

Activation Function

Activation functions introduce **non-linearity** into neural networks.

Without them, stacking multiple layers would still compute only a **linear transformation**, making deep networks no more powerful than a single linear layer.

- To model **non-linear** relationships in data (e.g., XOR, images, language).
- To control **neuron firing range** (probabilities, bounded outputs).
- To help **optimization** by shaping gradients during backpropagation.

Sigmoid (Logistic) Activation

Formula:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- **Range:** $(0, 1)$ — interpretable as probability.
- **Pros:** Smooth, interpretable as probability, useful for binary classification output.
- **Cons:** Vanishing gradient problem, not zero-centered.
- **Best use:** Output layer of **binary classification problems**.

ReLU (Rectified Linear Unit)

Formula:

$$ReLU(z) = \max(0, z)$$

- **Range:** $[0, \infty)$
- **Pros:** Solves vanishing gradient (for positive z), simple & fast, sparse activation.
- **Cons:** “Dying ReLU” problem (neuron stuck at 0 for all inputs).
- **Best use:** Default for **hidden layers** in deep networks.

Activation Functions in ANN Flow

$$z = Wx + b, \quad a = f(z)$$

$f(z)$ is the activation function (Sigmoid, ReLU, etc.).

Without $f(z)$, networks remain linear → limited power.

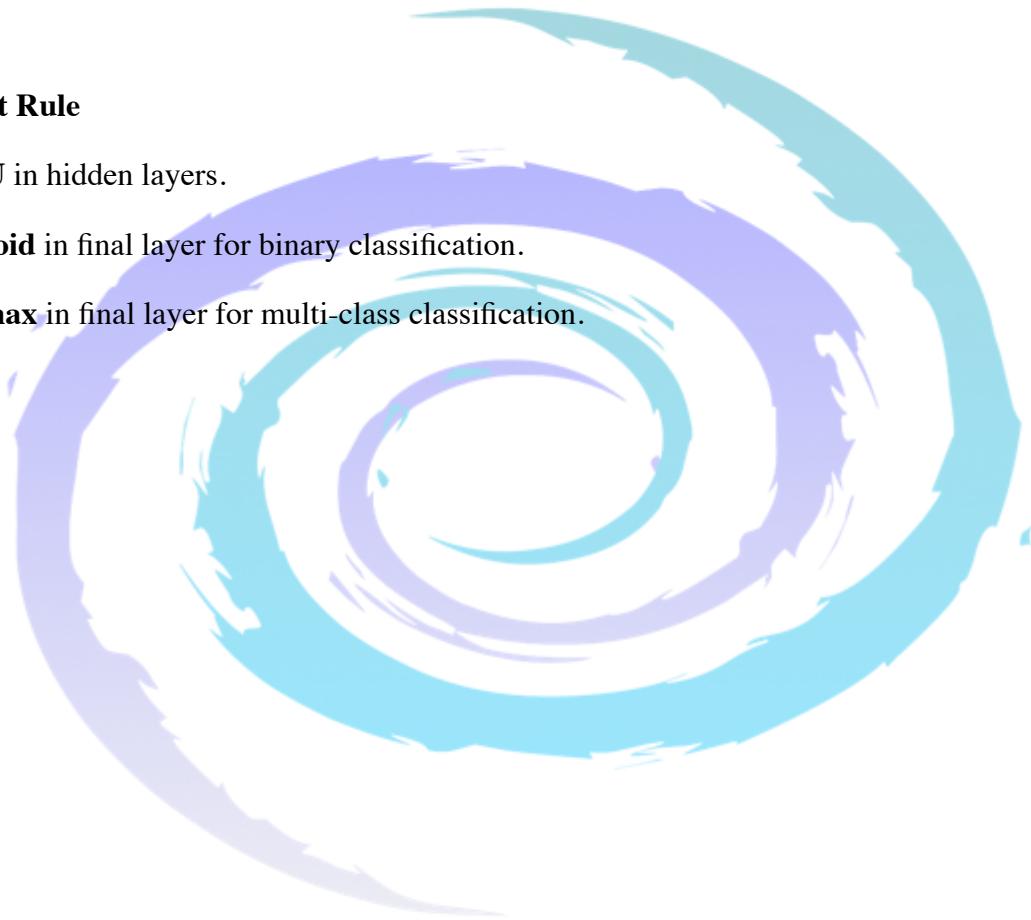
During backpropagation, gradients flow through $f'(z)$. If derivative ≈ 0 (Sigmoid saturation), learning slows.

Important Rule

Use **ReLU** in hidden layers.

Use **Sigmoid** in final layer for binary classification.

Use **Softmax** in final layer for multi-class classification.



Consider below application to demonstrate the concept of

Sigmoid function as a activation function

```

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

# Sigmoid activation function
def sigmoid(z):
    """Sigmoid function: squashes values to (0,1) range."""
    return 1 / (1 + math.exp(-z))

# ---- Neuron calculation ----
def Marvellous_neuron_forward(inputs, weights, bias):
    # 1) Display inputs and weights
    print("Inputs (x): ", inputs)
    print("Weights (w):", weights)
    print("Bias (b): ", bias)

    # 2) Summation  $z = w \cdot x + b$ 
    z = sum(w * x for w, x in zip(weights, inputs)) + bias
    print("Summation (z = w \cdot x + b):", z)

    # 3) Activation function output
    y_hat = sigmoid(z)
    print("Activation Function: Sigmoid")
    print("Output ( $\hat{y} = \text{sigmoid}(z)$ ):", y_hat)

    return z, y_hat

# ---- Plot Sigmoid function ----
def plot_sigmoid():
    z_values = np.linspace(-10, 10, 200)  # range of z values
    sigmoid_values = 1 / (1 + np.exp(-z_values))

    plt.figure(figsize=(8, 5))
    plt.plot(z_values, sigmoid_values, label="Sigmoid", linewidth=2, color="blue")
    plt.axhline(y=0, color="black", linewidth=0.5)
    plt.axhline(y=1, color="black", linewidth=0.5)
    plt.axvline(x=0, color="gray", linestyle="--")
    plt.title("Sigmoid Activation Function", fontsize=16)
    plt.xlabel("Summation (z)", fontsize=14)
    plt.ylabel("Activation Output", fontsize=14)
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.legend()
    plt.show()

def main():
    # Example inputs, weights, and bias
    inputs = [1.0, 2.0, 3.0]  # Example input features
    weights = [0.6, 0.4, -0.2] # Weights for each input
  
```

```
bias = 0.5          # Bias term

# Run the neuron forward pass
z, y_hat = Marvellous_neuron_forward(inputs, weights, bias)

# Plot sigmoid curve
plot_sigmoid()

if __name__ == "__main__":
    main()
```



Consider below application to demonstrate the concept of

ReLU function as a activation function

```

import matplotlib.pyplot as plt
import numpy as np

# ReLU activation function
def relu(z):
    return max(0, z)

# ---- Neuron calculation ----
def Marvellous_neuron_forward(inputs, weights, bias):
    # 1) Display inputs and weights
    print("Inputs (x): ", inputs)
    print("Weights (w):", weights)
    print("Bias (b): ", bias)

    # 2) Summation  $z = w \cdot x + b$ 
    z = sum(w * x for w, x in zip(weights, inputs)) + bias
    print("Summation (z = w \cdot x + b):", z)

    # 3) Activation function output
    y_hat = relu(z)
    print("Activation Function: ReLU")
    print("Output ( $\hat{y} = \text{relu}(z)$ ):", y_hat)

    return z, y_hat

def plot_relu():
    z_values = np.linspace(-10, 10, 200)  # range of z values
    relu_values = np.maximum(0, z_values)

    plt.figure(figsize=(8, 5))
    plt.plot(z_values, relu_values, label="ReLU", linewidth=2, color="green")
    plt.axhline(y=0, color="black", linewidth=0.5)
    plt.axvline(x=0, color="gray", linestyle="--")
    plt.title("ReLU Activation Function", fontsize=16)
    plt.xlabel("Summation (z)", fontsize=14)
    plt.ylabel("Activation Output", fontsize=14)
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.legend()
    plt.show()

def main():
    # Example inputs, weights, and bias
    inputs = [1.0, 2.0, 3.0]  # Example input features
    weights = [0.6, 0.4, -0.2]  # Weights for each input
    bias = 0.5  # Bias term

    # Run the neuron forward pass
    z, y_hat = Marvellous_neuron_forward(inputs, weights, bias)

    plot_relu()

if __name__ == "__main__":
    main()

```

Consider below application to demonstrate the comparison in Sigmoid and ReLU function as a activation function

```

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

# -----
# Activation functions
# -----
def sigmoid(z):
    """Sigmoid function: squashes values to (0,1)."""
    return 1 / (1 + math.exp(-z))

def relu(z):
    """ReLU function: outputs z if positive, else 0."""
    return max(0, z)

# -----
# Neuron calculation
# -----
def Marvellous_neuron_forward(inputs, weights, bias, activation_func):
    # 1) Display inputs and weights
    print("Inputs (x): ", inputs)
    print("Weights (w):", weights)
    print("Bias (b): ", bias)

    # 2) Summation
    z = sum(w * x for w, x in zip(weights, inputs)) + bias
    print("Summation (z = w·x + b):", z)

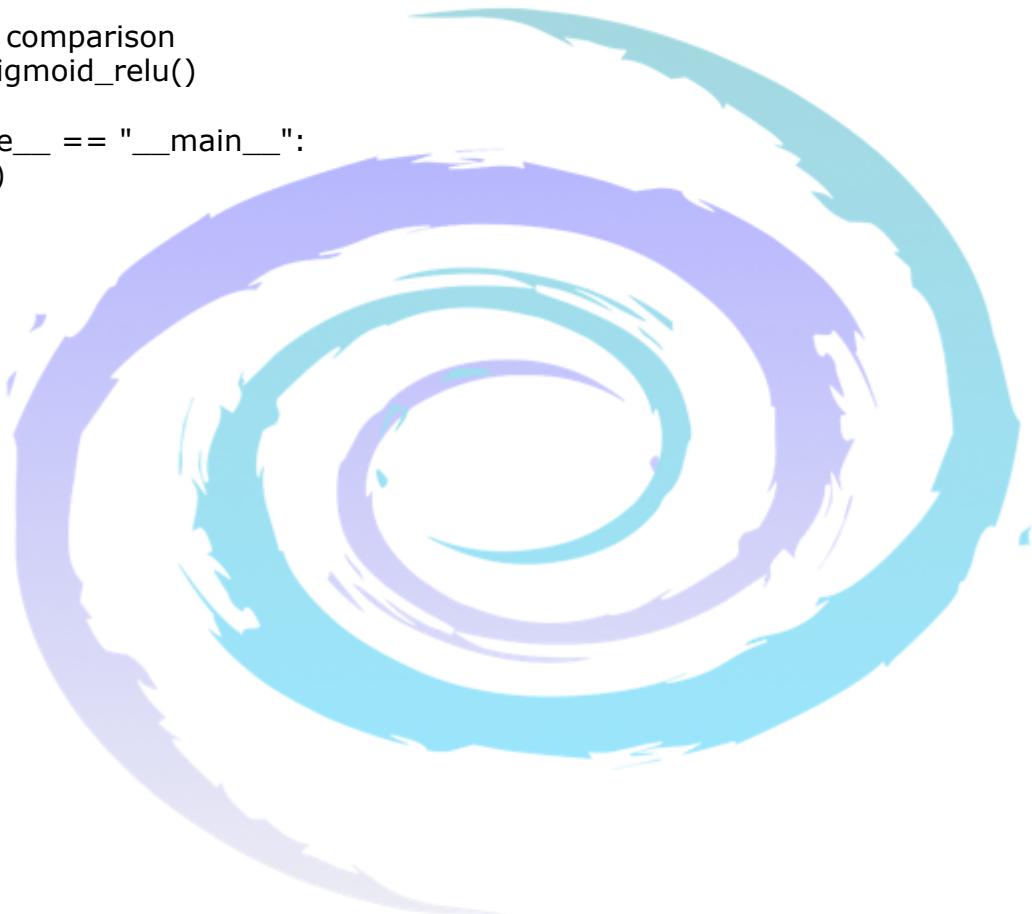
    # 3) Activation
    y_hat = activation_func(z)
    print(f"Activation Function: {activation_func.__name__}")
    print(f"Output (ŷ): {y_hat}\n")

    return z, y_hat

# -----
# Plot Sigmoid and ReLU
# -----
def plot_sigmoid_relu():
    z_values = np.linspace(-10, 10, 200)
    sigmoid_values = 1 / (1 + np.exp(-z_values))
    relu_values = np.maximum(0, z_values)

    plt.figure(figsize=(8, 5))
    plt.plot(z_values, sigmoid_values, label="Sigmoid", linewidth=2, color="blue")
    plt.plot(z_values, relu_values, label="ReLU", linewidth=2, color="green")
    plt.axhline(y=0, color="black", linewidth=0.5)
    plt.axhline(y=1, color="black", linewidth=0.5)
    plt.axvline(x=0, color="gray", linestyle="--")
    plt.title("Sigmoid vs ReLU Activation Functions", fontsize=16)
    plt.xlabel("Summation (z)", fontsize=14)
    plt.ylabel("Activation Output", fontsize=14)
    plt.grid(True, linestyle="--", alpha=0.6)
  
```

```
plt.legend()  
plt.show()  
  
def main():  
    # Example inputs, weights, bias  
    inputs = [1.0, 2.0, 3.0]  
    weights = [0.6, 0.4, -0.2]  
    bias = 0.5  
  
    print("==== Sigmoid Neuron ====")  
    Marvellous_neuron_forward(inputs, weights, bias, sigmoid)  
  
    print("==== ReLU Neuron ====")  
    Marvellous_neuron_forward(inputs, weights, bias, relu)  
  
    # Plot comparison  
    plot_sigmoid_relu()  
  
if __name__ == "__main__":  
    main()
```



Activation Functions of Artificial Neural Network

Activation functions introduce **non-linearity** into the neural network, allowing it to learn **complex patterns**. Without them, the network would act like just a linear regression model.

Sigmoid / Logistic Function

- **Definition:**

$$f(x) = \frac{1}{1 + e^{-x}}$$

- Range: $0 \rightarrow 1$

- **Explanation:**

- Converts any input into a probability-like value.
- Commonly used in binary classification.

- **Example:**

If input = 2 \rightarrow sigmoid(2) $\approx 0.88 \rightarrow$ interpreted as 88% probability of being “positive class.”

- **Limitation:** Can suffer from **vanishing gradients** for very large or very small values.

ReLU (Rectified Linear Unit)

- **Definition:**

$$f(x) = \max(0, x)$$

- **Range:** $0 \rightarrow \infty$

- **Explanation:**

- If input $< 0 \rightarrow$ output = 0.
- If input $\geq 0 \rightarrow$ output = input itself.
- Makes networks train faster and deeper.

- **Example:**

If inputs = [-2, 0, 3] \rightarrow ReLU outputs [0, 0, 3].

- **Limitation:** “Dead ReLU” problem when too many outputs become zero.

Tanh (Hyperbolic Tangent)

- **Definition:**

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **Range:** $-1 \rightarrow 1$

- **Explanation:**

- Like sigmoid, but centered at 0.
- Useful when data has both positive and negative values.

- **Example:**

If input = 1 $\rightarrow \tanh(1) \approx 0.76$.

- **Advantage:** Stronger gradients than sigmoid, better for training deep networks.

Softmax

- **Definition:**

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- **Range:** $0 \rightarrow 1$, sum of all outputs = 1.

- **Explanation:**

- Converts raw outputs into probability distribution.
- Mainly used in the **output layer** for multi-class classification.

- **Example:**

Input scores = [2, 1, 0.1] \rightarrow Softmax $\approx [0.65, 0.24, 0.11]$.
 (Class 1 has the highest probability).

Supervised Machine Learning Algorithms

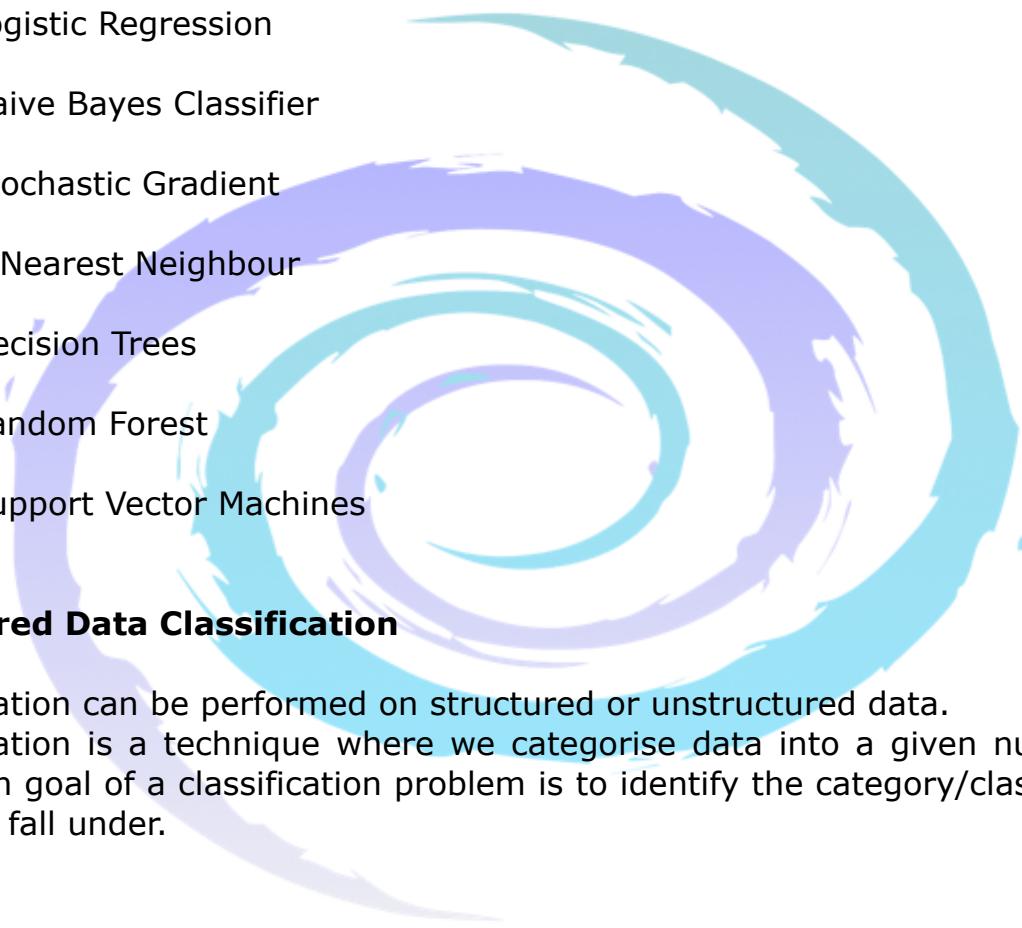
In machine learning, classification is a supervised learning approach in which the computer program learns from the data input given to it and then uses this learning to classify new observation.

This data set may simply be bi-class (like identifying whether the person is male or female or that the mail is spam or non-spam) or it may be multi-class too.

Some examples of classification problems are: speech recognition, handwriting recognition, bio metric identification, document classification etc.

Here we have the types of classification algorithms in Machine Learning:

- Logistic Regression
- Naive Bayes Classifier
- Stochastic Gradient
- K Nearest Neighbour
- Decision Trees
- Random Forest
- Support Vector Machines



Structured Data Classification

Classification can be performed on structured or unstructured data.

Classification is a technique where we categorise data into a given number of classes. The main goal of a classification problem is to identify the category/class to which a new data will fall under.

Few of the terminologies encountered in machine learning – classification:

Classifier:

An algorithm that maps the input data to a specific category.

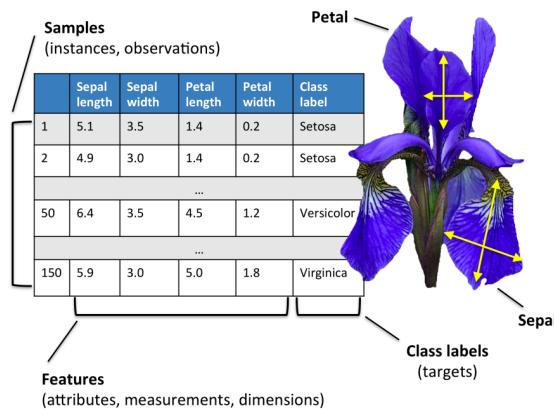
Classification model:

A classification model tries to draw some conclusion from the input values given for training.

It will predict the class labels/categories for the new data.

Feature:

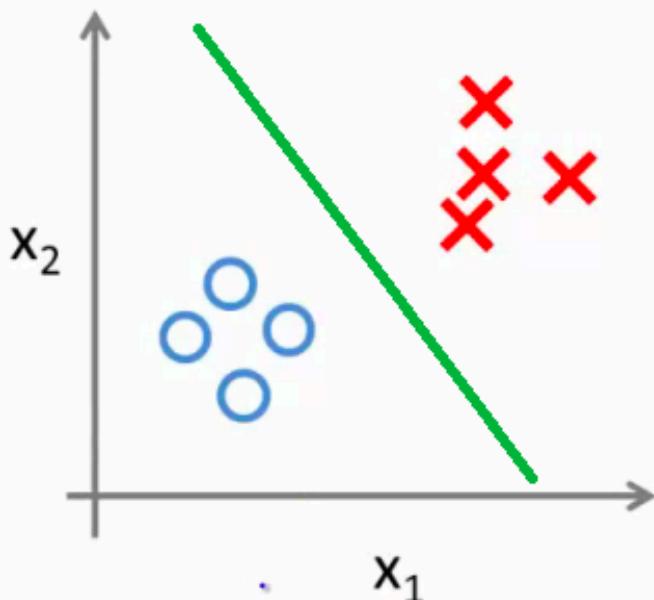
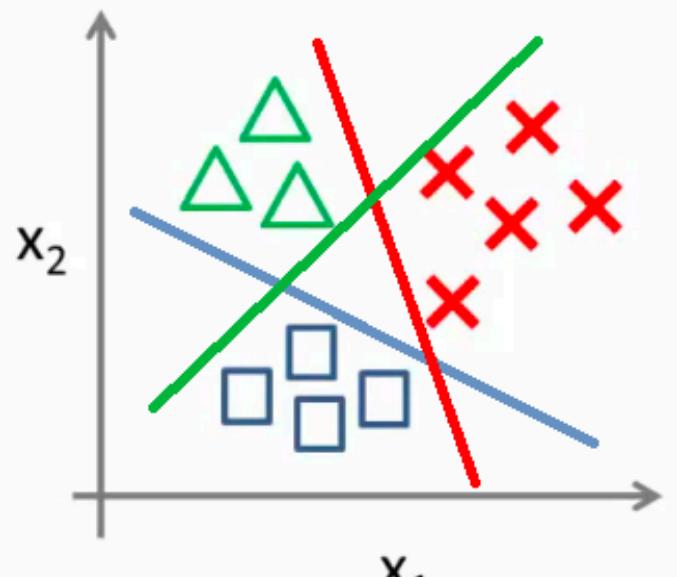
A feature is an individual measurable property of a phenomenon being observed.

**Binary Classification:**

Classification task with two possible outcomes. Eg: Gender classification (Male / Female)

Multi class classification:

Classification with more than two classes. In multi class classification each sample is assigned to one and only one target label. Eg: An animal can be cat or dog but not both at the same time

Binary classification:**Multi-class classification:****Multi label classification:**

Classification task where each sample is mapped to a set of target labels (more than one class). Eg: A news article can be about sports, a person, and location at the same time.

The following are the steps involved in building a classification model

Step 1:

Initialise the classifier to be used.

Step 2:

Train the classifier:

All classifiers in scikit-learn uses a **fit(X, y)** method to fit the model(training) for the given train data X and train label y.

Step 3:

Predict the target:

Given an unlabeled observation X, the **predict(X)** returns the predicted label y.

Step 4:

Evaluate the classifier model

Example :

```
from sklearn import tree
```

```
# Data Set
```

```
BallIsFeatures = [[35,1],[47,1],[90,0],[48,1],[90,0],[35,1],[92,0],[35,1],[35,1],[35,1],[96,0],[43,1],[110,0],[35,1],[95,0]]
```

```
# Features
```

```
Names = [1,1,2,1,2,1,2,1,1,1,2,1,2,1,2]
```

```
# Step 1 Initialise the classifier
```

```
clf = tree.DecisionTreeClassifier()
```

```
# Step 2 Train the classifier
```

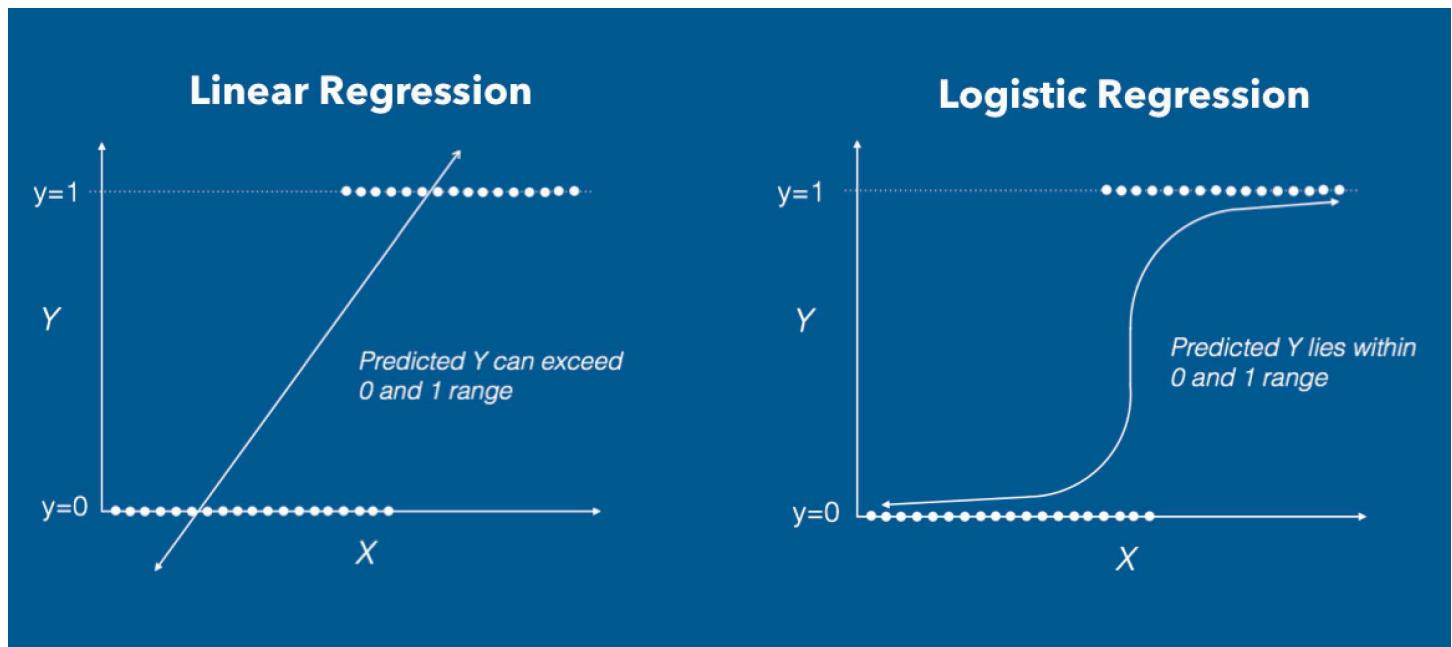
```
clf = clf.fit(BallIsFeatures,Names)
```

```
# Step 3 Predict the target
```

```
print(clf.predict([[44,1]]))
```

Classification Algorithms

Logistic Regression Algorithm (Predictive Learning Model)



Definition:

Logistic regression is a machine learning algorithm for classification. In this algorithm, the probabilities describing the possible outcomes of a single trial are modelled using a logistic function.

It is a statistical method for analysing a data set in which there are one or more independent variables that determine an outcome.

The outcome is measured with a dichotomous variable (in which there are only two possible outcomes).

The goal of logistic regression is to find the best fitting model to describe the relationship between the dichotomous characteristic of interest (dependent variable = response or outcome variable) and a set of independent (predictor or explanatory) variables.

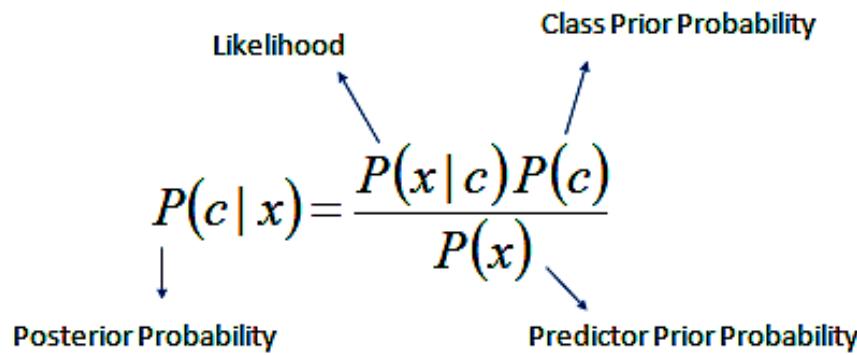
Advantages:

Logistic regression is designed for this purpose (classification), and is most useful for understanding the influence of several independent variables on a single outcome variable.

Disadvantages:

Works only when the predicted variable is binary, assumes all predictors are independent of each other, and assumes data is free of missing values.

Naïve Bayes Algorithm (Generative Learning Model)



$$P(c | X) = P(x_1 | c) \times P(x_2 | c) \times \dots \times P(x_n | c) \times P(c)$$

Definition:

Naive Bayes algorithm based on Bayes' theorem with the assumption of independence between every pair of features.

Naive Bayes classifiers work well in many real-world situations such as document classification and spam filtering.

It is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature.

Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability.

Naive Bayes model is easy to build and particularly useful for very large data sets.

Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

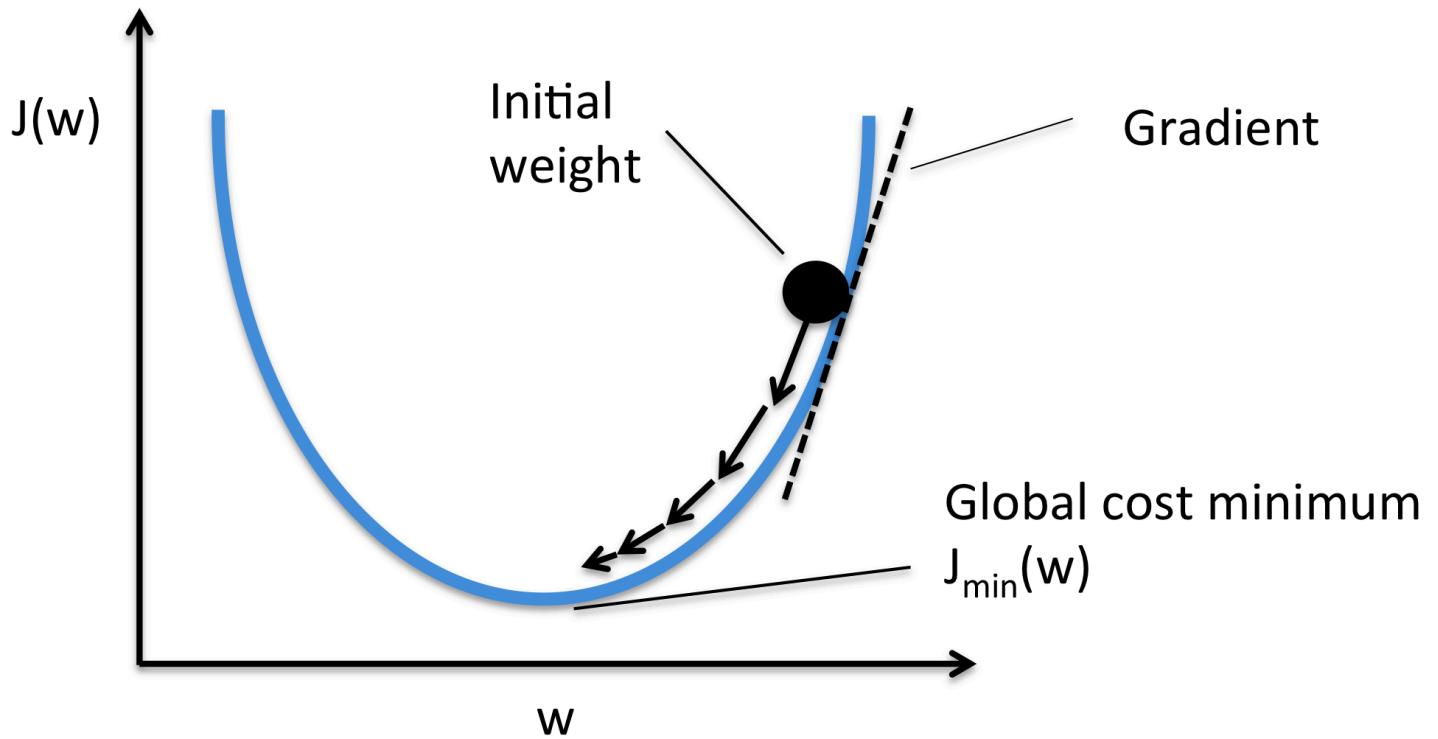
Advantages:

This algorithm requires a small amount of training data to estimate the necessary parameters. Naive Bayes classifiers are extremely fast compared to more sophisticated methods.

Disadvantages:

Naive Bayes is known to be a bad estimator.

Stochastic Gradient Descent Algorithm



Definition:

Stochastic gradient descent is a simple and very efficient approach to fit linear models. It is particularly useful when the number of samples is very large. It supports different loss functions and penalties for classification.

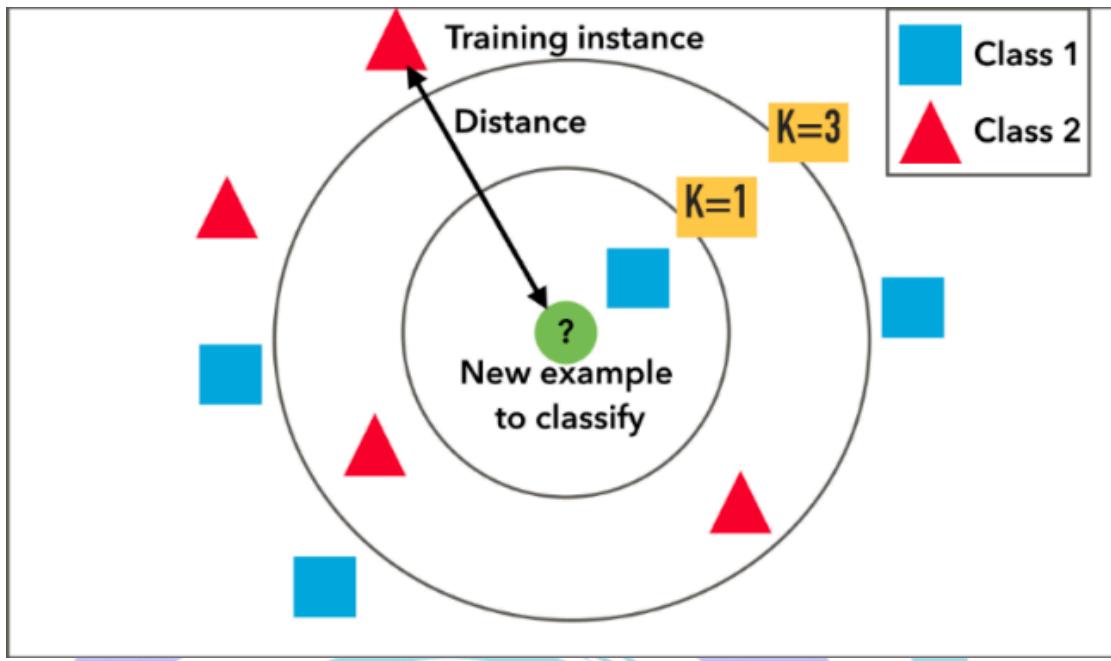
Advantages:

Efficiency and ease of implementation.

Disadvantages:

Requires a number of hyper-parameters and it is sensitive to feature scaling.

K-Nearest Neighbours Algorithm



Definition:

Neighbours based classification is a type of lazy learning as it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the k nearest neighbours of each point.

The k -nearest-neighbors algorithm is a classification algorithm, and it is supervised: it takes a bunch of labelled points and uses them to learn how to label other points.

To label a new point, it looks at the labelled points closest to that new point (those are its nearest neighbors), and has those neighbors vote, so whichever label the most of the neighbors have is the label for the new point (the " k " is the number of neighbors it checks).

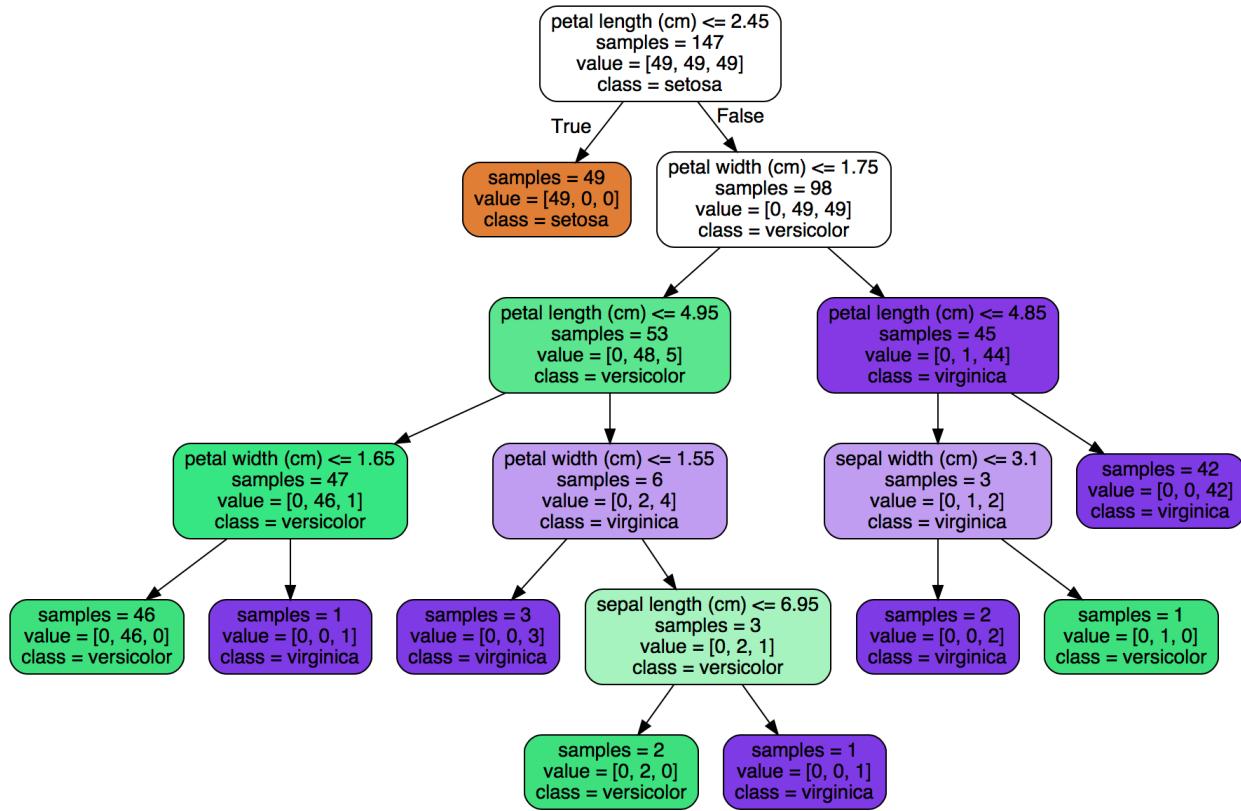
Advantages:

This algorithm is simple to implement, robust to noisy training data, and effective if training data is large.

