Missing Data

(1) Importing libraries.

** Python :

```
import numpy as np ; (Mathematical tools - mathematics)
import matplotlib.pyplot as plt
        ↳ plot charts in python.
import pandas as pd
        ↳ import & manage datasets.
```

** R :

We don't need to import libraries. By default some usual libraries are imported.

(2) Importing Dataset :

** Python :

(i) Goto file explorer → select machine learning AZ → section 2 - part 1 → dataset.csv (location).

(ii) Now save your python file in that location.

~~(iii) It will become your home directory.~~

(iii) Now click "Run" and it will become your working directory.

```
import numpy as np &
import matplotlib.pyplot as plt
import pandas as pd
# importing dataset.
dataset = pd.read_csv('name.csv')
```

[ctrl + enter - to execute] [format : ".0f"]

* In python you need to distinguish dependent and independent variables.

* In given dataset:

Country, Age, Salary - Independent Variables.

Purchased - Dependent Variable.

importing dataset -

```
dataset = pd.read_csv('name.csv')
X = dataset.iloc[:, :-1].values.
y = dataset.iloc[:, 3].values.
```

(: → rows; 3 → 3rd column.)

** R:

~~# importin~~

* In R - to set the directory as working directory, same steps as python except instead of 'Run' click more → Set as working directory.

* - here we don't need to distinguish variables.

importing dataset.

```
dataset = read.csv('name.csv')
```

(independent variables → Matrix of features).

Missing Values Dealing :

**Python :

importing library that takes care of missing data.

from sklearn.preprocessing import Imputer.

↳ Sykitlearn ~~class~~ → preprocessing library → imputer class.
Contains libraries ↳ contains class & methods ↳ deals with missing data.

create object of imputer class. [Ctr + I — for help].

imputer = Imputer(missing_values = 'NaN', Strategy = 'mean', axis = 0)

↳ impute column, 1 → row

fit the values in the matrix.

imputer = imputer.fit(X[:, 1:3])

↳ upperbound excluded.

Now you need to transform the dataset with impute values.

X[:, 1:3] = imputer.transform(X[:, 1:3])

**R :

Taking care of missing data.

dataset$Age = ifelse(is.na(dataset$Age), ave(dataset$Age, FUN = function(x) mean(x, is.missing)), dataset$Age)

↓ detecting Age column in dataset.

[ifelse(condition, if true, else)].

Categorical Data

In the given dataset there are 2 categorical variables.

Country — Age — Salary — Purchased.

↓

France, Spain, Germany } 3 Categories

Yes, No } 2 Categories.

In machine learning since we do mathematic calculations, to keep text is a problem. So we need to encode them into numbers, so here Country & Purchased are encoded into numbers.

** Python:

```
# importing the library that encodes.
from sklearn.preprocessing import LabelEncoder
                                  ↳ class.
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
```

France - 0

Spain - 2

Germany - 1

* Here we have a problem. Since Machine Learning is based on equations. It will think that Germany has higher value than France because (1 > 0). But that is not right because those are categories.

→ To prevent the System from thinking like that we use dummy encoding.

→ to do that encoding we use OneHotEncoder class

```
from sklearn.preprocessing import LabelEncoder,
                                  OneHotEncoder
onehotencoder = OneHotEncoder(categorical_features=[0])
X = onehotencoder.fit_transform(X).toarray()
```

** R:

```
dataset$country = factor(dataset$country,
                         levels = c('France', 'Spain', 'Germany'),
                         labels = c(1, 2, 3))
```

Splitting the data set into Trainingset & Testset

```
from sklearn.cross_validation import
                    train_test_split.
X_train, X_test, y_train, y_test = train_test_split(
                    X, y, test_size = 0.2, random_state = 0)
```

↳ for same results as video.

** Python

R:

#Install.packages('caTools') → install caTools package.

library(caTools) → To select caTools library

* In python we used RandomState to get same results, Here in R we use seed()

set.seed(123)

split = sample.split(dataset$Purchased, SplitRatio = 0.8)

Ctrl+I

→ Training set, but in python we do it for test set.

* In python we split train_set & test_set for dependable and non-dependable variables seperately.
* But in R we only do it for dependable variables.

trainingset = subset(dataset, split == TRUE)

test_set = subset(dataset, split == FALSE)

Feature Scaling:

* In the given dataset Age ranges from 0-50 and Salary from 0-100k.

∴ Age & Salary are not in same scale.

When calculating Euclidean distance Age doesn't affect because it has small values, so machine learning will neglect that.

So to avoid that we do feature scaling, So that both are in same scale.

→ We do feature scaling in two ways

(i) Standardisation — $x_{stand} = \frac{x - mean(x)}{Standard\ deviation(x)}$

(ii) Normalization

— $x_{Norm} = \frac{x - min(x)}{max(x) - min(x)}$

Python:

```
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler();
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.transform(x_test)
```

R:

In R, to do scaling.

```
training_set = scale(training_set)
test_set = scale(test_set)
```

But by running this we get error i.e x is not numeric.

But when we see training_set we see that all are in numerics.

The problem we have done factoring in R, and factoring means not dealing in numerics.

So here we leave the two columns that are factored and scale the remaining ones.

Data Preprocessing template :

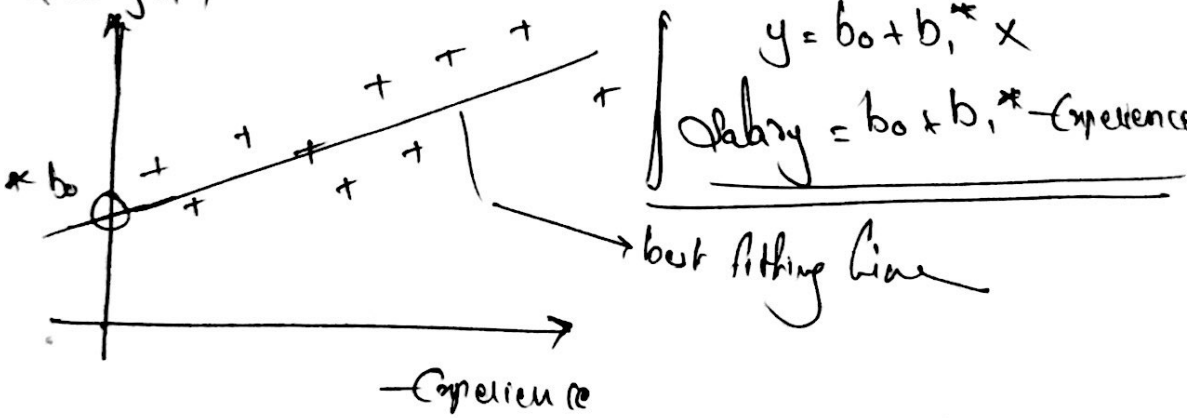
ection - 2

Regression - Simple - Linear :

constant → b_0 + b_1 * x_1 ← coefficient — slope formula

y ← dependent variable

x ← Independent Variable .



b_0 - means point where line crosses vertical axis.

b_1 - slope of line .

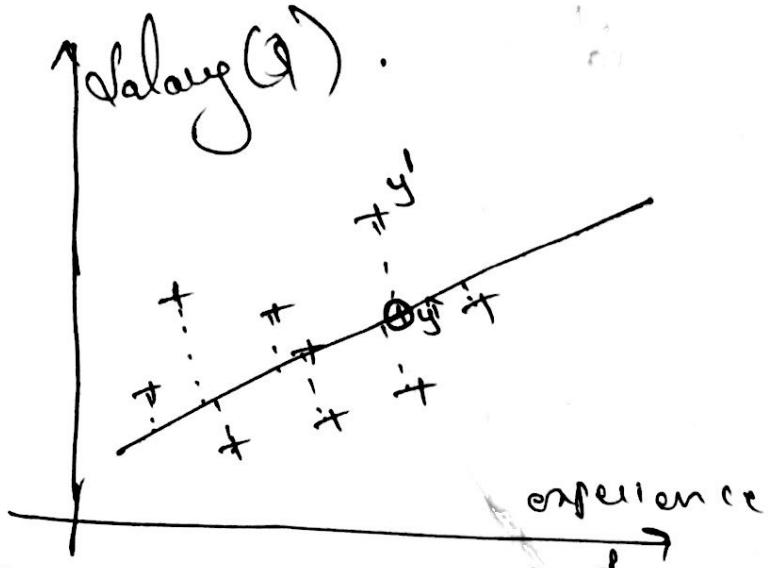
If slope is more, then - for more experience more salary.

→ Simple linear Regression is used to find out the best fitting line .

→ Ways of formulae

$y = b_0 + b_1 x$

$y = b_1 x$ *



The best fitting line is Sum of squared of differences between $(y - \hat{y})$

i.e $\text{sum}(y - \hat{y})^2 \longrightarrow \text{Min}$.

Python :

```
import numpy as np
np.set_printoptions(threshold = np.nan)
import matplotlib.pyplot as plt
import pandas as pd

#importing the dataset
dataset = pd.read_csv('Salary-Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].value
# splitting into test & training sets.
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split
                    (X, y, test_size = 1/3, random_stat
                    = 0)
```

* In Simple-Linear Regression, We don't need to do the Feature Scaling, the library does take care of it automatically.

Fitting Simple Linear Regression in train set.

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(x_train, y_train)
```

Predicting the test set results.

```
y_pred = regressor.predict(x_test)
```

Visualizing the training set results.

To plot the graphs we use "matplotlib" library.

```
plt.scatter(x_train, y_train, color='red')
plt.plot(x_train, regressor.predict(x_train), color='blue')
plt.title('Salary vs Experience (Training set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

* We don't change x_train in plt.plot of test set because we want to predict y_test values on the regressor line formed on training set
* If we change to x_test it will form a new regressor line

```
#visualising the test set results
plt.scatter(x_test, y_test, color='red')
plt.plot(x_train, regressor.predict(x_train), color='blu
plt.title('sal - ')
plt.xlabel( )
plt.ylabel( )
plt.show()
```

** R :

```
Salary - Dependent Variable
Experience - Independent Variable
#importing dataset
dataset = read.csv('Salary_Data')
#Splitting Data & enabling ca tools
library('caTools')
set.seed(123)
split = sample.split(dataset$Salary, SplitRatio = 2/3)
training_set = subset(dataset, split == TRUE)
test_set = subset(dataset, split == FALSE)
```

Fitting Simple Linear Regression to the Training set.

regressor = lm(formula = Salary ~ YearsExperience,
↓
F1 → help data = training_set)

Console : > summary(regressor)
↳ gives info.

predicting the test results.

y_pred = predict(regressor, newdata = test_set)
↳ vector of values ↳ Argument.

visualising the training set results.

ggplot() +
geom_point(aes(x = training_set$YearsExp,
y = $Salary), colour = 'red') +
geom_line(aes(x = training_set$YearsExperience,
y = predict(regressor, newdata = training_set),
colour = 'blue') +
ggtitle('Salary vs Exp') +
xlab('Year of Expe') +
ylab('Salary')
↳ ylabel.

Multiple Linear Regression :

Simple Linear regression : $y = b_0 + b_1 x_1$

Multiple Linear regression : $y = b_0 + b_1 x_1 + \cdots + b_n x_n$

y → Dependent Variable (DV)

$x_1, \ldots, x_n$ → Independent Variables (IV)

*****

When should we consider Linear Regression

1. Linearity
2. Homoscedasticity
3. Multivariate normality
4. Independence of errors
5. Lack of multicollinearity.

Given Dataset

Profit → DV

R&D spend, Admin, Marketing, State → IV

$$y = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 + \text{[illegible]} \; ???$$

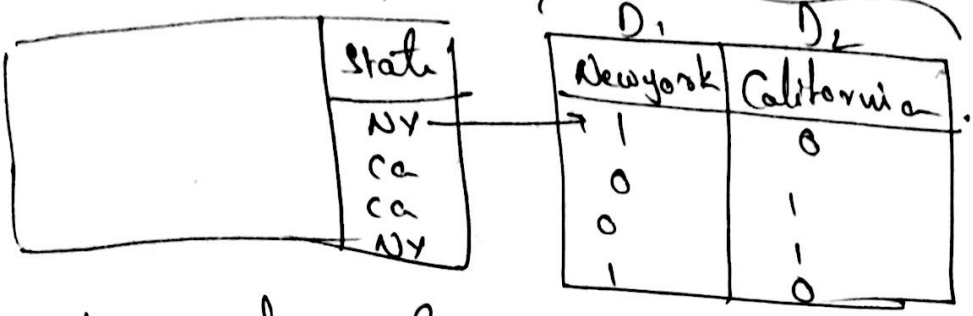
y → Profit, x_1 → R&D, x_2 → Admin, x_3 → Marketing

??? — because state is not a dollar value. It is a string and string cannot be inserted into an equation.

