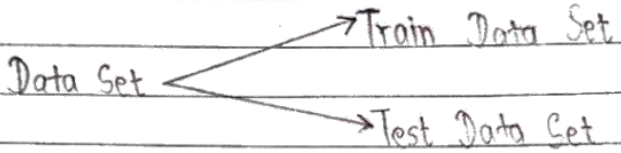
4. Supervised Learning in ML

1. Train Test Split in Dataset




```
import pandas as pd

dataset = pd.read_csv("Maths.csv")
dataset.head(3)

# Seperate features and target.
input_data = dataset.iloc[:, :-1] # features (x)
output_data = dataset["Total Sum"] # target (y)

# Import train_test_split from sklearn.model_selection
from sklearn.model_selection import train_test_split

# Train test split in database.
x_train, x_test, y_train, y_test = train_test_split(input_
                    data, output_data, test_size = 0.25)

dataset.shape # size of dataset

x_train.shape, x_test.shape # size of train and test for
#input_data.

y_train.shape, y_test.shape # size of train and test for
output_data.
```

2. Regression Analysis

- If the outcome is continous in nature.

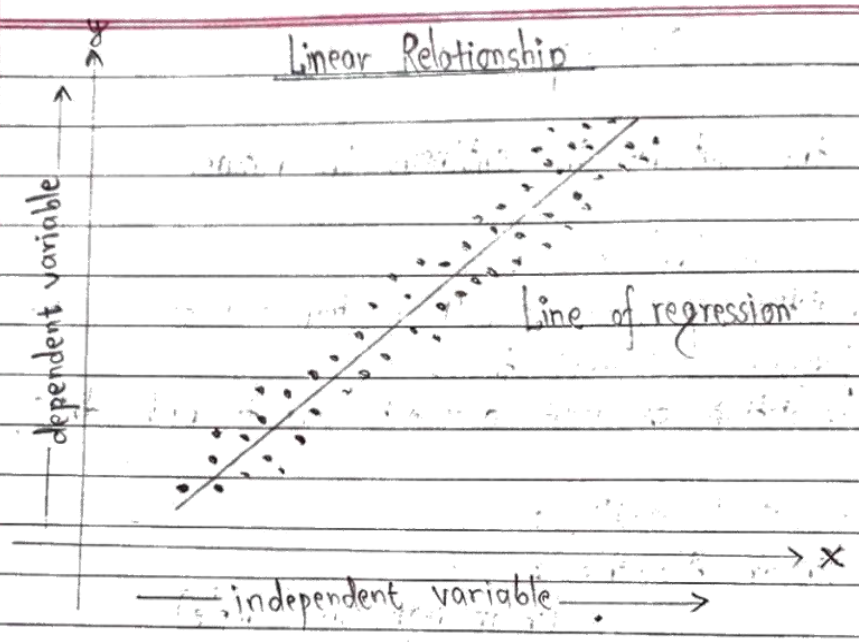
- Real World Applications :
  - Prediction of rain using tempreature and other factors.
  - Determining market trends.
  - Prediction of road accidents due to rash driving

- Types of Recognition :
  1. Linear Algorithms :
     - Linear regression (simple)
     - Multi-linear regression
     - Lasso regression
     - Ridge regression
  2. Non-linear Algorithms :
     - Polynomial regression
     - Decision tree regression
     - Random forest regression
     - Support vector regression
     - K-nearest neighbour regression

3. Linear Regression (Simple)

- Simple Linear Regression is a type of regression algorithms that models the relationship between a dependent variable and a single independent variable.
  Example: scholarship is dependent on marks linearly.