Disadvantages:

Need to determine the value of K and the computation cost is high as it needs to computer the distance of each instance to all the training samples.

Decision Tree Algorithm



Definition:

Given a data of attributes together with its classes, a decision tree produces a sequence of rules that can be used to classify the data.

Decision tree builds classification or regression models in the form of a tree structure. It breaks down a data set into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.

The final result is a tree with decision nodes and leaf nodes.

A decision node has two or more branches and a leaf node represents a classification or decision.

The topmost decision node in a tree which corresponds to the best predictor called root node. Decision trees can handle both categorical and numerical data.

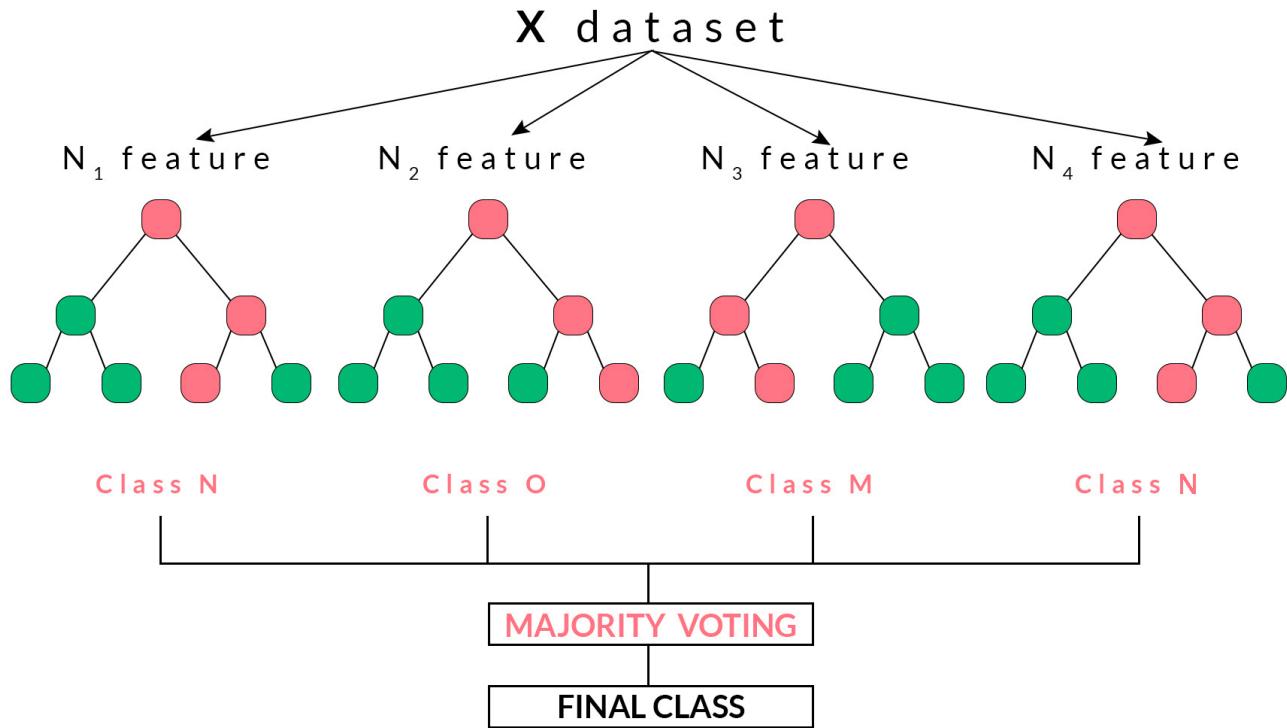
Advantages:

Decision Tree is simple to understand and visualise, requires little data preparation, and can handle both numerical and categorical data.

Disadvantages:

Decision tree can create complex trees that do not generalise well, and decision trees can be unstable because small variations in the data might result in a completely different tree being generated.

Random Forest Algorithm



Definition:

Random forest classifier is a meta-estimator that fits a number of decision trees on various sub-samples of datasets and uses average to improve the predictive accuracy of the model and controls over-fitting.

The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement.

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

Random decision forests correct for decision trees' habit of over fitting to their training set.

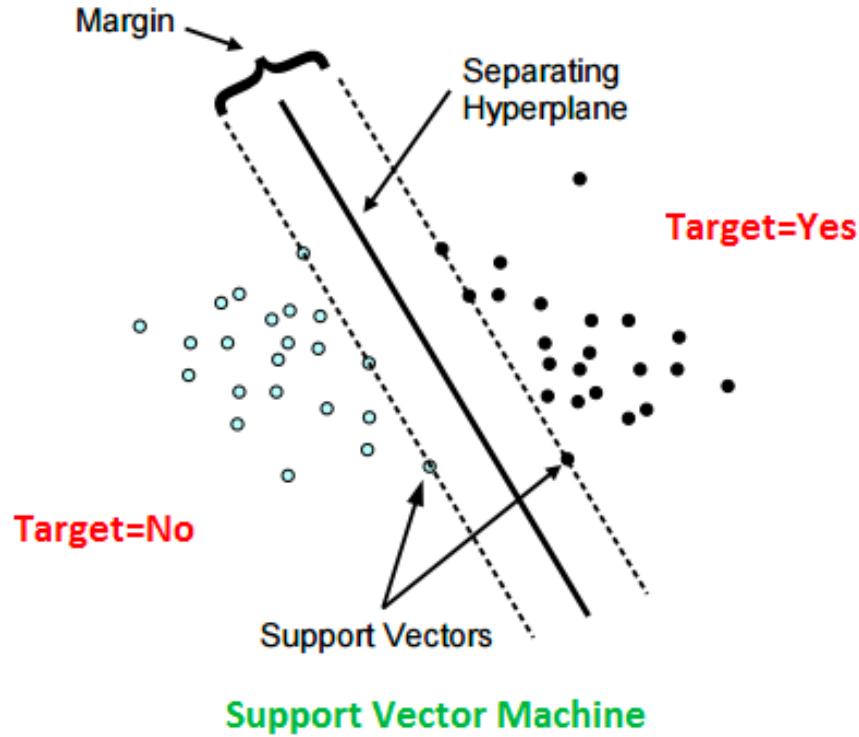
Advantages:

Reduction in over-fitting and random forest classifier is more accurate than decision trees in most cases.

Disadvantages:

Slow real time prediction, difficult to implement, and complex algorithm.

Support Vector Machine Algorithm



Definition:

Support vector machine is a representation of the training data as points in space separated into categories by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

Advantages:

Effective in high dimensional spaces and uses a subset of training points in the decision function so it is also memory efficient.

Disadvantages:

The algorithm does not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

We can compare each algorithm by considering its Accuracy and F1-Score

Accuracy:

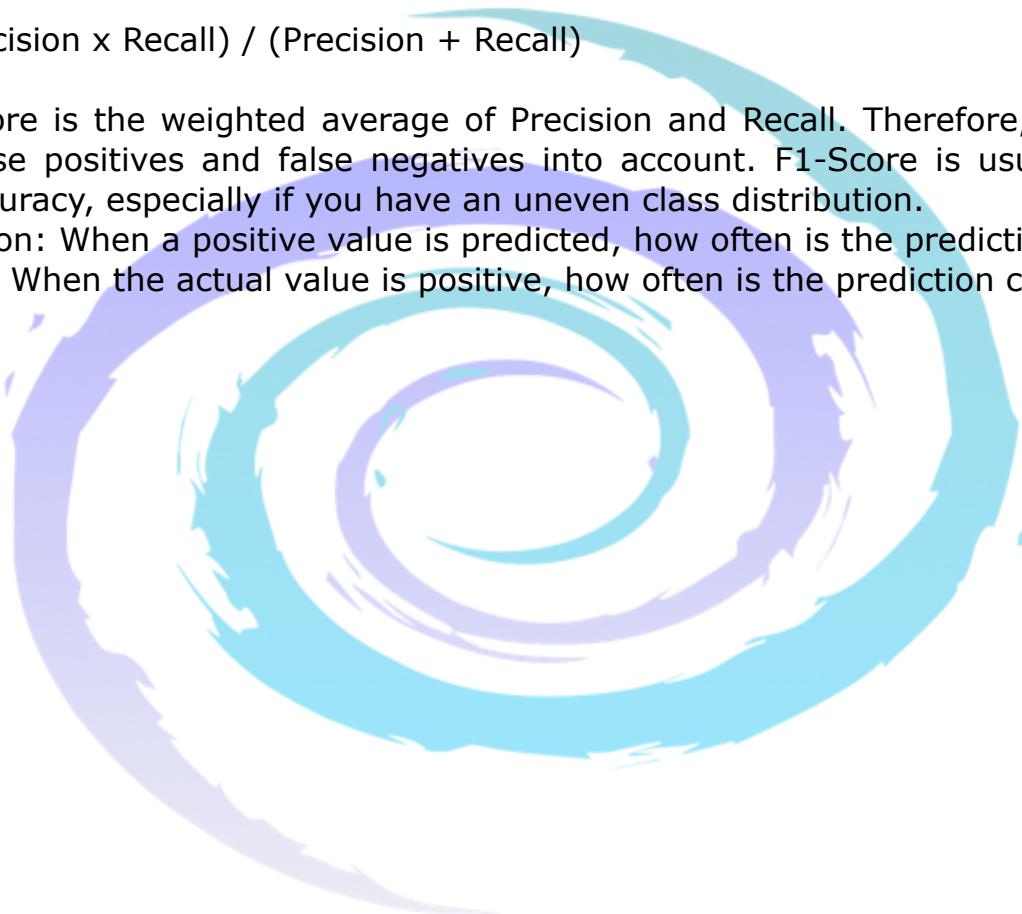
(True Positive + True Negative) / Total Population

- Accuracy is a ratio of correctly predicted observation to the total observations. Accuracy is the most intuitive performance measure.
- True Positive: The number of correct predictions that the occurrence is positive
- True Negative: The number of correct predictions that the occurrence is negative

F1-Score:

($2 \times \text{Precision} \times \text{Recall}$) / ($\text{Precision} + \text{Recall}$)

- F1-Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. F1-Score is usually more useful than accuracy, especially if you have an uneven class distribution.
- Precision: When a positive value is predicted, how often is the prediction correct?
- Recall: When the actual value is positive, how often is the prediction correct?



Apparel Classification System using Deep Learning

```
#####
# Function: load_data
# Inputs: val_split (float) - validation split ratio
# Outputs: (train_data, val_data, test_data) tuples
# Description: Loads Fashion-MNIST dataset, normalizes and splits into train, val, test sets.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def load_data(val_split=0.1) -> Tuple[Tuple[np.ndarray, np.ndarray],
                                         Tuple[np.ndarray, np.ndarray],
                                         Tuple[np.ndarray, np.ndarray]]:
    (x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
    x_train, x_val, y_train, y_val = train_test_split(
        x_train, y_train, test_size=val_split, random_state=SEED, stratify=y_train
    )
    x_train = (x_train.astype("float32") / 255.0)[..., None]
    x_val = (x_val.astype("float32") / 255.0)[..., None]
    x_test = (x_test.astype("float32") / 255.0)[..., None]
    return (x_train, y_train), (x_val, y_val), (x_test, y_test)

#####
# Function: load_flattened
# Inputs: None
# Outputs: Flattened train and test arrays with labels
# Description: Loads Fashion-MNIST and flattens images for classical ML models.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def load_flattened():
    (x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
    x_train = x_train.reshape(len(x_train), -1).astype("float32") / 255.0
    x_test = x_test.reshape(len(x_test), -1).astype("float32") / 255.0
    return x_train, y_train, x_test, y_test

#####
# Function: build_cnn
# Inputs: lr (float) - learning rate
# Outputs: Compiled Keras CNN model
# Description: Builds and compiles CNN with augmentation, Conv2D, Dropout, BatchNorm, Dense.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def build_cnn(lr=1e-3) -> keras.Model:
    data_augmentation = keras.Sequential(
        [
            layers.RandomTranslation(0.05, 0.05, fill_mode="nearest"),
            layers.RandomRotation(0.05, fill_mode="nearest"),
            layers.RandomZoom(0.05, 0.05, fill_mode="nearest"),
        ],
    )

    # Add layers here: Conv2D, BatchNorm, Dropout, Dense, etc.
```

```

    name="augmentation",
)
inputs = keras.Input(shape=(28, 28, 1))
x = data_augmentation(inputs)
x = layers.Conv2D(32, 3, padding="same", activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(32, 3, padding="same", activation="relu")(x)
x = layers.MaxPooling2D()(x)
x = layers.Dropout(0.25)(x)

x = layers.Conv2D(64, 3, padding="same", activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(64, 3, padding="same", activation="relu")(x)
x = layers.MaxPooling2D()(x)
x = layers.Dropout(0.25)(x)

x = layers.Flatten()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.40)(x)
outputs = layers.Dense(10, activation="softmax")(x)

model = keras.Model(inputs, outputs, name="fashion_cnn")
opt = keras.optimizers.Adam(learning_rate=lr)
model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])
return model

#####
# Function: train_and_evaluate
# Inputs: batch_size (int), epochs (int), lr (float)
# Outputs: Trained model + saved artifacts
# Description: Trains CNN on Fashion-MNIST, evaluates, saves curves, confusion matrix, misclassifications, report.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def train_and_evaluate(batch_size=128, epochs=15, lr=1e-3):
    ensure_dir(ARTIFACT_DIR)
    (x_train, y_train), (x_val, y_val), (x_test, y_test) = load_data(val_split=0.1)
    model = build_cnn(lr=lr)
    model.summary()
    callbacks = [
        keras.callbacks.ModelCheckpoint(BEST_MODEL, monitor="val_accuracy", save_best_only=True,
verbose=1),
        keras.callbacks.EarlyStopping(patience=4, restore_best_weights=True, monitor="val_loss"),
        keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=2, min_lr=1e-5, verbose=1),
    ]
    history = model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        batch_size=batch_size,
        epochs=epochs,
        callbacks=callbacks,
        verbose=2
    )

```

```

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"[TEST] loss: {test_loss:.4f} | acc: {test_acc:.4f}")

model.save(FINAL_MODEL)
plot_training_curves(history, ARTIFACT_DIR)
y_pred = model.predict(x_test, batch_size=256).argmax(axis=1)
plot_confusion_matrix(y_test, y_pred, ARTIFACT_DIR, normalize=True)
save_classification_report(y_test, y_pred, ARTIFACT_DIR)
show_misclassifications(x_test, y_test, y_pred, limit=25, out_dir=ARTIFACT_DIR)
save_label_map(ARTIFACT_DIR)
save_summary(test_acc, test_loss, len(history.history["loss"])), ARTIFACT_DIR)

#####
# Function: inference_grid
# Inputs: n_samples (int), seed (int)
# Outputs: Saved inference grid image
# Description: Loads saved model, predicts on random test samples, saves predictions grid.
# Author: Piyush Manohar Khairstar
# Date: 15/08/2025
#####

def inference_grid(n_samples=9, seed=7):
    if not os.path.exists(BEST_MODEL):
        print(f"Could not find {BEST_MODEL}. Train first with --train.")
        return
    (_, _), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
    x_test = (x_test.astype("float32") / 255.0)[..., None]
    model = keras.models.load_model(BEST_MODEL)

    rng = np.random.default_rng(seed)
    idx = rng.choice(len(x_test), size=n_samples, replace=False)
    imgs = x_test[idx]
    labs = y_test[idx]
    preds = model.predict(imgs, verbose=0).argmax(axis=1)

    cols = int(np.ceil(np.sqrt(n_samples)))
    rows = int(np.ceil(n_samples / cols))
    plt.figure(figsize=(2.8 * cols, 2.8 * rows))
    for i in range(n_samples):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(imgs[i].squeeze(), cmap="gray")
        plt.title(f"P:{FASHION_CLASSES[preds[i]]}\nT:{FASHION_CLASSES[labs[i]]}", fontsize=9)
        plt.axis("off")
    plt.tight_layout()
    ensure_dir(ARTIFACT_DIR)
    out_path = os.path.join(ARTIFACT_DIR, "inference_grid.png")
    plt.savefig(out_path)
    plt.close()
    print("Saved:", out_path)

#####
# Function: run_baselines
# Inputs: None
# Outputs: Prints accuracy of LogReg, LinearSVC, RandomForest
#####
  
```

```

# Description: Runs classical ML baselines on flattened Fashion-MNIST.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####
def run_baselines():
    x_train, y_train, x_test, y_test = load_flattened()
    logreg = Pipeline([
        ("scaler", StandardScaler(with_mean=True)),
        ("clf", LogisticRegression(max_iter=200, n_jobs=-1))
    ])
    logreg.fit(x_train, y_train)
    print("LogReg acc:", accuracy_score(y_test, logreg.predict(x_test)))

    svm = Pipeline([
        ("scaler", StandardScaler(with_mean=True)),
        ("clf", LinearSVC(C=1.0, dual=True, max_iter=5000))
    ])
    svm.fit(x_train, y_train)
    print("LinearSVC acc:", accuracy_score(y_test, svm.predict(x_test)))

    rf = RandomForestClassifier(n_estimators=200, max_depth=None, n_jobs=-1, random_state=SEED)
    rf.fit(x_train, y_train)
    print("RandomForest acc:", accuracy_score(y_test, rf.predict(x_test)))

#####
# Function: parse_args
# Inputs: None
# Outputs: Parsed CLI arguments
# Description: Defines command-line arguments for training, inference, baselines.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

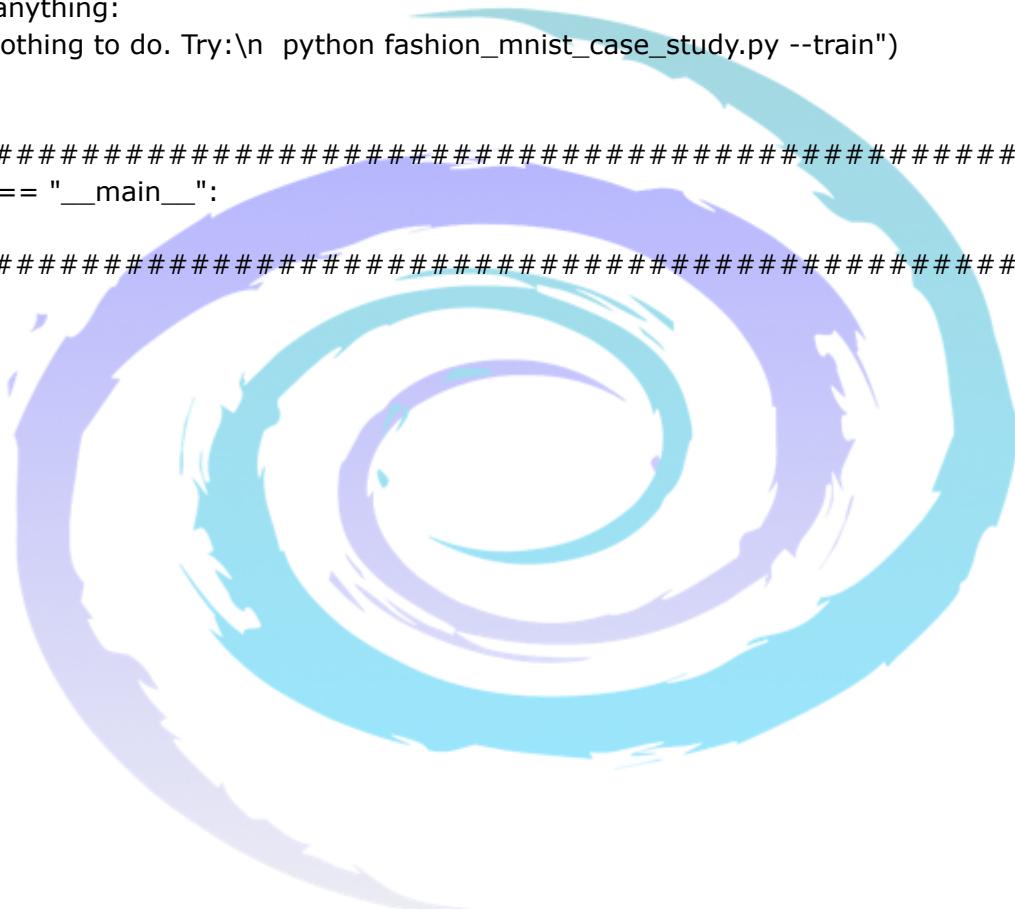
def parse_args():
    p = argparse.ArgumentParser(description="Fashion-MNIST Case Study")
    p.add_argument("--train", action="store_true", help="Train the CNN and save artifacts.")
    p.add_argument("--infer", action="store_true", help="Generate an inference grid using saved model.")
    p.add_argument("--samples", type=int, default=9, help="Number of samples for inference grid.")
    p.add_argument("--baselines", action="store_true", help="Run classical ML baselines.")
    p.add_argument("--epochs", type=int, default=15, help="Epochs for CNN training.")
    p.add_argument("--batch", type=int, default=128, help="Batch size for CNN training.")
    p.add_argument("--lr", type=float, default=1e-3, help="Learning rate for Adam.")
    return p.parse_args()

#####

# Function: main
# Inputs: None
# Outputs: None
# Description: Entry point. Parses CLI args, runs train/infer/baselines accordingly.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####
  
```

```
def main():
    set_seed(SEED)
    ensure_dir(ARTIFACT_DIR)
    args = parse_args()
    didAnything = False
    if args.train:
        train_and_evaluate(batch_size=args.batch, epochs=args.epochs, lr=args.lr)
        didAnything = True
    if args.infer:
        inference_grid(n_samples=args.samples)
        didAnything = True
    if args.baselines:
        run_baselines()
        didAnything = True
    if not didAnything:
        print("Nothing to do. Try:\n python fashion_mnist_case_study.py --train")
```

```
#####
if __name__ == "__main__":
    main()
#####
```



The above project contains :

1. Utility Functions

- set_seed, ensure_dir, plot_training_curves, plot_confusion_matrix, save_classification_report, show_misclassifications, save_label_map, save_summary

2. Data Loaders

- load_data → loads + normalizes dataset with train/val/test split
- load_flattened → prepares flattened arrays for classical ML baselines

3. Model

- build_cnn → defines and compiles a CNN with augmentation

4. Training & Evaluation

- train_and_evaluate → trains CNN, evaluates on test set, saves model + artifacts

5. Inference

- inference_grid → loads saved model and shows predictions for random samples

6. Baselines

- run_baselines → Logistic Regression, Linear SVM, Random Forest

7. CLI

- parse_args → command line flags (--train, --infer, --baselines)
- main → entry point to run training, inference, or baselines

Please follow below steps :

1. Save the code

Marvellous_fashion_mnist_case_study.py

2. Install dependencies

Open a terminal and create a virtual environment (recommended):

python -m venv venv

Activate it:

- **Windows:**
venv\Scripts\activate
- **Mac/Linux:**
source venv/bin/activate

Now install the required libraries:

pip install tensorflow scikit-learn matplotlib numpy pandas tqdm

3. Train the CNN

python Marvellous_fashion_mnist_case_study.py --train

This will:

- Train the CNN model
- Save the best model in artifacts/fashion_cnn.h5
- Save training curves (acc_curve.png, loss_curve.png)
- Save confusion matrix, misclassifications, classification report, etc.

All outputs will be in the artifacts/ folder.

4. Run Inference (predict on random test images)

After training, you can generate an inference grid:

python Marvellousfashion_mnist_case_study.py --infer --samples 9

This will create artifacts/inference_grid.png showing predictions vs true labels.

5. Try classical baselines (Logistic Regression, SVM, Random Forest)

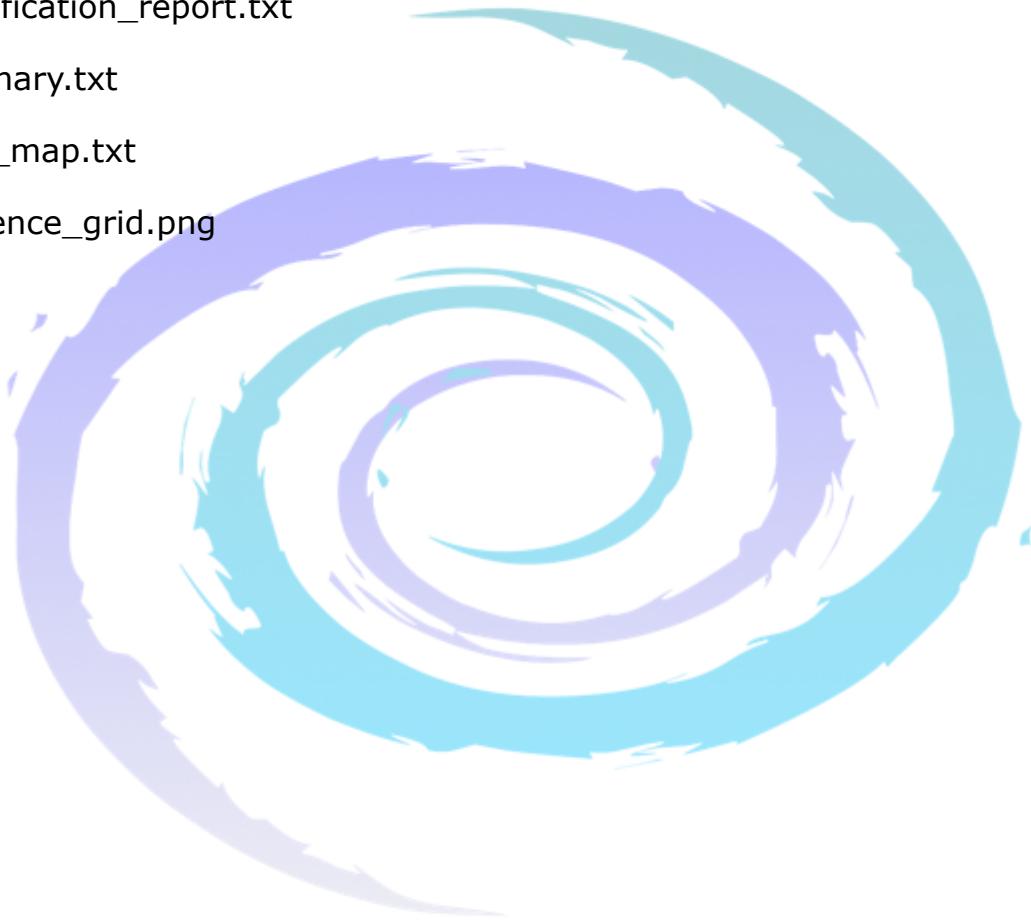
python Marvellousfashion_mnist_case_study.py --baselines

This compares simple ML methods vs CNN.

Outputs

All results go into the artifacts/ folder:

- fashion_cnn.h5 → best model
- fashion_cnn_final.h5 → final model
- acc_curve.png, loss_curve.png
- confusion_matrix.png
- misclassifications.png
- classification_report.txt
- summary.txt
- label_map.txt
- inference_grid.png



Python Task Scheduler

Consider below python application which schedule task after Minute, Hour as well as on specific Day or specific time.

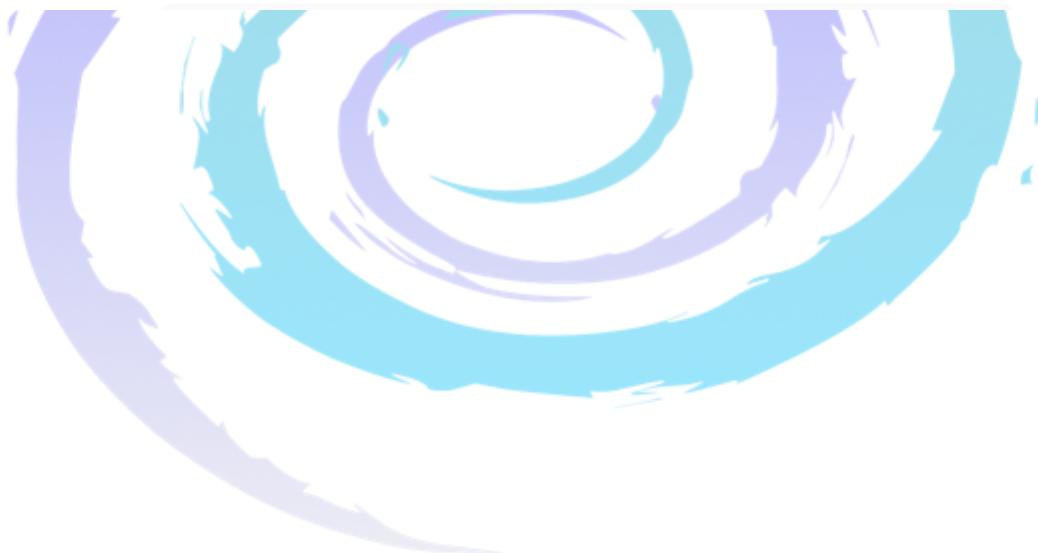
```

1 import schedule
2 import time
3 import datetime
4
5 def fun_Minute():
6     print("Current time is ")
7     print(datetime.datetime.now())
8     print("Scheduler executed after Minute")
9
10 def fun_Hour():
11     print("Current time is ")
12     print(datetime.datetime.now())
13     print("Scheduler executed after Hour")
14
15 def fun_Day():
16     print("Current time is ")
17     print(datetime.datetime.now())
18     print("Scheduler executed after Day")
19
20 def fun_Afternoon():
21     print("Current time is ")
22     print(datetime.datetime.now())
23     print("Scheduler executed at 12")
24
25 def main():
26     print("Marvellous Infosystems : Python Automation & Machine Learning")
27
28     print("Python Job Scheular")
29     print(datetime.datetime.now())
30
31     schedule.every(1).minutes.do(fun_Minute)
32
33     schedule.every().hour.do(fun_Hour)
34
35     schedule.every().day.at("00:00").do(fun_Afternoon)
36
37     schedule.every().sunday.do(fun_Day)
38
39     schedule.every().saturday.at("18:30").do(fun_Day)
40
41     while True:
42         schedule.run_pending()
43         time.sleep(1)
44
45 if __name__ == "__main__":
46     main()
47

```

Output of above script

```
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ python sch.py
Marvellous Infosystems : Python Automation & Machine Learning
Python Job Scheduler
2019-04-20 13:39:17.501788
Current time is
2019-04-20 13:40:17.653276
Scheduler executed after Minute
Current time is
2019-04-20 13:41:17.782483
Scheduler executed after Minute
Current time is
2019-04-20 13:42:17.899424
Scheduler executed after Minute
Current time is
2019-04-20 13:43:18.072133
Scheduler executed after Minute
```



BERT vs GPT as a transformer based model

BERT (Bidirectional Encoder Representations from Transformers)

- **Created by Google (2018).**
- It is a **transformer-based encoder model**.
- **Key idea:** Reads text **bidirectionally** (from left-to-right and right-to-left at the same time).
- **Goal:** Understand the *meaning of words in context*. For example, the word "bank" could mean a riverbank or a financial bank — BERT figures that out using surrounding words.
- **Uses:**
 - Text classification (spam detection, sentiment analysis)
 - Question answering
 - Named entity recognition
 - Search engines (Google uses BERT to improve search results)

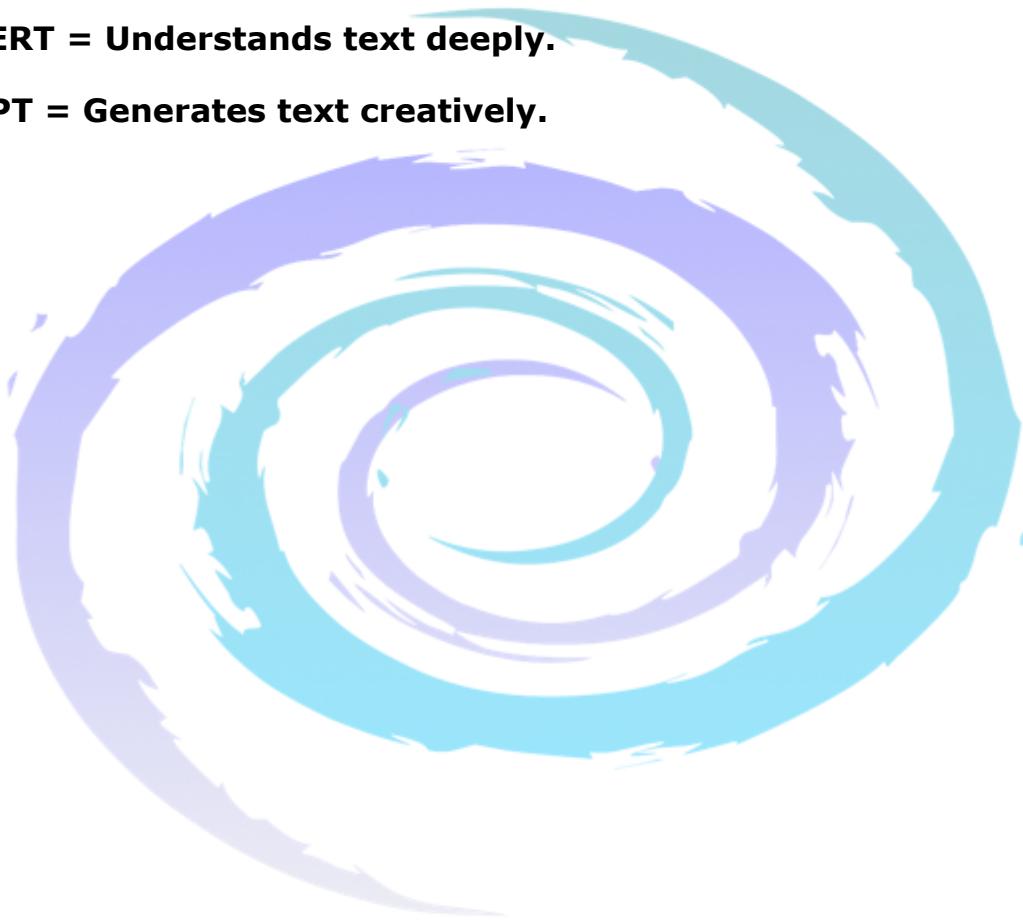
GPT (Generative Pre-trained Transformer)

- **Created by OpenAI (first in 2018, improved versions like GPT-2, GPT-3, GPT-4, GPT-5).**
- It is a **transformer-based decoder model**.
- **Key idea:** Predicts the *next word* in a sentence, one word at a time.
- It is **generative**, meaning it creates new text (stories, code, essays, conversations, etc.).
- **Uses:**
 - Chatbots
 - Text completion
 - Code generation
 - Summarization, translation, content creation

Difference Between BERT and GPT

Feature	BERT	GPT
Architecture	Encoder (Bidirectional)	Decoder (Unidirectional, left → right)
Main Focus	Understanding language (context)	Generating language (text)
Training Task	Masked Language Modeling (predict missing words)	Next Word Prediction (predict next word)
Typical Use Cases	Classification, search, question answering	Chat, text generation, summarization

- **BERT = Understands text deeply.**
- **GPT = Generates text creatively.**



Bias, Variance, Overfitting & Underfitting

Bias

- Meaning:** Error caused by a model making **oversimplified assumptions**, ignoring important details.
- Effect:** Predictions are **consistently wrong** (model under-learns).
- Example:**
Predicting every student's exam score as **50 marks**, no matter how much they studied.

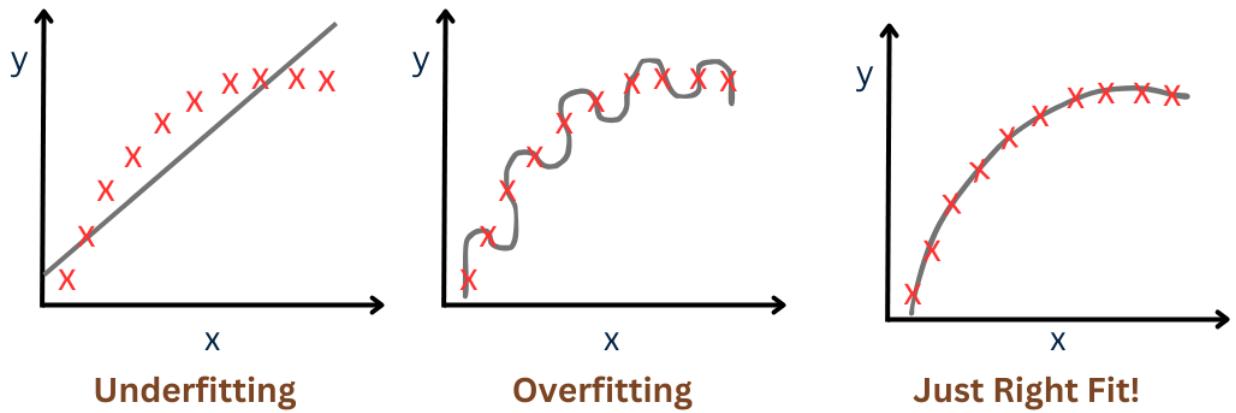
High Bias → Underfitting.

Variance

- Meaning:** Error caused by a model being **too sensitive to training data**, learning **noise and random patterns** instead of real trends.
- Effect:** Predictions are **unstable** – they change a lot for new data.
- Example:**
Model remembers every student's data exactly but **gives wrong results for new students**.

High Variance → Overfitting.

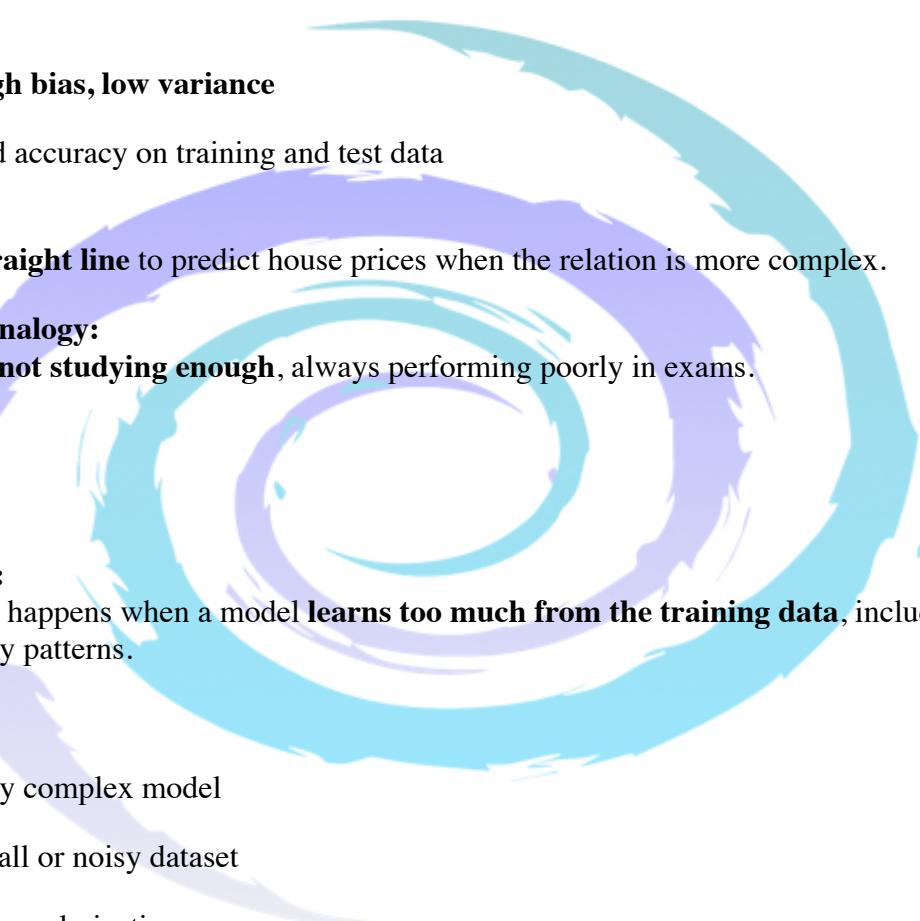
Model Underfitting, Overfitting and Right Fit



Bias	High	Low	Low
Variance	Low	High	Low

Underfitting

- **Definition:**
Underfitting happens when a model is **too simple** to learn the data properly.
- **Cause:**
 - Few features used
 - Very basic algorithm
 - Less training
- **Effect:**
 - **High bias, low variance**
 - Bad accuracy on training and test data
- **Example:**
Using a **straight line** to predict house prices when the relation is more complex.
- **Real-life analogy:**
A student **not studying enough**, always performing poorly in exams.