Since State is a String column We expand our dataset like

Dummy Variables.

dataset (ours)



In NY column for NY - 1, Ca - 0
In CA column for NY - 0, Ca - 1

Now eq becomes

$$y = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4D_1 \rightarrow (1)$$

We shouldn't take all the dummy variables.

Dummy Variable Trap:

If we include the last dummy variable D_2 also in the eq (1), the problem is $\boxed{D_2 = 1 - D_1}$.

So basically if you include D_2, you are duplicating D_1 again. then the above equation is used for prediction.

So if you duplicate the dummy variable, then your equation cannot predict properly.

This is called $\boxed{\text{Dummy Variable Trap}}$.

At ~~the same~~ any time, b_0 & b_4D_1, b_5D_2 all three cannot exist at same time.

** $b_0 + b_4 D_1$ (or) $b_0 + b_5 D_2$ is possible.

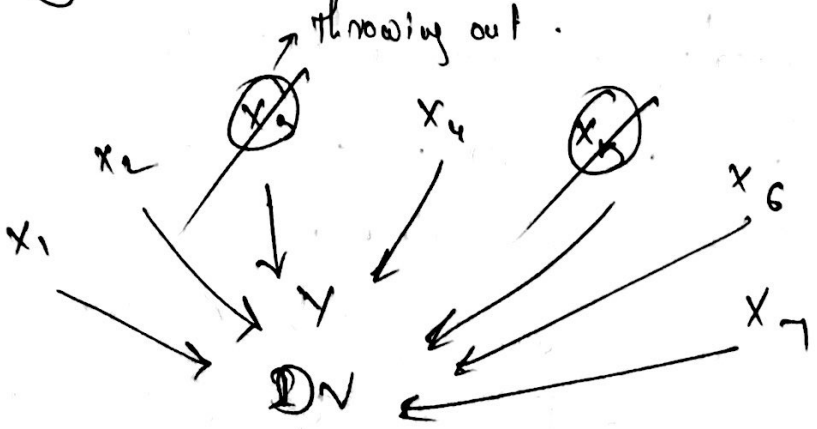
Let's say if we have 9 dummy variables then only include 8.

*** If we have n dummy variables we should only include (n-1) dummy variables.

**** If we have two sets of dummy variables then we should apply (n-1) dummy variables for each set.

***** The phenomenon where one (or) several independent variables predict another in a linear regression is called <u>multi-collinearity</u>.

→ Now in multiple linear regression, We have several IV's unlike Simple Line Regress where it only has one I.V.



We will remove unnecessary IV's while predicting.

1) If we remove lot of IV's then our model will not predict correctly.

2) If we have a thousand IV's then we should consider only those IV's whose behaviour change reflects on DV.

HOW TO CONSTRUCT A MODEL?

5 Methods of Building Models:

1. All-in
2. Backward elimination
3. Forward Selection } Step-wise regression
4. Bidirectional elimination
5. Score Comparison.

1) All-in

(i) Prior knowledge

(ii) You have to

If you know what variables to use (or) company asks you to include ~~at~~ the selected variables then you choose this method.

(iii) If you prepare for backward elimination

(2) Backward Elimination:

Step 1: Select a Significance level to stay in the model (eg: SL = 0.05).

Step 2: Fit the full model with all possible predictors. Include all the IV's.

Step 3: Consider the predictor with the highest P-value. If P > SL, go to step 4, otherwise go to FINISH

Step 4: Remove the predictor

Step 5: Fit model without this variable *.

Step 6: Repeat the process until the highest P-value < SL. then go to finish.

FIN: Model is ready.

(3) Forward Selection:

Step 1: Select a Significance level to enter the model (eg: SL = 0.05).

Step 2: Fit all simple regression model y-xn.

Select the one with lowest P-value.

Basically here we construct models with only

one IV & DV and calculate p value for each model and the lowest p value model is selected.

Step3: Now selecting the lowest p-value model. Add another IV to this model and again calculate p-values for all the models.

StepH: If the p-value for above model of 2 IV's & 1 DV is less than SL, then repeat Step 3. Otherwise FIN.
i.e The model will only be finished when $p > SL$

Step5: If you got, lets say $p(4\,DV's \& 1\,DV) > SL$ then consider the one you have selected with $p(3\,DV \& 1\,DV) < SL$. ~~select~~ That is the final model.

(4) Bi-Directional Elimination:

Step 1: Select a significance level to enter and to stay in the model.

Eg: SL enter = 0.05 & SL stay = 0.05

Step 2: perform the next step of forward selection (new variables must have $P < SL_{enter}$ to enter).

Step 3: perform ALL steps of Backward Elimination (old variables must have $P < SL_{STAY}$ to stay).

You have to iterate from Step 3 to Step 2 & 2 to 3 until you have no variables either to Add in Step 2 or to remove in Step 3.

Now proceed to Step 4.

Step 4: No new variables can enter and no old variables can exit.

↓

FIN: Your model is ready.

(5) All possible model : (Score Comparison)

Step 1: Select a criterion of goodness of fit.

Step 2: Construct all possible regression model. $2^n - 1$. total combinations.

Step 3: Select the one with best criterion.

↓

FIN: Your model is ready.

Ex: 10 columns means 1023 models.

In this tutorial We consider Backward Elimination. As it is the fastest one.

Multiple Linear Regression In Python:

Here since the given data has a string variable, we encode it.

```
# encoding the data
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_x = LabelEncoder()
x[:, 3] = labelencoder_x.fit_transform(x[:, 3])
onehotencoder = OneHotEncoder(categorical_features = [3])
x = onehotencoder.fit_transform(x).toarray()
```

We also have dummy variable in the above created dataset. We need to avoid the dummy variable trap.

```
# Avoiding the dummy Variable trap
x = x[:, 1:]
```

Here the first i.e 0th index dummy variable is removed.

```
# Splitting the data
    :
    :
# fitting multiple linear regression into the training set.
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
```

```
regressor.fit(x_train, y_train)
```

- # predicting the test results.

```
y_pred = regressor.predict(x_test)
```

Finding the highly significant IV in the Given dataset and removing the non-significant ones.

*** Here We are going to use Backward-Elimination.

Building the optimal model using backward elimination

```
import statsmodels.formula.api as sm
```

Usually the above library doesn't take care of b_0 (or) neglects it.

$y = b_0 + b_1 x_1 + \dots b_n x_n$

So we assume there x_0, where $x_0 = 1$

* So we need to add a column of 1's at 0th index.

```
x = np.append(arr = x, values = np.ones((50,1)).astype(int), axis = 1)
```

The above statement appends array of fifty 1's to last position of X. But we need to append it at first position. So, the trick is

```
X = np.append(arr = np.ones((50,1)).astype(int),
    values = X, axis = 1)
                ↓
        either column (or) row = 0
                = 1

X_opt = X[:, [0,1,2,3,4,5]]
regressor_OLS = sm.OLS(endog = y, exog = X_opt).fi
                ↓
        OLS class

regressor_OLS.summary()

#1
X_opt = X[:, [0,1,3,4,5]]
regressor_OLS = sm.OLS(endog = y, exog = X_opt).fit()
regressor_OLS.summary()

.
.
.

repeat until p < 0.05 (SL).
```

R:

```
# Importing the dataset
dataset = read.csv('50_Startups.csv')

# Encoding the data (or) Factoring
dataset$State = factor(dataset$State,
                       levels = c('New York', 'California',
                                  'Florida'),
                       labels = c(1, 2, 3))

# Fitting multiple linear regression to Training set.
regressor = lm(formula = Profit ~ R.D.Spend +
               Administration + Marketing.Spend + State)
               (or)
regression = lm(formula = Profit ~ .,
                data = training_set)
```

*** All the remaining IV's.

*** In R, if in our dataset we have spaces, then in R we replace them with '.'

Ex: R.D. Department.

*** R automatically takes care of Dummy Variable trap.

> Summary (regressor).

P - values less than 5%. - more significant
P - values greater than 5%. - less significant.

Significant codes of P:

P - lies b/w	-	Stars
P - 0 - 0.001	-	***
P - 0.001 - 0.01	-	**
P - 0.01 - 0.05	-	*
P - 0.05 - 0.1	-	.
P - 0.1 - 1	-	nothing

So, in the above given dataset, the only I.V significant is R.D. Development.

So, we can rewrite the formula as

***Formula : Profit ~ R.D. Development.

It still gives the same prediction.

Building Optimal Model in R:

Building the optimal model using backward elimination.

```
regressor = lm(formula = Profit ~ R.D.Spend +
                Administration + Marketing.Spend + State,
              data = dataset)
```

↓

You can also take training but usually recommend to take whole dataset.

```
summary(regressor)
```

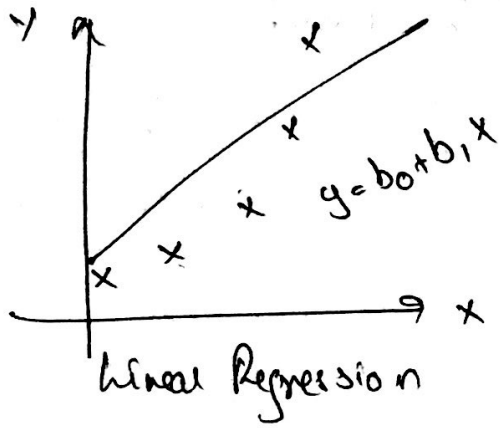
Polynomial Regression :

Intution :

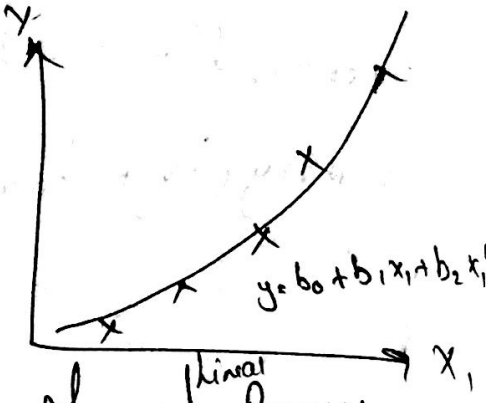
Simple Linear Reg : $y = b_0 + b_1 x_1$

Multiple Linear Reg : $y = b_0 + b_1 x_1 + b_2 x_2 + \dots b_n x_n$

Polynomial (linear) Reg : $y = b_0 + b_1 x_1 + b_2 x_1^2 + \dots b_n x_1^n$



Linear Regression



Polynomial (linear) Regression

← Here linear doesn't meant $x_1, x_2 \dots x_n$. But here linear means coefficients.

Linear : $Y = b_0 + b_1 x_1 + \dots b_n x_n$

Non-linear : $Y = \dfrac{b_0 + b_1 x_1}{b_2 x_2 + b_3 x_3}$

Polynomial linear Regression is a special case of Multiple linear Regression.

→ In machine learning, if you mention only the index like [:, 1], then it will create a vector.

→ But our matrix of features should always be in matrices not in vector.

No. X = dataset.iloc[:, 1:2].values()

↙ upper bound excluded.

*** If we have very less dataset, then we don't split our dataset into training & test_set. We will take the entire dataset and train the machine since data is less.

*** Polynomial regression automatically takes care of feature scaling.

→ Here in this tutorial we build linear regression & polynomial regression to compare them.

→ Polynomial regression automatically adds column of 1's like we did in Multiple Linear Regression. These 1's are used for b_0.

Fitting polynomial regression to the training_set

```
from sklearn.preprocessing import PolynomialFeatures
poly_reg = PolynomialFeatures(degree = 2)   # creates a col of squared values.
X_poly = poly_reg.fit_transform(X)
```

```
lin_reg_2 = LinearRegression()
lin_reg_2.fit(x_poly, y)
```

Here x_poly is contains squared values i.e polynomial values.

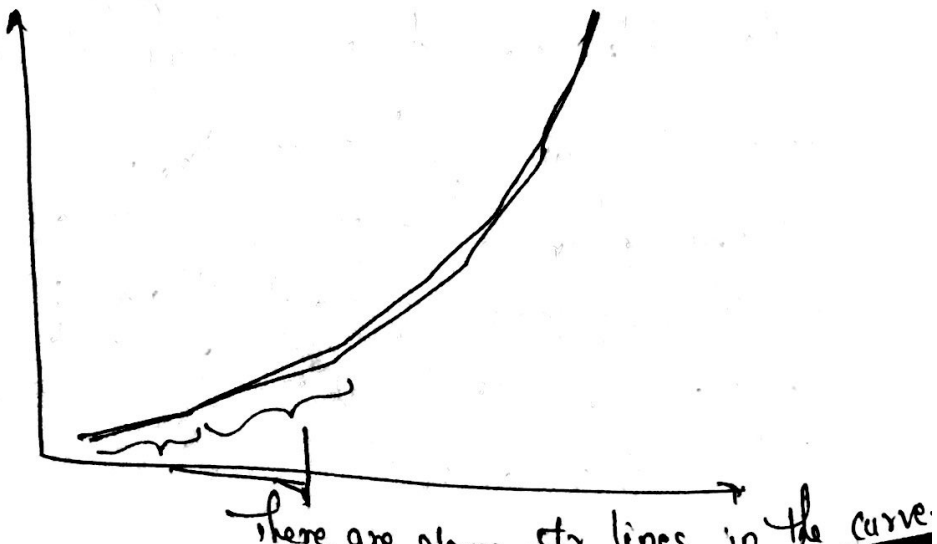
To obtain these polynomial features we use, Polynomial features library.

