ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

AICP408
MACHINE LEARNING
LAB MANUAL
III SEMESTER

Lab In-Charge: Dr. M. KALAISELVI GEETHA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING (Artificial Intelligence and Machine Learning)

VISION

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISSION

M1: Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.

M2: Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment.

M3: Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs.

M4: Expose the students to the emerging technological advancements for meeting the demands of the industry.

B.E. COMPUTER SCIENCE AND ENGINEERING (Artificial Intelligence and Machine Learning)

PROGRAMME EDUCATIONAL OBJECTIVES (PEO)

PEO	PEO Statements
PEO1	To prepare graduates with potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursue higher education and carry out research in the field of Computer Science and Engineering.
PEO3	To equip the graduates with the skills required to stay motivated and adapt to the dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates to communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

AICP408	MACHINE LEARNING LAB	L	T	P	C
		0	0	3	1.5

COURSE OBJECTIVES:

1.To understand the Gaussian densities and its implementation using Python.

2.To implement classification, clustering and regression algorithms in Python.

3.To implement the convolution neural network architecture using Python.

4.To solve the challenging research problems in the area of Speech and Image processing.

LIST OF EXERCISES

1.Linear and logistic regression with error estimation

2.Implementation of univariate and multivariate densities

3.Dimensionality reduction using principal component analysis (PCA)

4.Clustering using

- k-means
- Gaussian mixture modelin

5.Classification using

- Back propagation neural network (BPNN)
- Support vector machine (SVM)

6.Construction of decision tree and random forest

7.Implementation of convolution neural network (CNN)

8.Sequence prediction using recurrent neural network (RNN)

9.Isolated-word speech recognition

10.Face detection and tracking

11.Object recognition

Lab Incharge Head Of the Department

COURSE OUTCOMES:

At the end of this course, the students will be able to

1.Understand the basic concepts of machine learning.

2.Design and implement the classification, clustering and regression algorithms using Python.

3.Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.

Mapping of Course Outcomes with Programme Outcomes												
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	2	2	3	2	-	-	-	-	-	-	-	-
CO2	1	2	-	2	-	-	-	-	-	-	-	-
CO3	2	2	-	1	-	-	-	-	-	2	-	2

Rubric for CO3

Rubric for CO3 in Laboratory Courses				
Rubric	**Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks**			
	Up To 2.5 Marks	**Up To 5 Marks**	**Up To 7.5 Marks**	**Up To 10 marks**
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.

1. a) Linear and Logistic Regression with Error Estimates

Linear Regression model establish a linear relationship between the input variables(**X**) and single output variable(**Y**). When the input(**X**) is a single variable this model is called **Simple Linear Regression** and when there are mutiple input variables(**X**), it is called **Multiple Linear Regression**.

Simple Linear Regression Model Representation

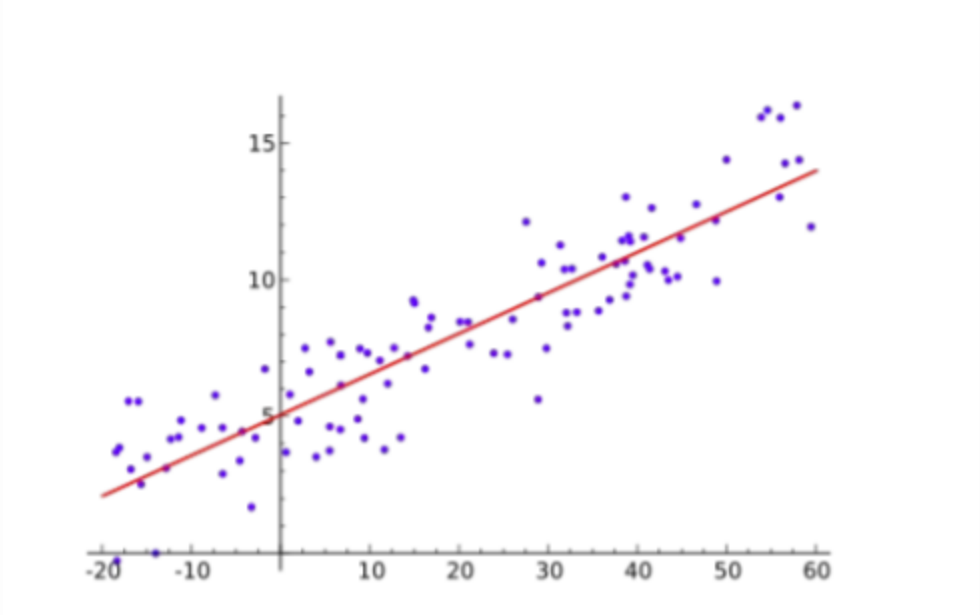
From the Dataset we have an input variable - **X** and one output variable - **Y**. And we want to build linear relationship between these variables.

$$Y = \beta_0 + \beta_1$$

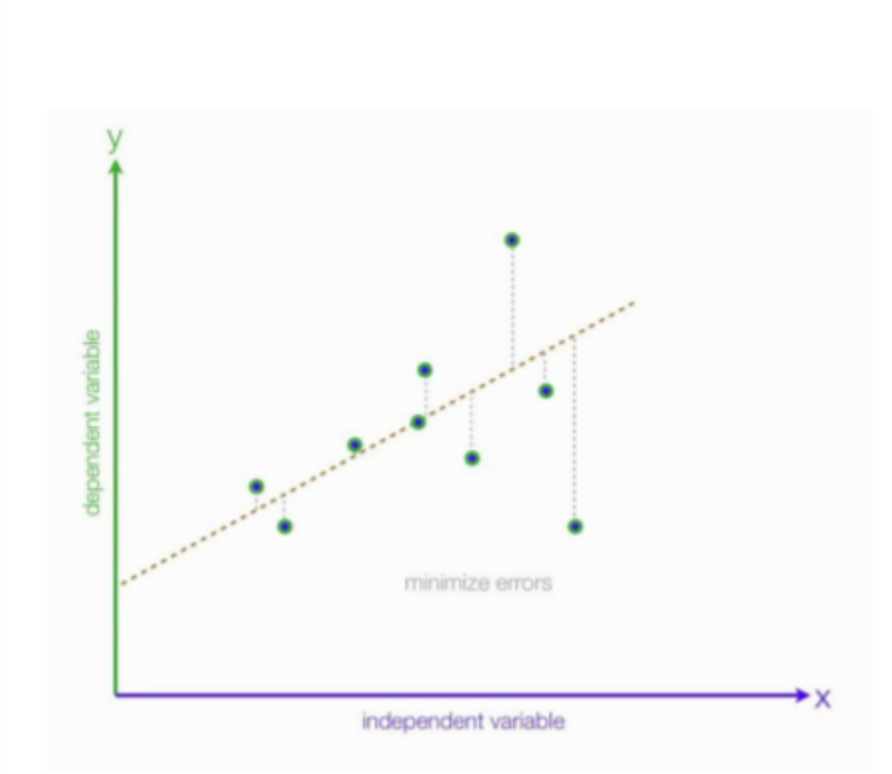
The β_1 is called a scale factor or **coefficient** and β_0 is called **bias coefficient**. The bias coeffient gives an extra degree of freedom to this model. This equation is similar to the line equation $y = mx + c$ with $m = \beta_1$(**Slope**) and $c = \beta_0$ (**Intercept**).
So in this Simple Linear Regression model we want to draw a line between X and Y which estimates the relationship between X and Y. Finding these coefficients is the learning procedure. We can find these using different approaches. One is called **Ordinary Least Square Method** and other one is called **Gradient Descent Approach**. We will use Ordinary Least Square Method in Simple Linear Regression

Ordinary Least Square Method

we are going to approximate the relationship between X and Y to a line. Let's say we have few inputs and outputs. And we plot these scatter points in 2D space, we will get something like the following image.



A line in the image. That's what we are going to accomplish. And we want to minimize the error of our model. A good model will always have least error. We can find this line by reducing the error. The error of each point is the distance between line and that point. This is illustrated as follows.



vs Dependent

And total error of this model is the sum of all errors of each point. ie
where, - Distance between line and ith point. - Total number of points
You might have noticed that we are squaring each of the distances. This is because, some points will be above the line and some points will be below the line. We can minimize the error in the model by minimizing D. And after the mathematics of minimizing D, we will get;
In these equations $\bar{x}$ is the mean value of input variable X and $\bar{y}$ is the mean value of output variable Y. Now we have the model. This method is called **Ordinary Least Square Method.**

Evaluation of the Regression model:

In-order to find how good is our model we go for RMSE evaluation. There are many methods to evaluate models. We will use **Root Mean Squared Error** and **Coefficient of Determination (R^2 Score)**. Root Mean Squared Error is the square root of sum of all errors divided by number of values.
In Mathematical expression,

$$\text{RMSE} = \sqrt{\frac{1}{m}\Sigma_{x=i}^{m}(\hat{yi} - yi)^2}$$

R^2-Score:
SS_t is the total sum of squares
$SS_t = \Sigma_{i=1}^{m}(yi - y)^2$
And SS_r is the total sum of squares of residuals
$SS_r = \Sigma_{i=1}^{m}(y_i - y)^2$
R^2 score has expressed as
$R^2 = 1 - \frac{SS_r}{SS_t}$
R^2 Score usually range from 0 to 1. It will also become negative if the model is completely wrong.
With this theoritical background,we will implement the Regression model in Python.

1. b) Logistic Regression

Logistic regression is a fundamental classification technique. It belongs to the group of **linear classifiers** and is somewhat similar to polynomial and **linear regression**. Logistic regression is fast and relatively uncomplicated, and it's convenient for us to interpret the results. Although it's essentially a method for binary classification, it can also be applied to multiclass problems.

The nature of the dependent variables differentiates regression and classification problems. **Regression** problems have continuous and usually unbounded outputs. An example is when you're estimating the salary as a function of experience and education level. On the other hand, **classification** problems have discrete and finite outputs called **classes** or **categories**.

Objective:

Our goal is to find the **logistic regression function** $p(\mathbf{x})$ such that the **predicted responses** $p(\mathbf{x}_i)$ are as close as possible to the **actual response** y_i for each observation $i = 1, \ldots, n$. Remember that the actual response can be only 0 or 1 in binary classification problems. This means that each $p(\mathbf{x}_i)$ should be close to either 0 or 1. That's why it's convenient to use the sigmoid function. Once we have the logistic regression function $p(\mathbf{x})$, we can use it to predict the outputs for new and unseen inputs, assuming that the underlying mathematical dependence is unchanged.

Theory:

Mathematically, a binary logistic model has a target variable which is dichotomous in nature i.e with two possible values, labeled "0" or "1". In the logistic model, the log-odds (the logarithm of the odds) for the value labeled "1" is a linear combination of one or more independent variables ("predictors"); the independent variables can each be a binary variable or a continuous variable (any real value).

Logistic Regression uses a log of odds as the dependent variable. It predicts the probability of occurrence of a binary event utilizing a logit function.

General Linear Regression Equation:

Where, y is dependent variable and $X_1, X_2, \ldots, X_n$ are explanatory variables.

Sigmoid Function

The sigmoid function, also called logistic function gives an 'S' shaped curve that can take any real-valued number and map it into a value between 0 and 1. If the curve goes to positive infinity, y predicted will become 1, and if the curve goes to negative infinity, y predicted will become 0. If the output of the sigmoid function is more than 0.5, we can classify the outcome as 1 or YES, and if it is less than 0.5, we can classify it as 0 or NO. If the output is 0.75, we can say in terms of probability as: There is a 75 percent chance that desired event will occur.

$$p = \frac{1}{(1 + e^{-y})}$$

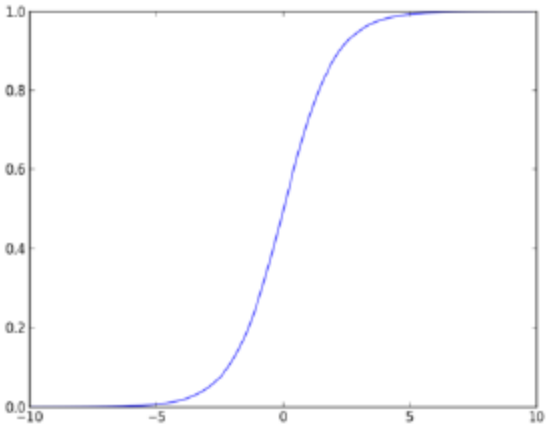


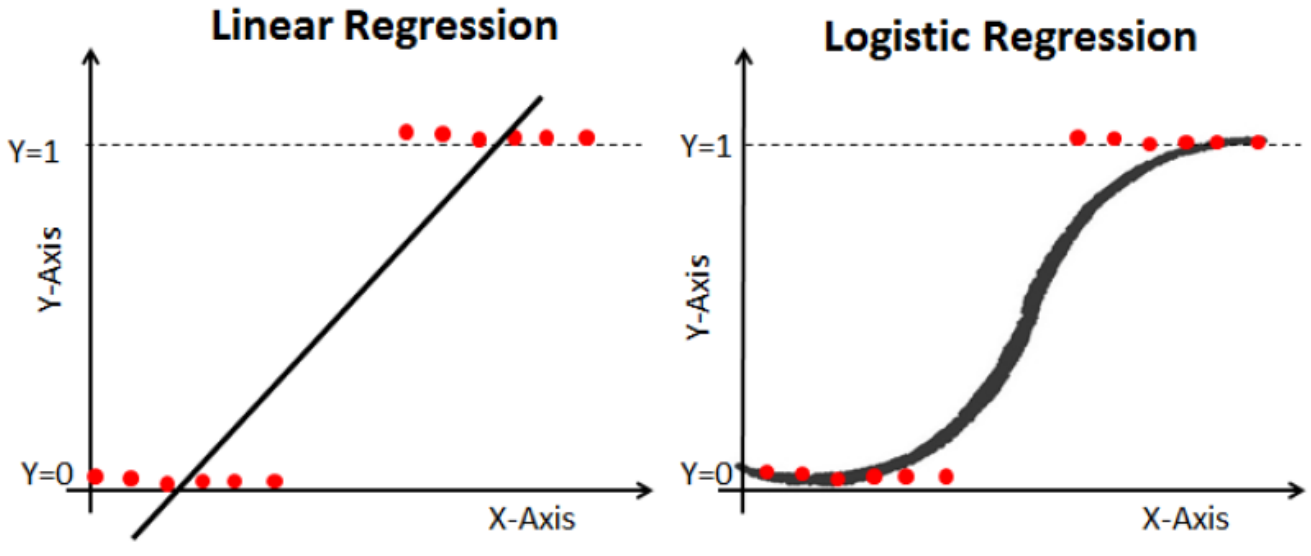
Fig.1. Sigmoid Function

Apply Sigmoid function on linear regression we get:

$$p = 1/1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n Xn)}$$

Properties of Logistic Regression:

- The dependent variable in logistic regression follows Bernoulli Distribution. Linear regression is estimated using Ordinary Least Squares (OLS) while logistic regression is estimated using Maximum Likelihood Estimation (MLE) approach.

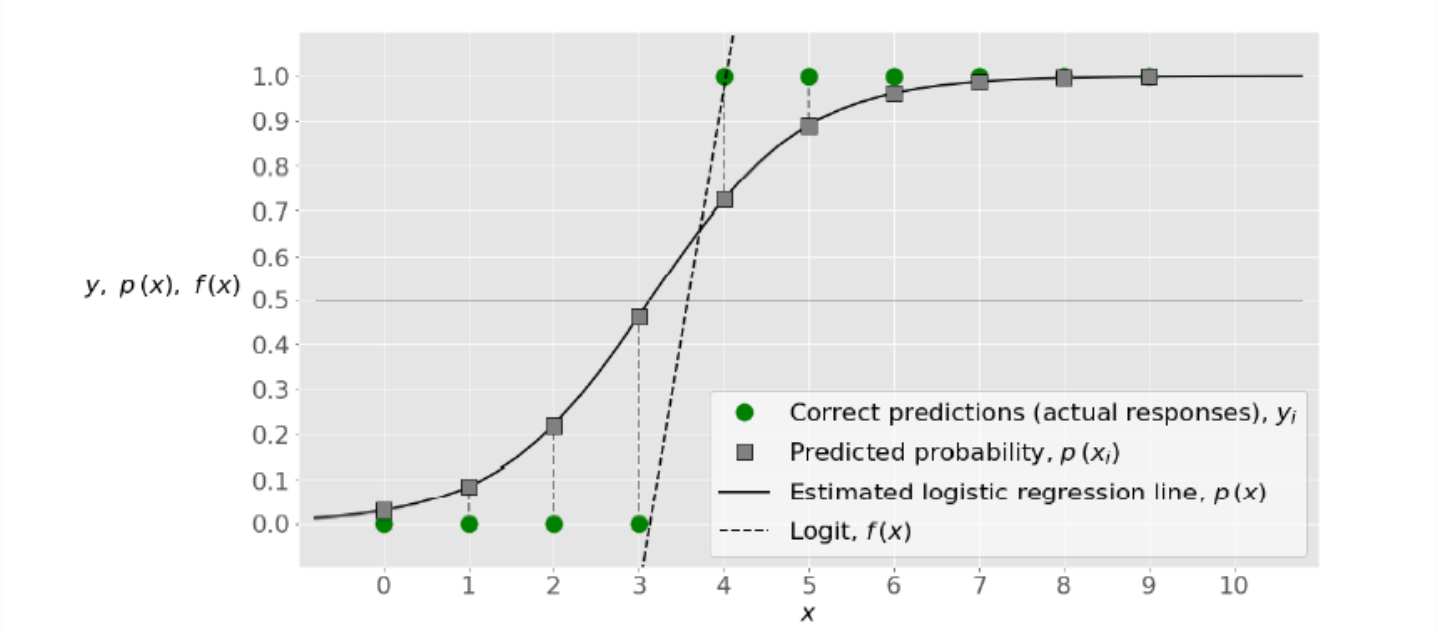


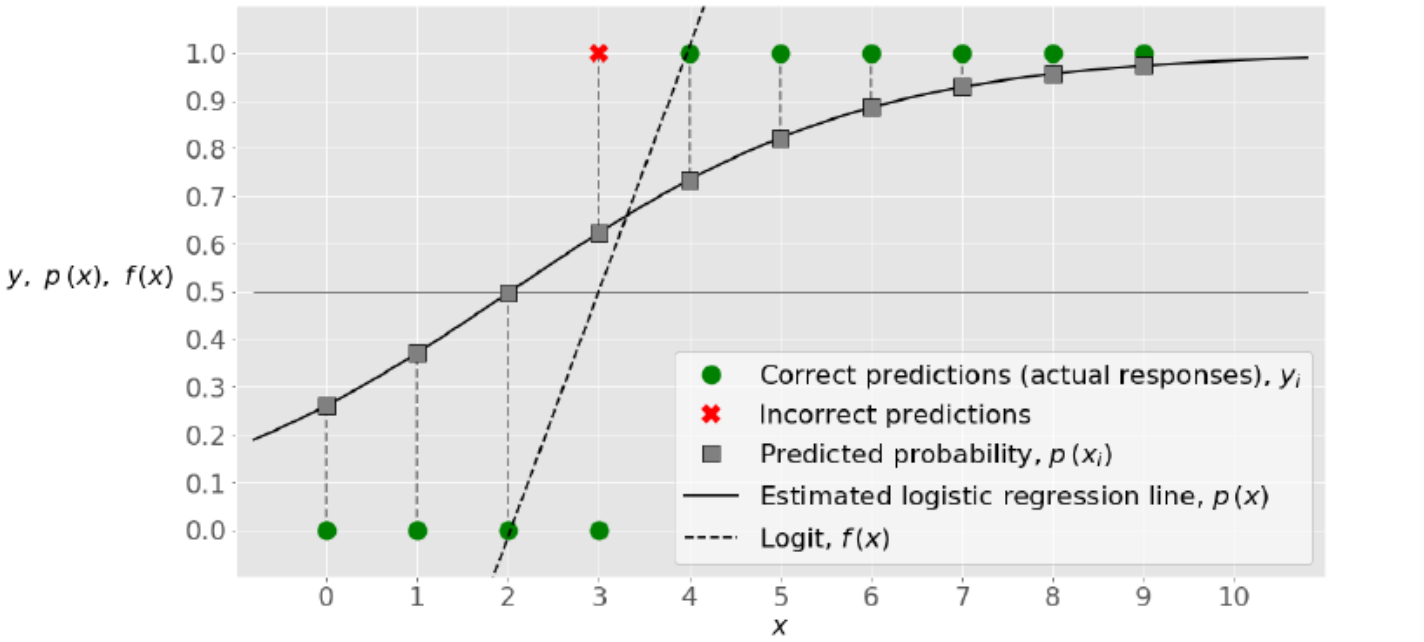
Maximum Likelihood Estimation Vs. Least Square Method

The MLE is a "likelihood" maximization method, while OLS is a distance-minimizing approximation method. Maximizing the likelihood function determines the parameters that are most likely to produce the observed data. From a statistical point of view, MLE sets the mean and variance as parameters in determining the specific parametric values for a given model. This set of parameters can be used for predicting the data needed in a normal distribution.
Ordinary Least squares estimates are computed by fitting a regression line on given data points that has the minimum sum of the squared deviations (least square error). Both are used to estimate the parameters of a linear regression model. MLE assumes a joint probability mass function, while OLS doesn't require any stochastic assumptions for minimizing distance.

Methodology

Logistic regression linear function $y = f(\mathbf{x}) = b_0 + b_1x_1 + \ldots + b_rx_r$, also called the **logit**. The variables $b_0, b_1, \ldots, b_r$ are the **estimators** of the regression coefficients, which are also called the **predicted weights** or just **coefficients**.
The logistic regression function $p(\mathbf{x})$ is the sigmoid function of $f(\mathbf{x})$: $p(\mathbf{x}) = 1/(1 + \exp(-f(\mathbf{x})))$. As such, it's often close to either 0 or 1. The function $p(\mathbf{x})$ is often interpreted as the predicted probability that the output for a given **x** is equal to 1. Therefore, $1 - p(\mathbf{x})$ is the probability that the output is 0.
Logistic regression determines the best predicted weights $b_0, b_1, \ldots, b_r$, such that the function $p(\mathbf{x})$ is as close as possible to all actual responses $y_i\ i = 1, \ldots, n$. where n is the number of observations. The process of calculating the best weights using available observations is called **model training** or **fitting**.
To get the best weights, we usually maximize the **log-likelihood function (LLF)** for all observations $i = 1, \ldots, n$. This method is called **maximum likelihood estimation** and is represented by the equation
$\text{LLF} = \sum_i(y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i)))$.
When $y_i = 0$, the LLF for the corresponding observation is equal to $\log(1 - p(\mathbf{x}_i))$. If $p(\mathbf{x}_i)$ is close $y_i = 0$, then $\log(1 - p(\mathbf{x}_i))$ is close to 0. This is the result you want. If $p(\mathbf{x}_i)$ is far from 0, then $\log(1 - p(\mathbf{x}_i))$ drops significantly. You don't want that result because your goal is to obtain the maximum LLF. Similarly, when $y_i = 1$, the LLF for that observation is $y_i \log(p(\mathbf{x}_i))$. If $p(\mathbf{x}_i)$ is close to $y_i = 1$, then $\log(p(\mathbf{x}_i))$ is close to 0.If $p(\mathbf{x}_i)$ is far from 1, then $log(p(\mathbf{x}_i))$ is a large negative number.
There are several mathematical approaches that will calculate the best weights that correspond to the maximum LLF, but that's beyond the scope.





Logistic Regression in Python With scikit-learn: 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

scikit-learn is a free machine learning library for Python.

Seaborn is a library for making statistical graphics in Python. It builds on top of matplotlib and integrates closely with pandas data structures. Seaborn helps you explore and understand your data. Its plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots. **Model Development and Prediction**

Import the Logistic Regression module and create a Logistic Regression classifier object using LogisticRegression() function.Then, fit your model on the train set using fit() and perform prediction on the test set using predict()

Model Evaluation using Confusion Matrix

A confusion matrix is a table that is used to evaluate the performance of a classification model. You can also visualize the performance of an algorithm. The fundamental of a confusion matrix is the number of correct and incorrect predictions are summed up class-wise.

Exercise 1 - Linear and Logistic Regression with Error Estimation

a) Linear regression

Program 1 - Implementation of Linear Regression (From scratch and using Sklearn)

AIM:

To implement linear regression in python from scratch and using scikit learn

FORMULA:

Equation of the regression line, $y = b_0 + b_1 x$

$$b_1 = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m}(x_i - \bar{x})^2} = \frac{(\mathbf{x} - \bar{x})^T(\mathbf{y} - \bar{y})}{(\mathbf{x} - \bar{x})^T(\mathbf{x} - \bar{x})}$$

$$b_0 = \bar{y} - b_1\bar{x}$$

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2} = \sqrt{\frac{(\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}})}{m}}$$

$$\text{SS}_{\text{tot}} = \sum_{i=1}^{m}(y_i - \bar{y})^2 = (\mathbf{y} - \bar{y})^T(\mathbf{y} - \bar{y})$$

$$\text{SS}_{\text{res}} = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = (\mathbf{y} - \hat{\mathbf{y}})^T(\mathbf{y} - \hat{\mathbf{y}})$$

$$R^2\text{score} = 1 - \frac{\text{SS}_{\text{res}}}{\text{SS}_{\text{tot}}}$$

where,
$\mathbf{x} = (x_1, x_2, \ldots, x_n)$ - input vector
$\mathbf{y} = (y_1, y_2, \ldots, y_n)$ - output vector
$\bar{x}$ - mean of $\mathbf{x}$
$\bar{y}$ - mean of $\mathbf{y}$
$\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n)$ - Predicted output vector

Part 1 - Importing modules and defining class for Linear Regression

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

```
class LinearRegression:
    def fit(self,X,y):
        m = X.shape[0]
        X_mean, y_mean = np.mean(X), np.mean(y)
        X_mean_diff, y_mean_diff = X-X_mean, y-y_mean
        self.b1 = (X_mean_diff @ y_mean_diff) / (X_mean_diff @ X_mean_diff)
        self.b0 = y_mean - (self.b1 * X_mean)
        print(f"(b0,b1):({self.b0:.3f},{self.b1:.3f})")
        return self

    def predict(self,X):
        return self.b0 + X*self.b1

    def evaluate(self,X,y):
        y_pred = self.predict(X)
        y_diff,y_mean_diff  = y-y_pred , y-np.mean(y)
        rmse = np.sqrt(y_diff @ y_diff/X.shape[0])
        ss_tot = y_mean_diff @ y_mean_diff
        ss_res = y_diff @ y_diff
        r2 = 1 - ss_res/ss_tot
        print("Root mean squared Error:",rmse)
        print("R^2 value:",r2)
```

Part 2 - Ploting function for regression

```
def regression_plot(X,y,model,title=""):
    plt.figure(figsize=(14,7))
    plt.title(title)
    plt.xlabel("Head Size(cm^3)")
    plt.ylabel("Brain Weights(grams)")

    x_line = np.array([np.min(X) - 100,np.max(X) + 100]).reshape(-1,1)
    y_line = model.predict(x_line)

    plt.scatter(X, y,c='orange', label='Original Data Points')
    plt.plot(x_line, y_line,linewidth=4, label='Regression Line')
    plt.legend()
```

Part 3 - Loading and processing the dataset

```
data = pd.read_csv('./datasets/headbrain.csv')
print("size:",data.size,"; shape",data.shape)
data.head()
```

Output:

```
size: 948 ; shape (237, 4)
```

	Gender	Age Range	Head Size(cm^3)	Brain Weight(grams)
0	1	1	4512	1530
1	1	1	3738	1297
2	1	1	4261	1335
3	1	1	3777	1282
4	1	1	4177	1590

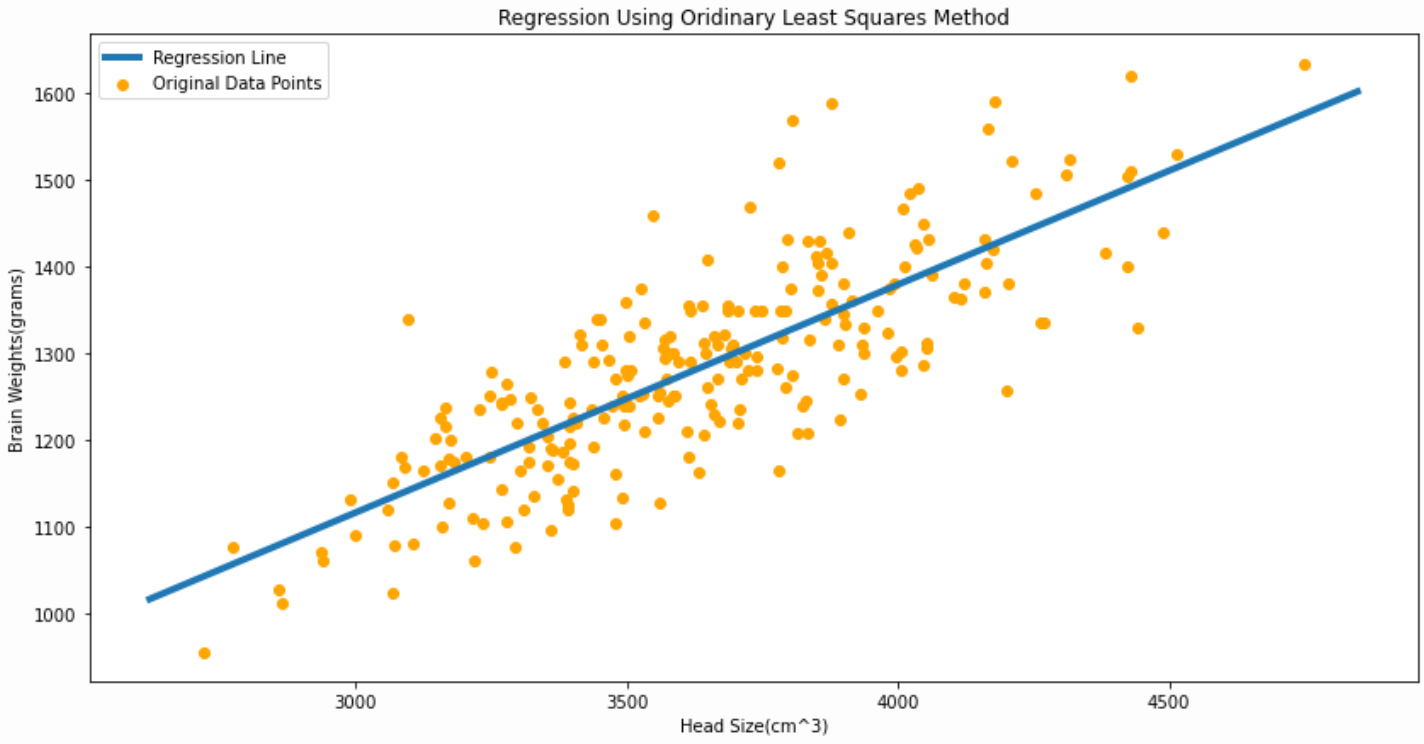
Part 4 - Implementing Linear Regression from scratch

```
X = data['Head Size(cm^3)'].values
y = data['Brain Weight(grams)'].values
```

```
lin_reg_model= LinearRegression()
lin_reg_model.fit(X,y)
regression_plot(X,y,lin_reg_model,title="Regression Using Oridinary Least Squares Method")
lin_reg_model.evaluate(X,y)
```

Output:

```
(b0,b1):(325.573,0.263)
Root mean squared Error: 72.1206213783709
R^2 value: 0.639311719957
```



Part 5 - Implementing Linear Regression using Scikit Learn

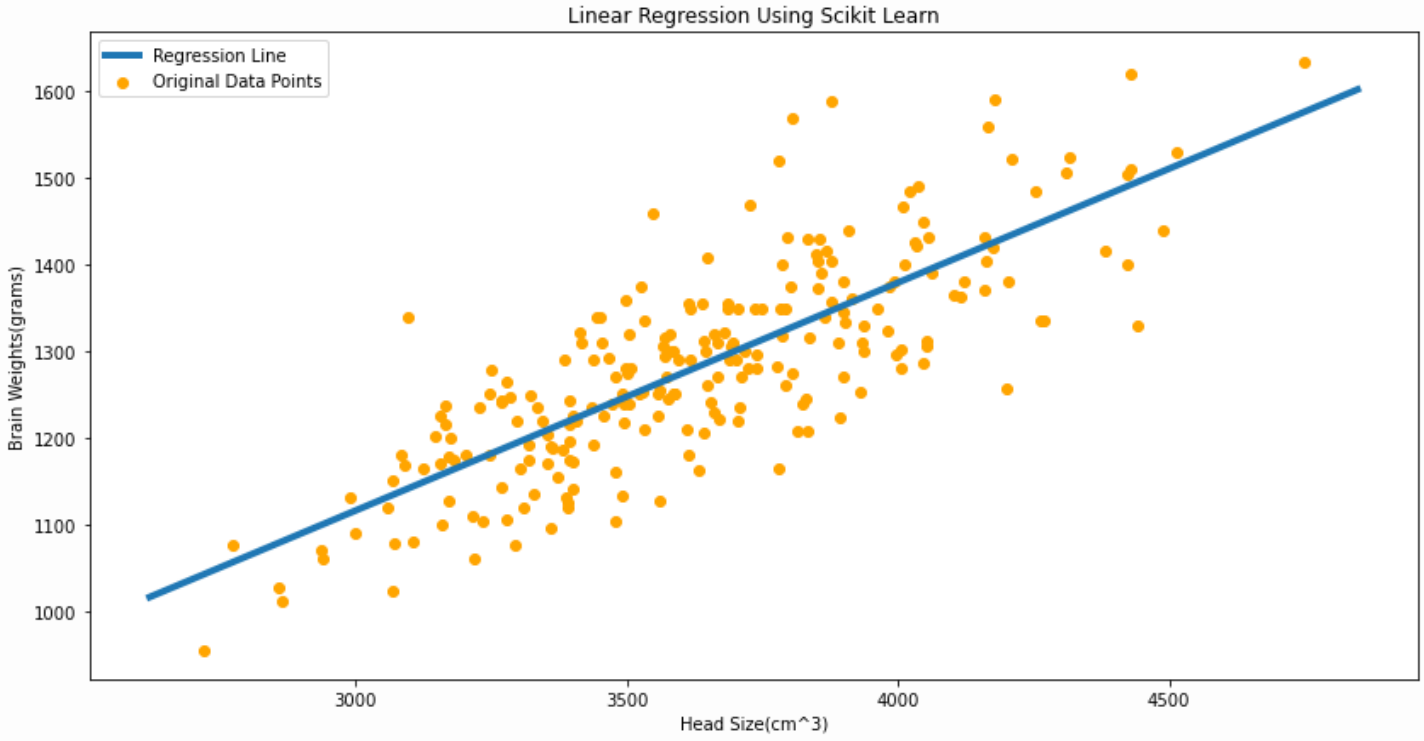
```
from sklearn.linear_model import LinearRegression as SkLinearRegression
from sklearn.metrics import mean_squared_error
```

```
# sklearn requires 2d array for X values
# thus we reshape X to X1 as follows
X1 = X.reshape(-1,1)
```

```
sk_lin_reg_model = SkLinearRegression().fit(X1, y)

regression_plot(X1,y,sk_lin_reg_model,title="Linear Regression Using Scikit Learn")

y_hat = sk_lin_reg_model.predict(X1)
rmse = np.sqrt(mean_squared_error(y, y_hat))
r2_score = sk_lin_reg_model.score(X1, y)
print("Root Mean Squared Error:",rmse)
print("R^2 value:",r2_score)
```

Output:

```
Root Mean Squared Error: 72.1206213783709
R^2 value: 0.639311719957
```



b) Logistic Regression

Program 1 - Preprocessing and implementing Logistic Regression on titanic dataset using Scikit Learn

AIM:

To preprocess and implement Logistic Regression in titanic dataset using Scikit Learn