Overfitting

- **Definition:**
Overfitting happens when a model **learns too much from the training data**, including noise and unnecessary patterns.
- **Cause:**
 - Very complex model
 - Small or noisy dataset
 - No regularization
- **Effect:**
 - **Low bias, high variance**
 - Perfect accuracy on training data but poor on test data
- **Example:**
A **zigzag curve** that passes through every training point but fails on new data.
- **Real-life analogy:**
A student **memorizing answers word-for-word**, failing to answer new questions.

Best Fit Model

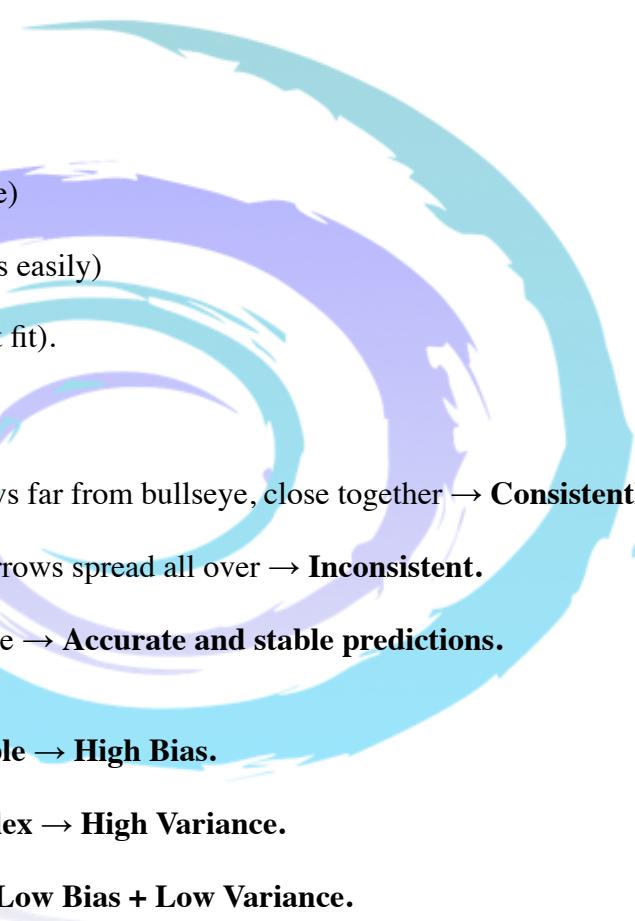
- **Goal:** Achieve **low bias and low variance**.
- **Effect:** Model learns the **right patterns**, ignoring noise, and works well on unseen data.
- **Example:**
A **balanced study approach** – understanding concepts, not just memorizing.

Bias-Variance Trade-off

- **Rule:**

Total Error = Bias² + Variance + Noise

- Increasing model complexity:
 - Bias ↓ (model learns more)
 - Variance ↑ (model overfits easily)
- **Ideal Point:** Balanced error (best fit).



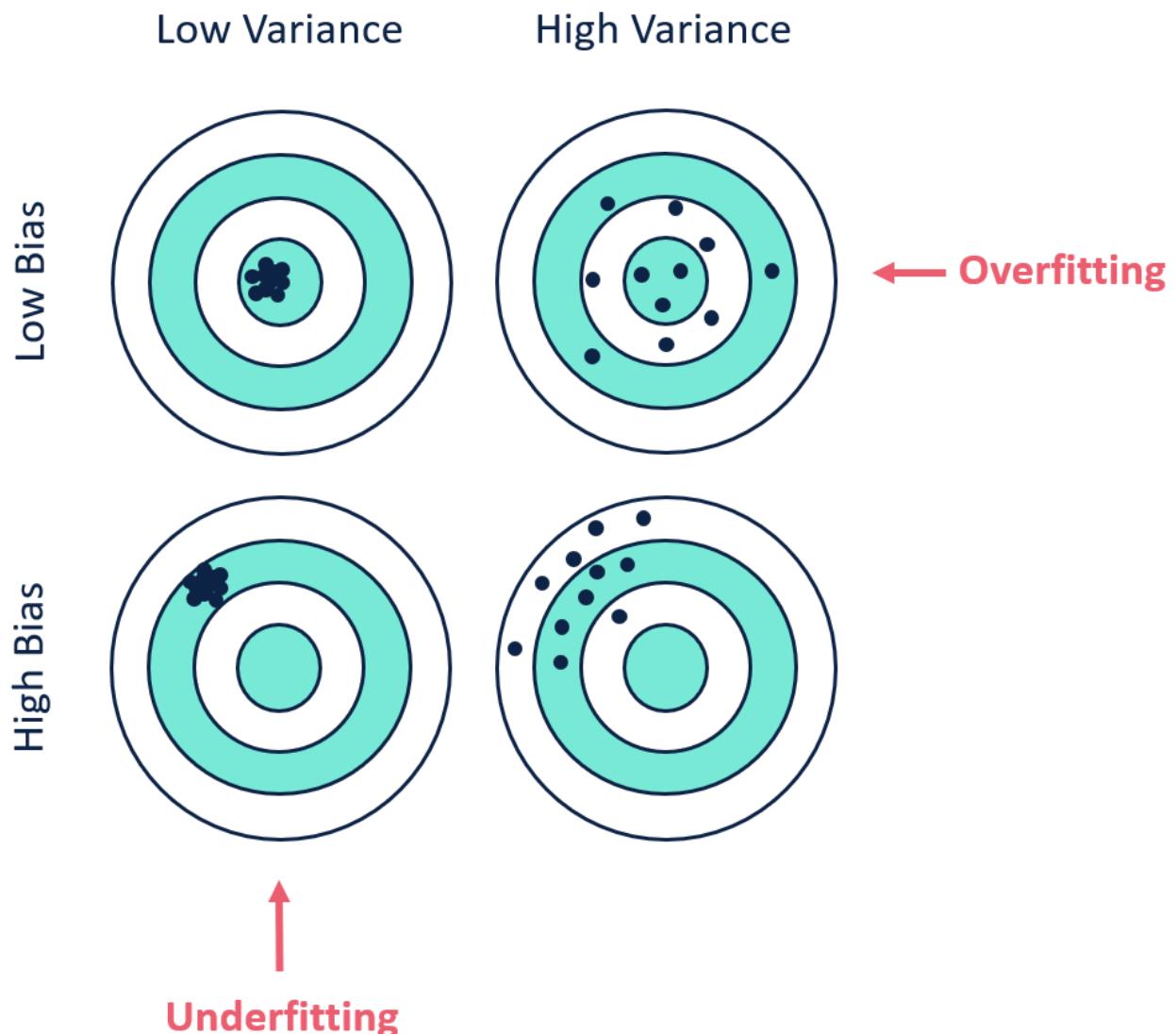
Archery Target Analogy

- **High Bias (Underfitting):** Arrows far from bullseye, close together → **Consistently wrong**.
- **High Variance (Overfitting):** Arrows spread all over → **Inconsistent**.
- **Best Fit:** Arrows close to bullseye → **Accurate and stable predictions**.
- **Underfitting** → **Model too simple** → **High Bias**.
- **Overfitting** → **Model too complex** → **High Variance**.
- **Best Fit** → **Balanced model** → **Low Bias + Low Variance**.

Definition of Noise

Noise = **Errors or variations in data** that occur due to:

- Measurement mistakes
- Random fluctuations
- Missing or incorrect data entries
- Factors that are **not part of the actual pattern** we are trying to learn.



Characteristics of Noise

- Noise is **unwanted information** in the dataset.
- It **cannot be predicted** by the model.
- It **adds complexity** and can cause **overfitting** if the model tries to learn it.
- Even the **best model cannot fully eliminate noise** (called **irreducible error**).

Examples of Noise

Example 1: House Price Prediction

- Input features: Size of house, location, number of rooms.
- **Noise:**
 - Typo in data: Price entered as **10,000,000 instead of 1,000,000**.
 - A sudden one-time **discount or premium** due to seller urgency.

Example 2: Student Exam Scores

- Inputs: Hours studied, practice tests taken.
- **Noise:**
 - A student **feels unwell on exam day** → unusually low score.
 - Random guessing of answers → unpredictable results.

Training Error and Testing Error

- **Training Error:**
The error (difference between predicted and actual values) when the model is tested on **data it was trained on**.
 - Shows **how well the model learned the training data**.
- **Testing Error (Generalization Error):**
The error when the model is tested on **new, unseen data**.
 - Shows **how well the model generalizes** to unknown data.

Role of Bias and Variance

- **Bias:**
 - Error due to **wrong assumptions** (model too simple).
 - Leads to **high training and testing error** (underfitting).
- **Variance:**
 - Error due to **over-sensitivity** to training data (model too complex).
 - Leads to **low training error but high testing error** (overfitting).
- **Noise:**
 - Random errors in data that no model can fix.

Training vs Testing Error Behavior

Model Complexity	Bias	Variance	Training Error	Testing Error
Underfitting (Too Simple)	High	Low	High	High
Overfitting (Too Complex)	Low	High	Very Low	High
Best Fit (Balanced)	Low	Low	Low	Low

Example: Student Exam Score Prediction

- **Underfitting:**

Model says every student scores **50 marks**, ignoring all features.

- Training error = **High**
- Testing error = **High**
- Reason = **High Bias**

- **Overfitting:**

Model memorizes every student's exact data.

- Training error = **Almost 0**
- Testing error = **High** (fails on new students)
- Reason = **High Variance**

- **Best Fit:**

Model learns a general rule: "*More study hours = Higher score*".

- Training error = **Low**
- Testing error = **Low**
- Reason = **Balanced Bias and Variance**

Build Process of Python

PVM (Python Virtual Machine)

- PVM is the **runtime engine** of Python.
- It reads **.pyc (compiled bytecode)** files and executes them.
- Provides **platform independence**.
- Handles memory allocation, garbage collection, and exception handling.

Build Process of Python Code

1. **Source Code (.py):**
 - Developer writes Python code in a **.py** file.
2. **Lexical Analysis:**
 - The Python interpreter scans the code and breaks it into tokens.
3. **Parsing & AST Generation:**
 - Tokens are parsed and transformed into an **Abstract Syntax Tree (AST)**.
 - Checks for syntax errors during this step.
4. **Compilation to Bytecode:**
 - AST is compiled into Python **bytecode (.pyc files)**.
 - Bytecode is platform-independent and stored in **__pycache__** folder.
5. **Execution by PVM:**
 - The Python Virtual Machine reads the bytecode and executes it line-by-line.
 - Handles runtime actions such as memory management, method calls, and control flow.

Complete Python Toolchain (Execution Flow)

1. **Editor/IDE:**
 - Tools like VS Code, PyCharm, Jupyter Notebooks.
 - Used to write and edit **.py** files.
2. **Tokenizer / Lexer:**

- Converts code into smaller tokens (identifiers, keywords, literals).

3. Parser:

- Validates the syntax and forms the AST.

4. Bytecode Compiler:

- Converts AST into bytecode.
- Bytecode is a low-level set of instructions understandable by the PVM.

5. Bytecode Storage:

- Stored as `.pyc` files in the `__pycache__` directory.

6. Python Virtual Machine (PVM):

- Executes bytecode line-by-line.
- Manages memory, exceptions, object references, and more.

7. Standard Libraries:

- Built-in modules like `os`, `math`, `random`, etc. are loaded if used.

8. Third-Party Libraries:

- Installed using `pip`. Stored in site-packages directory.
- Imported and linked at runtime if present in the script.

Python Career Streams & Industry Opportunities

Data Science

- **Core stack:** numpy, pandas, matplotlib/plotly, seaborn, scikit-learn, statsmodels, Jupyter, mlflow (tracking), great_expectations (data quality).
- **Concepts:** EDA, feature engineering, train/validation/test splits, cross-validation, metrics (RMSE/MAE, accuracy/F1/AUC), leakage avoidance, bias/variance, model explainability (SHAP).
- **Roles:** Data Analyst, Data Scientist.
- **Resume keywords:** EDA, feature engineering, cross-validation, SHAP, MLflow, reproducible pipeline.

Web Backends & APIs

- **Core stack:** FastAPI/Django (ORM: SQLAlchemy/Django ORM), PostgreSQL/MySQL, Redis, JWT/OAuth2, Celery/RQ, Docker.
- **Concepts:** REST, async I/O, pagination, auth, caching, background jobs, testing (pytest), CI/CD.
- **Roles:** Python/Backend Developer, API Engineer.
- **Resume keywords:** FastAPI/Django, ORM, JWT, Redis caching, Celery, Docker, CI/CD.

Data Engineering & Pipelines

- **Core stack:** Airflow/Prefect (orchestrate DAGs), PySpark/Dask (big data), Kafka, dbt, Parquet/Delta, S3/GCS, DuckDB.
- **Concepts:** Batch vs streaming, partitioning, schema evolution, lineage, SLAs, data tests, CDC.
- **Roles:** Data Engineer, ETL Developer.
- **Resume keywords:** Airflow DAGs, PySpark, Kafka, Parquet, dbt models, data quality.

DevOps/SRE & Cloud Automation

- **Core stack:** AWS Boto3/Azure/GCP SDKs, Terraform/Pulumi (Python), Kubernetes Python client, GitHub Actions, Prometheus/Grafana.
- **Concepts:** IaC, blue-green/canary, observability (logs/metrics/traces), drift detection, cost controls.
- **Roles:** DevOps Engineer, SRE, Cloud Engineer.
- **Resume keywords:** IaC (Terraform/Pulumi), Kubernetes client, CI/CD, Boto3, observability.

Cybersecurity & DFIR

- **Core stack:** scapy, python-nmap, cryptography, yara-python, volatility3, paramiko.
- **Concepts:** Network scanning, log parsing, IOC rules, secure key management, forensics triage.
- **Roles:** Security Analyst, DFIR Engineer (add certs: Sec+, eJPT, etc.).
- **Resume keywords:** IOC detection, YARA, Volatility, secure crypto, SSH automation.

IoT/Embedded & Edge

- **Core stack:** MicroPython/CircuitPython (ESP32), Raspberry Pi (gpiozero), MQTT (paho-mqtt), OpenCV, TF-Lite.
- **Concepts:** Sensor I/O, pub/sub messaging, edge inference, reliability & watchdogs, OTA updates (basic).
- **Roles:** IoT/Embedded Engineer, Robotics (entry).
- **Resume keywords:** MicroPython, MQTT, Raspberry Pi, edge inference, OpenCV.

Components of Artificial Neural Network

Neuron / Node / Perceptron

- **Definition:**
The smallest unit of an Artificial Neural Network, inspired by biological brain neurons.
 - **Explanation:**
Each neuron takes inputs, multiplies them with weights, adds a bias, applies an activation function, and produces an output.
 - **Formula:**
- $$y = f(\sum(x_i \cdot w_i) + b)$$
- where:
 x_i = input values
 w_i = weights
 b = bias
 f = activation function
 - **Example :**
Suppose a student's marks in Maths = 80, Science = 70. The network predicts "Pass/Fail".
 - Input = [80, 70]
 - Weights = [0.5, 0.4], Bias = 10
 - Output = $(80 \times 0.5 + 70 \times 0.4 + 10) = 40 + 28 + 10 = 78 \rightarrow \text{Passed.}$

Input Layer

- **Definition:**
The first layer of the network where raw data enters.
- **Explanation:**
Each node in this layer represents one feature from the dataset.
- **Example:**
In handwriting recognition (MNIST), each image (28×28 pixels) is flattened into **784 inputs**.

Hidden Layer

- **Definition:**
Layers between input and output layers where transformations occur.
- **Explanation:**
Hidden layers learn complex patterns by applying weights, bias, and nonlinear functions.
- **Example:**
In image classification, hidden layers detect **edges, shapes, and higher-level features** step by step.

Output Layer

- **Definition:**
The final layer that produces the network's predictions.
- **Explanation:**
Its size depends on the task:
 - Binary classification → 1 node (e.g., Yes/No).
 - Multi-class classification → N nodes (softmax gives probabilities).
- **Example:**
Predicting digits 0–9 → Output layer with **10 neurons**.

Weights

- **Definition:**
Numerical values representing the strength of connections between neurons.
- **Explanation:**
 - High weight → strong influence of input on output.
 - Low/zero weight → weak/no influence.
- **Example:**
In a spam filter, the word "free" may get a **high weight** (strong spam indicator).

Bias

- **Definition:**
An additional parameter that allows shifting the activation function.
- **Explanation:**
It helps the model fit data better, even if all inputs are zero.
- **Example:**
In linear regression y

$$y = mx + c$$

$y=mx+c$, the **bias** is like the intercept c

Confusion Matrix

N = 165	Predicted NO	Predicted YES	
Actual NO	TN = 50	FP = 10	60
Actual YES	FN = 5	TP = 100	105
	55	110	

- true positives (TP): These are cases in which we predicted yes (they have the disease), and they do have the disease.
- true negatives (TN): We predicted no, and they don't have the disease.
- false positives (FP): We predicted yes, but they don't actually have the disease. (Also known as a "Type I error.")
- false negatives (FN): We predicted no, but they actually do have the disease. (Also known as a "Type II error.")

- Accuracy: How often is the classifier correct?
 - $(TP+TN)/\text{total} = (100+50)/165 = 0.91$
- Misclassification Rate: How often is it wrong?
 - $(FP+FN)/\text{total} = (10+5)/165 = 0.09$
 - equivalent to 1 minus Accuracy
 - also known as "Error Rate"
- True Positive Rate: When it's actually yes, how often does it predict yes? (Recall)
 - $TP/\text{actual yes} = 100/105 = 0.95$
 - also known as "Sensitivity" or "Recall"
- False Positive Rate: When it's actually no, how often does it predict yes?
 - $FP/\text{actual no} = 10/60 = 0.17$
- True Negative Rate: When it's actually no, how often does it predict no?
 - $TN/\text{actual no} = 50/60 = 0.83$
 - equivalent to 1 minus False Positive Rate
 - also known as "Specificity"
- Precision: When it predicts yes, how often is it correct?
 - $TP/\text{predicted yes} = 100/110 = 0.91$
- Prevalence: How often does the yes condition actually occur in our sample?
 - $\text{actual yes}/\text{total} = 105/165 = 0.64$

F1 Score

The F-score, also called the F1-score, is a measure of a model's accuracy on a dataset. It is used to evaluate binary classification systems, which classify examples into 'positive' or 'negative'.

F1 is calculated as follows:

$$F_1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

where:

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$

In "macro" F1 a separate F1 score is calculated for each `species` value and then averaged.

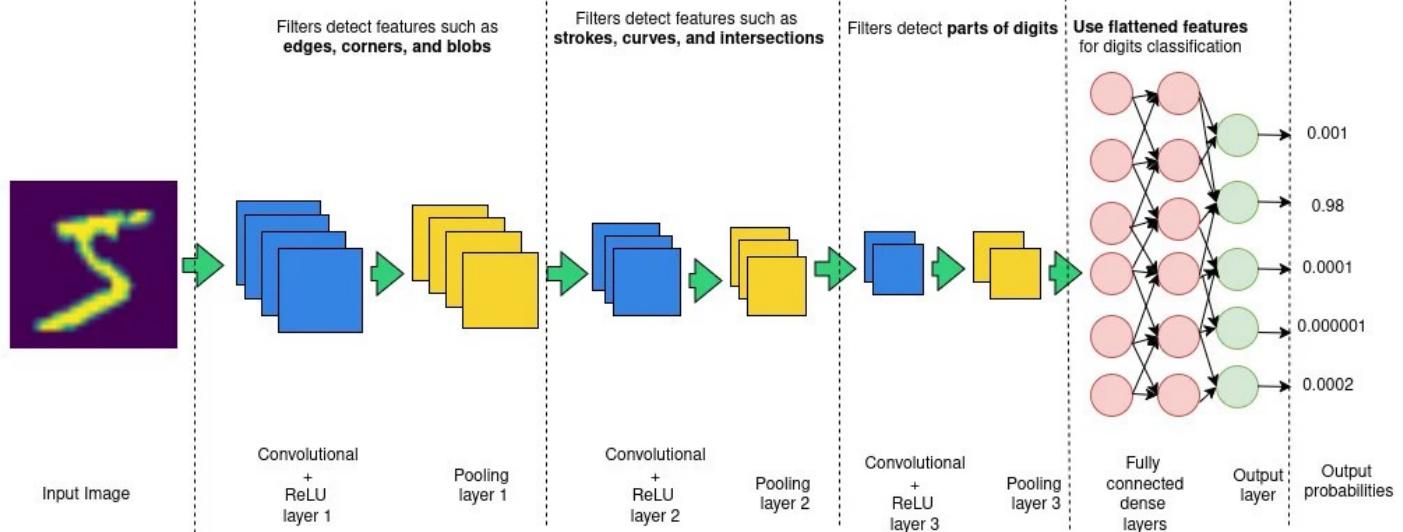
Convolutional Neural Network (CNN)

- A **Convolutional Neural Network (CNN)** is a type of deep learning model designed for processing **grid-like data** such as images.
- Instead of fully connecting every neuron to every pixel (like in Feedforward Neural Networks), CNNs use **convolutional layers** that apply small filters (kernels) across the image to detect **patterns (edges, shapes, textures)**.

Why CNNs :

- Images are large (e.g., $224 \times 224 \times 3 = 150\text{k+}$ values). Fully connected layers would be too big.
- CNNs exploit:
 - **Local connectivity** → look at small regions.
 - **Weight sharing** → same filter applied everywhere.
 - **Hierarchical learning** → low-level features (edges) → high-level features (objects).

Key Components of CNN



Convolution Layer

- Applies filters (small matrices like $3 \times 3, 5 \times 5$) across the image.
- Each filter extracts specific features (edges, corners, textures).

👉 Example filter (edge detector):

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Pooling Layer

- Reduces the size of feature maps.
- Max Pooling** (most common): takes the maximum value in a small window (e.g., 2×2).
- Helps with **dimensionality reduction** and prevents overfitting.

Output Layer

- Usually uses **Softmax** for classification.
- Example: For 10 classes → softmax gives 10 probabilities.

Input Image

- Images are stored as **pixel grids**.
- Grayscale → 2D matrix of values (0–255).
- RGB → 3D matrix (Height × Width × 3 channels).
Example: 224×224×3 for a color image.

Convolution

- The main operation in CNN.
- A **filter/kernel** (small matrix, e.g., 3×3) slides over the image.
- Computes a **dot product** → creates a **feature map**.
- Helps detect features like **edges, corners, textures**.

Formula:

$$\text{Feature Map} = \sum (\text{Input} \times \text{Kernel})$$

Kernel / Filter

- Small matrix (e.g., 3×3, 5×5) with weights.

- Shared across the image → fewer parameters than fully connected networks.
- Each kernel detects a **specific feature**.

Feature Map

- Output of a convolution operation.
- Shows **where the filter detected its pattern** in the image.

Stride

- Number of steps the filter moves when sliding.
- **Stride = 1** → moves 1 pixel at a time (large feature map).
- **Stride = 2** → moves 2 pixels at a time (smaller feature map).

Padding

- Adding extra pixels around the border of the image.
- Prevents shrinking of output after convolution.
- **Same padding** → output same size as input.
- **Valid padding** → no padding; output shrinks.

Channels

- **Depth** of the input image.
- Example: RGB = 3 channels.
- CNN learns **multiple filters per layer** (e.g., 32 filters → 32 feature maps).

Activation Function (ReLU)

- Introduces non-linearity.
- Most common: **ReLU** → replaces negative values with 0.

$$f(x) = \max(0, x)$$

Pooling

- Reduces size of feature maps → less computation, avoids overfitting.
- **Max Pooling:** takes max value in a region.

- **Average Pooling:** takes average.

Example (2×2 Max Pooling):

[1 3]	6	
[5 6]	\rightarrow	(max value)

Flattening

- Converts 2D feature maps into a 1D vector.
- Required before connecting to fully connected (dense) layers.

Fully Connected Layer (Dense)

- Works like a regular neural network.
- Combines extracted features → predicts final class.

Softmax Layer

- Final layer for classification tasks.
- Converts raw scores into probabilities (sums to 1).
- Example: Cat → 0.85, Dog → 0.10, Car → 0.05.

Epoch

- One full pass of training data through the network.

Batch Size

- Number of samples processed before updating model weights.

Loss Function

- Measures error between predicted and actual labels.
- Common for CNN: **Cross-Entropy Loss**.

Optimizer

- Algorithm that updates weights to minimize loss.
- Examples: **SGD, Adam, RMSprop**.

Backpropagation

- Process of calculating gradients and updating weights using **chain rule**.
- CNNs use **backpropagation through convolution and pooling layers**.

Dropout

- Regularization technique → randomly drops neurons during training.
- Prevents overfitting.

Data Augmentation

- Artificially increasing dataset by applying transformations:
 - Rotation, flipping, zooming, cropping.
- Helps CNN generalize better.

Transfer Learning

- Using a pre-trained CNN (like VGG16, ResNet, MobileNet) and fine-tuning it for a new task.
- Saves training time and resources.

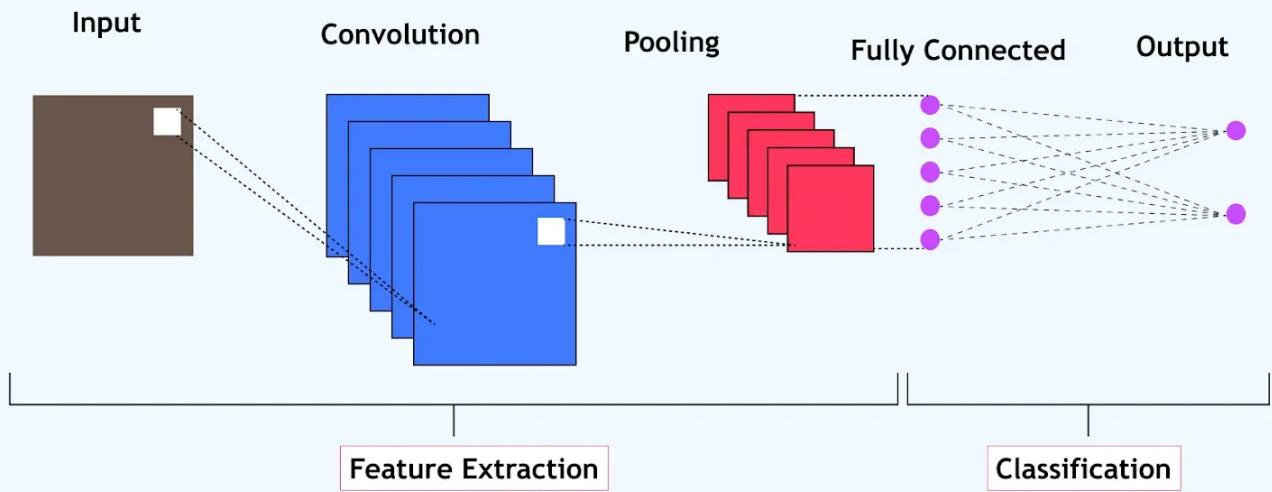
CNN Architectures

- Famous CNN models:
 - **LeNet-5** → First CNN (1998).
 - **AlexNet** → Won ImageNet 2012.
 - **VGGNet** → Deep, simple 3×3 filters.
 - **ResNet** → Introduced skip connections.
 - **MobileNet** → Lightweight for mobile/edge devices.

CNN Architecture (Typical Flow)

Input Image → Convolution → ReLU → Pooling → Convolution → ReLU → Pooling → Fully Connected → Output

The Architecture of Convolutional Neural Networks



Data Science

Data Science

Data Science is the study of extracting **useful information** and **insights** from data using **mathematics, statistics, computer programming**, and domain knowledge.

Fields of Data Science

Data Science connects many different fields:

- **Statistics** – for analyzing numbers.
- **Machine Learning** – for making predictions.
- **Data Visualization** – for making charts and graphs.
- **Big Data** – for handling huge amounts of data.
- **Data Engineering** – for building systems to collect and store data.
- **Domain Knowledge** – understanding the field like finance, health, sports, etc.

Data :

Data means **raw facts or figures**. Data is meaningless unless we understand and process it.

Example:

- "21" is data. (Is it age, temperature, or score?)

Information :

When data is **organized and given meaning**, it becomes **information**.

Example:

- "Amit is 21 years old" — Now it's useful and meaningful = **information**.

Types of Data

Type	Description	Example
Structured	Organized in rows & columns	Excel sheet, databases
Unstructured	No fixed format	Images, videos, emails
Semi-structured	Some structure, not full table	JSON, XML files

Artificial Intelligence (AI) :

AI is the ability of a machine to **think, learn, and act** like humans.

Example:

- Alexa answering your questions.
- Google Maps giving directions.

Machine Learning (ML) :

Machine Learning is a part of AI that teaches machines to **learn from data and improve automatically without being told what to do every time**.

Simple example:

- If we show 1000 pictures of cats and dogs, a machine can **learn** to identify them on its own.

Dataset :

A dataset is a **collection of related data** used for analysis or training a machine.

Example:

Name	Age	Gender	Result
Ram	21	Male	Pass
Laxmi	23	Female	Pass

This table is a **dataset**.

Types of Datasets in ML

- **Training Dataset** – used to teach the machine.
- **Testing Dataset** – used to check the accuracy of the model.

Features and Labels

- **Features** – Input data used to predict something.
- **Label** – The output or result we want to predict.

Example:

Height (cm)	Weight (kg)	Gender
170	60	Male

- Features = Height, Weight
- Label = Gender

Types of Machine Learning

1. Supervised Learning

- You give input **and** the correct output (label).
- Machine learns from examples.

Example: Predicting house prices, classifying emails as spam.

2. Unsupervised Learning

- You give only input, **no correct answers**.
- Machine finds patterns.

Example: Grouping similar customers for marketing.

3. Reinforcement Learning

- Machine learns by **trial and error** using rewards and punishment.

Example: Self-driving cars

Data Visualization using Seaborn & Matplotlib

In the world of **data science and machine learning**, understanding the structure, distribution, and relationships within data is extremely important.

Raw data in tables or spreadsheets can often be overwhelming and difficult to interpret.

This is where **data visualization** comes in — it allows us to **see patterns, spot outliers, and understand trends** clearly and effectively.

Two of the most powerful Python libraries used for creating such visualizations are:

- **Matplotlib** – the foundation library for static, animated, and interactive plots in Python.
- **Seaborn** – built on top of Matplotlib, it provides a high-level interface for creating attractive and informative statistical graphics.

1. Histogram using Matplotlib

File: Histogram.py

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

def main():
    df = pd.read_csv("iris.csv")

    plt.hist(df['sepal.length'], bins=10, color="skyblue",
edgecolor='black')

    plt.xlabel("Sepal Length")
    plt.ylabel("Frequency")
    plt.title("Marvellous Histogram for IRIS")

    plt.show()

if __name__ == "__main__":
    main()
```

✓ Explanation:

- **Histogram** is used to visualize the **distribution** of a numerical feature.
- **plt.hist()** plots a histogram.
- **bins=10**: Divides data into 10 ranges.
- **color** and **edgecolor**: Used to enhance visibility.
- This chart shows how frequently each sepal length appears in the dataset.

2. Boxplot using Seaborn

File: boxplot.py

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

def main():
    df = pd.read_csv("iris.csv")

    sns.boxplot(x="variety", y="petal.length", data=df)

    plt.title("Marvellous Boxplot for Petal length by variety")
    plt.show()

if __name__ == "__main__":
    main()
```

✓ Explanation:

- **Boxplot** is used to show the **spread and outliers** in the data.
- **x="variety"** groups the boxplots by flower variety.
- **y="petal.length"** shows the distribution of petal lengths.
- Very useful for **comparing distributions across categories**.

3. Pairplot using Seaborn

File: pairplot.py

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

def main():
    df = pd.read_csv("iris.csv")

    sns.pairplot(df, hue="variety")

    plt.show()

if __name__ == "__main__":
    main()
```

✓ Explanation:

- **Pairplot** plots all possible **pairwise scatter plots** of features.
- `hue="variety"` colors the points based on flower type.
- Great for **understanding feature relationships** and **visualizing class separation**.
- Very helpful in **Exploratory Data Analysis (EDA)**.

4. 3D Scatter Plot using Matplotlib

File: subplot.py

```
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

def main():
    iris = load_iris()
    x = iris.data
    y = iris.target

    fig = plt.figure(figsize=(8,6))

    ax = fig.add_subplot(111, projection='3d')

    ax.scatter(x[:,2], x[:,3], x[:,0], c=y, cmap="viridis",
edgecolor='k')

    ax.set_xlabel("Petal length")
    ax.set_ylabel("Petal width")
    ax.set_zlabel("Sepal length")

    plt.title("Marvellous 3D visualisation for IRIS")

    plt.show()

if __name__ == "__main__":
    main()
```

✓ Explanation:

- This is a **3D scatter plot** using `mpl_toolkits.mplot3d`.
- `projection='3d'` enables 3D plotting.
- `ax.scatter()` plots the points with different colors for each class (`c=y`).
- `x[:, 2], x[:, 3], x[:, 0]`: These are features chosen as axes.
- Useful for visualizing **multi-dimensional relationships**.



Data Types in Python

A **data type** defines the type of a value and the set of operations that can be performed on that value. In simple words, it tells **what kind of data** a variable can hold, like numbers, text, boolean values, collections, etc.

In Python, **everything is an object**, and every object belongs to some **class** or **data type**.

Python is a Dynamically Typed Language

- In Python, **you don't need to declare the type** of a variable while creating it.
- The **type is automatically assigned** during **runtime** based on the value assigned.
- You can even **change the type of a variable** by assigning a new value of different type.

Dynamic Typing Means:

- Variable type can change any time.
- **Type safety is ensured during execution**, not while writing code.

Advantages:

- Easier and faster to write code.
- No need for lengthy type declarations.

Important Behavior:

- Python checks types at **runtime**, not at compile-time.
- Variables are just **labels pointing to objects**, and the object carries the type information, **not the variable** itself.

Example:

```
x = 10          # x is an integer
x = "Python"    # Now x is a string
```

Variable creation in Python

- A **variable** is a **name** that refers to a **location in memory** where data is stored.
- It acts as a container to store values.

Data Types in Python

Type	Name	Example
int	Integer	x = 11
float	Floating number	pi = 3.14
bool	Boolean	is_active = True
str	String	name = "Python"
list	List	Batches = ["PPA", "LB"]
tuple	Tuple	Fees = (21000, 22000)
dict	Dictionary	person = { "name": "Piyush", "age": 34}
set	Set	numbers = {1, 2, 3}
complex	Complex number	z = 5 + 6j
NoneType	None value	value = None

Variable creation in Python

```
# Creating variables
a = 10
b = 3.14
c = "Jay Ganesh"
d = True

# Multiple assignments
x, y, z = 1, 2, 3
```

Functions to Check Data Type of a Variable

type() Function:

Used to find the **data type** of any variable.