Visualizing the polynomial regression results.

```
plt.scatter(x, y, color = 'red')
plt.plot(x, lin_reg_2.predict(poly_reg.fit_transform(x
                color = 'blue')
```

Here instead of writing x_poly, we wrote the whole poly_reg.fit_transform( ) because, we cannot write x_poly if its test_set_x ie x_test.

Now the graph will be like



There are some str lines in the curve.

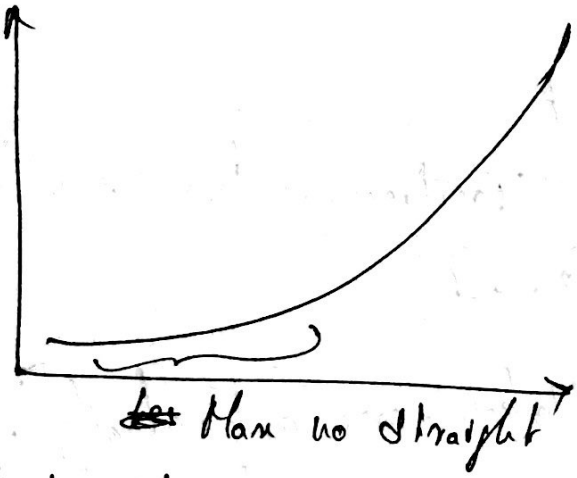
To even reduce those straight lines

We use

```
# visualising the polynomial reg
X_grid = np.arange(min(X), max(X), 0.1)
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, lin_reg2.predict(poly_reg.fit_transform(X_grid))
                                  , color = 'blue')
                    :
plt.show()
```

We use these 2 lines.

Now the graph will be like.



~~Let~~ Max no straight lines.

* X_grid contains 90 rows ranging from 1, 1.01, 1.2 . . . 89.9, 90.

** To increase the prediction we move the degree from 2 to 3.

degree = 3. (More accuracy).

*** To increase even more

degree = 4 (Even more accuracy. Hit eye).

*** On Degree = 4 let's predict the results.

lin_reg (~~[6.5]~~).predict (6.5) = 3,30,378

far away

lin_reg_2.predict (poly_reg.fit_transform(6.5))

= 1,58,862

which is very near to 160k & the predict is true.

R:

```
# import the dataset
dataset = read.csv('Position_Salaries.csv')
dataset = dataset[2:3]
```

*** Feature scaling is taken care automatically.

To add polynomial features to our matrix of features

```
# Fitting polynomial features
dataset$Level2 = dataset$Level^2   (Adding Squared values as level)
"  $Level3 = "  "  ^3
poly_reg = lm(formula = Salary ~ ., data = dataset)
```

Visualising the polynomial regression

```
library(ggplot2)
ggplot() +
  geom_point(aes(x = dataset$level, y = dataset$Salary),
             colour = 'red') +
  geom_line(aes(x = dataset$level, y = predict(poly_reg,
            newdata = dataset)), colour = 'blue') +
  ggtitle(                    ) +
  xlab(                       ) +
  ylab(                       )
```

*** Predicting the values :

In R, you cannot simply write

```
y_pred = predict(lin_reg, 6.5)   X
                          ↓
                          y
```

The above one is false

According to syntax, in y place there should be a dataset.

So, you need to create a dataset with single value. To do that we take data.frame

```
y_pred = predict(lin_reg, data.frame(Level = 6.5))
                                     ↓
                                creating a column
```

→ Predicting Linear Regression.

predicting new result with polynomial regression

```
y_pred = predict(poly_reg, data.frame(Level = 6.5,
                                      Level2 = 6.5^2,
                                      Level3 = 6.5^3,
                                      Level4 = 6.5^4))
```

We should take Level2, 3, 4 because our poly_reg is trained with dataset having Level2,3,4

2nd Non-Linear Model SVR:

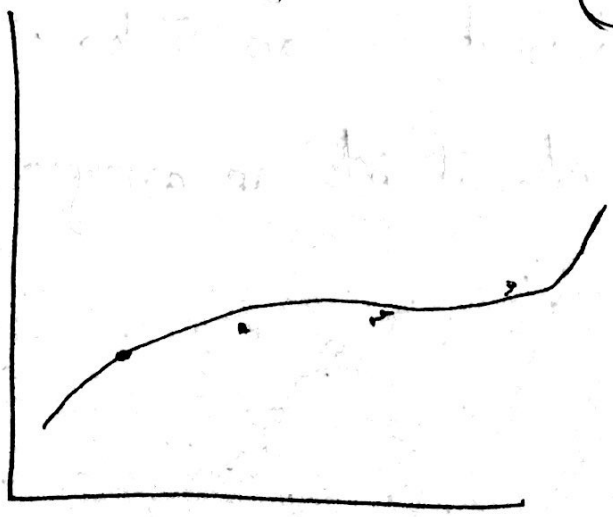
SVR In Python (Support Vector Regression)

In SVR, we need to do feature scaling.
The library in python doesn't care of itself.

```
# Feature scaling
from sklearn.preprocessing import StandardScalar
sc_x = StandardScalar()
sc_y = StandardScalar()
x = sc_x.fit_transform(x)
y = sc_y.fit_transform(y)

# fitting SVR to our regression.
~~regressor = SVR~~
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(x, y)
```

⊙ - outlier.



*** If you take the graph, the outlier point is CEO point with level 10. The SVR considers it as an outlier because the distinction b/w each point and to this point is very large. It suddenly jumped to very high, so SVR marks it as an outlier.

predicting the result.

```
y_pred = regressor.predict(sc_x.transform(6.5))
```

X

Because the input in sc_x.transform must be an array. So

```
y_pred = regressor.predict(sc_x.transform(
        np.array([[6.5]])))
```

Why second bracket?

*** Because if we only put one bracket it will consider it as a vector.

To make it into an array take another bracket.

** The above command will return the scaled prediction of Salary like '0.05'. But we want the real Salary.

So,

```
y_pred = sc_y.inverse_transform(regressor.predict(
        sc_x.transform(np.array([[6.5]]))))
```

*** Why sc_y, why not sc_x.

Because, we are here predicting y, value and the object related to y value is sc_y not sc_x.

*** The inverse_transform returns the original value of the scaled object.

R:

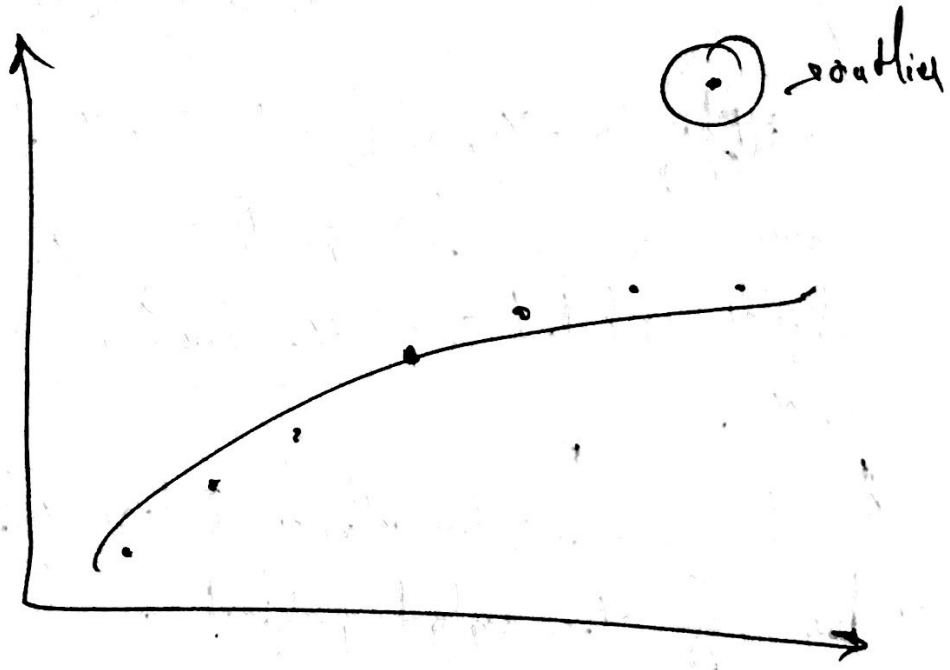
In R, you need to install package : e1071,

```
#install.packages('e1071')
library(e1071)
# creating & fitting the regressor to dataset.
regressor = svm(formula = Salary ~ .,
                data = 'Position_Salaries.csv',
                type = 'eps-regression')
```

*** In type, if we do regression then we choose 'eps-regression' (or) if we do classification then we choose 'C-classification'

*** Here kernel is by default 'rbf'.

If we donot take 'kernel' argument.



Decision TREE

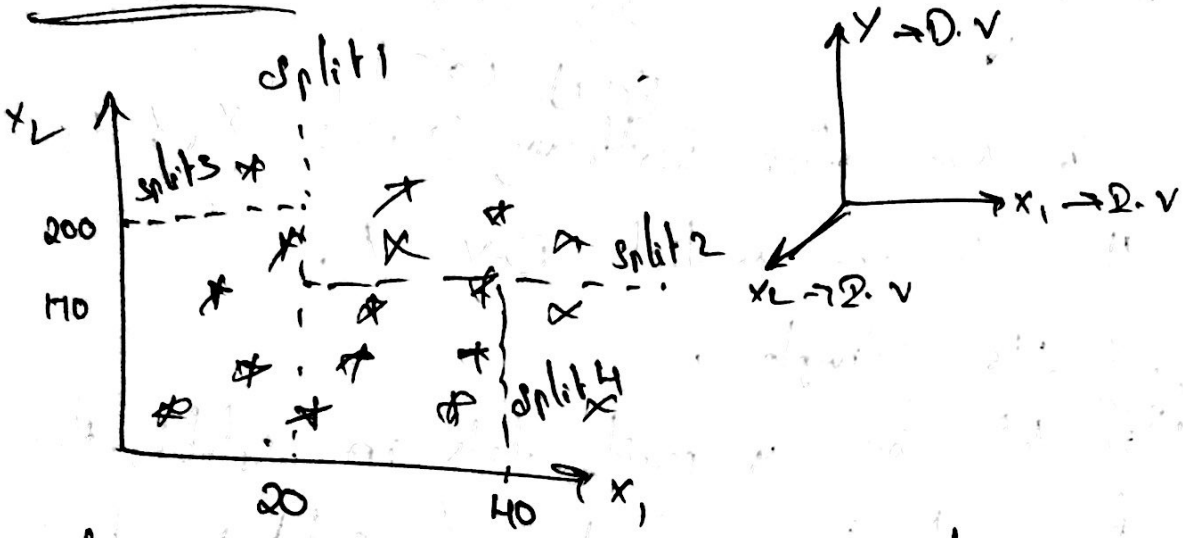
(Non-linear
Not-Continuous
1st M.L Model)

Intution:

CART

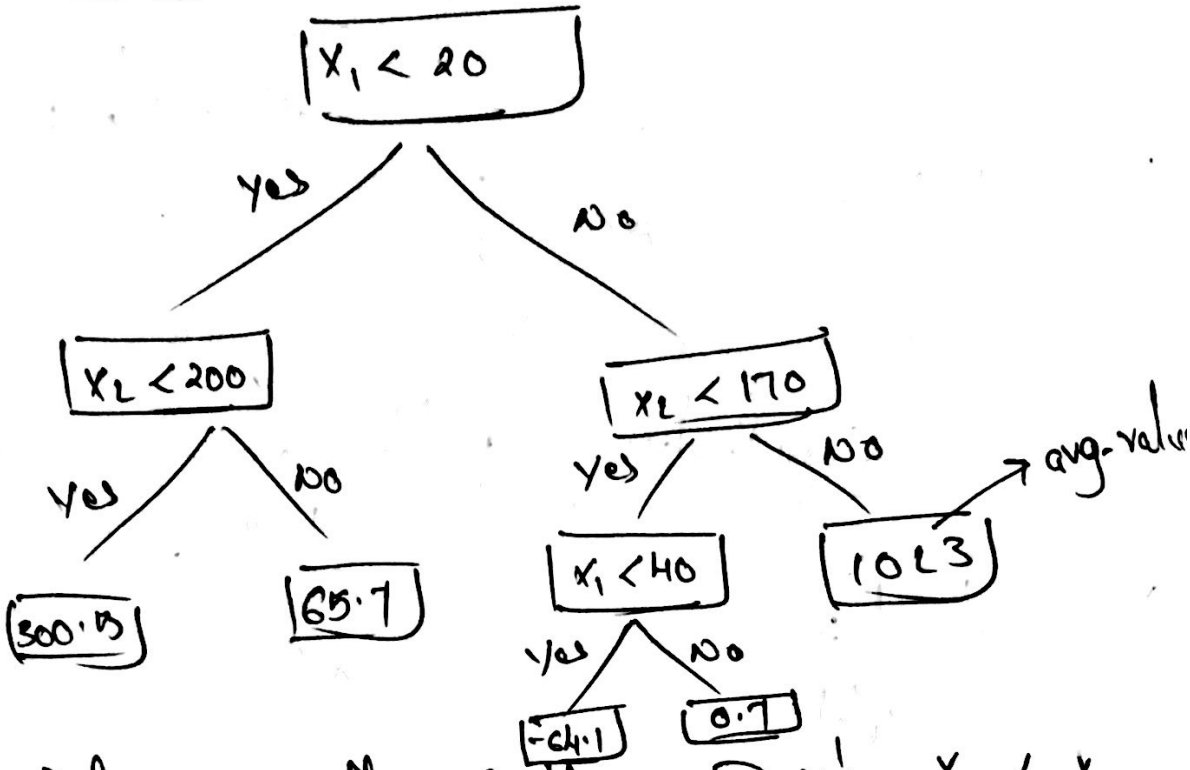
- Classification Tree
- Regression Tree → are more complex

Given Dataset:



The algorithm divides the given dataset into splits according to information entry. These splits are also known leaves. The final leaves are known as 'Terminal leaves'. These leaves are optimal.