Part 1 - Importing modules and Loading the dataset

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

titanic_df = pd.read_csv('./datasets/titanic.csv')
titanic_df.head()
```

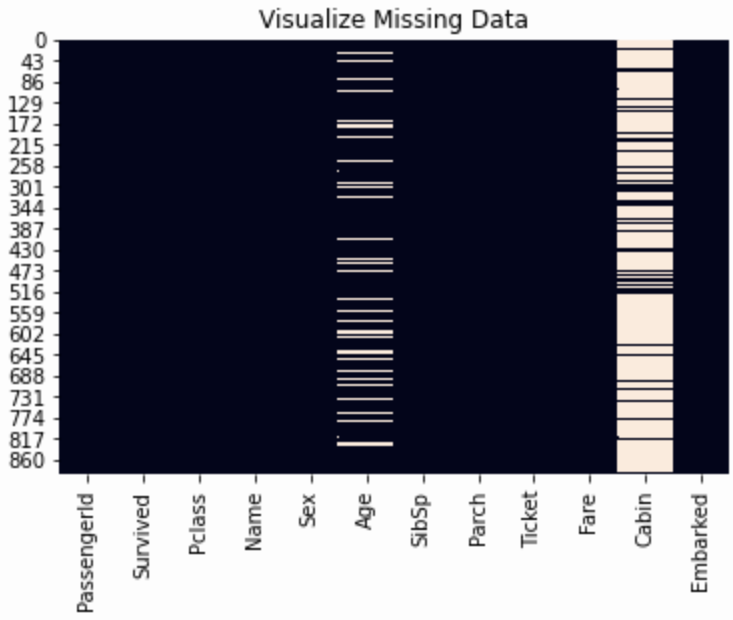
Output:

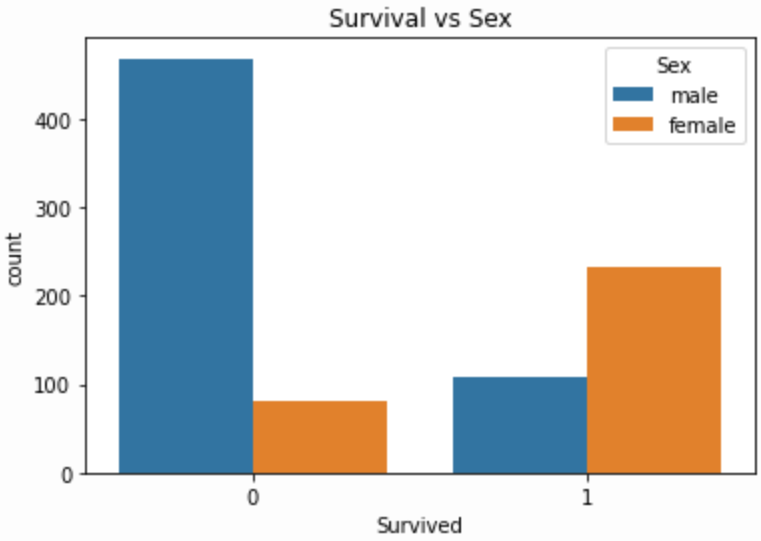
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

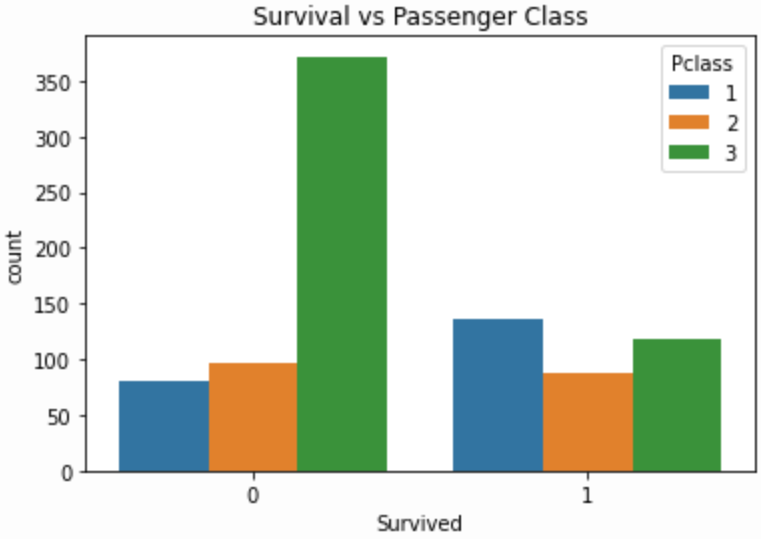
Part 2 - Visualizing the dataset

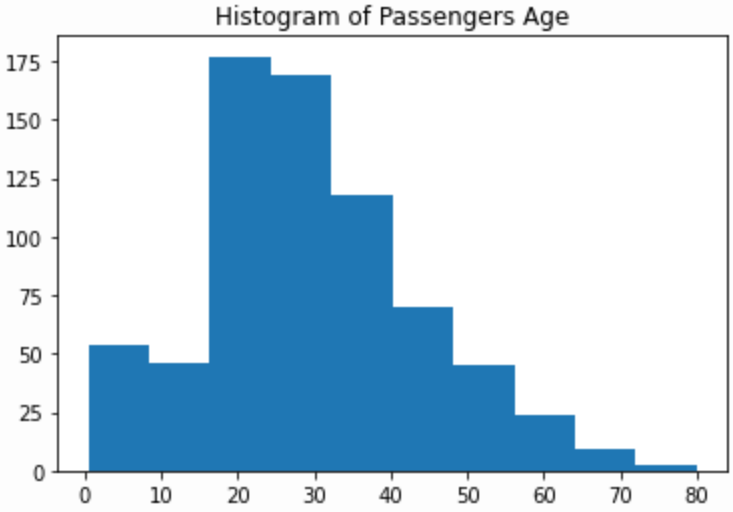
```
def plot(title,plot_func,*args,**kwargs):
    plt.title(title)
    plot_func(*args,**kwargs)
    plt.show()

plot('Visualize Missing Data',
   sns.heatmap,titanic_df.isnull(), cbar=False)
plot("Survival vs Sex",
    sns.countplot,x='Survived', hue='Sex', data=titanic_df)
plot("Survival vs Passenger Class",
    sns.countplot,x='Survived', hue='Pclass', data=titanic_df)
plot("Histogram of Passengers Age",
    plt.hist,titanic_df["Age"].dropna())
plot("Histogram of ship Fare",
    plt.hist,titanic_df['Fare'])
plot("Passenger class vs Age",
    sns.boxplot,x='Pclass', y='Age',data=titanic_df)
```

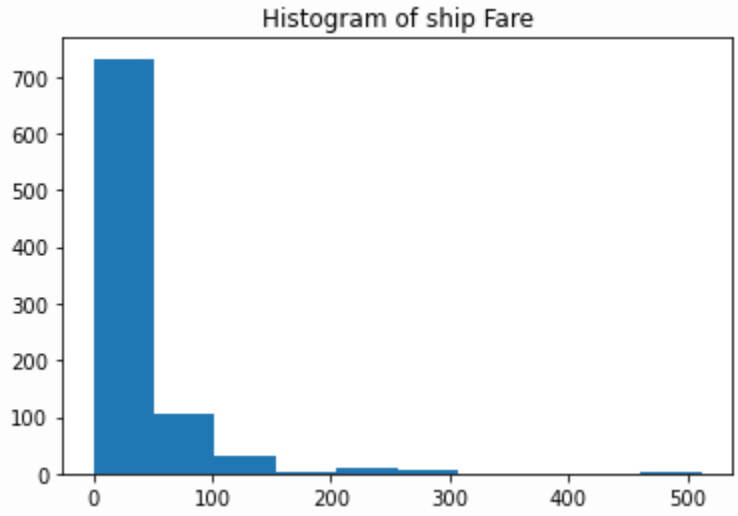
Output:

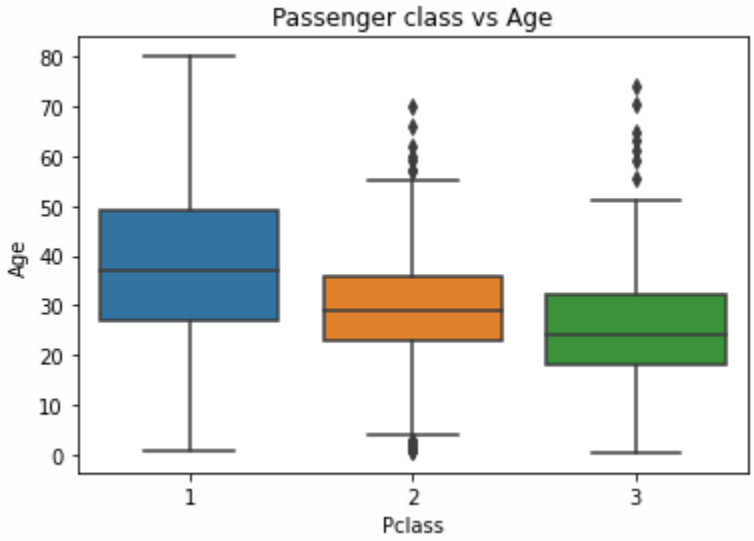



Survival vs Sex
Sex
male
female
count
Survived
0
1
0
100
200
300
400


Survival vs Passenger Class
Pclass
1
2
3
count
Survived
0
1
0
50
100
150
200
250
300
350


Histogram of Passengers Age
0
25
50
75
100
125
150
175
0
10
20
30
40
50
60
70
80

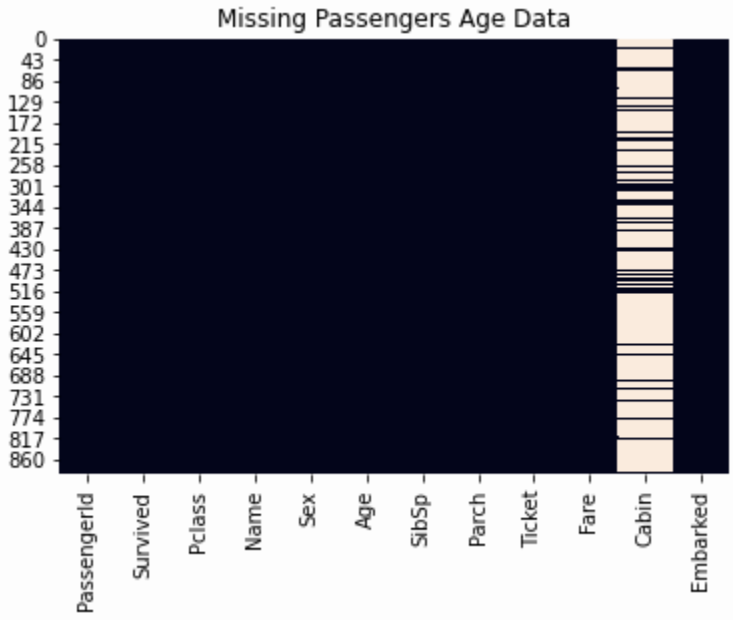




Part 3 - Dealing with missing, categorial and irrelevant data

```
mean_ages = {
    p_class:titanic_df[titanic_df["Pclass"]==p_class]["Age"].mean()
    for p_class in titanic_df["Pclass"].unique()
}

def impute_missing_age(columns):
    age , p_class = columns
    if pd.isnull(age):
        return mean_ages[p_class]
    return age

titanic_df['Age'] = titanic_df[['Age', 'Pclass']].apply(
    impute_missing_age,axis = 1
)
plot("Missing Passengers Age Data",
    sns.heatmap,titanic_df.isnull(), cbar=False)
```

Output:




```
titanic_df.drop('Cabin', axis=1, inplace = True)
titanic_df.dropna(inplace = True)
sex_data = pd.get_dummies(titanic_df['Sex'], drop_first = True)
embarked_data = pd.get_dummies(titanic_df['Embarked'], drop_first = True)
titanic_df = pd.concat([titanic_df, sex_data, embarked_data], axis = 1)
titanic_df.drop(
    ['Name', 'PassengerId', 'Ticket', 'Sex', 'Embarked'],
    axis =1, inplace = True
)
titanic_df.head()
```

Output:

	Survived	Pclass	Age	SibSp	Parch	Fare	male	Q	S
0	0	3	22.0	1	0	7.2500	1	0	1
1	1	1	38.0	1	0	71.2833	0	0	0
2	1	3	26.0	0	0	7.9250	0	0	1
3	1	1	35.0	1	0	53.1000	0	0	1
4	0	3	35.0	0	0	8.0500	1	0	1

Part 4 - Implementing Logistic Regression using Scikit Learn

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import minmax_scale

titanic_df[["Age","Fare"]] = minmax_scale(titanic_df[["Age","Fare"]])

X = titanic_df.drop('Survived', axis = 1)
y = titanic_df['Survived']
X_train, X_test, y_train, y_test =(
    train_test_split(X, y, test_size = 0.3)
)
model = LogisticRegression()
model.fit(X_train, y_train)
y_hat = model.predict(X_test)

def print_title(title): print(f"{title:^50}\n{'='*50}")

print_title("Classification Report")
print(classification_report(y_test, y_hat))
print_title("Confusion Matrix")
print(confusion_matrix(y_test, y_hat))
```

Output:

```
              Classification Report
==================================================
              precision    recall  f1-score   support

           0       0.84      0.87      0.85       178
           1       0.71      0.67      0.69        89

    accuracy                           0.80       267
   macro avg       0.78      0.77      0.77       267
weighted avg       0.80      0.80      0.80       267

                 Confusion Matrix
==================================================
[[154  24]
 [ 29  60]]
```

Program 2 - Logistic regression for diabetes prediction(From scrath vs using Scikit Learn)

AIM:

To implement Logistic regression using Gradient descent from scratch and compare it with Logistic regression with Scikit Learn

FORMULA:

$$\text{Log-likelihood}, LL = \sum_{i=1}^{m} y_i log(p_i) + (1 - y_i) log(1 - p_i)$$

where,

$$p_i = p(\mathbf{x}_i, \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}}$$

$\mathbf{w} = (w_0, w_1, \ldots, w_n)$ - weight vector
$\mathbf{x}_i = (1, x_1, x_2, \ldots, x_n)$ - ith input vector
To find $\mathbf{w}$ such that,

$$\max_{\mathbf{w}} LL = \min_{\mathbf{w}} (-LL)$$

gradient of $-LL$,

$$\frac{d(-LL)}{d\mathbf{w}} = \sum_{i=1}^{m} (p_i - y_i) \mathbf{x}_i$$

weight update step is given by,

$$\mathbf{w} := \mathbf{w} - \alpha \frac{d(-LL)}{d\mathbf{w}} = \mathbf{w} - \alpha \sum_{i=1}^{m} (p_i - y_i) \mathbf{x}_i = \mathbf{w} - \alpha (\mathbf{p} - \mathbf{y})^T X$$

where α - learning rate

Part 1 - Defining Class for Logistic Regression

```
import numpy as np
import pandas as pd
```

```
class LogitRegression() :
    def __init__( self, learning_rate, iterations) :
        self.learning_rate = learning_rate
        self.iterations = iterations

    def p(self,X):
        return 1/(1+np.exp(-(X @ self.w)))

    def fit(self, X, y) :
        m,n = X.shape
        X = np.hstack([np.ones((m,1)),X])
        y = y.squeeze()
        self.w = np.zeros(n+1)

        for i in range(self.iterations) :
            self.w = self.w - self.learning_rate * ((self.p(X)-y) @ X)

    def predict(self, X) :
        m = X.shape[0]
        X = np.hstack([np.ones((m,1)),X])
        y_hat = np.where( self.p(X) > 0.5, 1, 0 )
        return y_hat
```

Part 2 - Loading and Processing Dataset

```
from sklearn.preprocessing import minmax_scale
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
diabetes_df = pd.read_csv( "./datasets/diabetes.csv" )
X = minmax_scale(diabetes_df.iloc[:,:-1].values)
y = diabetes_df.iloc[:,-1:].values.reshape(-1)
X_train, X_test, y_train, y_test =train_test_split(
    X, y, test_size = 1/3, random_state =6
)
```

Part 3 - Comparing Models

```
models = [
    LogitRegression(learning_rate = .1, iterations = 1000),
    LogisticRegression()
]
for model in models:
    model.fit(X_train,y_train)

def compute_accuracy(model,X_test,y_test):
    y_hat = model.predict(X_test)
    return (y_hat==y_test).mean() * 100

print("Accuracy on test set by our implementation of Logistic Reg model :",
    compute_accuracy(models[0],X_test,y_test)
)
print("Accuracy on test set by sklearn model :",
    compute_accuracy(models[1],X_test,y_test)
)
```

Output:

```
Accuracy on test set by our implementation of Logistic Reg model : 64.0625
Accuracy on test set by sklearn model : 78.515625
```

2. Implementation of Univariate Multivariate Gaussian Density

Normal Distribution

Normal distribution or Gaussian distribution is a continuous probability distribution that describes data that cluster around a mean or average. The graph of the associated probability density function is bell-shaped, with a peak at the mean, and is known as the Gaussian function or bell curve. The variable is distributed normally with mean and variance. It can be categories into 1. Univariate Density: It involves single variable (one dimension) Example: Height of person 2. Multivariate Density: It involves multivariable (two or more dimensions) Example: Height and weight of person

Univariate Density

Probability density function for univariate density is written as

$$p(x) = \frac{1}{\sqrt{2}pi\sigma}\exp\left[-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2\right]$$

where μ is mean, σ^2 is variance

Mean (μ) **:**

Mean is the average of the given feature and it is given by

$$\mu = \frac{\sum x}{n}$$

Variance (σ^2) **:**

The spread of the data can be measured using variance.

$$\sigma^2 = \frac{\sum(x-\mu)^2}{n}$$

Example: Measure of the variability of the heights of the person. - The normal density is traditionally described as a bell-shaped curve and is shown in Fig1. - The normal distribution is symmetrical about mean *(μ).* - Peak of the univariate normal distribution occurs $x = \mu$ at and its value is $1/\sqrt{2\pi\sigma^2}$ which is shown in Fig.1. - Width of the univariate normal distribution is proportional to standard deviation (σ).

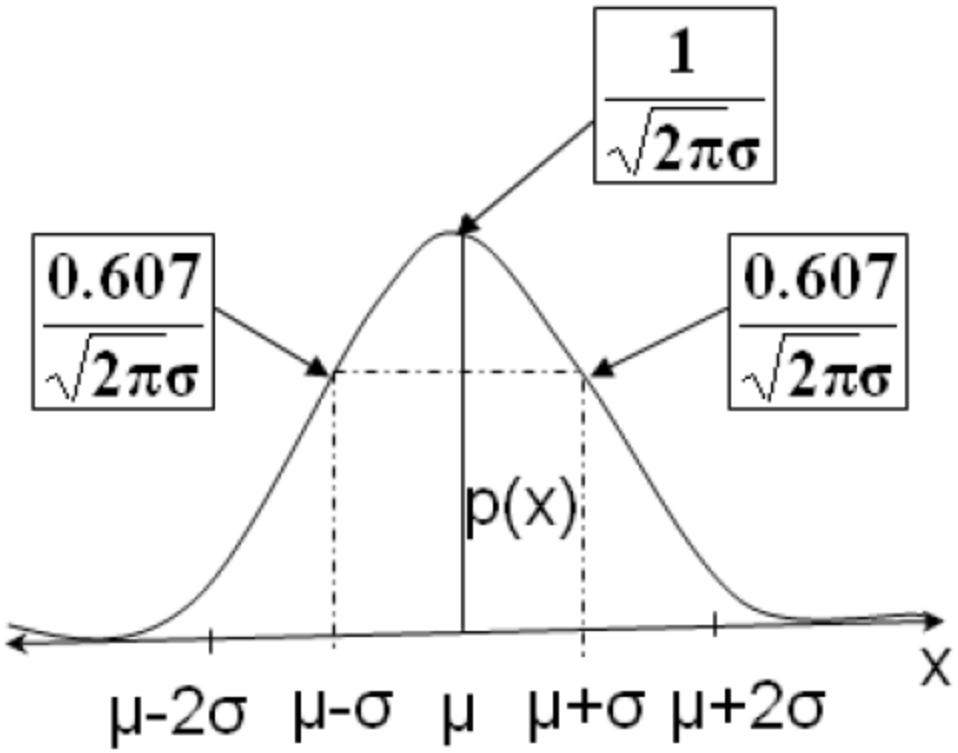


Fig.1. Peak of the univariate normal distribution occurs at x = μ

Multivariate Density

The general multivariate normal density in d-dimensions is written as

where x is the d-component column vector,

μ is the d-component mean vector,

$\sum$ is the d-by-d covariance matrix

$|\sum|$ is the determinant of covariance matrix

$\sum^{-1}$ is the inverse of covariance matrix

$(x-\mu)^t$ is transpose of $(x-\mu)$

Mean Vector (μ) and Covariance Matrix ($\sum$)

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_n \end{bmatrix}, \Sigma = \begin{bmatrix} \sigma_{11} & \cdots & \sigma_{1d} \\ \vdots & \ddots & \vdots \\ \sigma_{d1} & \cdots & \sigma_{dd} \end{bmatrix}$$

$$\sigma_{i,j} = \sigma_{j,i} = \frac{1}{n} \sum_{i,j=1}^{n} \left[(x_i - \mu_i)\left(x_j - \mu_j\right) \right]$$

where

σ_{11} is variance within x_1

σ_{12} is variance between x_1 and x_2

.

.

σ_{ij} is variance between x_i and x_j

$\sum$ is symmetric and its diagonal elements are variances within x which can never be negative.

Off-diagonal elements are the covariances which can be +ve and -ve

Statistically Dependent Variables:

The variables which are causally related are called statistically dependent variables.

Example: engine temperature and oil temperature.

Statistically Independent Variables:

The variables which are not causally related are called statistically independent variables. Example: oil pressure in engine and air pressure in tire.

If the variables are statistically independent, the covariances are zero and covariance matrix is diagonal.

$$\Sigma = \begin{bmatrix} {\sigma_1}^2 & 0 & \cdots & 0 \\ 0 & {\sigma_2}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & {\sigma_d}^2 \end{bmatrix}, \quad \Sigma^{-1} = \begin{bmatrix} 1/{\sigma_1}^2 & 0 & \cdots & 0 \\ 0 & 1/{\sigma_2}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/\sigma_d^2 \end{bmatrix} \Sigma = \left[{\sigma_1}^2 \times {\sigma_2}^2 \times \cdots \times \sigma_d^2\right]$$

Multivariate Density (Bivariate density):

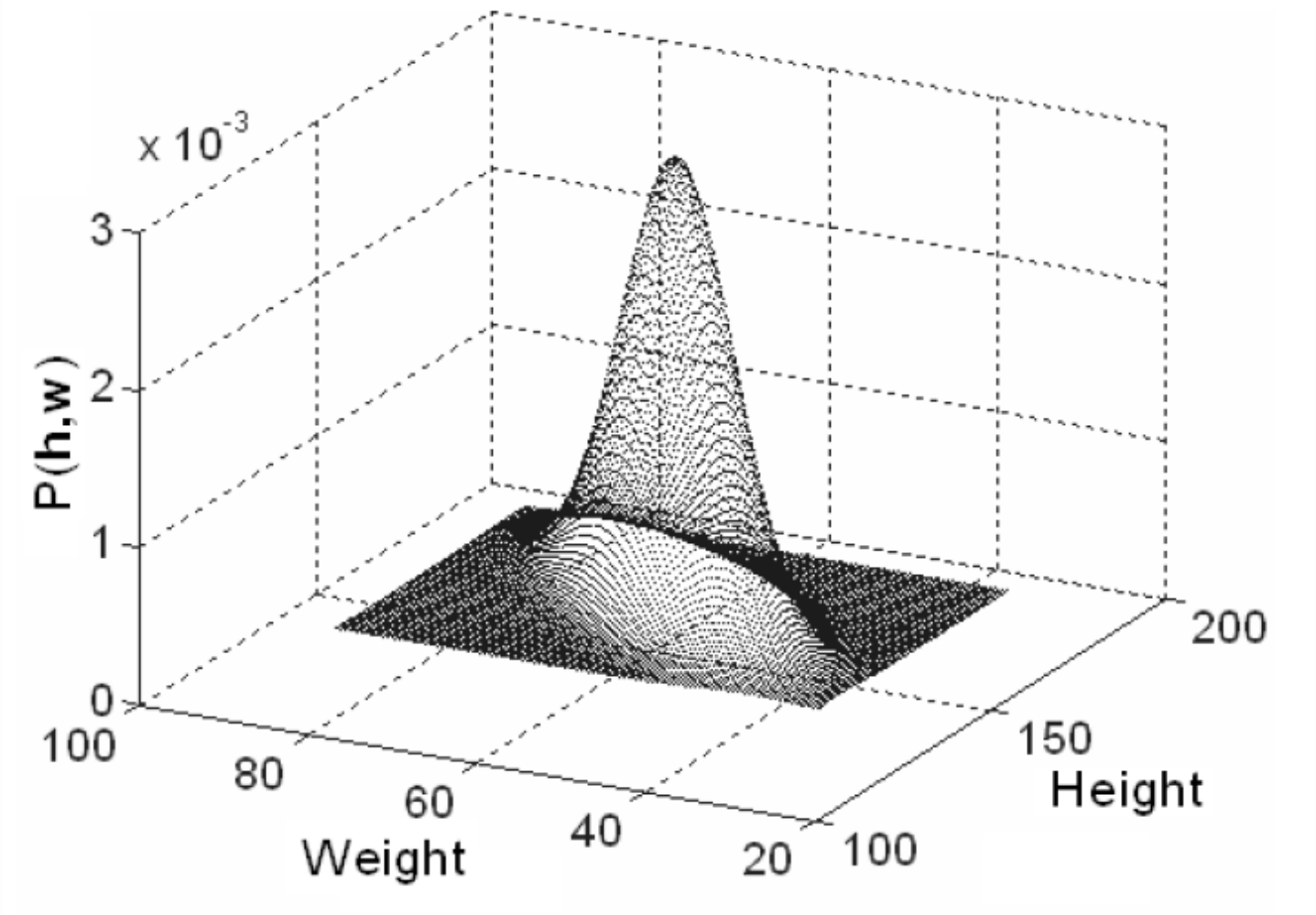
$p(x_1, x_2)$ is a hill shaped surface over the (x_1, x_2) plane. Peak of the bivariate normal distribution occurs at the point $(x_1, x_2) = (\mu_1, \mu 2)$ that is at the mean vector. The shape of the hump depends on the two variances σ_1^2, σ_2^2 and correlation coefficient(ρ) by

$$\rho = \frac{\sigma_{12}}{\sigma_1 \sigma_2}$$

where σ_{12} is variance between x_1 and x_2

σ_1 is variance of x_1

σ_2 is variance of x_2



Bivariate Normal distribution

Exercise 2 - Implementation of Univariate and Multivariate Gaussian density

a)Univariate and Multivariate distributions

Program 1 - Generating and visualizing Univariate and Multivariate distributions

AIM:

To generate, visualize and analyze various univariate and multivariate normal distributions.

Formula:

Univariate Normal:

$$\mathcal{N}(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{\sigma^2}\right)$$

where,
μ - Mean
σ - Variance

Multivariate Normal:

$$\mathcal{N}(\mathbf{x}, \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-(\mathbf{x}-\mu)^T \Sigma^{-1} (\mathbf{x}-\mu)\right)$$

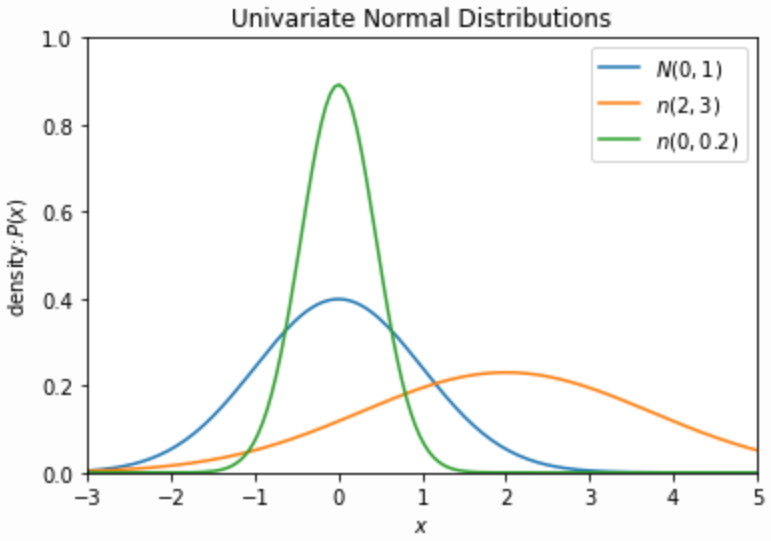
where,
μ - Mean vector
Σ - Covariance matrix
$|\Sigma|$ - Determinant of Σ
Σ^{-1} - Inverse of Σ
d - No. of dimensions

Part 1 - Univariate Normal Distribution

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

```
def univariate_normal(x, mean, variance):
    return 1/np.sqrt(2*np.pi*variance) \
        *np.exp(-(x-mean)**2/(2*variance))

x = np.linspace(-3, 5, 150)
fig = plt.figure(figsize=(6, 4))
plt.plot(x, univariate_normal(x, 0, 1), label="$N(0,1)$")
plt.plot(x, univariate_normal(x, 2, 3), label="$n(2,3)$")
plt.plot(x, univariate_normal(x, 0, 0.2), label="$n(0,0.2)$")
plt.title("Univariate Normal Distributions")
plt.xlabel("$x$")
plt.ylabel("density:$P(x)$")
plt.xlim((-3, 5))
plt.ylim((0, 1))
plt.legend()
plt.show()
```

Output:

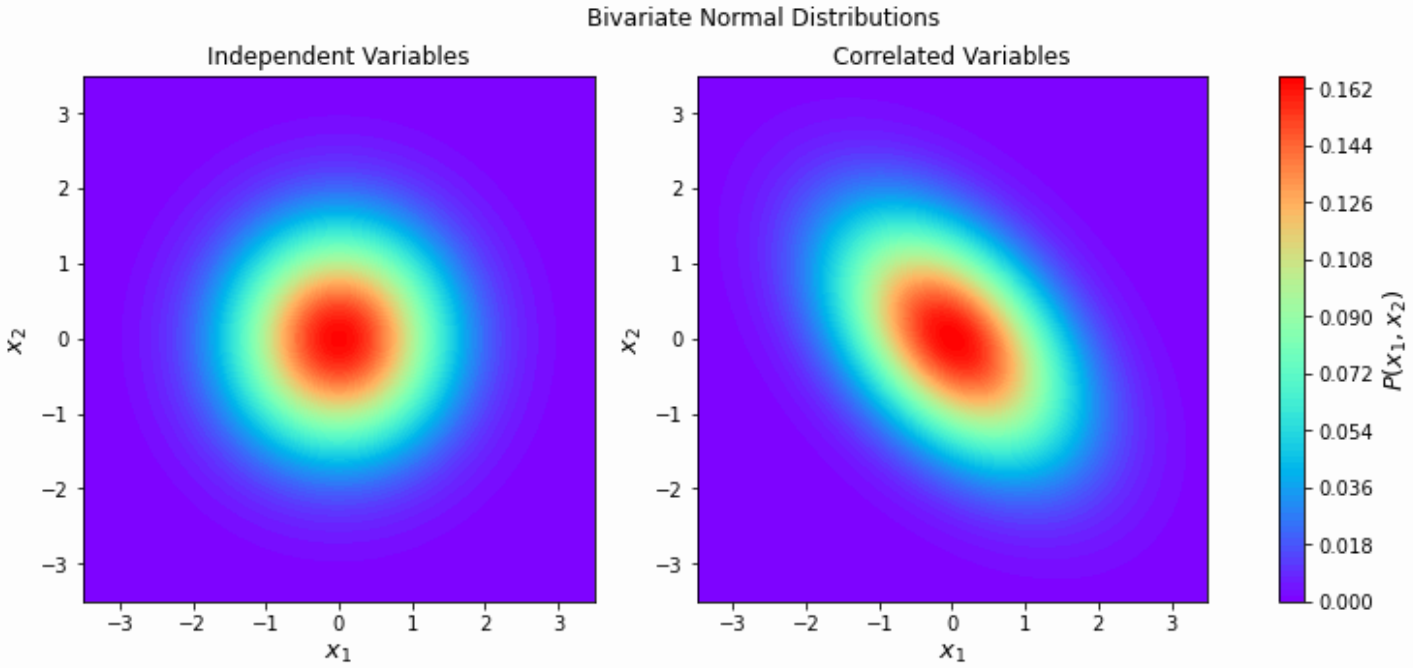


Part 2 - Multivariate Normal Distribution

```
def multivariate_normal(x, mean, cov):
    d,x_m = mean.shape[0],x-mean
    cov_det, cov_inv = np.linalg.det(cov),np.linalg.inv(cov)
    return 1/np.sqrt((2*np.pi)**d*cov_det)\
        *np.exp(-(np.einsum("...i,ij,...j",x_m,cov_inv,x_m))/2)

def generate_surface(mean, cov):
    x = np.linspace(-5, 5,100)
    x1, x2= np.meshgrid(x,x)
    pdf = multivariate_normal(np.dstack([x1,x2]),mean,cov)
    return x1,x2, pdf
```

```
distributions = [
    (np.zeros(2),np.eye(2)),
    (np.zeros(2),np.array([[1, -0.1], [-0.8, 1]]))
]

fig, ax = plt.subplots(1, 2, figsize=(13, 5))
plt.suptitle("Bivariate Normal Distributions")
ax[0].set_title("Independent Variables")
ax[1].set_title("Correlated Variables")
for i, dist in enumerate(distributions):
    x1, x2, pdf = generate_surface(*dist)
    con = ax[i].contourf(x1, x2, pdf, 100, cmap='rainbow')
    ax[i].set_xlabel("$x_1$", fontsize=13)
    ax[i].set_ylabel("$x_2$", fontsize=13)
    ax[i].axis([-3.5, 3.5, -3.5, 3.5])

c_bar = fig.colorbar(con, ax = ax)
c_bar.ax.set_ylabel("$P(x_1,x_2)$", fontsize=13)
plt.show()
```

Output:



3. Principal Component Analysis

Introduction:

Development in the data collection and storage techniques in recent days have led to an information overload. Researchers working on diversified domains such as engineering, astronomy, biology, remote sensing, consumer transactions, etc., come across an huge or high dimensional datasets which present many mathematical challenges to handle due to curse of dimensionality. As the dimensionality of the dataset increases, its performance decreases as seen in Fig.1. Moreover, if the dimensionality of the input space is higher, more feature vectors are needed for training. The major problem with these high-dimensional data sets is that, in most cases, not all the measured data are important for understanding the underlying phenomena of interest. Thus, dimension reduction is necessary for effective analysis of high dimensional data sets. Principal components analysis (PCA) and Linear discriminant analysis (LDA) are well-known schemes for dimension reduction. PCA finds a set of most representative projection vectors, and thus the projected samples preserves the most relevant information about original dataset.

Principal Components Analysis

Principal components analysis(PCA) is a useful statistical technique that has found application in many fields such as video/audio classification, face recognition, image compression etc., It is a simple method of extracting relevant information from high dimensional data sets. With minimal effort,

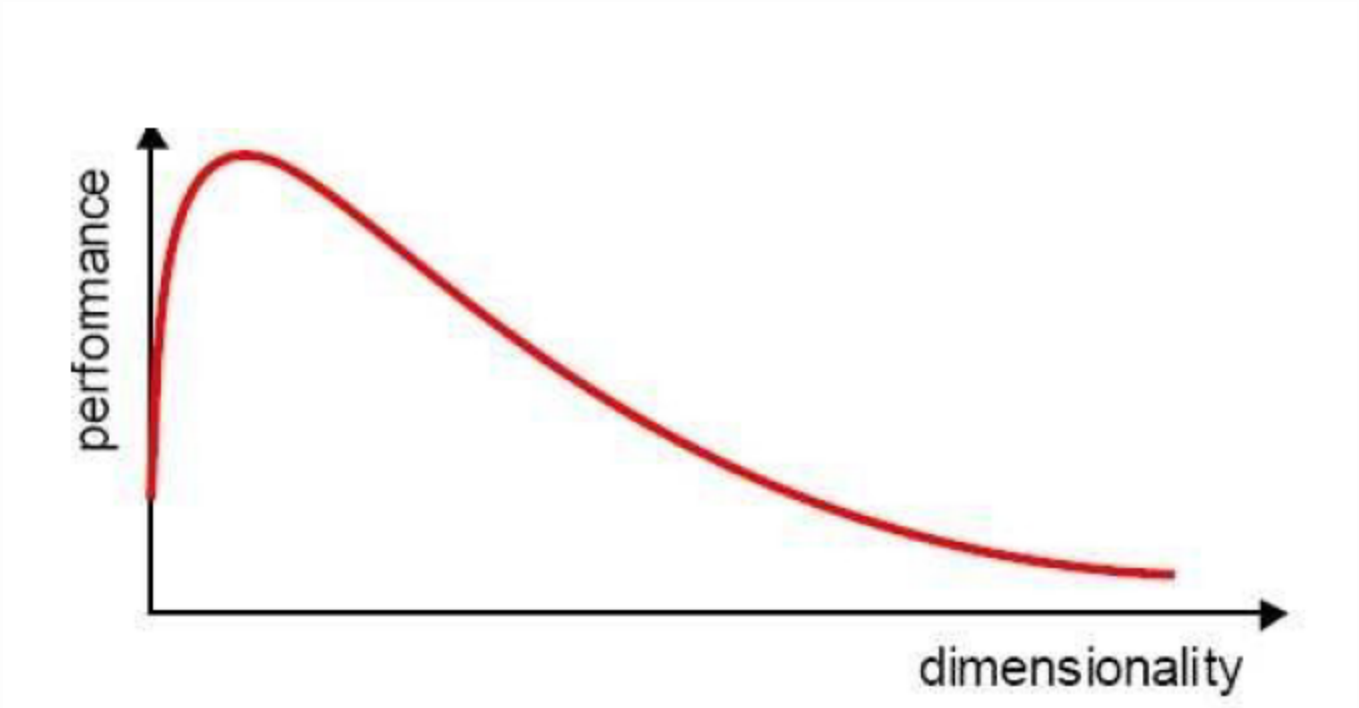
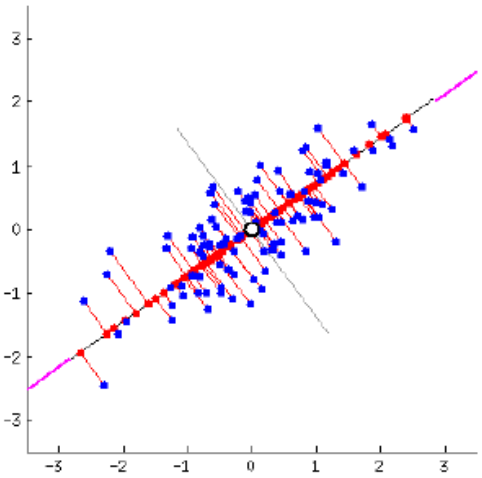


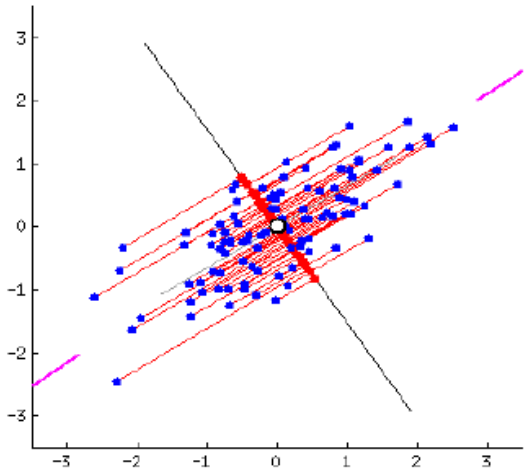
Fig.1: Curse of dimensionality

PCA provides a road map for reducing the complex data set to a lower dimensional dataset. Principal component analysis (PCA) was first introduced by Pearson in 1901 and later independently developed by Hotelling in 1933, where the name principal components first appears. In various fields, it is also known as the singular value decomposition (SVD), the Karhunen-Loeve transform, the Hotelling transform, and the empirical orthogonal function (EOF) method.

scree plot below.
Geometrically principal components represent the directions of the data that explain a **maximal amount of variance**, that is to say, the lines that capture most information of the data. The relationship between variance and information, i.e, the larger the variance carried by a line, the larger the dispersion of the data points along it, and the larger the dispersion along a line, the more the information it has.



Principal Components 1



Principal Components 2

Principal Components Analysis - Algorithm
PCA is a useful statistical procedure that has found importance in many fields, and is a wellknown technique for finding patterns in high dimensional data. It is a way of identifying patterns in data, and expressing the data in such a way to highlight their similarities and differences. The other main advantage of PCA is that once these patterns are found, the data can be compressed by reducing the number of dimensions, without much loss of information. Thus Principal Components 'combines' the essence of attributes by creating an alternative, smaller set of variables . The initial data can then be projected onto this principal component.
Steps to find Principal Components:
Standardize the Dataset (Subtract the mean from the data items)
Calculate **covariance matrix**
Compute the **eigenvectors and eigenvalues** of the covariance matrix to identify the **principal components**
Compute the new projected data set
the aim is to use the feature vector formed using the eigenvectors of the covariance matrix, to reorient the data from the original axes to the ones represented by the principal components (hence the name Principal Components Analysis). This can be done by multiplying the transpose of the original data set by the transpose of the feature vector.
$\mathrm{FinalDataset} = \mathrm{FeatureVector}^T \times \mathrm{StandardizdOriginalDataSet}^T$

Exercise 3 - Principal Component Analysis

a)Reducing the Dimension of the features Using PCA

Program 1 - Implementation of PCA in Iris Data Set(From Scratch and using Scikit Learn)

AIM:

To implement Principal component analysis from scratch and using Scikit Learn.

ALGORITHM:

1. Standardize the dataset.
2. Calculate the Covariance Matrix.
3. Calculate eigenvalues and eigenvectors of the covariance matrix.
4. Take the n eigenvectors with the highest eigenvalues as the eigenvector subset.
5. Project the dataset into lower dimension by transorming the dataset with the eigenveector subset.