```
x = 101
print(type(x))    # <class 'int'>

y = "Marvellous"
print(type(y))    # <class 'str'>
```

Function to Check Size of Variable in Memory

sys.getsizeof() Function:

Used to check how much **memory (in bytes)** a variable occupies.

Syntax:

```
import sys
sys.getsizeof(variable_name)
```

Example:

```
import sys

x = 11
print(sys.getsizeof(x)) # 28 bytes
```

Size of Different Data Types

Data Type	Example	Size (Bytes)
int	11	28
float	3.14	24
bool	TRUE	28
str	"Marvellous"	52 (variable)
list	[11,21,51]	88
tuple	(1,12,151)	72
dict	{"a":1}	232
set	{1,2}	216

id() Function to Get Information About Variable

- Every object in Python has a **unique id** (memory address).
- **id()** returns that address as an **integer**.

Example:

```
a = 100
b = a

print(id(a))      # Memory location of 'a'
print(id(b))      # Memory location of 'b'
```

Code to explain the above concepts

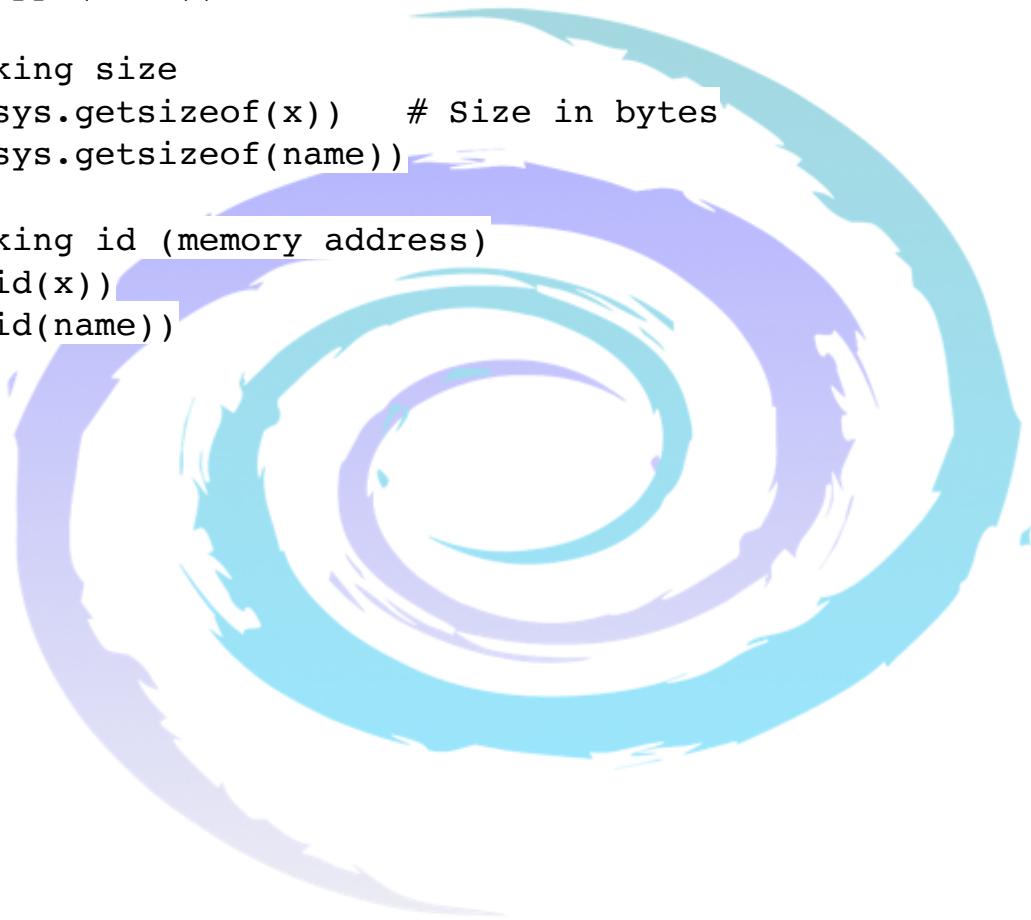
```
import sys

# Variable Creation
x = 11
name = "Marvellous"

# Checking type
print(type(x))      # <class 'int'>
print(type(name))    # <class 'str'>

# Checking size
print(sys.getsizeof(x))  # Size in bytes
print(sys.getsizeof(name))

# Checking id (memory address)
print(id(x))
print(id(name))
```



Datasets for Machine Learning

Please refer below links of datasets that we can use to solve case studies.

In Case study round almost below datasets are referred.

[Open Datasets](#)

[Kaggle](#)

A data science community with tools and resources which include externally contributed machine learning datasets of all kinds. From health, through sports, food, travel, education, and more, Kaggle is one of the best places to look for quality training data.

[Google Dataset Search](#)

A search engine from Google that helps researchers locate freely available online data. It works similarly to Google Scholar, and it contains over 25 million datasets. You can find here economic and financial data, as well as datasets uploaded by organizations like WHO, Statista, or Harvard.

[OpenML](#)

An online machine learning platform for sharing and organizing data with more than 21.000 datasets. It's regularly updated and it automatically versions and analyses each dataset and annotates it with rich meta-data to streamline analysis.

[DataHub](#)

A collection of thousands of machine learning datasets from financial market data, macroeconomic data, and population growth to cryptocurrency prices. You can access it without any registration.

[Papers with Code](#)

A community project with free and open resources, currently including 3937 datasets for data science and machine learning, including natural

language processing tasks. You can easily filter them by modality, task, or language.

VisualData

A search engine for computer vision datasets. You can easily filter them by category, date, popularity or use a search box to find a theme-specific dataset. A great source of datasets for image classification, image processing, and image segmentation projects.



Public Government datasets

[Data.gov](#)

The US government's open data site. You can filter it by various industries like healthcare, climate, education, etc. Be aware that much of this open-source data might require additional research.

[EU Open Data Portal](#)

The point of access to public data published by the EU institutions, agencies, and other entities. It contains data related to economics, agriculture, education, employment, climate, finance, science, etc.

[World Bank](#)

The open data from the World Bank that you can access without registration. It contains data concerning population demographics, macroeconomic data, and key indicators for development. A great source of data to perform data analysis at a large scale.

[US Healthcare Data](#)

Statistics and datasets for health care and public health. You can find data about population health, diseases, drugs, and health plans collected from the FDA and USDA Food composition databases.

[The US National Center for Education Statistics](#)

It's a website with data on educational institutions and education demographics in the U.S. and internationally.

[The UK Data Service](#)

It's a platform that provides access to over 7,000 digital data collections for research and teaching purposes. You can find here economic and social data from the Economic and Social Data Service (ESDS), Census Programme, and others, including some international data sets.

Data USA

A free platform with the most comprehensive visualization of U.S. public data.



Machine Learning Datasets for Finance and Economics

[Global Financial Development \(GFD\)](#)

An extensive dataset of financial system characteristics for 214 economies around the world. It contains annual data which has been collected since 1960.

[Financial Times Markets Data](#)

Up-to-date source of data on financial markets from around the world. The dataset contains information about share and stock prices, equities, currencies, bonds, and commodities performance.

[Quandl](#)

A platform with rich datasets of financial, economic, and alternative data. Quandl's data comes in two formats: Time-series (data taken over a period of time) and Tables (numerical and unsorted data types such as strings, etc.) You can download it either as a JSON or CSV file.

[IMF Data](#)

International Monetary Fund publishes data related to the IMF lending, exchange rates, and other economic and financial indicators.

[American Economic Association \(AEA\)](#)

A website with links to some of the most useful and popular economic data sources. It includes data on U.S macroeconomic as well as individual-level global data on income, employment, and health.

Deep Learning Project

Realtime Image Classification System using CNN

This project is a **real-time image classification system** using a **pre-trained Convolutional Neural Network (CNN)** (MobileNetV2) and our webcam.

- **Goal:** Capture live video from your webcam and classify each frame into one of the **ImageNet** categories (like dog, cat, keyboard, etc.).
- **Core Idea:** Use a **pre-trained CNN model (MobileNetV2)** to recognize objects without having to train from scratch.
- **Output:** Display the webcam feed with a label (class name + confidence percentage) overlaid in real-time.

Complete Source Code

```

import cv2
import numpy as np
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2, preprocess_input,
decode_predictions

def MarvellousImageClassifier():
    # 1) Load pre-trained CNN (ImageNet)
    model = MobileNetV2(weights="imagenet")

    # 2) Open webcam
    cap = cv2.VideoCapture(0)
    if not cap.isOpened():
        print("Error: Could not open webcam.")
        return

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Error: Could not read frame.")
            break

        # 3) Preprocess for MobileNetV2: BGR -> RGB, resize to 224x224, scale
        img = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        img_resized = cv2.resize(img, (224, 224))
        x = np.expand_dims(img_resized, axis=0).astype(np.float32)
        x = preprocess_input(x)

        # 4) Predict
        preds = model.predict(x, verbose=0)
        decoded = decode_predictions(preds, top=1)[0][0] # (class_id, class_name, score)
        label = f"{decoded[1]}: {decoded[2]*100:.1f}%"

        # 5) Overlay prediction on the frame
        cv2.putText(frame, label, (16, 40), cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 255, 0), 2, cv2.LINE_AA)

        cv2.imshow("Real-time CNN Classification (MobileNetV2)", frame)

    # 6) Exit
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
  
```

```

# 7) Cleanup
cap.release()
cv2.destroyAllWindows()

def main():
    MarvellousImageClassifier()

if __name__ == "__main__":
    main()
  
```

Explanation of project

1. Importing Libraries

```

import cv2
import numpy as np
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2,
preprocess_input, decode_predictions
  
```

- **cv2 (OpenCV):** For webcam capture, image processing, and display.
- **numpy:** For numerical operations (reshape, expand dimensions).
- **MobileNetV2 + utils:** Pre-trained CNN model + preprocessing + decoding predictions.

2. Loading the Pre-trained Model

```

model = MobileNetV2(weights="imagenet")
  
```

- Loads **MobileNetV2**, a **lightweight CNN trained on ImageNet (1.4M images, 1000 classes)**.
- Since it's pre-trained, you don't need to train it again — it's ready for inference.

3. Accessing the Webcam

```

cap = cv2.VideoCapture(0)
  
```

- Opens the **default webcam** (index 0).
- If webcam isn't found, it shows an error.

4. Frame Capture & Preprocessing

```

ret, frame = cap.read()
img = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
img_resized = cv2.resize(img, (224, 224))
x = np.expand_dims(img_resized, axis=0).astype(np.float32)
x = preprocess_input(x)
  
```

- **Frame:** Captures one video frame.
- **BGR → RGB:** OpenCV uses BGR, but MobileNet expects RGB.

- **Resize:** CNN input size = 224x224.
- **Expand dimensions:** CNN expects input shape (1, 224, 224, 3).
- **Preprocess:** Normalizes pixels for MobileNetV2.

5. Prediction

```
preds = model.predict(x, verbose=0)
decoded = decode_predictions(preds, top=1)[0][0]
label = f"{decoded[1]}: {decoded[2]*100:.1f}%"
  • model.predict: Runs forward pass through CNN.
```

- **decode_predictions:** Converts raw model output into readable labels (e.g., "Labrador retriever").
- **Top-1 label:** Takes the best match with confidence percentage.

6. Overlay Prediction

```
cv2.putText(frame, label, (16, 40), cv2.FONT_HERSHEY_SIMPLEX, 1.0,
(0, 255, 0), 2, cv2.LINE_AA)
cv2.imshow("Real-time CNN Classification (MobileNetV2)", frame)
  • Draws the prediction text on the video frame.
  • Displays the live webcam feed with classification results.
```

7. Exit & Cleanup

```
if cv2.waitKey(1) & 0xFF == ord('q'):
    break
cap.release()
cv2.destroyAllWindows()
  • Press 'q' to quit.
  • Releases webcam and closes all windows.
```

Important Concepts

- **CNN (Convolutional Neural Network):** Learns spatial features from images (edges → textures → objects).
- **Transfer Learning:** Using a model trained on a huge dataset (ImageNet) for your own application.
- **MobileNetV2:** A lightweight CNN optimized for speed and accuracy, good for real-time applications.

Artificial Intelligence

Machine Learning & Deep Learning

Artificial Intelligence (AI)

- **Definition:**
The science of making machines “think” and “act” like humans.
- **Goal:**
To build systems that can **perceive, reason, learn, and decide**.
- **Examples:**
 - Google Maps suggesting routes
 - Alexa/Siri voice assistants
 - Self-driving cars

AI is the **umbrella term** under which ML, DL, and GenAI exist.

Machine Learning (ML)

- **Definition:**
A subset of AI where machines **learn from data** instead of being explicitly programmed.
- **Working principle:**
 - Give data → Machine **finds patterns** → Makes predictions.
- **Types of ML:**
 - **Supervised Learning** – Labeled data (e.g., Spam vs Not Spam).
 - **Unsupervised Learning** – No labels, only patterns (e.g., Customer segmentation).

Example: Training a model to predict house prices based on past sales data.

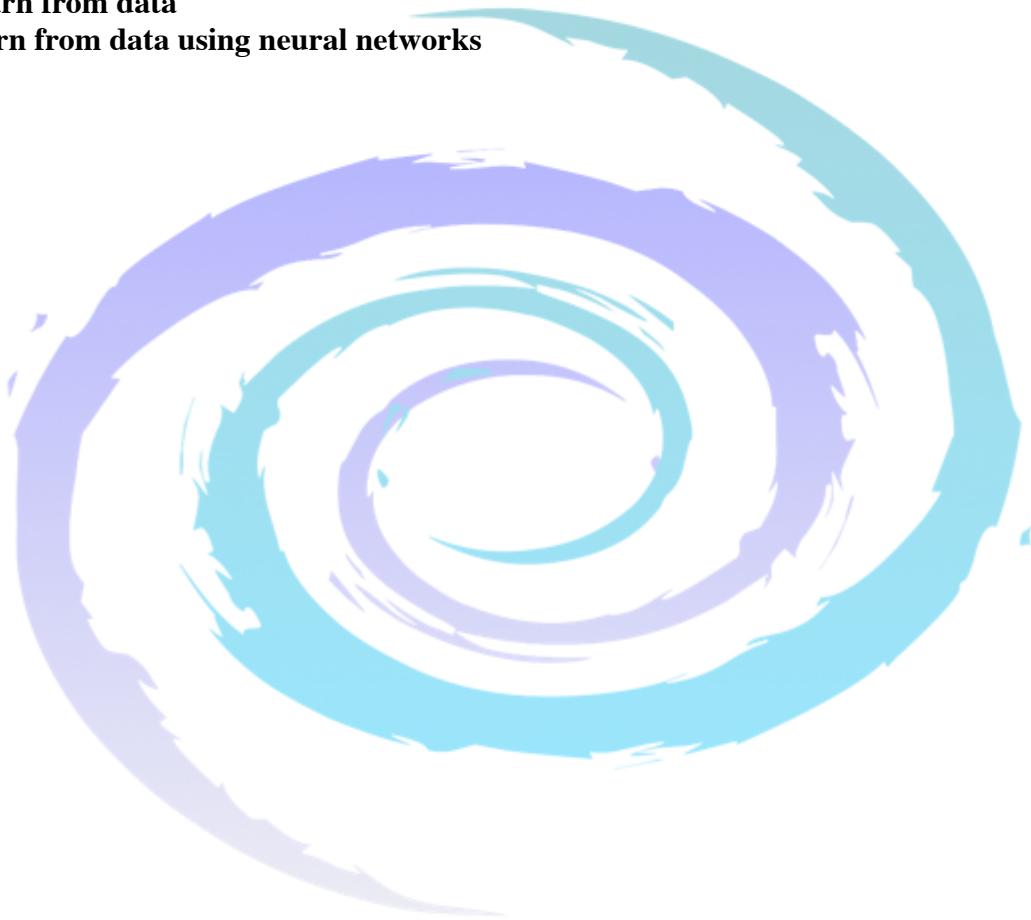
Deep Learning (DL)

- **Definition:**
A special branch of ML using **artificial neural networks (ANNs)** with multiple layers to learn complex patterns.
- **Why “Deep”?**
Because it has **many hidden layers** between input and output.

- **Strengths:**
 - Works on images, audio, video, natural language.
 - Can automatically extract features (no manual feature engineering).
- **Examples:**
 - Face recognition on phones
 - Speech-to-text systems
 - Autonomous driving vision systems

ML = Learn from data

DL = Learn from data using neural networks



Encoder vs Decoder for Transformers

Encoder

- **Purpose:** *Understand and represent the input text.*
- **How it works:**
 1. Takes an input sentence (e.g., “The cat sat on the mat”).
 2. Converts each word into embeddings (vectors of numbers).
 3. Uses **self-attention** to understand how each word relates to others.
 - e.g., “bank” → should I connect it with “river” or “money”?
 4. Produces a **contextual representation** of the sentence.
- **Key idea:** It does **not generate new text**, only encodes meaning.
- **Used in:** BERT, translation encoders, text classification.

Think of the encoder as a **reader** who *understands* everything in the input.

Decoder

- **Purpose:** *Generate output step by step.*
- **How it works:**
 1. Starts with encoded information (from encoder, or past tokens if standalone like GPT).
 2. Generates one word at a time → predicts next word.
 3. Uses **self-attention** (to check previous words) + **encoder-decoder attention** (if encoder is present).
 4. Continues until output is complete.
- **Key idea:** It's **generative**.
- **Used in:** GPT (text generation), translation decoders, chatbots.

Think of the decoder as a **writer** who *produces* text word by word.

Putting It Together

- **Encoder-only models:** BERT → good at *understanding*.
- **Decoder-only models:** GPT → good at *generating*.
- **Encoder–Decoder models:** Used in *machine translation* (e.g., Google Translate).
 - Encoder: Understands source language sentence.
 - Decoder: Generates translated sentence in target language.



Ensemble Machine Learning application with Boosting technique

MNIST case study :

```

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

data = pd.read_csv('mnist.csv')

df_x = data.iloc[:,1:] # Labels
df_y = data.iloc[:,0] # Pixels

x_train, x_test, y_train, y_test = train_test_split(df_x, df_y, test_size=0.2,
random_state=4)

obj = DecisionTreeClassifier(___, ___, ___);
adb = AdaBoostClassifier(obj,n_estimators = ___, learning_rate = ___);
adb = AdaBoostClassifier(DecisionTreeClassifier(),n_estimators = 100 , learning_rate = 1)
adb.fit(x_train,y_train)

print("Testing accuracy using bagging classifier : ",adb.score(x_test,y_test)*100)

print("Training accuracy using bagging classifier : ",adb.score(x_train,y_train)*100)

```

Iris Case study :

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets
# Import train_test_split function
from sklearn.model_selection import train_test_split
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics

# Load data
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training
and 30% test

# Create adaboost classifier object
abc = AdaBoostClassifier(n_estimators=50,
                         learning_rate=1)
# Train Adaboost Classifier
model = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = model.predict(X_test)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Ensemble Machine Learning application with heterogeneous algorithm technique

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
iris = load_iris()
x = iris['data']
y = iris['target']
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y,
random_state = 42, train_size = 0.85)
```

```
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
knn_clf = KNeighborsClassifier()
```

```
vot_clf = VotingClassifier(estimators = [('lr', log_clf), ('rnd',
rnd_clf), ('knn', knn_clf)], voting = 'hard')
```

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

```
pred = vot_clf.predict(x_test)
```

```
print("Testing accuracy is : ",accuracy_score(y_test,
pred)*100)
```

Ensemble Machine Learning

Ensemble learning is a machine learning technique where **multiple models (weak learners)** are trained and combined to solve the same problem in order to get **better accuracy, stability, and robustness**.

Types of Ensemble Methods

1. **Bagging (Bootstrap Aggregating)**
2. **Boosting**

Bagging (Bootstrap Aggregating)

Train **multiple models independently** on **random subsets** of the data and **average/vote their predictions**.

How it works:

1. Create **n random subsets** of data from the original dataset **with replacement** (bootstrap sampling).
2. Train a **separate model** (often the same algorithm, like Decision Tree) on each subset.
3. Combine the predictions:
 - **Classification** → Majority Voting
 - **Regression** → Averaging

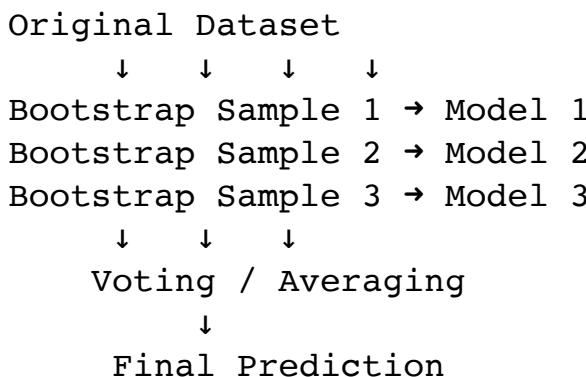
Benefits:

- Reduces **variance**
- Avoids **overfitting**
- Works well with **unstable models** (e.g., Decision Trees)

Popular Bagging Algorithm:

- **Random Forest** – An ensemble of decision trees using bagging + feature randomness.

Diagram:



Boosting

Train models **sequentially**, where **each model learns from the mistakes** of the previous ones.

How it works:

1. Start with an initial model (usually weak like a shallow tree).
2. Evaluate errors → focus more on **incorrectly predicted data points**.
3. Train the next model on the updated data (adjusted weights).
4. Repeat for n iterations.
5. Combine all models using **weighted voting/averaging**.

Benefits:

- Reduces **bias**
- Improves performance on **complex datasets**
- Learns from mistakes

Drawbacks:

- Can **overfit** if not tuned properly
- **Slower** than bagging (sequential training)

Popular Boosting Algorithms:

Algorithm	Key Features
AdaBoost	Adjusts weights based on errors
Gradient Boosting	Trains new models to predict residuals (errors)
XGBoost	Optimized version of Gradient Boosting (fast, regularized)
LightGBM	Faster and more scalable boosting using histogram-based splits
CatBoost	Handles categorical features automatically

Bagging vs Boosting Comparison:

Feature	Bagging	Boosting
Model Training	Parallel (independent)	Sequential (one after another)
Focus	Reduce variance	Reduce bias and variance
Example	Random Forest	AdaBoost, Gradient Boosting
Overfitting	Less prone	More prone if not regularized
Speed	Faster (can train in parallel)	Slower (sequential training)

When to Use What :

Situation	Best Technique
High variance model (overfitting)	Bagging
High bias model (underfitting)	Boosting
Small to medium data	Boosting
Large-scale data	Bagging or XGBoost

Real-Life Example:

- **Bagging** (Random Forest): Useful in spam detection, credit scoring, etc.
- **Boosting** (XGBoost, LightGBM): Common in Kaggle competitions, customer churn prediction, fraud detection.

Error & Loss Functions of Artificial Neural Network

Loss functions measure **how wrong** the network's predictions are. The training process tries to minimize this error.

Loss Function

- **Definition:** A mathematical function that measures the difference between predicted output and actual (true) output.
- **Types:**

- **Mean Squared Error (MSE):** For regression.

$$L = \frac{1}{n} \sum (y_{true} - y_{pred})^2$$

- **Cross-Entropy Loss:** For classification.

$$L = - \sum y_{true} \log(y_{pred})$$

- **Example:**

True = [0,1], Predicted = [0.2,0.8] → Cross-Entropy Loss is small (good prediction).

Cost Function

- **Definition:** The **average loss** over all training samples.
- **Explanation:**
 - Loss is calculated per sample, Cost = average over batch/epoch.
- **Example:**
If loss for 3 samples = [0.1, 0.2, 0.3], Cost = $(0.1+0.2+0.3)/3 = 0.2$.

Overfitting

- **Definition:** When ANN memorizes training data but fails to generalize to new data.
- **Symptoms:**
 - Very low training error, high test error.
- **Example:**
A student memorizing answers but failing in real exams.

Underfitting

- **Definition:** When ANN fails to capture underlying patterns in data.
- **Symptoms:**
 - High training error and high test error.
- **Example:**
A student not studying enough → poor performance everywhere.

Regularization Techniques

Used to prevent **overfitting** by reducing model complexity.

- **L1 Regularization (Lasso):**
Adds absolute values of weights as penalty.
Encourages **sparsity** (some weights become zero).
- **L2 Regularization (Ridge):**
Adds squared values of weights as penalty.
Keeps weights small, prevents large unstable updates.
- **Dropout:**
Randomly turns off some neurons during training.
Forces network to learn robust features instead of memorizing.
- **Example:**
In image classification, dropout ensures the model learns **general patterns** (like edges, shapes) instead of memorizing exact training images.

Important terms :

- **Activation Functions:** Decide how neurons fire (Sigmoid, ReLU, Tanh, Softmax).
- **Loss Functions:** Measure how wrong predictions are.
- **Overfitting/Underfitting:** Explain generalization performance.
- **Regularization:** Prevents overfitting and improves robustness.

Exception Handling

Exception :

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

Error vs Exception :

Error: An Error indicates serious problem that a reasonable application should not try to catch.

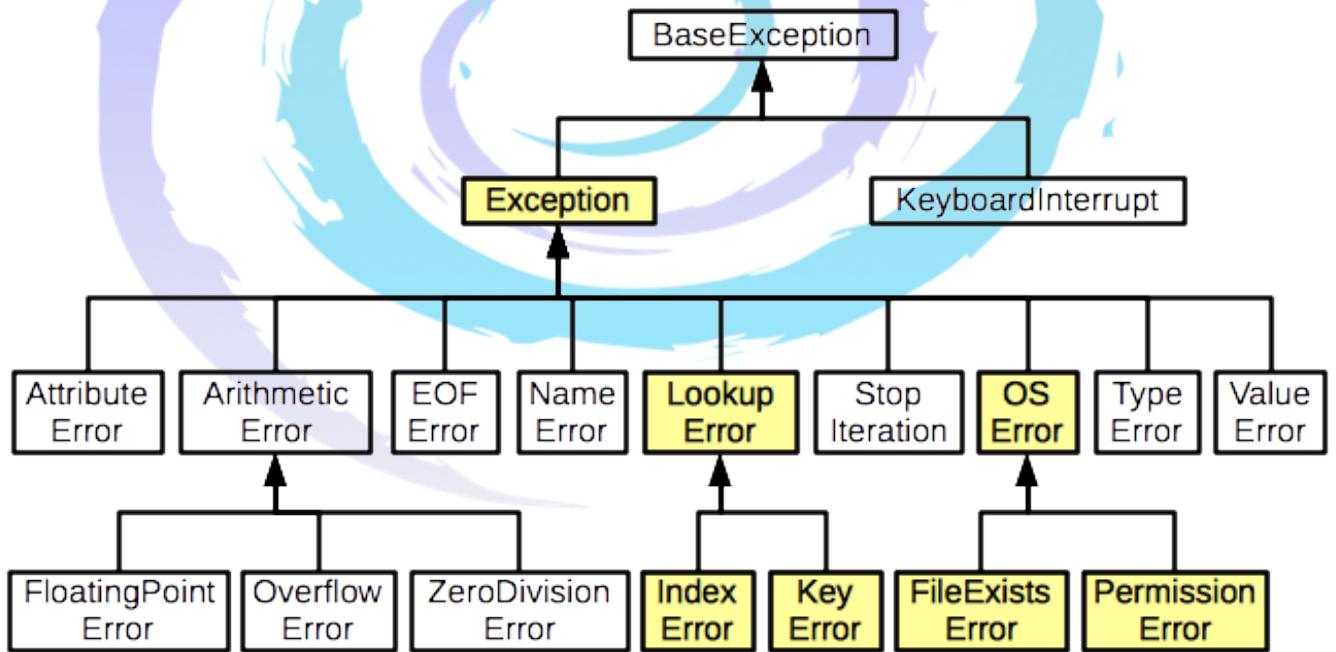
Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy :

All exception and errors types are sub classes of class `Exception`, which is base class of exception hierarchy.

One branch is headed by `Exception`.

This class is used for exceptional conditions that user programs should catch.



Built in Exceptions in Python

According to above diagram there are multiple inbuilt exceptions as

Exception	Cause of Error
AssertionError	Raised when <code>assert</code> statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the <code>input()</code> functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's <code>close()</code> method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.

RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by <code>next()</code> function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by <code>sys.exit()</code> function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

Internal working of PVM to handle an Exception

- Default Exception Handling : Whenever inside a python application, if an exception has occurred, the PVM creates an Object known as Exception Object and hands it off to the run-time system(PVM).
- The exception object contains name and description of the exception, and current state of the program where exception has occurred.
- Creating the Exception Object and handing it to the run-time system is called throwing an Exception.
- There might be the list of the methods that had been called to get to the method where exception was occurred.
- This ordered list of the methods is called Call Stack.

Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called Exception handler. ie except block
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to default exception handler , which is part of run-time system. This handler prints the exception information and terminates the program abnormally.

For Exception handling in Python we have to use three keyword as

try :

The code which is written inside try block is considered as an exception prone code means which may generate exception.

except :

This block is called as exception handler which gets executed when exception is occurred.

finally :

This block gets executed always irrespective of exception. Generally this block is used to release all resources.

In short the error handling is done through the use of exceptions that are caught in try blocks and handled in except blocks. If an error is encountered, a try block code execution is stopped and transferred down to the except block.

In addition to using an except block after the try block, you can also use the finally block.

The code in the finally block will be executed regardless of whether an exception occurs.

Consider below application which demonstrate concept of Exception Handling

```

print("---- Marvellous Infosystems by Piyush Khairnar----")
print("Demonstration of Exception Handling")

no1 = int(input("Enter first number"))
no2 = int(input("Enter second number"))

try:
    ans = no1/no2
    print("Division is ",ans)

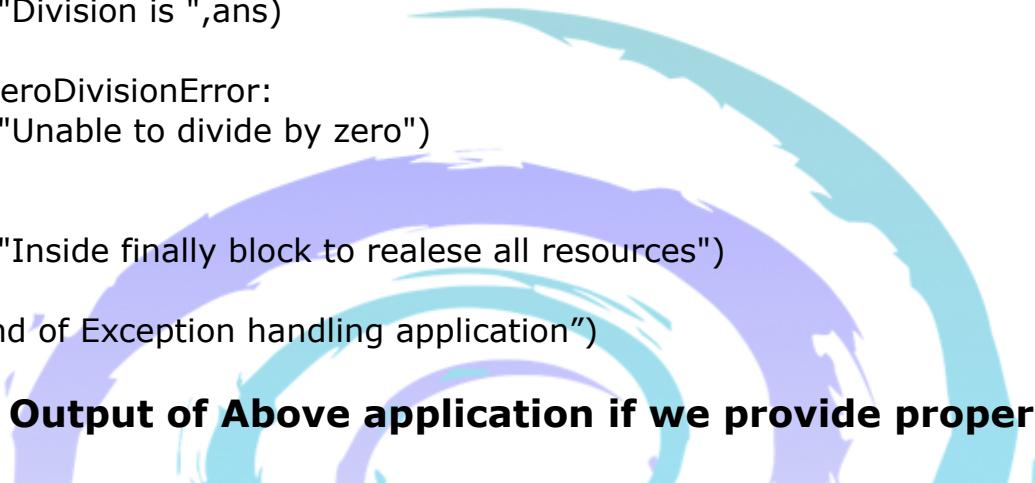
except ZeroDivisionError:
    print("Unable to divide by zero")

finally:
    print("Inside finally block to realese all resources")

print("End of Exception handling application")

```

Output of Above application if we provide proper input



```

MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ python Exceptions.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Exception Handling
Enter first number12
Enter second number4
('Division is ', 3)
Inside finally block to realese all resources
End of Exception handling application
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ █

```

Output of above application if we enter second number as Zero

```

MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ python Exceptions.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Exception Handling
Enter first number21
Enter second number0
Unable to divide by zero
Inside finally block to realese all resources
End of Exception handling application
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ █

```

Feedforward Neural Network (FNN)

A **Feedforward Neural Network (FNN)** is the simplest kind of neural network where information flows in one direction: **inputs → hidden layer(s) → output**.

There are **no loops** or feedback connections.

It maps an input vector to an output (a prediction) using layers of neurons that apply linear combinations and non-linear activations.

An FNN repeatedly applies:

Linear: $z = W \cdot a + b \rightarrow$ **Non-linear:** $a = g(z) \rightarrow$ next layer \rightarrow final output \rightarrow compute loss \rightarrow adjust W, b by gradient descent.

Important terms

- **Neuron / Node:** A tiny function that takes numbers, multiplies by **weights (W)**, adds a **bias (b)**, applies an **activation**.
- **Layer:** A group of neurons.
 - **Input layer:** just the features (no computation).
 - **Hidden layer(s):** do most of the learning.
 - **Output layer:** produces the prediction (class scores or a number).
- **Weights (W):** Strength of connections; learned values.
- **Bias (b):** Constant added to each neuron; lets the neuron shift.
- **Activation (g):** Adds nonlinearity so the network can learn curved/complex patterns. Common:
 - $\text{ReLU}(x) = \max(0, x)$ (fast, popular)
 - $\text{sigmoid}(x) = 1 / (1 + e^{-x})$ (probabilities for binary)
 - $\tanh(x)$ (values in $-1..1$)
 - **softmax** (turns scores into class probabilities for multi-class)
- **Forward pass:** Compute outputs given inputs and current parameters.
- **Loss function:** How wrong we are.
 - Regression: **MSE**
 - Binary classification: **Binary Cross-Entropy (Log loss)**
 - Multi-class: **Cross-Entropy**

- **Backward pass (Backpropagation):** Compute gradients of loss w.r.t. parameters using the chain rule.
- **Gradient Descent:** Update rule to reduce loss.
 $W \leftarrow W - \alpha * (\partial L / \partial W)$; α is **learning rate**.
- **Epoch:** One full pass over the training set.
- **Batch size:** How many samples we use before one update (full-batch, mini-batch, or stochastic).
- **Hyperparameters:** Not learned—you choose them (learning rate, hidden units, layers, epochs, batch size, activation, optimizer, regularization).
- **Regularization:** Prevent overfitting (L2, dropout, early stopping).

Mathematical explanation

For layer ℓ :

- **Linear:** $z^\wedge(\ell) = W^\wedge(\ell) \cdot a^\wedge(\ell-1) + b^\wedge(\ell)$
- **Activation:** $a^\wedge(\ell) = g^\wedge(\ell)(z^\wedge(\ell))$
 with $a^\wedge(0) = x$ (input), and $a^\wedge(L)$ the network output.

Binary classification head (one output):

- $\hat{y} = \sigma(z) = 1 / (1 + e^{-z})$
- Loss (per sample): $L = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$
- Useful gradient: $\partial L / \partial z = \hat{y} - y$ (beautifully simple!)

Softmax head (multi-class):

- $\hat{y}_k = \exp(z_k) / \sum_j \exp(z_j)$
- Loss: cross-entropy $L = -\sum_k y_k \log(\hat{y}_k)$

Step-by-step numeric example (forward pass)

A $2 \rightarrow 2 \rightarrow 1$ network, ReLU hidden, sigmoid output.

Input: $x = [2.0, 3.0]$

Hidden layer (2 neurons):

Weights $W1 = [[0.5, -0.2], [0.8, 0.4]]$, Bias $b1 = [0.1, -0.1]$

- Neuron 1:

$$z_1 = 0.5*2 + (-0.2)*3 + 0.1 = 1.0 - 0.6 + 0.1 = 0.5$$

$$a_1 = \text{ReLU}(0.5) = 0.5$$
- Neuron 2:

$$z_2 = 0.8*2 + 0.4*3 - 0.1 = 1.6 + 1.2 - 0.1 = 2.7$$

$$a_2 = \text{ReLU}(2.7) = 2.7$$

Hidden output $\hat{a}^{(1)} = [0.5, 2.7]$

Output layer (1 neuron, sigmoid):

Weights $W2 = [1.2, -0.7]$, Bias $b2 = 0.05$

$$z_{\text{out}} = 1.2*0.5 + (-0.7)*2.7 + 0.05 = 0.6 - 1.89 + 0.05 = -1.24$$

$$\hat{y} = \text{sigmoid}(-1.24) \approx 0.2244 (\approx 22.44\% \text{ positive class probability})$$

This is exactly how a forward pass works, layer by layer.

Practical implementation of above example :

```

import math

def relu(x): return max(0.0, x)
def sigmoid(z): return 1/(1+math.exp(-z))

x = [2.0, 3.0]
W1 = [[0.5, -0.2],
      [0.8, 0.4]]
b1 = [0.1, -0.1]

# Hidden neuron 1
z1 = W1[0][0]*x[0] + W1[0][1]*x[1] + b1[0]
a1 = relu(z1)
print(f"Neuron1: z={z1:.3f}, a={a1:.3f}")

# Hidden neuron 2
z2 = W1[1][0]*x[0] + W1[1][1]*x[1] + b1[1]
a2 = relu(z2)
print(f"Neuron2: z={z2:.3f}, a={a2:.3f}")

# Output layer
W2 = [1.2, -0.7]; b2 = 0.05
z_out = W2[0]*a1 + W2[1]*a2 + b2
yhat = sigmoid(z_out)
print(f"Output: z={z_out:.3f}, sigmoid(y)={yhat:.4f}")

```

Example using SKlearn

```

from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Synthetic 2D classification
X, y = make_moons(n_samples=1000, noise=0.2, random_state=42)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.25,
random_state=42)

# 2 hidden layers: 16 and 8 neurons, ReLU + Adam optimizer
clf = MLPClassifier(hidden_layer_sizes=(16,8), activation='relu',
                     solver='adam', max_iter=500, random_state=42)
clf.fit(Xtr, ytr)

print("Test accuracy:", accuracy_score(yte, clf.predict(Xte)))

```

- Same feedforward concept; library handles backprop/optimizers.
- You only choose architecture and hyperparameters.

When to use an FNN

- Tabular data (numerical/categorical features).
- Simple image/text problems as baselines

Common issues in FNN

- **No nonlinearity** \Rightarrow **can't model XOR/complex patterns.** Always use an activation in hidden layers.
- **Learning rate too high/low:** Too high diverges; too low is painfully slow.
- **Unscaled features:** Standardize inputs for stability.
- **Overfitting:** Use validation set, L2, dropout, early stopping.
- **Too shallow or too few neurons:** Underfits; add capacity gradually.

File IO in Python

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first.

When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

Open file :

Python has a built-in function `open()` to open a file.

This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

`Fd1 = open("Marvellous.txt",'r')`

Search file in current directory as we provide relative path of file

`Fd2 = open("/Users/marvellous/Desktop/Today/Marvellous.txt")`

Search in specific path as we provide absolute path

We can specify the mode while opening a file.

In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file.

We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

```
f = open("Marvellous.txt")           # equivalent to 'r' or 'rt'  
f = open("Marvellous.txt",'w')        # write in text mode  
f = open("LogoMarvellous.bmp",'r+b')  # read and write in binary mode
```

Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).

Moreover, the default encoding is platform dependent.

In windows, it is 'cp1252' but 'utf-8' in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("Marvellous.txt",mode = 'r',encoding = 'utf-8')
```

Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Close File :

When we are done with operations to the file, we need to properly close the file. Closing a file will free up the resources that were tied with the file and is done using Python `close()` method. Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

```
f = open("Marvellous.txt",encoding = 'utf-8')

# perform file operations

f.close()
```

Read data from file:

Consider below file that we refer for reading

Marvellous.txt

Marvellous Infosystems by Piyush Manohar Khairnar
 Karve Road Pune 411004
 Educating for better tomorrow..

To read a file in Python, we must open the file in reading mode.

There are various methods available for this purpose. We can use the `read(size)` method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.

```
fd = open("Marvellous.txt",'r',encoding = 'utf-8')
```

```
fd.read(10) # read the first 10 data
```

Output :

"Marvellous"

```
fd.read(12) # read the next 12 data
```

Output :

"Infosystems"

```
fd.read() # read in the rest till end of file
```

Output :

by Piyush Manohar Khairnar

Karve Road Pune 411004

Educating for better tomorrow..

We can change our current file cursor (position) using the `seek()` method. Similarly, the `tell()` method returns our current position (in number of bytes).

```
print("Current file position is",fd.tell()) # get the current file position
```

```
fd.seek(0) # bring file cursor to initial position
```

```
print("Contents of Whole file")
```

```
print(fd.read())
```

Writing data into file :

In order to write into a file in Python, we need to open it in write '`w`', append '`a`' or exclusive creation '`x`' mode.

We need to be careful with the '`w`' mode as it will overwrite into the file if it already exists. All previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using `write()` method. This method returns the number of characters written to the file.

```
fd = open("Marvellous.txt",'w+a',encoding = 'utf-8')
```

```
fd.write("Python : Automation and Machine Learning\n")
```

```
fd.write("Angular : Web Development\n")
```

File IO methods in Python

Method	Description
close()	Close an open file. It has no effect if the file is already closed.
detach()	Separate the underlying binary buffer from the <code>TextIOBase</code> and return it.
fileno()	Return an integer number (file descriptor) of the file.
flush()	Flush the write buffer of the file stream.
isatty()	Return <code>True</code> if the file stream is interactive.
read(<code>n</code>)	Read atmost <code>n</code> characters form the file. Reads till end of file if it is negative or <code>None</code> .
readable()	Returns <code>True</code> if the file stream can be read from.
readline(<code>n</code> =-1)	Read and return one line from the file. Reads in at most <code>n</code> bytes if specified.
readlines(<code>n</code> =-1)	Read and return a list of lines from the file. Reads in at most <code>n</code> bytes/characters if specified.
seek(<code>offset</code> , <code>from</code> = <code>SEEK_SET</code>)	Change the file position to <code>offset</code> bytes, in reference to <code>from</code> (start, current, end).
seekable()	Returns <code>True</code> if the file stream supports random access.
tell()	Returns the current file location.
truncate(<code>size</code> = <code>None</code>)	Resize the file stream to <code>size</code> bytes. If <code>size</code> is not specified, resize to current location.
writable()	Returns <code>True</code> if the file stream can be written to.
write(<code>s</code>)	Write string <code>s</code> to the file and return the number of characters written.
writelines(<code>lines</code>)	Write a list of <code>lines</code> to the file.

Consider below application which demonstrates file IO

```
fd = open("Marvellous.txt",'r')
print("Information about file : ",fd)
print("Contents of Whole file")
print(fd.read())

print("Reading single line from file")
print(fd.readline())

print("Current file position is",fd.tell())    # get the current file position
fd.seek(0)  # bring file cursor to initial position

print("Contents of Whole file")
print(fd.read())

fd.close()

fd = open("Marvellous.txt",'a+r')
fd.write("Python : Automation and Machine Learning\n")
fd.write("Angular : Web Development\n")

fd.seek(0)

print(fd.read())

fd.close()
```

Output of above application

```
MacBook-Pro-de-MARVELLOUS: Today marvellous$ python
FileIO.py
('Information about file : ', <open file 'Marvellous.txt', mode 'r' at 0x10397c5d0>)
Contents of Whole file
Marvellous Infosystems by Piyush Manohar Khaire
Karve Road Pune 411004
Educating for better tomorrow.. Python : Automation
and Machine Learning
Angular : Web Development
```

Reading single line from file

```
('Current file position is', 171)
Contents of Whole file
Marvellous Infosystems by Piyush Manohar Khaire
Karve Road Pune 411004
Educating for better tomorrow.. Python : Automation
and Machine Learning
Angular : Web Development
Marvellous Infosystems by Piyush Manohar Khaire
Karve Road Pune 411004
Educating for better tomorrow.. Python : Automation
and Machine Learning
Angular : Web Development
Python : Automation and Machine Learning
Angular : Web Development
```

filter(), map(), and reduce() in Python

Functional Programming in Python

Python supports a **functional programming style**, where **functions can be passed as arguments** to other functions.

The three most commonly used higher-order functions are:

- `filter()`
- `map()`
- `reduce()`

filter() Function

Purpose:

Filters elements of a sequence based on a condition (returns only those elements that satisfy the condition).

`filter(function, iterable)`

- **function:** A function that returns `True` or `False`
- **iterable:** A sequence (like list, tuple, etc.)

Returns an iterator – wrap it in `list()` to get final results.

Example:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)  # Output: [2, 4, 6]
```

Explanation:

- The lambda function checks for even numbers.
- `filter()` keeps only values for which function returns `True`.