Building D.Tree :



Ok. Now after Scattering I.V's X_1 & X_2 How do you predict D.V Y.

*** It's Simple.

*** If lets say y point falls in split 1, then there will be avg-value of split 1 and this value will be assigned to y.

*** This avg-value is calculated using all points in that split and will be assigned to any 'y' value that falls into that split.

Importance of splits:

If we ~~do not~~ have not done any splits then we ~~also~~ might take avg of whole dataset and assign to every 'y' which is very less accurate.

By splitting the data and assigning that split avg value, we get a very accurate value.

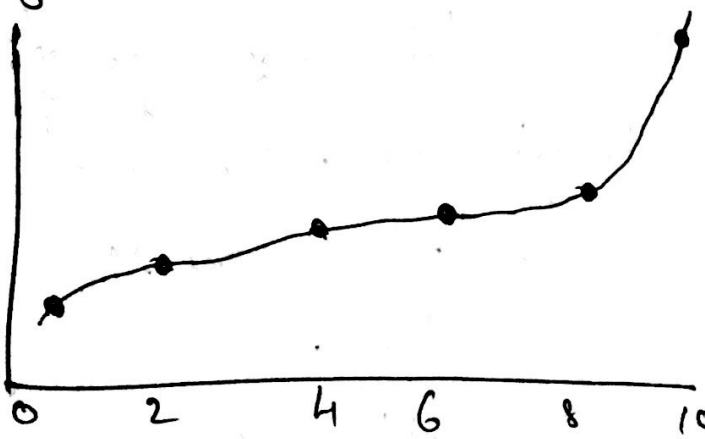
Python

Fitting the D.T regression to the dataset.

```
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state = 0)
regressor.fit(x, y)
```

↳ to get same results as tutorial.

*** If you plot normally as in L.R.L. Poly - then the graph will be like



But in this interval there should only be one pred - value that is avg.

**** So, To clearly see the interval have a constant value for the interval, you may plot the higher-resolution Graph i.e

```
# visualising with higher resolution
x_grid = np.arrange(min(x), max(x), 0.01)
x_grid = x_grid.reshape(len(x_grid), 1)
plt.scatter(x, y, color='red')
plt.plot(x_grid, y, color='blue')
```

R :

Install library 'rpart'

```
# install.packages('rpart')
regressor = rpart(formula = Salary ~ .,
                  data = dataset
```

** If you simply plot then you get a single straight-line. We encountered like this in SVR-python, there we didn't do feature Scaling.

But D.T-R takes care of feature Scaling. So. What's Problem Here?

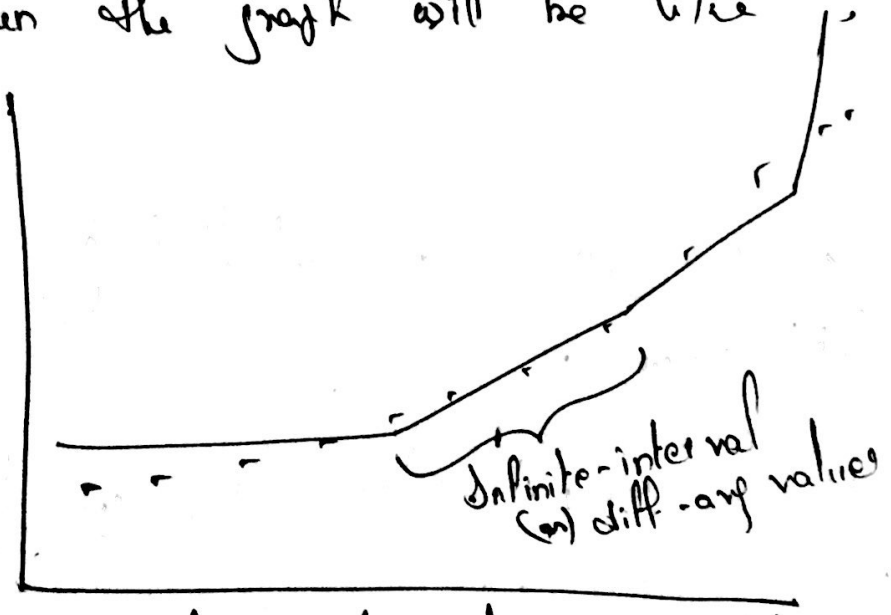
Example Graph.

The problem is it took the whole dataset as a single split

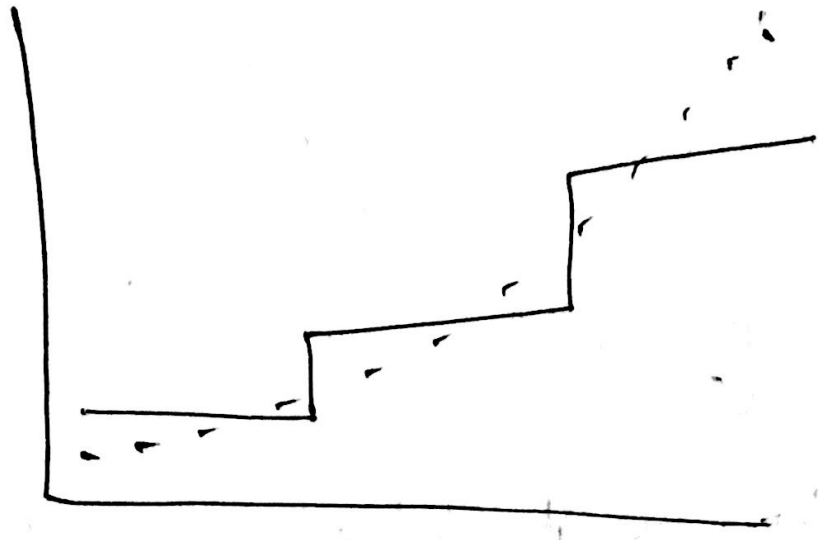
So, To solve this problem, we put a parameter i.e control.

```
regressor = rpart (formula = Salary ~ .,
                   data = dataset, control = rpart.control(
                   minsplit = 1))
```

Then the graph will be like



So To solve the above problem, Consider higher resolution plotting.



Perfect Graph

Visualising using higher resolution.

```
library(ggplot2)
x_grid <- seq(min(dataset$level), max(dataset$level),
              0.01)
ggplot() +
  geom_point( ) +
  geom_line( ) +
  :
```

****

- Here, the y_pred is controlled by 'minsplit' values.

***** Change the 'minsplit' value then y_pred value will be changed.

Random - Forest Regression

Non-linear Reg (or) Multiple Deep Forest trees

Ensemble Learning:

Is when you take multiple algorithms (or) same algorithm multiple times to get more powerful than the original one.

Random - Forest is one type of ensemble learning.

Intution:

Step 1: pick a random k points from training set.

Step 2: Build the Decision Tree associated to these k data points.

Step 3: Choose the number of trees you want to build and repeat Steps 1 & 2.

Step 4: For a new data point, make each one of your N trees predict y value. Now calculate the avg of all these y values and assigned the avg(y-values) as the predicted value.

Python :

```
# Fitting random forest regression into the dataset
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators = 10,
                                  random_state = 0)
regressor.fit(x, y)
y_pred = regressor.predict(6.5)
```

* When n_est = 10 — the prediction was closer
* When n_est = 100 — " " " even closer
* " " = 300 — " " " more even closer and collided with real value (hit bulls eye).

Random forest has beaten Polynomial ~~the~~ Regression Prediction.

R

* If I use dataset$Level (or) dataset$Salary.
The $ sign creates a vector.
* To have a dataframe use dataset[1] → index.

# Fitting the randomForest Regressor to the dataset
# install.packages('randomForest')
library(randomForest)
set.seed(1234)
regressor = randomForest(x = dataset[1],
y = dataset$Salary, ntree = 500)
y_pred = predict(regressor, data.frame(Level = 6.5))

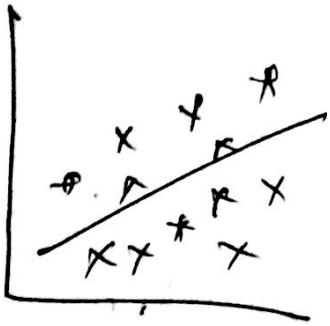
***
*** The Best M.L models are combination of M.L. Models.

R-Squared :

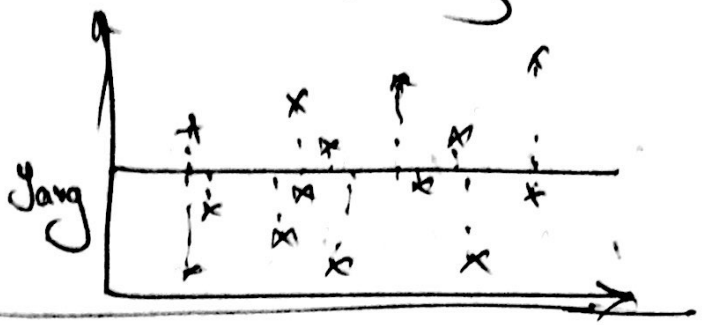
Linear Regression :

$$SS_{res} = SUM(y_i - \hat{y_i})^2$$

↳ Sum of Squared of residuals.



Now instead take avg line i.e



$$SS_{tot} = SUM(y_i - y_{avg})^2$$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

R^2 - tells us how good is our line (res) compare to the avg line.

The closer R^2 is to 1, the better.

SS_{Res} should be smaller to get R^2 closer to 1

R^2 can be -ve. The model here is broken.

Normally R^2 lies b/w 0 & 1.

Adjusted R^2 :

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Problem : the more variables you add.

So when adding new variable, this new I.V will affect SS_{res} such that SS_{res} will be decreased and R^2 will be increased.

In the worst case, if the model thinks this new variable will worsen the prediction (or) increase SS_{res}, it will put the coefficient of I.V as 0, making it 0.

Ex: $y = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3$ (b_3 → coefficient, x_3 → I.V)

$$\text{Adjusted } R^2 = 1 - (n - R^2)\frac{n-1}{n-p-1}$$

p = no. of regressors

n - sample size

I.V increase, p increase

~~Evaluating Regression Models :~~

Evaluating Regression Models :

- Adj R^2 should be more.
- R^2 should be more.

Observe the coefficients. See if they are +ve +ve means, changing them will change the value of D.V in the same direction.

