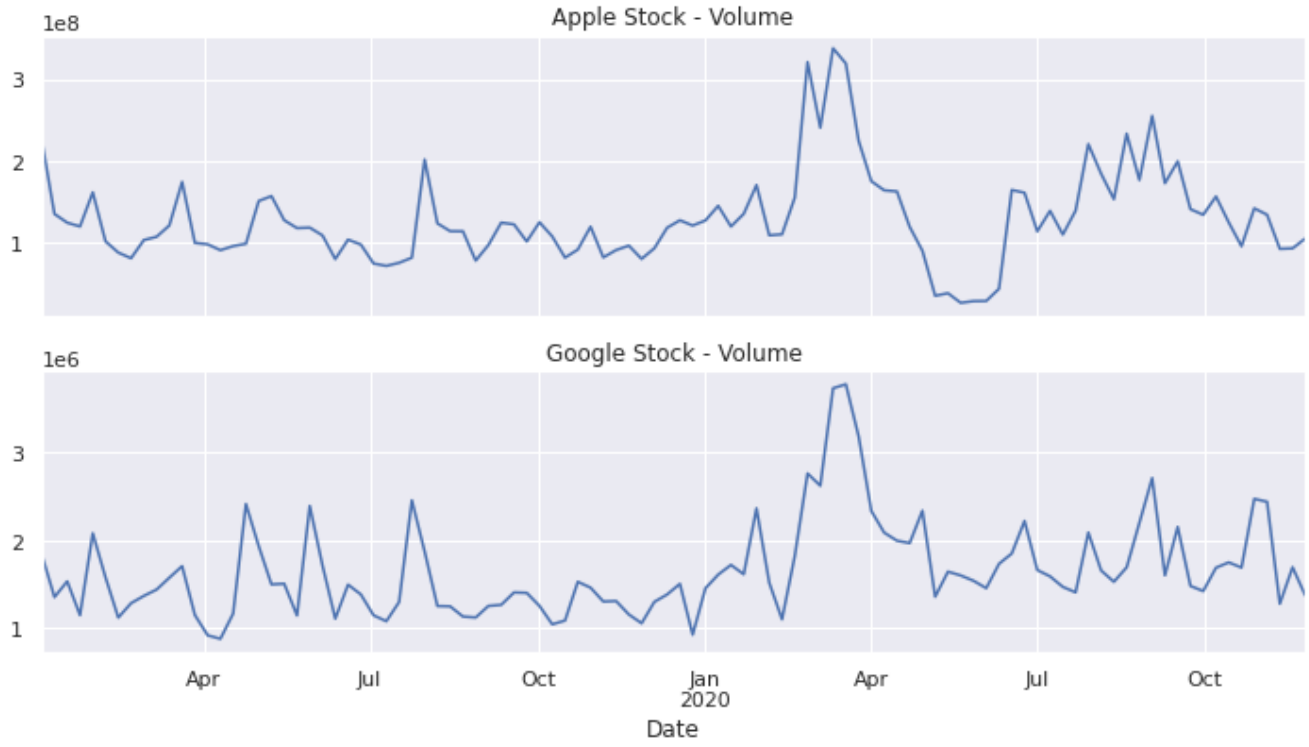
- Resample: Aggregates data based on specified frequency and aggregation function.
- Asfreq: Selects data based on the specified frequency and returns the value at the end of the specified interval.



I will use the resample function to down-sample the volume data to 7-day periods.

After selecting the data to be plotted, we will call the resample function along with the aggregation (mean in our case). The resampled data will be plotted instead of the original data.

```
fig, ax = plt.subplots(nrows=2, sharex=True, figsize=(12,6))apple['Volume'].resample('7D').mean()\
.plot(ax=ax[0], title="Apple Stock - Volume", legend=False)google['Volume'].resample('7D').mean()\
.plot(ax=ax[1], title="Google Stock - Volume", legend=False)
```



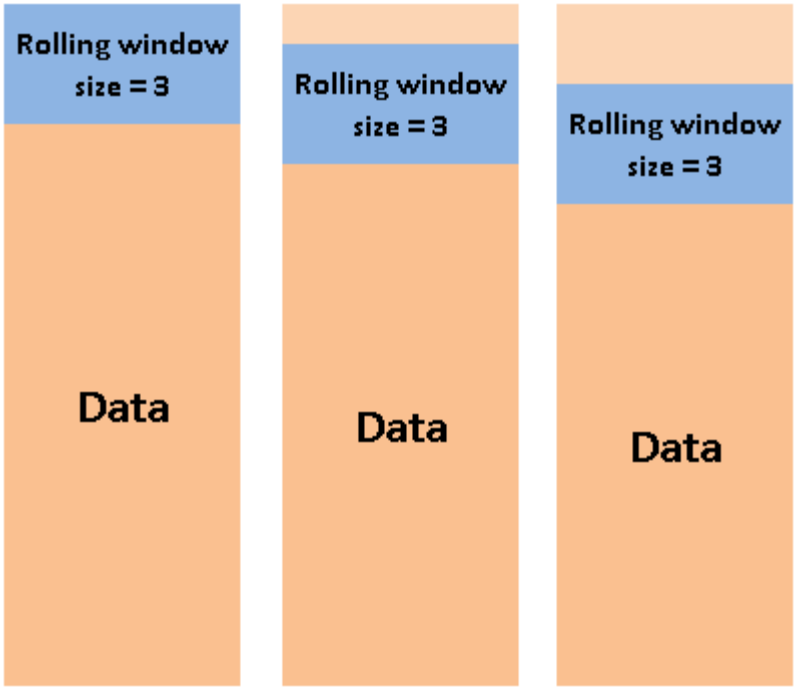
Down-sampled to 7-day periods

The resample function is quite flexible in terms of defining the frequency. You can even pass a string to specify the desired frequency.