Without Lambda (using `def`):

```
def is_even(n):
    return n % 2 == 0

even_nums = list(filter(is_even, nums))
```

map() Function

Purpose:

Applies a function to **each item** in an iterable and returns a new sequence.

Syntax:

```
map(function, iterable)
```

Returns an iterator – use `list()` to see the result.

Example:

```
nums = [1, 2, 3, 4]
Data = list(map(lambda x: x+1, nums))
print(Data)    # Output: [2,3,4,5]
```

Explanation:

- The lambda returns `x` increased by 1
- `map()` applies it to every element of `nums`

Without Lambda:

```
def increase(x):
    return x + 1

Data = list(map(increase, nums))
```

reduce() Function

Purpose:

Applies a function cumulatively to reduce the iterable to a **single value**.

Syntax:

```
from functools import reduce

reduce(function, iterable)
  • The function takes two arguments
  • Applies cumulatively: f(f(f(x1, x2), x3), x4)...
```

Example:

```
from functools import reduce

nums = [1, 2, 3, 4]
Sum = reduce(lambda x, y: x + y, nums)
print(Sum)    # Output: 10
```

Explanation:

- $(1+2) = 3, (2+3) = 5, (5+4) = 10$

Without Lambda:

```
def Add(x, y):
    return x + y

product = reduce(Add, nums)
```

Comparison Table:

Function	Purpose	Returns	Requires Import
<code>filter()</code>	Filters items based on condition	Filtered items	No
<code>map()</code>	Applies a function to each item	Transformed list	No
<code>reduce()</code>	Reduces to a single value	Single result	Yes (<code>functools</code>)

Function Arguments

In Python, user-defined functions can take four different types of arguments. The argument types and their meanings, however, are pre-defined and can't be changed. But a developer can, instead, follow these pre-defined rules to make their own custom functions.

The following are the four types of arguments and their rules.

Default arguments :

Python has a different way of representing syntax and default values for function arguments. Default values indicate that the function argument will take that value if no argument value is passed during function call. The default value is assigned by using assignment (=) operator.

Required arguments / Position Argument :

Required arguments are the mandatory arguments of a function. These argument values must be passed in correct number and order during function call.

Keyword arguments :

Keyword arguments are relevant for Python function calls. The keywords are mentioned during the function call along with their corresponding values. These keywords are mapped with the function arguments so the function can easily identify the corresponding values even if the order is not maintained during the function call.

Variable number of arguments:

This is very useful when we do not know the exact number of arguments that will be passed to a function. Or we can have a design where any number of arguments can be passed based on the requirement.

Consider below application which demonstrate concept of Function Arguments

```

print("---- Marvellous Infosystems by Piyush Khairnar----")
print("Demonstration of Types of Function Arguments")
# Position arguments

def Batches1(name,fees):
    print("Batch name is ", name)
    print("Fees are ", fees)

print("Demonstration of Position Arguments")
Batches1('Python', 5000)

```

```
Batches1(5000,'Angular')
# Keyword Arguments

def Batches2(name,fees):
    print("Batch name is ", name)
    print("Fees are ", fees)

print("Demonstration Keyword of Arguments")
```

```
Batches2(fees=9000, name='PPA')
Batches2(name='LB',fees=7500)
```

```
# Default Arguments
```

```
def Batches3(name,fees = 5000):
    print("Batch name is ", name)
    print("Fees are ", fees)
```

```
print("Demonstration of Default Arguments")
```

```
Batches3('Angular',7500)
Batches3('Angular')
Batches3(fees=9000, name='PPA')
Batches3(name='LB')
```

```
# Variable number of arguments
```

```
def Add(*no):
    ans = 0
    for i in no:
        ans = ans + i

    return ans
```

```
print("Demonstration of Variable number of Arguments")
```

```
ret = Add(10,20,30)
print("Addition is ",ret)
```

```
ret = Add(10,20,30,40,50,60)
print("Addition is ",ret)
```

```
ret = Add(10,20)
print("Addition is ",ret)
```

```
# Keyword Variable number of arguments
```

```
def StudentInfo(**other):
    print(other)
    for i,j in other.items():
        print(i,j)
```

```
print("Demonstration of Keyword Variable number of Arguments")
```

```
StudentInfo(age=28, address="Sinhagad Road", mobile=7588945488,
company="Marvellous")
```

Output of Above application

```
MacBook-Pro-de-MARVELLOUS: Today marvellous$ python FunctionArguments.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Types of Function Arguments
Demonstration of Position Arguments
('Batch name is ', 'Python')
('Fees are ', 5000)
('Batch name is ', 5000)
('Fees are ', 'Angular')
Demonstration Keyword of Arguments
('Batch name is ', 'PPA')
('Fees are ', 9000)
('Batch name is ', 'LB')
('Fees are ', 7500)
Demonstration of Default Arguments
('Batch name is ', 'Angular')
('Fees are ', 7500)
('Batch name is ', 'Angular')
('Fees are ', 5000)
('Batch name is ', 'PPA')
('Fees are ', 9000)
('Batch name is ', 'LB')
('Fees are ', 5000)
Demonstration of Keyword Variable number of Arguments
{'mobile': 7588945488, 'age': 28, 'company': 'Marvellous', 'address': 'Sinhagad
Road'}
('mobile', 7588945488)
('age', 28)
('company', 'Marvellous')
('address', 'Sinhagad Road')
MacBook-Pro-de-MARVELLOUS: Today marvellous$ █
```

Handwritten Digits Classification using Deep Learning Techniques

```

import os
import argparse
import random
from typing import Tuple

import numpy as np
import matplotlib.pyplot as plt

# TensorFlow / Keras
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Sklearn for metrics and baselines
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier

#
=====
=====
# Global config
#
=====
=====

SEED = 42
DIGIT_CLASSES = [str(i) for i in range(10)]
ARTIFACT_DIR = "artifacts_digits"
BEST_MODEL = os.path.join(ARTIFACT_DIR, "digits_cnn.h5")
FINAL_MODEL = os.path.join(ARTIFACT_DIR, "digits_cnn_final.h5")

#
=====
=====
# Utils
#
=====
=====

##########
#####

# Function: set_seed
# Inputs: seed (int) - random seed value
# Outputs: None
# Description: Sets seed for Python, NumPy, and TensorFlow to ensure reproducibility.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
##########

def set_seed(seed: int = SEED):
    os.environ["TF_DETERMINISTIC_OPS"] = "1"

```

```

random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)

#####
# Function: ensure_dir
# Inputs: path (str) - directory path
# Outputs: None
# Description: Creates directory if it does not exist.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####
def ensure_dir(path: str):
    os.makedirs(path, exist_ok=True)

#####

# Function: plot_training_curves
# Inputs: history (keras.callbacks.History), out_dir (str)
# Outputs: Saves accuracy and loss curve plots
# Description: Plots and saves training/validation accuracy and loss curves.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####
def plot_training_curves(history, out_dir=ARTIFACT_DIR):
    ensure_dir(out_dir)
    # Accuracy
    plt.figure(figsize=(7, 5))
    plt.plot(history.history["accuracy"], label="train_acc")
    plt.plot(history.history["val_accuracy"], label="val_acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title("Training vs Validation Accuracy")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "acc_curve.png"))
    plt.close()

    # Loss
    plt.figure(figsize=(7, 5))
    plt.plot(history.history["loss"], label="train_loss")
    plt.plot(history.history["val_loss"], label="val_loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training vs Validation Loss")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "loss_curve.png"))
    plt.close()

#####
# Function: _annotate_confmat
# Inputs: ax (matplotlib.axes.Axes), cm (np.ndarray), fmt (str)
# Outputs: Annotated confusion matrix
# Description: Adds numeric annotations to confusion matrix heatmap.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

```

```
#####
def _annotate_confmat(ax, cm, fmt=".2f"):
    n, m = cm.shape
    thresh = cm.max() / 2.0
    for i in range(n):
        for j in range(m):
            val = cm[i, j]
            color = "white" if val > thresh else "black"
            txt = f"{val:.2f}" if isinstance(val, float) else f"{val:d}"
            ax.text(j, i, txt, ha="center", va="center", color=color, fontsize=8)

#####
# Function: plot_confusion_matrix
# Inputs: y_true (np.ndarray), y_pred (np.ndarray), out_dir (str), normalize (bool)
# Outputs: Saves confusion matrix plot
# Description: Generates and saves confusion matrix as image.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def plot_confusion_matrix(y_true, y_pred, out_dir=ARTIFACT_DIR, normalize=True):
    ensure_dir(out_dir)
    cm = confusion_matrix(y_true, y_pred)
    if normalize:
        cm = cm.astype("float") / cm.sum(axis=1, keepdims=True)

    fig, ax = plt.subplots(figsize=(8.5, 7))
    im = ax.imshow(cm, interpolation="nearest", cmap="Blues")
    ax.figure.colorbar(im, ax=ax)
    ax.set(
        xticks=np.arange(len(DIGIT_CLASSES)),
        yticks=np.arange(len(DIGIT_CLASSES)),
        xticklabels=DIGIT_CLASSES,
        yticklabels=DIGIT_CLASSES,
        ylabel="True label",
        xlabel="Predicted label",
        title="Confusion Matrix (normalized)" if normalize else "Confusion Matrix",
    )
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")
    _annotate_confmat(ax, cm)
    fig.tight_layout()
    fig.savefig(os.path.join(out_dir, "confusion_matrix.png"))
    plt.close(fig)

#####
# Function: save_classification_report
# Inputs: y_true (np.ndarray), y_pred (np.ndarray), out_dir (str)
# Outputs: Saves classification report
# Description: Prints and saves classification report with precision, recall, and F1 score.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def save_classification_report(y_true, y_pred, out_dir=ARTIFACT_DIR):
    ensure_dir(out_dir)
```

```

report = classification_report(
    y_true, y_pred, target_names=DIGIT_CLASSES, digits=4
)
print(report)
with open(os.path.join(out_dir, "classification_report.txt"), "w") as f:
    f.write(report)

#####
# Function: show_misclassifications
# Inputs: x (np.ndarray), y_true (np.ndarray), y_pred (np.ndarray), limit (int), out_dir (str)
# Outputs: Saves misclassified images grid
# Description: Displays and saves examples of misclassified test images.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def show_misclassifications(x, y_true, y_pred, limit=25, out_dir=ARTIFACT_DIR):
    ensure_dir(out_dir)
    wrong = np.where(y_true != y_pred)[0]
    if len(wrong) == 0:
        print("No misclassifications")
        return
    sel = wrong[:limit]
    cols = 5
    rows = int(np.ceil(len(sel) / cols))
    plt.figure(figsize=(12, 2.6 * rows))
    for i, idx in enumerate(sel, 1):
        img = x[idx].squeeze()
        plt.subplot(rows, cols, i)
        plt.imshow(img, cmap="gray")
        plt.title(f"T:{DIGIT_CLASSES[y_true[idx]]}\nP:{DIGIT_CLASSES[y_pred[idx]]}", fontsize=9)
        plt.axis("off")
    plt.tight_layout()
    plt.savefig(os.path.join(out_dir, "misclassifications.png"))
    plt.close()

#####

# Function: save_label_map
# Inputs: out_dir (str)
# Outputs: Saves label mapping file
# Description: Saves numeric-to-class name mapping (0-9) for MNIST digits.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def save_label_map(out_dir=ARTIFACT_DIR):
    ensure_dir(out_dir)
    with open(os.path.join(out_dir, "label_map.txt"), "w") as f:
        for i, name in enumerate(DIGIT_CLASSES):
            f.write(f"{i}: {name}\n")

#####

# Function: save_summary
# Inputs: test_acc (float), test_loss (float), epochs (int), out_dir (str)

```

```

# Outputs: Saves summary text file
# Description: Saves final model performance summary and artifact list.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####
def save_summary(test_acc, test_loss, epochs, out_dir=ARTIFACT_DIR):
    ensure_dir(out_dir)
    with open(os.path.join(out_dir, "summary.txt"), "w") as f:
        f.write(
            "MNIST Digits CNN Summary\n"
            "=====\\n"
            f"Test Accuracy: {test_acc:.4f}\\n"
            f"Test Loss   : {test_loss:.4f}\\n"
            f"Epochs     : {epochs}\\n"
            "Artifacts   : acc_curve.png, loss_curve.png, confusion_matrix.png,\\n"
            "               misclassifications.png, classification_report.txt,\\n"
            "               label_map.txt, digits_cnn.h5, digits_cnn_final.h5\\n"
        )
    )

#####
# =====
# Data
# =====
#####

# Function: load_data
# Inputs: val_split (float) - validation split ratio
# Outputs: (train_data, val_data, test_data) tuples
# Description: Loads MNIST dataset, normalizes and splits into train, val, test sets.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def load_data(val_split=0.1) -> Tuple[Tuple[np.ndarray, np.ndarray],
                                         Tuple[np.ndarray, np.ndarray],
                                         Tuple[np.ndarray, np.ndarray]]:
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    x_train, x_val, y_train, y_val = train_test_split(
        x_train, y_train, test_size=val_split, random_state=SEED, stratify=y_train
    )
    # Scale and add channel dim
    x_train = (x_train.astype("float32") / 255.0)[..., None]
    x_val   = (x_val.astype("float32") / 255.0)[..., None]
    x_test  = (x_test.astype("float32") / 255.0)[..., None]
    return (x_train, y_train), (x_val, y_val), (x_test, y_test)

#####

# Function: load_flattened
# Inputs: None
# Outputs: Flattened train and test arrays with labels
# Description: Loads MNIST and flattens images for classical ML models.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def load_flattened():
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

```

```

x_train = x_train.reshape(len(x_train), -1).astype("float32") / 255.0
x_test = x_test.reshape(len(x_test), -1).astype("float32") / 255.0
return x_train, y_train, x_test, y_test

# =====
# Model
# =====
##### #####
# Function: build_cnn
# Inputs: lr (float) - learning rate
# Outputs: Compiled Keras CNN model
# Description: Builds and compiles CNN with augmentation, Conv2D, Dropout, BatchNorm, Dense.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
##### #####
def build_cnn(lr=1e-3) -> keras.Model:
    data_augmentation = keras.Sequential(
        [
            layers.RandomTranslation(0.05, 0.05, fill_mode="nearest"),
            layers.RandomRotation(0.05, fill_mode="nearest"),
            layers.RandomZoom(0.05, 0.05, fill_mode="nearest"),
        ],
        name="augmentation",
    )
    inputs = keras.Input(shape=(28, 28, 1))
    x = data_augmentation(inputs)

    x = layers.Conv2D(32, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(32, 3, padding="same", activation="relu")(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Dropout(0.25)(x)

    x = layers.Conv2D(64, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2D(64, 3, padding="same", activation="relu")(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Dropout(0.25)(x)

    x = layers.Flatten()(x)
    x = layers.Dense(128, activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(0.40)(x)

    outputs = layers.Dense(10, activation="softmax")(x)

    model = keras.Model(inputs, outputs, name="digits_cnn")
    opt = keras.optimizers.Adam(learning_rate=lr)
    model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model

# =====
# Train / Evaluate
# =====
#####

```

```

# Function: train_and_evaluate
# Inputs: batch_size (int), epochs (int), lr (float)
# Outputs: Trained model + saved artifacts
# Description: Trains CNN on MNIST, evaluates, saves curves, confusion matrix, misclassifications, report.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####
def train_and_evaluate(batch_size=128, epochs=10, lr=1e-3):
    ensure_dir(ARTIFACT_DIR)
    (x_train, y_train), (x_val, y_val), (x_test, y_test) = load_data(val_split=0.1)
    model = build_cnn(lr=lr)
    model.summary()

    callbacks = [
        keras.callbacks.ModelCheckpoint(BEST_MODEL, monitor="val_accuracy",
save_best_only=True, verbose=1),
        keras.callbacks.EarlyStopping(patience=3, restore_best_weights=True, monitor="val_loss"),
        keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=2, min_lr=1e-5, verbose=1),
    ]

    history = model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        batch_size=batch_size,
        epochs=epochs,
        callbacks=callbacks,
        verbose=2
    )

    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
    print(f"[TEST] loss: {test_loss:.4f} | acc: {test_acc:.4f}")

    model.save(FINAL_MODEL)

    # Metrics & artifacts
    plot_training_curves(history, ARTIFACT_DIR)
    y_pred = model.predict(x_test, batch_size=256).argmax(axis=1)
    plot_confusion_matrix(y_test, y_pred, ARTIFACT_DIR, normalize=True)
    save_classification_report(y_test, y_pred, ARTIFACT_DIR)
    show_misclassifications(x_test, y_test, y_pred, limit=25, out_dir=ARTIFACT_DIR)
    save_label_map(ARTIFACT_DIR)
    save_summary(test_acc, test_loss, len(history.history["loss"])), ARTIFACT_DIR)

#
# =====
# Inference
# =====
#####

# Function: inference_grid
# Inputs: n_samples (int), seed (int)
# Outputs: Saved inference grid image
# Description: Loads saved model, predicts on random test samples, saves predictions grid.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

```

```

def inference_grid(n_samples=9, seed=7):
    if not os.path.exists(BEST_MODEL):
        print(f"Could not find {BEST_MODEL}. Train first with --train.")
        return
    (_, _), (x_test, y_test) = keras.datasets.mnist.load_data()
    x_test = (x_test.astype("float32") / 255.0)[..., None]

    model = keras.models.load_model(BEST_MODEL)

    rng = np.random.default_rng(seed)
    idx = rng.choice(len(x_test), size=n_samples, replace=False)

    imgs = x_test[idx]
    labs = y_test[idx]
    preds = model.predict(imgs, verbose=0).argmax(axis=1)

    cols = int(np.ceil(np.sqrt(n_samples)))
    rows = int(np.ceil(n_samples / cols))
    plt.figure(figsize=(2.8 * cols, 2.8 * rows))
    for i in range(n_samples):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(imgs[i].squeeze(), cmap="gray")
        plt.title(f"P:{DIGIT_CLASSES[preds[i]]}\nT:{DIGIT_CLASSES[labs[i]]}", fontsize=9)
        plt.axis("off")
    plt.tight_layout()
    ensure_dir(ARTIFACT_DIR)
    out_path = os.path.join(ARTIFACT_DIR, "inference_grid.png")
    plt.savefig(out_path)
    plt.close()
    print("Saved:", out_path)

# =====
# Baselines
# =====
#####
# Function: run_baselines
# Inputs: None
# Outputs: Prints accuracy of LogReg, LinearSVC, RandomForest
# Description: Runs classical ML baselines on flattened MNIST digits.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
#####

def run_baselines():
    x_train, y_train, x_test, y_test = load_flattened()

    # Logistic Regression
    logreg = Pipeline([
        ("scaler", StandardScaler(with_mean=True)),
        ("clf", LogisticRegression(max_iter=200, n_jobs=-1))
    ])
    logreg.fit(x_train, y_train)
    print("LogReg acc:", accuracy_score(y_test, logreg.predict(x_test)))

    # Linear SVM
    svm = Pipeline([
        ("scaler", StandardScaler(with_mean=True)),
        ("clf", LinearSVC(C=1.0, dual=True, max_iter=5000))
    ])
  
```

```

])
svm.fit(x_train, y_train)
print("LinearSVC acc:", accuracy_score(y_test, svm.predict(x_test)))

# Random Forest
rf = RandomForestClassifier(n_estimators=200, max_depth=None, n_jobs=-1,
random_state=SEED)
rf.fit(x_train, y_train)
print("RandomForest acc:", accuracy_score(y_test, rf.predict(x_test)))

# =====
# CLI
# =====
##########
# Function: parse_args
# Inputs: None
# Outputs: Parsed CLI arguments
# Description: Defines command-line arguments for training, inference, baselines.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
##########

def parse_args():
  p = argparse.ArgumentParser(description="MNIST Digits Classification Case Study")
  p.add_argument("--train", action="store_true", help="Train the CNN and save artifacts.")
    p.add_argument("--infer", action="store_true", help="Generate an inference grid using saved
model.")
    p.add_argument("--samples", type=int, default=9, help="Number of samples for inference
grid.")
  p.add_argument("--baselines", action="store_true", help="Run classical ML baselines.")
  p.add_argument("--epochs", type=int, default=10, help="Epochs for CNN training.")
  p.add_argument("--batch", type=int, default=128, help="Batch size for CNN training.")
  p.add_argument("--lr", type=float, default=1e-3, help="Learning rate for Adam.")
  return p.parse_args()

# =====
# Entry point
# =====
##########

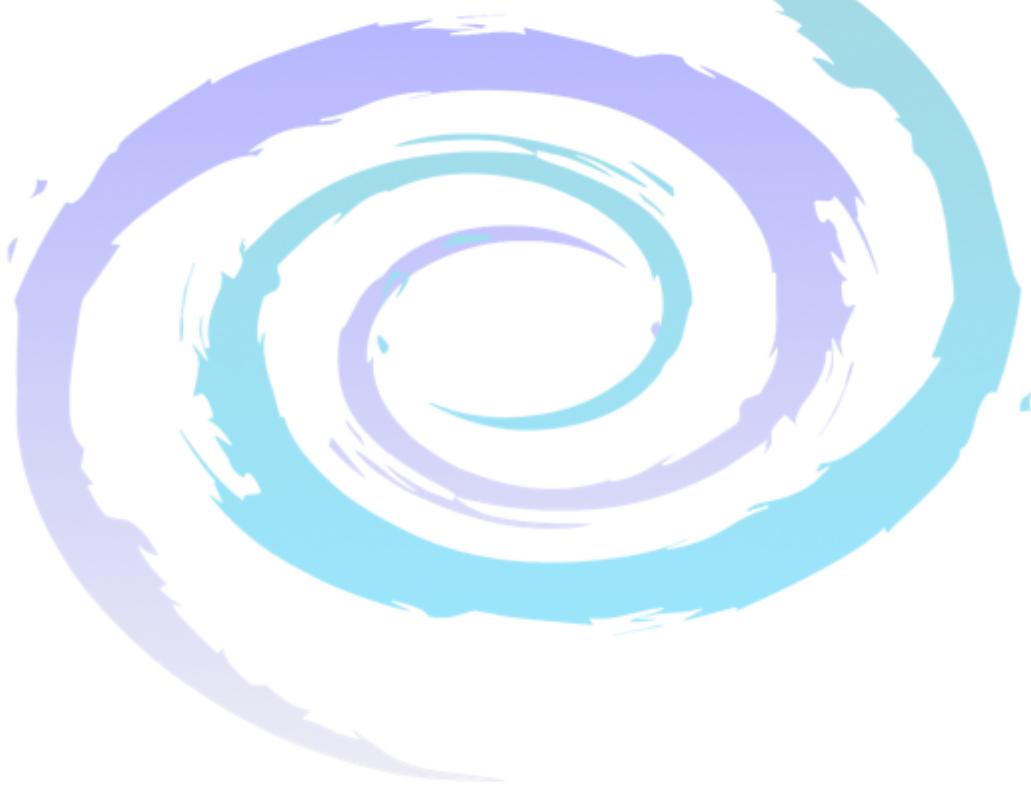
# Function: main
# Inputs: None
# Outputs: None
# Description: Entry point. Parses CLI args, runs train/infer/baselines accordingly.
# Author: Piyush Manohar Khairnar
# Date: 15/08/2025
##########

def main():
  set_seed(SEED)
  ensure_dir(ARTIFACT_DIR)
  args = parse_args()

  didAnything = False
  if args.train:
    train_and_evaluate(batch_size=args.batch, epochs=args.epochs, lr=args.lr)
    didAnything = True

```

```
if args.infer:  
    inference_grid(n_samples=args.samples)  
    didAnything = True  
if args.baselines:  
    runBaselines()  
    didAnything = True  
  
if not didAnything:  
    print(  
        "Nothing to do. Try one of:\n"  
        " python digits_classification_case_study.py --train\n"  
        " python digits_classification_case_study.py --infer --samples 9\n"  
        " python digits_classification_case_study.py --baselines\n"  
    )  
  
#####  
if __name__ == "__main__":  
    main()  
#####
```



Important Concepts of Machine Learning

Dataset :

A **dataset** is a structured collection of data used for analysis and training machine learning models. In ML, each **row** typically represents an example (sample or instance), and each **column** represents a **feature (input)** or **label (output)**.

Example (Iris Dataset Sample)

Sepal Length	Sepal Width	Petal Length	Petal Width	Species
5.1	3.5	1.4	0.2	Setosa

- **Features:** Sepal Length, Sepal Width, Petal Length, Petal Width
- **Label:** Species

Ways to Load Datasets in Python :

Method 1: Load Built-in Datasets (from sklearn)

```
from sklearn.datasets import load_iris
data = load_iris()
print(data.data)
```

Method 2: Load from a CSV file

```
import pandas as pd
df = pd.read_csv('iris.csv')
print(df.head())
```

Method 3: Load from a URL

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
iris/iris.data"
df = pd.read_csv(url)
print(df.head())
```

Features, Labels, Independent & Dependent Variables

Features (Input Variables)

- These are the inputs to the model.
- Also called **independent variables**.
- Example: Age, Height, Weight, Income

Labels (Target Variable)

- This is the output we are trying to predict.
- Also called **dependent variable**.
- Example: Disease (Yes/No), House Price, Species

Train-Test Splitting

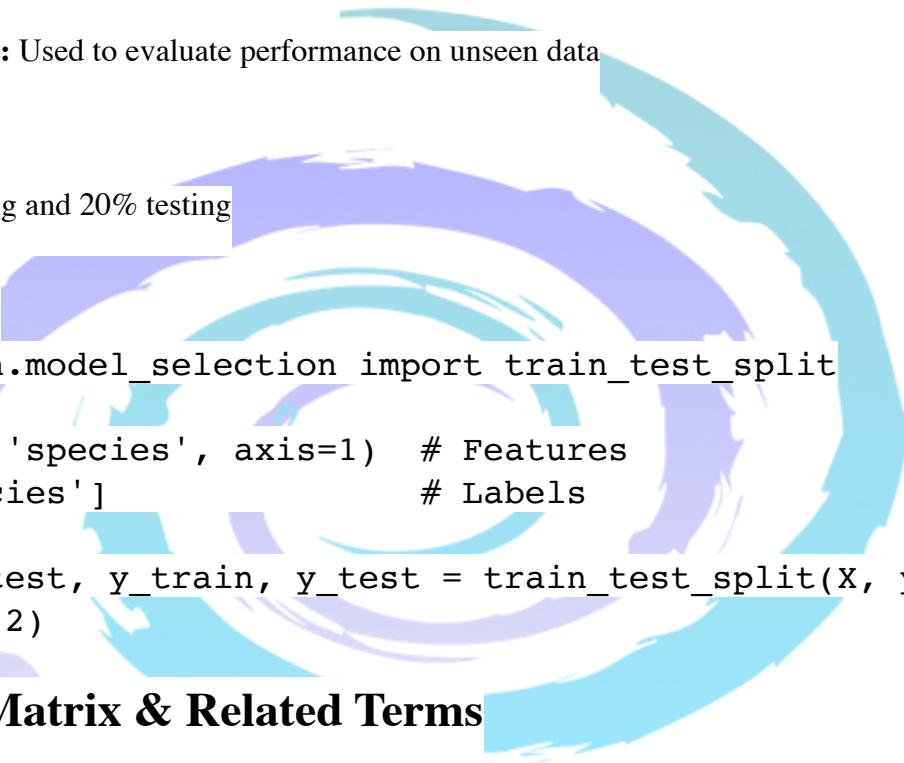
We split our dataset into **training** and **testing** parts to evaluate the model's performance.

- **Training Set:** Used to train the model
- **Testing Set:** Used to evaluate performance on unseen data

Typical Split:

- 80% training and 20% testing

Code Example:



```
from sklearn.model_selection import train_test_split

X = df.drop('species', axis=1) # Features
y = df['species'] # Labels

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)
```

Confusion Matrix & Related Terms

Used in classification problems to evaluate the performance of the model.

	Predicted: Positive	Predicted: Negative
Actual: Positive	True Positive (TP)	False Negative (FN)
Actual: Negative	False Positive (FP)	True Negative (TN)

Terms:

- **True Positive (TP):** Model correctly predicted Positive
- **True Negative (TN):** Model correctly predicted Negative
- **False Positive (FP):** Model incorrectly predicted Positive
- **False Negative (FN):** Model incorrectly predicted Negative

Accuracy and Other Evaluation Metrics

◆ Accuracy:

Measures overall correctness of the model.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

◆ Precision:

How many of the predicted positives are actually positive?

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

◆ Recall (Sensitivity):

How many actual positives were correctly predicted?

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

◆ F1 Score:

Harmonic mean of precision and recall

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Code Example: Confusion Matrix & Accuracy

```

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix

# 1. Load the Iris dataset
iris = load_iris()
X = iris.data      # Feature data (sepal & petal measurements)
y = iris.target    # Labels (species: 0, 1, 2)

# 2. Split the dataset (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

```

```
# 3. Create and train the Decision Tree model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# 4. Predict using the test set
y_pred = model.predict(X_test)

# 5. Evaluate performance
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Accuracy Score:", accuracy_score(y_test, y_pred))
```

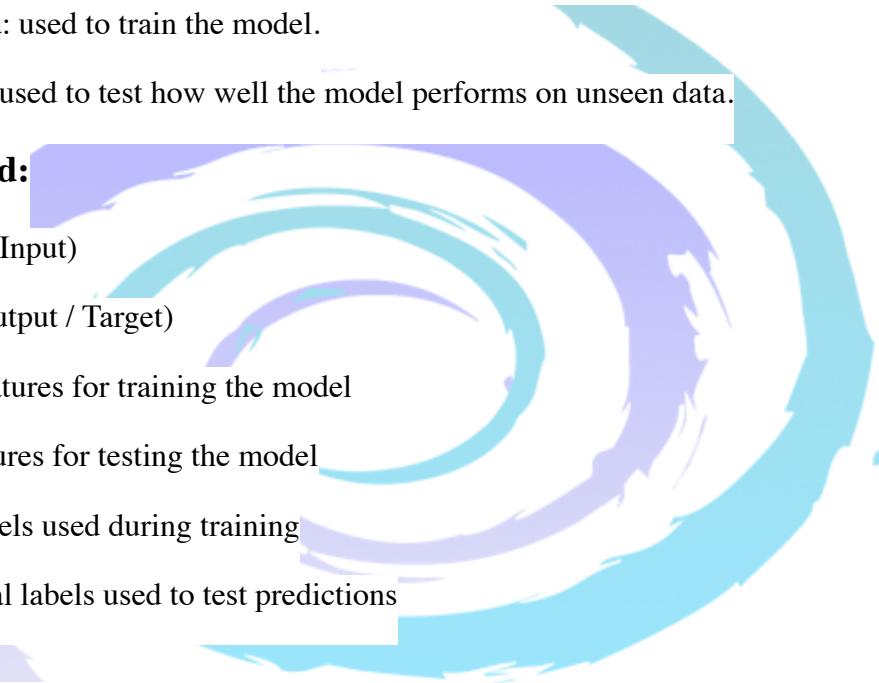
Train-Test Split in Iris Dataset

When using machine learning, it's important to divide the data into two parts:

- **Training data:** used to train the model.
- **Testing data:** used to test how well the model performs on unseen data.

Variables Involved:

- **X** = Features (Input)
- **y** = Labels (Output / Target)
- **X_train** = Features for training the model
- **X_test** = Features for testing the model
- **y_train** = Labels used during training
- **y_test** = Actual labels used to test predictions



Iris Dataset Example with Explanation:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load dataset
iris = load_iris()
X = iris.data      # All feature columns (sepal/petal length/width)

y = iris.target    # Labels (0 = Setosa, 1 = Versicolor, 2 =
Virginica)

# Split into train and test sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Print the shapes to verify
print("X_train shape:", X_train.shape) # e.g., (120, 4)
```

```
print("X_test shape:", X_test.shape)      # e.g., (30, 4)
print("y_train shape:", y_train.shape)    # e.g., (120,)
print("y_test shape:", y_test.shape)      # e.g., (30,)
```

Variable	Meaning
X_train	80% of feature data used for training
X_test	20% of feature data used for testing
y_train	Labels corresponding to X_train
y_test	Labels corresponding to X_test, used for evaluation



Industrial Way to handle Machine Learning Project

```
#####
# Required Python Packages
#####

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

from sklearn.metrics import (
    accuracy_score,
    confusion_matrix,
    roc_curve,
    auc,
    classification_report,
)

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
import joblib

#####
# File Paths
#####

INPUT_PATH = "breast-cancer-wisconsin.data"
OUTPUT_PATH = "breast-cancer-wisconsin.csv"
MODEL_PATH = "bc_rf_pipeline.joblib"

#####
# Headers
#####

HEADERS = [
    "CodeNumber", "ClumpThickness", "UniformityCellSize", "UniformityCellShape",
    "MarginalAdhesion",
    "SingleEpithelialCellSize", "BareNuclei", "BlandChromatin", "NormalNucleoli", "Mitoses",
    "CancerType"
]
```

```
#####
# Function name :      read_data
# Description :      Read the data into pandas dataframe
# Inpt :            path of CSV file
# Output :           Gives the data
# Author :          Piyush Manohar Khairnar
# Date :            09/08/2025
#####

def read_data(path):
    """Read the data into pandas dataframe"""
    data = pd.read_csv(path, header=None)
    return data

#####
# Function name :      get_headers
# Description :      dataset headers
# Input :            dataset
# Output :           Returns the header
# Author :          Piyush Manohar Khairnar
# Date :            09/08/2025
#####

def get_headers(dataset):
    """Return dataset headers"""
    return dataset.columns.values

#####
# Function name :      add_headers
# Description :      Add the headers to the dataset
# Input :            dataset
# Output :           Updated dataset
# Author :          Piyush Manohar Khairnar
# Date :            09/08/2025
#####

def add_headers(dataset, headers):
    """Add headers to dataset"""
    dataset.columns = headers
    return dataset
```

```
#####
# Function name :      data_file_to_csv
# Input :              Nothing
# Output :             Write the data to CSV
# Author :             Piyush Manohar Khairnar
# Date :               09/08/2025
#####

def data_file_to_csv():
    """Convert raw .data file to CSV with headers"""
    dataset = read_data(INPUT_PATH)
    dataset = add_headers(dataset, HEADERS)
    dataset.to_csv(OUTPUT_PATH, index=False)
    print("File saved ...!")

#####

# Function name :      handle_missing_values
# Description :        Filter missing values from the dataset
# Input :              Dataset with mising values
# Output :             Dataset by remocing missing values
# Author :             Piyush Manohar Khairnar
# Date :               09/08/2025
#####

def handle_missing_values_with_imputer(df, feature_headers):
    """
    Convert '?' to NaN and let SimpleImputer handle them inside the Pipeline.
    Keep only numeric columns in features.
    """

    # Replace '?' in the whole dataframe
    df = df.replace('?', np.nan)

    # Cast features to numeric
    df[feature_headers] = df[feature_headers].apply(pd.to_numeric, errors='coerce')

    return df
```

```
#####
# Function name :      split_dataset
# Description :      Split the dataset with train_percentage
# Input :            Dataset with related information
# Output :           Dataset after splitting
# Author :           Piyush Manohar Khairnar
# Date :             09/08/2025
#####

def split_dataset(dataset, train_percentage, feature_headers, target_header,
random_state=42):
    """Split dataset into train/test"""
    train_x, test_x, train_y, test_y = train_test_split(
        dataset[feature_headers], dataset[target_header],
        train_size=train_percentage, random_state=random_state,
stratify=dataset[target_header]
    )

    return train_x, test_x, train_y, test_y

#####

# Function name :      dataset_statistics
# Description :      Display the statistics
# Author :           Piyush Manohar Khairnar
# Date :             09/08/2025
#####

def dataset_statistics(dataset):
    """Print basic stats"""
    print(dataset.describe(include='all'))

#####

# Function name :      build_pipeline
# Description :      Build a Pipeline:
# SimpleImputer:    replace missing with median
#                   RandomForestClassifier: robust baseline
# Author :           Piyush Manohar Khairnar
# Date :             09/08/2025
#####

def build_pipeline():

    pipe = Pipeline(steps=[
        ("imputer", SimpleImputer(strategy="median")),
        ("rf", RandomForestClassifier(
            n_estimators=300,
            random_state=42,
            n_jobs=-1,
```

```

    class_weight=None
  ))
])
return pipe

#####
# Function name :      train_pipeline
# Description :      Train a Pipeline:
# Author :            Piyush Manohar Khairnar
# Date :              09/08/2025
#####

def train_pipeline(pipeline, X_train, y_train):
  pipeline.fit(X_train, y_train)
  return pipeline

#####
# Function name :      save_model
# Description :      Save the model
# Author :            Piyush Manohar Khairnar
# Date :              09/08/2025
#####

def save_model(model, path=MODEL_PATH):
  joblib.dump(model, path)
  print(f"Model saved to {path}")

#####
# Function name :      load_model
# Description :      Load the trained model
# Author :            Piyush Manohar Khairnar
# Date :              09/08/2025
#####

def load_model(path=MODEL_PATH):
  model = joblib.load(path)
  print(f"Model loaded from {path}")
  return model

```

```
#####
# Function name :      plot_confusion_matrix_matshow
# Description :       Display Confusion Matrix
# Author :           Piyush Manohar Khairnar
# Date :            09/08/2025
#####

def plot_confusion_matrix_matshow(y_true, y_pred, title="Confusion Matrix"):
    cm = confusion_matrix(y_true, y_pred)
    fig, ax = plt.subplots()
    cax = ax.matshow(cm)
    fig.colorbar(cax)
    for (i, j), v in np.ndenumerate(cm):
        ax.text(j, i, str(v), ha='center', va='center')
    ax.set_xlabel("Predicted")
    ax.set_ylabel("Actual")
    ax.set_title(title)
    plt.tight_layout()
    plt.show()

#####

# Function name :      plot_feature_importances
# Description :       Display the fUTURE importance
# Author :           Piyush Manohar Khairnar
# Date :            09/08/2025
#####

def plot_feature_importances(model, feature_names, title="Feature Importances (Random Forest)"):
    if hasattr(model, "named_steps") and "rf" in model.named_steps:
        rf = model.named_steps["rf"]
        importances = rf.feature_importances_
    elif hasattr(model, "feature_importances_"):
        importances = model.feature_importances_
    else:
        print("Feature importances not available for this model.")
        return

    idx = np.argsort(importances)[::-1]
    plt.figure(figsize=(8, 4))
    plt.bar(range(len(importances)), importances[idx])
    plt.xticks(range(len(importances)), [feature_names[i] for i in idx], rotation=45, ha='right')
    plt.ylabel("Importance")
    plt.title(title)
    plt.tight_layout()
    plt.show()
```

```
#####
# Function name :      main
# Description :      Main function from where execution starts
# Author :            Piyush Manohar Khairnar
# Date :              09/08/2025
#####

def main():
    # 1) Ensure CSV exists (run once if needed)
    # data_file_to_csv()

    # 2) Load CSV
    dataset = pd.read_csv(OUTPUT_PATH)

    # 3) Basic stats
    dataset_statistics(dataset)

    # 4) Prepare features/target
    feature_headers = HEADERS[1:-1]  # drop CodeNumber, keep all features
    target_header = HEADERS[-1]       # CancerType (2=benign, 4=malignant)

    # 5) Handle '?' and coerce to numeric; imputation will happen inside Pipeline
    dataset = handle_missing_values_with_imputer(dataset, feature_headers)

    # 6) Split
    train_x, test_x, train_y, test_y = split_dataset(dataset, 0.7, feature_headers, target_header)

    print("Train_x Shape :: ", train_x.shape)
    print("Train_y Shape :: ", train_y.shape)
    print("Test_x Shape :: ", test_x.shape)
    print("Test_y Shape :: ", test_y.shape)

    # 7) Build + Train Pipeline
    pipeline = build_pipeline()
    trained_model = train_pipeline(pipeline, train_x, train_y)
    print("Trained Pipeline :: ", trained_model)

    # 8) Predictions
    predictions = trained_model.predict(test_x)

    # 9) Metrics
    print("Train Accuracy :: ", accuracy_score(train_y, trained_model.predict(train_x)))
    print("Test Accuracy :: ", accuracy_score(test_y, predictions))
    print("Classification Report:\n", classification_report(test_y, predictions))
    print("Confusion Matrix:\n", confusion_matrix(test_y, predictions))
```

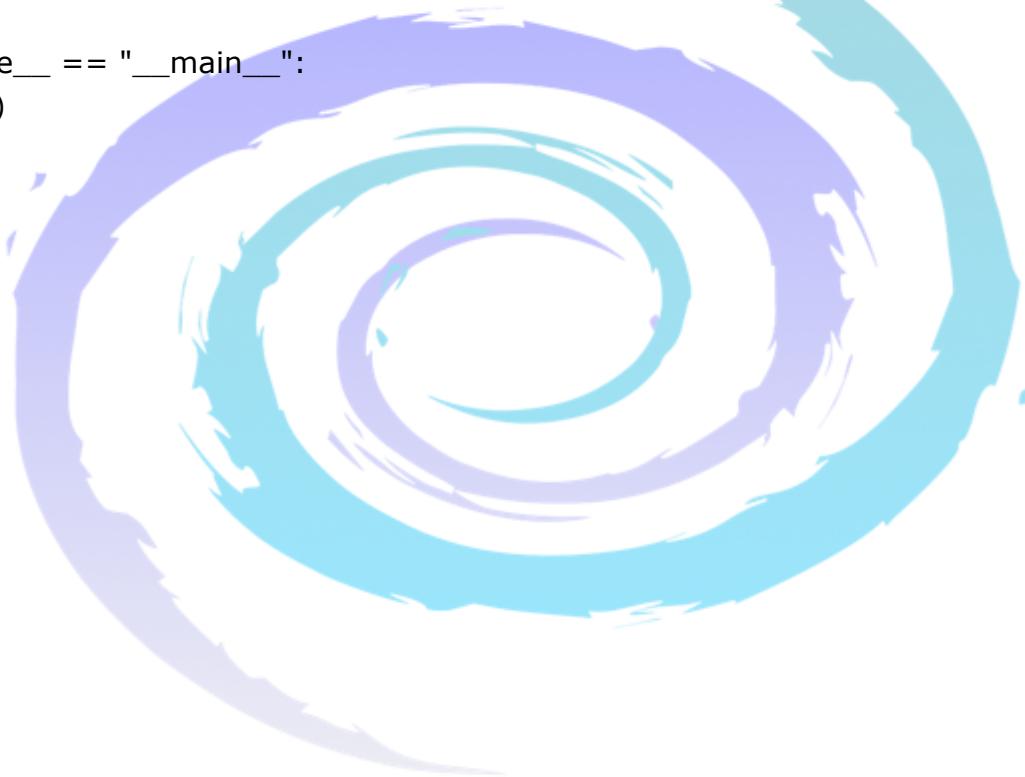
```
# Feature importances (tree-based)
plot_feature_importances(trained_model, feature_headers, title="Feature Importances (RF)")

# 10) Save model (Pipeline) using joblib
save_model(trained_model, MODEL_PATH)

# 11) Load model and test a sample
loaded = load_model(MODEL_PATH)
sample = test_x.iloc[[0]]
pred_loaded = loaded.predict(sample)
print(f"Loaded model prediction for first test sample: {pred_loaded[0]}")

#####
# Application starter
#####

if __name__ == "__main__":
    main()
```



Your Name
Your Address in Short
Mobile: +91- ----- Email: -----@gmail.com

Career Objective

Aspiring Python Developer & Machine Learning Engineer with expertise in automation, data science, and deep learning. Skilled in designing intelligent systems, building automation scripts, and applying ML/DL models for real-world problem solving. Seeking opportunities to contribute to impactful projects using modern AI technologies.