* If a value is like

~~7.9~~ 7.966e-01 means

0.7966

(or)

2.991e-02 means

0.02991

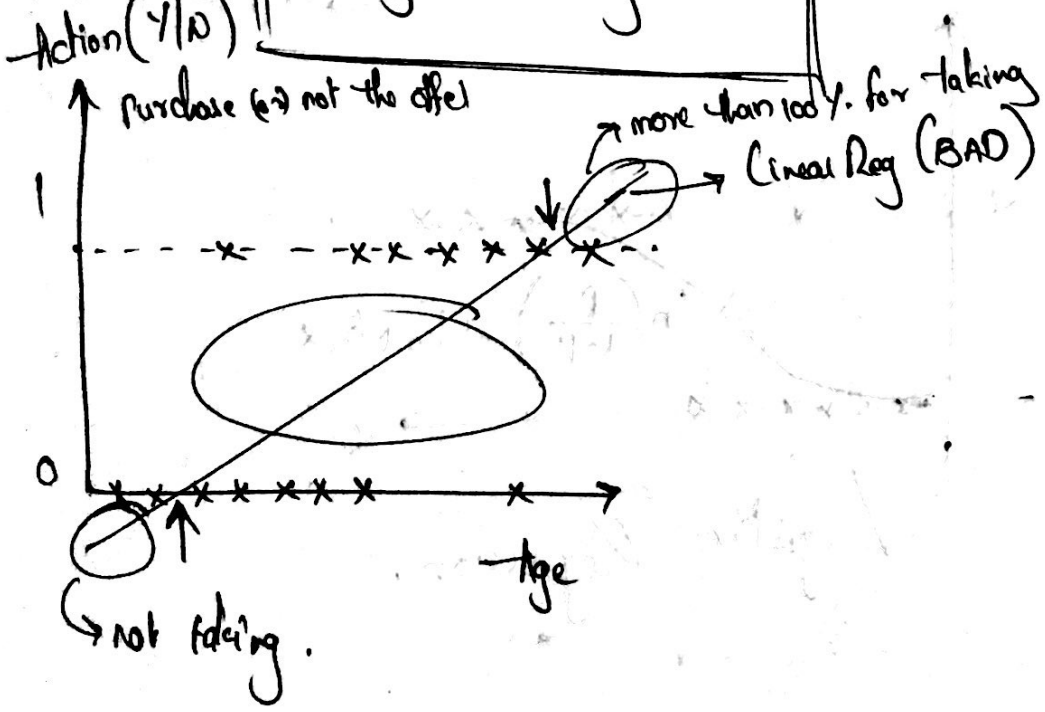
Classification

Regression : the o/p variable takes continous values.
classification : the o/p variable takes class labels.

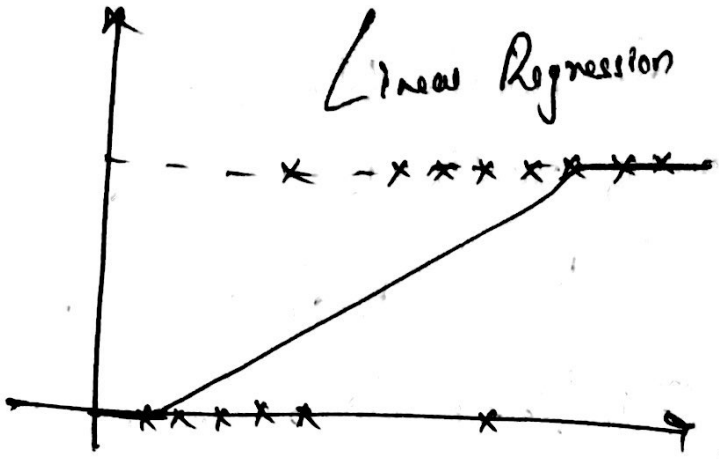
Regression is used to predict o/p values.
classification is group the o/p into a class.

Regression trees have D.p variables that are continous values (or) ordered values.
classification trees have Categorical ~~data~~ and Unordered values.

Logistic Regression



Modifying the above Diag



Linear regression.

$$y = b_0 + b_1 x \rightarrow (1)$$

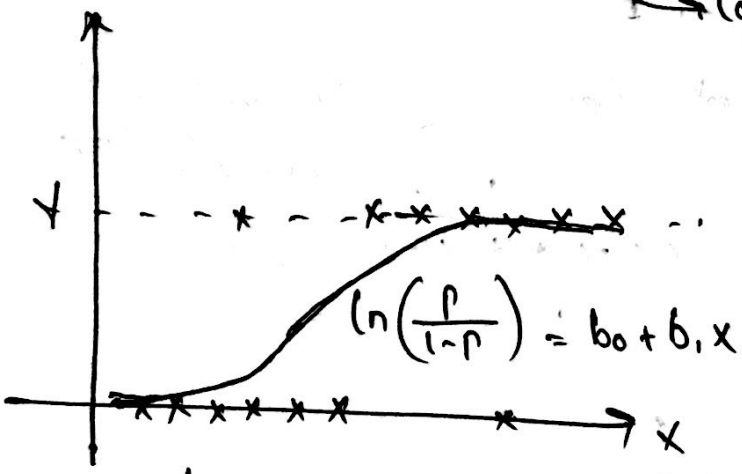
$$p = \frac{1}{1+e^{-y}} \Rightarrow y = \ln\left(\frac{p}{1-p}\right) \rightarrow (2)$$

sigmoid function.

(2) in (1).

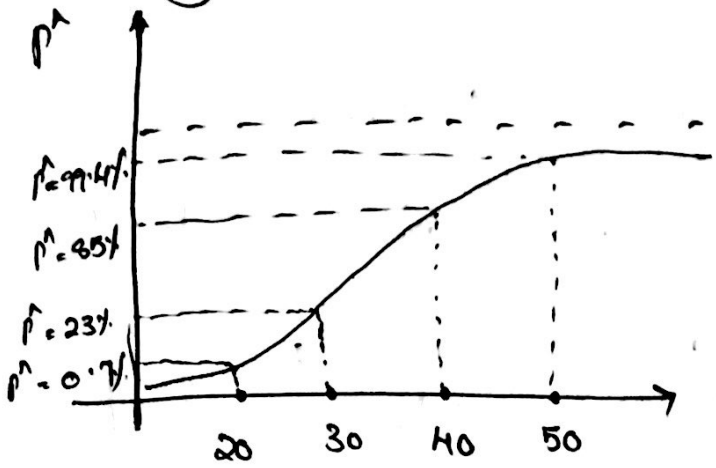
$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1 x$$

→ Logistic regression



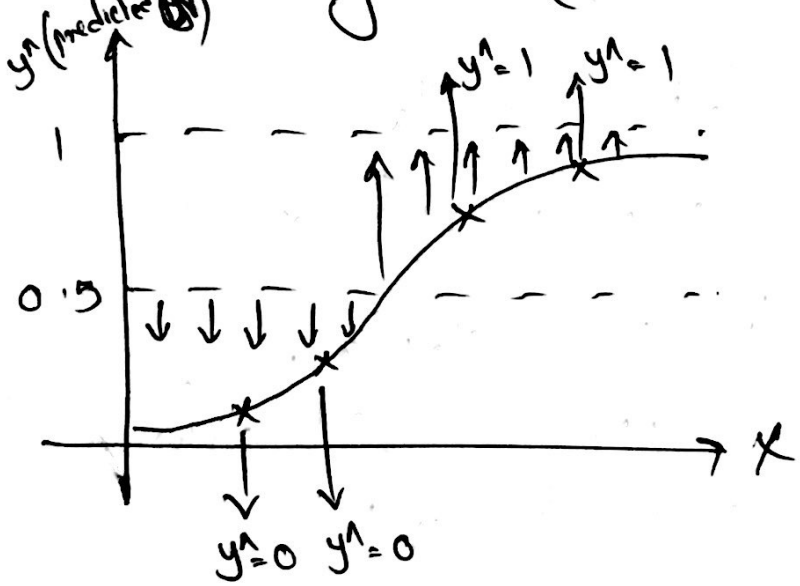
Logistic Regression.

Probability → P-hat ($\hat{p}$)



How do you predict ?

Consider a arbitary line (here it is 0.5)



Python :

Use Data Preprocessing template

*** In logistic regression We have to do feature Scaling.

fitting logistic regression to our dataset

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(x_train, y_train)
```

predicting test set results

```
y_pred = classifier.predict(x_test)
```

Making the Confusion Matrix.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

*** This confusion matrix has the correct & wrong predicted values of y_pred compared to y_test.

Output of cm in terminal will be like

```
array([[65, 3],
       [8, 24]])
```

incorrect, correct

65 + 24 = correct

8 + 3 = incorrect.

n=100	Predicted: NO	Yes
Actual NO	65	3
Yes	8	24

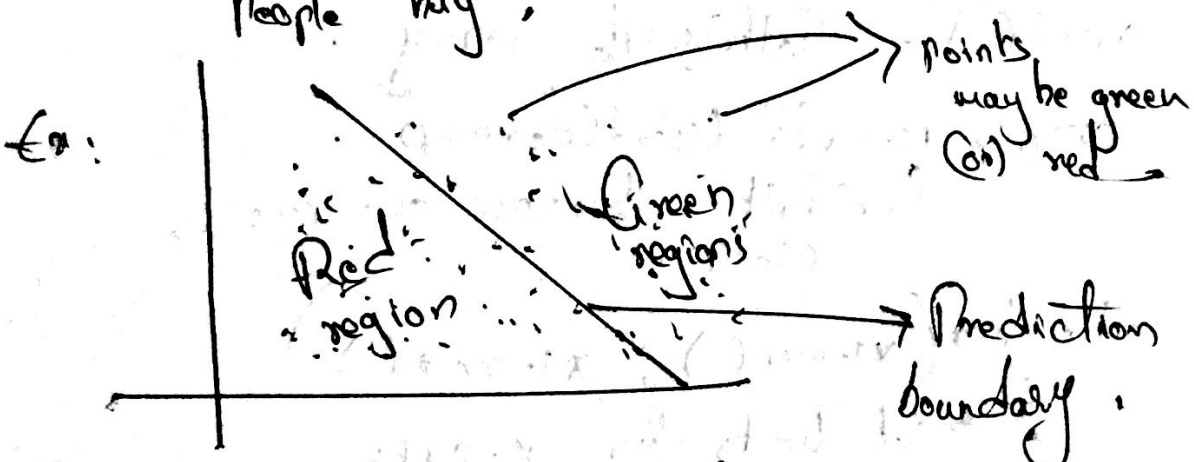
Graph (Very Important *******) :

red point – User didn't buy

Green point – User bought.

Red region – the region is predicted region where people don't buy.

Green region – The region is predicted region where people buy.



Point – It is the truth.

Region – It's Prediction

*** Logistic Regression is a linear classifier. So the prediction boundary is a straight line.

#visualising the training set results.

```
from matplotlib.colors import ListedColormap
X_set, Y_set = X_train, Y_train
X1, X2 = np.meshgrid(np.arrange(start = X_set[:,0].min
    -1, stop = X_set[:,0].max()+1, 0.01), npa
plt.contourf(X1, X2, classified.predict(np.array(
    [X1.ravel(), X2.ravel()]).T).reshape(X1.shape
    alpha = 0.75, cmap = ListedColormap(('red',
```

contourf - It draws a line and shades colors to either side of line.

ravel() - flattens the array.

cmap - none (or) ListedColormap
↳ default ↳ your own colors.
colors.

```
plt.xlim(X1.min(), X1.max())
    ↳ set limits for X-axis
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set==j,0], X_set[y_set==j
        c = ListedColormap(('red', 'green'))(i),
            label = j)
```

```
plt.title (          )
plt.xlabel (         )
plt.ylabel (         )
plt.legend()
plt.show()
```

** Also visualise the training set.

R:

* Use Data preprocessing template
* Change the needed requirement

*** ** Logistic Regression needs feature Scaling.

Here we only scale column 1, 2.

```
# feature Scaling
training_set[, 1:2] = scale(training_set[, 1:2])
test_set[, 1:2] = scale(test_set[, 1:2])
```

Fitting logistic regressor to our training set:

Here we use existing function of R i.e 'glm' to build logistic regression.

↳ Generalised linear Model.

classifier = glm(formula = Purchased ~ .,
family = binomial, data = training_set)

↳ Logistic regression belongs to binomial family.

Predicting Test Results :

Usually we predict in one step. But here we are calculating in 2 steps adding calculation of Probability. Logistic

prob_pred = predict(classifier, type = 'response',
newdata = test_set[-3]) ← Removing D.V

Here 'type = response' for logistic regression we choose 'type = response', so that it will return the results in a single vector.

y_pred = ifelse(prob_pred > 0.5, 1, 0)
↳ True
↳ and false

~~*** etc add~~

*** To add [-3] i.e to remove D.V, its your wish.

*** It is Mandatory to add response.

***** Logistic Regression returns probability that you need to convert.

Build Confusion Matrix i.e Evaluating:

```
cm = table(test_set[, 3], y_pred)
```

↳ creates confusion matrix.