Part 1 - Defining function for PCA

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

```
class PCA:
    def __init__(self,n_components):
        self.n_components = n_components

    def fit_transform(self,X):
        n = self.n_components
        X_standardized = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
        cov_mat = np.cov(X_standardized, rowvar=False)
        _, eigen_vectors = np.linalg.eigh(cov_mat)  # sorted in ascending order
        self.eigenvector_subset = eigen_vectors[:, :-n-1:-1]  # take last n eigen vec
        return X_standardized @ self.eigenvector_subset
```

Part 2 - Loading Iris Dataset

```
iris_df = pd.read_csv(
    "./datasets/iris.csv",
    names = ["sepal length", "sepal width","petal length", "petal width","target"]
)
print('Iris Dataset contains 4 features column to determine the species Name')
print('Shape of the Iris Data:',iris_df.shape)
display(iris_df)
```

Output:

```
Iris Dataset contains 4 features column to determine the species Name
Shape of the Iris Data: (150, 5)
```

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

Part 3 - Implementing PCA in the dataset

```
iris_X = iris_df.iloc[:,:-1].values
iris_y = iris_df.iloc[:,-1].values
pca = PCA(2)
reduced_iris_X = pca.fit_transform(iris_X)
reduced_iris_df = pd.DataFrame(reduced_iris_X,columns=["PC1","PC2"])
reduced_iris_df["target"] = iris_df["target"]
print("After Feature Dimensional Reduction "
      "from 4 Features column Reduced into 2 PC")
display(reduced_iris_df)
```

Output:

```
After Feature Dimensional Reduction from 4 Features column Reduced into 2 PC
```

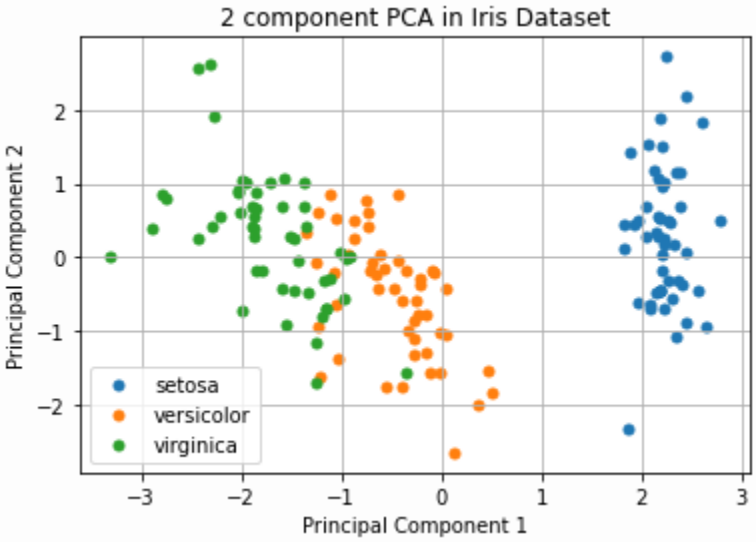
	PC1	PC2	target
0	2.264542	0.505704	setosa
1	2.086426	-0.655405	setosa
2	2.367950	-0.318477	setosa
3	2.304197	-0.575368	setosa
4	2.388777	0.674767	setosa
...	...	...	...
145	-1.870522	0.382822	virginica
146	-1.558492	-0.905314	virginica
147	-1.520845	0.266795	virginica
148	-1.376391	1.016362	virginica
149	-0.959299	-0.022284	virginica

150 rows × 3 columns

Part 4 - Visualizing the Principal components of the reduced iris dataset

```
plt.title('2 component PCA in Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
targets = np.unique(iris_y)
for target in targets:
    idxs = iris_y == target
    plt.scatter(
        reduced_iris_X[idxs,0],
        reduced_iris_X[idxs,1],
        s=25,label = target
    )
plt.legend()
plt.grid()
plt.show()
```

Output:



Part 5 - Loading the Breast cancer Dataset from scikit Learn

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA as SklPCA
```

```
cancer_data = load_breast_cancer()
cancer_X = cancer_data.data
cancer_y = cancer_data.target_names[cancer_data.target]
cancer_df = pd.DataFrame(
    cancer_X, columns=cancer_data.feature_names)
cancer_df["target"] = cancer_y
cancer_df
```

Output:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture	worst perim
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.07871	...	17.33	184.60
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.05667	...	23.41	158.80
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.05999	...	25.53	152.50
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	0.09744	...	26.50	98.87
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	0.05883	...	16.67	152.20
...	...	...	...	...	...	...	...	...	...	...	...	...	...
564	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1726	0.05623	...	26.40	166.10
565	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1752	0.05533	...	38.25	155.00
566	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1590	0.05648	...	34.12	126.70
567	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2397	0.07016	...	39.42	184.60
568	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1587	0.05884	...	30.37	59.16

569 rows × 31 columns

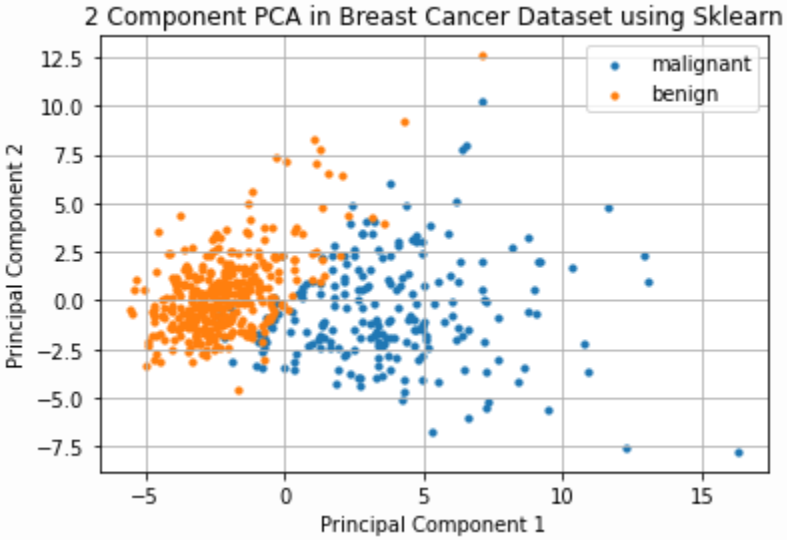
Part 6 - Implementing PCA Breast cancer dataset using scikit learn

```
data_scaler = StandardScaler()
scaled_cancer_X = data_scaler.fit_transform(cancer_X)

skl_pca = SklPCA(n_components=2)
reduced_cancer_X = skl_pca.fit_transform(scaled_cancer_X)
```

Part 7 - Visualizing the Prinical Components of the reduced breast cancer dataset

```
plt.title("2 Component PCA in Breast Cancer Dataset using Sklearn")
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')

for target in cancer_data.target_names:
    idxs = cancer_y == target
    scatter = plt.scatter(
        reduced_cancer_X[idxs, 0],
        reduced_cancer_X[idxs, 1],
        s=10,label = target
    )
plt.legend()
plt.grid()
plt.show()
```

Output:


2 Component PCA in Breast Cancer Dataset using Sklearn
malignant
benign
Principal Component 2
Principal Component 1
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−5
0
5
10
15

4. a) K Means Clustering

Introduction:

Clustering in a Machine Learining is an unsupervised learning method which deals with finding a structure in a collection of unlabeled data. A loose definition of clustering could be the process of organizing objects into groups whose members are similar in some way. A cluster is a collection of objects which are similar between them and are dissimilar to the objects belonging to other clusters.

Clustering:

Clustering is one of the most common exploratory data analysis technique used to get an intuition about the structure of the data. It can be defined as the task of identifying subgroups in the data such that data points in the same subgroup (cluster) are very similar while data points in different clusters are very different. In other words, we try to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure such as euclidean-based distance or correlation-based distance. The decision of which similarity measure to use is application-specific.
Clustering analysis can be done on the basis of features where we try to find subgroups of samples based on features or on the basis of samples where we try to find subgroups of features based on samples.
We'll cover here clustering based on features. Clustering is used image segmentation/compression; where we try to group similar regions together, document clustering based on topics, etc.
Unlike supervised learning, clustering is considered an unsupervised learning method since we don't have the ground truth to compare the output of the clustering algorithm to the true labels to evaluate its performance. We only want to try to investigate the structure of the data by grouping the data points into distinct subgroups.
K-means which is considered as one of the most used clustering algorithms due to its simplicity. k-means clustering algorithm group objects based on attributes/features into k number of groups where k is a positive integer. The grouping (clustering) is done by minimizing the Euclidean distance between data and the corresponding cluster centroid. Thus the purpose of k-means clustering is to cluster the data.

Kmeans Algorithm

It is an iterative algorithm that tries to partition the dataset into K-pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to **only one group**. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.
The way kmeans algorithm works is as follows: 1. Specify number of clusters K. 2. Initialize centroids by first shuffling the dataset and then randomly selecting K data points for the centroids without replacement. 3. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.

- Compute the sum of the squared distance between data points and all centroids.
- Assign each data point to the closest cluster (centroid).
- Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.
  In Expectation-Maximization perspective The E-step is assigning the data points to the closest cluster. The M-step is computing the centroid of each cluster. Below is a break down of how we can solve it mathematically.

The objective function is:

$$J = \sum_{i=1}^{m} \sum_{k=1}^{K} w_{ik} \|x_i - \mu_k\|^2$$

where $w_{ik} = 1$ for data point x_i if it belongs to cluster k; otherwise $w_{ik} = 0$. Also, μ_k is the centroid of x_i's cluster.
It's a minimization problem of two parts. We first minimize w.r.t. $w_{i k} and treat μ_k fixed. Then we minimize w.r.t. μ_k and treat $w_{ik} fixed. Technically speaking, we differentiate w.r. t.w_{i k} first and update cluster assignments (E-step). Then we differentiate w.r.t. μ_k and recompute the centroids after the cluster assignments from previous step (M-step).
Therefore, E-step is:

$$\frac{\partial J}{\partial w_{ik}} = \sum_{i=1}^{m} \sum_{k=1}^{K} \|x_i - \mu_k\|^2$$

$$\Rightarrow w_{ik} = \begin{cases} 1 & \text{if } k = \operatorname{argmin}_j \|x_i - \mu_j\|^2 \\ 0 & \text{otherwise} \end{cases}$$

In other words, assign the data point x_i to the closest cluster judged by its sum of squared distance from cluster's centroid.
And M-step is:

$$\frac{\partial J}{\partial \mu_k} = 2 \sum_{i=1}^{m} w_{ik} (x_i - \mu_k) = 0$$

$$\Rightarrow \mu_k = \frac{\sum_{i=1}^{m} w_{ik} x_i}{\sum_{i=1}^{m} w_{ik}}$$

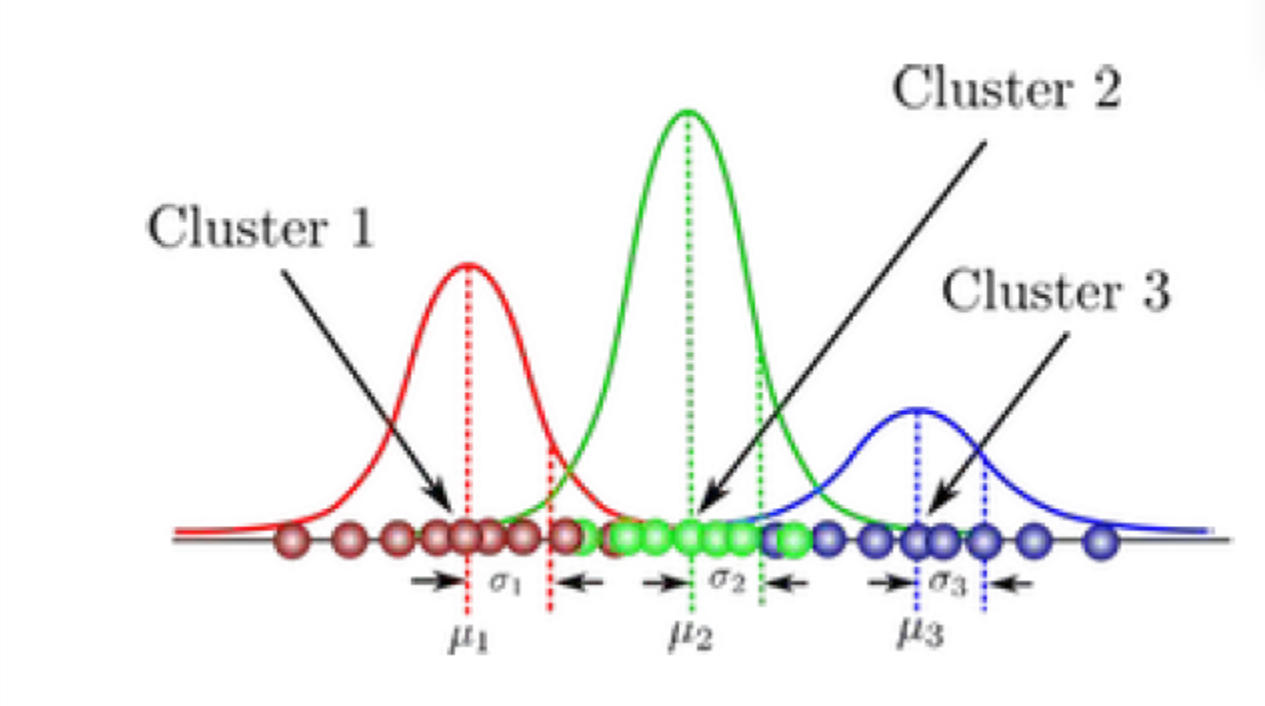
Which is translates to recomputing the centroid of each cluster to reflect the new assignments.
Few things to note here:

- Since clustering algorithms including kmeans use distance-based measurements to determine the similarity between data points, it's recommended to standardize the data to have a mean of zero and a standard deviation of one since almost always the features in any dataset would have different units of measurements such as age vs income.
- Given kmeans iterative nature and the random initialization of centroids at the start of the algorithm, different initializations may lead to different clusters since kmeans algorithm may stuck in a local optimum and may not converge to global optimum. Therefore, it's recommended to run the algorithm using different initializations of centroids and pick the results of the run that that yielded the lower sum of squared distance.
- Assignment of examples isn't changing is the same thing as no change in within-cluster variation:

$$\frac{1}{m_k} \sum_{i=1}^{m_k} \|x_i - \mu_{c^k}\|^2$$

4. b) Gaussian Mixture

Data measurements of many properties are often normally distributed, but with heterogeneous populations, sometimes data measurements reflect a mixture of normal distributions. Mixture models are a type of density model which comprise a number of component functions, usually Gaussian. These component functions are combined to provide a multi model density.Gaussian mixture models are formed by combining multivariate normal density components
Gaussian mixture models are often used for data clustering. Like k-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than k-means clustering when clusters have different sizes and correlation within them.
Gaussian mixture model involves the mixture (i.e. superposition) of multiple Gaussian distributions. Rather than identifying clusters by "nearest" centroids, as in k-means in this experiment we fit a set of k gaussians to the data and then we estimate gaussian distribution parameters such as mean and variance for each cluster and weight of a cluster. After learning the parameters for each data point we can calculate the probabilities of it belonging to each of the clusters.



Maximum Likelihood

For most sensible models, we will find that certain data are more probable than other data. The aim of maximum likelihood estimation is to find the parameter (mean, covariance) value(s) that makes the observed data most likely. This is because the likelihood of the parameters given the data is defined to be equal to the probability of the data given the parameters. However, in the case of data analysis, we have already observed all the data: once they have been observed they are fixed, there is no 'probabilistic' part to them anymore. We are much more interested in the likelihood of the model parameters that underly the fixed data.
Probability: Knowing parameters Prediction of outcome
Likelihood: Observation of data = Estimation of parameters
For example, suppose you are interested in the heights of Americans. You have a sample of some number of Americans, but not the entire population, and record their heights. Further, you are willing to assume that heights are normally distributed with some unknown mean and variance. The sample mean is then the maximum likelihood estimator of the population mean, and the sample variance is a close approximation to the maximum likelihood estimator of the population variance.

GMM

Gaussian mixture model (GMM) is a mixture of several Gaussian distributions and can therefore represent different subclasses inside one class. Figure.1 shows themixture of two Gaussians. The probability density function of GMM is defined as a weighted sum of Gaussians.

$$p(x \mid \Theta) = \sum_{k=1}^{K} \alpha_k p_k \left(x \mid \theta_k\right)$$

where k is the number of mixtures and

$$\sum_{k=1}^{K} \alpha_k = 1$$

$$\sum_{k=1}^{K} \alpha_k = 1$$

are the mixture weights
where

$$\Theta = \{\alpha_1, \ldots\ldots, \alpha_k, \theta_1, \ldots\ldots, \theta_k\}$$

We are given dataset $D = x_1, x_2, \ldots, x_n$ where x_i is a d−dimensional vector.
Assume that the pointsare genetrated in an identically independent fashion from an underlying density p(x), We further assume a Gaussian mixture model with K components.

$$p_k \left(x \mid \theta_k\right) = \frac{1}{(2\pi)^{\frac{d}{2}} \left|\Sigma_k\right|^{\frac{1}{2}}} e^{-\frac{1}{2}\left[(x-\mu_k)^T \Sigma_k^{-1} (x-\mu_k)\right]}$$

with its own parameters $\theta_k = \{\mu_k, \Sigma_k\}$ We can compute the membership of data point in cluster k given parameters Θ as

$$w_{i,k} = \frac{p_k \left(x_i \mid \theta_k\right) \cdot \alpha_k}{\sum_{m=1}^{K} p_m \left(x_i \mid \theta_m\right) \cdot \alpha_m}$$

GMM Training
Several approaches exist for estimating the parameters of the GMM given a set of data points. The most popular, and the one used here, is the expectation-maximization (EM) algorithm, which iteratively optimizes the model using maximum likelihood estimates. Expectation maximization (EM) algorithm is an iterative algorithm consisting of expectation step (E-step) and maximization step (M-step), and is widely used for model training
The EM Algorithm: We define the EM (Expectation-Maximization) algorithm for Gaussian mixtures as follows.
E-Step: Denote the current parameter value as Θ. Compute w_{ik}, using equation $p_k \left(x \mid \theta_k\right)$ for all data points $x_i, 1 \leq i \leq N$ and all mixture components $1 \leq k \leq K$. Note that for each data point x_i the membership weights are defined such that $\sum_{k=1}^{K} w_{ik} = 1$. This yields an N × K matrix of membership weights, where each of the rows sum to 1 .
M-Step: Now use the membership weights and the data to calculate new parameter values.Specifically,

$$\alpha_k^{new} = \frac{1}{N}\sum_{i=1}^{N} w_{ik} \cdot x_i \quad 1 \le k \le K$$

$$\mu_k^{\text{new}} = \left(\frac{1}{\sum_{i=1}^{N} w_{ik}}\right)\sum_{i=1}^{N} w_{ik} \cdot x_i \quad 1 \le k \le K$$

$$\sum_k^{new} = \left(\frac{1}{\sum_{i=1}^{N} w_{ik}}\right)\sum_{i=1}^{N} w_{ik} \cdot \left(x_I - \mu_k^{new}\right)\left(x_i - \mu_k^{ncw}\right)^t \quad 1 \le k \le K$$

The equations in the M-Step need to be computed in this order, i.e., first compute the K new α′s,s, then the K new μ′s,s, and finally the K new Σ' s. .

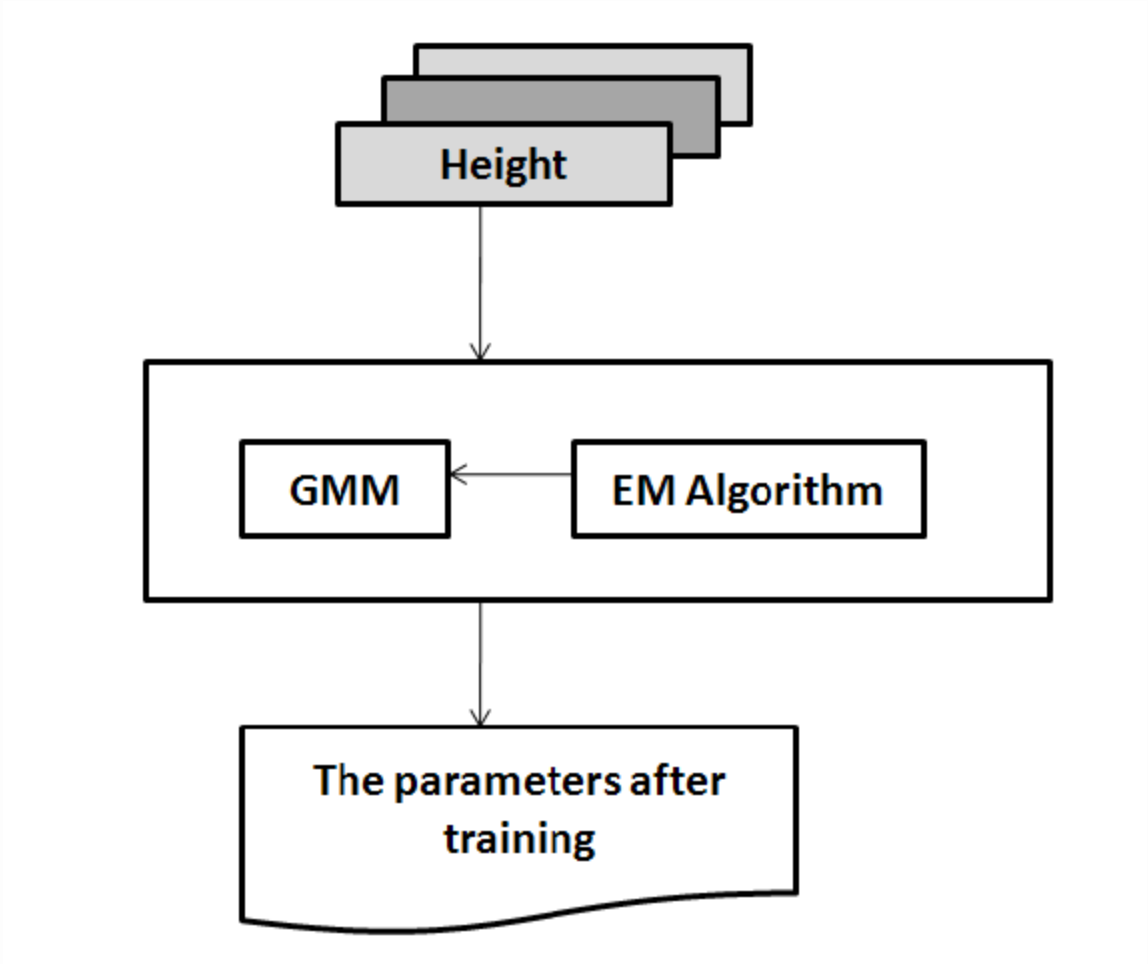
After we have computed all of the new parameters, the M-Step is complete and we can now go back and recompute the membership weights in the E-Step, then recompute the parameters again in the M-Step , and continue updating the parameters in this manner. Each pair of E and M steps is considered to be an iteration.

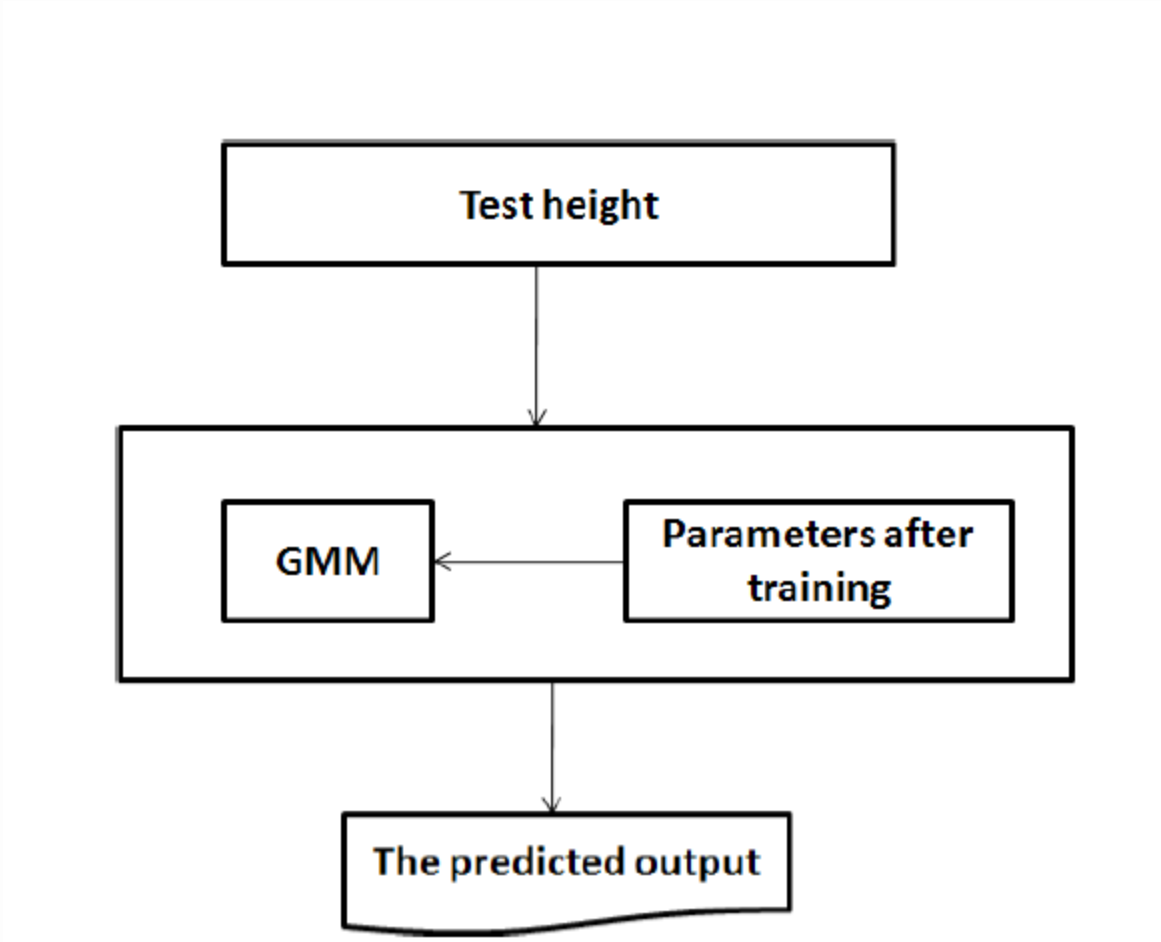
Initialization Issues

The initial parameters or weights can be chosen randomly (e.g. select K random data points as initial means and select the covariance matrix of the whole data set for each of the initial K covariance matrices) or could be chosen via some heuristic method (such as by using k-Means to cluster the data first and then defining weights based on k-Means memberships).

GMM Testing

After obtaining the maximum likelihood estimates of the various components of the mixture using EM algorithm the probability density function is found out using equation $p_k\,(\,x \quad \theta_k\,)$. Given a test data the probability density function is found out for each class as a weighted sum of Gaussians using the estimated parameters. The test data belongs to the class which has the highest probability.





Algorithm

1. Initialize the mean μ_k , the covariance matrix and the mixing coefficients α_k by some random values(or other values).
2. Compute the $w_{i,k}$ values for all k.
3. Again Estimate all the parameters using the current $w_{i,k}$ values.
4. Compute log-likelihood function.
5. Put some convergence criterion
6. If the log-likelihood value converges to some value (or if all the parameters converge to some values) then stop, else return to Step 2.

This algorithm only guarantee that we land to a local optimal point, but it do not guarantee that this local optima is also the global one. And so, if the algorithm starts from different initialization points, in general it lands into different configurations. For demonstraton purpose in this exercise data were generated from a random mixture of Gaussians for which we find the best fit clusters.
DATA generated from the stat package: scipy.stats
from scipy.stats import multivariate_normal

Exercise 4 - Clustering Algorithms

a) Clustering using K-means

Program 1 - Implementing K-means Clustering from scratch and using scikit learn

AIM:

To implement K-means clustering algorithm from scratch and also implement it with scikit learn.

ALGORITHM:

K-means:

1. Initialize randomly selected points as centroids.
2. Find the clusters for each point.
3. Compute new centroids by computing the mean of each cluster.
4. Repeat 1-3 until convergence.

Distrotion(metric):

Distrotion is calculated as the average of the squared distances from the cluster centers of the respective clusters. which can be mathematically defined as:

$$\text{Distrotion} = \sum_{\mathbf{c}_k, X_k \in C} \sum_{\mathbf{x}_i \in X_k} (\mathbf{x}_i - \mathbf{c}_k)^2$$

where,
C - Set of clusters
$\mathbf{c}_k$ - centroid of the kth cluster
X_i - Points in the kth cluster

Sillhoutte Score(metric):

Silhoette Score is defined as the mean of the sillhouette coefficents of the follows: Let A be the cluster where the ith points is assigned and C be any cluster.

$$\text{SilhouetteScore} = \frac{1}{m} \sum_{i=1}^{m} s(i)$$

$$\text{silhouetteCoeffient. } s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

$$b(i) = \min_{C \neq A} d(i, C)$$

where,
$a(i)$ - average distance from ith point to all points in A
$d(i, C)$ - average distaance from ith point to all points in cluster C

Part 1 - Defining class for KMeans

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
```

```
class KMeans:
    def __init__(self, n_clusters):
        self.n_clusters = n_clusters

    def fit(self, train_x, random_state = 0):
        m = train_x.shape[0]
        np.random.seed(random_state)
        self.centroids = train_x[np.random.choice(m,self.n_clusters,replace=False)]
        while True:
            old_centroids = self.centroids.copy()
            self.labels = self.predict(train_x)
            for cluster_id in range(self.n_clusters):
                cluster = train_x[self.labels == cluster_id]
                if len(cluster):
                    self.centroids[cluster_id] = cluster.mean(axis=0)
            if np.all(old_centroids != self.centroids) : break

    def predict(self, points):
        return np.argmin(cdist(points,self.centroids),axis=-1)
```

Part 2 - Defining evaluation metrics

```
def distortion(X,labels,centroids):
    _, label_codes = np.unique(labels,return_inverse=True)
    return np.linalg.norm((X-centroids[label_codes]),axis=-1).mean()

def silhouette_score(X, labels):
    m = X.shape[0]
    clusters = np.unique(labels)
    n_clusters = clusters.shape[0]
    if n_clusters == 1: return -1
    cluster_codes = np.arange(n_clusters)
    dist_mask = labels.reshape(-1,1) == \
        ((labels + cluster_codes.reshape(-1,1))%n_clusters)[:,np.newaxis,:]
    dist_mask[0][np.eye(m).astype(bool)] = 0
    dist_mat = cdist(X,X)
    mean_dist = (dist_mask*dist_mat).sum(axis=-1)/dist_mask.sum(axis=-1)
    A = mean_dist[0]
    B = np.min(mean_dist[1:],axis=0)
    return ((B-A)/np.max(np.dstack([A,B]),axis=-1)).mean()
```

Part 3 - Loading the iris dataset

```
names = ['s_len', 's_wid', 'p_len', 'p_wid', 'species']
iris_df = pd.read_csv('./datasets/iris.csv', header=None, names=names)
display(iris_df.head())
print(
    f"The data set contains {iris_df.shape[0]} records "
    f"and {iris_df.shape[1]} features.",
    iris_df['species'].value_counts(), sep="\n"
)
```

Output:

	s_len	s_wid	p_len	p_wid	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
The data set contains 150 records and 5 features.
setosa        50
versicolor    50
virginica     50
Name: species, dtype: int64
```

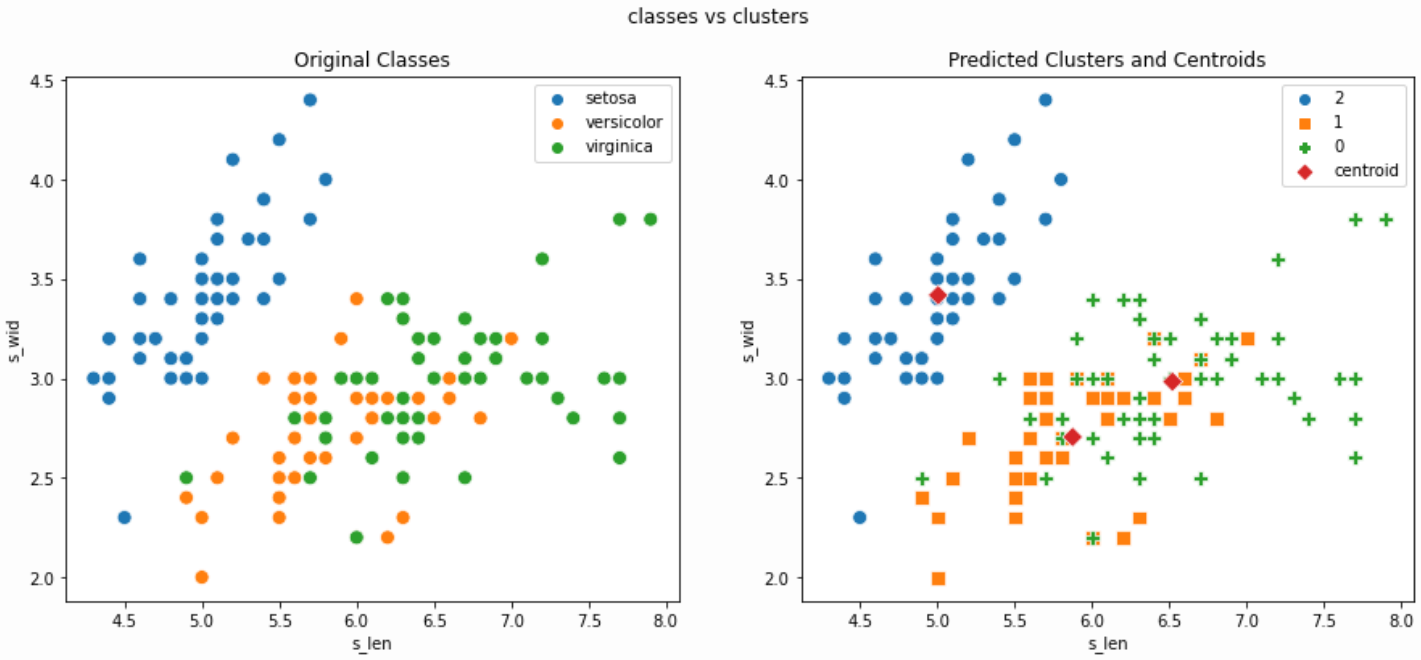
Part 4 - Implementing K-means Clustering

```
train_x = iris_df.drop("species", axis=1).values
k = 3
kmeans = KMeans(n_clusters=k)
kmeans.fit(train_x)
all_df = iris_df.copy(deep = True)
centroids = pd.DataFrame(kmeans.centroids, columns=names[:-1])
centroids["cluster"] = "centroid"
all_df['cluster'] = kmeans.labels.astype("str")
all_df = pd.concat([all_df, centroids])
```

Part 5 - Comparing the Clustering results with original classes

```
plt.figure(figsize=(15,6))
plt.suptitle("classes vs clusters")
plt.subplot(121)
plt.title("Original Classes")
sns.scatterplot(data=all_df,x="s_len",y="s_wid",hue="species",s=80)
plt.legend(loc="upper right")

plt.subplot(122)
plt.title("Predicted Clusters and Centroids")
sns.scatterplot(
    data=all_df,x="s_len",y="s_wid",
    hue="cluster",style = "cluster",
    markers="osPD",s=80
)
plt.legend(loc="upper right")

plt.show()
```

Output:



Part 6 - Testing with multiple values of k(Hyperparameter tuning)

```
sil_coefs = []
distortions = []
K = np.arange(2,6)
for k in K:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(train_x)
    avg_sil_coef = silhouette_score(train_x,kmeans.labels)
    dist = distortion(train_x,kmeans.labels,kmeans.centroids)
    print(f"For k={k:<4} Avg.Sil.Coef: {avg_sil_coef:<10.5f} Distortion: {dist:.5f}")
    distortions.append(dist)
    sil_coefs.append(avg_sil_coef)
```

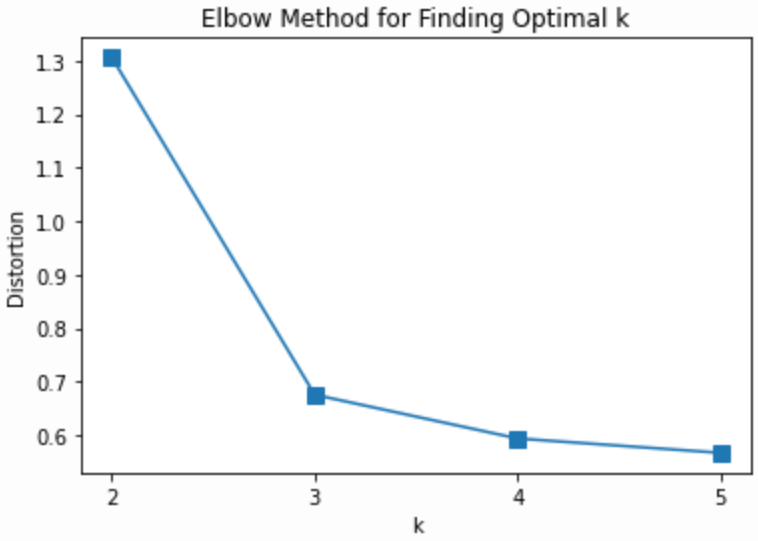
Output:

```
For k=2    Avg.Sil.Coef: 0.41657    Distortion: 1.30604
For k=3    Avg.Sil.Coef: 0.49127    Distortion: 0.67576
For k=4    Avg.Sil.Coef: 0.45307    Distortion: 0.59402
For k=5    Avg.Sil.Coef: 0.32360    Distortion: 0.56721
```

Part 7 - Using Elbow method to Find the optimal value of k

```
plt.plot(K, distortions, 's-', markersize=8)
plt.xlabel('k')
plt.xticks(K)
plt.ylabel('Distortion')
plt.title('Elbow Method for Finding Optimal k')
plt.show()
```

Output:



Part 8 - Implementing K-means clustering with scikit learn

```
from sklearn.cluster import KMeans as sklKMeans
from sklearn.metrics import silhouette_score as sk_silhouette_score
```

```
k = 3
kmeans = sklKMeans(n_clusters=k, init='random').fit(train_x)
ss = sk_silhouette_score(train_x, kmeans.labels_)
dist = distortion(train_x,kmeans.labels_, kmeans.cluster_centers_)
print(f"For k={k}\tAvg. Sil. Coef: {ss:.5f}\tDistortion: {dist:.5f}")
```

Output:

```
For k=3 Avg. Sil. Coef: 0.55259 Distortion: 0.64884
```

b) Clustering using Gaussian Mixture Model(GMM)

Program 1 - Implementing GMM from scratch using Expectation and Maximization algorithm

AIM:

To implement GMM from scratch in python using Expectation and Maximization algorithm.

Formula:

Log-likelihood

$$l(\theta) = \sum_i \log\left(\sum_k p(\mathbf{x}_i, Z = k|\theta)\right) = \sum_i \log\left(\sum_k \pi_k \mathcal{N}(\mathbf{x}_i|\mu_k. \mathbf{\Sigma}_k))\right)$$

Auxilary Function

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)}) = \sum_i \sum_k r_{ik} \log \pi_k + \sum_i \sum_k r_{ik} \log p(\mathbf{x}_i|\boldsymbol{\theta}_k)$$

where $r_{ik} \triangleq p(z_i = k|x_i, \boldsymbol{\theta}^{(t-1)})$ is the responsibility that cluster k takes for data point i.

Responibility

$$r_{ik} = \frac{\pi_k p(\mathbf{x}_i|\boldsymbol{\theta}_k^{(t-1)})}{\sum_{k'} \pi_{k'} p(\mathbf{x}_i|\boldsymbol{\theta}_{k'}^{(t-1)})} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \mathbf{\Sigma}_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_{k'}, \mathbf{\Sigma}_{k'}))}$$

Parameter updates

$$\pi_k = \frac{1}{N} \sum_i r_{ik} = \frac{r_k}{N}$$
$$\boldsymbol{\mu}_k = \frac{\sum_i r_{ik} \mathbf{x}_i}{r_k}$$
$$\mathbf{\Sigma}_k = \frac{\sum_i r_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T}{r_k}$$

where
$r_k \triangleq \sum_i r_{ik}$ - the weighted number of points.
N - number of Points

ALGORITHM:

GMM Using EM:

1. Initialize the parmeters of the gaussian mixture distribution $\theta^0 = (\boldsymbol{\theta_k}, \boldsymbol{\pi})$
2. Until convergence, do the following:
    1. E-step - Compute the terms inside the auxilary function $Q(\theta, \theta^{(t-1)})$, i.e, the responsibilities r_{ik}
    2. M-step - Compute the updated parameters $(\pi_k, \boldsymbol{\mu}_k, \mathbf{\Sigma}_k)$ for each cluster k of the Gaussian Mixture distribution

Part 1 - Defining the class for GMM

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sklearn.cluster import KMeans as SklKMeans
```

```
class GMM:
    def __init__(self,
        n_components: int, n_iters: int, tol: float,
        random_state: int=0, init_params="random"
    ):
        self.n_components = n_components
        self.n_iters = n_iters
        self.tol = tol
        self.init_params = init_params
        self.random_state = random_state


    def fit(self, X,plot=False,plot_params={}):
        m,n= X.shape
        self.X = X
        k = self.n_components

        self.resp = np.zeros((m, k))
        self.weights = np.full(k, 1 / k)


        if self.init_params == "kmeans":
            kmeans = SklKMeans(k,random_state=self.random_state)
            kmeans.fit(X)
            self.means = kmeans.cluster_centers_
        if self.init_params == "random" :
            np.random.seed(self.random_state)
            self.means = X[np.random.choice(m, k , replace=False)]


        self.covs = np.full((k, n, n), np.cov(X, rowvar=False))

        self.converged = False
        self.log_likelihood_trace = []

        if plot:
            fig,ax = plt.subplots(1,4,figsize=(20,5))
            fig.suptitle("GMM Using Estimation and Maximization")
            self.draw(ax[0],"Initial Clusters", **plot_params)

        for i in range(self.n_iters):
```