Educational Qualification :

Standard	Institute	Board / University	Percentage	Class
MCS				
BCS				
HSC				
SSC				

Technical Skills :

Programming Paradigms & Languages

- **Procedural Programming:** C
- **Object-Oriented Programming (OOP):** C++, Python
- **Virtual Machine-based Programming:** Java, C# .NET
- **Scripting Languages:** PHP, JavaScript

Domain Expertise

- **Python Libraries & Tools:** NumPy, Pandas, Matplotlib, Scikit-learn, TensorFlow, Keras, OpenCV
- **Machine Learning:** Regression, Classification, Clustering, Feature Engineering, Model Evaluation
- **Deep Learning:** ANN, CNN, RNN, Computer Vision, NLP
- **Automation Tools:** OS, Scripting, Email Automation (smtplib), Scheduling, Logging
- **GenAI & LLMs:** Hugging Face Transformers, GPT/BERT, DALL-E
- **Databases:** MySQL, MongoDB
- **Version Control:** Git
- **Development Environment:** Anaconda, Jupyter Notebook, VS Code

Python Automation Project

Duplicate File Cleaner & Log Automation

Description: Developed a Python-based automation script to detect and delete duplicate files from a directory periodically. The system automatically generates log files of operations performed and shares them via email for audit purposes.

- Implemented checksum-based duplicate detection using hashlib (MD5).
- Automated log generation with timestamps for every execution.
- Used schedule library for periodic execution of duplicate file cleanup.
- Integrated email automation (smtplib) to send logs automatically.

Tech Stack: Python, OS module, Hashlib, Schedule, smtplib

 **GitHub Repository:** github.com/username/-----

System Process Logger with Scheduling

Description: Designed a Python automation project to periodically log details of running processes (PID, name, user, memory usage) on the system. Each execution generates a new log file with a timestamped filename for easy tracking.

- Implemented process scanning using psutil to extract process details.
- Automated log file creation with timestamps and proper formatting.
- Used the schedule library to run logging tasks periodically without manual intervention.
- Enhanced usability by allowing custom folder name & interval input.

Tech Stack: Python, psutil, OS, Schedule, Time**GitHub Repository:** [github.com/username/-----](https://github.com/username/)

Machine Learning Projects

Worked on real-world ML projects covering classification, regression, and clustering.

- **Titanic Survival Predictor** – Built a classification model to predict passenger survival using Decision Trees & Logistic Regression.
- **Diabetes Detection** – Developed a health classification system using medical data to detect diabetes.
- Breast Cancer Detection – Achieved high accuracy using Logistic Regression & Support Vector Machine (SVM).
- **Ad Click Predictor** – Classification model predicting probability of user clicks on advertisements.
- **Wine Classifier** – Multi-class classification of wine type using KNN & Random Forest.
- **Head Brain Classification** – Built regression/classification models to analyze the correlation between brain size and intelligence.
- **House Price Prediction** – Regression-based project to predict property prices using multiple features.
- **Iris Classifier** – Implemented classic ML classification on the Iris dataset using Decision Tree & KNN.
- **Advertisement Predictor** – Classification system predicting advertisement response probability.
- **Customer Segmentation (Unsupervised – K-Means Clustering)**
Grouped retail customers into distinct clusters based on purchasing behavior for targeted marketing.
- Movie Recommendation System (Unsupervised – K-Means)
Built a recommendation engine using clustering & similarity measures.
- **Loan Default Prediction (Ensemble – Random Forest & Gradient Boosting)**
Boosted model accuracy by combining multiple classifiers to predict loan repayment defaults.
- **Sentiment Analysis Enhancement (Ensemble – Bagging & Boosting)**
Improved sentiment classification accuracy by combining weak learners into a strong predictive model.

Tech Stack: Python, Pandas, Scikit-learn, NumPy, Matplotlib**GitHub Repository:** [github.com/username/-----](https://github.com/username/)

Deep Learning Projects

Image Classification using CNN (MNIST Dataset)

Description: Built and trained a Convolutional Neural Network (CNN) model to classify handwritten digits (0–9) from the MNIST dataset. Achieved >98% accuracy through optimized architecture and hyperparameter tuning.

- Preprocessed dataset with normalization & reshaping for efficient training.
- Designed CNN with convolutional, pooling, dropout, and fully connected layers for feature extraction and classification.
- Applied ReLU activation for non-linearity and Softmax for multi-class output.
- Implemented Adam optimizer and categorical cross-entropy loss to improve convergence.
- Evaluated model using accuracy score, confusion matrix, and loss curves.
- Visualized predictions with Matplotlib to compare correctly vs. misclassified digits.

Tech Stack: Python, TensorFlow, Keras, NumPy, Matplotlib**GitHub Repository:** [github.com/username/-----](https://github.com/username/)

Real-time Emotion Detection (CNN + OpenCV)

Description: Developed a real-time facial emotion recognition system using Convolutional Neural Networks (CNNs) integrated with OpenCV for live video capture. The model successfully detects and classifies human emotions such as happy, sad, angry, surprised, and neutral from webcam input.

- Preprocessed facial images using grayscale conversion, resizing, and normalization for consistent model input.
- Designed and trained a CNN architecture with convolutional, pooling, dropout, and dense layers for robust emotion classification.
- Implemented Softmax activation to classify multiple emotion categories.
- Integrated OpenCV Haar Cascade & DNN face detectors to capture real-time faces from live webcam stream.
- Displayed emotion labels on video frames with bounding boxes for clear visualization.
- Achieved high accuracy on validation data and optimized inference speed for real-time deployment.

Tech Stack: Python, TensorFlow, Keras, OpenCV, NumPy, Matplotlib

 **GitHub Repository:** github.com/username/-----

Text Sentiment Analysis using RNN/LSTM

Description: Built a Recurrent Neural Network (RNN) with LSTM layers to predict sentiment (positive/negative/neutral) from text data such as movie reviews and social media posts. The model effectively captured long-term dependencies in text for improved sentiment classification.

- Collected and preprocessed textual dataset with tokenization, stopword removal, padding, and embedding.
- Used Word Embeddings (Word2Vec/Glove/Embedding layer) to represent words in dense vector space.
- Designed an LSTM-based deep learning model capable of handling sequential dependencies in text.
- Implemented dropout layers to prevent overfitting and improve generalization.
- Trained the model with Adam optimizer & binary/multiclass cross-entropy loss.

Tech Stack: Python, TensorFlow, Keras, NLTK, NumPy, Matplotlib

 **GitHub Repository:** github.com/username/-----

Generative AI Projects

FLAN-T5 Summarizer & Q&A Assistant

Description : Built a CPU-friendly CLI app that performs abstractive text summarization and context-aware Q&A using google/flan-t5-small—ideal for classroom demos and quick local workflows (no training required).

- Implemented two modes: Summarize text and Q&A from local notes (context.txt) with a strict "Not found" fallback if answers aren't in context.
- Used Hugging Face Transformers (Seq2Seq) with clean prompt engineering, safe generation (top-p, temperature), and length controls.
- Designed a simple, cross-platform CLI UX with clear instructions and modular code for students.
- Runs fully offline on CPU (Torch), no external APIs; fast first-token latency for short inputs.
- Demonstrates practical LLM skills: tokenization (SentencePiece), decoding, inference, and context grounding without fine-tuning.

Tech Stack: Python, Hugging Face Transformers, PyTorch, FLAN-T5, SentencePiece

 **GitHub Repository:** github.com/username/-----

Achievements

- Completed Python – Machine Learning with Automations, Deep Learning & GenAI training with multiple projects in Marvellous Infosystems.
- Built strong GitHub portfolio with 20+ AI/ML projects.
- Conducted hands-on coding in real-time case studies and automation tasks.

Work history : (If applicable)

Experience : ____ Years

Company name : -----

Domain Knowledge : -----

Responsibilities : -----

Profiles

- GitHub: github.com/username
- LinkedIn: linkedin.com/in/username

Personal Information:

- Date of Birth:
- Father's Name:
- Marital Status:
- Nationality:

The above mentioned information is authentic to the best of my knowledge.

Your Name

FLAN-T5 = Fine-Tuned Language Net + T5

1. T5 (Text-to-Text Transfer Transformer)

- Introduced by Google in 2019 (“Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”).
- **Encoder–Decoder (Seq2Seq) Transformer** → unlike GPT (decoder-only) or BERT (encoder-only).
- Treats **every NLP task as text-to-text**:
 - Input: "translate English to German: The book is on the table."
 - Output: "Das Buch liegt auf dem Tisch."
- Can handle summarization, translation, classification, QA, etc.

2. FLAN (Fine-tuned Language Net)

- In 2022, Google released **FLAN-T5**: T5 that was **instruction-tuned** on a very large collection of **instructions + tasks**.
- Instead of just pretraining, FLAN-T5 was fine-tuned to **follow natural language instructions** like:
 - “Summarize the following paragraph...”
 - “Translate this sentence into French...”
 - “Classify the sentiment of this review...”

GENAI Project

FLAN-T5 Summarizer & Q&A Assistant

```
#####
##### Marvellous FLAN-T5 Model
#####

import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"

# Import from Hugging Face Transformers
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

# AutoTokenizer : A tokenizer loader that automatically picks the right tokenizer for the model you choose.
# AutoModelForSeq2SeqLM : A pre-trained model loader for Sequence-to-Sequence Language Models (Seq2SeqLM).

# Choose a instruction-tuned model
MODEL_NAME = "google/flan-t5-small"
# A lightweight version of FLAN-T5.
# About 80 million parameters.

print(f"Marvellous FLAN-T5_Summarizer_Q&A_Assistant {MODEL_NAME} model loading...")

# Load tokenizer (handles text <-> tokens).
# AutoTokenizer picks the right tokenizer for the model.
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)

# Load the sequence-to-sequence model
model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)
```

```
#####
##### This function takes a text prompt (string) uses a FLAN-T5 model to generate a
# continuation/answer and returns the generated text.
#
#####
##### This function takes a text prompt (string) uses a FLAN-T5 model to generate a
# continuation/answer and returns the generated text.
#
#####

def Marvellous_run_flan(prompt: str, max_new_tokens: int = 128) -> str:

    # Tokenisation

    # Tokenize the input prompt; return PyTorch tensors; truncate if too long
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True)

    # Generation

    # Generate text from the model with light sampling for naturalness
    outputs = model.generate(
        **inputs,                      # pass tokenized inputs (input_ids, attention_mask)
        max_new_tokens=max_new_tokens, # how many new tokens to generate
        do_sample=True,                # enable random sampling
        top_p=0.9,                     # nucleus sampling: only consider tokens in the top 90% probability mass
        temperature=0.7               # control randomness (lower = safer/more deterministic)
    )

    # Decode token IDs back into a clean string
    # Example IDs: [71, 867, 1234, 42, 1]
    # Text:      "Hello, how are you?"

    return tokenizer.decode(outputs[0], skip_special_tokens=True).strip()

#####
##### This function takes a text prompt (string) uses a FLAN-T5 model to generate a
# continuation/answer and returns the generated text.
#
#####

# This function is used for summarisation
```

It creates a prompt with 4 to 6 bullet points

#

```
def Marvellous_summarize_text(text: str) -> str:
```

Prompt template instructing the model to produce 4-6 bullet points

```
prompt = f"Summarize the following text in 4-6 bullet points:\n\n{text}"
```

```
# Allow a slightly longer output for bullet lists
```

```
return Marvellous_run_flan(prompt, max_new_tokens=160)
```

#

```
# This function is used to load the contents from our local file
```

```
# And return the complete file contents in one string
```

#

#####

```
def Marvellous_load_context(path: str = "context.txt") -> str:
```

try:

```
# Read the entire file as a single string
```

```
with open(path, "r", encoding="Utf-8") as f:
```

```
return f.read()
```

```
except FileNotFoundError:
```

```
return ""
```

#

This function ask FLAN to answer using ONLY the given context.

If answer isn't present, ask it to say 'Not found'.

#

```

#####
##### def Marvellous_answer_from_context(question: str, context: str) -> str:
#####

if not context.strip():
    return "Context file not found or empty. Create 'context.txt' first."


# Construct a strict prompt for FLAN-T5
prompt = (
    "You are a helpful assistant. Answer the question ONLY using the context.\n"
    "If the answer is not in the context, reply exactly: Not found.\n\n"
    f"Context:\n{context}\n\n"
    f"Question: {question}\nAnswer:"
)

# Generate a concise answer grounded in the provided notes
return Marvellous_run_flan(prompt, max_new_tokens=120)
##### def main():
#####

# Entry point function
#
#####
##### def main():
#####

print("-----")
print("\n----- Marvellous FLAN-T5 Model -----")
print("1. Summarize the data")
print("2. Questions & Answers over local context.txt")
print("0. Exit")
print("-----")

```

```
while True:
```

```
choice = input("\nChoose an option (1/2/0): ").strip()
```

```
if choice == "0":
```

```
  print("Thank you for using Marvellous FLAN-T5 Model")
```

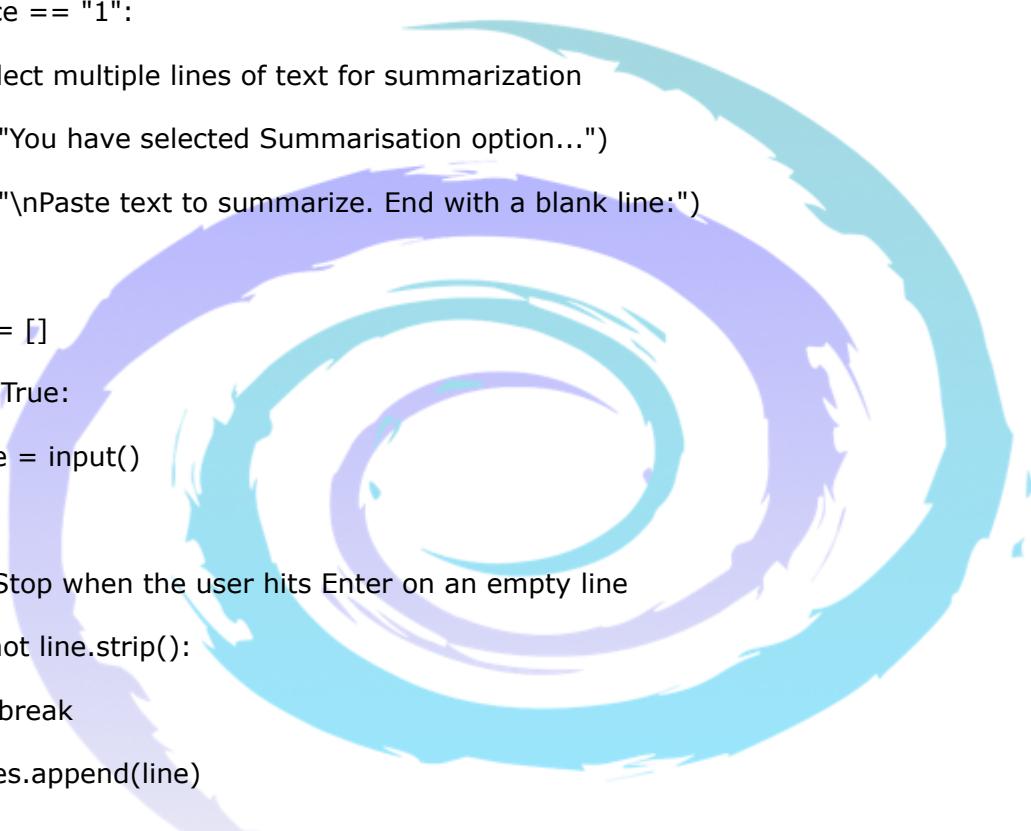
```
  break
```

```
elif choice == "1":
```

```
  # Collect multiple lines of text for summarization
```

```
  print("You have selected Summarisation option...")
```

```
  print("\nPaste text to summarize. End with a blank line:")
```



```
lines = []
```

```
while True:
```

```
  line = input()
```

```
  # Stop when the user hits Enter on an empty line
```

```
  if not line.strip():
```

```
    break
```

```
  lines.append(line)
```

```
# Join lines into a single block of text
```

```
text = "\n".join(lines).strip()
```

```
# If no text was provided, prompt again
```

```
if not text:
```

```
  print("Marvellous FLAN-T5 says : No text received.")
```

```
  continue
```

```
# Run summarization and print the result
```

```
print("\nSummary generated by Marvellous FLAN model : ")
```

```
print(Marvellous_summarize_text(text))
```

```

elif choice == "2":
    # Load context from local file 'context.txt'
    ctx = Marvellous_load_context("context.txt")

if not ctx.strip():
    # Help the user if the context is missing/empty
    print("Missing 'context.txt'. Create it in the same folder and try again.")
    continue

    # Ask a question related to the provided context
    q = input("\nAsk a question about your context to Marvellous FLAN model : ").strip()
    if not q:
        print("No question received.")
        continue

    # Generate an answer grounded only in the context
    print("\nAnswer from Marvellous FLAN model : ")
    print(Marvellous_answer_from_context(q, ctx))

else:
    print("Please choose 1, 2, or 0.")

#####
# Starter
#
#####

if __name__ == "__main__":
    main()

```

Python Interview Questions Part 1

1. What is Python? What are the benefits of using Python?

Python is a programming language with objects, modules, threads, exceptions and automatic memory management. The benefits of pythons are that it is simple and easy, portable, extensible, build-in data structure and it is an open source.

2. What is pickling and unpickling?

Pickle module accepts any Python object and converts it into a string representation and dumps it into a file by using dump function, this process is called pickling. While the process of retrieving original Python objects from the stored string representation is called unpickling.

3. How Python is interpreted?

Python language is an interpreted language. Python program runs directly from the source code. It converts the source code that is written by the programmer into an intermediate language, which is again translated into machine language that has to be executed.

4. How memory is managed in Python?

- Python memory is managed by Python private heap space. All Python objects and data structures are located in a private heap. The programmer does not have an access to this private heap and interpreter takes care of this Python private heap.
- The allocation of Python heap space for Python objects is done by Python memory manager. The core API gives access to some tools for the programmer to code.
- Python also have an inbuilt garbage collector, which recycle all the unused memory and frees the memory and makes it available to the heap space.

5. What are the tools that help to find bugs or perform static analysis?

PyChecker is a static analysis tool that detects the bugs in Python source code and warns about the style and complexity of the bug. Pylint is another tool that verifies whether the module meets the coding standard.

6. What are Python decorators?

A Python decorator is a specific change that we make in Python syntax to alter functions easily.

7. What is the difference between list and tuple?

The difference between list and tuple is that list is mutable while tuple is not. Tuple can be hashed for e.g as a key for dictionaries.

8. How are arguments passed by value or by reference?

Everything in Python is an object and all variables hold references to the objects. The references values are according to the functions; as a result you cannot

change the value of the references. However, you can change the objects if it is mutable.

9. What is Dict and List comprehensions are?

They are syntax constructions to ease the creation of a Dictionary or List based on existing iterable.

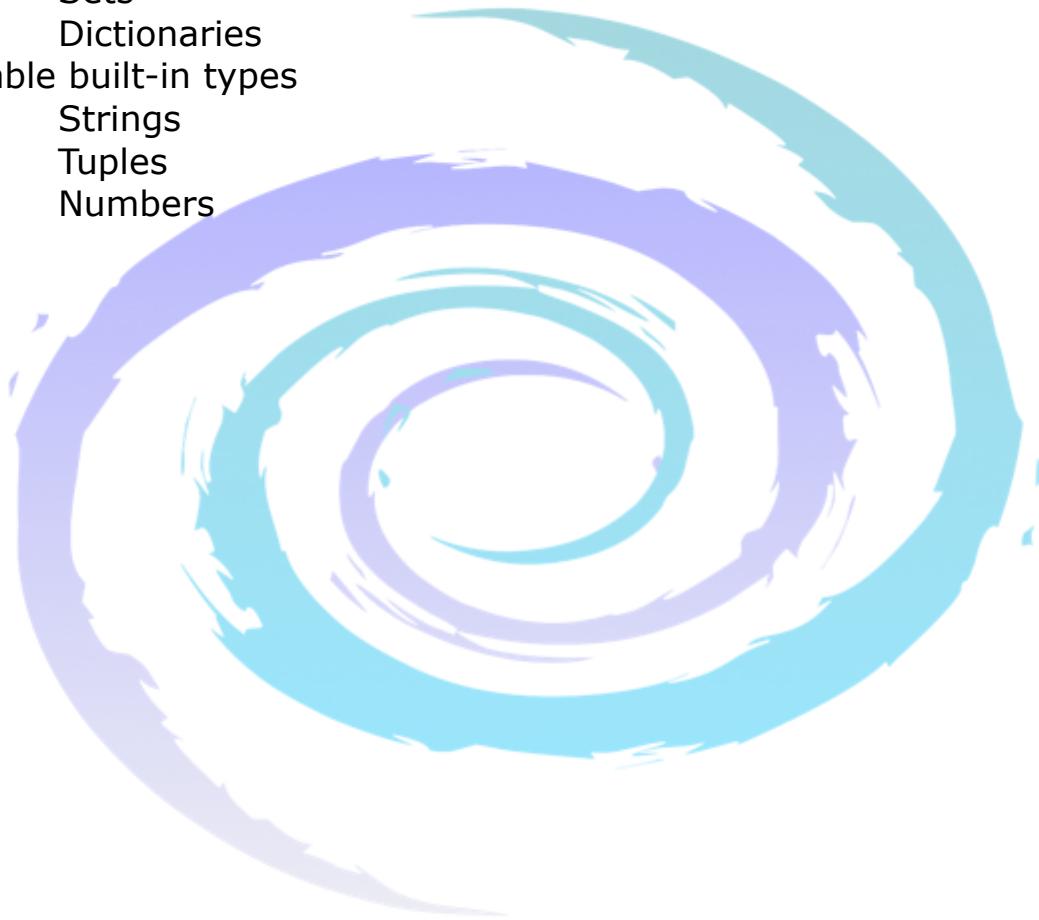
10. What are the built-in type does python provides?

There are mutable and Immutable types of Pythons built in types Mutable built-in types

- List
- Sets
- Dictionaries

Immutable built-in types

- Strings
- Tuples
- Numbers



Python Interview Questions Part 2

1. What is namespace in Python?

In Python, every name introduced has a place where it lives and can be hooked for. This is known as namespace. It is like a box where a variable name is mapped to the object placed. Whenever the variable is searched out, this box will be searched, to get corresponding object.

2. What is lambda in Python?

It is a single expression anonymous function often used as inline function.

3. Why lambda forms in python does not have statements?

A lambda form in python does not have statements as it is used to make new function object and then return them at runtime.

4. What is pass in Python?

Pass means, no-operation Python statement, or in other words it is a place holder in compound statement, where there should be a blank left and nothing has to be written there.

5. In Python what are iterators?

In Python, iterators are used to iterate a group of elements, containers like list.

6. What is unittest in Python?

A unit testing framework in Python is known as unittest. It supports sharing of setups, automation testing, shutdown code for tests, aggregation of tests into collections etc.

7. In Python what is slicing?

A mechanism to select a range of items from sequence types like list, tuple, strings etc. is known as slicing.

8. What are generators in Python?

The way of implementing iterators are known as generators. It is a normal function except that it yields expression in the function.

9. What is docstring in Python?

A Python documentation string is known as docstring, it is a way of documenting Python functions, modules and classes.

10. How can you copy an object in Python?

To copy an object in Python, you can try `copy.copy()` or `copy.deepcopy()` for the general case. You cannot copy all objects but most of them.

Python Interview Questions Part 3

1. What is negative index in Python?

Python sequences can be indexed in positive and negative numbers. For positive index, 0 is the first index, 1 is the second index and so forth. For negative index, (-1) is the last index and (-2) is the second last index and so forth.

2. How you can convert a number to a string?

In order to convert a number into a string, use the inbuilt function `str()`. If you want a octal or hexadecimal representation, use the inbuilt function `oct()` or `hex()`.

3. What is the difference between Xrange and range?

Xrange returns the xrange object while range returns the list, and uses the same memory and no matter what the range size is.

4. What is module and package in Python?

In Python, module is the way to structure program. Each Python program file is a module, which imports other modules like objects and attributes.

The folder of Python program is a package of modules. A package can have modules or subfolders.

5. Mention what are the rules for local and global variables in Python?

Local variables: If a variable is assigned a new value anywhere within the function's body, it's assumed to be local.

Global variables: Those variables that are only referenced inside a function are implicitly global.

6. How can you share global variables across modules?

To share global variables across modules within a single program, create a special module. Import the config module in all modules of your application. The module will be available as a global variable across modules.

7. Explain how can you make a Python Script executable on Unix?

To make a Python Script executable on Unix, you need to do two things,

- Script file's mode must be executable and
- the first line must begin with # (#!/usr/local/bin/python)

8. Explain how to delete a file in Python?

By using a command `os.remove (filename)` or `os.unlink(filename)`

9. Explain how can you generate random numbers in Python?

To generate random numbers in Python, you need to import command as

`import random`

`random.random()`

This returns a random floating point number in the range [0,1)

10. Explain how can you access a module written in Python from C?
You can access a module written in Python from C by following method,
Module = PyImport_ImportModule("<modulename>");



Python Interview Questions Part 4

1. Mention the use of // operator in Python?

It is a Floor Division operator , which is used for dividing two operands with the result as quotient showing only digits before the decimal point. For instance, $10//5 = 2$ and $10.0//5.0 = 2.0$.

2. Mention five benefits of using Python?

- Python comprises of a huge standard library for most Internet platforms like Email, HTML, etc.
- Python does not require explicit memory management as the interpreter itself allocates the memory to new variables and free them automatically
- Provide easy readability due to use of square brackets
- Easy-to-learn for beginners
- Having the built-in data types saves programming time and effort from declaring variables

3. Mention the use of the split function in Python?

The use of the split function in Python is that it breaks a string into shorter strings using the defined separator. It gives a list of all words present in the string.

4. What is __init__?

`__init__` is a method or constructor in Python. This method is automatically called to allocate memory when a new object/ instance of a class is created. All classes have the `__init__` method.

5. What are functions in Python?

A function is a block of code which is executed only when it is called. To define a Python function, the `def` keyword is used.

6. Is indentation required in python?

Indentation is necessary for Python. It specifies a block of code. All code within loops, classes, functions, etc is specified within an indented block. It is usually done using four space characters. If your code is not indented necessarily, it will not execute accurately and will throw errors as well.

7.What is type conversion in Python?

Type conversion refers to the conversion of one data type into another.

`int()` – converts any data type into integer type

`float()` – converts any data type into float type

`ord()` – converts characters into integer

`hex()` – converts integers to hexadecimal

`oct()` – converts integer to octal

`tuple()` – This function is used to convert to a tuple.

`set()` – This function returns the type after converting to set.

`list()` – This function is used to convert any data type to a list type.

dict() – This function is used to convert a tuple of order (key,value) into a dictionary.

str() – Used to convert integer into a string.

complex(real,imag) – This function converts real numbers to complex(real,imag) number.

8. What is self in Python?

Self is an instance or an object of a class. In Python, this is explicitly included as the first parameter. However, this is not the case in Java where it's optional. It helps to differentiate between the methods and attributes of a class with local variables.

The self variable in the init method refers to the newly created object while in other methods, it refers to the object whose method was called.

9. How will you capitalize the first letter of string?

In Python, the capitalize() method capitalizes the first letter of a string. If the string already consists of a capital letter at the beginning, then, it returns the original string.

10. How to comment multiple lines in python?

Multi-line comments appear in more than one line. All the lines to be commented are to be prefixed by a #. You can also use a very good shortcut method to comment multiple lines. All you need to do is hold the ctrl key and left click in every place wherever you want to include a # character and type a # just once. This will comment all the lines where you introduced your cursor.

Python Interview Questions Part 5

1. What is the purpose of is, not and in operators?

Operators are special functions. They take one or more values and produce a corresponding result.

is: returns true when 2 operands are true (Example: "a" is 'a')

not: returns the inverse of the boolean value

in: checks if some element is present in some sequence

2. What does len() do?

It is used to determine the length of a string, a list, an array, etc.

3. How can files be deleted in Python?

To delete a file in Python, you need to import the OS Module. After that, you need to use the os.remove() function.

4. How to remove values to a python array?

Array elements can be removed using pop() or remove() method. The difference between these two functions is that the former returns the deleted value whereas the latter does not.

5. What is the difference between deep and shallow copy?

Shallow copy is used when a new instance type gets created and it keeps the values that are copied in the new instance. Shallow copy is used to copy the reference pointers just like it copies the values. These references point to the original objects and the changes made in any member of the class will also affect the original copy of it. Shallow copy allows faster execution of the program and it depends on the size of the data that is used.

Deep copy is used to store the values that are already copied. Deep copy doesn't copy the reference pointers to the objects. It makes the reference to an object and the new object that is pointed by some other object gets stored. The changes made in the original copy won't affect any other copy that uses the object. Deep copy makes execution of the program slower due to making certain copies for each object that is been called.

6. How is Multithreading achieved in Python?

1. Python has a multi-threading package but if you want to multi-thread to speed your code up, then it's usually not a good idea to use it.

2. Python has a construct called the Global Interpreter Lock (GIL). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread.

3. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core.

4. All this GIL passing adds overhead to execution. This means that if you want to make your code run faster then using the threading package often isn't a good idea.

7. Explain Inheritance in Python with an example.

Inheritance allows One class to gain all the members(say attributes and methods) of another class. Inheritance provides code reusability, makes it easier to create and maintain an application. The class from which we are inheriting is called super-class and the class that is inherited is called a derived / child class.

They are different types of inheritance supported by Python:

1. Single Inheritance – where a derived class acquires the members of a single super class.

2. Multi-level inheritance – a derived class d1 is inherited from base class base1, and d2 are inherited from base2.

3. Hierarchical inheritance – from one base class you can inherit any number of child classes

4. Multiple inheritance – a derived class is inherited from more than one base class.

8. How are classes created in Python?

Class in Python is created using the `class` keyword.

9. What is Polymorphism in Python?

Polymorphism means the ability to take multiple forms. So, for instance, if the parent class has a method named ABC then the child class also can have a method with the same name ABC having its own parameters and variables. Python allows polymorphism.

10. Define encapsulation in Python?

Encapsulation means binding the code and the data together. A Python class is an example of encapsulation.

Python Programming Language

Python as a Programming Language

- **High-Level Language:** Easy to read and write like English.
- **General Purpose:** Suitable for Web Development, AI, ML, Automation, Data Science, etc.
- **Interpreted:** Executes code line-by-line without prior compilation.
- **Dynamically Typed:** No need to specify variable types.
- **Object-Oriented:** Supports concepts like classes and objects.
- **Extensible and Embeddable:** Integrates with C, C++ easily.

Features of Python

- **Simple and Easy to Learn:** Beginner-friendly syntax.
- **Free and Open Source:** Available at no cost with a huge community.
- **Portable:** Runs on Windows, MacOS, and Linux without modification.
- **Extensive Libraries:** Access to NumPy, Pandas, TensorFlow, Scikit-learn, etc.
- **Robust Community Support:** Millions of developers contribute worldwide.
- **Automatic Memory Management:** Garbage collection handled internally.

History of Python

- **Creator:** Guido van Rossum
- **Start Year:** 1989
- **First Release:** 1991
- **Developed at:** Centrum Wiskunde & Informatica (CWI), Netherlands
- **Motivation:** To address issues found in ABC language and make programming easy.

Versions of Python

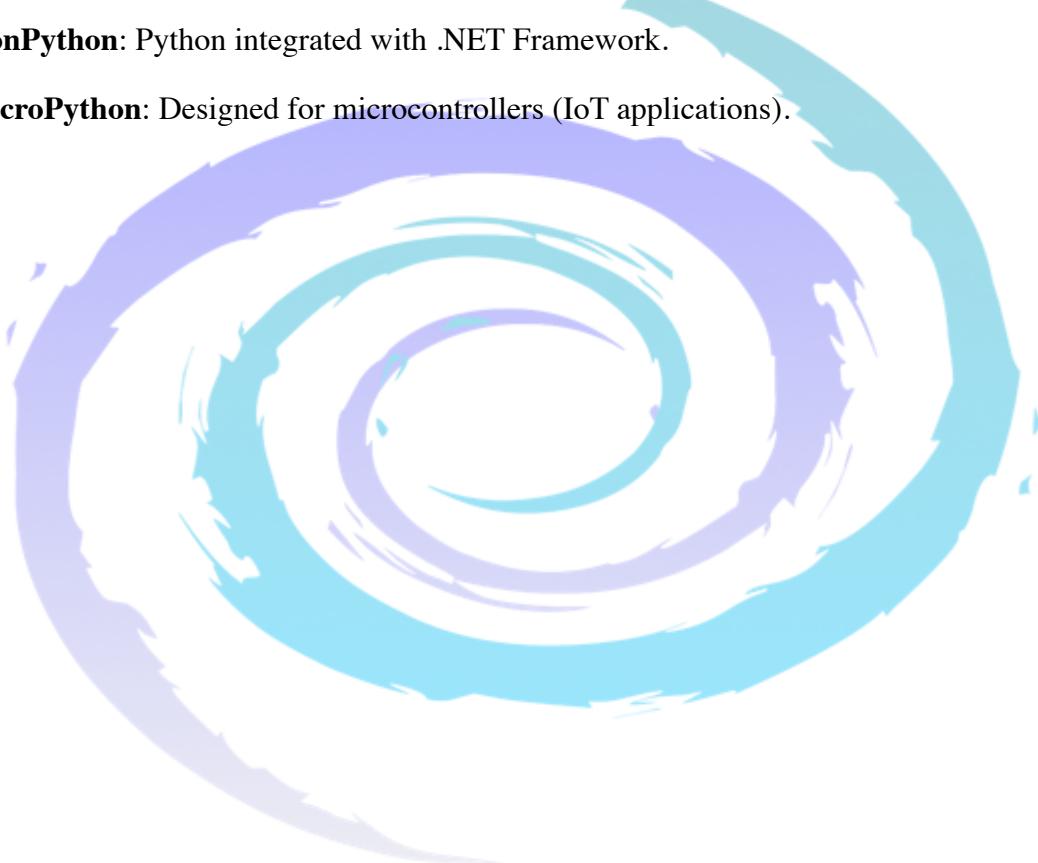
- **Python 1.x:** Initial release (1991)
- **Python 2.x:** Improved features (2000), not fully compatible with 3.x

- **Python 3.x:** Major changes (2008), modern standard
- **Latest Stable Version (2025):** Python 3.12+

Note: Python 2 support officially ended on **January 1, 2020.**

Implementations of Python

- **CPython:** Default, written in C language.
- **PyPy:** Faster execution using Just-In-Time (JIT) compiler.
- **Jython:** Python code execution on Java Virtual Machine (JVM).
- **IronPython:** Python integrated with .NET Framework.
- **MicroPython:** Designed for microcontrollers (IoT applications).



K Mean Algorithm For Clustering

Clustering is a type of Unsupervised learning.

This is very often used when you don't have labeled data.

K-Means Clustering is one of the popular clustering algorithm.

The goal of this algorithm is to find groups(clusters) in the given data.