Linear Relationship



Linear Equation:

$$y = mx + c$$

Where,

y = dependent variable
x = independent variable
m = slope / gradient / coefficient
c = intercept

Working Practically in Jupyter:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

data = {
    "Score" : [689, 512, 782, 742, 694, 789, 673, 675,
```

```
        609, 831, 532, 661],
"Prize": [3060, 1980, 3050, 3270, 3370, 2990, 2600,
          2480, 2310, 3510, 1860, 2600],
}
dataset = pd.DataFrame(data)
dataset.head(3)

dataset.isnull.sum() # Check for null values first

x=dataset[["Score"]] # x should be in two dimension
y= dataset["Prize"] # y should be in one dimension.
x.ndim, y.ndim # Shows dimension

# Check if the data follows linear regression by graph
sns.scatterplot(x="Score", y="Prize", data=dataset)
plt.show()

# Train test split
x_train, x_test, y_train, y_test = train_test_split(x, y,
    test_size=0.2, random_state=42)
#random_state ensures that the same random splits are
# generated everytime you run the code.

# Creating a simple linear model
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(x_train, y_train) # Train model on data.

lr.predict([[689]]) # New prediction by model.
```

```
# Check accuracy score
lr.score (x_test, y_test)

# Get coefficient (m) and intercept (c) of our linear
# model in y = mx + c
lr.coef_ , lr.intercept_

# Test the prediction manually if it is true based on
# model.
output = lr.coef_ * 689 + lr.intercept_  #y=mx+c
output  # Output will be same as predicted by model

# Check prediction line in our data by graph
sns.scatterplot (x = "Score" , y = "Prize", data = dataset)
plt.plot (dataset ["Score"], lr.predict (x), color = "red")
plt.legend (["Original Data" , "Prediction Line" ])
# To save graph
plt.savefig ("SavedGraph.jpg")
# To show graph
plt.show()
```

4. Multi-Linear Regression

↳ Multiple linear regression is an extension of simple linear regression as it takes more than one predicator variable to predict the response variable.
Example: Prediction of salary based on age and experience of employee.

↳ Mathematically, in multiple linear regression

$$y = m_1x_1 + m_2x_2 + m_3x_3 + \dots + m_nx_n + c$$

Working practically in Jupyter:

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

data = {
    "Age" : [i for i in range (25,50)],
    "Experience" : [i for i in range (25)],
    "Salary" : [35000+i*1000 for i in range (25)],
}

dataset = pd.DataFrame (data)
dataset.head ()

dataset.shape ()

dataset.isnull ().sum () # Check and fill null values first

#Visualization of data
sns.pairplot (data=dataset)
plt.show ()
"The graphs are highly linear"

sns.heatmap (data=dataset.corr (), annot = True)
plt.show ()
"High correlation is found so, Highly linear"
```

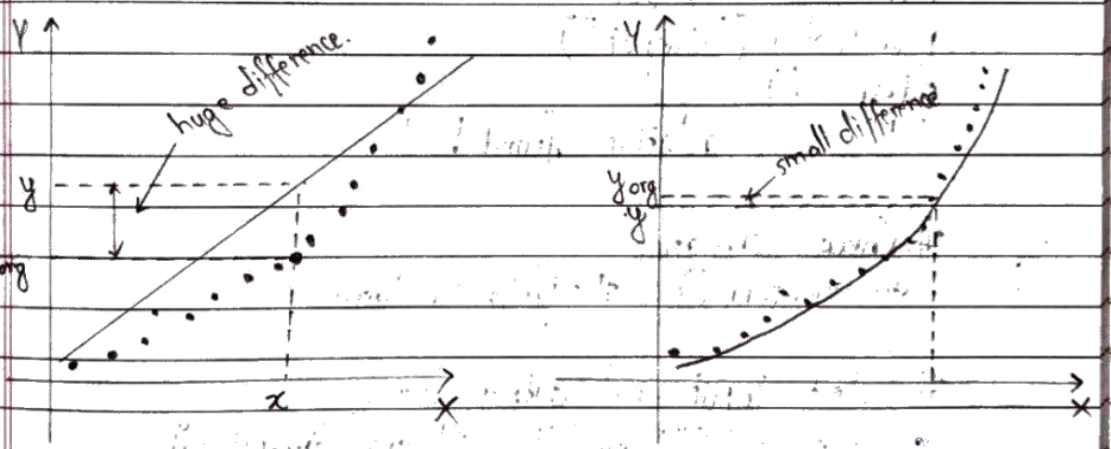
```
# Select input and output
x = dataset.iloc[:, :-1]
y = dataset["Salary"]

# Train test split
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y,
    test_size = 0.2, random_state = 42)

# Build the model
from sklearn.linear_model import LinearRegression

x.ndim  # Two dimensional

lr = LinearRegression()
lr.fit(x_train, y_train)  # y = m1x1 + m2x2 + c

# Test the model
lr.score(x_test, y_test)

# New predictions
lr.predict([[50, 25]])  # [[Age, Experience]]

lr.coef_, lr.intercept_  # y = m1x1 + m2x2 + c
# You will get values of m1, m2 and c.
# By using these values you can also do manual
# calculations.
```