```
cm =      0     1
      0  57     7
      1  10    26
```

} 100.

Visualising The Training Set Results:

Install 'ElemStatLearn'

```
install.packages('ElemStatLearn')
library(ElemStatLearn)
set = ~~test_set~~ training_set
X1 = seq(min(set[, 1]) - 1, max(set[, 1]) + 1,
         by = 0.01)
X2 = seq(min(set[, 2]) - 1, max(set[, 2]) + 1,
         by = 0.01)
grid_set = expand.grid(X1, X2)
colnames(grid_set) = c('Age', 'EstimatedSalary')
```

*** Setting column names for newly created grid_set which contains X1 & X2 as its column. We then predict for this ~~X_grid~~

grid-set giving colors 'springgreen3' & 'tomato'

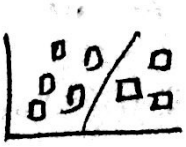
```
prob_set = predict(classifier, type = 'response',
                   newdata = grid_set)
y_grid = ifelse(prob_set > 0.5, 1, 0)
plot(set[, -3],
     main = 'Logistic Regression (Training set)',
     xlab = 'Age', ylab = 'Estimated Salary',
     xlim = range(x1), ylim = range(x2))
contour(x1, x2, matrix(as.numeric(y_grid),
        length(x1), length(x2)), add = TRUE)
points(grid_set, pch = '.', col = ifelse(y_grid == 1,
       'springgreen3', 'tomato'))
points(set, pch = 19, bg = ifelse(set[, 3]
       == 1, 'green4', 'red3'))
```

→ Drawing the line — Logistic Reg line

→ coloring the plane

→ Should the points be in circular type, rectangle, etc

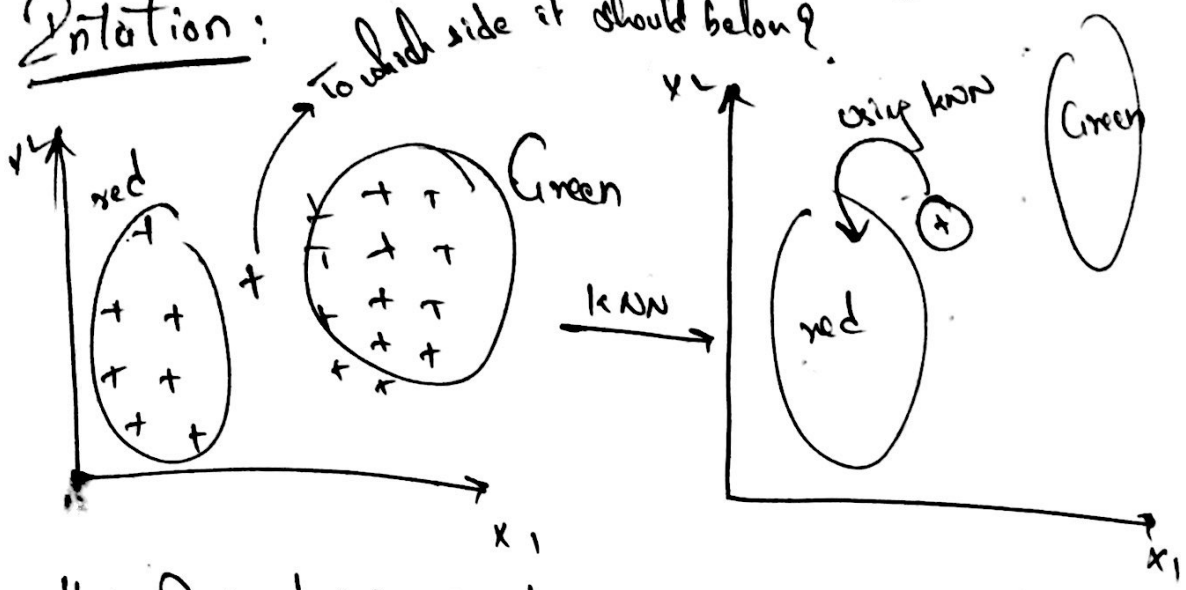
circ 

rect  etc.

K - Nearest Neighbours

Non-Linear Classified.

Intuition:



How Does KNN Work:

Step 1: Choose no of k neighbours. By default $k = 5$.

Step 2:

Take k nearest neighbours of the new data point, according to the Euclidean distance.

Step 3:

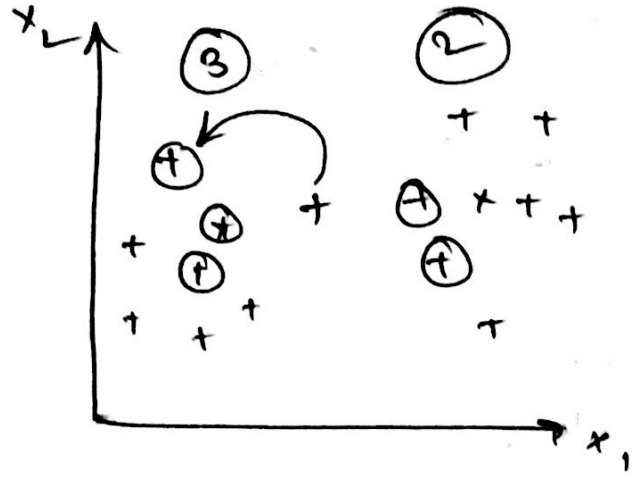
Among these neighbours, count the number of data points in each category.

Step 4:

Assign the new data point to the category where you counted the most neighbours.

Euclidean distance b/w two points $p_1(x_1, y$ & $p_2(x_2, y_2)$

$$= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



Python :

~~##~~ Use classification - template.py.

fitting the kNN to our training set.

from sklearn.~~preprocessing~~ neighbours import kNeighboursClassi

classifier = kNeighboursClassifier(n_neighbours=5
metric = 'minkowski', p = 2)

*** To choose Euclidean distance for kNN, You have to take metric = 'minkowski' & p = 2.

classifier.fit(x_train, y_train).

~~Alter~~ # predict x_test

:

Build confusion_matrix

	0	1
0	64	4
1	3	29

64+29 - Correctly predicted.

7 - Incorrectly predicted.

Comparing Incorrectly predicted results.

*** Logistic Reg (Incorrect) - ~~[illegible]~~ 17

*** KNN (Incorrect) - 7 (Much Better)

Visualise the Graph.

*** You might think that the graph should change for training_set & test_set. It doesn't because, the graph is build with every pixel point with difference of 0.01. So, the test-points are also in that array containing points with diff of 0.01. So when you take test_set it will just scatter points.

R :

Use the template.

fitting knn to the training set & predicting the test_set results.

```
y_pred = knn(train = training_set[, -3],
             test = test_set[, -3], cl = training_set[,
             k = 5)
```

k = 5 → no. of neighbours

cl → telling D.V's.

Note: Usually every time, we first create regressor & then predict the test_set. But in knn, we do both at same time.

Making confusion matrix.

```
cm = table(test_set[-3], y_pred)
```

visualising

Everything is same except only one change.

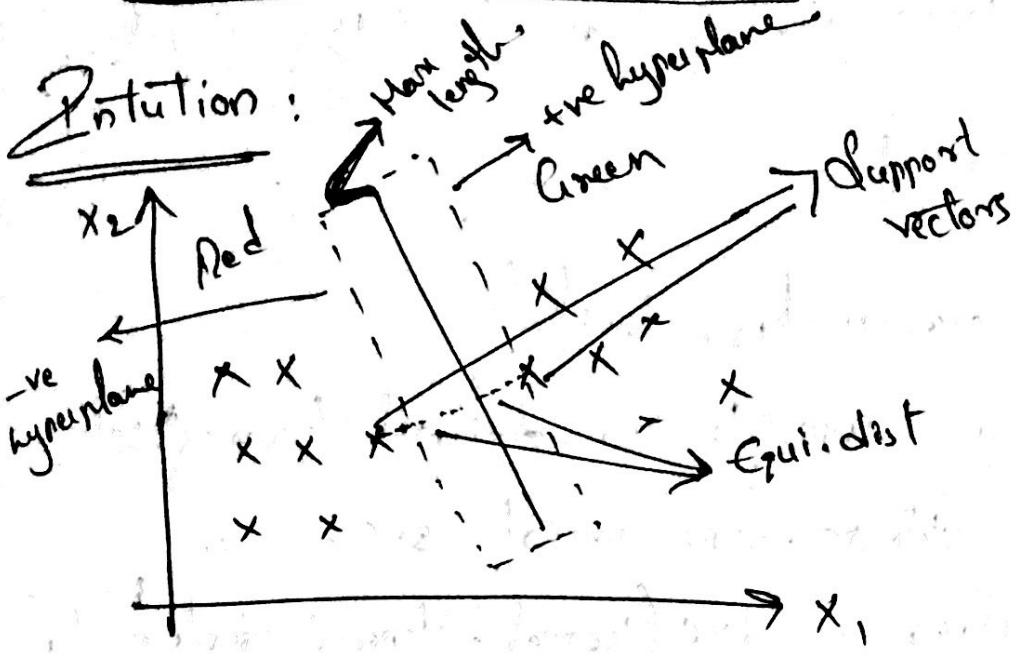
Usually

```
y_grid = predict(classifier, newdata = grid_set
```

Change to

```
y_grid = knn(train = training_set[, -3],
             test = grid_set, cl = training_set[, 3],
             k = 5)
```

Support Vector Machine (SVM)



What is Popular about SVM?

Problem: Machine must be able to identify apples & oranges.

Usually: Usually we take apples ~~that are~~ that are more likely apples (100% apples) (or) oranges and train machine on them & then predict

SVM: But in SVM we consider apples that are less likely apples but apples (or) oranges and train machine on them.

**** These less likely apples (or) oranges are called Support Vectors.

*** This SVM may seem like risky because we train machine on boundary objects.

Python :

Use classification template

fitting svm to our ~~dataset~~ training set.

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state
                                        = 0)
classifier.fit(x_train, y_train)
```

cm:

	0	1
0	66	2
1	8	24

incorrect - 10
correct - 90

Its similar to Logistic Regression.

visualise as in classification template.

R:

Use the classification_template

```
# fitting our svm to the dataset
library(e1071)
classifier = svm(formula = Purchased ~ .,
                 data = training_set, type = 'C-classification',
                 kernel = 'linear')
# predicting test_set results.
y_pred = predict(classifier, newdata = test_set).
```

cm:

	0	1
0	57	7
1	13	23

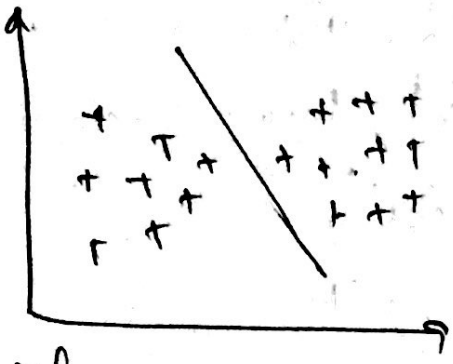
Correct

Incorrect

Kernel SVM

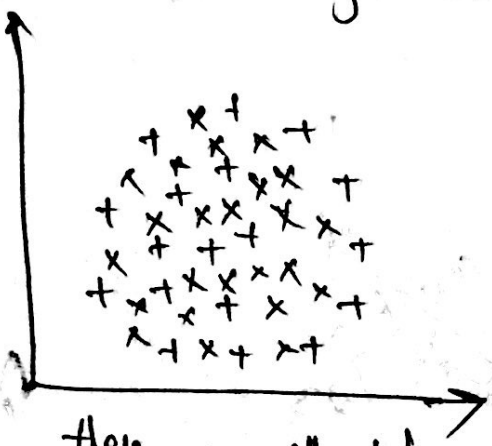
Intution :

Usually you can draw a line when



→ Linearly Seperable

Then how will you draw when

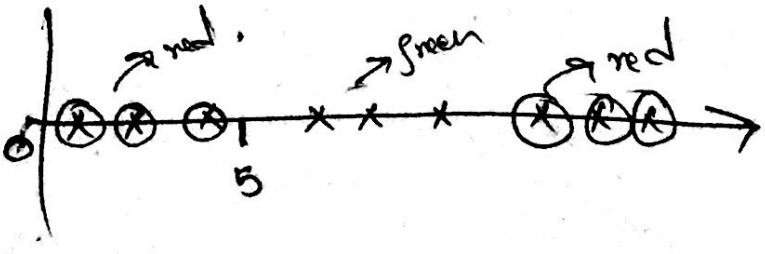


→ Here data is not linearly Seperable.

*** So, here we will take a higher dimension and make the data to linearly seperable and then do kernel. SVM.

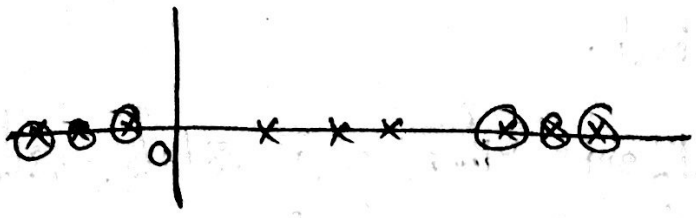
Higher - Dimensional Space :

Consider a Single dimension with 9 points.