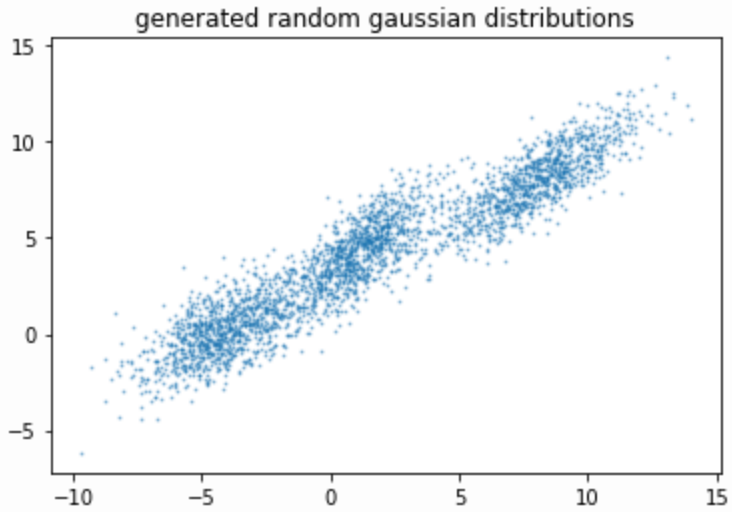
```
self._do_mstep(X)
            self.log_likelihood_trace.append(self.log_likelihood)
            if i == 0:
                if plot:
                    self.draw(ax[1],"Clusters after 1 iteration", **plot_params)
            else:
                ol,nl = self.log_likelihood_trace[-2:]
                if nl-ol <= self.tol:  #likelihood difference less than tol
                    self.converged = True
                    break
        if plot:
            self.draw(ax[2],f"clusters after {i+1} iterations", **plot_params)
            ax[3].plot(self.log_likelihood_trace, "-o")
            ax[3].set_title("Log Likelihood")
            plt.show()

    def _do_estep(self, X):
        for k in range(self.n_components):
            self.resp[:, k] = self.weights[k] * multivariate_normal(self.means[k], self.covs[k]).pdf(X
        p_x = np.sum(self.resp, axis=1,keepdims=1)
        self.resp = self.resp / p_x
        self.log_likelihood = np.sum(np.log(p_x))

    def _do_mstep(self, X):
        resp_weights = self.resp.sum(axis=0)
        self.weights = resp_weights / X.shape[0]
        self.means = self.resp.T @ X/ resp_weights.reshape(-1, 1)
        diff = X[:,np.newaxis,:] - self.means
        self.covs = np.einsum('ik,ikj,ikl->kjl',self.resp,diff,diff)/resp_weights.reshape(-1,1,1)


    def draw(self,ax,  title="", **plot_params):
        ax.set_title(title)
        ax.scatter(self.X[:, 0], self.X[:, 1],**plot_params)

        delta = 0.05
        k = self.n_components
        x = np.arange(*ax.get_xlim(), delta)
        y = np.arange(*ax.get_ylim(), delta)
        x, y = np.meshgrid(x, y)
        col = [f"C{(i+1)%10}" for i in range(k)]
        for i in range(k):
            mean, cov = self.means[i], self.covs[i]
            z = multivariate_normal(mean, cov).pdf(np.dstack([x, y]))
            ax.scatter(mean[0], mean[1], color=col[i])
            ax.contour(x, y, z, levels=[.01], colors=col[i])
```

Part 2 - Generating Random data from a Gaussian Mixture

```
def gen_data(k=3, dim=2, points_per_cluster=200, lim=[-10, 10],seed = 1):
    np.random.seed(seed)
    x = np.ndarray((k,points_per_cluster,dim))
    mean = np.random.rand(k, dim)*(lim[1]-lim[0]) + lim[0]
    for i in range(k):
        cov = np.random.rand(dim, dim+10)
        cov = cov @ cov.T
        x[i] = np.random.multivariate_normal(mean[i], cov, points_per_cluster)
    x = x.reshape(-1,dim)

    if(dim == 2):
        plt.figure()
        plt.title("generated random gaussian distributions")
        plt.scatter(x[:, 0], x[:, 1], s=1, alpha=0.4)
        plt.show()

    return x
```

```
X = gen_data(k=3, dim=2, points_per_cluster=1000, seed=3)
```

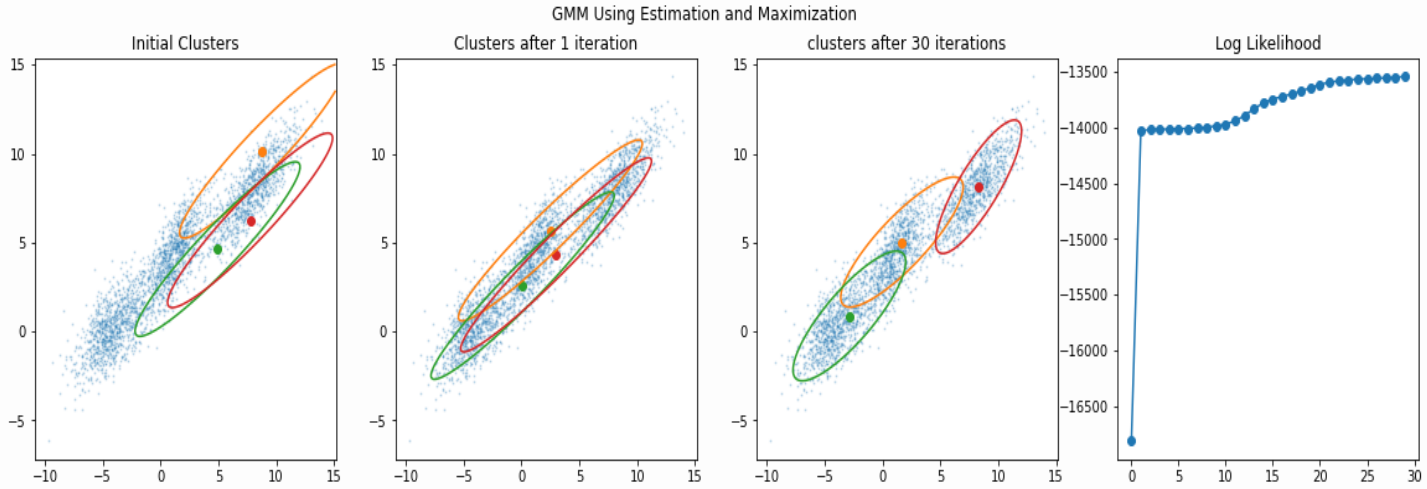
Output:



Part 3 - Implementing GMM on generated random data

```
gmm = GMM(n_components =3,n_iters=30,tol=10e-4, random_state=5, init_params="random")
gmm.fit(X, plot= True,plot_params={"s":1,"alpha":.2})
```

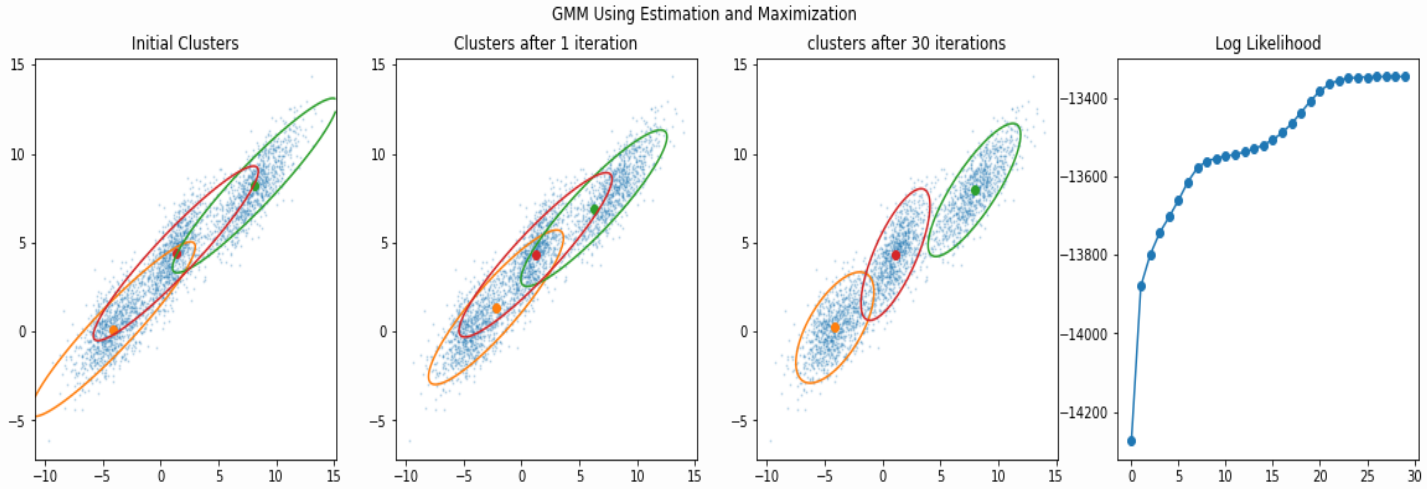
Output:



Part 4 - Implementing GMM by using kmeans to initialize means

```
k = 3
gmm = GMM(n_components = k,n_iters = 30,tol=10e-4,random_state=1, init_params="kmeans")
gmm.fit(X,plot=True,plot_params={"s":1,"alpha":.2})
```

Output:



5. a) Backpropagation Neural Network

Neural Network:

A neural network is a machine that is designed to model the way in which the brain performs a particular task or functions; the network is usually implemented by using electronic components or is simulated in software on a digital computer. To achieve good performance, neural networks employs a massive interconnections of simple computing cells referred to as “neurons” or “processing units”. Thus the definition of neural network is “A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural capability for storing experimental knowledge and making it available for use”.

Backpropagation neural network (BPNN):

Backpropagation neural network (BPNN) is a multi layer feedforward neural network (i.e., propagating the error backward to adjust the weights). The basic idea is to efficiently compute partial derivatives of an approximating function f(W,X) realized by the network with respect to all the elements of the adjustable weights vector W for a given value of input vector X. Fig. 1. shows the architecture of backpropagation neural network.

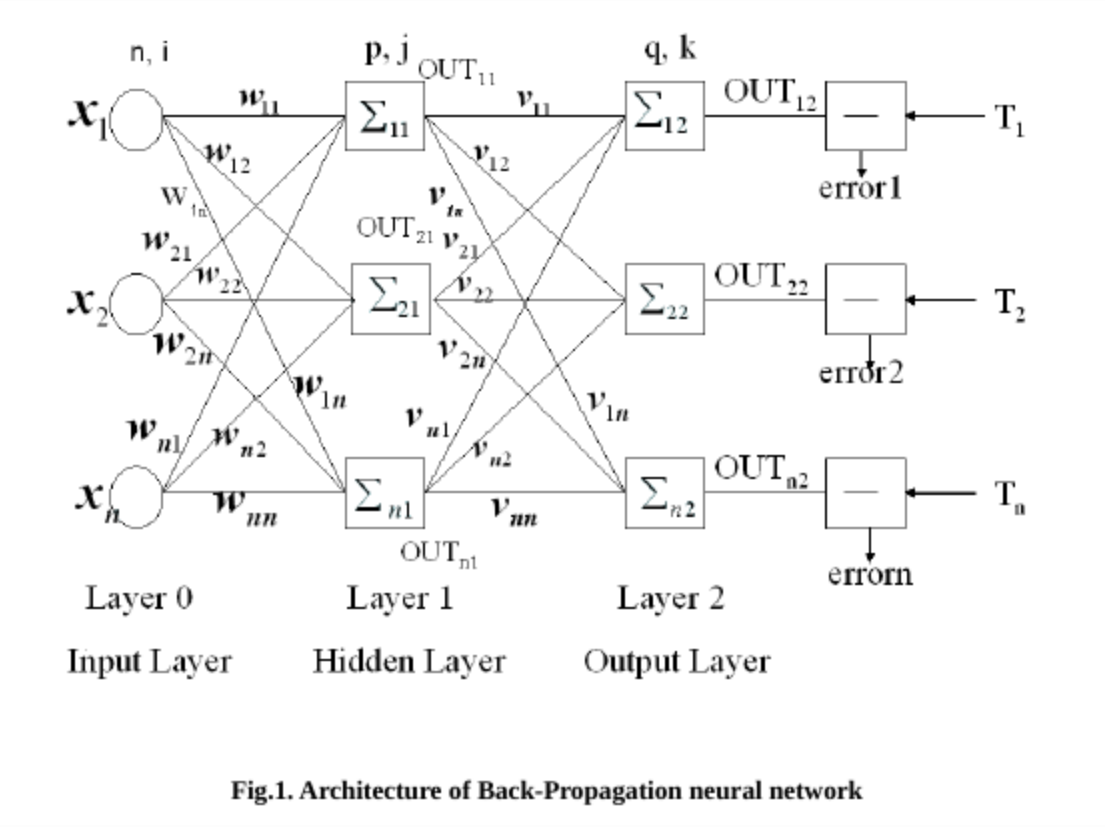


Fig.1. Architecture of Back-Propagation neural network

Backpropagation Training Algorithm:

1. Select the next training pair from the training set and apply to the network.
2. Calculate the output of the network.
3. Calculate the error between the output of the network and the desired output.
4. Adjust the weights (V,W matrix) in such a way that it minimize the error.
5. Repeat steps 1 to 4 for all the training pairs.
6. Repeat steps 1 to 5 until the network recognizes the training set or for certain number of iterations called epochs. 1 The activation function used by BPNN training algorithm is sigmoid or squashing or logistic function, and it is defined as

$$\mathrm{OUT} = \frac{1}{(1 + e^{-\mathrm{NET}})}$$

BPNN training algorithm uses the derivative of activation function and defined as

$$\frac{\partial \mathrm{OUT}}{\partial \mathrm{NET}} = \mathrm{OUT}(1 - \mathrm{OUT})$$

BPNN training algorithm consists of two passes: (i)Forward pass and (ii)Reverse pass.

(i)Forward pass:

In this pass, output of the network is calculated as,

$$\begin{aligned}
&\mathrm{NET}_{1j} = x_1 w_{11} + x_2 w_{21} + \ldots + x_n \\
&\mathrm{OUT}_{1j} = 1/\left(1 + e^{-\mathrm{NET}_{1j}}\right) \\
&\mathrm{NET}_{1k} = \mathrm{OUT}_{11 V_{11}} + \mathrm{OUT}_{21 V_{21}} + \ldots + \mathrm{OUT}_{n1 V_{n1}} \\
&\mathrm{OUT}_{1k} = 1/\left(1 + e^{\mathrm{a}} \mathrm{NET}_{1k}\right)
\end{aligned}$$

This is repeated for all the neurons.

(ii)Reverse pass:

This pass consist of two parts. They are

(a) Adjusting the weights of the output layer:

To adjust the weights of the output layer generalized delta rule is used.

$$\delta_{qk} = \mathrm{OUT}_{qk}\left(1 - \mathrm{OUT}_{qk}\right)\left(\mathrm{Target} - \mathrm{OUT}_{qk}\right)$$

The new weight of the V matrix is calculated as

$$V_{pq}(n+1) = V_{pq}(n) + \eta \delta_{qk} \mathrm{OUT} pj$$

where

$V_{pq}(n+1)$ is new weight,

$V_{pq}(n)$ is old weight and

η is learning or training rate coefficient.

(b)Adjusting the weights of the hidden layer:

$$\delta_{pj} = \mathrm{OUT}_{pj}\left(1 - \mathrm{OUT}_{pj}\right) \sum_{q=1}^{n} \delta_{qk} V_{pq}$$

Where

δ_{pj} is error for neuron p in the hidden layer j,

OUT_{pj} is output of neuron p in the hidden layer j,

δ_{qk} is error neuron q in the output layer k and

V_{pq} is weight from neuron p in the hidden layer to neuron q in the output layer.

The new weight of the W matrix is calculated as

$$W_{mp}(n+1) = W_{mp} + \eta \delta_{pj} x_m$$

where

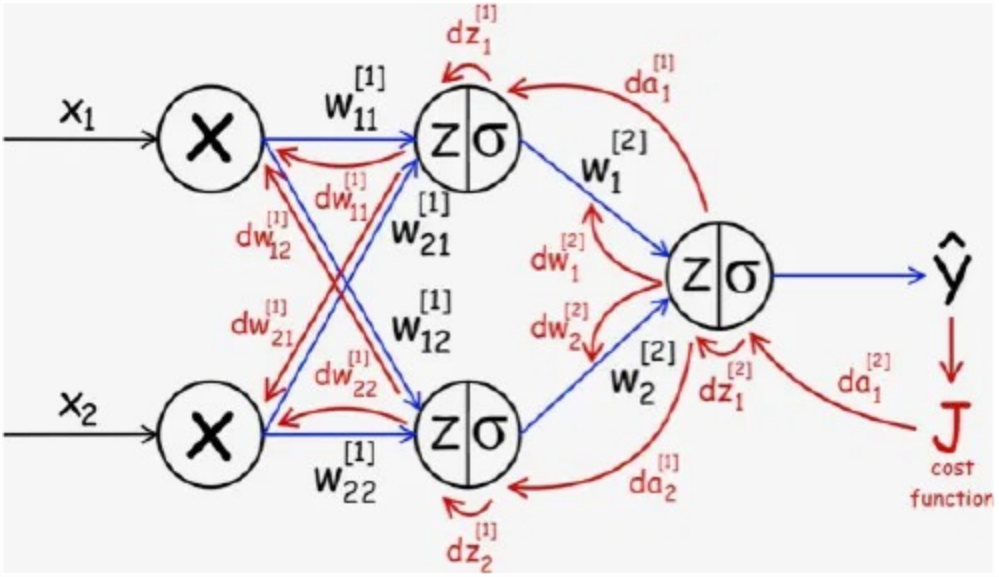
$W_{mp}(n+1)$ is new weight,

W_{mp} is old weight and

η is learning or training rate coefficient.

With these Paramaters and functions we can implement BPNN algorithm using python.

BackPropagation implementation model in Detail



BPNN Implementation Algorithm

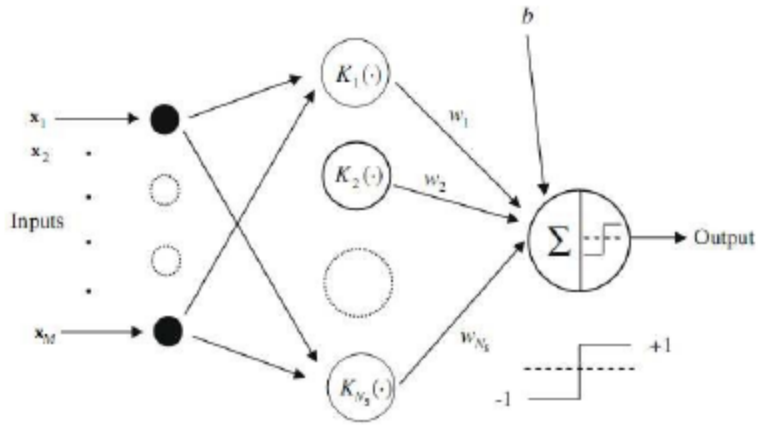
- Initialize $W^{[1]} \dots W^{[L]}, b^{[1]} \dots b^{[E]}$
- Set $A^{[0]} = X$ (Input),$L =$ Total Layers
- Loop epoch $= 1$ to max iteration
  - Forward Propagation
    - Loop $l = 1$ to $L - 1$
      - $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
      - $A^{[L]} = g\left(b^{l}\right)$
      - Save $A^{[n]}, W^{[l]}$ in memory for later use
    - $Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$
    - $A^{L]} = \sigma\left(Z^{[L]}\right)$
  - Cost $J = -\frac{1}{n}\left(Y \log\left(A^{[2]}\right) - (1 - Y) \log\left(1 - A^{[2]}\right)\right)$
  - Backward Propagation
    - $dA^{[L]} = -\dfrac{Y}{A^{[L]}} + \dfrac{1 - Y}{1 - A^{[L]}}$
    - $dZ^{[L]} = dA^{[L]} \sigma'\left(dA^{[L]}\right)$
    - $dW^{[L]} = dZ^{[L]} dA^{[L-1]}$
    - $db^{[L]} = dZ^{LL}$
    - $dA^{[L-1]} = dZ^{[L]} W^{[L]}$
    - Loop $l = L - 1$ to 1
      - $dZ^{[l]} = dA^{[\eta]} g'\left(dA^{[l]}\right)$
      - $dW^{[l]} = dZ^{[l]} dA^{(l-1]}$
      - $db^{[l]} = dZ^{[l]}$
      - $dA^{[l-1]} = dZ^{[l]} W^{[r]}$
  - Update W and b
    - Loop $l = 1$ to L
      - $W^{[l]} = W^{[l]} - \alpha \cdot dW^{[l]}$
      - $b^{[l]} = b^{[l]} - \alpha \cdot db^{[l]}$

5. b) Support Vector Machine

Support vector machine (SVM) is based on the principle of structural risk minimization (SRM). Like RBFNN, support vector machines can be used for pattern classification and nonlinear regression. SVM constructs a linear model to estimate the decision function using non-linear class boundaries based on support vectors. If the data are linearly separated, SVM trains linear machines for an optimal hyperplane that separates the data without error and into the maximum distance between the hyperplane and the closest training points. The training points that are closest to the optimal separating hyperplane are called support vectors. Fig shows the architecture of the SVM. SVM maps the input patterns into a higher dimensional feature space through some nonlinear mapping chosen a priori. A linear decision sur.face is then constructed in this high dimensional feature space. Thus, SVM is a linear classifier in the parameter space, but it becomes a nonlinear classifier as a result of the nonlinear mapping of the space of the input patterns into the high dimensional feature space, SVM also supports the kernel method also called the kernel SVM which allows us to tackle non-linearity.

SVM Principle

Support vector machine (SVM) ban be used for classifying the obtained data. SVM are a set of related supervised learning methods used for classification and regression. They belong to a family of generalized linear classifiers. Let us denote a feature vector (termed as pattern) by $x = (x_1, x_2, \ldots, x_n)$ and its class label by y such that $y = \{-1, +1\}$. Therefore, consider the problem of separating the set of n-training patterns belonging to two classes.

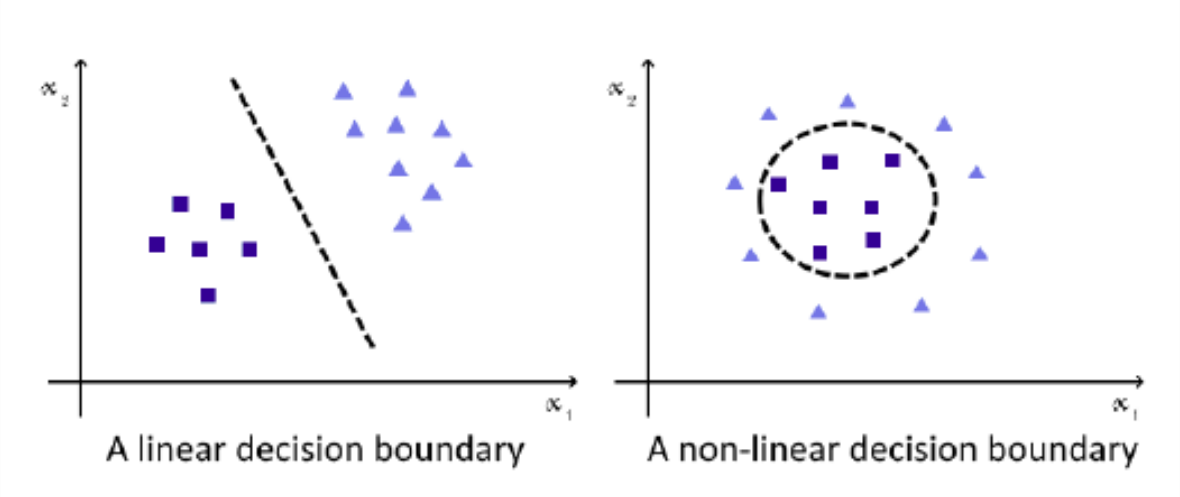


Architecture of the SVM (N s is the number of support vectors)

$$(x_i, y_i), \quad x_i \in \mathbb{R}^{\ltimes}, \quad y = \{+1, -1\}, \quad i = 1, 2, \ldots, n$$

A decision function $g(x)$ that can correctly classify an input pattern x that is not necessarily from the training set.

SVM for Linearly Separable Data



A linear SVM is used to classify data sets which are linearly separable. The SVM linear classifier tries to maximize the margin between the separating hyperplane. The patterns lying on the maximal margins are called support vectors. Such a hyperplane with maximum margin is called maximum margin hyperplane. In case of linear SVM, the discriminant function is of the form:

$$g(x) = w^t x + b$$

such that $g(x_i) \geq 0 \quad \text{for} y_i = +1$ and $g(x_i) \leq 0 \quad \text{for} y_i = -1$. In other words, training samples from the two different classes are separated by the hyperplane $g(x) = w^t x + b = 0$. SVM finds the hyperplane that causes the largest separation between the decision function values from the two classes. Now the total width between two margins is $\frac{2}{w^t w}$, which is to be maximized. Mathematically, this hyperplane can be found by minimizing the following cost function:

$$J(w) = \frac{1}{2} w^t w$$

Subject to separability constraints

$$\begin{aligned} g(x_i) &\geq +1 \quad \text{for} \quad y_i = +1 \\ g(x_i) &\leq -1 \quad \text{for} \quad y_i = -1 \end{aligned}$$

Equivalently, these constraints can be re-written more compactly as

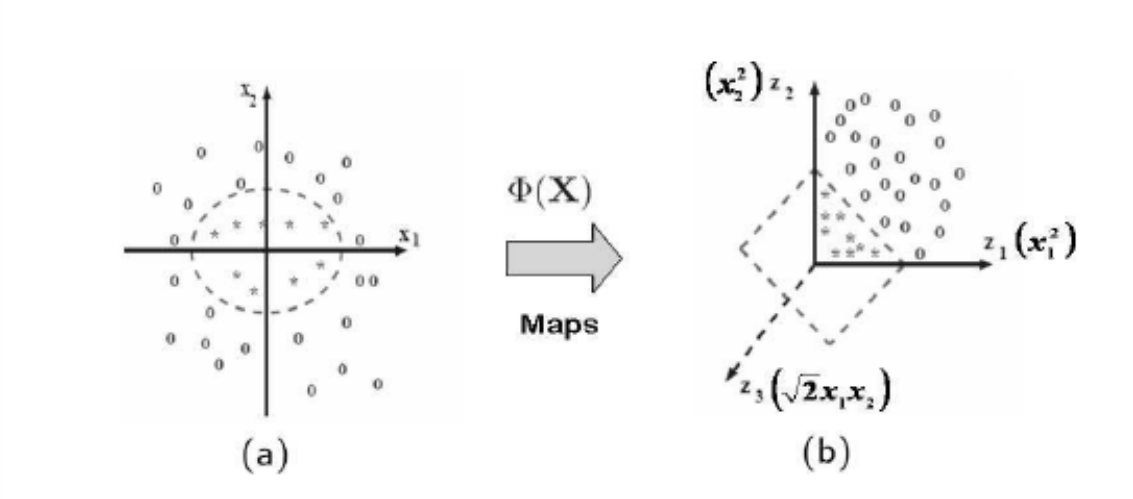
$$y_i(w^t x_i + b) \geq 1; \quad i = 1, 2, \ldots, n$$

For the linearly separable case, the decision rules defined by an optimal hyperplane separating the binary decision classes are given in the following equation in terms of the support vectors:

$$Y = \text{sign}\left(\sum_{i=1}^{i=N_s} y_i \alpha_i \left(x x_i \right) + b \right)$$

where Y is the outcome, y_i is the class value of the training example x_i, and represents the inner product. The vector corresponds to an input and the vectors $x_i, \quad i = 1, \ldots, N_s$ are the support vectors. In Eq. 10.4, b and α i are parameters that determine the Hyperplane.

SVM for Linearly Non-separable Data



For non-linearly separable data, it maps the data in the input space into a high dimension space $x \in \mathbb{R}^I \mapsto \Phi(x) \in \mathbb{R}^H$ with kernel function $\Phi(x)$, to find the separating hyperplane. A high-dimensional version of Eq. 10.4 is given as follows:

$$Y = \text{sign}\left(\sum_{i=1}^{i=N} y_i \alpha_i K\left(x, x_i\right) + b\right)$$

Kernel

The learning of the hyperplane in linear SVM is done by transforming the problem to higher dimension space. This is where the kernel plays role.The function K is defined as the kernel function for generating the inner products to construct machines with different types of non-linear decision surfaces in the input space.

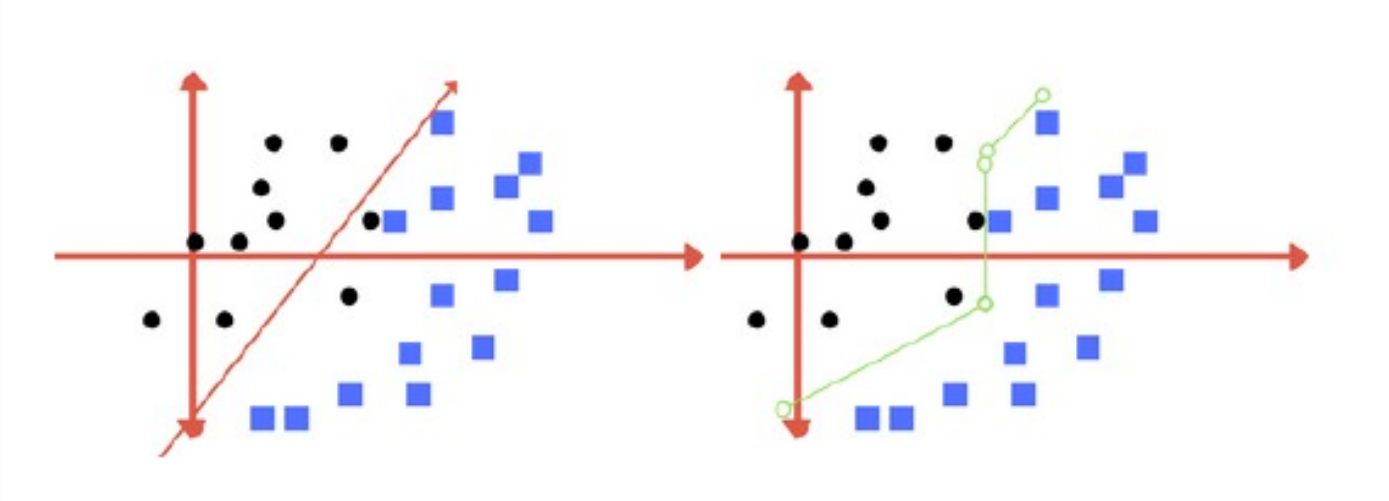
$$K(x, x_i) = \Phi(x) \cdot \Phi(x_i)$$

An example for SVM kernel function $\Phi(x)$ maps 2-dimensional input space (x_1, x_2) to higher 3- dimensional feature space $(x_1^2, x_2^2, \sqrt{2}x_1, x_2)$ .The kernel function may be any of the symmetric functions that satisfy the Mercer's conditions
For **linear kernel** the equation for prediction for a new input using the dot product between the input (x) and each support vector (x_i) is calculated
The **polynomial kernel** $K(x^T x_i + 1)^p$ Polynomial and exponential kernels calculates separation line in higher dimension. This is called **kernel trick**

Tuning parameters:Regularization, Gamma and Margin

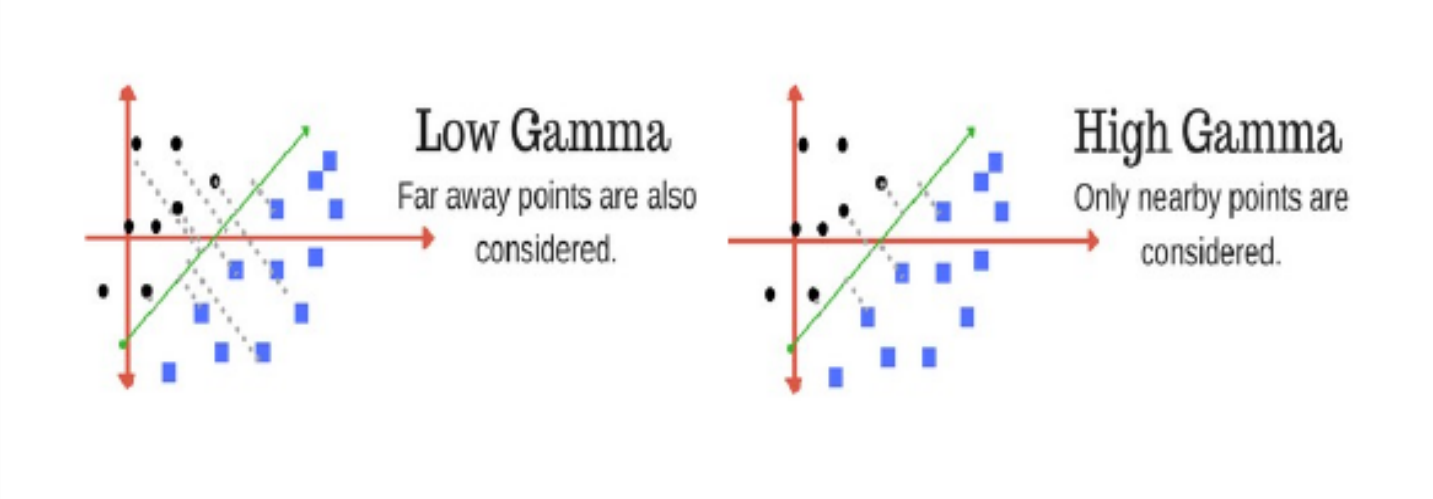
Regularization

The Regularization parameter (often termed as C parameter in python's sklearn library) tells the SVM optimization how much we want to avoid misclassifying each training example. The images below (same as image 1 and image 2 in section 2) are example of two different regularization parameter. Left one has some misclassification due to lower regularization value. Higher value leads to results like right one.



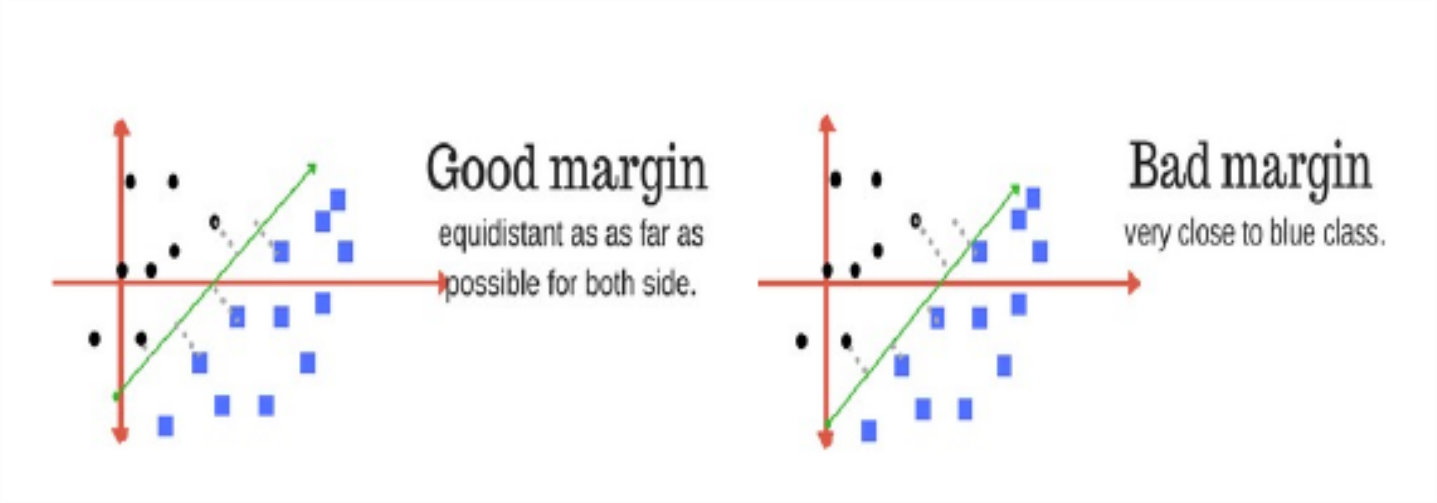
Gamma

The gamma parameter defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. In other words, with low gamma, points far away from plausible seperation line are considered in calculation for the seperation line. Where as high gamma means the points close to plausible line are considered in calculation.



Margin

And finally last but very importrant characteristic of SVM classifier. SVM to core tries to achieve a good margin. A *margin* is a separation of line to the *closest class points*. A *good margin* is one where this separation is larger for both the classes. Images below gives to visual example of good and bad margin. A good margin allows the points to be in their respective classes without crossing to other class.



SVMs, always guarantee convergence due to no restricition on dimensionality of solution space.

Implementation:

From the dataset. 1. Splitting the dataset into training and test samples. 2. Classifying the predictors and target. 3. Initializing **Support Vector Machine** and fitting the training data. 4. Predicting the classes for test set. 5. Attaching the predictions to test set for comparision.

Exercise 5 - Classification Algorithms (BPNN & SVM)

a) Back Propogation Neural Network

Program 1 - Implementing BPNN from scratch

AIM

To implement Back Propogation Neural Network from scratch in python.