5. Polynomial Regression

- Polynomial regression is a regression algorithm that models that models a relationship between a dependent (y) and independent (x) as n^{th} degree polynomial.

- Mathematically,

$$y = b_0 + b_1 x_1 + b_2 x_1^2 + b_3 x_1^3 + \dots b_n x_1^n$$



Simple linear model

Polynomial model

Working practically in Jupyter:

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

data = {
    "Input" : [i for i in range(100)],
    "Output" : [3*i**2 + 5*i**3 for i in range(100)],
}
```

```
dataset = pd.DataFrame(data)
dataset.head(3)

# Check and remove null values
dataset.isnull().sum()

# Visualize data
plt.scatter(dataset["Input"], dataset["Output"])
plt.xlabel("Input")
plt.ylabel("Output")
plt.show()
"Polynomial relation found!"

# Check correlation
datascore.corr()  # Highly correlated

# Select input and output data.
x = dataset[["Input"]]  # Two dimensional
y = dataset["Output"]  # One dimensional

# Convert the data to polynomial nature.
from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=3)  # degree is max power of x
# which you want to follow.
pf.fit(x)
x = pf.transform(x)  # Returns an array (2D).

# Train test split
from sklearn.model_selection import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split (x, y,
                    test_size = 0.2, random_state = 42)

# Build the polynomial model
from sklearn.linear_model import LinearRegression

lr = LinearRegression ()
lr.fit (x_train, y_train)