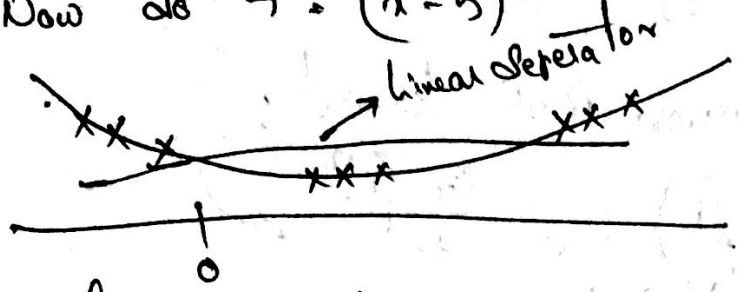


consider 3rd point at 5.

(1) then $f = x - 5$ will be



(2) Now do $f = (x-5)^2$



this happened possible by adding another dimension.

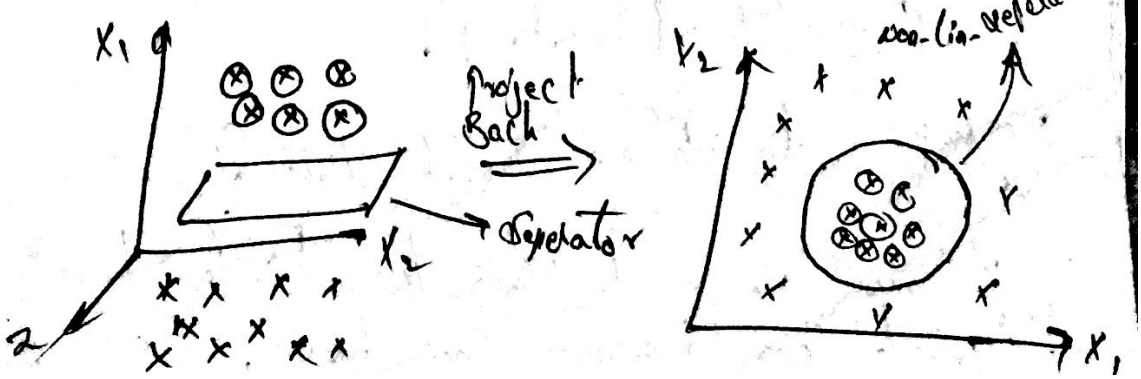
Similarly for 2D, You add another dimension and make it 3D.

$$\phi(x_1, x_2) = (x_1, x_2, z)$$

Mapping function

(3) Now after having linear seperator We project it back to original dimension.

Ex: When you dim to 2D and make it 3D take linear seperator as plane.



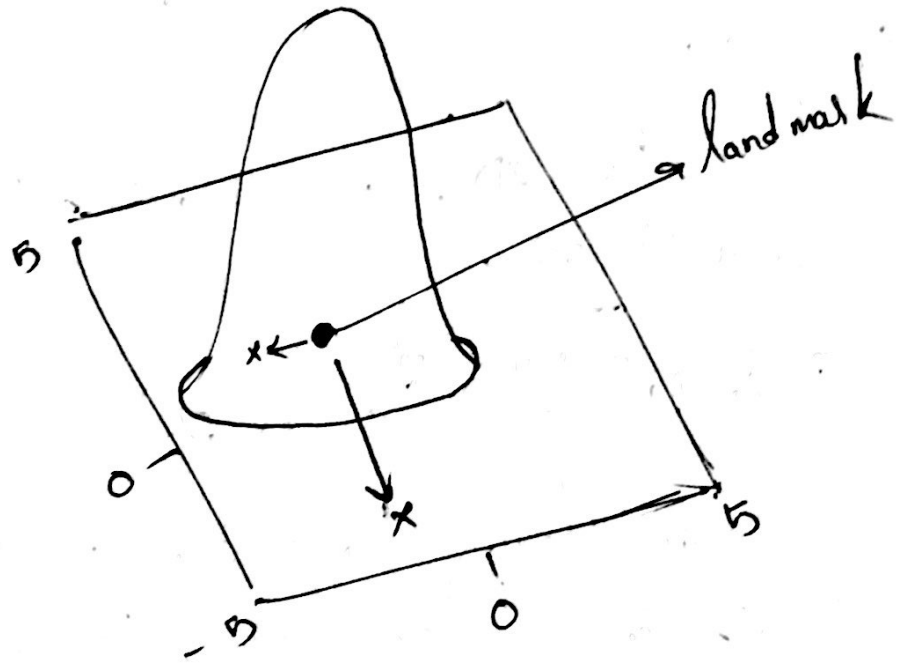
*** Mapping to higher Dimensional Space is highly Compute intensive. So this approach isn't the best.

But in kernel-SVM there is a catch, without going into higher Dimensional Space.

The Gaussian RBF kernel:

$$k(\vec{x}, \vec{l}^{i}) = e^{-\frac{\|\vec{x} - \vec{l}^{i}\|^2}{2\sigma^2}}$$

point — landmark

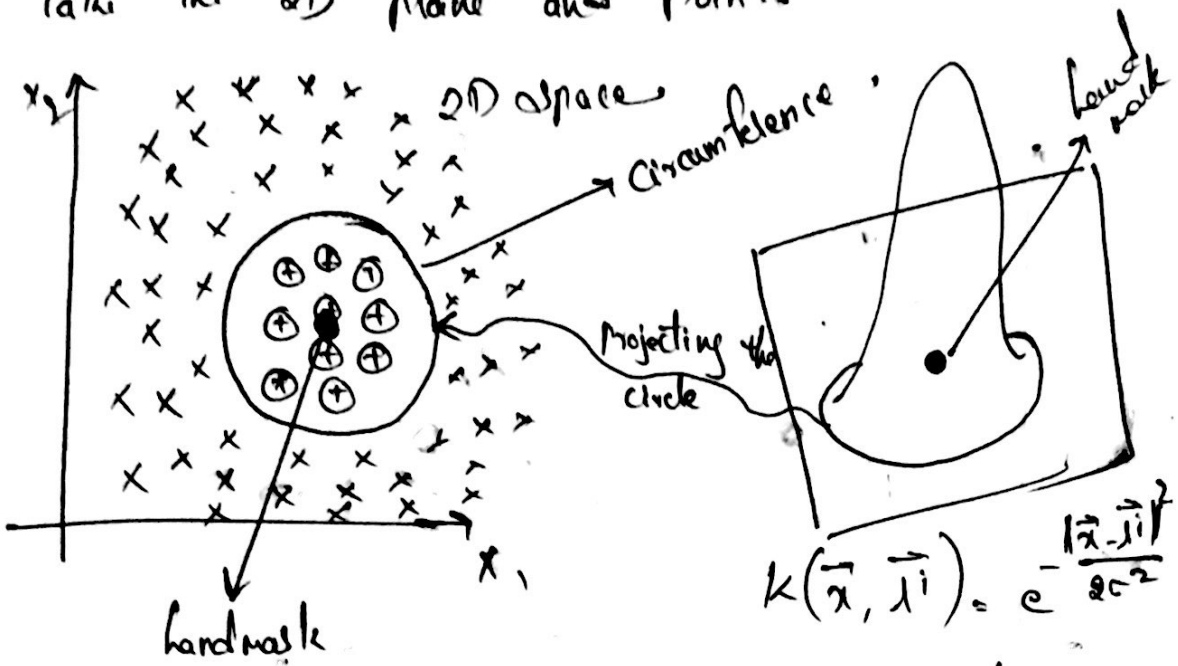


** If your point is closer to landmark then the k will be scaled to 1 and vice versa (By subtle in formula).

***** Why do we need this ?

Because we need this kernel function to seperate our decision boundary.

Take the 2D plane and Points



*** Any point that will fall out of the circumference then the value of that point will be nearer to 0.

*** Any point that will fall out of the circumference, its value will be 1.

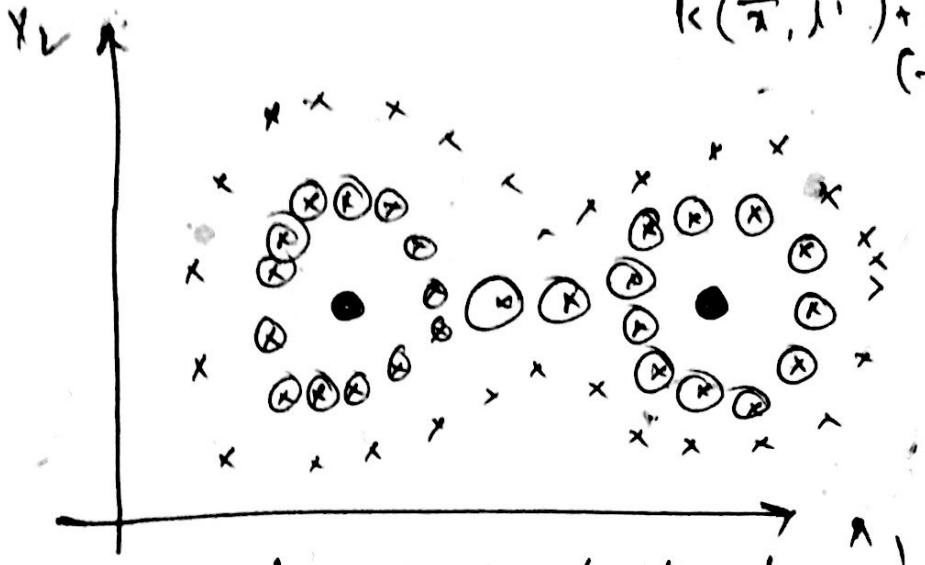
This How you Seperate Green points and Red points.

*** Role of σ:

σ Defines the circumference. Increase the σ, circumferance increases. So to get best results, You need to find best 'σ'.

Complex Example :

$k(\vec{x}, \vec{l^1}) + k(\vec{x} + \vec{l^2})$
(two kernel function



Now take two kernel functions :

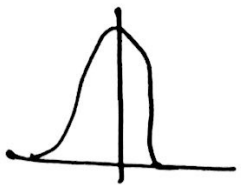
$k(\vec{x}, \vec{l^1}) + k(\vec{x} + \vec{l^2})$ → ①.

Substitute the point in ① . in both kernel
If its close to 0 - red point (x):
close to 1 - Green point (x)

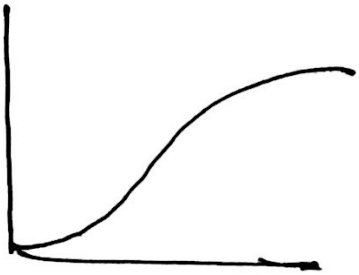
Types of Kernel functions:

In all these types, the essence is same. You choose a landmark and from there you calculate the distance using the formulae.

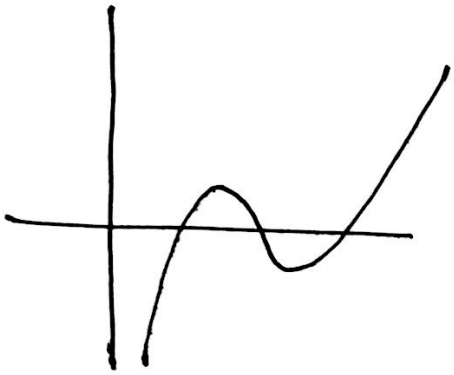
Types :



Gaussian RBF kernel ⟹ $k(\vec{x}, \vec{x^i}) = e^{-\frac{\|\vec{x} - \vec{x^i}\|^2}{2\sigma^2}}$



Sigmoid kernel ⟹ $k(x, y) = \tanh(\gamma \cdot x^T y + r)$



Polynomial kernel ⟹ $k(x, y) = (\gamma \cdot x^T y + r)^d, \gamma > 0$

Note : You do not need to go deep in every type of kernel, just know the types.

To just see how these kernels exists in 3D :

Visit : ~~mlkernels~~

ml kernels.readthedocs.io/en/latest/kernelfunctions.html

Python :

#fitting the regression to our trainingset.

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'rbf', random_state = 0)
classifier.fit(~~training_set~~ x_train, y_train)
```

** Rest is same

Confusion Matrix :

	0	1
0	64	4
1	3	29

*** After using kernel-svm, it makes the data linearly separable.

R :

#**** You have to install package 'e1071' (or) kernlab.