ALGORITHM

Forward propogation

Single input

$$\begin{aligned}\mathbf{o}_{-1} &= \mathbf{x}_i \\ \mathbf{o}_k &= \phi_k\left(\mathbf{W}_k\begin{bmatrix}1 \\ \mathbf{o}_{k-1}\end{bmatrix}\right) \quad \text{for } k = 0,..,n-1\end{aligned}$$

Batch input

$$\begin{aligned}\mathbf{O}_{-1} &= \mathbf{X}^T \\ \mathbf{O}_k &= \phi_k\left(\mathbf{W}_k^T\begin{bmatrix}\mathbf{1} \\ \mathbf{O}_{k-1}\end{bmatrix}\right) \quad \text{for } k = 0,\ldots,n-1\end{aligned}$$

$$\begin{aligned}\mathbf{O}_{-1} &= \mathbf{X}^T \\ \mathbf{O}_k &= \phi_k\left(\mathbf{W}_k^T\begin{bmatrix}\mathbf{1} \\ \mathbf{O}_{k-1}\end{bmatrix}\right) \quad \text{for } k = 0,..,n-1\end{aligned}$$

Error

Single Input

$$E = \frac{1}{2}||\mathbf{o}_{n-1} - \mathbf{t}||_2$$

####### Batch Input

$$E = \frac{1}{2m}\sum_{i=1}^{m}||\mathbf{o}_{n-1}^{(i)} - \mathbf{t}||_2$$

Backward Propogation

Single input

$$\begin{aligned}\boldsymbol{\delta}_k &= \frac{\mathrm{d}E}{\mathrm{d}\mathbf{o}_k} \circ \boldsymbol{\phi}'(\mathrm{net}_k) \\ \frac{\mathrm{d}E}{\mathrm{d}\mathbf{o}_k} &= \begin{cases} (\mathbf{o}_{n-1} - \mathbf{t}) & k = n-1 \\ \mathbf{W}_{k+1}^T \boldsymbol{\delta}_{k+1} & k = n-2, \ldots, -1 \end{cases} \\ \boldsymbol{\phi}'(\mathrm{net}_k) &= \mathbf{o}_k \circ (\mathbf{1} - \mathbf{o}_k) \text{ for } k = n-1, \ldots, 0\end{aligned}$$

Batch input

$$\begin{aligned}\boldsymbol{\Delta}_k &= \frac{\mathrm{d}E}{\mathrm{d}\mathbf{O}_k} \circ \boldsymbol{\phi}'(\mathrm{net}_k) \\ \frac{\mathrm{d}E}{\mathrm{d}\mathbf{O}_k} &= \begin{cases} (\mathbf{O}_{n-1} - \mathbf{T}) & k = n-1 \\ \mathbf{W}_{k+1}^T \boldsymbol{\Delta}_{k+1} & k = n-2, \ldots, -1 \end{cases} \\ \boldsymbol{\phi}'(\mathrm{net}_k) &= \mathbf{O}_k \circ (\mathbf{1} - \mathbf{O}_k) \text{ for } k = n-1, \ldots, 0\end{aligned}$$

Weight Update

Single Input

$$\begin{aligned}\mathbf{W}_k &:= \mathbf{W}_k - \alpha \frac{\mathrm{d}E}{\mathrm{d}\mathbf{W}_k} \\ \frac{\mathrm{d}E}{\mathrm{d}\mathbf{W}_k} &= \boldsymbol{\delta}_k \begin{bmatrix} 1 \\ \mathbf{o}_{k-1} \end{bmatrix}^T \text{ for } k = n-1, \ldots, 0\end{aligned}$$

Batch Input

$$\begin{aligned}\mathbf{W}_k &:= \mathbf{W}_k - \alpha \frac{\mathrm{d}E}{\mathrm{d}\mathbf{W}_k} \\ \frac{\mathrm{d}E}{\mathrm{d}\mathbf{W}_k} &= \frac{1}{m} \boldsymbol{\Delta}_k \begin{bmatrix} 1 \\ \mathbf{O}_{k-1} \end{bmatrix}^T \text{ for } k = n-1, \ldots, 0\end{aligned}$$

where
$\mathbf{x}_i$ - the input vector of the ith data point $\mathbf{o}_k$ - the output vector of the kth layer
$\boldsymbol{\delta}_k$ - the delta vector of the kth layer
$\mathbf{W}_k$ - the weight matrix of the kth layer
$\mathbf{X}$ - the batch input matfix
$\mathbf{O}_k$ - the batch output matrix of the kth layer
$\boldsymbol{\Delta}_k$ - the batch delta matrix of the kth layer

Part 1 - Defining class for BPNN

```
import numpy as np
import pandas as pd
```

```
class BPNN:
    def __init__(
        self,hidden_layer_sizes,
        l_rate=.001,
        n_epoch=20,
        batch_size=20,
        random_state=0
    ):
        self.hidden_layer_sizes = hidden_layer_sizes
        self.l_rate= float(l_rate)
        self.n_epoch = int(n_epoch)
        self.batch_size = int(batch_size)
        self.random_state = random_state

        self.n = len(hidden_layer_sizes)+1
        self.outputs = [None]*(self.n+1)
        self.delta = [None]*self.n


    def activation(self,x):
        return 1/(1+np.exp(-x))

    def d_activation(self,x):
         return x * (1 - x)

    def pad_ones(X):
        pad_width = [(1,0),(0,0)]
        return np.pad(X,pad_width=pad_width,constant_values=1)

    def inputs(self,i):
        return BPNN.pad_ones(self.outputs[i-1])

    def initialize_random_weights(self):
        np.random.seed(self.random_state)
        self.weights = [
            np.random.standard_normal((o,i+1))*.001
            for i,o in zip(
                [self.n_inputs]+self.hidden_layer_sizes,
                self.hidden_layer_sizes+[self.n_outputs]
            )
        ]

    def forward_propagate(self, inputs):
```

```
self.outputs[-1] = inputs
        for i in range(self.n):
            self.outputs[i] = self.activation(
                self.weights[i] @ self.inputs(i)
            )
        return self.outputs[self.n-1]

    def backward_propagate_error(self, target):
        for i in range(self.n-1, -1, -1):
            if i == self.n-1:
                errors = self.outputs[i] - target
            else:
                errors = self.weights[i+1][:,1:].T @ self.delta[i+1]
            self.delta[i] = errors * self.d_activation(self.outputs[i])

    def update_weights(self):
        for i in range(self.n):
            self.weights[i] -= self.l_rate * self.delta[i] @ self.inputs(i).T/self.batch_size

    def error(self,actual,predicted):
        error = actual-predicted
        return np.sum(error * error)

    def fit(self, X_train,y_train,verbose=False):
        m = X_train.shape[0]
        input_matrix = X_train.T
        classes = np.unique(y_train)
        target_matrix = (y_train.reshape(-1,1) == classes).T
        self.n_inputs = input_matrix.shape[0]
        self.n_outputs = target_matrix.shape[0]
        self.initialize_random_weights()
        if verbose:
            print("Initial Weights:")
            print(*self.weights,sep="\n")
            print("Training:")

        for epoch in range(self.n_epoch):
            sum_error = 0
            fs = range(m+self.batch_size)
            for f,t in zip(fs,fs[1:]):
                outputs = self.forward_propagate(input_matrix[:,f:t])
                sum_error += self.error(target_matrix[:,f:t],outputs)
                self.backward_propagate_error(target_matrix[:,f:t])
                self.update_weights()
            if verbose:
                print(f'> epoch={epoch+1}, lrate={self.l_rate:.3}, error={sum_error/m:.5f}')
        if verbose:
            print("Trained Weights:")
            print(*self.weights,sep="\n")
```

```
def predict(self, inputs):
        outputs = self.forward_propagate(inputs.T).T.argmax(axis=-1)
        return outputs

    def accuracy(self,X_test,y_test):
        return (self.predict(X_test)==y_test).mean()
```

Part 2 - Loading and processing dataset

```
titanic_df = pd.read_csv("datasets/titanic_processed.csv")
X = titanic_df.drop('Survived',axis = 1).values
y = titanic_df['Survived'].values
split_ratio = 0.8
s = int(split_ratio*X.shape[0])
X_train, X_test = X[:s],X[s:]
y_train, y_test = y[:s],y[s:]
```

Part 3 - Implementing BPNN

```
b = BPNN(
    hidden_layer_sizes = [5,3],
    l_rate=30, n_epoch=20, batch_size=25,random_state=10
)
b.fit(X_train,y_train,verbose=True)
print("Evaluation:")
print(f"Acuracy of the classifier: {b.accuracy(X_test,y_test)*100:.2f}%")
```

Output:

```
Initial Weights:
[[ 1.33158650e-03  7.15278974e-04 -1.54540029e-03 -8.38384993e-06
   6.21335974e-04 -7.20085561e-04  2.65511586e-04  1.08548526e-04
   4.29143093e-06 -1.74600211e-04]
 [ 4.33026190e-04  1.20303737e-03 -9.65065671e-04  1.02827408e-03
   2.28630130e-04  4.45137613e-04 -1.13660221e-03  1.35136878e-04
   1.48453700e-03 -1.07980489e-03]
 [-1.97772828e-03 -1.74337230e-03  2.66070164e-04  2.38496733e-03
   1.12369125e-03  1.67262221e-03  9.91492158e-05  1.39799638e-03
  -2.71247988e-04  6.13204185e-04]
 [-2.67317189e-04 -5.49309014e-04  1.32708296e-04 -4.76142015e-04
   1.30847308e-03  1.95013279e-04  4.00209988e-04 -3.37632337e-04
   1.25647226e-03 -7.31969502e-04]
 [ 6.60231551e-04 -3.50871891e-04 -9.39433360e-04 -4.89337217e-04
  -8.04591142e-04 -2.12697639e-04 -3.39140246e-04  3.12169936e-04
   5.65152670e-04 -1.47420258e-04]]
[[-2.59053368e-05  2.89094204e-04 -5.39879071e-04  7.08160020e-04
   8.42224738e-04  2.03580797e-04]
 [ 2.39470366e-03  9.17458938e-04 -1.12272471e-04 -3.62180447e-04
```

```
[ 1.12878515e-03 -6.97810030e-04 -8.11221838e-05 -5.29296081e-04
   1.04618286e-03 -1.41855603e-03]]
[[-0.0003625  -0.00012191  0.00031936  0.0004609 ]
 [-0.00021579  0.00098907  0.00031475  0.00246765]]
Training:
> epoch=1, lrate=30.0, error=0.49515
> epoch=2, lrate=30.0, error=0.49066
> epoch=3, lrate=30.0, error=0.44759
> epoch=4, lrate=30.0, error=0.32836
> epoch=5, lrate=30.0, error=0.32012

.
.
.
[[-2.60724793  1.55437491  1.68003918  1.61398662]
 [ 2.60724975 -1.55498627 -1.68088901 -1.61254007]]
Evaluation:
Acuracy of the classifier: 85.96%
```

b) Support Vector Machine

Program 1 - Implementing SVM from scratch using CVXOPT

AIM

To implement SVM from scratch using CVXOPT.

Formula

SVM statement in dual form

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \frac{1}{2}\boldsymbol{\alpha}^T \left(\mathbf{y}\mathbf{y}^T \circ (\Phi(\mathbf{X})\Phi(\mathbf{X})^T\right) \boldsymbol{\alpha} - \mathbf{1}^T\boldsymbol{\alpha} \\ s.t. -\alpha_i \leq 0 \\ \alpha_i \leq C \\ y^T\boldsymbol{\alpha} = 0 \end{aligned}$$

Standard form of a quadratic program

$$\begin{aligned} \min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{P}\mathbf{x} - \mathbf{q}^T\mathbf{x} \\ s.t. \mathbf{G}\mathbf{x} \leq \mathbf{h} \\ A\mathbf{x} = \mathbf{b} \end{aligned}$$

Formulization

$$\begin{aligned} \mathbf{P} &= \mathbf{y}\mathbf{y}^T \circ \Phi(\mathbf{X})\Phi(\mathbf{X})^T & \mathbf{q} &= -\mathbf{1}_n \\ \mathbf{G} &= \begin{bmatrix} -\mathbf{I}_n \\ \mathbf{I}_n \end{bmatrix} & \mathbf{h} &= \begin{bmatrix} \mathbf{0}_n^T \\ C \cdot \mathbf{1}_n^T \end{bmatrix} \\ \mathbf{A} &= [\mathbf{y}] & \mathbf{b} &= [0] \end{aligned}$$

Radial Basis Function

$$\Phi(\mathbf{x}_i)\Phi(\mathbf{x}_j)^T = \exp\left(\frac{-||\mathbf{x}_i - \mathbf{x}_j||_2^2}{2\sigma^2}\right)$$

Decision Rule

$$\hat{\mathbf{y}} = \text{sign}\left((\boldsymbol{\alpha}^T \circ \mathbf{y}^T)\Phi(\mathbf{X})\Phi(\hat{\mathbf{X}})^T + b\right)$$

Part 1 - Defining class for SVM

```
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

from cvxopt import solvers
from cvxopt import matrix

from scipy.spatial.distance import cdist
```

```
class SVM:
    def __init__(self,C,kernel):
        self.C = C
        self.kernel = kernel

    def fit(self,X_train,y_train):
        self.scaler = StandardScaler()
        self.X = self.scaler.fit_transform(X_train)
        self.y = y_train.reshape(-1,1)

        n=self.X.shape[0]
        I_n = np.eye(n)
        P=(self.y@self.y.T)*self.kernel(self.X,self.X)
        q=np.full(n,-1)
        G=np.vstack((-1*I_n,I_n))
        h=np.hstack((np.zeros(n),np.full(n,self.C)))
        A=y_train.reshape(1,-1)
        b=np.zeros(1)

        P,q,G,h,A,b = map(lambda x : matrix(x,tc="d"),(P,q,G,h,A,b))

        solution = solvers.qp(P, q, G, h, A, b)
        self.a = np.asarray(solution['x']).squeeze()

        support_indices = np.logical_and(self.a>=1e-10, self.a<self.C)
        X_S = self.X[support_indices]
        self.b = np.mean(self.y - self.a*self.y.T @ self.kernel(self.X, X_S))

    def predict(self,X_test):
        X_test=self.scaler.transform(X_test)
        return np.sign(self.a*self.y.T @ self.kernel(self.X, X_test) + self.b)
```

Part 2 - Defining Radial Basis Function(RBF) Kernel

```
def rbf_kernel(X1,X2,sigma):
    return np.exp(-cdist(X1, X2, 'sqeuclidean') / (2*sigma**2))
```

Part 3 - Loading and Processing Dataset

```
titanic_df = pd.read_csv('datasets/titanic_processed.csv')
X = titanic_df.drop('Survived',axis = 1).values
y = titanic_df['Survived'].values
y[y==0] = -1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Part 4 - Implementing SVM

```
from functools import partial
C = .35
sigma = 2.5
kernel = partial(rbf_kernel,sigma=sigma)

svm_classifier = SVM(C,kernel)
svm_classifier.fit(X_train,y_train)
y_pred = svm_classifier.predict(X_test)
print(f'Accuracy of the classifer: {(y_test == y_pred).mean()*100:.2f}%')
```

Output:

```
     pcost       dcost       gap    pres   dres
 0: -2.1166e+02 -5.8558e+02  6e+03  8e+00  4e-15
 1: -8.8848e+01 -4.9225e+02  6e+02  4e-01  3e-15
 2: -8.4608e+01 -1.3669e+02  5e+01  2e-03  2e-15
 3: -9.2566e+01 -1.1007e+02  2e+01  5e-04  1e-15
 4: -9.5567e+01 -1.0239e+02  7e+00  1e-04  1e-15
 5: -9.6721e+01 -9.9922e+01  3e+00  6e-05  1e-15
 6: -9.7365e+01 -9.8738e+01  1e+00  8e-06  2e-15
 7: -9.7606e+01 -9.8266e+01  7e-01  2e-06  2e-15
 8: -9.7781e+01 -9.7964e+01  2e-01  2e-07  2e-15
 9: -9.7826e+01 -9.7894e+01  7e-02  4e-08  2e-15
10: -9.7850e+01 -9.7861e+01  1e-02  5e-09  2e-15
11: -9.7852e+01 -9.7857e+01  4e-03  7e-10  2e-15
12: -9.7854e+01 -9.7855e+01  1e-03  1e-10  1e-15
13: -9.7854e+01 -9.7855e+01  2e-04  2e-11  1e-15
14: -9.7854e+01 -9.7854e+01  4e-05  3e-12  2e-15
Optimal solution found.
Accuracy of the classifer: 75.28%
```

6. Decision Tree and Random Forest

Brief Introduction to Decision Trees

A decision tree is a supervised machine learning algorithm that can be used for both classification and regression problems. A decision tree is simply a series of sequential decisions made to reach a specific result. The features/attributes and conditions can change based on the data and complexity of the problem but the overall idea remains the same. So, a decision tree makes a series of decisions based on a set of features/attributes present in the data. Feature importance and the sequence of attributes were checked and decided on the basis of criteria like **Gini Impurity Index or Information Gain.** A decision tree is a visual representation of a problem. A decision tree helps decompose a complex problem into smaller, more manageable undertakings. This allows the decision-makers to make smaller determinations along the way to achieve the optimal overall decision. Decision tree analysis is a formal, structured approach to making decisions also intuitive to classify a pattern through a sequence of questions in which the next question is depends upon the answer to the current question. But this approach is useful for nonmetric data only. Because all the questions can be asked in a yes/no (or) true/false style.
Such a sequence of questions is displayed in a directed decision tree (or) tree. This tree can be used for classification. The classification of a particular pattern begins at the root node - which asks for the value of a particular property of the pattern.The root node has different possible value and so it has different links. Based on the answer we follow the appropriate link to a subsequent node. But we follow only one link. Continue the process until we reach a leaf node, which has no further question. Each leaf node having a category label, and the test pattern is assigned the category of the leaf node reached.

Advantages of Decision Trees

Train fast, Evaluate fast, Compact models, Intelligible if small, Do feature selection, Don’t use all features, Experts understand/accept them, Easy to convert to rules, Can handle missing values.

Disadvantages of Decision Trees

Not good at regression (predicting continuous values), Not good at non-axis parallel Disadvantages of Decision Trees Not good at regression (predicting continuous values), Not good at non-axis parallel splits, Trees for problems with continuous attributes can be large, Large trees are not intelligible, Split ordering often counterintuitive to experts, Not good at learning frommany inputs (e.g., pixels), The tree is very broad and depth. Therefore if we want to search any element in the tree, it is complicated and it takes much time.

Types

Decision tree has three other names: Classification tree analysis is a term used when the predicted outcome is the class to which the data belongs.Regression tree analysis is a term used when the predicted outcome can be considered as a real number (e.g. the price of a house, or a patients length of stay in a hospital). CART analysis is a term used to refer to both of the above procedures.

Classification and Regression Tree

CART is a binary recursive tree [16]. The tree will progressively split the set of training data into smaller and smaller subsets. If all the samples in each subset had the same category label then it would be ideal. In that case, we say that each subset was pure 1 and could terminate that portion of the tree. Usually for each branch we will decide to either stop splitting and accept an imperfect decision (or) to select another property and grow the tree further. The number of splits at a node is related to the property to be tested at each node. The root node splits the full training set. For nonnumerical data, there is a problem for geometrical interpretation of how the query at a node splits the data. But for numerical data,

we can easily visualize the decision boundaries that are produced by decision tree. That decision boundaries are perpendicular to the co-ordinate axes. Therefore, a property query T at each node N makes the data to reach the immediate descendant nodes as pure as possible. Classification tree is built through a process known as binary recursive partitioning. This is an iterative process of splitting the data into partitions, and then splitting it up further on each of the branches.

Finding the Initial Split

The process starts with a training set consisting of pre-classified records. Pre-classified means that the target value, or dependent variable, has a known class or label. The goal is to build a tree that distinguishes among the classes. For simplicity, assume that there are only two target classes and that each split is binary partitioning. The splitting criterion easily generalizes to multiple classes, and any multi-way partitioning can be achieved through repeated binary splits [16]. Every possible split is tried and considered, and the best split is the one which produces the largest decrease in diversity of the classification label within each partition (this is just another way of saying "the increase in homogeneity"). This is repeated for all values, and the winner is chosen as the best splitter for that node. The process is continued at the next node and, in this manner, a full tree Fig. 14.5 is generated for the given training data Table. 14.1.

R_1 / w_1 (black)		R_2 / w_2 (red)	
x_1	x_2	x_1	x_2
0.15	0.83	0.10	0.29
0.09	0.55	0.08	0.15
0.29	0.35	0.23	0.16
0.38	0.70	0.70	0.19
0.52	0.48	0.62	0.47
0.57	0.73	0.91	0.27
0.73	0.75	0.65	0.90
0.47	0.06	0.75	0.36^{*} (0.32^{+})

Pruning the Tree

An alternate approach to stop splitting is pruning. In this method, a tree is gro wnfully until leaf nodes have minimum impurity. After that, all pairs of neighboring leaf nodes are considered for elimination. Any pair whose elimination yields a small increase in impurity is eliminated and the common parent node is declared a leaf. Such merging (or) joining of the two leaf nodes is the inverse of splitting. In pruning, directly use all information in the training set. For small problems the computational cost is low. But it is not possible for larger problem, because of using all information.
Pruning is the process of removing leaves and branches to improve the performance of the decision tree when it moves from the training data (where the classification is known) to real world applications (where the classification is unknown- it is what you are trying to predict). The tree makes the best split at the root node where there are the largest number of records and, hence, a lot of information. Each subsequent split has a smaller and less representative population with which to work. Toward the end, training records at a particular node display patterns that are peculiar only

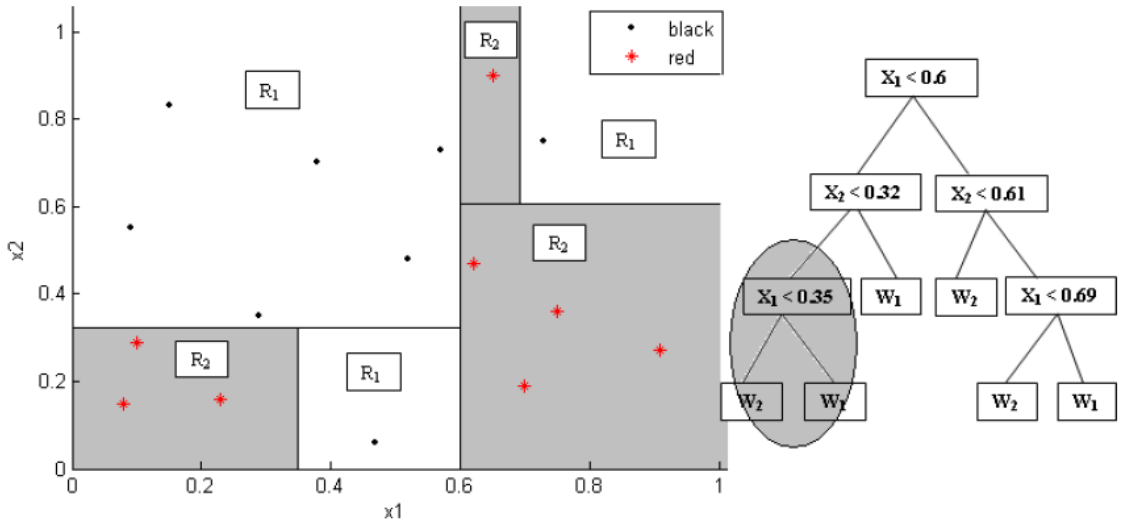


Fig.1. Decision region and unpruned Classification tree

to those records. These patterns can become meaningless and sometimes harmful for prediction if you try to extend rules based on them to larger populations. For example, say the classification tree is trying to predict height and it comes to a node containing one tall person named X and several other shorter people. It can decrease diversity at that node by a new rule saying "people named X are tall" and thus classify the training data.

Pruning methods solve this problem-they let the tree grow to maximum size, then remove smaller branches that fail to generalize. Rather than purity, we calculate the impurity. Let $i(N)$ denote the impurity of a node N. If all the patterns that reach the node bear the same category label then $i(N)$ to be 0. The popular measurement of impurity is entropy impurity.

$$i(N) = -\sum_{j} P\left(w_j\right) \log_2 P\left(w_j\right)$$

-where $P\left(w_j\right)$ is the fraction of patterns at node N that are in category $\boldsymbol{w}_j$.

In Fig.1 we apply number of search of the x_1 positions for the feature and $n-1$ position for the $x2$ feature we find the greatest reduction in the impurity occurs near $x1_s = 0.6$, and hence this becomes the decision criterion at the root node. Then continue for each subtree until each final node represent a single category (has the lowest impurity, 0). Suppose if all the patterns are of the same category then impurity is 0. Otherwise, impurity is positive value. Therefore to choose the query that decreases the impurity as much as possible. The impurity reduction corresponds to an information obtained by the query.

Since the tree is grown from the training data set, when it has reached full structure it usually suffers from over-fitting (i.e. it is "explaining" random elements of the training data that are not likely to be features of the larger population of data). This results in poor performance on real life data. Therefore, it has to be pruned using the validation data set. If pruning were invoked in Fig.

1 the pair of leaf nodes at the left would be the first to be deleted (gray shading) because the impurity is increased the least. If the point marked in the Fig.1 is moved slightly, the decision $*$ in the Fig.1 is moved slightly, the decision region and tree differ significantly, as shown in Fig.2

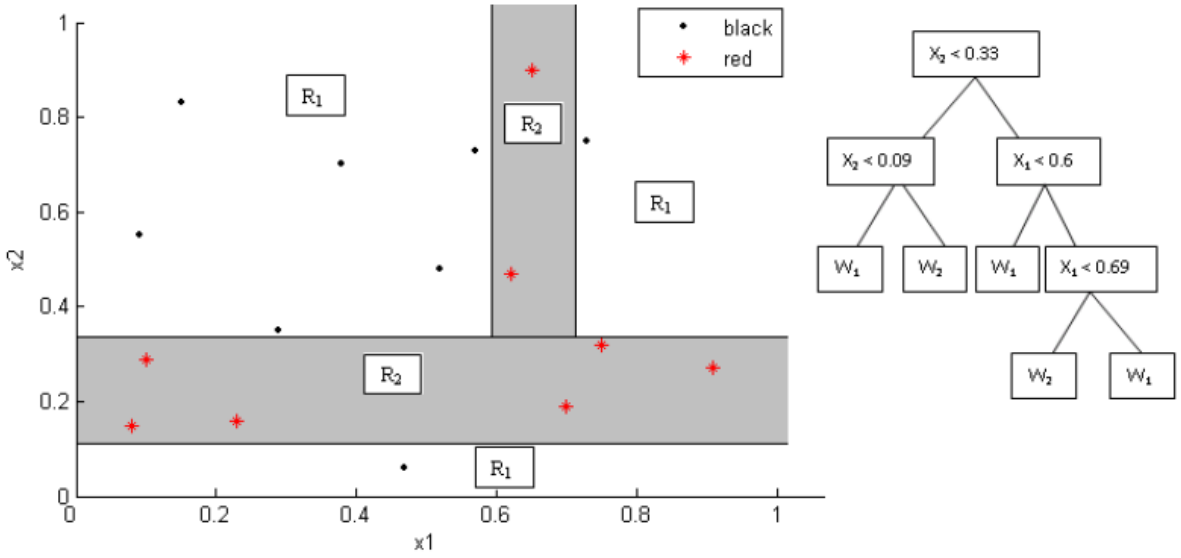


Fig.2.Decision region and classification tree

In Fig.2. we find the greatest reduction in the impurity occurs near $x2_s$= 0.33, and hence this becomes the decision criterion at the root node

Random Forest

Random Forest is a tree-based supervised machine learning algorithm that leverages the power of multiple decision trees for making decisions. As the name suggests, it is a "forest" of trees, it is a forest of randomly created decision trees an ensemble of decision trees, usually trained with the "bagging" method. The general idea of the bagging method is that a combination of learning models increases the overall result. Each node in the decision tree works on a random subset of features to calculate the output and then combines the output of individual decision trees to generate the final output."Tree" and "Forest," a Random Forest is essentially a collection of Decision Trees.
A decision tree is built on an entire dataset, using all the features/variables of interest, whereas a random forest randomly selects observations/rows and specific features/variables to build multiple decision trees from and then averages the results. After a large number of trees are built using this method, each tree "votes" or chooses the class, and the class receiving the most votes by a simple majority is the "winner" or predicted class One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems.

Working of Random Forest Algorithm

The working of Random Forest algorithm with the help of following steps –

- Step 1 – First, start with the selection of random samples from a given dataset.
- Step 2 – Next, this algorithm will construct a decision tree for every sample. Then it will get the prediction result from every decision tree.
- Step 3 – In this step, voting will be performed for every predicted result.
- Step 4 – At last, select the most voted prediction result as the final prediction result. The following diagram will illustrate its working –

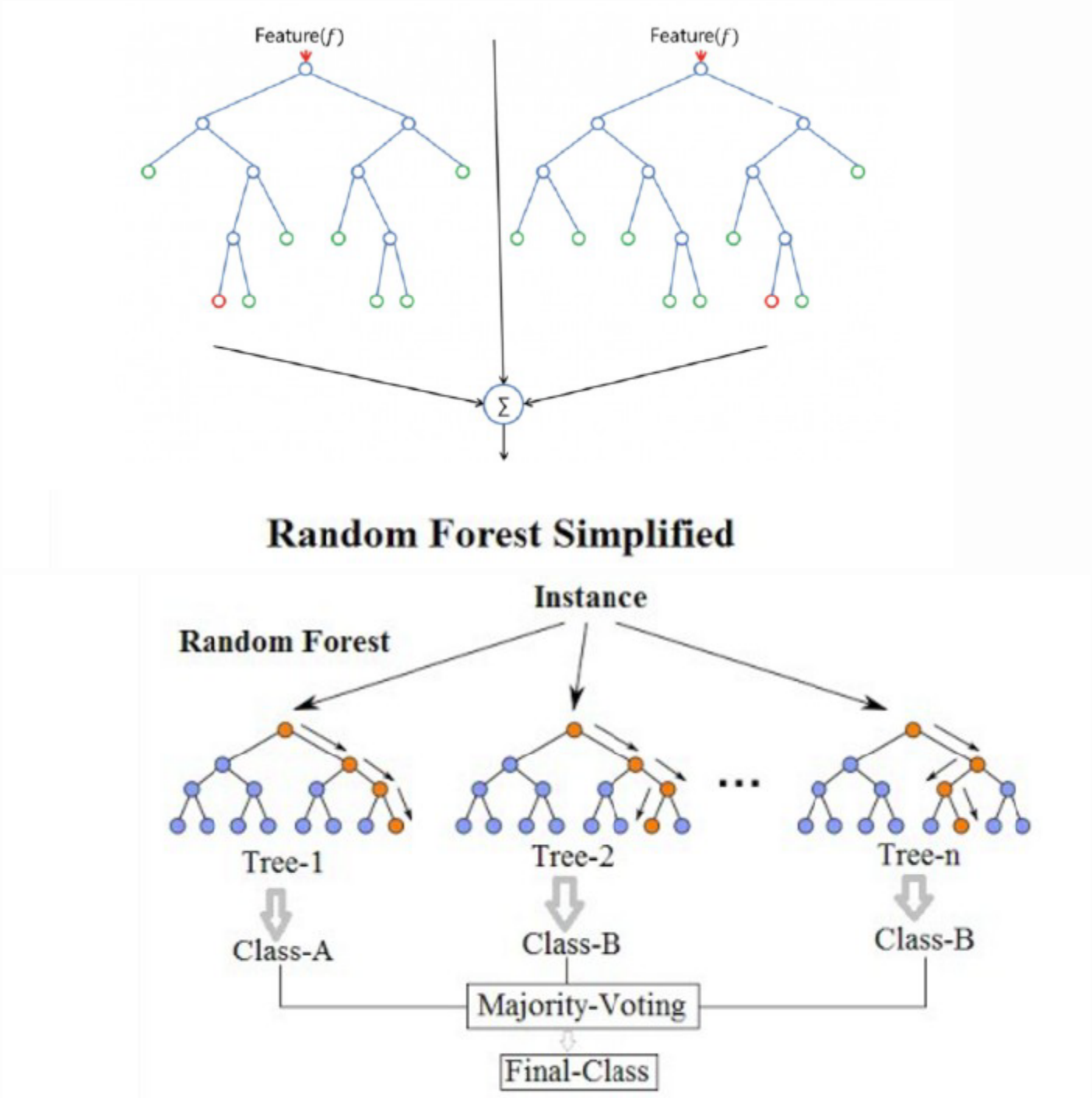


Fig.1. Random Forest

The hyperparameters in random forest are either used to increase the predictive power of the model or to make the model faster. The hyperparameters of sklearns built-in random forest function.

1. Increasing the predictive power

Firstly, there is the **n_estimators** hyperparameter, which is just the number of trees the algorithm builds before taking the maximum voting or taking the averages of predictions. In general, a higher number of trees increases the performance and makes the predictions more stable, but it also slows down the computation. Another important hyperparameter is **max_features**, which is the maximum number of features random forest considers to split a node. Sklearn provides several options, all described in the documentation.

The last important hyperparameter is **min_sample_leaf**. This determines the minimum number of leafs required to split an internal node.

2. Increasing the model's speed

The **n_jobs** hyperparameter tells the engine how many processors it is allowed to use. If it has a value of one, it can only use one processor. A value of "-1" means that there is no limit.
The **random_state** hyperparameter makes the model's output replicable. The model will always produce the same results when it has a definite value of random_state and if it has been given the same hyperparameters and the same training data.
Lastly, there is the **oob_score** (also called oob sampling), which is a random forest cross-validation method. In this sampling, about one-third of the data is not used to train the model and can be used to evaluate its performance. These samples are called the out-of-bag samples. It's very similar to the leave-one-out-cross-validation method, but almost no additional computational burden goes along with it.

Exercise 6 - Decision Tree and Random Forest Classsifiers

a)Decision Tree and Random forest using scikit Learn

Program 1 - Implementing Decision tree and Random forest on Kyphosis Dataset using Scikit Learn

AIM

To implement Decision tree abd random forest classifier in kyphosis dataset using scikit learn.

Part 1 - Loading the dataset

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

```
kyphosis_df = pd.read_csv('./datasets/kyphosis.csv')
display(kyphosis_df.head())
kyphosis_df.info()
```

Output:

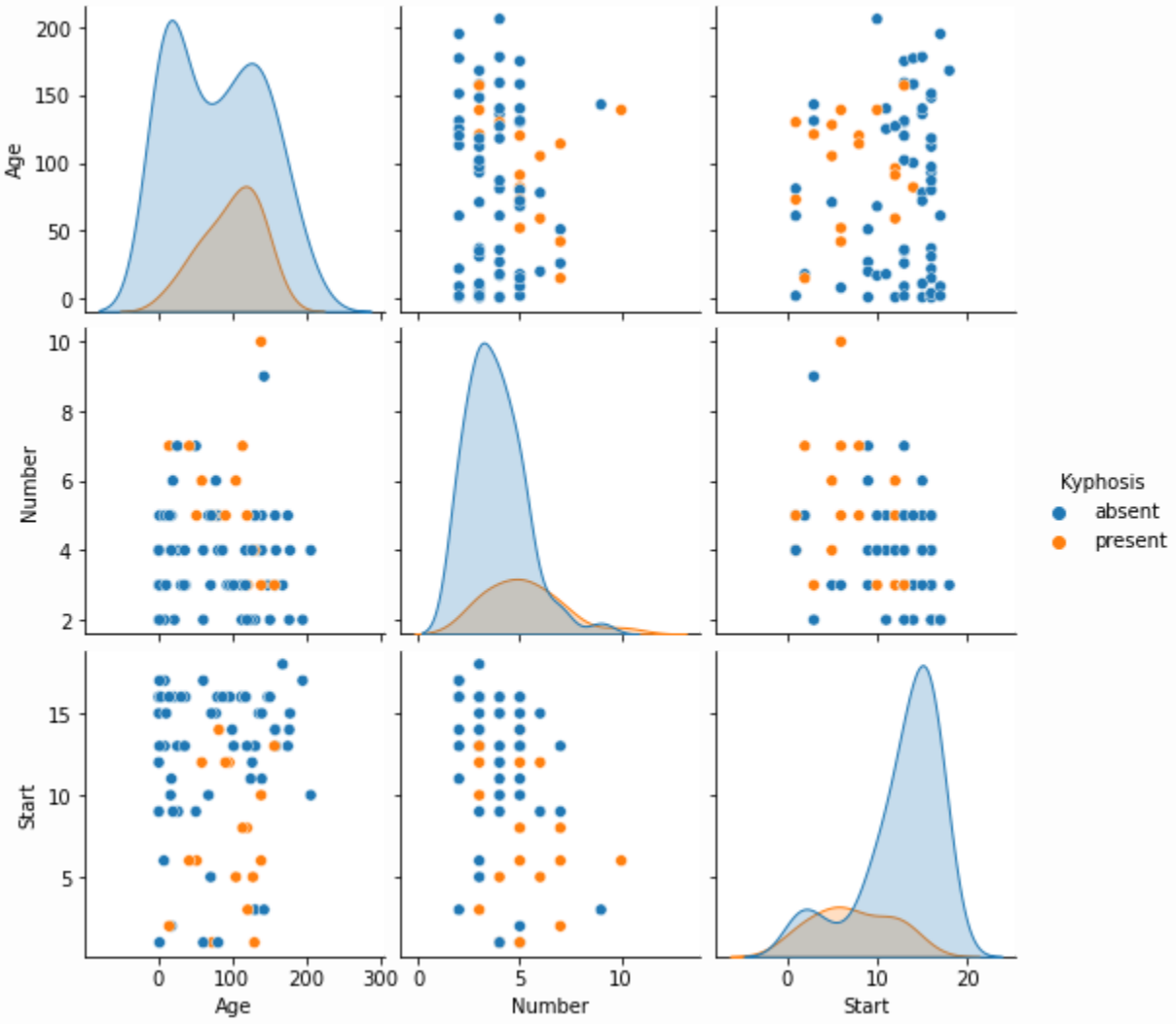
	Kyphosis	Age	Number	Start
0	absent	71	3	5
1	absent	158	3	14
2	present	128	4	5
3	absent	2	5	1
4	absent	1	4	15

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 81 entries, 0 to 80
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Kyphosis  81 non-null     object
 1   Age       81 non-null     int64
 2   Number    81 non-null     int64
 3   Start     81 non-null     int64
dtypes: int64(3), object(1)
memory usage: 2.7+ KB
```

Part 2 - Visualizing the dataset

```
sns.pairplot(kyphosis_df, hue = 'Kyphosis')
plt.show()
```

Output:



Part 3 - Train - Test Split

```
from sklearn.model_selection import train_test_split
X = kyphosis_df.drop('Kyphosis', axis = 1)
y = kyphosis_df['Kyphosis']
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3)
```

Part 4 - Implementing Decision tree classifier using Scikit Learn

```
from sklearn.tree import DecisionTreeClassifier
decision_tree_model = DecisionTreeClassifier()
decision_tree_model.fit(X_train, y_train)
decision_tree_y_pred = decision_tree_model.predict(X_test)
```

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
print("Classification Report for Decision Tree Model using Kyphosis Data:")
print(classification_report(y_test,decision_tree_y_pred))
print("Confusion Matrix:")
print(confusion_matrix(y_test, decision_tree_y_pred))
```

Output:

```
Classification Report for Decision Tree Model using Kyphosis Data:
              precision    recall  f1-score   support

      absent       0.74      0.74      0.74        19
     present       0.17      0.17      0.17         6

    accuracy                           0.60        25
   macro avg       0.45      0.45      0.45        25
weighted avg       0.60      0.60      0.60        25

Confusion Matrix:
[[14  5]
 [ 5  1]]
```

Part 4 - Implementing Random Forest Classifier using Scikit Learn

```
from sklearn.ensemble import RandomForestClassifier
random_forest_model = RandomForestClassifier()
random_forest_model.fit(X_train, y_train)
random_forest_y_pred = random_forest_model.predict(X_test)
```

```
print("Classification Report for Random Forest Model using Kyphosis Data:")
print(classification_report(y_test, random_forest_y_pred))
print("Confusion Matrix:")
print(confusion_matrix(y_test, random_forest_y_pred))
```

Output:

```
Classification Report for Random Forest Model using Kyphosis Data:
              precision    recall  f1-score   support

      absent       0.78      0.95      0.86        19
     present       0.50      0.17      0.25         6

    accuracy                           0.76        25
   macro avg       0.64      0.56      0.55        25
weighted avg       0.71      0.76      0.71        25