# Test the model
lr.score (x_test, y_test)

# New predictions
lr.predict (pf.transform ([[123]])) # New data should be
# in polynomial form

# Visualize prediction line
plt.scatter (dataset ["Input"], dataset ["Output"])
plt.plot (dataset ["Input"], lr.predict (pf.transform (dataset
                    [["Input"]])), color = "red")
plt.xlabel ("Input")
plt.ylabel ("Output")
plt.show ()
```

6. Cost Function

- A cost function is an important parameter that determines how well a machine learning model performs for a given dataset.

Date
Page 38

↳ Cost function is a measure of how wrong the model is in estimating the relationship between x (input) and y (output) parameter.

Types of Cost Function

Regression cost function
Classification cost function
↳ Binary classification cost functions.
↳ Multi-class classification cost functions.

1. Regression Cost Function

Regression models are used to make a prediction for the continous variables.

- MSE (Mean Square Error)
- RMSE (Root Mean Square Error)
- MAE (Mean Absolute Error)
- R^2 Accuracy.

2. Binary Classification Cost Functions

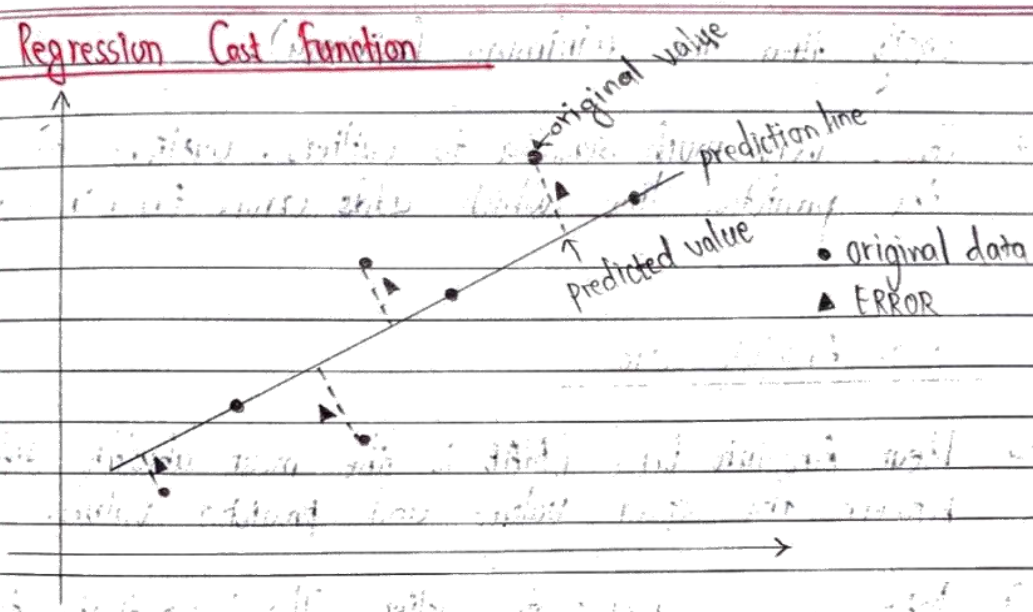
Classification models are used to make predictions of categorical variables such as predictions for 0, or 1, cat or dog, True or false, etc.

3. Multi-class Classification Cost Functions

A multi-class classification cost function is used in the classification problems for which instances are allocated to one of more than two classes.

- Binary Cross Entropy Cost Function or Log Loss Function.

1. Regression Cost Function



Mean Square Error

- Mean Squared Error (MSE) is the mean squared difference between the actual and predicted values. MSE penalizes high errors caused by outliers by squaring the errors.
- Mean Squared Error is also known as L2 loss.
- Mathematically,

$$\text{Mean Squared Error} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where,

y = original value
$\hat{y}$ = predicted value.
n = no. of rows.

- Pros : It is a quardatic differential equation so, we can

easily find the minimum (minima).

- Cons: Very much sensitive to outliers. Outliers can change the prediction line which adds error. Error in square.

Mean Absolute Error

- Mean Absolute Error (MAE is the mean absolute difference between the actual values and predicted values.
- MAE is more robust to outliers. The insensitivity to outliers is because it doesn't penalize high errors caused by outliers.
- Mathematically,

$$\text{Mean Absolute Error} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

where,

y = original value
$\hat{y}$ = predicted value
n = no. of rows.

- Pros: Error in original value. Insensitive to outliers than Mean Square Error.
- Cons: It is not a differential equation as $|x|$ is not differential.

Root Mean Squared Error

- Root Mean Squared Error (RMSE) is the root squared mean of the difference between actual and predicted values.

- RMSE can be used in situations where we want to penalize high errors but not as much as MSE does.

- Mathematically,

$$\text{Root Mean Squared Error} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Finding Best Fit Line (Gradient Descent Technique)

- Gradient Descent is an optimization algorithm used to minimize the cost function in machine learning models. It iteratively adjusts the model parameters in the direction of the negative gradient of the cost function to find optimal set of parameters.

1. **Initialize Parameters:** Start with initial guesses for the model parameters (weights and biases)

2. **Compute the gradient:** Calculate the gradient of the cost function with respect to each parameter. The gradient is a vector of partial derivatives that points in the direction of the steepest ascent of the cost function.

3. Update Parameters: Adjust the parameters in the opposite direction of the gradient.
The update rule is:

$$\theta = \theta - \alpha \nabla J(\theta)$$

where,

θ represents the parameters.
α is learning rate.
$\nabla J(\theta)$ is the gradient of cost function.

The same formula can be written as:

$$m_{new} = m_{old} - \lambda \left(\frac{dL}{dm}\right)$$

where, λ is learning rate
L is cost function

4. Repeat: Repeat the process until the cost function converges to minimum or a predefined number of iteration is reached.

7. Regularization

- This is a form of regression, that constrains / regularizes or shrinks the coefficient estimates towards zero.
- This technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting
- Regularization techniques are:
  1. Lasso regularization / L1
  2. Ridge regularization / L2

1. Lasso Regularization (L1)

- This is a regularization technique used in feature selection using a Shrinkage method also referred to as the penalized regression method.
- Lasso Regression magnitude of coefficients can be exactly zero.
- Mathematically,

  cost function = loss + $\lambda \Sigma |w|$

  where,

  loss = sum of squared residual
  λ = penalty
  w = slope of the curve.

2. Ridge Regularization (L2)

- Ridge regression, also knows as L2 regularization, is an extension to linear regression that introduces a regularization term to reduce model complexity and help prevent overfitting.
- Ridge Regression is working value / magnitude of coefficients is almost equal to zero.
- Mathematically,

  cost function = loss + $\lambda \Sigma |w|^2$

  where,

  loss = sum of residual

λ = penalty
w = slope of the curve.

Working Regularization Practically in Jupyter :

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandScalar
from sklearn.model_selection import train_test_split

dataset = pd.read_csv("Maths.csv")
dataset.head(3)

# Check for null values
dataset.isnull().sum()

# Fill all the null values
for column in dataset.columns:
    dataset[column].fillna(dataset[column].mean(),
                           inplace=True)

# for visualization
sns.heatmap(data = dataset.corr(), annot=True)
plt.show()  # heatmap of correlation among columns.

# Select dependent and independent variables
x = dataset.iloc[:, :-1]
y = dataset["Total Sum"]
```

```
# Standardize features by removing the mean and scaling
# to unit variance.
sc = StandardScalar()
sc.fit(x)
x = pd.DataFrame(sc.transform(x), columns = x.columns)
x.head(5)

# Train test split
x_train, x_test, y_train, y_test = train_test_split(x, y,
                    test_size = 0.2, random_state = 42)

from sklearn.linear_model import LinearRegression, Lasso,
                                 Ridge
from sklearn.metrics import mean_absolute_error,
                            mean_squared_error
import numpy as np
```