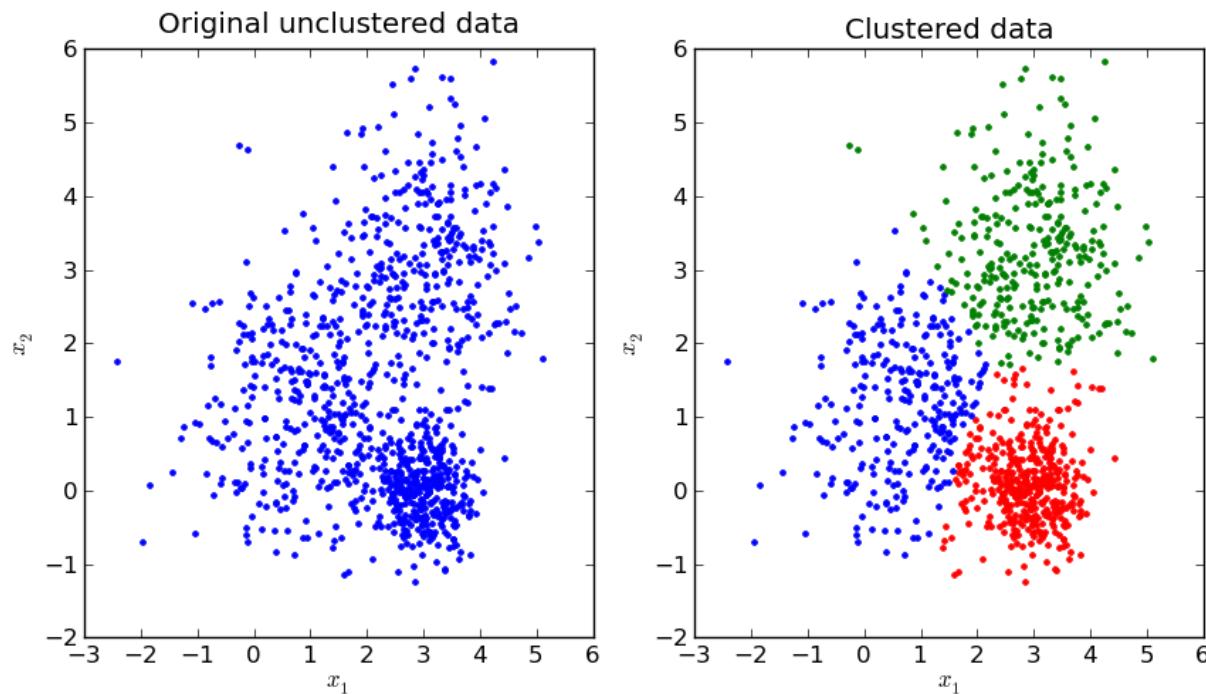
K Means Clustering is one of the most popular Machine Learning algorithms for cluster analysis in data mining. K-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

K Means algorithm is an unsupervised learning algorithm, ie. it needs no training data, it performs the computation on the actual dataset. This should be apparent from the fact that in K Means, we are just trying to group similar data points into clusters, there is no prediction involved.

The K Means algorithm is easy to understand and to implement. It works well in a large number of cases and is a powerful tool to have in the closet.

The algorithm has a loose relationship to the k-nearest neighbor classifier, a popular machine learning technique for classification.

K-Means is a very simple algorithm which clusters the data into K number of clusters. The following image from PyPR is an example of K-Means Clustering.



The K Means algorithm is iterative based, it repeatedly calculates the cluster centroids, refining the values until they do not change much.

The k-means algorithm takes a dataset of 'n' points as input, together with an integer parameter 'k' specifying how many clusters to create(supplied by the programmer). The output is a set of 'k' cluster centroids and a labeling of the dataset that maps each of the data points to a unique cluster.

In the beginning, the algorithm chooses k centroids in the dataset. Then it calculates the distance of each point to each centroid. Each centroid represents a cluster and the points closest to the centroid are assigned to the cluster. At the end of the first iteration, the centroid values are recalculated, usually taking the arithmetic mean of all points in the cluster.

After the new values of centroid are found, the algorithm performs the same set of steps over and over again until the differences between old centroids and the new centroids are negligible.

K Mean Algorithm

Our algorithm works as follows, assuming we have inputs $x_1, x_2, x_3, \dots, x_n$ and value of K

Step 1 -

Pick K random points as cluster centers called centroids.

Step 2 -

Assign each x_i to nearest cluster by calculating its distance to each centroid.

Step 3 -

Find new cluster center by taking the average of the assigned points.

Step 4 -

Repeat Step 2 and 3 until none of the cluster assignments change.

Detailed Explanation :

Step 1

We randomly pick K cluster centers(centroids). Let's assume these are c_1, c_2, \dots, c_k , and we can say that;

$$C = c_1, c_2, \dots, c_k$$

C is the set of all centroids.

Step 2

In this step we assign each input value to closest center. This is done by calculating Euclidean(L2) distance between the point and the each centroid.

$$\arg \min_{c_i \in C} dist(c_i, x)^2$$

Where $dist(\cdot)$ is the Euclidean distance.

Step 3

In this step, we find the new centroid by taking the average of all the points assigned to that cluster.

$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

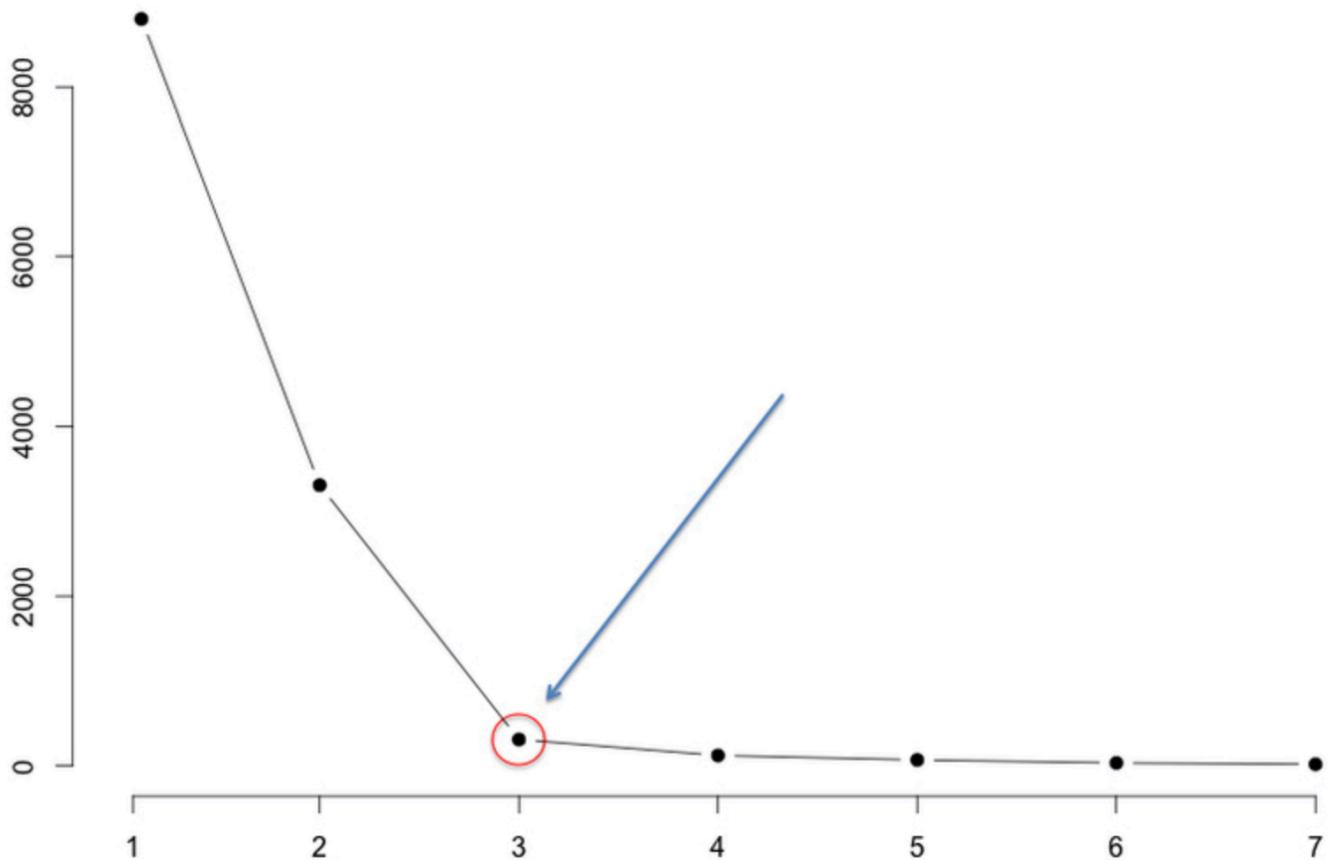
S_i is the set of all points assigned to the i th cluster.

Step 4

In this step, we repeat step 2 and 3 until none of the cluster assignments change. That means until our clusters remain stable, we repeat the algorithm.

Choosing the Value of K

We often know the value of K. In that case we use the value of K. Else we use the Elbow Method.



We run the algorithm for different values of K(say $K = 10$ to 1) and plot the K values against SSE(Sum of Squared Errors). And select the value of K for the elbow point as shown in the figure.

K-Means Clustering

K-Means is an **unsupervised learning algorithm** used for **clustering** data into **K groups** based on feature similarity.

- It tries to **minimize intra-cluster variance** (data points within a cluster should be close to each other).
- It is iterative and works well with numeric data.

Algorithm Steps

1. **Select the value of K (number of clusters).**
2. **Initialize K centroids randomly.**
3. **Assign each data point to the closest centroid** (using distance like Euclidean).
4. **Recalculate the centroids** (mean of points in each cluster).
5. **Repeat steps 3 and 4 until centroids do not change** (or max iterations reached).

K-Means Clustering (2D Example)

We will work with **2D data points** to understand clustering better using **Euclidean distance**.

Sample 2D Data Points:

Let's assume we have 6 points in 2D space:

```
P1 = (2, 10)
P2 = (2, 5)
P3 = (8, 4)
P4 = (5, 8)
P5 = (7, 5)
P6 = (6, 4)
```

We'll choose **K = 2 (2 clusters)**

Step 1: Initialization

Randomly select any **2 points** as **initial centroids**:

Let:

- $C_1 = P_1 = (2, 10)$
- $C_2 = P_4 = (5, 8)$

Step 2: Calculate Distance and Assign Cluster

Calculate distances of each point to centroids C1 and C2:

Point	Coordinates	d(C1) from (2,10)	d(C2) from (5,8)	Assigned Cluster
P1	(2, 10)	0.00	3.61	C1
P2	(2, 5)	5.00	3.61	C2
P3	(8, 4)	8.48	5.00	C2
P4	(5, 8)	3.61	0.00	C2
P5	(7, 5)	7.07	3.61	C2
P6	(6, 4)	7.21	4.47	C2

Step 3: Recalculate Centroids

Clusters:

- Cluster 1 (C1): P1 = (2, 10)
- Cluster 2 (C2): P2, P3, P4, P5, P6

New Centroid C1: mean of [(2,10)] = (2, 10) — unchanged

New Centroid C2: mean of P2–P6

$$x = (2+8+5+7+6)/5 = 28/5 = 5.6$$

$$y = (5+4+8+5+4)/5 = 26/5 = 5.2$$

$$\text{New Centroid C2} = (5.6, 5.2)$$

Step 4: Reassign Points Using New Centroids

Point	Coordinates	d(C1: 2,10)	d(C2: 5.6, 5.2)	New Cluster
P1	(2, 10)	0.00	6.23	C1
P2	(2, 5)	5.00	3.61	C2
P3	(8, 4)	8.48	2.55	C2
P4	(5, 8)	3.61	2.87	C2
P5	(7, 5)	7.07	1.45	C2
P6	(6, 4)	7.21	1.62	C2

No change in cluster assignment. Algorithm **converged**.

Final Clusters:

- **Cluster 1:** (2, 10)
Centroid: (2, 10)
- **Cluster 2:** (2, 5), (8, 4), (5, 8), (7, 5), (6, 4)
Centroid: (5.6, 5.2)

Consider below Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# 2D points,
X = np.array([[2,10], [2,5], [8,4], [5,8], [7,5], [6,4]])

# KMeans with 2 clusters
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)

# Plot
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, s=100)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='black', s=200, marker='X', label='Centroids')
plt.title("K-Means Clustering (2D)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.grid(True)
plt.show()
```

Lambda Functions in Python

Lambda Function :

- A **lambda function** is a **small anonymous function** in Python.
- It is used to create small, **throwaway functions** without formally defining them using **def**.

Anonymous Function means a function **without a name**.

Syntax of Lambda Function

```
lambda arguments: expression
```

- Can have **any number of arguments**, but **only one expression**.
- The result of the expression is **automatically returned**.

Example 1: Simple Lambda Function

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

Explanation:

- `lambda x: x * x` creates an anonymous function that returns square of x.
- It is assigned to variable `square`.

Example 2: Lambda with Multiple Arguments

python

```
add = lambda a, b: a + b
print(add(10, 20))  # Output: 30
```

Lambda vs Regular Function

Regular Function	Lambda Function
Uses <code>def</code> to define function	Uses <code>lambda</code> keyword
Can have multiple expressions	Only one expression
Can have a name	Usually anonymous
More readable for complex logic	Suitable for small, short functions

Use Cases of Lambda Functions

Lambda functions are commonly used with:

1. `filter()` – filter items based on a condition
2. `map()` – apply a function to all elements
3. `reduce()` – reduce elements to a single value



Large Language Model

- An **artificial intelligence model** trained on **massive amounts of text data** to **understand and generate human-like language**.
- Built using the **Transformer architecture**.
- Called “*Large*” because they have **billions or trillions of parameters** (the “knobs” of a neural network).

Example: GPT-3 has **175 billion** parameters, GPT-4 has **over a trillion (estimated)**.

Internal Working of LLM

1. Training:

- Trained on text from books, websites, code, articles, etc.
- Learns *patterns of language*, not exact memorization.
- Core task: **predict the next word**.
- Example:
Input: “The cat sat on the ...”
Prediction: “mat”.

2. Embeddings:

- Converts words/tokens into vectors (numbers).
- Example: “dog” is close to “cat” in vector space, far from “banana”.

3. Self-Attention (Transformer core):

- Model can relate any word to any other word in the sequence.
- Example: In “*The book that I read yesterday was amazing*”, the model links **book ↔ amazing**.

4. Fine-Tuning:

- After pretraining, models are often fine-tuned:
 - **Instruction tuning** → make it follow user commands.
 - **RLHF (Reinforcement Learning with Human Feedback)** → align it with human expectations.

Actual Meaning of Large

- The **size of the model (parameters)** allows it to store rich language patterns.
- Small models: Learn basic grammar, short context.
- Large models: Learn reasoning, long-term dependencies, creativity.

More parameters + more data = better understanding and generation.

Capabilities of LLMs

Language Understanding → summarization, Q&A, sentiment analysis.

Text Generation → essays, stories, reports, scripts.

Translation → English ⇔ French ⇔ Hindi, etc.

Coding → generate/debug code (e.g., Copilot, ChatGPT Code Interpreter).

Reasoning → solve logical/mathematical problems.

Multimodal extensions → some LLMs handle **text + images + audio** (e.g., GPT-4, Gemini).

Examples of LLMs

- **OpenAI** → GPT-3, GPT-4, GPT-5 (used in ChatGPT).
- **Google** → PaLM, Gemini.
- **Meta** → LLaMA series.
- **Anthropic** → Claude series.
- **Mistral** → Mistral 7B, Mixtral.

Analogy

Imagine teaching a **child to speak**:

- At first, they read a **lot of books** (pretraining).
- Then they practice **Q&A with a teacher** (fine-tuning).
- Finally, they learn **manners, safety, and alignment** with society (RLHF).

That's exactly how LLMs evolve.

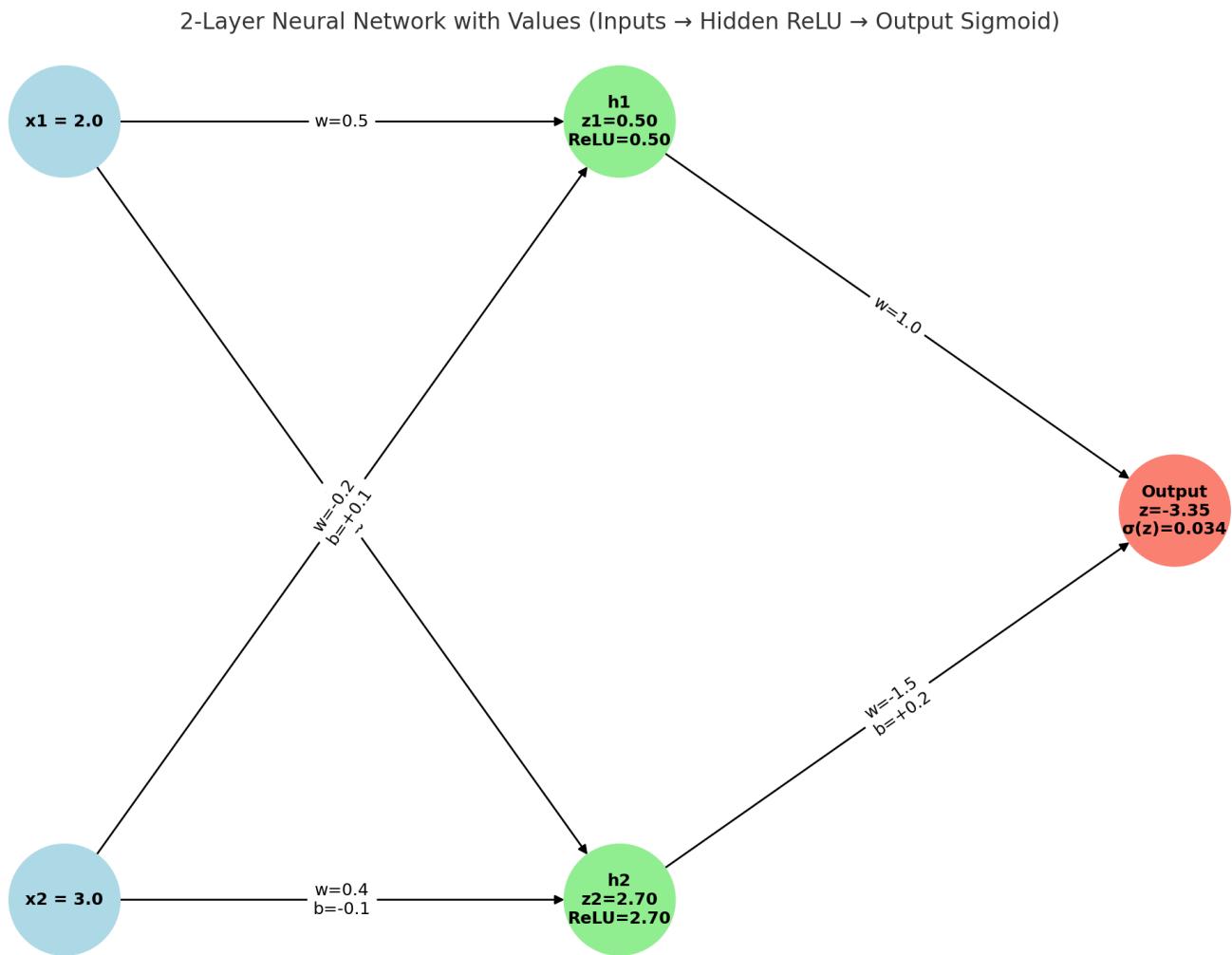
Summary

- LLMs are AI models trained on huge text data to understand and generate human-like text.
- They are built on **Transformers**, use **billions/trillions of parameters**, and power applications like **ChatGPT, Claude, Gemini, Copilot**.
- They excel at **understanding, generating, translating, reasoning**, but also have **limitations** like hallucination and bias.

RLHF Workflow



Application which implements 2 Layered Neural Network Design



```

import math

# ----- Activation Functions -----
def relu(x):
    return max(0, x)

def sigmoid(z):
    return 1 / (1 + math.exp(-z))

# ----- Forward Pass with Step-by-Step Printing -----
def Marvellous_forward_pass(inputs):
    print("==== INPUT LAYER ====")
    print(f"Inputs: x1 = {inputs[0]}, x2 = {inputs[1]}")

    # ----- Hidden Layer -----
    weights_hidden = [
        [0.5, -0.2], # Neuron 1 weights
        [0.8, 0.4] # Neuron 2 weights
    ]
    bias_hidden = [0.1, -0.1] # Bias for each hidden neuron
  
```

```

hidden_outputs = []
print("\n==== HIDDEN LAYER ====")
for i in range(len(weights_hidden)):
    print(f"\nNeuron {i+1}:")
    w = weights_hidden[i]
    b = bias_hidden[i]

    # Detailed multiplication steps
    print(f" Step 1: Multiply inputs by weights:")
    print(f" ({w[0]}) * {inputs[0]} = {w[0] * inputs[0]:.3f}")
    print(f" ({w[1]}) * {inputs[1]} = {w[1] * inputs[1]:.3f}")

    # Summation
    z = sum(w_j * x_j for w_j, x_j in zip(w, inputs)) + b
    print(f" Step 2: Add results and bias {b}: z = {z:.3f}")

    # Activation
    a = relu(z)
    print(f" Step 3: Apply ReLU: max(0, {z:.3f}) = {a:.3f}")

    hidden_outputs.append(a)

# ----- Output Layer -----
weights_output = [1.0, -1.5] # Weights from hidden to output
bias_output = 0.2

print("\n==== OUTPUT LAYER ====")
print(" Step 1: Multiply hidden outputs by weights:")
print(f" ({weights_output[0]} * {hidden_outputs[0]}) = {weights_output[0] * hidden_outputs[0]:.3f}")
print(f" ({weights_output[1]} * {hidden_outputs[1]}) = {weights_output[1] * hidden_outputs[1]:.3f}")

z_out = sum(w_o * h for w_o, h in zip(weights_output, hidden_outputs)) + bias_output
print(f" Step 2: Add results and bias {bias_output}: z = {z_out:.3f}")

y_hat = sigmoid(z_out)
print(f" Step 3: Apply Sigmoid: 1 / (1 + e^{-{z_out:.3f}}) = {y_hat:.3f}")

print(f"\nFinal Output: {y_hat:.3f} → {y_hat*100:.2f}% confidence in Positive Class")

if __name__ == "__main__":
    inputs = [2.0, 3.0] # Example input features
    Marvellous_forward_pass(inputs)
  
```

```

Terminal Shell Edit View Window Help
Codes — zsh — 81x34
Sat 16 Aug 10:02 PM

marvellous@Marvellouss-MacBook-Pro Codes % python3 MultilayerANN.py
==== INPUT LAYER ====
Inputs: x1 = 2.0, x2 = 3.0

==== HIDDEN LAYER ====

Neuron 1:
Step 1: Multiply inputs by weights:
(0.5 * 2.0) = 1.000
(-0.2 * 3.0) = -0.600
Step 2: Add results and bias 0.1: z = 0.500
Step 3: Apply ReLU: max(0, 0.500) = 0.500

Neuron 2:
Step 1: Multiply inputs by weights:
(0.8 * 2.0) = 1.600
(0.4 * 3.0) = 1.200
Step 2: Add results and bias -0.1: z = 2.700
Step 3: Apply ReLU: max(0, 2.700) = 2.700

==== OUTPUT LAYER ====
Step 1: Multiply hidden outputs by weights:
(1.0 * 0.4999999999999999) = 0.500
(-1.5 * 2.7) = -4.050
Step 2: Add results and bias 0.2: z = -3.350
Step 3: Apply Sigmoid: 1 / (1 + e^(-3.350)) = 0.034

Final Output: 0.034 → 3.39% confidence in Positive Class
marvellous@Marvellouss-MacBook-Pro Codes %

```

Explanation of above code

It performs a **single forward pass** through a tiny neural network with:

- **Inputs:** x1, x2
- **Hidden layer:** 2 neurons with **ReLU**
- **Output layer:** 1 neuron with **Sigmoid** and prints every intermediate computation.

Activations

```

def relu(x):      return max(0, x)
def sigmoid(z):   return 1 / (1 + math.exp(-z))

```

- **ReLU** keeps positive values and zeroes out negatives → helps non-linearity and avoids vanishing gradients for $z > 0$
- **Sigmoid** maps any real number to (0,1) → interpretable as a probability for the “positive” class.

The forward function

```

def Marvellous_forward_pass(inputs):
    print("==== INPUT LAYER ====")

```

```
print(f"Inputs: x1 = {inputs[0]}, x2 = {inputs[1]}")
```

- `inputs` is a 2-item list [`x1`, `x2`]. In `__main__` you pass [2.0, 3.0].

Hidden layer setup

```
weights_hidden = [
    [0.5, -0.2],    # Neuron 1 weights (w11,w12)
    [0.8, 0.4]      # Neuron 2 weights (w21,w22)
]
```

`bias_hidden` = [0.1, -0.1] # `b1`, `b2`

- Two neurons. Each has **2 weights** (one per input) and **one bias**.
- We'll compute each hidden neuron's pre-activation z and activation a

Hidden layer — per-neuron math (matches printed steps)

For Neuron 1

```
w = [0.5, -0.2]; b = 0.1
z1 = (0.5*2.0) + (-0.2*3.0) + 0.1
      = 1.0          + (-0.6)        + 0.1
      = 0.5
a1 = ReLU(z1) = max(0, 0.5) = 0.5
```

For Neuron 2

```
w = [0.8, 0.4]; b = -0.1
z2 = (0.8*2.0) + (0.4*3.0) + (-0.1)
      = 1.6          + 1.2          - 0.1
      = 2.7
a2 = ReLU(z2) = max(0, 2.7) = 2.7
```

The code stores these in `hidden_outputs` = [`a1`, `a2`] = [0.5, 2.7].

Output layer — weighted sum then Sigmoid

```
weights_output = [1.0, -1.5] # v1 (from h1), v2 (from h2)
bias_output = 0.2
```

Compute pre-activation z and final output y^{\wedge} :

```
z_out = (1.0*0.5) + (-1.5*2.7) + 0.2
      = 0.5          + (-4.05)       + 0.2
      = -3.35
```

```
y_hat = sigmoid(z_out) = 1 / (1 + e^{3.35}) ≈ 0.034
```

So the network outputs **~0.034**, i.e., **3.4% confidence** in the positive class for inputs
[2.0 , 3.0]

- **ReLU in the hidden layer:** keeps the layer piece-wise linear but avoids saturating small gradients for positive inputs; encourages sparse activations.
- **Sigmoid at output:** converts the final score into a probability for a **binary** decision.

Final conclusions :

Interpretation: low probability → model leans toward the **negative** class.



Linear Regression

X (Independent)	Y (Dependent)	X - X_Bar	Y - Y_Bar	(X-X_Bar)^2	(X - X_Bar) * (Y - Y_Bar)	Yp	(Yp - Y_Bar)	(Yp - Y_Bar) ^ 2	(Y - Y_Bar)^2
1	3	-2	-0.6	4	1.2	2.8	-0.8	0.64	0.36
2	4	-1	0.4	1	-0.4	3.2	-0.4	0.16	0.16
3	2	0	-1.6	0	0	3.6	0	0	2.56
4	4	1	0.4	1	0.4	4.0	0.4	0.16	0.16
5	5	2	1.4	4	2.8	4.4	0.8	0.64	1.96
X_Bar 3	Y_Bar. 3.6			Sum = 10	Sum = 4.0			Sum = 1.6	Sum = 5.2

Equation of line : $Y = mX + C$

Y Dependent Variable
 X Independent Variable
 m Slope of line
 c Y intercept of line

$$m = \frac{\sum (X - X_{\text{Bar}})(Y - Y_{\text{Bar}})}{\sum (X - X_{\text{Bar}})^2}$$

$$m = 4/10$$

m = 0.4

$$\begin{aligned}
 Y &= mX + C \\
 3.6 &= 0.4 * 3 + C \\
 3.6 &= 1.2 + C \\
 C &= 3.6 - 1.2 \\
 C &= 2.4
 \end{aligned}$$

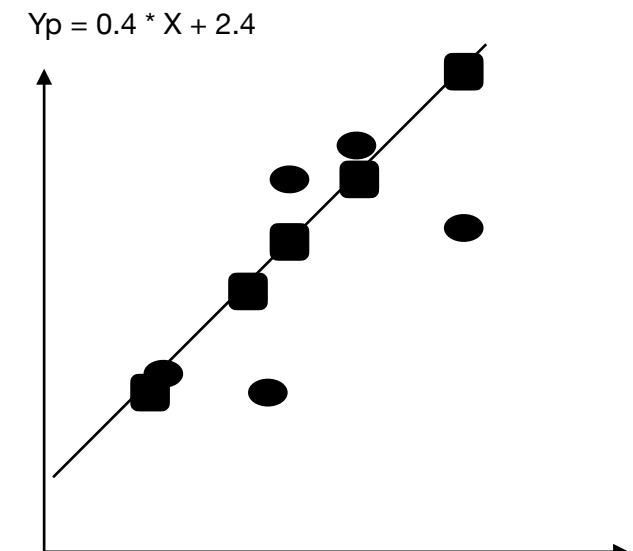
R Square method

Distance (predicted - mean)
VS
Distance (actual - mean)

R_Square formula

$$\frac{\sum(Y_p - Y_{\text{Bar}})^2}{\sum(Y - Y_{\text{Bar}})^2}$$

$$\begin{aligned}
 R^2 &= 1.6 / 5.2 \\
 R^2 &= 0.3
 \end{aligned}$$



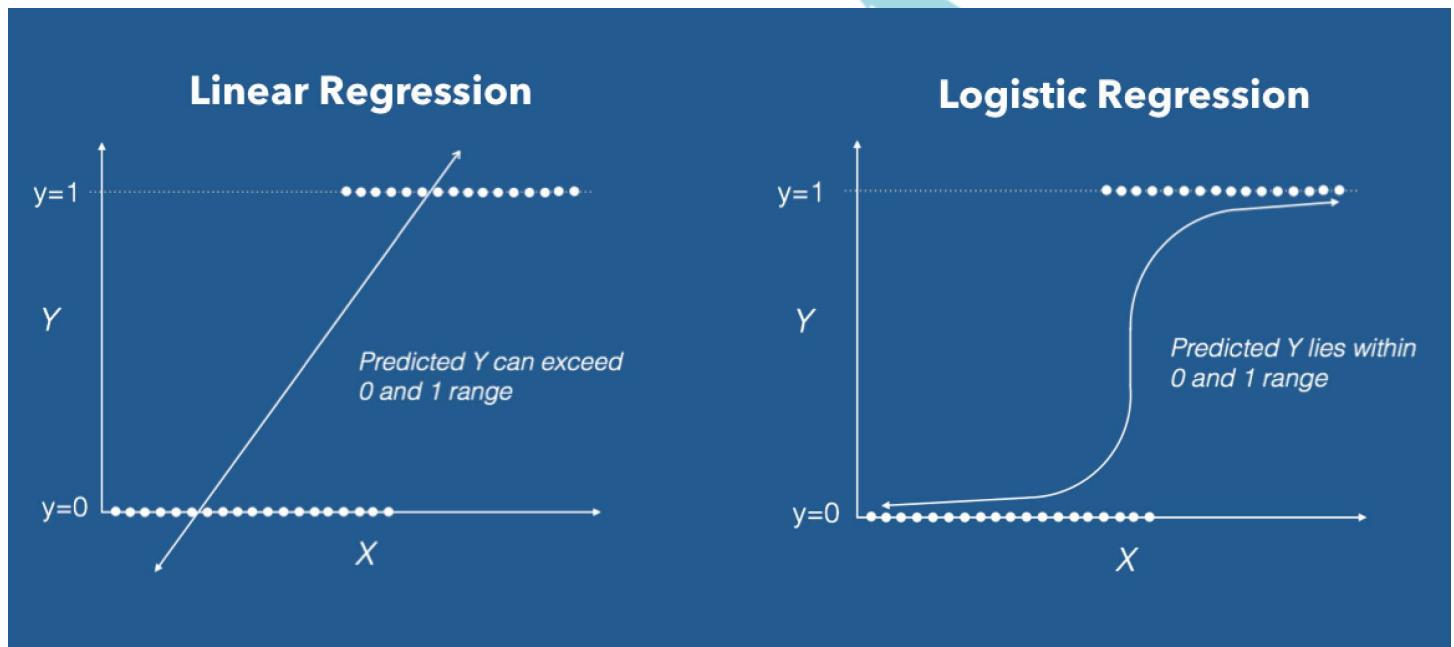
Logistic Regression

Logistic regression predicts the probability of an outcome that can only have two values (i.e. a dichotomy).

The prediction is based on the use of one or several predictors (numerical and categorical).

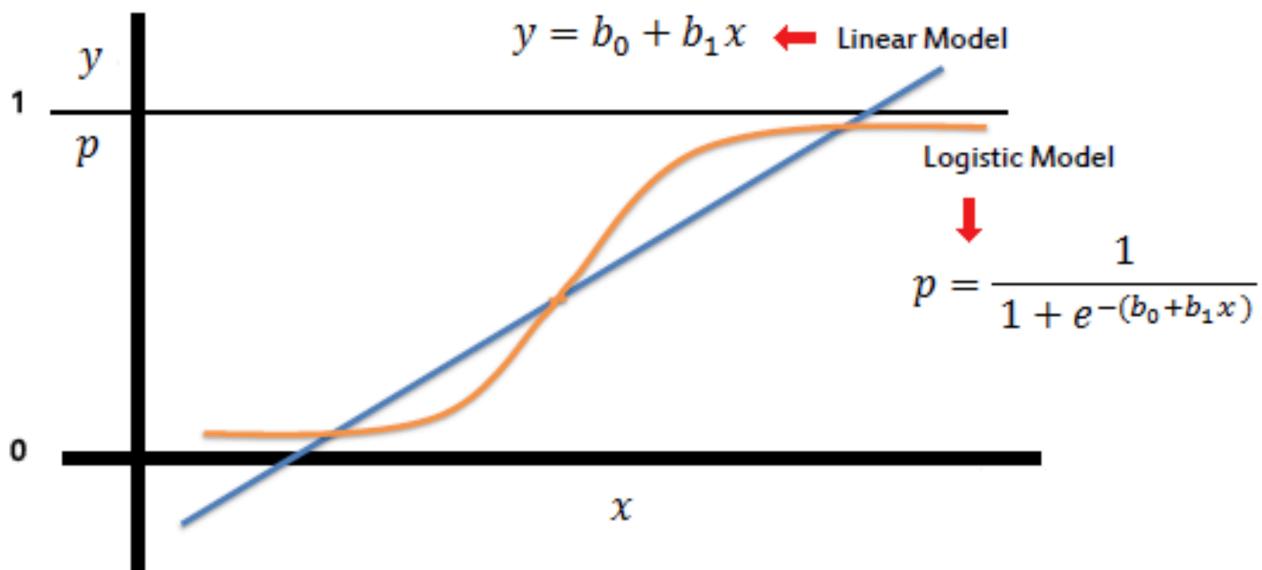
A linear regression is not appropriate for predicting the value of a binary variable for two reasons:

- A linear regression will predict values outside the acceptable range (e.g. predicting probabilities outside the range 0 to 1)
- Since the dichotomous experiments can only have one of two possible values for each experiment, the residuals will not be normally distributed about the predicted line.



On the other hand, a logistic regression produces a logistic curve, which is limited to values between 0 and 1.

Logistic regression is similar to a linear regression, but the curve is constructed using the natural logarithm of the "odds" of the target variable, rather than the probability. Moreover, the predictors do not have to be normally distributed or have equal variance in each group.



In the logistic regression the constant (b_0) moves the curve left and right and the slope (b_1) defines the steepness of the curve. By simple transformation, the logistic regression equation can be written in terms of an odds ratio.

$$\frac{p}{1-p} = \exp(b_0 + b_1 x)$$

Finally, taking the natural log of both sides, we can write the equation in terms of log-odds (logit) which is a linear function of the predictors. The coefficient (b_1) is the amount the logit (log-odds) changes with a one unit change in x .

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

As mentioned before, logistic regression can handle any number of numerical and/or categorical variables.

$$p = \frac{1}{1 + e^{-(b_0 + b_1 x_1 + b_2 x_2 + \dots + b_p x_p)}}$$

There are several analogies between linear regression and logistic regression. Just as ordinary least square regression is the method used to estimate coefficients for the best fit line in linear regression, logistic regression uses maximum likelihood estimation (MLE).

to obtain the model coefficients that relate predictors to the target. After this initial function is estimated, the process is repeated until LL (Log Likelihood) does not change significantly.

$$\beta^1 = \beta^0 + [X^T W X]^{-1} \cdot X^T (y - \mu)$$

β is a vector of the logistic regression coefficients.

W is a square matrix of order N with elements $n_i \pi_i (1 - \pi_i)$ on the diagonal and zeros everywhere else.

μ is a vector of length N with elements $\mu_i = n_i \pi_i$.



Logistic Regression – Concept + Mathematical Example

- Logistic Regression is a **supervised learning algorithm** used for **binary classification** problems.
- It predicts **probability** that a data point belongs to a particular class using a **sigmoid function**.
- Output is between **0 and 1** – typically converted to 0 or 1 using a **threshold** (commonly 0.5).

◆ Sigmoid Function

The logistic function (sigmoid) is:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Where:

- $z = mx + c$ (linear equation)
- m is slope/weight
- c is intercept/bias

Example with Numerical Values

We have:

Age (X)	Result (Y)
50	1
30	0
20	0
40	1
60	1

Let's fit a logistic regression model on this data and use **gradient descent** to learn parameters m and c .

Step-by-Step Manual Calculation

Let's simplify with initial guess:

Assume $m = 0.1$ $c = -4$ (Just for calculation)

Predict probability using sigmoid:

$$z = mx + c = 0.1x - 4$$

Let's calculate \hat{y} for each input (Age):

Age (x)	$z = 0.1x - 4$	$\hat{y} = \frac{1}{1+e^{-z}}$
50	1	$\frac{1}{1+e^{-1}} \approx 0.731$
30	-1	$\frac{1}{1+e^1} \approx 0.268$
20	-2	$\frac{1}{1+e^2} \approx 0.119$
40	0	$\frac{1}{1+e^0} = 0.5$
60	2	$\frac{1}{1+e^{-2}} \approx 0.881$

Prediction:

- $x = 15$:

$$z = 0.1 * 15 - 4 = -2.5 \Rightarrow \hat{y} \approx \frac{1}{1 + e^{-2.5}} \approx 0.075 \Rightarrow \text{Predicted Class} = 0$$

- $x = 45$:

$$z = 0.1 * 45 - 4 = 0.5 \Rightarrow \hat{y} \approx \frac{1}{1 + e^{-0.5}} \approx 0.622 \Rightarrow \text{Predicted Class} = 1$$

Loops in Python

Loop :

A loop is a **control structure** that allows us to **repeat a block of code** multiple times. Its used to achieve the concept of iteration. This helps in automating repetitive tasks efficiently.

Types of Loops in Python

1. **for** loop – used to iterate over a sequence
2. **while** loop – used when you want to loop based on a condition

for Loop

Syntax:

```
for variable in sequence:
    # block of code
```

- The **for** loop iterates over items in a **sequence** (like a list, tuple, dictionary, set, or string).

Example 1: Print numbers from 1 to 5

```
for i in range(1, 6):
    print(i)
```

Example 2: Loop through a list

```
Marvellous = [ 'PPA', 'LB', 'Python' ]
for name in Marvellous:
    print("Batch name is", name)
```

while Loop

Syntax:

```
while condition:
    # block of code
• The while loop continues until the condition becomes False.
```

Example: Print numbers from 1 to 5

```
i = 1
while i <= 5:
    print(i)
```

```
i += 1
```

Loop Control Statements

break: Exit the loop immediately

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
# Output: 1 2 3 4
```

continue: Skip the current iteration

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
# Output: 1 2 4 5
```

Nested Loops

You can place one loop inside another.

Example:

python

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, "*", j, "=", i*j)
```

Common Loop Patterns

Print sum of numbers from 1 to 10

```
total = 0
for i in range(1, 11):
    total += i
print("Sum:", total)
```

Count even numbers from a list

```
numbers = [1, 4, 6, 9, 12, 15]
count = 0
for num in numbers:
    if num % 2 == 0:
```

```

    count += 1
print("Even numbers:", count)
  
```

Understanding **range()** Function in Python

The **range()** function is one of the most commonly used functions in Python loops. It generates a sequence of numbers, which is useful for iteration.