Confusion Matrix:
```

```
[[18  1]
 [ 5  1]]
```

7. Convolution Neural Network

The field of machine learning has taken a dramatic twist in recent times, with the rise of the Artificial Neural Network (ANN). These biologically inspired computational models are able to far exceed the performance of previous forms of artificial intelligence in common machine learning tasks. One of the most impressive forms of ANN architecture is that of the Convolutional Neural Network (CNN). CNNs are primarily used to solve difficult image-driven pattern recognition tasks and with their precise yet simple architecture, offers a simplified method of getting started with ANNs. In the domain of Computer Vision, Deep Learning architecture has been constructed and perfected with time, primarily over **Convolutional Neural Network** algorithm

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

The CNN is a combination of two basic building blocks:

1. **The Convolution Block** — Consists of the Convolution Layer and the Pooling Layer. This layer forms the essential component of Feature-Extraction
2. **The Fully Connected Block** — Consists of a fully connected simple neural network architecture. This layer performs the task of Classification based on the input from the convolutional block.

Layers used to build ConvNets

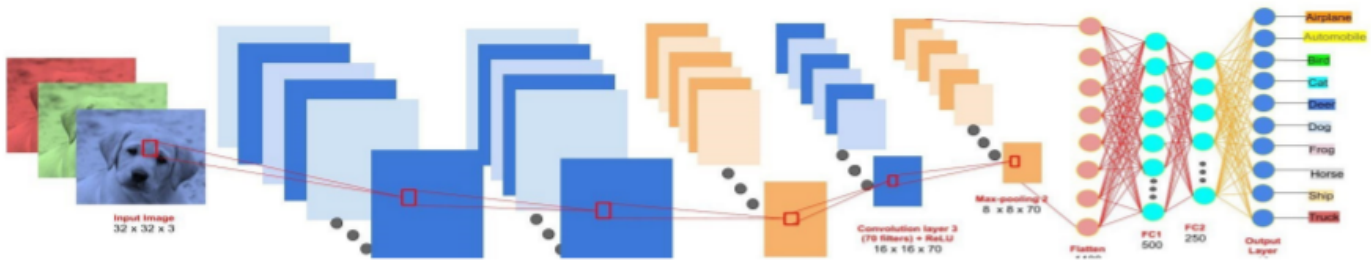
A covnets is a sequence of layers, and every layer transforms one volume to another through differentiable function.

Types of layers:

Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

1. **Input Layer:** This layer holds the raw input of image with width 32, height 32 and depth 3.
2. **Convolution Layer: The convolution layers are the main powerhouse of a CNN model.** This layer computes the output volume by computing dot product between all filters and image patch. Suppose we use total 12 filters for this layer we'll get output volume of dimension 32 x 32 x 12.

CNN- Architecture Deployment for CIFAR-10 Object Classification

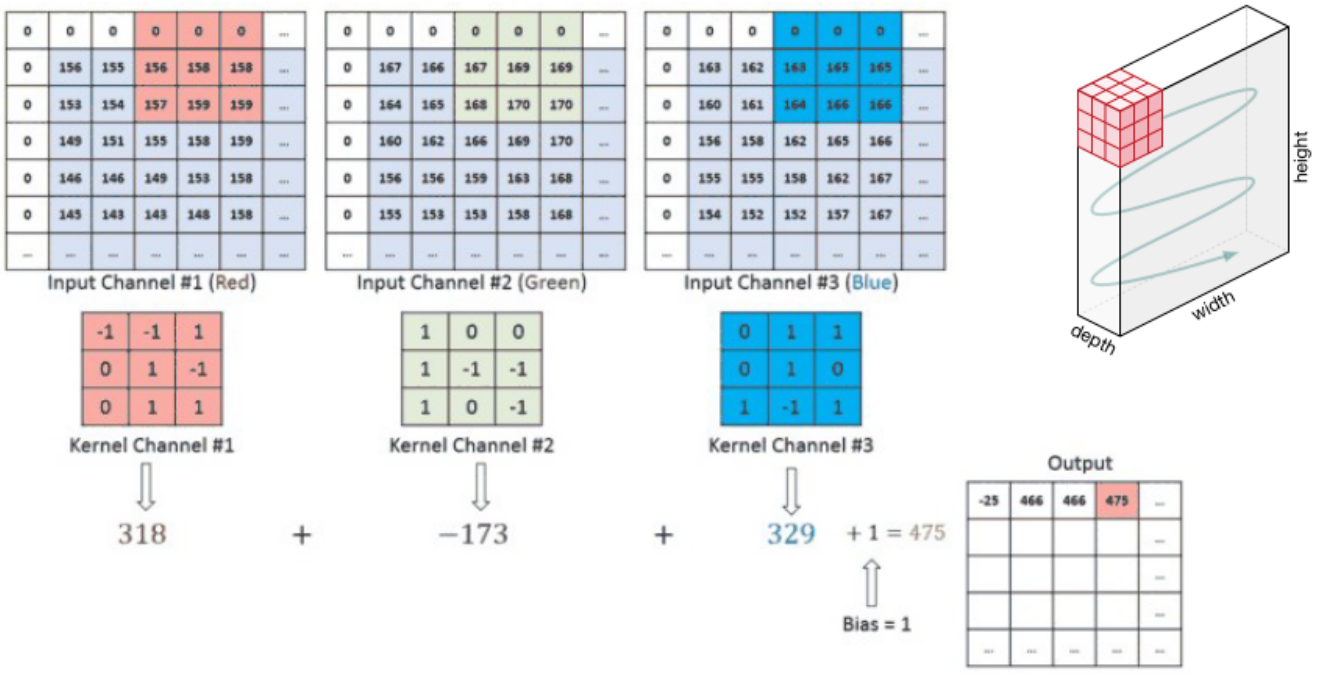


No. of Trainable Parameters/weights:

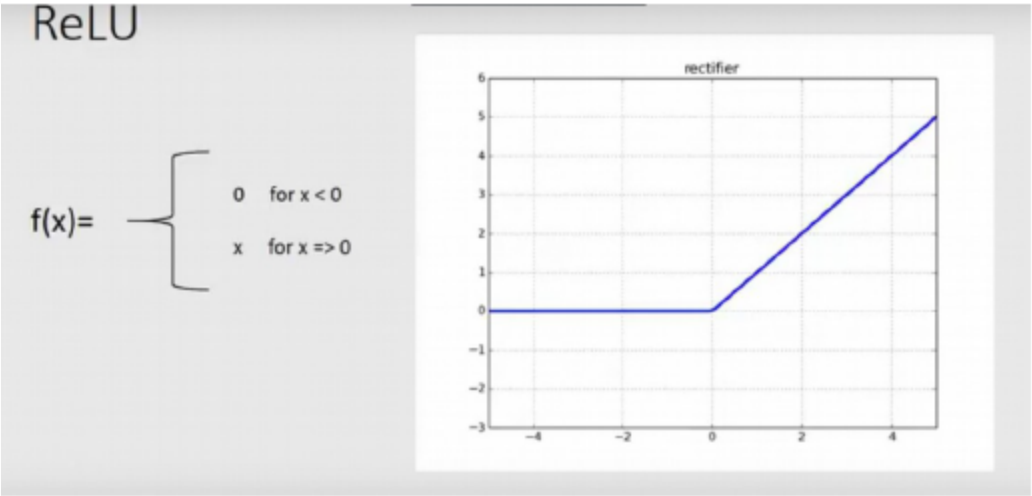
CONV1:	CONV2	CONV3	FC1	FC2	SoftMax
((3 x (3x3)) +1)x25 700	((25 x(3x3) +1) x50 11300	((50x(3x3) +1)x70 31570	(500 x 4480) + 500 2240500	(250 x 500) + 250 125250	(10 x 250) + 10 2510

Convolution Kernel size : 3 x 3
Max Pooling Window size: 2 x 2

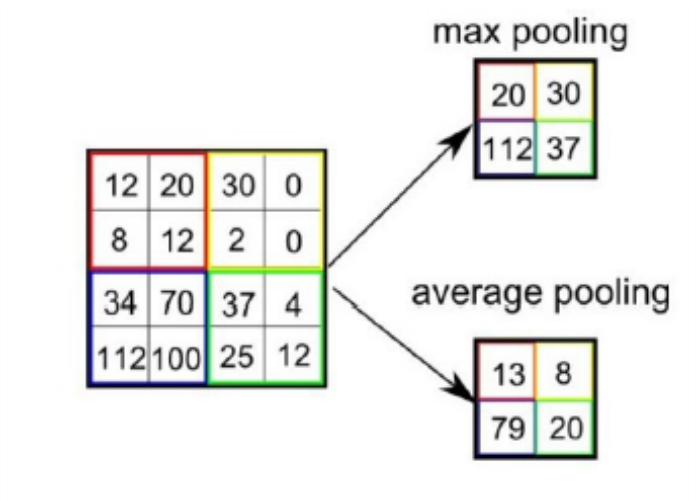
Total Trainable Parameters: 24,11,830



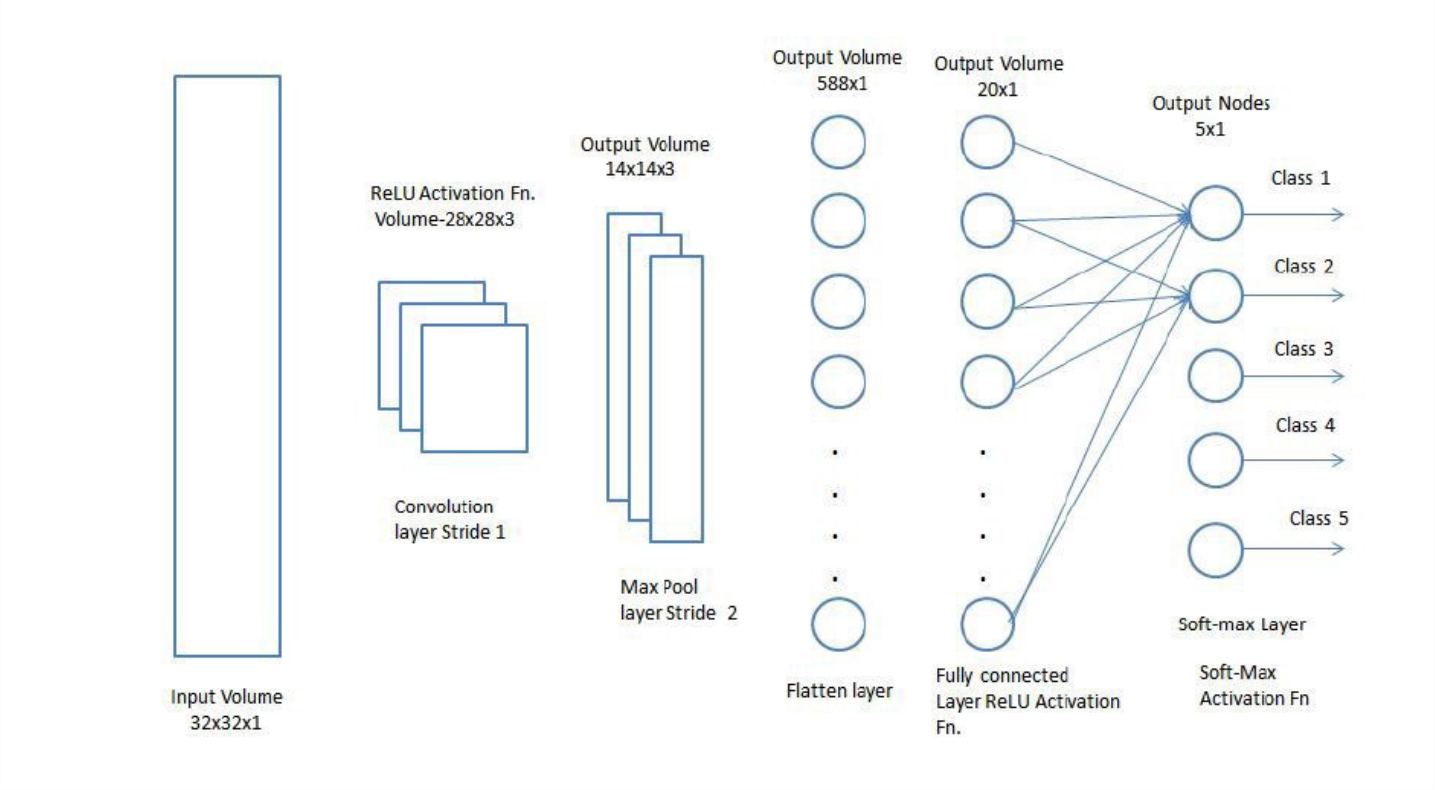
3. **Activation Function Layer:** This layer will apply element wise activation function to the output of convolution layer. Some common activation functions are RELU: $\max(0, x)$, Sigmoid: $1/\left(1 + e^{-x}\right)$, Leaky RELU, etc. The volume remains unchanged hence output volume will have dimension 32 x 32 x 12.



4. **Pool Layer:** This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents from overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.



1. **Fully-Connected Layer:** This layer is regular neural network layer which takes input from the previous layer and computes the class scores and outputs the 1-D array of size equal to the number of classes.



Trainning the CNN

The CNN is trained over a larger no of image during the training phase and for each time, the error generated is fed back into to CNN to adjust the values of the matrices in each layer. The basic working is similar to that of the training of a Simple Neural Network. The concept is **backpropagation**. The mathematical background of backpropagation and the adjustment of values will be studied in future .
backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the **Softmax Classification** technique
There are various architectures of CNNs available which have been key in building algorithms which power and shall power AI as a whole in the foreseeable future. Some of them have been listed below:

1. LeNet
2. AlexNet
3. VGGNet

Deep learning Framework keras

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research.
TensorFlow 2: an approchable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

Building CNN with keras

2D convolution layer (e.g. spatial convolution over images).
This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If **use_bias** is True, a bias vector is created and added to the outputs. Finally, if **activation** is not None, it is applied to the outputs as well.

When using this layer as the first layer in a model, provide the keyword argument input_shape (tuple of integers, does not include the sample axis), **e.g. input_shape=(128, 128, 3) for 128x128 RGB pictures in data_format="channels_last".**

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Key Arguments to discuss

- **filters:** Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size:** An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides:** An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
- **padding:** one of "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
- **activation:** Activation function to use. If you don't specify anything, no activation is applied (see keras.activations).
- **use_bias:** Boolean, whether the layer uses a bias vector.
- **kernel_initializer:** Initializer for the kernel weights matrix.
- **bias_initializer:** Initializer for the bias vector (see keras.initializers).
- **kernel_regularizer:** Regularizer function applied to the kernel weights matrix .

MaxPooling2D class

```
tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2),
    strides=None,
    padding="valid",
    data_format=None,
    **kwargs
)
```

Max pooling operation for 2D spatial data.

Downsamples the input representation by taking the maximum value over the window defined by pool_size for each dimension along the features axis. The window is shifted by strides in each dimension. The resulting output when using “valid” padding option has a shape(number of rows or columns) of: output_shape = (input_shape - pool_size + 1) / strides) The resulting output shape when using the “same” padding option is: output_shape = input_shape / strides

Relu function

```
tf.keras.activations.relu(
    x,
    alpha=0.001,
    max_value=None,
    threshold=0
)
```

Applies the rectified linear unit activation function. With default values, this returns the standard ReLU activation: max(x, 0), the element-wise maximum of 0 and the input tensor.

Softmax function

```
tf.keras.activations.softmax(x, axis=-1)
```

Softmax converts a real vector to a vector of categorical probabilities. The elements of the output vector are in range (0, 1) and sum to 1.

Compile method

Configures the model for training.

```
model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

FIT method

Trains the model for a fixed number of epochs (iterations on a dataset)

```
model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose=1,
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
```

```
sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

Evaluate method

Returns the loss value & metrics values for the model in test mode.

```
model.evaluate(
    x=None,
    y=None,
    batch_size=None,
    verbose=1,
    sample_weight=None,
    steps=None,
    callbacks=None,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
    return_dict=False,
)
```

Predict method

Generates output predictions for the input samples.

```
model.predict(
    x,
    batch_size=None,
    verbose=0,
    steps=None,
    callbacks=None,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

See Documentation for further details

Exercise 7 - Convolutional Neural Networks

a) Image classification using CNN

Program 1 - Classifying handwritten digits from the MNIST dataset using CNN

AIM:

To build a CNN architecture to classify handwritten digits using the MNIST dataset.

ABOUT THE DATASET

MNIST ("Modified National Institute of Standards and Technology") is the de facto "hello world" dataset of computer vision.
The dataset is formatted as csv files in which each row has the flattened pixels of an image and the class label for the image.

Neural Network Architecture

Layer (type)	Output Shape
Conv2D	(None, 26, 26, 64)
Conv2D	(None, 24, 24, 64)
MaxPooling2D	(None, 12, 12, 64)
BatchNormalization	(None, 12, 12, 64)
Conv2D	(None, 10, 10, 128)
Conv2D	(None, 8, 8, 128)
MaxPooling2D	(None, 4, 4, 128)
BatchNormalization	(None, 4, 4, 128)
Conv2D	(None, 2, 2, 256)
MaxPooling2D	(None, 1, 1, 256)
Flatten	(None, 256)
BatchNormalization	(None, 256)
Dense	(None, 512)
Dense	(None, 10)

Part 1 - Importing necesary libraries

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Conv2D, MaxPooling2D,
    Dense, Dropout, Flatten,
    BatchNormalization
)

from keras.utils.np_utils import to_categorical

import joblib
```

Part 2 - Data preparation

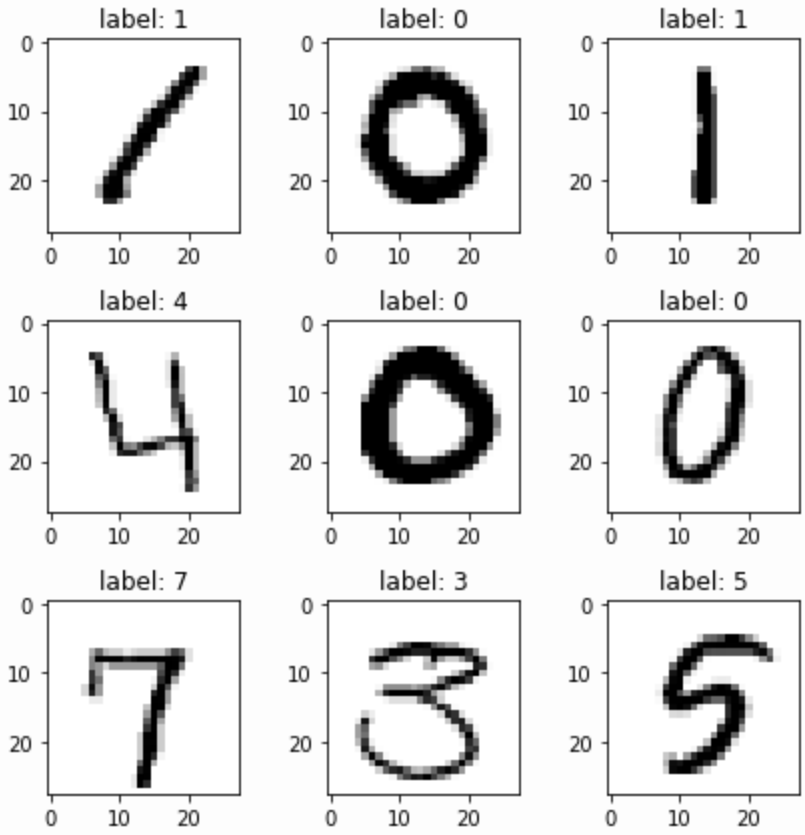
```
train = pd.read_csv('datasets/mnist/train.csv')
test = pd.read_csv('datasets/mnist/test.csv')
```

```
# converting to np arrays
X_train = train.drop(['label'], axis=1).values
y_train = train['label'].values
X_test = test.values

# Reshape image in 3 dimensions (height = 28px, width = 28px , channel = 1)
# channel = 1 => For gray scale
X_train = X_train.reshape(-1,28,28,1)
X_test = X_test.reshape(-1,28,28,1)

# Normalizing to a range of [0..1] from [0..255]
X_train = X_train /255
X_test = X_test/255

# Encode labels to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0])
y_train = to_categorical(y_train)
```

Part 3 - Visualizing samples from dataset

```
fig, axes = plt.subplots(3, 3, figsize=(6, 6))
for i, ax in enumerate(axes.flat):
    ax.imshow(X_train[i].squeeze(), cmap='binary')
    digit = y_train[i].argmax()
    ax.set(title = f"label: {digit}")
fig.tight_layout()
```

Output:



Part 4 - Defining the model

```
def get_model():
    model=Sequential()

    model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu", input_shape=(28,28,1)))
    model.add(Conv2D(filters=64, kernel_size = (3,3), activation="relu"))

    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(BatchNormalization())
    model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))
    model.add(Conv2D(filters=128, kernel_size = (3,3), activation="relu"))

    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(BatchNormalization())
    model.add(Conv2D(filters=256, kernel_size = (3,3), activation="relu"))

    model.add(MaxPooling2D(pool_size=(2,2)))

    model.add(Flatten())
    model.add(BatchNormalization())
    model.add(Dense(512,activation="relu"))

    model.add(Dense(10,activation="softmax"))

    model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
    return model
```

Part 5 - Training the model

```
tf.random.set_seed(0)
model = get_model()
print(model.summary())
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 12, 12, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 128)       73856
```

```
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
batch_normalization_1 (Batch (None, 4, 4, 128)         512
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 2, 2, 256)         295168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 1, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
batch_normalization_2 (Batch (None, 256)               1024
_________________________________________________________________
dense (Dense)                (None, 512)               131584
_________________________________________________________________
```

```
history = model.fit(
    X_train, y_train,
    batch_size = 64,
    epochs = 20,
    validation_split=.2
)
tf.keras.models.save_model(model,"models/mnist_cnn.h5")
joblib.dump(history.history, "models/mnist_cnn.history")

# Use only the below lines if model is not re trained
model = tf.keras.models.load_model("models/mnist_cnn.h5")
history = joblib.load("models/mnist_cnn.history")
```

Output:

```
Epoch 1/20
525/525 [==============================] - 14s 22ms/step - loss: 0.1081 - accuracy: 0.9670 - val_loss: 0.
```

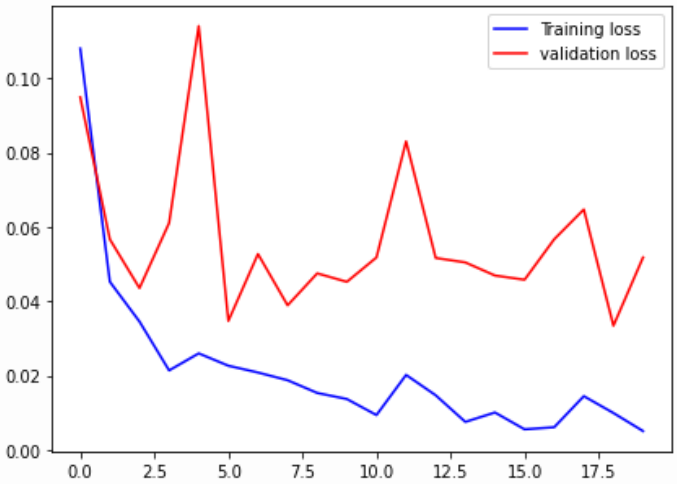
```
950 - val_accuracy: 0.9710
Epoch 2/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0453 - accuracy: 0.9861 - val_loss: 0.
Epoch 3/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0345 - accuracy: 0.9893 - val_loss: 0.
Epoch 4/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0214 - accuracy: 0.9933 - val_loss: 0.
Epoch 5/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0260 - accuracy: 0.9920 - val_loss: 0.
Epoch 6/20
525/525 [==============================] - 11s 20ms/step - loss: 0.0226 - accuracy: 0.9929 - val_loss: 0.
Epoch 7/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0208 - accuracy: 0.9936 - val_loss: 0.
Epoch 8/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0187 - accuracy: 0.9945 - val_loss: 0.
Epoch 9/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0153 - accuracy: 0.9951 - val_loss: 0.
Epoch 10/20
525/525 [==============================] - 11s 20ms/step - loss: 0.0137 - accuracy: 0.9959 - val_loss: 0.
Epoch 11/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0093 - accuracy: 0.9969 - val_loss: 0.
Epoch 12/20
525/525 [==============================] - 11s 20ms/step - loss: 0.0202 - accuracy: 0.9943 - val_loss: 0.
Epoch 13/20
525/525 [==============================] - 11s 20ms/step - loss: 0.0147 - accuracy: 0.9962 - val_loss: 0.
Epoch 14/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0075 - accuracy: 0.9980 - val_loss: 0.
Epoch 15/20
525/525 [==============================] - 11s 21ms/step - loss: 0.0100 - accuracy: 0.9971 - val_loss: 0.
```

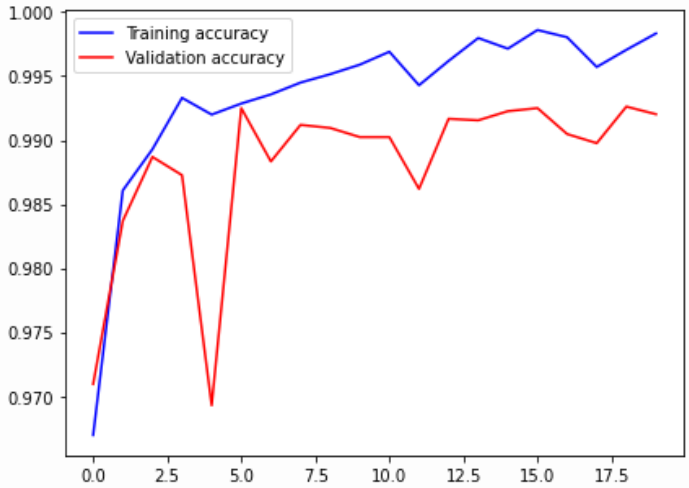
Part 6 - Plot the loss and accuracy curves for training and validation

```
fig, ax = plt.subplots(1,2, figsize=(15,5))
ax[0].plot(history['loss'], color='b', label="Training loss")
ax[0].plot(history['val_loss'], color='r', label="validation loss",axes =ax[0])
ax[0].legend()

ax[1].plot(history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history['val_accuracy'], color='r',label="Validation accuracy")
ax[1].legend()
```

Output:

```
<matplotlib.legend.Legend at 0x7f8a2daea750>
```



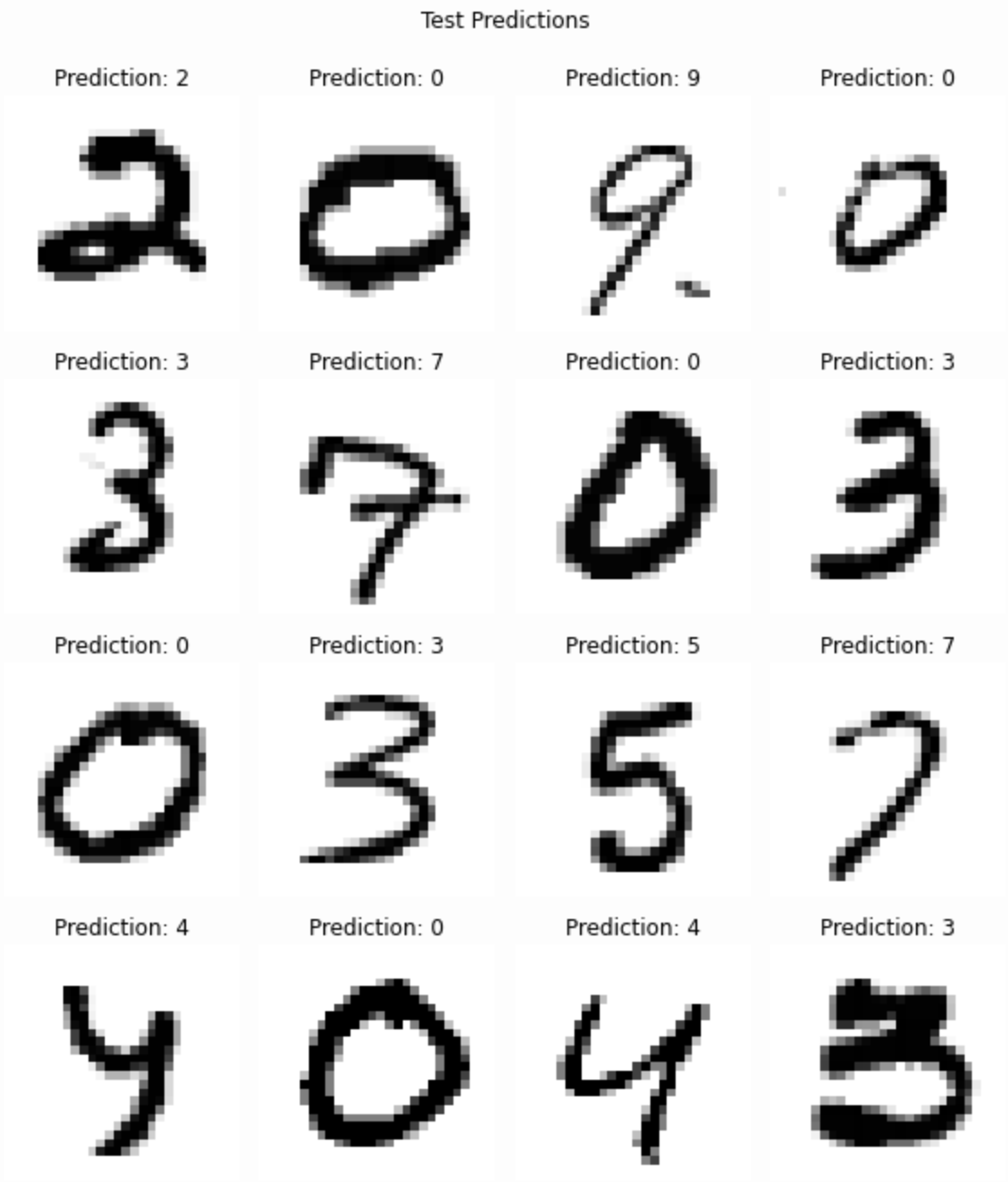


Part 7 - Testing the model with test set

```
y_pred = model.predict(X_test)
```

```
fig, axis = plt.subplots(4, 4, figsize=(8,10))
for i, ax in enumerate(axis.flat):
    ax.imshow(X_test[i].squeeze(), cmap='binary')
    ax.set(title = f"Prediction: {y_pred[i].argmax()}");
    ax.axis("off")
fig.suptitle("Test Predictions")
fig.tight_layout(rect=[0, 0.05, 1, 0.95])
```

Output:


Test Predictions
Prediction: 2
Prediction: 0
Prediction: 9
Prediction: 0
Prediction: 3
Prediction: 7
Prediction: 0
Prediction: 3
Prediction: 0
Prediction: 3
Prediction: 5
Prediction: 7
Prediction: 4
Prediction: 0
Prediction: 4
Prediction: 3

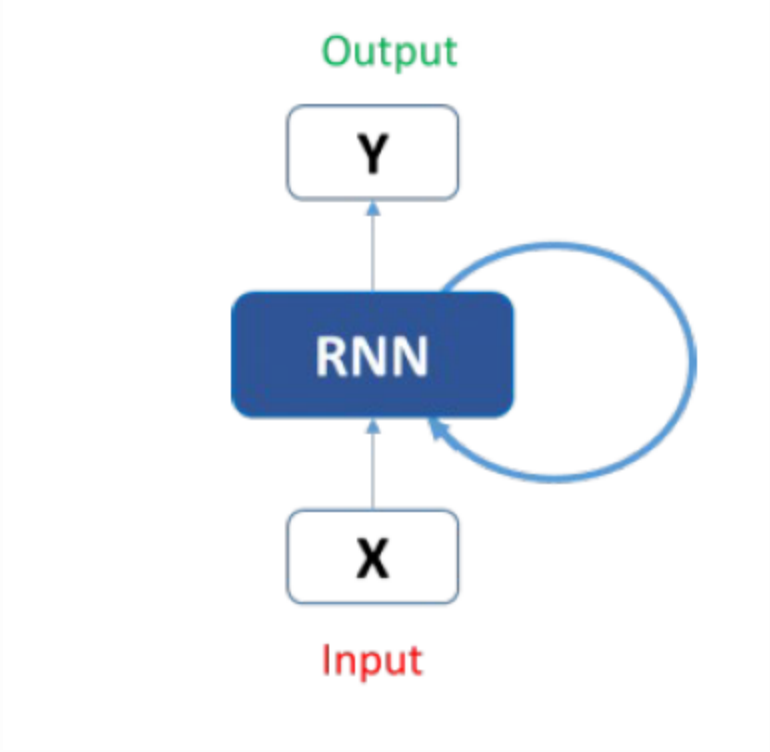
8. Sequence Prediction Using Recurrent Neural Network (RNN)

Recurrent Neural Networks.

A **recurrent neural network (RNN)** is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition structured sequence of data.

The term "recurrent neural network" is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that can not be unrolled.

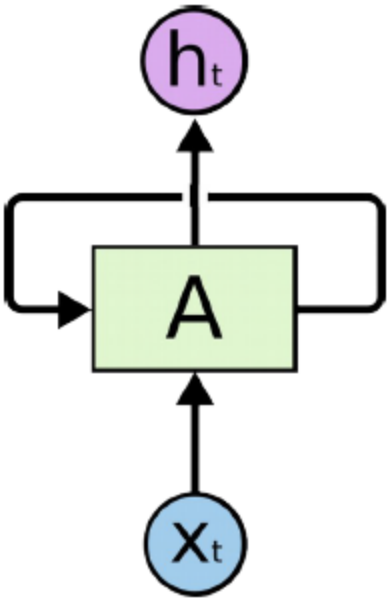
Both finite impulse and infinite impulse recurrent networks can have additional stored states, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units. This is also called Feedback Neural Network (FNN). The concept we can lean on when faced with time sensitive data – Recurrent Neural Networks (RNN),A typical RNN looks like this:



This may seem intimidating at first. But once we unfold it, things start looking a lot simpler:

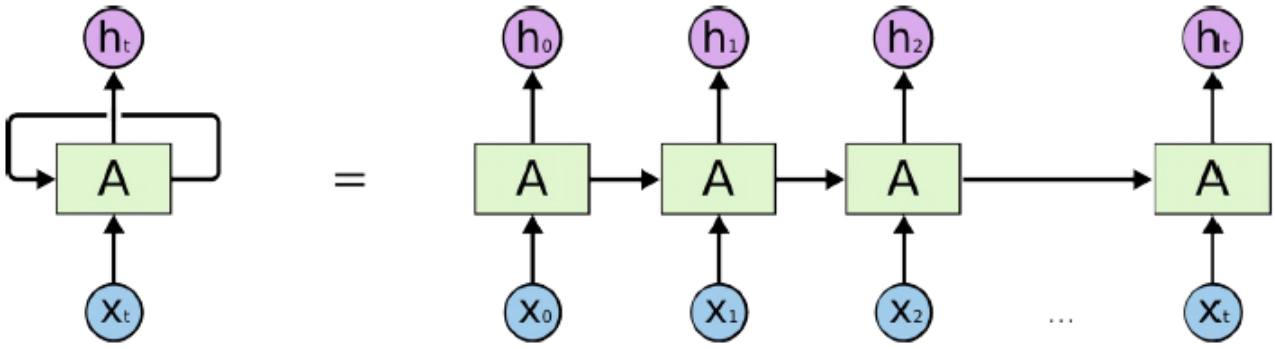
Understanding LSTM Networks

Recurrent Neural Networks with loops in them, allowing information to persist.



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A, looks at some input Xt and outputs a value ht. A loop allows information to be passed from one step of the network to the next. These loops make recurrent neural networks seem kind of mysterious. However, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



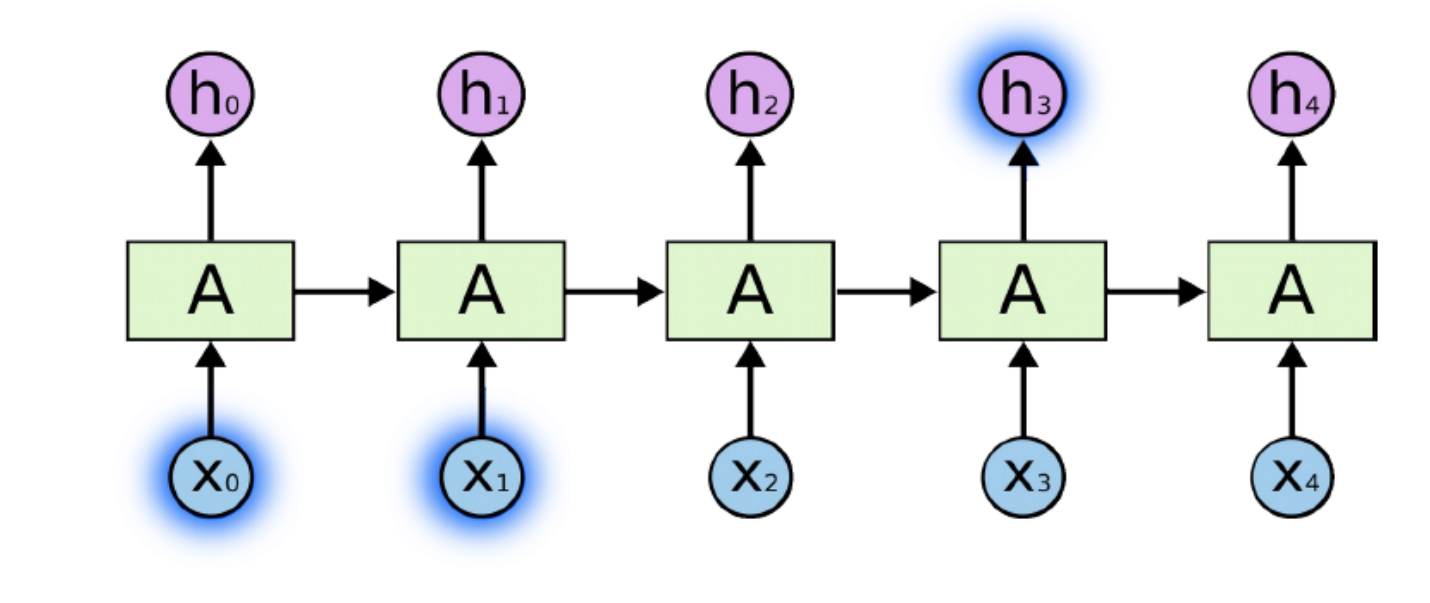
An unrolled recurrent neural network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning etc. Essential to these successes is the use of "LSTMs," a very special kind of recurrent neural network which works, for many tasks, much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It's these LSTMs we will explore.

The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the sky," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France… I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. But in practice, RNNs don't seem to be able to learn them, it might be difficult. Thankfully, LSTMs don't have this problem.

LSTM Networks

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), this network model work tremendously well on a large variety of problems, and are now widely used. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior. All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

The repeating module in a standard RNN contains a single layer.

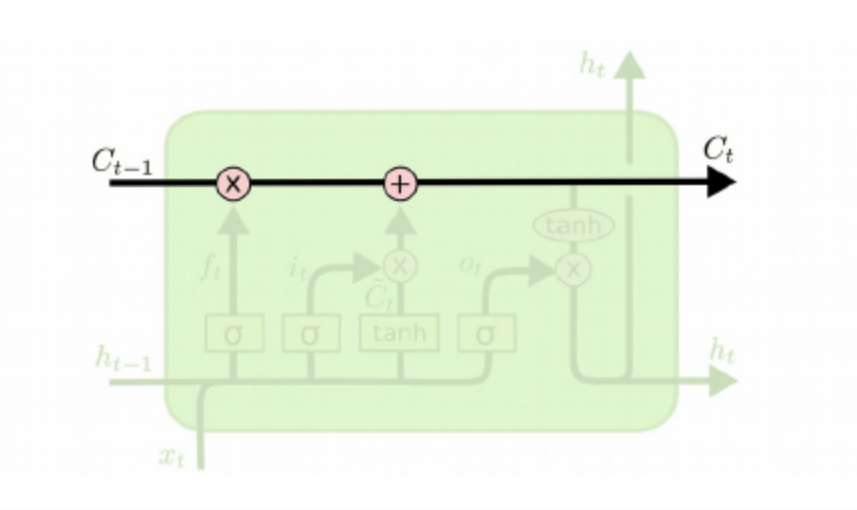
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way. The repeating module in an LSTM contains four interacting layers. LSTM diagram notation used.

In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The Core Idea Behind LSTMs

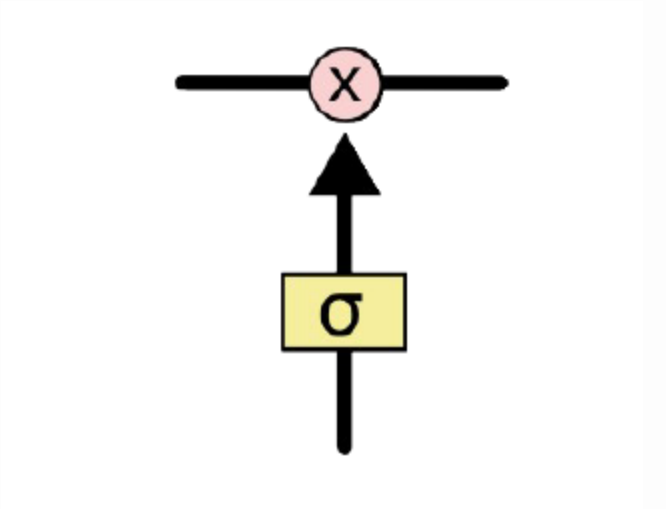
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

Cell state:



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

Gate:

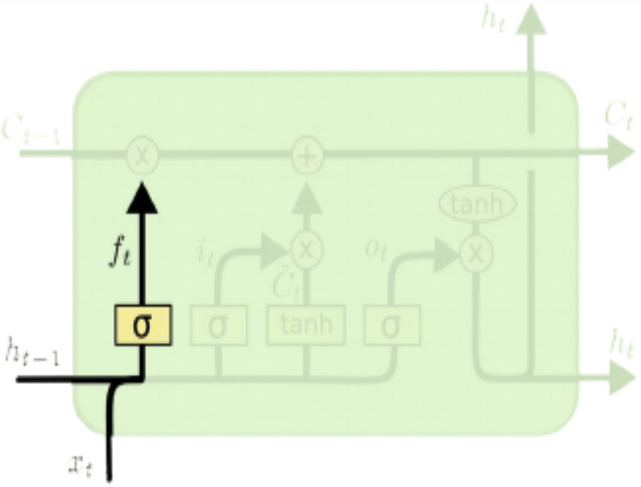


The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through" An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through

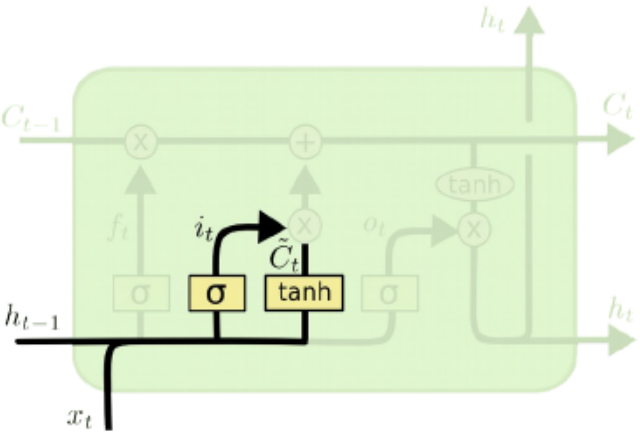
The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the **"forget gate layer."** It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A '1' represents "completely keep this" while a '0' represents "completely get rid of this."

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



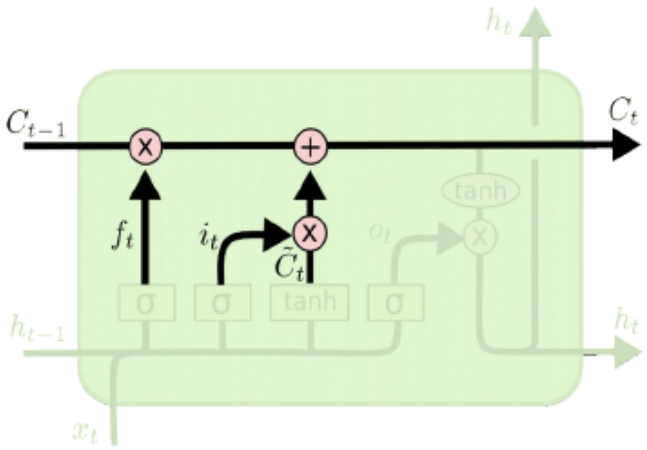
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The **next step** is to decide **what new information we're going to store in the cell state**. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update.
Next, a tanh layer creates a vector of new candidate values, , that could be added to the state. In the next step, we'll combine these two to create an update to the state.
In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

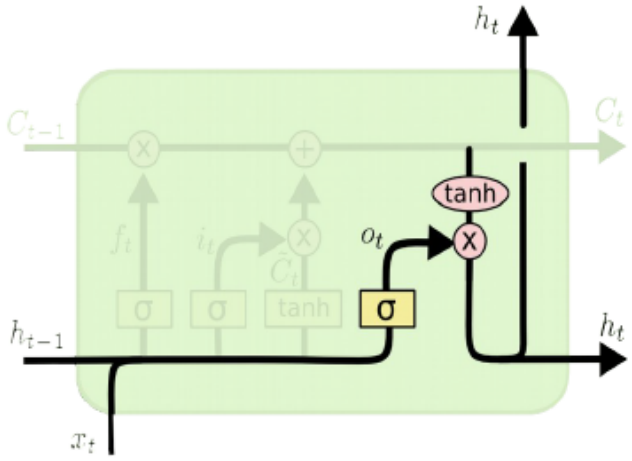
It's **now time to update the old cell state**,C_{t-1} , into the new cell state C_t . The previous steps already decided what to do, we just need to actually do it.
We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * C_t$. This is the new candidate values, scaled by how much we decided to update each state value.
In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through . (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Remarkable results people are achieving with RNNs. Essentially all of these are achieved using LSTMs. They really work a lot better for most tasks LSTMs were a big step in what we can accomplish with RNNs. There is a next step and it's attention!" The idea is to let every step of an RNN pick information to look at from some larger collection of information. Xu, et al. (2015) manuscript as starting point if you want to explore attention! There's been a number of really exciting results using attention, and it seems like a lot more are around the corner

Exercise 8 - Sequence prediction using Recurrent Neural Network

a) Prediction using LSTM

Program 1 - Predicting number sequence using LSTM

AIM:

To predict an arithemetic sequence using LSTM with tensorflow and keras.

Modules used:

Modules	Version
tensorflow	2.6.0
numpy	1.19.5
pandas	1.3.0
matplotlib	3.4.3
scikit-learn	0.24.2

Neural Network Architeture:

Layer (type)	Output Shape
LSTM	(None, 50)
Dense	(None, 1)

Part 1 - Importing modules

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

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import LSTM,Dense
from tensorflow.keras.optimizers import Adam,Adadelta
```

```
from sklearn.model_selection import train_test_split
```