```
#install.packages('e1071')
library('e1071')
classifier = svm(formula = Purchased ~ .,
                 data = training_set, type = 'C-classification',
                 kernel = 'radial')
```

Here 'radial' is Gaussian kernel

Remaining is Laine

CM :

	0	1
0	58	6
1	4	32

Naive - Bayes Theorem :

$$P(A/B) = \frac{P(B/A) * P(A)}{P(B)}$$ } Bayes Theorem

P - probability.

Lets take :

Machine 1 : 30 wrenches / hr

Machine 2 : 20 wrenches / hr

Out of all produced parts :

We can see 1 % are defective.

Out of all the defective parts :

Machine 1 - 50 %.

Machine 2 - 50 %.

Question :

What is the probability that a part produced by machine 2 is defective.

P(Mach 1) producing parts = 30 / 50 = 0.6

P(Mach 2) Producing parts : 20 / 50 = 0.4

P(Defective) = 1%.

P(Machine 1 / Defect) = 50%.

P(Mach 2 / Defect) = 50%.

P(Defect / Mach 2) = ?

Using Bayes Theorem

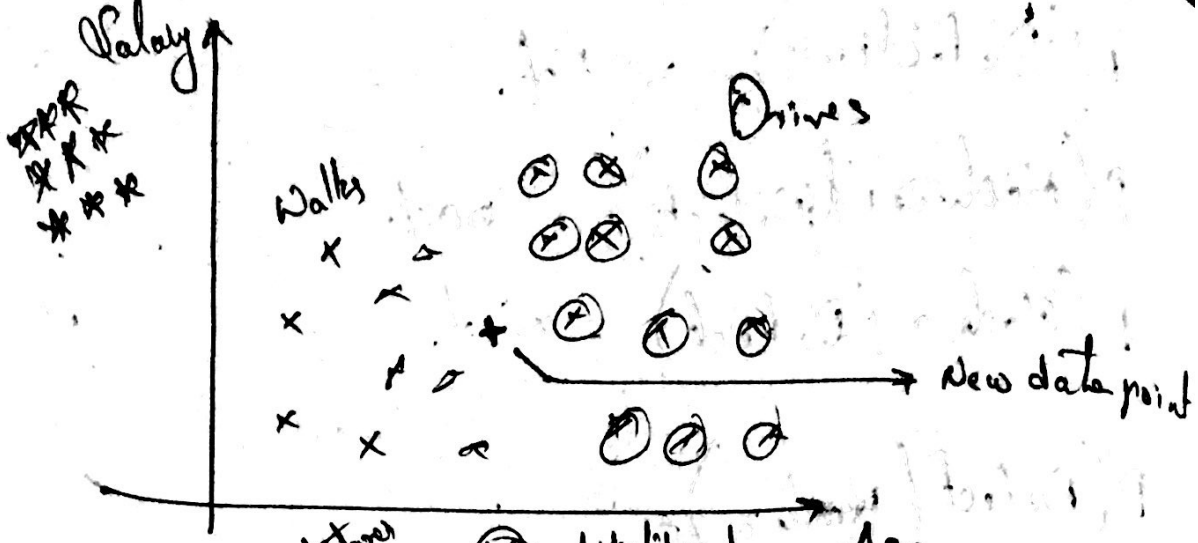
$$= \frac{P(\text{Mach2}/\text{Defect}) * P(\text{Defect})}{P(\text{Mach2})}$$

$$= \frac{0.5 * 0.1}{0.4} = \frac{0.5}{4} = 0.0125$$

= 1.25%.

Solution:

$$P(A/B) = \frac{P(B/A) * P(A)}{P(B)}$$



$P(\text{Walks}/x) = \dfrac{P(x/\text{Walks}) * P(\text{Walks})}{P(x)}$

(#4) Posterior probability ← features

(#3) Likelihood

(#2) Marginal likelihood

(#1) Prior probability

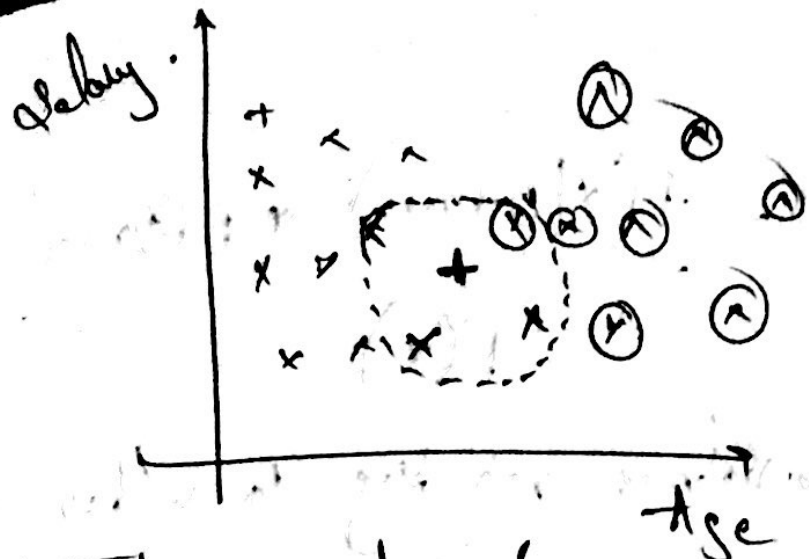
$P(\text{Drives}/x) = \dfrac{P(x/\text{Drives}) * P(\text{Drives})}{P(x)}$

$P(\text{Walks}/x)$ vs $P(\text{Drives}/x)$

(#1) $P(\text{Walks}) = \dfrac{\text{No. of Walkers}}{\text{Total observations.}}$

(#2) How do you calculate $P(x)$?

P.T.O



1. Take a radius of your own choice and draw a circle around the new point.
2. Take all the points inside the circle except the new one.

$$P(x) = \frac{\text{No. of similar observations}}{\text{Total observations.}}$$

$P(x)$ is ~~not~~ by adding new points, the probability of the new point falling in the radius.

#3 $P(x/\text{walking})$

In the circle in the above diag, since we are calculating for walking, the probability of ~~pair~~ the newly added point to be walking in the circle

$$P(x/\text{walking}) = \frac{\text{No. of Walkers in circle}}{\text{Total no of Walkers}}$$

Similarly.

$$P(\text{Drives}|x) = \frac{P(x|\text{Drives}) * P(\text{Drives})}{P(x)}$$

After solving according to video

$P(\text{Walks}|x) = 0.75$.

$P(\text{Drives}|x) = 0.25$

Since $0.75 > 0.25$

That person with features x, will most like Walk.

****

1. Why Naive?
2. $P(x)$
3. What happens when there is more than 2 features.

1.A) Naive Bayes theorem takes independence assumptions. These assumptions are some time wrong. That is why its called "Naive".

1) In our example, consider Age, Salary.
Naive Bayes assumes that Age & Salary are independent. But whereas they are not.
For Naive Bayes they have to be independent.

2.) $P(X)$
Go Back to calculating $P(X)$. (Formulae)

3.) If were there 2 features like walking & Driving. Then you have to calculate atleast 2.

In our example we have Walking & Driving. If We have walking probability then we have $p(\text{Driving})$. We do not need to calculate $p(\text{Driving})$ because
$P(\text{Walking}) + P(\text{Driving}) = 1$.

Python : ***** There will no Arguments in NB (diff)

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(x=train, y=train)
```

cm :

	0	1
0	65	3
1	7	25

R :

```
library(e1071)
classifier = naiveBayes(x = training-set[-3],
                        y = training-set[3])
## predicting test-set
y-pred = predict(classifier, newdata = test-se
```

***** After running the above one, in console

You can see factor levels : 0.

That is because naiveBayes() class doesn't identify (or) cannot automatically encode ~~the factor~~.

You have to manually encode. ~~the factor~~ feature scaling

So here without encoding, ~~factoring~~ doesn't work. (Encoding ≠ Factoring)

Encoding the target feature as factor.

dataset$Purchased = factor(dataset$Purchased,
levels = c(0, 1))

cm :

	0	1
0	57	7
1	7	29

1 Decision Tree Classification

Intution:

Splits are taken according to maximize the category. In background it is very deeper

History:

* These are very old methods. They were verge on dying.
* So they came with upgrades.
  1) Random Forest
  2) Gradient Boosting
  3) etc
* Decision Trees are used in other methods that creates very powerfull algorithms that are used in facial recognition, xbox kinect etc (Wii).

  Microsoft has used Random Forest.

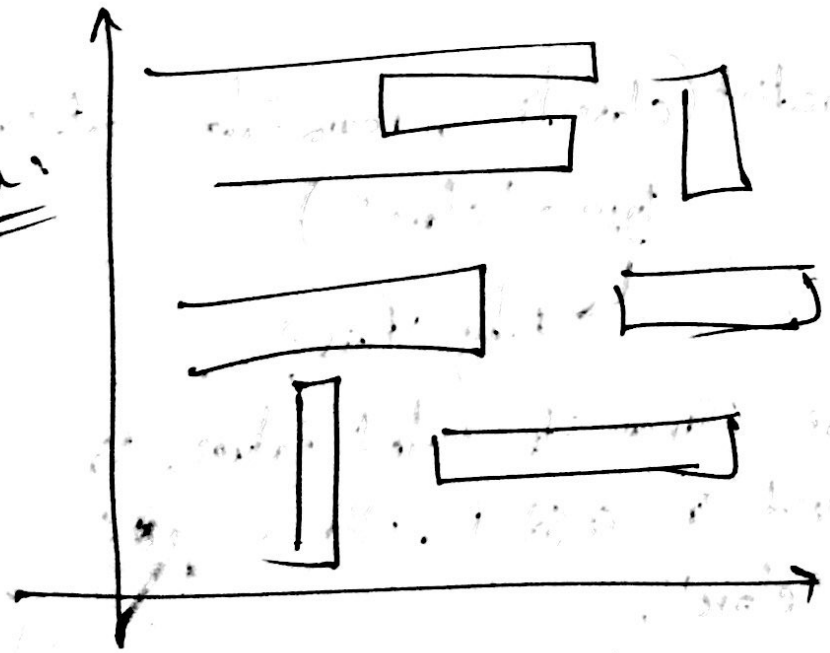
Python :

```
# fitting the regressor to our dataset
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion =
                'entropy', random_state
                = 0)
```

criterion (ctrl + I : to see info)
↳ entropy - for info gain
↳ gini - for impurity

```
classifier.fit(x_train, y_train)
```

for :



*** It tries to catch every observation in the training set. It's called overfitting.

Cm :

	0	1
0	62	6
1	3	29

R :

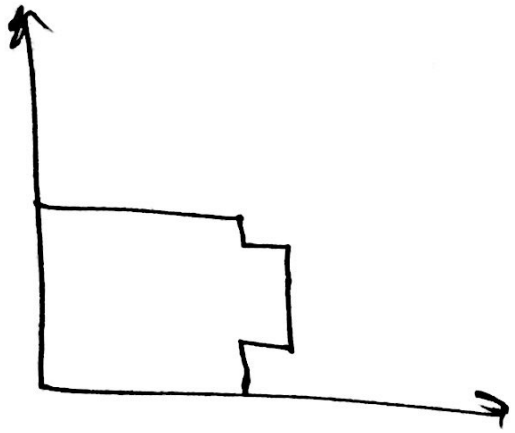
```
# fitting the classifier
library(rpart)
classifier = rpart(formula = Purchased ~ .,
                   data = training_set)

# predicting
y_pred = predict(classifier, newdata = test_set[-3],
                 type = 'class')
```

↳ Why this?

Because, Since it's scaled values, You want y_pred in 0 or 1. So we add type = 'class'.

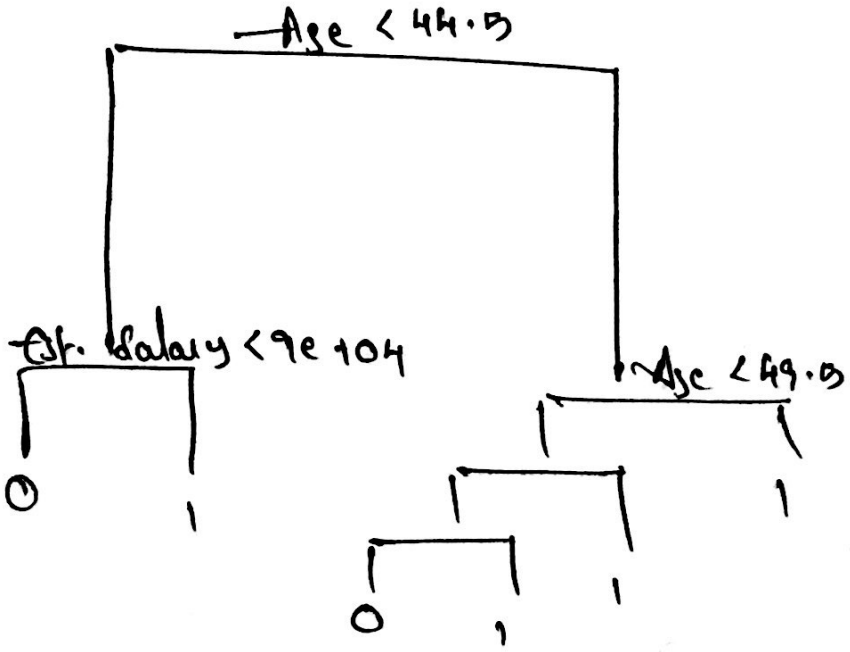
*** Here there is no overfit. So, the graph will be different than python.



plotting the decision tree

plot (classifier)

text (classifier)



*** Since here feature scaling is not required, if you don't use feature scaling then you will have original values in the graph.