Linear Regression :

```
lr = LinearRegression()
lr.fit(x_train, y_train)
lr.score(x_test, y_test)

# View regression cost
print(mean_squared_error(y_test, lr.predict(x_test)))
print(mean_absolute_error(y_test, lr.predict(x_test)))
print(np.sqrt(mean_squared_error(y_test, lr.predict(
                x_test)))) # Root mean squared error.

# Visualize linear regression
plt.figure(figsize = (5,5))
```

```
plt.bar (x.columns , lr.coef_)
plt.title ("Linear Regression")
plt.xlabel (" columns")
plt.ylabel (" coef_ ")
plt.show ()
```

Lasso :

```
la = Lasso (alpha = 0.01)
la.fit (x_train , y_train)
la.score (x_test, y_test)

# View regression cost
print ( mean_squared_error (y_test , la. predict (x_test)))
print ( mean_absolute_error (y_test , la. predict (x_test)))
print (np.sqrt ( mean_squared_error ( y_test , la. predict (
        x_test )))) # Root mean squared error

# Visualize Lasso
plt. figure (figsize = (5, 5))
plt. bar (x.columns , la.coef_)
plt. title ("Lasso")
plt.xlabel (" columns ")
plt. ylabel (" coef_ ")
plt.show()
```

Ridge :

```
ri = Ridge (alpha = 0.01)
ri. fit (x_train , y_train )
ri. score (x_test , y_test )
```

```
# View regression cost
print( mean_squared_error (y_test , la.predict (x_test )))
print ( mean_absolute_error ( y_test , la. predict (x_test )))
print (np.sqrt ('mean_squared_error (y_test , la.predict (
                x_test )))) # Root mean squared error.

# Visualize ridge.
plt.figure (figsize = (5,5))
plt.bar (x.columns, ri.coef_)
plt.title ("Ridge")
plt.xlabel ("columns")
plt.ylabel ("coef_")
plt.show()

# Compare the coef_ of regularizations using Data frame
df = pd. Dataframe (
    {
        "Column Name" : x.columns,
        "Linear Regression" : lr.coef_,
        "Lasso" : la.coef_,
        "Ridge" : ri.coef_,
    }
)
```