Part 2 - Creating the arithemetic sequence

```
X = np.arange(1,50)
Y = X*15
print("Sequence:",X,Y, sep="\n")
X = X.reshape(-1,1,1)
Y = Y.reshape(-1,1)
X_train,X_test,Y_train,Y_test = train_test_split(X,Y,shuffle=False)
```

Output:

```
Sequence:
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49]
[ 15  30  45  60  75  90 105 120 135 150 165 180 195 210 225 240 255 270
 285 300 315 330 345 360 375 390 405 420 435 450 465 480 495 510 525 540
 555 570 585 600 615 630 645 660 675 690 705 720 735]
```

Part 3 - Constructing the Neural Network archtecture with LSTM

```
def get_model():
    model = Sequential()
    model.add(LSTM(50, activation='relu', input_shape=(1,1)))
    model.add(Dense(1))
    model.compile(optimizer=Adadelta(learning_rate=.03), loss='mse')
    return model
```

```
tf.random.set_seed(0)
model = get_model()
print(model.summary())
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 50)                10400
_________________________________________________________________
dense (Dense)                (None, 1)                 51
=================================================================
Total params: 10,451
Trainable params: 10,451
Non-trainable params: 0
_________________________________________________________________
None
```

Part 4 - Training the model

```
model.fit(X_train, Y_train, epochs=2000, validation_split=0.2, batch_size=10)
tf.keras.models.save_model(model,"./models/seq_pred_model.h5")
model = tf.keras.models.load_model("./models/seq_pred_model.h5")
```

Output:

```
Epoch 1/2000
3/3 [==============================] - 1s 119ms/step - loss: 62007.9180 - val_loss: 239098.1562
```

```
Epoch 2/2000
3/3 [==============================] - 0s 12ms/step - loss: 62005.2695 - val_loss: 239088.6094
Epoch 3/2000
3/3 [==============================] - 0s 12ms/step - loss: 62002.8125 - val_loss: 239079.7344
Epoch 4/2000
3/3 [==============================] - 0s 11ms/step - loss: 62000.4727 - val_loss: 239071.0938
Epoch 5/2000
3/3 [==============================] - 0s 11ms/step - loss: 61998.1875 - val_loss: 239062.3438
Epoch 6/2000
3/3 [==============================] - 0s 12ms/step - loss: 61995.8516 - val_loss: 239053.2500
Epoch 7/2000
3/3 [==============================] - 0s 11ms/step - loss: 61993.4375 - val_loss: 239044.2344
Epoch 8/2000
3/3 [==============================] - 0s 11ms/step - loss: 61991.0664 - val_loss: 239035.3906
Epoch 9/2000
3/3 [==============================] - 0s 12ms/step - loss: 61988.7070 - val_loss: 239026.2969
Epoch 10/2000
3/3 [==============================] - 0s 13ms/step - loss: 61986.2930 - val_loss: 239017.2812
Epoch 11/2000
3/3 [==============================] - 0s 12ms/step - loss: 61983.9297 - val_loss: 239008.4062
Epoch 12/2000
3/3 [==============================] - 0s 11ms/step - loss: 61981.5664 - val_loss: 238999.4219
Epoch 13/2000
3/3 [==============================] - 0s 12ms/step - loss: 61979.2461 - val_loss: 238990.6562
Epoch 14/2000
3/3 [==============================] - 0s 13ms/step - loss: 61976.8203 - val_loss: 238981.5625
Epoch 15/2000
3/3 [==============================] - 0s 12ms/step - loss: 61974.5078 - val_loss: 238972.8750

.
.
.
Epoch 1999/2000
3/3 [==============================] - 0s 16ms/step - loss: 0.3618 - val_loss: 0.0151
Epoch 2000/2000
3/3 [==============================] - 0s 25ms/step - loss: 0.3618 - val_loss: 0.0146
```

Part 5 - Evaluating the model

```
import pandas as pd
```

```
Y_pred = model.predict(X_test, verbose=0)
Y_diff = Y_test - Y_pred
pd.DataFrame(np.hstack([Y_test,Y_pred,Y_diff]),columns=["Actual","Predicted","Difference"])
```

Output:

	Actual	Predicted	Difference
0	555.0	554.936829	0.063171
1	570.0	569.935181	0.064819
2	585.0	584.928589	0.071411
3	600.0	599.916504	0.083496
4	615.0	614.898804	0.101196
5	630.0	629.875366	0.124634
6	645.0	644.846375	0.153625
7	660.0	659.811462	0.188538
8	675.0	674.770752	0.229248
9	690.0	689.724365	0.275635
10	705.0	704.672180	0.327820
11	720.0	719.614685	0.385315
12	735.0	734.551636	0.448364

9. Isolated word Speech Recognition

Introduction:

Speech is a natural mode of communication among human beings. Speech signal caries information about the message to be conveyed, speaker identity and language information. For communication among human beings, there is no need for speech processing, since they are endowed with both speech production and perception mechanisms. But, if a machine is placed in the communication chain, it needs speech processing because it does not have the knowledge of production and perception. All the information required to perform the basic speech processing tasks is implicitly present in the speech. The fundamental issue in speech processing is how to extract specific features to perform the desired speech processing tasks.
In machine learning domain Speech recognition, computer **speech recognition**, or **speech-to-text**, is a capability which enables a program to process human **speech** into a written format. Automatic Speech Recognition (ASR) has historically been a driving force behind many machine learning (ML) techniques, including the ubiquitously used hidden Markov model, discriminative learning, Bayesian learning, and adaptive learning. Moreover, ML can and occasionally does use ASR as a large-scale, realistic application to rigorously test the effectiveness of a given technique, and to inspire new problems arising from the inherently sequential nature of speech. New insight from modern Deep Learning (DL) methodology shows great promise to advance the state-of-the-art in ASR technology.

Conventional Speech Recognition

Speech recognition systems consider that the speech signal is a realization of some message encoded as a sequence of one or more symbols. The essential goal is to "decode"this message and then convert it either into writing or into commands to be processed. Types of Speech Recognition Speech recognition systems can be divided into distinct classes by describing what types of utterances they can recognize. These classes are identified as the following:

Connected Words:

Connected word systems (or more precisely 'connected utterances'), allow apportioned utterances to be 'run-together' with a minimal pause between them.

Continuous Speech:

Continuous speech recognizers let users speak almost naturally, while the computer detects the content. Basically, it is a computer dictation.

Spontaneous Speech:

This can be thought of as speech that is natural sounding and not rehearsed. An ASR system for such a speech deals with a variety of natural speech features i.e. words being run together," ums"," ahs", and even slight stutters.

Isolated Words:

Isolated word recognizers usually need each utterance to have silence on both sides of the sample window. The user speaks individual words or phrases and the recognizer accepts single words or a single utterance at a time.
These systems have "Listen/Not-Listen" states, where they require the speaker to wait (pause) between utterances. The discrete utterance is dealt with in two implicit assumptions. The first assumption is that the speech consists of a signal that is going to be recognized as a complete entity with no explicit knowledge for the phonetic content of the word/phrase. The second assumption is that each spoken word/phrase has an apparently defined beginning and ending point. In this excercise, an isolated word speech recognition system was developed to recognize two spoken words

Digitizing Speech Signals

For processing speech signal, the acoustic variations (pressure variations) are to be represented in digital domain. This can be done as follows: A microphone is used to pickup these acoustic variations (air pressures) and convert them into equivalent analog electrical variations. This analog electrical signal is converted to digital signal by sampling followed by quantization according to nyquist criteria of sampling frequency.

Sampling the signal

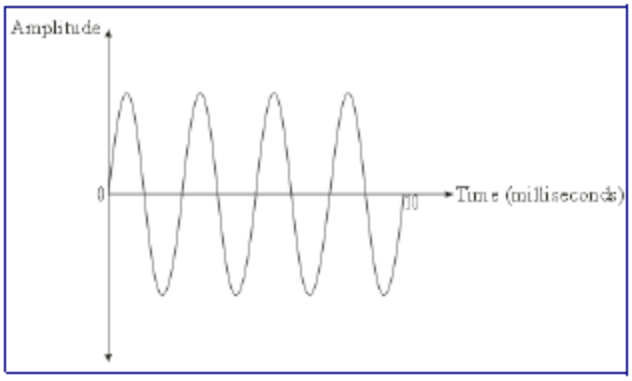
An audio signal is a continuous representation of amplitude as it varies with time. Here, time can even be in picoseconds. That is why an audio signal is an analog signal. Analog signals are memory hogging since they have an infinite number of samples and processing them is highly computationally demanding. Therefore, we need a technique to convert analog signals to digital signals so that we can work with them easily. Sampling the signal is a process of converting an analog signal to a digital signal by selecting a certain number of samples per second from the analog signal. We are converting an audio signal to a discrete signal through sampling so that it can be stored and processed efficiently in memory. able to reconstruct an almost similar audio wave even after sampling the analog signal since we have chosen a high sampling rate.

Feature Extraction Techniques for an Audio Signal

The first step in speech recognition is to extract the features from an audio signal which we will input to our model later. There are different ways of extracting features from the audio signal.

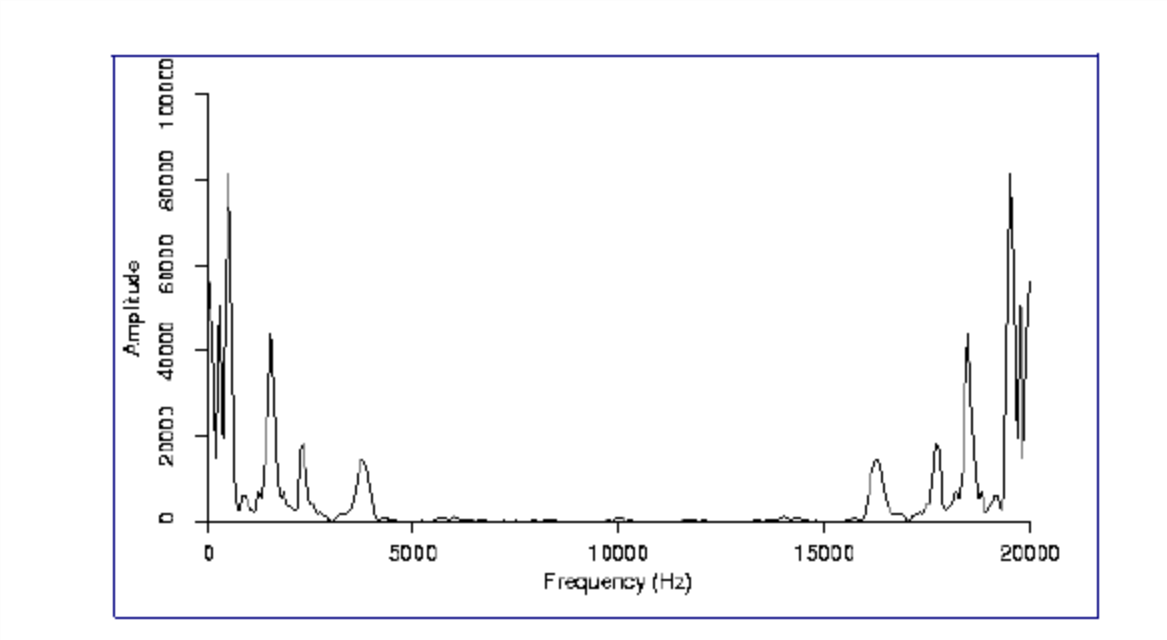
Time-domain

Here, the audio signal is represented by the amplitude as a function of time. In simple words, it is a plot between amplitude and time. The features are the amplitudes which are recorded at different time intervals.
The limitation of the time-domain analysis is that it completely ignores the information about the rate of the signal which is addressed by the frequency domain analysis. So let's discuss that in the next section.



Frequency domain

In the frequency domain, the audio signal is represented by amplitude as a function of frequency. Simply put – it is a plot between frequency and amplitude. The features are the amplitudes recorded at different frequencies.



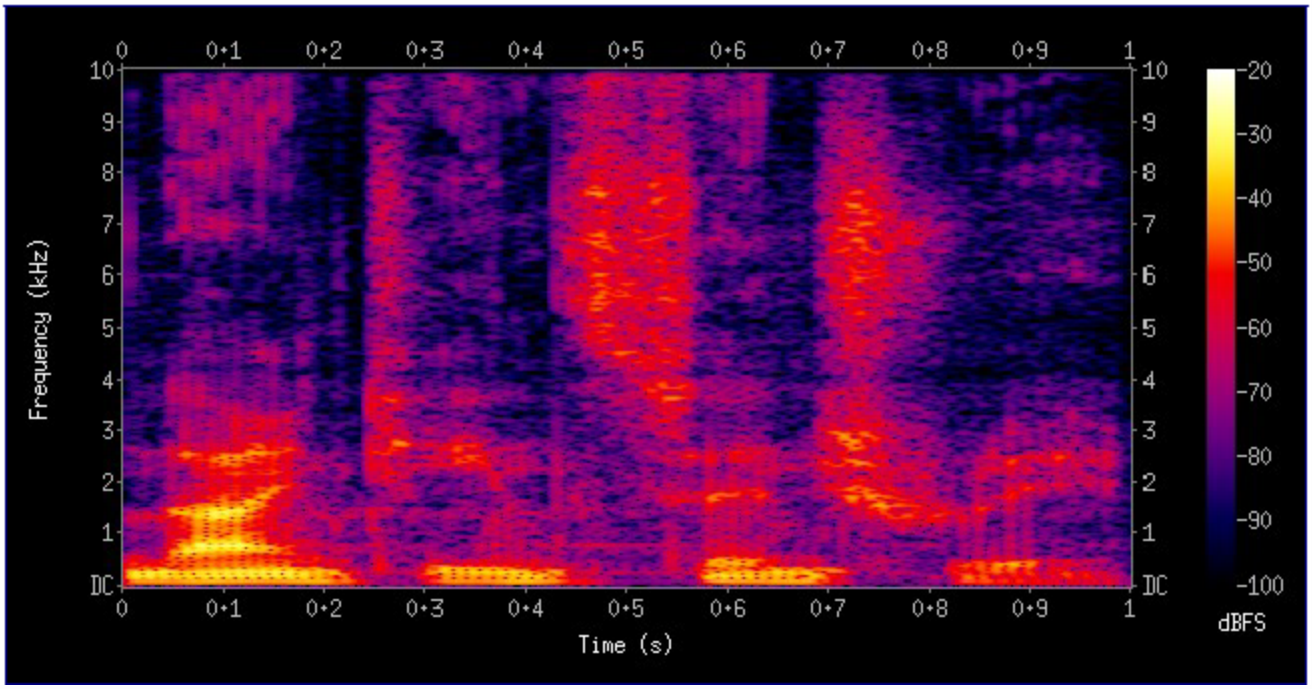
The limitation of this frequency domain analysis is that it completely ignores the order or sequence of the signal which is addressed by time-domain analysis.

Time-domain analysis completely ignores the frequency component whereas frequency domain analysis pays no attention to the time component.

We can get the time-dependent frequencies with the help of a spectrogram.

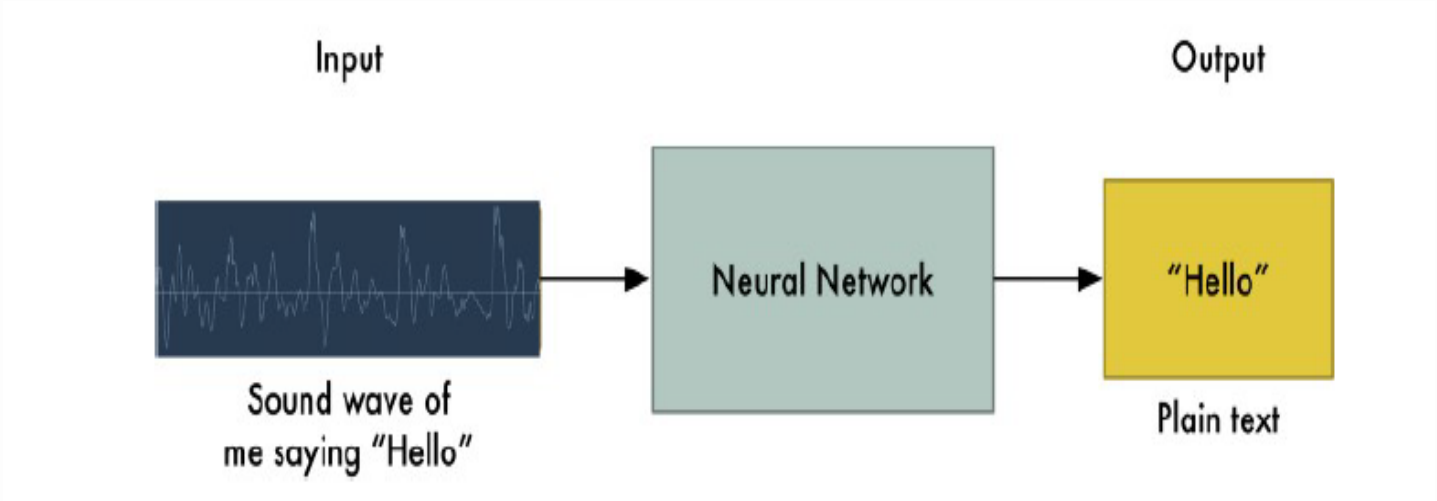
Spectrogram

Spectrogram is a 2D plot between time and frequency where each point in the plot represents the amplitude of a particular frequency at a particular time in terms of intensity of color. In simple terms, the spectrogram is a spectrum (broad range of colors) of frequencies as it varies with time.



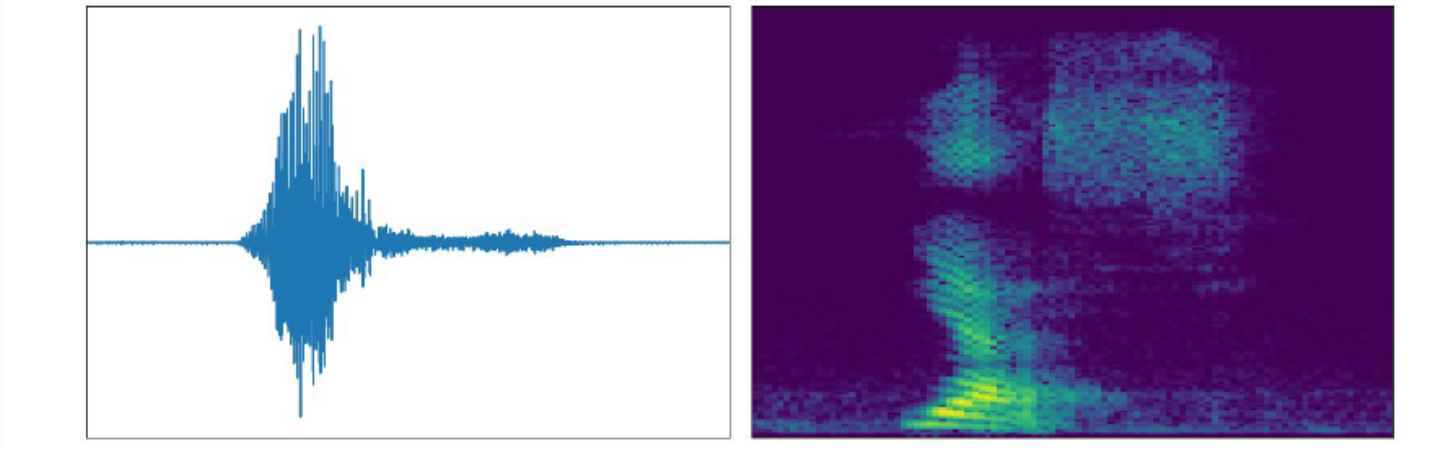
The right features to extract from audio depends on the use case we are working with. However wih the advent of Deep Learning features were learned and extracted automaticaly according to the problem space.

A common strategy for speech recognition is to first extract features from the raw waveform. Commonly used speech features like spectrograms, log-Mel filter banks and Mel-frequency cepstral coefficients (MFCC) convert the raw waveform into a time-frequency domain . These features are then used as an input to a model. Sainath and Paranda (2015) show how log-Mel filter banks can be used as input features to a neural network.
Dai et al. (2017) discuss the challenges with audio-feature-engineering, which requires certain domain knowledge without necessarily building optimal features. Instead of using feature engineering of any kind, this work aims to leverage the power of deep learning to discover features during training from the raw waveform



Single word speech Recognition:

Although single-word speech recognition differs a lot from full scale speech recognition, many of the underlying ideas are the same. Traditional speech recognition systems commonly use Hidden Markov models (HMMs) along with a lot of feature engineering and Gaussian mixture models (GMMs) for acoustic models. The first steps towards using neural networks in speech recognition were using neural networks for acoustic modeling instead of GMMs. (LeCun, Bengio & Hinton 2015) These have since been mostly replaced by end-to-end trained neural architectures such as Deep Speech (Hannun et al. 2014, Amodei et al. 2016).



To build a basic speech recognition network that recognizes ten different words. It's important to know that real speech and audio recognition systems are much more complex, but like MNIST for images, it should give you a basic understanding of the techniques involved. Once you've completed this exercise you'll have a model that tries to classify a one second audio clip as "down", "go", "left", "no", "right", "stop", "up" and "yes".

Implementation Guide:

Reading audio files and their labels

The audio file will initially be read as a binary file, which you'll want to convert into a numerical tensor. To load an audio file, you will use `tf.audio.decode_wav`, which returns the `WAV`encoded audio as a Tensor and the sample rate.
A **WAV** file contains time series data with a set number of samples per second. Each sample represents the amplitude of the audio signal at that specific time. In a 16-bit system, like the files in mini_speech_commands, the values range from -32768 to 32767. The sample rate for this dataset is $16kHz$. tf.audio.decode_wav will normalize the values to the range [-1.0, 1.0].
The label for each WAV file is its parent directory.

Spectrogram:

Convert the waveform into a spectrogram, which shows frequency changes over time and can be represented as a 2D image. This can be done by applying the short-time Fourier transform (**STFT**) to convert the audio into the time-frequency domain.
A Fourier transform (`tf.signal.fft`) converts a signal to its component frequencies, but loses all time information. The **STFT** (`tf.signal.stft`) splits the signal into windows of time and runs a Fourier transform on each window, preserving some time information, and returning a 2D tensor that you can run standard convolutions on.
STFT produces an array of complex numbers representing magnitude and phase. However, we only need the magnitude for this excersice, which can be derived by applying tf.abs on the output of `tf.signal.stft`.
Choose **frame_length** and **frame_step** parameters such that the generated spectrogram "image" is almost square.
You also want the waveforms to have the same length, so that when you convert it to a spectrogram image, the results will have similar dimensions. This can be done by simply zero padding the audio clips that are shorter than one second.
Using the spectrogram input for every class of word we fit the CNN model which has to evaluated using test sample speech word signal data.

Exercise 9 - Isolated Word Speech Recognition

a)Isolated Word speech recognition using CNN

Program 1 - Implementing Isolated word speech recognition in speech commands dataset Using CNN

AIM

To build isolated word speech recognition model using CNN on the spech commands dataset and test it with recorded audio which is not from the dataset

About the Dataset

[18] Speech commands dataset version 2. Available: http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz

Note: Only the words bed, cat and happy are used in this exercise

Modules used:

Modules	Version
tensorflow	2.6.0
numpy	1.19.5
librosa	0.8.1
matplotlib	3.4.3
ipython	7.26.0

Neural Network Architecture

Layer (type)	Output Shape
Conv2D	(None, 11, 29, 32)
Conv2D	(None, 9, 27, 48)
Conv2D	(None, 6, 24, 64)
MaxPooling2D	(None, 1, 6, 64)
Dropout	(None, 1, 6, 64)
Flatten	(None, 384)
Dense	(None, 128)
Dropout	(None, 128)
Dense	(None, 64)
Dropout	(None, 64)
Dense	(None, 3)

Part 1 - Loading the the audio files and extracting mfcc feature from the audio files

```
import numpy as np
import librosa
```

```
data_path = "./datasets/isolated_word_dataset"
labels = np.array(["bed","cat","happy"])
n_classes = labels.shape[0]
n_mfcc = 12 # no. of mfc coefficients
t = 30 # no. of time windows on which the mfc coefficients are computed
input_shape = (-1,n_mfcc,t,1)
```

```
def wav2mfcc(file,t,n_mfcc):
    data, sr = librosa.load(file,sr=None)
    mfcc = librosa.feature.mfcc(data,sr,n_mfcc =n_mfcc)
    if mfcc.shape[1]>t:
        mfcc = mfcc[:,:t]
    if mfcc.shape[1]<t:
        mfcc = np.pad(mfcc,pad_width=((0,0),(0,t- mfcc.shape[1])))
    return mfcc
```

```
def load_speech_dataset_features(data_path,labels):
    X,y = [],[]
    for i,label in enumerate(labels):
        file_name = f"{data_path}/{label}.npy"
        try:
            data = np.load(file_name)
        except(FileNotFoundError):
            data = np.array([
                wav2mfcc(file,t = t,n_mfcc=n_mfcc)
                for file in librosa.util.find_files(f"{data_path}/{label}/")
            ])
            np.save(file_name,data)
        X.append(data)
        y.append(np.full((data.shape[0],1),i))
    X = np.vstack(X).reshape(input_shape)
    y = np.vstack(y)==np.arange(n_classes) # y is one hot encoded
    return X,y
```

```
X,y = load_speech_dataset_features(data_path,labels)
```

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=.2)
X_train,X_test = X_train,X_test
```

Part - CNN Architecture Design

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.optimizers import Adadelta
```

```
def get_model():
    model = Sequential()

    model.add(Conv2D(
        32, kernel_size=(2, 2), activation='relu',
        input_shape=(n_mfcc,t , 1)
    ))
    model.add(Conv2D(48, kernel_size=(3, 3), activation='relu'))
    model.add(Conv2D(64, kernel_size=(4, 4), activation='relu'))
    model.add(MaxPooling2D(pool_size=(4, 4)))
    model.add(Dropout(0.25))

    model.add(Flatten())

    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.25))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(n_classes, activation='softmax'))
    model.compile(
        loss=CategoricalCrossentropy(),
        optimizer=Adadelta(.3),
        metrics=['accuracy']
    )
    return model
```

Part 3 - Training the model

```
tf.random.set_seed(0)
model = get_model()
print(model.summary())
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 11, 29, 32)        160
```

```
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 6, 24, 64)         49216
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 1, 6, 64)          0
_________________________________________________________________
dropout (Dropout)            (None, 1, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 384)               0
_________________________________________________________________
dense (Dense)                (None, 128)               49280
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 195
=================================================================
Total params: 120,979
Trainable params: 120,979
Non-trainable params: 0
_________________________________________________________________
```

```
model.fit(
    X_train, y_train, batch_size=50, epochs=50,
    verbose=True, validation_data=(X_test, y_test)
)
tf.keras.models.save_model(model,"./models/isolated_word_speech_recognition_model.h5")
model = tf.keras.models.load_model("./models/isolated_word_speech_recognition_model.h5")
```

Output:

```
Epoch 1/50
83/83 [==============================] - 10s 112ms/step - loss: 1.6681 - accuracy: 0.4725 - val_loss: 0.6
```

```
29 - val_accuracy: 0.7669
Epoch 2/50
83/83 [==============================] - 9s 108ms/step - loss: 0.7353 - accuracy: 0.6896 - val_loss: 0.44
Epoch 3/50
83/83 [==============================] - 9s 109ms/step - loss: 0.5424 - accuracy: 0.7945 - val_loss: 0.33
Epoch 4/50
83/83 [==============================] - 9s 110ms/step - loss: 0.4257 - accuracy: 0.8388 - val_loss: 0.27
Epoch 5/50
83/83 [==============================] - 9s 110ms/step - loss: 0.3286 - accuracy: 0.8798 - val_loss: 0.22
Epoch 6/50
83/83 [==============================] - 9s 109ms/step - loss: 0.2682 - accuracy: 0.9005 - val_loss: 0.18
Epoch 7/50
83/83 [==============================] - 9s 110ms/step - loss: 0.2198 - accuracy: 0.9169 - val_loss: 0.16
Epoch 8/50
83/83 [==============================] - 9s 110ms/step - loss: 0.1976 - accuracy: 0.9361 - val_loss: 0.17
Epoch 9/50
83/83 [==============================] - 9s 110ms/step - loss: 0.1679 - accuracy: 0.9393 - val_loss: 0.20
Epoch 10/50
83/83 [==============================] - 9s 110ms/step - loss: 0.1489 - accuracy: 0.9455 - val_loss: 0.13
Epoch 11/50
83/83 [==============================] - 9s 110ms/step - loss: 0.1396 - accuracy: 0.9533 - val_loss: 0.10
Epoch 12/50
83/83 [==============================] - 9s 110ms/step - loss: 0.1294 - accuracy: 0.9525 - val_loss: 0.20
Epoch 13/50
83/83 [==============================] - 9s 110ms/step - loss: 0.1140 - accuracy: 0.9593 - val_loss: 0.10
Epoch 14/50
83/83 [==============================] - 9s 109ms/step - loss: 0.1022 - accuracy: 0.9670 - val_loss: 0.09
Epoch 15/50
83/83 [==============================] - 9s 110ms/step - loss: 0.0835 - accuracy: 0.9694 - val_loss: 0.13

.
.
.
Epoch 49/50
83/83 [==============================] - 9s 110ms/step - loss: 0.0224 - accuracy: 0.9947 - val_loss: 0.05
Epoch 50/50
83/83 [==============================] - 9s 113ms/step - loss: 0.0203 - accuracy: 0.9925 - val_loss: 0.08
```

Part 4 - Testing with real recorded audio speech

```
from IPython.display import Audio
```

Output:

```
File Nme: bed.wav
```

Your browser does not support the audio element.

```
Predicted Output : bed
---------------------------------------
File Nme: cat.wav
```

Your browser does not support the audio element.

```
Predicted Output : cat
---------------------------------------
File Nme: happy.wav
```

Your browser does not support the audio element.

```
Predicted Output : happy
---------------------------------------
```

10. Face Detection and Tracking

Introduction:

Image or Object Detection is a computer technology that processes the image and detects objects in it.Its a generalization of object localization process. Also It has applications in many areas of computer vision, including image retrieval and video surveillance

Face detection and tracking:

Face detection only (not recognition) - The goal is to distinguish faces from non-faces (detection is the first step in the recognition process) in an video image frame. Face recognition system identify the person and face verification system verify the person who is claimed to be (by matching it in certain database). Its applications are found in various fields like school, colleges, organisations, factories, public places, surveillance etc. These techniques are gaining momentum worldwide and extended with human emotion recognition, its applications are huge.
Object Detection using Haar feature-based cascade classifiers is an effective object detection method most commonly adopted for simple applications and academic demonstration proposed by Paul Viola and Michael Jones in their paper, “Rapid Object Detection using a Boosted Cascade of Simple Features” in 2001. The key aspect in face recognition is detecting relevant features in human face like eyes, eyebrows, nose, lips for which Haar wavelets of haar features were used through the detection algorithm called as Viola-Jones Algorithm.
Haar features are sequence of rescaled square shape functions proposed by Alfred Haar in 1909. We will apply these haar features to all relevant parts of face so as to detect human face. To detect eyebrow, 2 rectangle Haar feature shown in the fig: is used because forehead and eyebrow form lighter pixels - darker pixel like image. Similarly, to detect lips we use similar to Haar like feature (3 rectangle feature fig.) with lighter-darker-lighter pixels. To detect nose, we might use darker-lighter Harr like feature from (image(1)). And so on. So for this feature extration Viola– Jones requires full view frontal upright faces. Thus in order to be detected, the entire face must point towards the camera and should not be tilted to either side.
The algorithm has four stages: 1. Haar Feature Selection 2. Creating an Integral Image 3. Adaboost Training 4. Cascading Classifiers
There are three key contributions of this algorithm. The first is the introduction of a new image representation called the “Integral Image” which allows the features used by our detector to be computed very quickly. The second is a learning algorithm, based on AdaBoost, which selects a small number of critical visual features and yields extremely efficient classifiers. The third contribution is a method for combining classifiers in a “cascade” which allows background regions of the image to be quickly discarded while spending more computation on promising object-like regions.

Haar Feature selection:

Each feature is a single value obtained by subtracting sum of pixels under white rectangle from sum of pixels under black rectangle.