The down-sampled data provides a more clear picture for comparison. The trend in volume for Google and Apple stocks seem be to similar with some exceptions.

Another useful and commonly used operation on time series data is rolling. The idea is based on creating a window of specific size that rolls through the data. While the window is rolling, some kind of calculations are done on the data inside the window.

The figure below explains the concept of rolling.



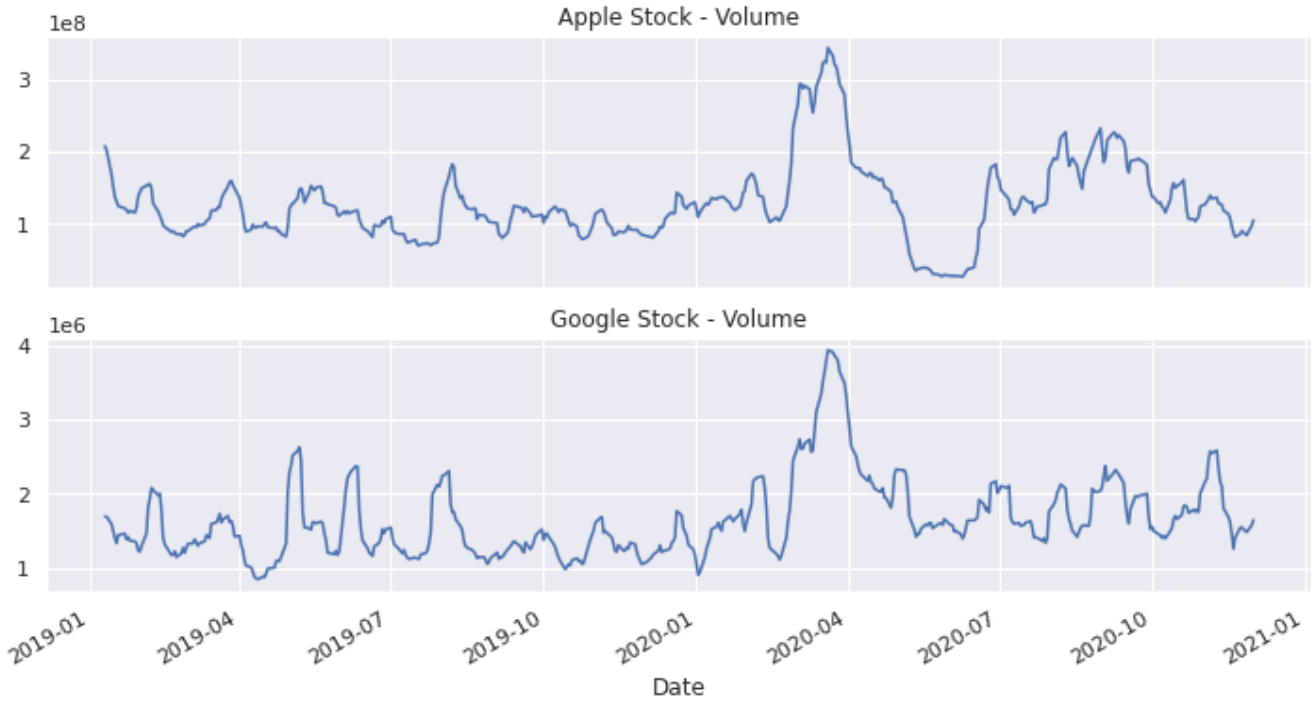
Rolling

It is important to note that the calculation starts when the whole window is in the data. In other words, if the size of the window is three, the first aggregation is done at the third row.

Rolling is similar to down-sampling in some sense. In both techniques, we divide the data into smaller chunks and do some kind of aggregation. The difference is that each data point is used once in down-sampling. When we do rolling, each data point is used as many times as the size of the window (except for the first ones).

Let’s create the volume plots using a rolling window of size 7. We can then compare it to the previous one created by down-sampling to 7-day periods.

```
fig, ax = plt.subplots(nrows=2, sharex=True, figsize=(12,6))apple['Volume'].rolling(7).mean()\
.plot(ax=ax[0], title="Apple Stock - Volume", legend=False)google['Volume'].rolling(7).mean()\
.plot(ax=ax[1], title="Google Stock - Volume", legend=False)
```



Rolling window of size 7

The only difference in syntax is that we have used the rolling function with 7, instead of using the resample function with '7D'.

The plots are quite similar as expected. We can easily observe the same trend. However, the plot created by rolling is not as smooth as the one created by down-sampling. In that sense, rolling carries more detail than down-sampling.

CONCLUSION:

Predictive analytics is highly valuable in the data science field and time series data is at the core of many problems that predictive analytics aims to solve.

There are many tools to work with time series data. Pandas is one of the highly efficient and common ones. Hence, if you plan to do time series analysis, I suggest to get familiar with Pandas.

What we have covered in this article can be considered as the basics of time series analysis. Once you are comfortable with the basics, you can easily built up your knowledge.

APPLICATION:

Industries in all sectors generate and use time series data to make important business decisions. We understand following quite accurately using time series:

Understanding the past ,Forecasting into the future , Anomaly, detection ,Comparative analysis , Correlation analysis , Econometric analysis.