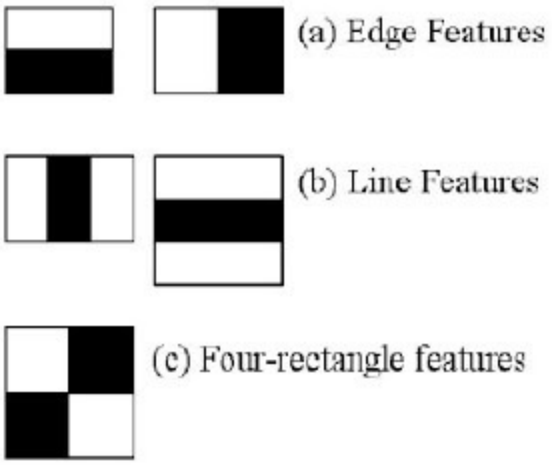
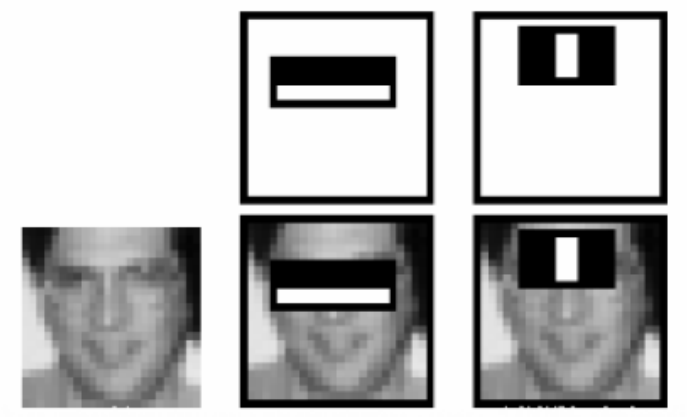
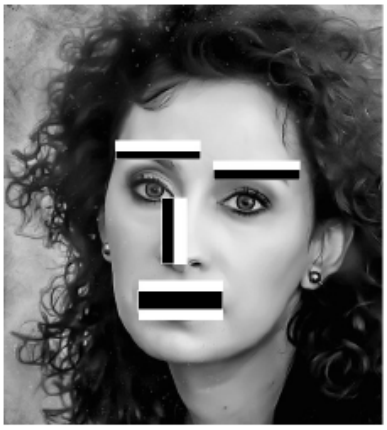


Fig.1. Haar Feature

Haar Feature Matching:





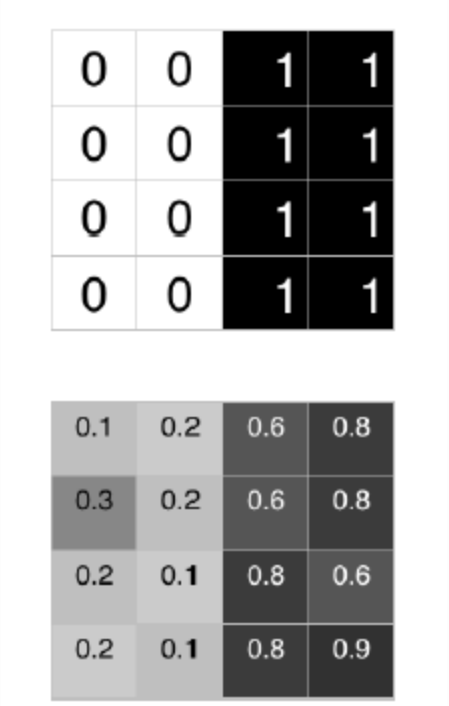
$\Delta = dark - white = \frac{1}{n}\Sigma_{dark}^{n} I(x) - \frac{1}{n}\Sigma_{white}^{n} I(x)$

for Instance:

Ideal case : Delta = (1/8)*(8)* — *(1/8)*0 = 1

Real case: Delta = (1/8)*(5.9)* — *(1/8)*(1.3)=0.575

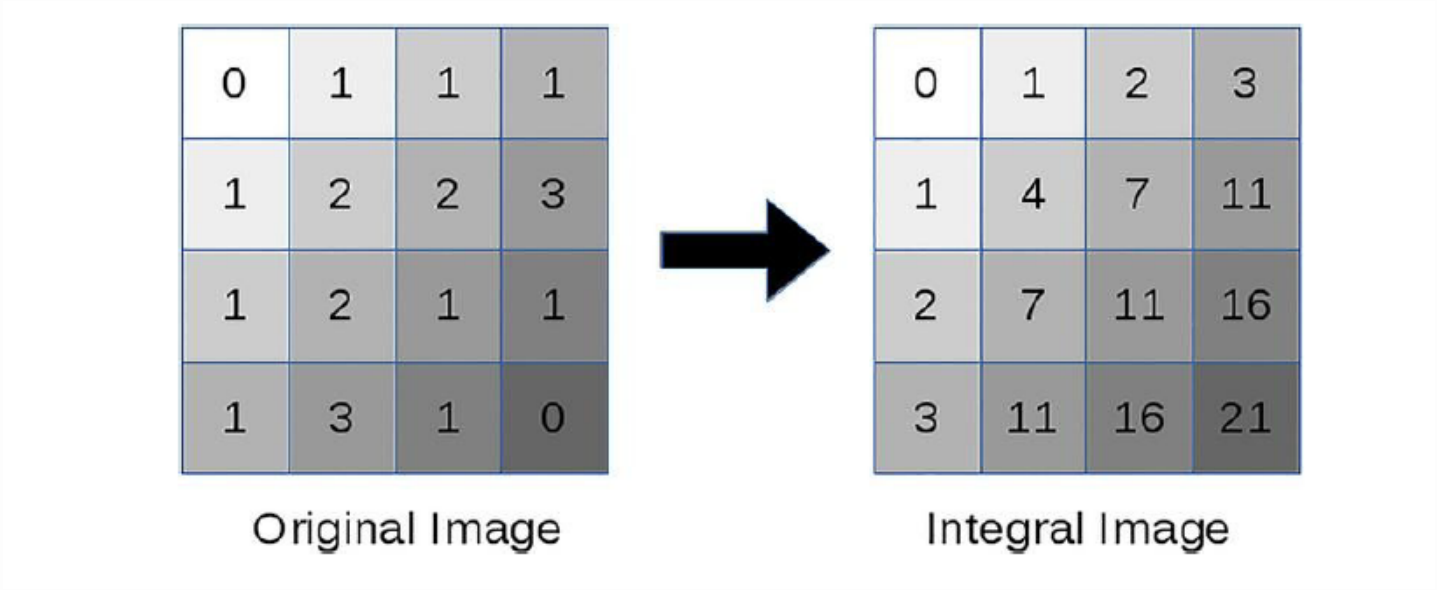
(For greyscale image, assume we have set White-Dark threshold to 0.3. Meaning pixels with value less than or equal to



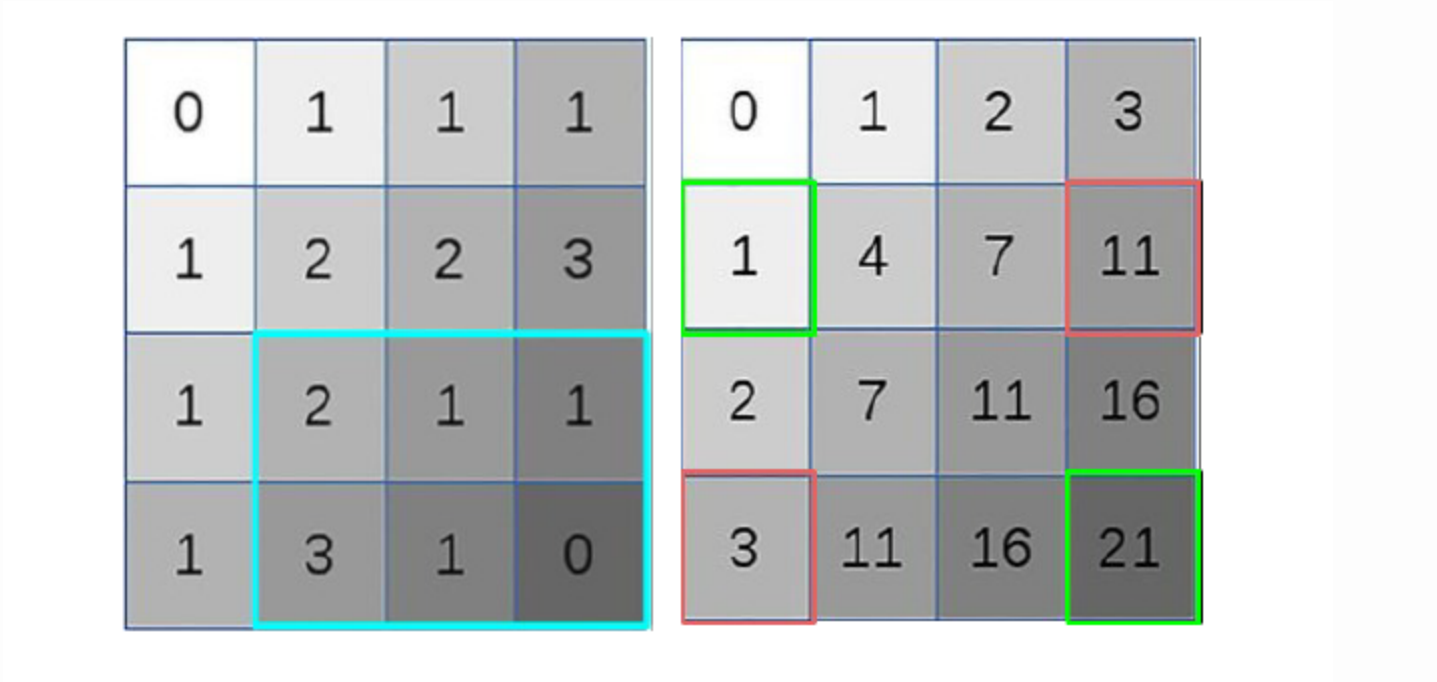
0.3 are considered white and anything greater that 0.3 is considered as dark)
According to Viola-Jonas algorithm, to detect Haar like feature present in an image, above formula should give result closer to 1. The closer the value is to 1, the greater the chance of detecting Haar feature in image.
Now all possible sizes and locations of each kernel is used to calculate plenty of features. (Just imagine how much computation it needs? Even a 24x24 window results over 160000 features). For each feature calculation, we need to find sum of pixels under white and black rectangles. To solve this, they introduced the integral images. It simplifies calculation of sum of pixels, how large may be the number of pixels, to an operation involving just four pixels. It makes things computation faster.

Integral Images

To calculate a value for each feature, we need to perform computations on all the pixels inside that particular feature. In reality, these calculations can be very intensive since the number of pixels would be much greater when we are dealing with a large feature.
The integral image plays its part in allowing us to perform these intensive calculations quickly so we can understand whether a feature of several features fit the criteria.
An integral image (also known as a summed-area table) is the name of both a data structure and an algorithm used to obtain this data structure. It is used as a quick and efficient way to calculate the sum of pixel values in an image or rectangular part of an image.
In an integral image, the value of each point is the sum of all pixels above and to the left, including the target pixel:



Using these integral images, we save a lot of time calculating the summation of all the pixels in a rectangle as we only have to perform calculations on four edges of the rectangle. See the example below to understand.
When we add the pixels in the blue box, we get 8 as the sum of all pixels and here we had six elements involved in your calculation. Now to calculate the sum of these same pixels using the integral image, you just need to find the corners of the rectangle and then add the vertices which are green and subtract the vertices in the red boxes. Now doing that here 21+1 - 11 -3 =8
We get the same answer and only four numbers are involved in calculations. No matter how many pixels are in the rectangle box, we will just need to compute on these 4 vertices. To calculate the value of any haar-like feature, we have a simple way to calculate the difference between the sums of pixel values of two rectangles.



But among all these features we have calculated most of them are irrelevant. For example, consider the image below. Top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. But the same windows applying on cheeks or any other place is irrelevant. So for selecting the best features out of 160000+ features **Adaboost** is deployed.

For this, we apply each and every feature on all the training images. For each feature, it finds the best threshold which will classify the faces to positive and negative, But obviously, there will be errors or misclassifications. We select the features with minimum error rate, which means they are the features that best classifies the face and non-face images. (The process is not as simple as this.

Each image is given an equal weight in the beginning. After each classification, weights of misclassified images are increased. Then again same process is done, New error rates are calculated. Also new weights. The process is continued until required accuracy or error rate is achieved or required number of features are found).

Final classifier is a weighted sum of these weak classifiers. It is called weak because it alone can't classify the image, but together with others forms a strong classifier. The viola jones research manuscript says even 200 features provide detection with 95% accuracy. Their final setup had around 6000 features. (a reduction from 160000+ features to 6000 features).

So in an image take each 24x24 window. Apply 6000 features to it to Check if it is face or not. But it's a little inefficient and time consuming so domain experts (viola-jones) comeup with a solution in which, a simple method to check if a window is not a face region. If it is not, discard it in a single shot. Don't process it again. Instead focus on region where there can be a face. This way, we can find more time to check a possible face region. Since most of the image region is non-face region. So it is a better idea to discard these region without processing.

"Instead of applying all the 6000 features on a window, group the features into different stages of classifiers and apply one-by-one. (Normally first few stages will contain very less number of features). If a window fails the first stage, discard it. We don't consider remaining features on it. If it passes, apply the second stage of features and continue the process. The window which passes all stages is a face region." - Face Detection using Haar Cascades.

Fig.1. Cascaded Classifiers

It is a machine learning based approach in which a cascade function is trained from a lot of positive and negative images, then it is used to detect objects in other images, this concept is know as **Cascade of Classifiers.**

They are series of classifiers or features (as we have seen above) used to identify object in an image. Using sliding windows and number of haar features (increases as number of stages increase), finally leading to detect face or not. There are total 38 stages defined for Viola Jonas Method, Depending upon the sliding windows size and face location, number of features, face can be detected at a certain stage.

Instead of applying all the 6000 features on a window, group the features into different stages of classifiers and apply one-by-one. (Normally first few stages will contain very less number of features). If a window fails the first stage, discard it. We don't consider remaining features on it. If it passes, apply the second stage of features and continue the process. The window which passes all stages is a face region.

Viola jone's detector had 6000+ features with 38 stages with 1, 10, 25, 25 and 50 features in first five stages. (Two features in the above image is actually obtained as the best two features from Adaboost). According to authors, on an average, 10 features out of 6000+ are evaluated per subwindow.

Haar-cascade Detection in OpenCV

Face detection using Haar cascades is a machine learning based approach where a cascade function is trained with a set of input data. OpenCV already contains many pre-trained classifiers for face, eyes, smiles, etc.. In our experiment we will use the face classifier.

OpenCV comes with a trainer as well as detector. If you want to train your own classifier for any object like car, planes etc. you can use OpenCV to create one. i.e Cascade Classifier Training. These classifiers as XML files are stored in opencv/data/haarcascades/ folder.

First we need to load the required XML classifiers. Then load our input image (or video) in grayscale mode.

face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

Then we find the faces in the image. If faces are found, it returns the positions of detected faces as Rect(x,y,w,h).

Once we get these locations, we can create a ROI for the face.

Exercise 10 - Face Detection and Tracking

a) Face Detection and Tracking using OpenCV

Program 1 - Face Detection and Tracking using OpenCV

AIM:

To Implement Face Detection and Tracking using OpenCV

MODULES REQUIRED:

Module	version
opencv-python	4.5.1.48

Part 1 - Importing the module and loading the Model

```
import cv2
face_cascade = cv2.CascadeClassifier('models/haarcascade_frontalface_default.xml')
```

Part 2 - Tracking the face from sample video

```
cap = cv2.VideoCapture('datasets/CERN_Higgs boson_EDIT.mp4')
while True:
## Read the frame
    ok, img = cap.read()
    if not ok:
        print('Video Ending')
        cap.release()
        cv2.destroyAllWindows()
        break
    # Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Detect the faces
    faces = face_cascade.detectMultiScale(gray, 1.1,4)
    # Draw the rectangle around each face
    for (x, y, w, h) in faces:
       cv2.rectangle(img, (x, y), (x+w, y+h),(0,0,255))
    # Display
    cv2.imshow('img', img)
    # quit Press Key Q to quit and Close window
    if cv2.waitKey(1) & 0xFF == ord('q'):
        # Release the VideoCapture object
        cap.release()
        # Close all window
        cv2.destroyAllWindows()
        break
```

11. Object Recognition

Introduction:

Object recognition is a computer vision technique for identifying objects in images or videos. Object recognition is a key output of machine learning algorithms. When humans look at a photograph or watch a video, we can readily spot people, objects, scenes, and visual details. The goal is to teach a computer to do what comes naturally to humans: to gain a level of understanding of what an image contains. This technology has been time tested also the work horse behind driverless cars, enabling them to recognize a stop sign or to distinguish a pedestrian from a lamppost. It is also useful in a variety of applications such as disease identification in bio-imaging, industrial inspection, and robotic vision.
An object recognition system finds objects in the real world from an image of the world, using object models which are known a priori. This task is surprisingly difficult. Humans perform object recognition effortlessly and instantaneously. Algorithmic description of this task for implementation on machines has been very difficult. In another aspect in machine learning technique object recognition problem can be defined as a labeling problem based on models of known objects. Formally, given an image containing one or more objects of interest (and background) and a set of labels corresponding to a set of models known to the system, the system should assign correct labels to regions, or a set of regions, in the image. Object detection and object recognition are similar techniques for identifying objects, but they vary in their execution. Object detection is the process of finding instances of objects in images. In the case of deep learning, object detection is a subset of object recognition, where the object is not only identified but also located in an image. This allows for multiple objects to be identified and located within the same image. In classical approach Object recognition is closely tied to the segmentation problem: without at least a partial recognition of objects, segmentation cannot be done, and without segmentation, object recognition is not possible with the advent of deep learning this depenency of segmentation has eliminated.

Classical approach vs Modern Paradigm

The traditional approach is to use well-established Computer Vision (CV) techniques such as feature descriptors (SIFT, SURF, BRIEF, etc.) for object detection. Before the emergence of Deep Learning (DL), a step called feature extraction was carried out for tasks such as image classification. Features are small “interesting”, descriptive or informative patches in images. Several CV algorithms, such as edge detection, corner detection or threshold segmentation may be involvedin this step. As many features as practicable are extracted from images and these features form a definition (known as a bag-of-words) of each object class. At the deployment stage, these definitions are searched for in other images. If a significant number of features from one bag-ofwords are in another image, the image is classified as containing that specific object (i.e. chair, horse, etc.)
Difficulty with the traditional approach is that it is necessary to choose which features are important in each given image. As the number of classes to classify increases, feature extraction becomes more and more cumbersome. It is up to the CV engineer’s judgment and a long trial and error process to decide which features best describe different classes of objects. Moreover, each feature definition requires dealing with a plethora of parameters, all of which must be fine-tuned by the CV engineer.

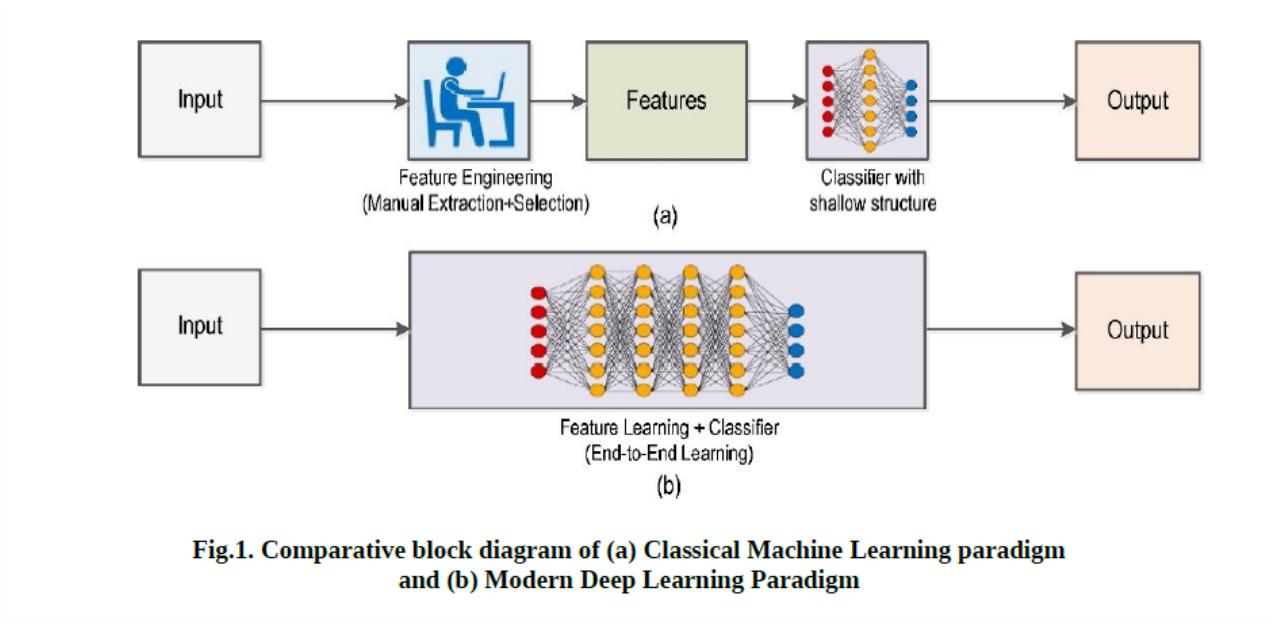


Fig.1. Comparative block diagram of (a) Classical Machine Learning paradigm and (b) Modern Deep Learning Paradigm

Image1

The traditional feature-based approaches such as those listed below have been shown to be useful in improving performance in CV tasks: - Hough transforms - Scale Invariant Feature Transform (SIFT) - Speeded Up Robust Features (SURF) - HoG Histogram of Gradient Feature descriptors such as SIFT and SURF are generally combined with traditional machine learning classification algorithms such as Support Vector Machines and K-Nearest Neighbours to solve the object recognition sCV problems

Deep Learning for Object Recognition/Classification Problem:

DL introduced the concept of end-to-end learning where the machine is just given a dataset of images which have been annotated with what classes of object are present in each image. Thereby a DL model is 'trained' on the given data, where neural networks discover the underlying patterns in classes of images and automatically works out the most descriptive and salient features with respect to each specific class of object for each object. It has been well-established that DNNs perform far better than traditional algorithms, albeit with trade-offs with respect to computing requirements.

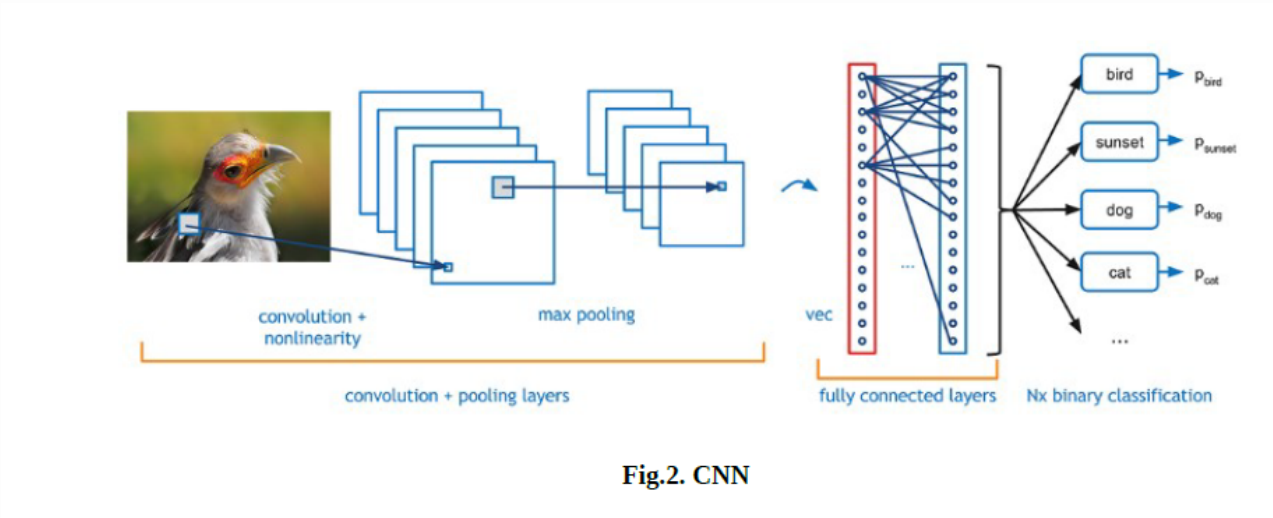


Fig.2. CNN

Image2

CNNs make use of kernels (also known as filters), to detect features (e.g. edges) throughout an image. A kernel is just a matrix of values, called weights, which are trained to detect specific features. As their name indicates, the main idea behind the CNNs is to spatially convolve the kernel on a given input image check if the feature it is meant to detect is present. To provide a value representing how confident it is that a specific feature is present, a convolution operation is carried out by computing the dot product of the kernel and the input area where kernel is overlapped (the area of the original image the kernel is looking at is known as the receptive field).

To facilitate the learning of kernel weights, the convolution layer's output is summed with a bias term and then fed to a non-linear activation function. Activation Functions are usually nonlinear functions like Sigmoid, tanh and ReLU (Rectified Linear Unit). Depending on the nature of data and classification tasks, these activation functions are selected accordingly. For example, ReLUs are known to have more biological representation (neurons in the brain either fire or they don't). As a result, it yields favourable results for image recognition tasks as it is less susceptible to the vanishing gradient problem and it produces sparser, more efficient representations.

To speed up the training process and reduce the amount of memory consumed by the network, the convolutional layer is often followed by a pooling layer to remove redundancy present in the input feature. For example, max pooling moves a window over the input and simply outputs the maximum value in that window effectively reducing to the important pixels in an image. As shown in Fig. 2, deep CNNs may have several pairs of convolutional and pooling layers . Finally, a Fully Connected layer flattens the previous layer volume into a feature vector and then an output layer which computes the scores (confidence or probabilities) for the output classes/features through a dense network. This output is then passed to a regression function such as Softmax,for example, which maps everything to a vector whose elements sum up to one. In our excercise we deploy CNN via Deep Learning Keras Framework for Demonstrating object recognition problem.

Exercise 11 - Object Recognition

a)Object Recognition using CNN

Program 1 - Implementation of Object Recognition in Cifar10 dataset using CNN

AIM:

To implement Object Recognition in Cidataset using CNN.

Modules used:

Modules	Version
tensorflow	2.6.0
numpy	1.19.5
pandas	1.3.0
matplotlib	3.4.3
joblib	1.0.1

Neural Network Architecture:

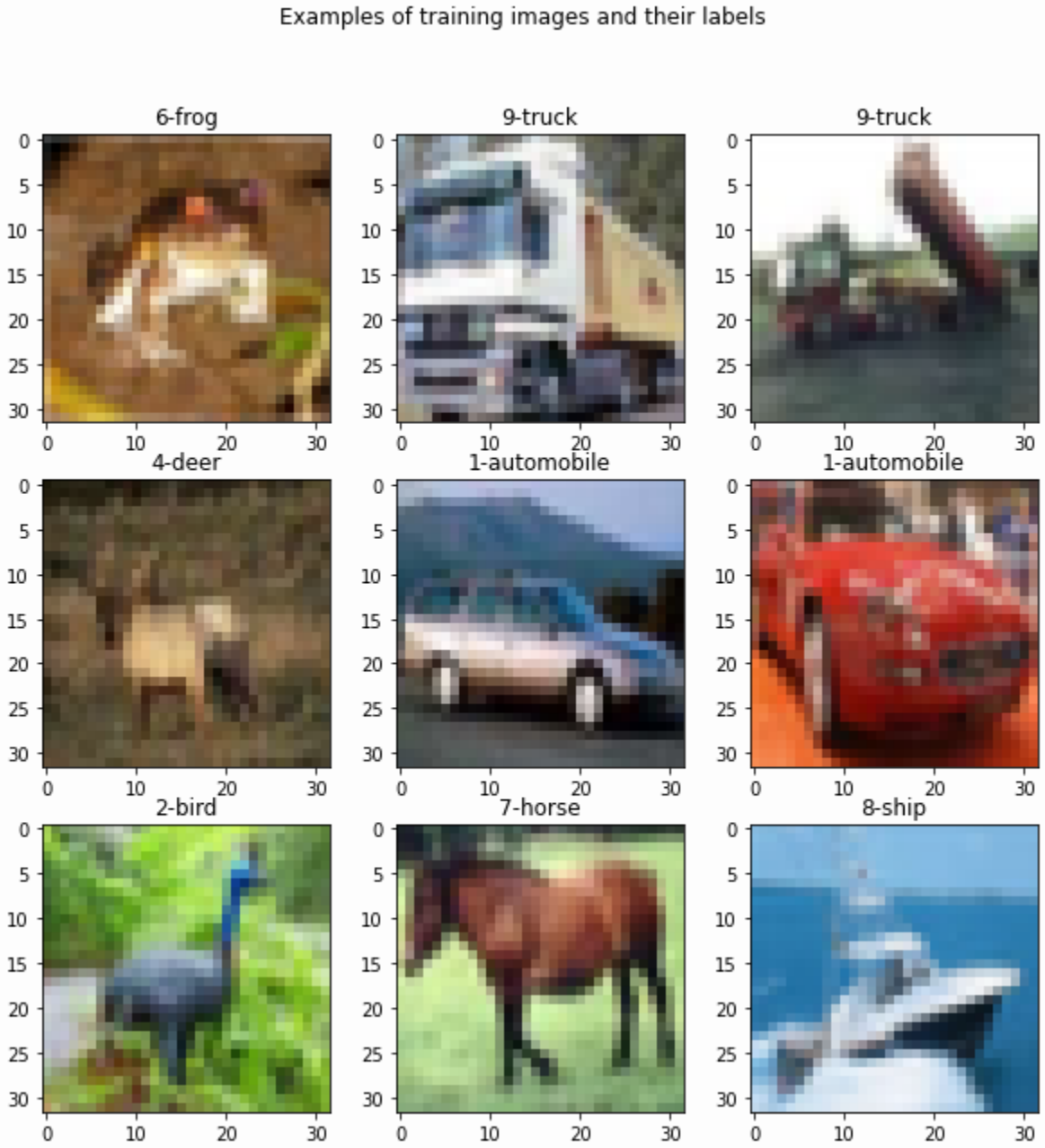
Layer (type)	Output Shape
Conv2D	(None, 32, 32, 25)
Conv2D(1)	(None, 32, 32, 50)
MaxPooling2D	(None, 16, 16, 50)
Dropout	(None, 16, 16, 50)
Conv2D(2)	(None, 16, 16, 70)
MaxPooling2D(1)	(None, 8, 8, 70)
Dropout(1)	(None, 8, 8, 70)
Flatten	(None, 4480)
Dense	(None, 500)
Dropout(2)	(None, 500)
Dense(1)	(None, 250)
Dropout(3)	(None, 250)
Dense(2)	(None, 10)

Part 1 - Importing the Modules and Loading the dataset

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import Sequential
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPool2D, Flatten
from tensorflow.keras.callbacks import Callback, ModelCheckpoint, EarlyStopping
from tensorflow import random
import joblib
import pandas as pd
```

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
cifar_classes =np.array( ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog','horse', 'shi
plt.figure(figsize=(10,10))
plt.suptitle('Examples of training images and their labels')
for i in range(9):
    plt.subplot(331 + i,title=f'{y_train[i,0]}-{cifar_classes[y_train[i,0]]}')
    plt.imshow(X_train[i])
plt.show()
```

Output:



```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
print(f"Shape of training data:\nX-train_shape={X_train.shape} , y-train_shape={y_train.shape}")
print(f"Shape of testing data:\nX-test_shape={X_test.shape} , y-test_shape={y_test.shape}")
```

Output:

```
Shape of training data:
X-train_shape=(50000, 32, 32, 3) , y-train_shape=(50000, 10)
Shape of testing data:
X-test_shape=(10000, 32, 32, 3) , y-test_shape=(10000, 10)
```

Part 2 - Creating the model

Building a linear stack of layers with the sequential model in tensorflow Keras

```
def get_model():
    model = Sequential()
    # convolutional layer
    model.add(Conv2D(25, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu', input_sh
    # convolutional layer
    model.add(Conv2D(50, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
    model.add(MaxPool2D(pool_size=(2,2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(70, kernel_size=(3,3), strides=(1,1), padding='same', activation='relu'))
    model.add(MaxPool2D(pool_size=(2,2)))
    model.add(Dropout(0.25))
    # flatten output of conv
    model.add(Flatten())
    # hidden layer
    model.add(Dense(500, activation='relu'))
    model.add(Dropout(0.4))
    model.add(Dense(250, activation='relu'))
    model.add(Dropout(0.3))
    # output layer
    model.add(Dense(10, activation='softmax'))
    # Compiling the Model
    model.compile(loss='categorical_crossentropy', metrics=['accuracy'],optimizer='adam')
    return model
```

```
random.set_seed(0)
model=get_model()
print(model.summary())
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 25)        700
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 50)        11300
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 50)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 50)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 70)        31570
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 70)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 70)          0
_________________________________________________________________
flatten (Flatten)            (None, 4480)              0
_________________________________________________________________
dense (Dense)                (None, 500)               2240500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 250)               125250
_________________________________________________________________
dropout_3 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2510
=================================================================
```

Part 3 - Training the Model

```
checkpoint= ModelCheckpoint("./model/cifar10_epo_cnn.h5",monitor='val_accuracy',mode='max', save_best_
early_stop = EarlyStopping(monitor="val_accuracy", patience=5, mode="max")
call_backs=[early_stop,checkpoint]
history = model.fit(X_train, y_train, batch_size=32, epochs=50, verbose=True,validation_split=0.2)
# Store Trained Model for Testing and Future use
model.save("./models/cifar10_epo_cnn.h5")
joblib.dump(history.history, "./models/cifar10_epo_cnn.history")
```

Output:

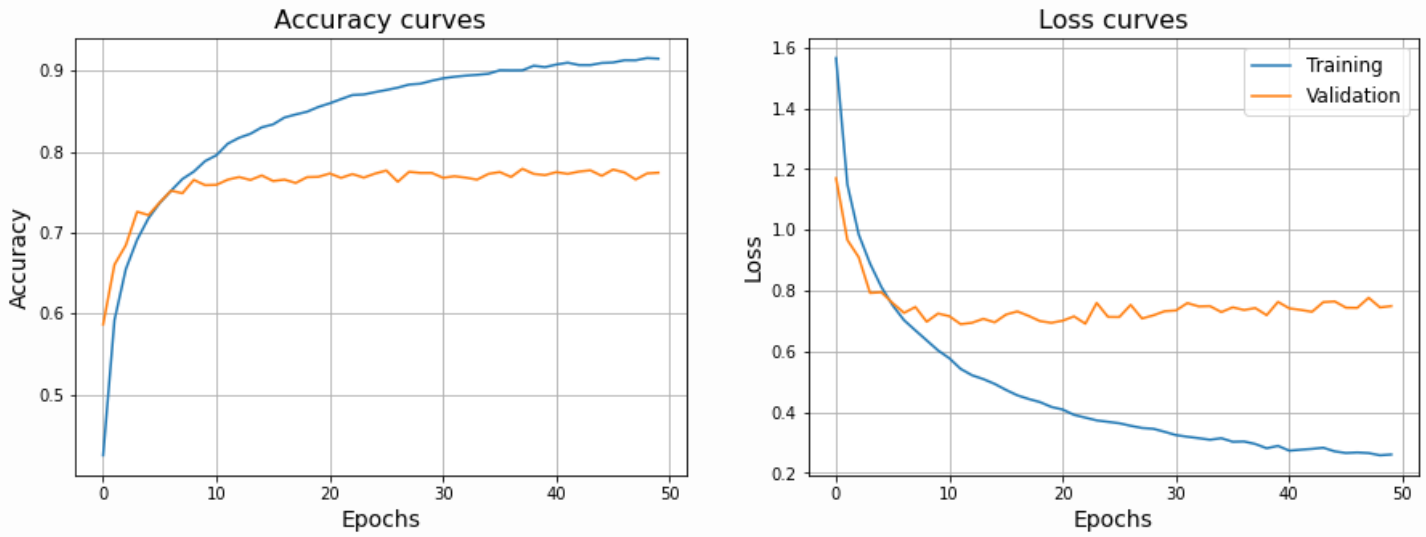
```
Epoch 1/50
1250/1250 [==============================] - 34s 10ms/step - loss: 1.5650 - accuracy: 0.4250 - val_loss:
```

```
.1693 - val_accuracy: 0.5866
Epoch 2/50
1250/1250 [==============================] - 11s 9ms/step - loss: 1.1485 - accuracy: 0.5918 - val_loss: 0
Epoch 3/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.9844 - accuracy: 0.6548 - val_loss: 0
Epoch 4/50
1250/1250 [==============================] - 12s 9ms/step - loss: 0.8886 - accuracy: 0.6912 - val_loss: 0
Epoch 5/50
1250/1250 [==============================] - 12s 9ms/step - loss: 0.8116 - accuracy: 0.7177 - val_loss: 0
Epoch 6/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.7510 - accuracy: 0.7366 - val_loss: 0
Epoch 7/50
1250/1250 [==============================] - 12s 9ms/step - loss: 0.7022 - accuracy: 0.7518 - val_loss: 0
Epoch 8/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.6688 - accuracy: 0.7662 - val_loss: 0
Epoch 9/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.6358 - accuracy: 0.7753 - val_loss: 0
Epoch 10/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.6029 - accuracy: 0.7883 - val_loss: 0
Epoch 11/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.5769 - accuracy: 0.7956 - val_loss: 0
Epoch 12/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.5419 - accuracy: 0.8098 - val_loss: 0
Epoch 13/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.5211 - accuracy: 0.8171 - val_loss: 0
Epoch 14/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.5084 - accuracy: 0.8220 - val_loss: 0
Epoch 15/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.4927 - accuracy: 0.8300 - val_loss: 0

.
.
.
Epoch 49/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.2577 - accuracy: 0.9156 - val_loss: 0
Epoch 50/50
1250/1250 [==============================] - 11s 9ms/step - loss: 0.2603 - accuracy: 0.9148 - val_loss: 0



['./models/cifar10_epo_cnn.history']
```

Part 4 - Plotting Accuracy and Loss curves

```
# Use only the below lines if model is not re trained
from keras.models import  load_model
model = load_model('./models/cifar10_epo_cnn.h5')
history = joblib.load("./models/cifar10_epo_cnn.history")

fig,ax=plt.subplots(1,2,figsize=(15,5))
# plotting Accuracy curves
ax[0].plot(history['accuracy'],'C0')
ax[0].plot(history['val_accuracy'],'C1')
ax[0].set_title(label="Accuracy curves",fontsize=16)
ax[0].set_xlabel('Epochs',fontsize=14)
ax[0].set_ylabel('Accuracy',fontsize=14)
ax[0].grid()
# plotting Loss curves
ax[1].plot(history['loss'],'C0')
ax[1].plot(history['val_loss'],'C1')
ax[1].set_title(label="Loss curves",fontsize=16)
ax[1].set_xlabel('Epochs',fontsize=14)
ax[1].set_ylabel('Loss',fontsize=14)
ax[1].grid()
plt.legend(['Training','Validation'],fontsize=12)
plt.show()
```

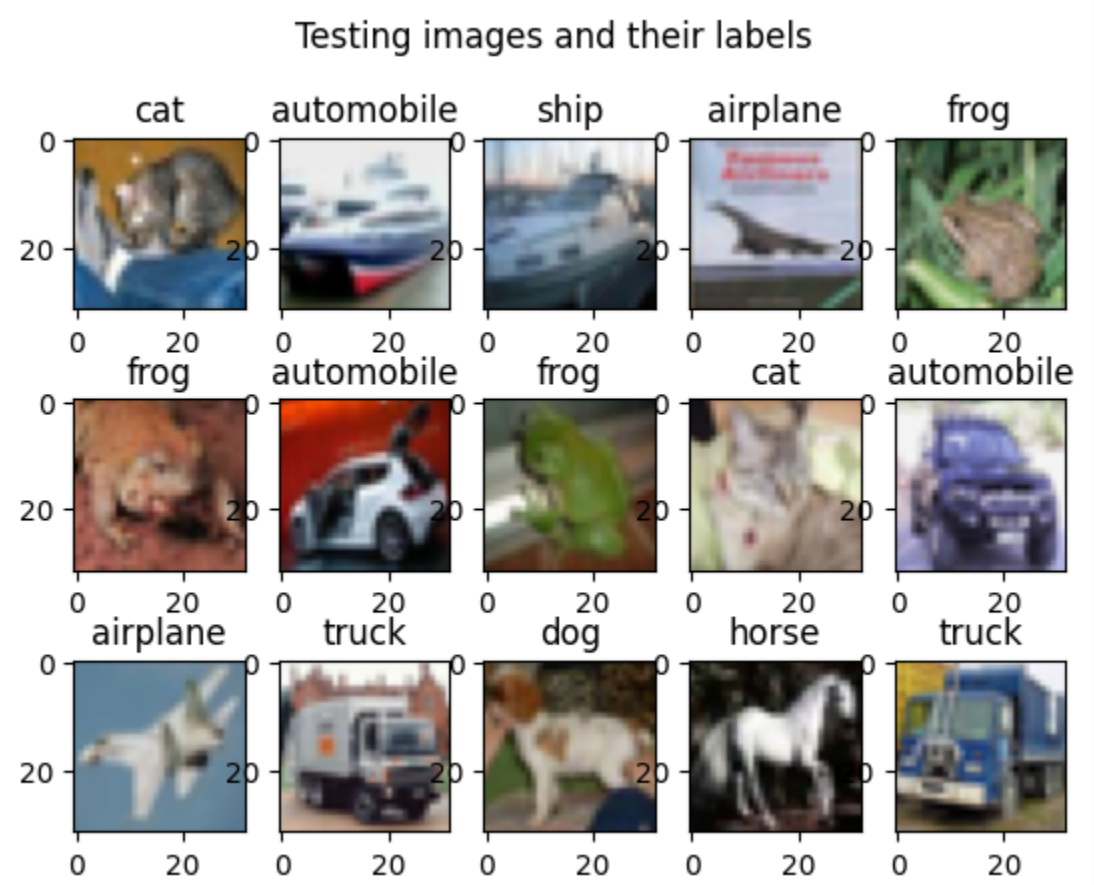
Output:



Part 6 - Testing

```
m=15
y_pred_=model.predict(X_test[:m])
y_pred_class=y_pred_.argmax(-1)
unique,counts=np.unique(y_pred_class,return_counts=True)
plt.suptitle('Testing images and their labels')
for i in range(15):
    plt.subplot(3,5,i+1,title=f'{cifar_classes[y_pred_class[i]]}')
    plt.imshow(X_test[i])
plt.show()
freq=list(zip(cifar_classes[unique],counts))
pd.DataFrame(freq,columns=["class name","count"])
```

Output:



	class name	count
0	airplane	2
1	automobile	3
2	cat	2
3	dog	1
4	frog	3
5	horse	1
6	ship	1
7	truck	2