Syntax of **range()**:

```
range(start, stop, step)
```

- **start** → The starting value of the sequence (inclusive). Default is 0 if not provided.
- **stop** → The end value (exclusive). This is **not included** in the output.
- **step** → The difference between each number in the sequence. Default is 1.

Examples:

1. Only **stop** is provided:

```

for i in range(5):
    print(i)
# Output: 0 1 2 3 4
  
```

2. **start** and **stop**:

```

for i in range(2, 6):
    print(i)
# Output: 2 3 4 5
  
```

3. **start**, **stop**, and **step**:

```

for i in range(1, 10, 2):
    print(i)
# Output: 1 3 5 7 9
  
```

4. Negative **step** (counting backward):

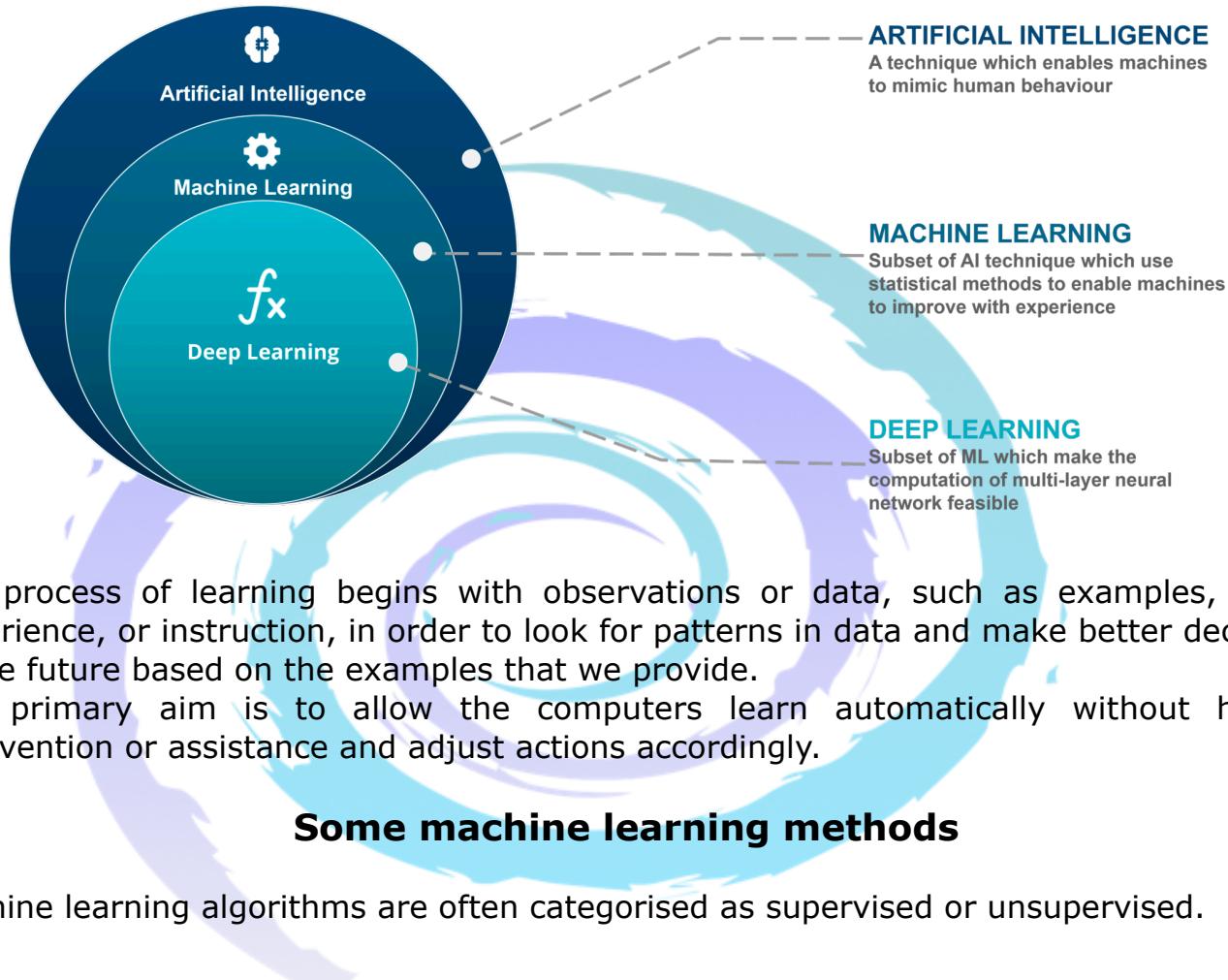
```

for i in range(10, 0, -2):
    print(i)
# Output: 10 8 6 4 2
  
```

Machine Learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed.

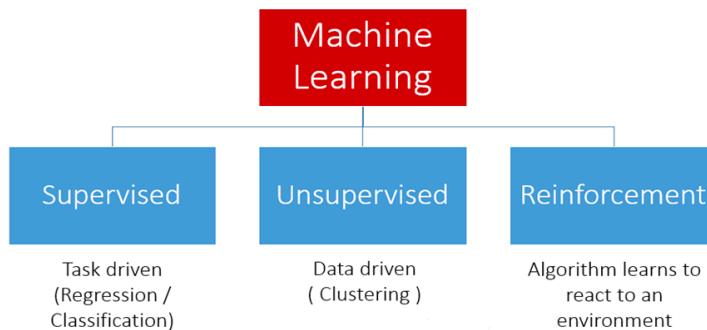
Machine learning focuses on the development of computer programs that can access data and use it learn for themselves.



Some machine learning methods

Machine learning algorithms are often categorised as supervised or unsupervised.

Types of Machine Learning



Supervised machine learning :

Supervised machine learning algorithms can apply what has been learned in the past to new data using labeled examples to predict future events.

Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values.

The system is able to provide targets for any new input after sufficient training.

The learning algorithm can also compare its output with the correct, intended output and find errors in order to modify the model accordingly.

Unsupervised machine learning :

In contrast, unsupervised machine learning algorithms are used when the information used to train is neither classified nor labeled.

Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data.

The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data.

Semi-supervised machine learning :

Semi-supervised machine learning algorithms fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training – typically a small amount of labeled data and a large amount of unlabeled data.

The systems that use this method are able to considerably improve learning accuracy. Usually, semi-supervised learning is chosen when the acquired labeled data requires skilled and relevant resources in order to train it / learn from it.

Otherwise, acquiring unlabeled data generally doesn't require additional resources.

Reinforcement machine learning :

Reinforcement machine learning algorithms is a learning method that interacts with its environment by producing actions and discovers errors or rewards.

Trial and error search and delayed reward are the most relevant characteristics of reinforcement learning.

This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance.

Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal.

Machine learning enables analysis of massive quantities of data.

While it generally delivers faster, more accurate results in order to identify profitable opportunities or dangerous risks, it may also require additional time and resources to train it properly.

Combining machine learning with AI and cognitive technologies can make it even more effective in processing large volumes of information.

Machine Learning Libraries in Python

1. NumPy (Numerical Python)

NumPy is the foundation for numerical computing in Python. It allows you to work with **arrays**, **matrices**, and perform **high-performance mathematical operations**.

Key Features:

- Multidimensional array support (ndarray)
- Fast mathematical functions (mean, median, std, etc.)
- Broadcasting and vectorized operations
- Linear algebra, Fourier transform, random number generation

All data in ML is ultimately numbers. NumPy helps perform fast mathematical operations during **data preprocessing**, **feature engineering**, and **model calculations**.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print("Mean:", np.mean(arr))
print("Standard Deviation:", np.std(arr))
```

2. Pandas (Python Data Analysis Library)

Pandas makes it easy to **load**, **analyze**, **clean**, and **manipulate data** in Python. It provides powerful **DataFrame** and **Series** structures similar to Excel tables.

Key Features:

- Load data from CSV, Excel, SQL, JSON, etc.
- Handle missing values, filter rows, group data
- Merge, join, and reshape datasets
- Excellent integration with NumPy and Matplotlib

Pandas is used for **data exploration**, **cleaning**, and **preprocessing**, which is the most time-consuming and critical part of the ML pipeline.

Example:

```
import pandas as pd

df = pd.read_csv("iris.csv")
print(df.head()) # Shows first 5 rows
print(df.describe()) # Statistical summary
```

3. Matplotlib

Matplotlib is the most widely used library for **basic data visualization** in Python.

Key Features:

- Create line plots, bar charts, histograms, scatter plots
- Full control over plot style, color, labels, and axes
- Supports exporting images in various formats (PNG, SVG)

Used to **visualize data distributions, loss curves, and performance metrics** during training.

Example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3]
y = [10, 20, 30]

plt.plot(x, y)
plt.title("Sample Line Plot")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

4. Seaborn

Seaborn is a statistical data visualization library built on top of Matplotlib, offering **beautiful and informative plots**.

Key Features:

- Visualize distributions, correlations, and categorical data
- Built-in themes and color palettes
- Works seamlessly with Pandas DataFrames

Great for **exploratory data analysis (EDA)**—understanding the relationship between features and labels.

Example:

```
import seaborn as sns
import pandas as pd

df = sns.load_dataset('iris')
sns.pairplot(df, hue='species')
plt.show()
```

5. Scikit-learn (sklearn)

Scikit-learn is the **standard machine learning library** in Python. It provides tools for building, training, and evaluating ML models.

Key Features:

- Classification, regression, clustering algorithms
- Data preprocessing (scaling, encoding, splitting)
- Model evaluation tools (accuracy, confusion matrix)
- Pipeline for combining steps

It's used for everything from **model building** to **evaluation** in classical machine learning projects.

Example:

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Train model
model = RandomForestClassifier()
model.fit(X_train, y_train)
print("Accuracy:", model.score(X_test, y_test))
```

6. TensorFlow

TensorFlow is a powerful open-source framework developed by **Google** for **building and training deep learning models**.

Key Features:

- Low-level control of neural networks
- Automatic differentiation and GPU acceleration
- Works for both research and production-level applications
- Supports mobile and embedded deployment (TensorFlow Lite)

Used for **deep learning**, especially where large data and powerful neural network architectures are required.

Example:

```
import tensorflow as tf
a = tf.constant([1, 2, 3])
print("Tensor:", a)
```

7. Keras (High-Level API for TensorFlow)

Keras is a **high-level neural network library** that simplifies the creation and training of deep learning models. It's now integrated within TensorFlow as `tf.keras`.

Key Features:

- Easy model building using Sequential or Functional API
- Abstracts complex TensorFlow code
- Supports custom layers and loss functions
- Fast prototyping of deep learning models

Used to **quickly build and train** deep learning models with very few lines of code.

Example:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Simple neural network
model = Sequential([
    Dense(8, activation='relu', input_shape=(4,)),
    Dense(3, activation='softmax')
])
```

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



Mathematical Operations of Artificial Neural Network

Forward Propagation

- **Definition:**
Process of sending inputs through the network (layer by layer) to generate an output.
- **Explanation:**
Each layer transforms data → Output layer gives predictions.
- **Example:**
Image → [edges detected] → [digits recognized] → "Digit = 7".

Backward Propagation (Backpropagation)

- **Definition:**
Algorithm used to update weights by minimizing prediction error.
- **Explanation:**
 - Calculates error (Loss).
 - Finds gradient (slope) of error wrt weights.
 - Adjusts weights in opposite direction of gradient.
- **Example:**
If prediction = 0.9 but actual = 1, error is high → backprop adjusts weights to reduce error next time.

Gradient Descent

- **Definition:**
Optimization algorithm that reduces the loss function by updating weights gradually.
- **Explanation:**
Imagine descending a hill → we move step by step toward the lowest point (minimum loss).
- **Formula:**

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot (\partial w / \partial L)$$
- where
 α = learning rate.
- **Example:**
Suppose error = height of hill. The algorithm adjusts weights like moving downhill to reach the bottom (minimum error).

Learning Rate (α)

- **Definition:**
A parameter controlling how much weights are updated during training.
- **Explanation:**
 - Small $\alpha \rightarrow$ slow but stable learning.
 - Large $\alpha \rightarrow$ fast learning but may overshoot.
- **Example:**
If $\alpha = 0.01$, weights are updated in **tiny steps**. If $\alpha = 1.0$, steps are **huge**, risking instability.

Epoch

- **Definition:**
One full pass of the training dataset through the network.
- **Explanation:**
 - During each epoch, all data is seen once.
 - Multiple epochs are needed to train properly.
- **Example:**
Training on 1000 images for 10 epochs \rightarrow Model sees **10,000 images in total** (same data, multiple passes).

Batch / Mini-batch

- **Definition:**
A subset of training data processed before updating weights.
- **Explanation:**
 - **Batch Gradient Descent:** Uses the whole dataset for one update.
 - **Mini-batch Gradient Descent:** Uses small groups (like 32 or 64 samples).
- **Example:**
If dataset = 10,000 images, batch size = 100 \rightarrow 100 updates per epoch.

Important terms :

- **Neurons, layers, weights, and bias** form the **architecture** of ANN.
- **Forward propagation** \rightarrow gives predictions.
- **Backward propagation + Gradient Descent** \rightarrow improves accuracy.
- **Learning rate, epochs, batches** \rightarrow control **how well & how fast the model learns**.

Python Modules

Module in Python

A **module** in Python is simply a **file containing Python code** — variables, functions, classes — that you can **reuse in other Python programs**.

File with .py extension is treated as a module.

Use of Modules

- Reusability: Write once, use anywhere.
- Organization: Keep code modular and manageable.
- Avoid Code Duplication: Common code can be placed in a module.
- Easier Maintenance & Debugging

Types of Modules

Type	Description	Example
Built-in	Pre-installed with Python	math, os, sys
User-defined	Created by users as .py files	Marvellous.py
Third-party	Installed via pip from external packages	numpy, pandas, flask

Creating and Using a User-Defined Module

Step 1: Create a Module

👉 File: Marvellous.py

```
def Display():
    print("Jai Ganesh..")
```

PI = 3.14159

Step 2: Import and Use in Another File

👉 File: main.py

```
import Marvellous
```

```
Marvellous.Display()
print("Value of PI:", Marvellous.PI)
```

Output:

Jai Ganesh..

Value of pi: 3.14159

This is **how we import and use our own code written in other files**.

Ways to Import Modules

Syntax	Usage
<code>import module_name</code>	Imports whole module
<code>from module_name import x</code>	Imports specific object/function
<code>from module_name import *</code>	Imports everything (not recommended)
<code>import module_name as alias</code>	Imports with alias name for shorter usage

Example:

```
from Marvellous import Display

Display()
```

Built-in Modules

Example with math:

```
import math

print(math.sqrt(16))      # Output: 4.0
print(math.pi)            # Output: 3.141592653589793
```

Third-party Module

Example with numpy (after installation):

```
pip install numpy
```

```
import numpy as np
```

```
a = np.array([1, 2, 3])
print(a * 2)
```

__name__ and __main__ in Python

In every Python module, there is a special built-in variable called __name__.

- When a Python file is run **directly**, the value of __name__ is set to "__main__".
- When a Python file is **imported as a module**, the value of __name__ is set to the **module's file name** (without .py).

Use of if __name__ == "__main__":

To **control the execution** of certain parts of your code. It allows you to write:

- Code that runs **only when the file is executed directly**
- Code that does **not run** when the file is **imported** in another module

Example:

File: Marvellous_Module.py

```
def Display():
    print("Welcome from Marvellous Infosystems!")

print("This line always runs.")
print("Value of __name__ is:", __name__)

if __name__ == "__main__":
    print("This code runs only when run directly.")
    Display()
```

Scenario 1: Run directly

python Marvellous_Module.py

Output:

This line always runs.

Value of __name__ is: __main__

This code runs only when run directly.

Welcome from Marvellous Infosystems!

Scenario 2: Import into another file

```
# main.py
import module_demo
```

Output:

This line always runs.

Value of __name__ is: module_demo

Display() is not called, because the if __name__ == "__main__" block is skipped.

Multiprocessing

The multiprocessing library uses separate memory space, multiple CPU cores.

Consider below application which demonstrates the concept of Multiprocessing

```

import multiprocessing
import os

print("---- Marvellous Infosystems by Piyush Khairnar----")
print("Demonstration of Multiprocessing")

def fun(number):
    print('parent process of fun:', os.getppid())
    print('process id of fun:', os.getpid())
    for i in range(number):
        print(i)

def gun(number):
    print('parent process of gun:', os.getppid())
    print('process id of gun:', os.getpid())
    for i in range(number):
        print(i)

if __name__ == "__main__":
    print("Total cores available : ",multiprocessing.cpu_count())

    print('parent process of main:', os.getppid())
    print('process id of main:', os.getpid())
    number = 3
    result = None

    p1 = multiprocessing.Process(target=fun, args=(number,))
    p2 = multiprocessing.Process(target=gun, args=(number,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
  
```

Output of above application

```
MacBook- Pro-de-MARVELLOUS: Today marvellous$ python |  
multip.py  
---- Marvellous Infosystems by Piyush Khairnar-----  
Demonstration of Multiprocessing  
('Total cores available:', 4)  
('parent process of main:', 1251)  
('process id of main:', 2464)  
('parent process of fun:', 2464)  
('process id of fun:', 2466)  
0  
1  
2  
('parent process of gun:', 2464)  
('process id of gun:', 2467)  
0  
1  
2  
MacBook- Pro-de-MARVELLOUS: Today marvellous$ █
```

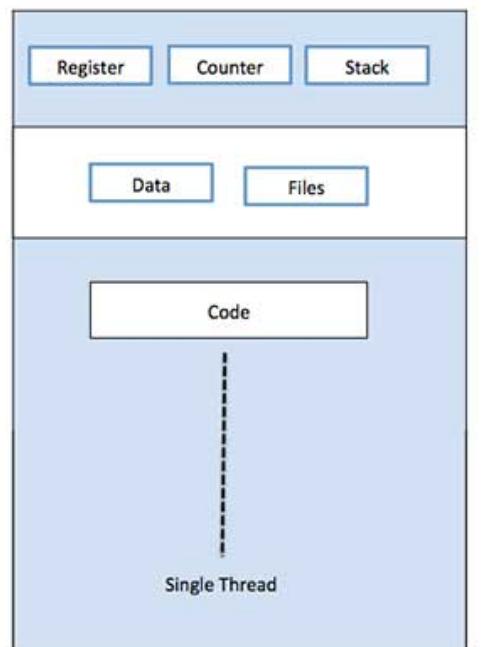


Multitasking

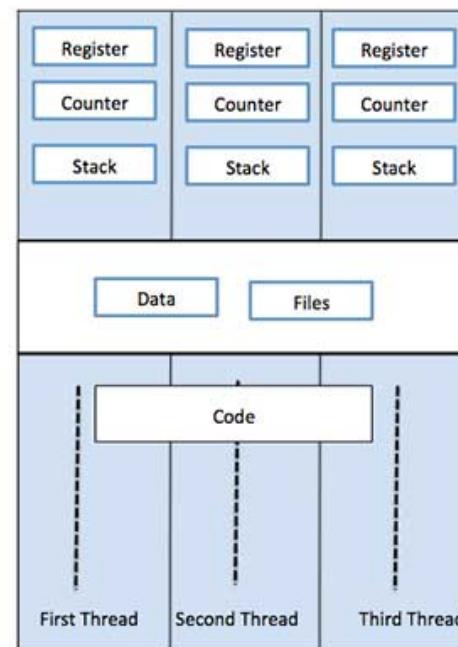
- Multitasking, in an operating system, is allowing a user to perform more than one computer task (such as the operation of an application program) at a time.
- The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information.
- Microsoft Windows 2000, IBM's OS/390, and Linux are examples of operating systems that can do multitasking (almost all of today's operating systems can).
- When you open your Web browser and then open Word at the same time, you are causing the operating system to do multitasking.
- Being able to do multitasking doesn't mean that an unlimited number of tasks can be juggled at the same time.
- Each task consumes system storage and other resources.
- As more tasks are started, the system may slow down or begin to run out of shared storage.
- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilise the CPU.

Multitasking can be achieved in two ways:

1. Process-based Multitasking (Multiprocessing)
2. Thread-based Multitasking (Multithreading)



Single Process P with single thread



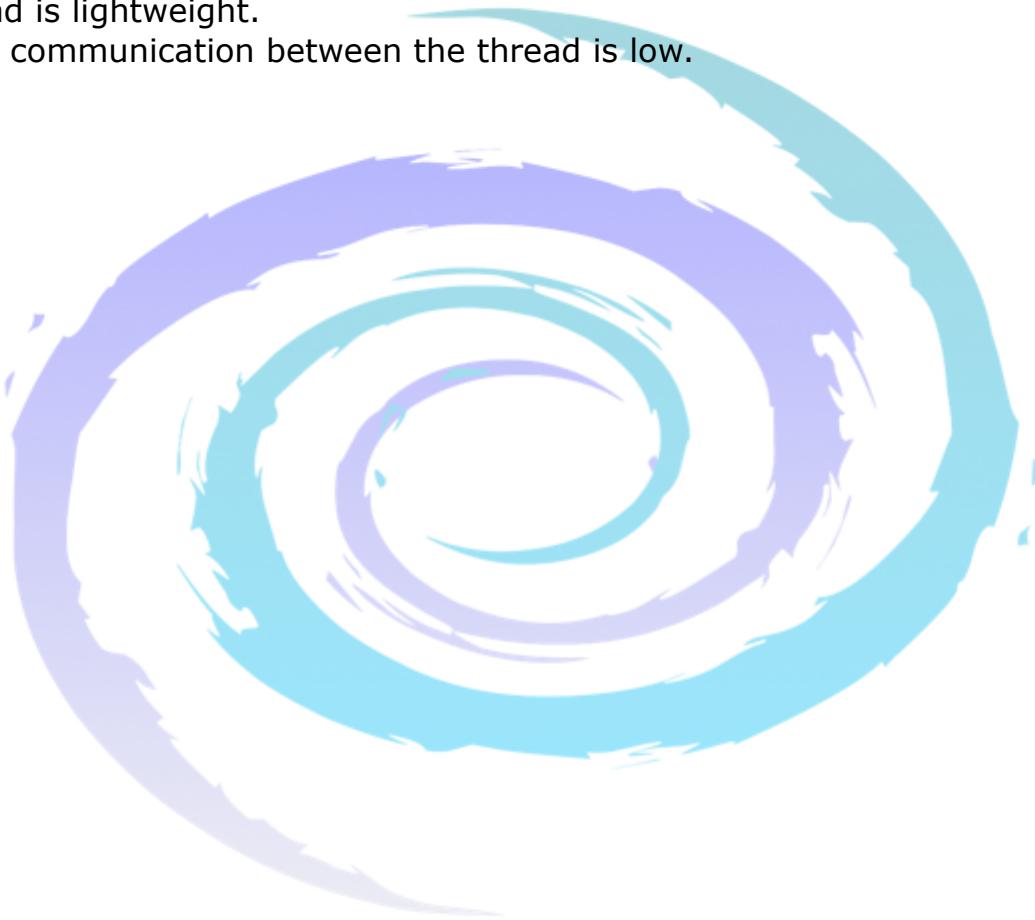
Single Process P with three threads

Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.



Multitasking in Python

Program

A **program** is a set of static instructions written in any programming language to perform a specific task.

Example: A simple program that adds two numbers:

```
a = 10
b = 5
print("Sum =", a + b)
```

Process

A **process** is a program in **execution**. It has its own memory space, code, data, and system resources. Every process runs independently.

To check running processes in Python:

```
import os
print("Current Process ID:", os.getpid())
```

PID – Process ID

Each process is given a **unique ID** called a **PID (Process ID)** by the operating system to identify it.

Use `os.getpid()` to get PID:

```
import os
print("PID:", os.getpid())
```

Thread

A **thread** is the smallest unit of execution within a process. A process can have **multiple threads** sharing the same memory space.

Difference:

- **Process** = Runs independently with its own memory.
- **Thread** = Runs within a process and shares the same memory.

TID – Thread ID

Each thread inside a process has a unique **Thread ID**.

Get Thread ID using `threading.get_ident()`:

```

import threading

def task():
    print("Thread ID:", threading.get_ident())

t1 = threading.Thread(target=task)
t1.start()
t1.join()
  
```

Address Space of a Process

The **address space** refers to the memory allocated to a process. All threads in a process share:

- Code
- Data
- File descriptors
- Heap & stack (different stack for each thread)

Threads share the same address space. Processes do **not**.

Multitasking

Multitasking is the ability of the CPU to execute **multiple tasks** simultaneously or in quick succession.

Types:

- **Process-based multitasking:** Running multiple applications.
- **Thread-based multitasking:** Multiple threads in a single application.

Multithreading

Multithreading is a technique where **multiple threads** run concurrently within a single process.

Python Multithreading Example:

```

python

import threading

def fun():
    print("Hello from", threading.current_thread().name)

t1 = threading.Thread(target=fun)
t2 = threading.Thread(target=fun)

t1.start()
  
```

```
t2.start()

t1.join()
t2.join()
```

Multicore Programming

Multicore programming means using **multiple cores of a CPU** to execute code in **parallel**, increasing performance.

Python's **multiprocessing** module is used for this.

Example: Using multiple processes:

```
from multiprocessing import Process
import os

def fun():
    print("Running process with PID:", os.getpid())

if __name__ == "__main__":
    for _ in range(4):
        p = Process(target=fun)
        p.start()
```

multiprocessing.Pool – Parallel Task Execution Made Easy

Pool allows you to **manage a pool of worker processes** to run tasks in parallel. It automatically handles process creation and load distribution.

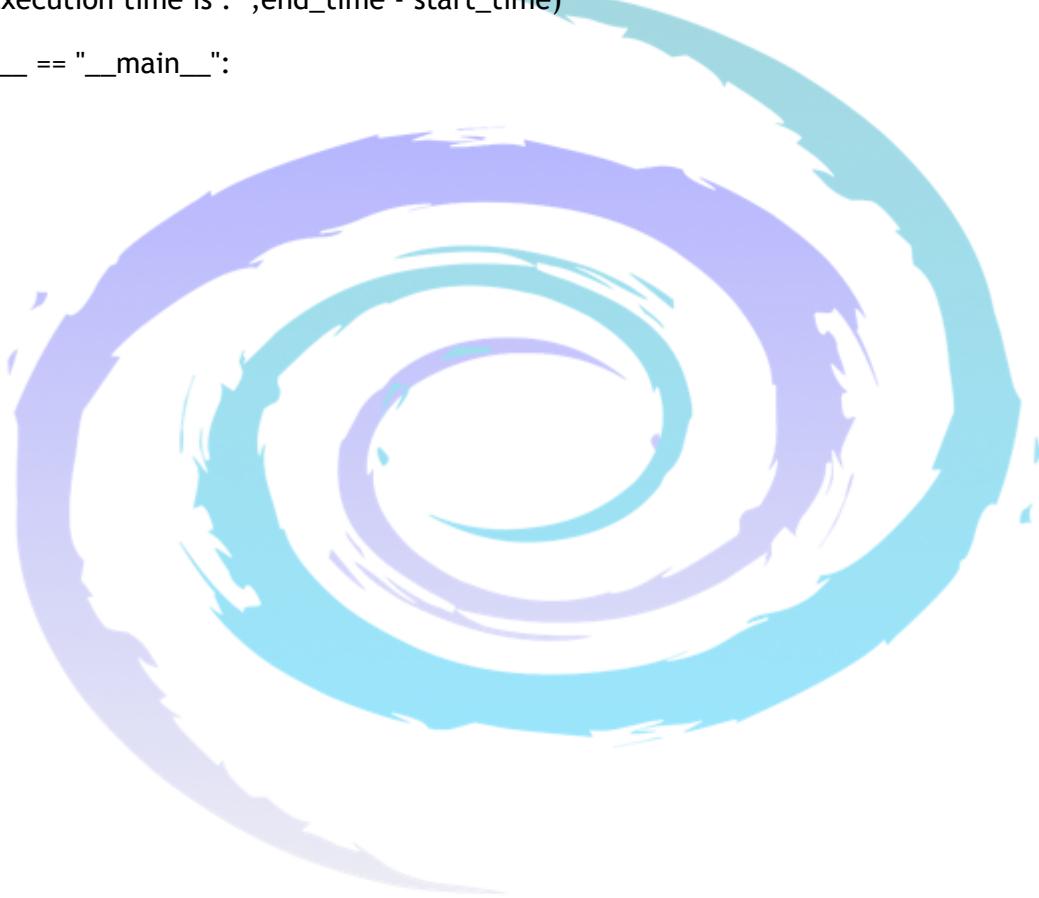
Example: Square numbers using Pool

```
def fun(no):
    print("PID is : ",os.getpid())
    sum = 0
    for i in range(1,no):
        sum = sum + (i*i*i)
    return sum

def main():
    start_time = time.time()

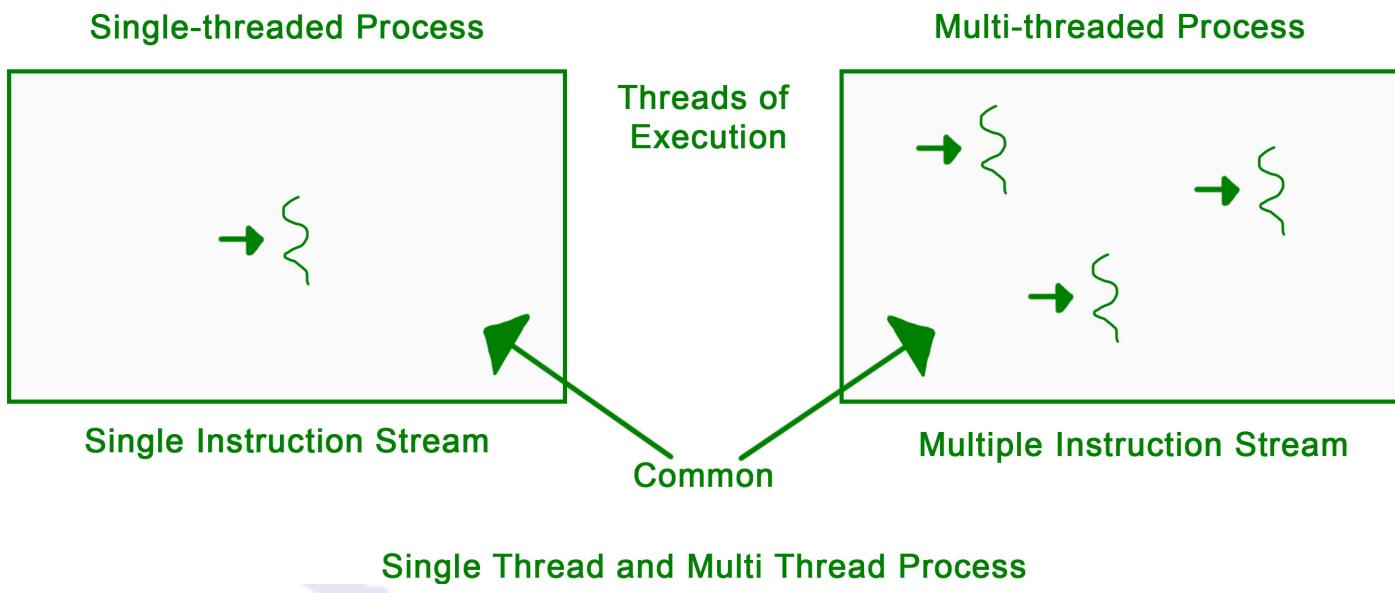
    data = [1000000,2000000,3000000,4000000,5000000,6000000,7000000,8000000,9000000,10000000]
```

```
result = []\n\np = multiprocessing.Pool()\nresult = p.map(fun,data)\np.close()\np.join()\nprint(result)\nend_time = time.time()\nprint("Execution time is : ",end_time - start_time)\nif __name__ == "__main__":\n    main()
```



Multithreading

- Python is a multi-threaded programming language which means we can develop multi-threaded program using python.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- By definition, multitasking is when multiple processes share common processing resources such as a CPU.
- Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.
- Each of the threads can run in parallel.
- The OS divides processing time not only among different applications, but also among each thread within an application.
- Multi-threading enables us to write in a way where multiple activities can proceed concurrently in the same program.



Thread :

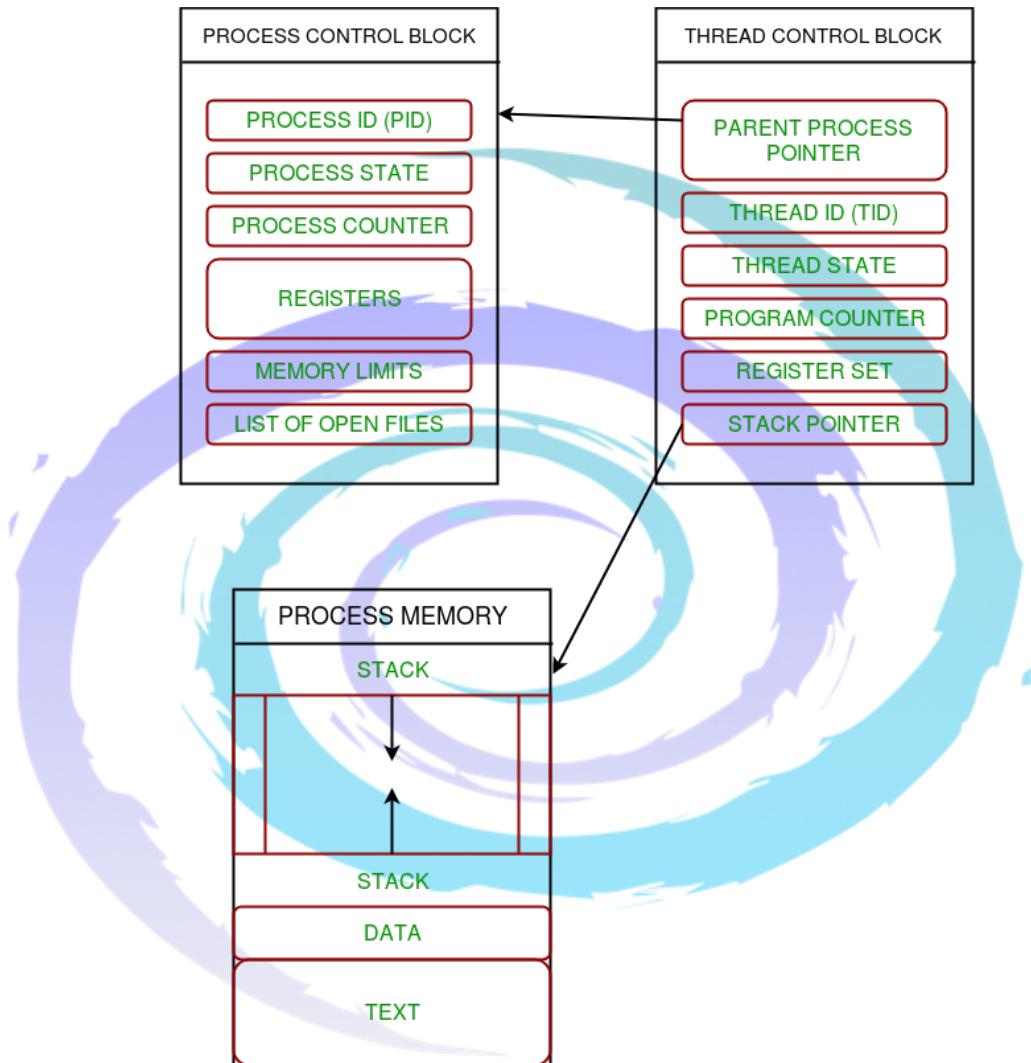
- A thread is an entity within a process that can be scheduled for execution.
- Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
- In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code.
- For simplicity, we can assume that a thread is simply a subset of a process!

A thread contains all this information in a Thread Control Block (TCB):

- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter:** a register which stores the address of the instruction currently being executed by thread.

- **Thread state:** can be running, ready, waiting, start or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

Consider the below diagram to understand the relation between process and its thread



Consider below application which demonstrates the concept of Multithreading

```

import threading

print("---- Marvellous Infosystems by Piyush Khairnar----")

print("Demonstration of Multithreading")

def fun(number):
    for i in range(number):
        print(i)

def gun(number):
    for i in range(number):
        print(i)

if __name__ == "__main__":
    number = 5
    thread1 = threading.Thread(target=fun, args=(number,))

    thread2 = threading.Thread(target=gun, args=(number,))

    # Will execute both in parallel
    thread1.start()
    thread2.start()

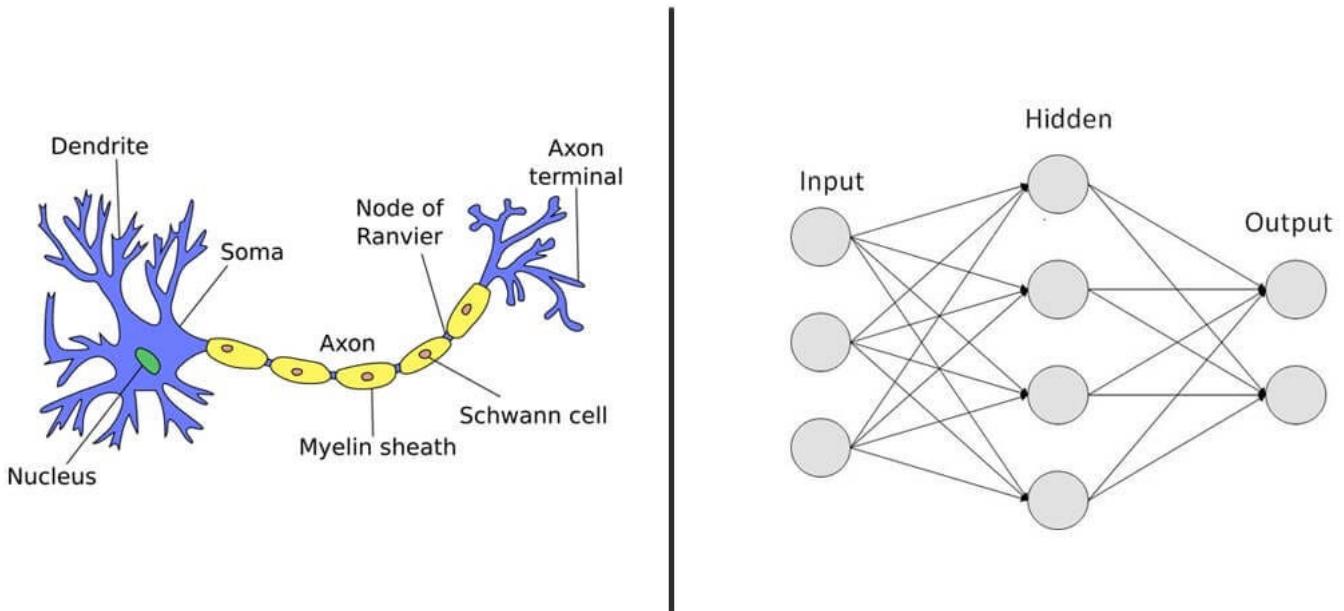
    # Joins threads back to the parent process, which is this
    # program
    thread1.join()
    thread2.join()
  
```

Output of above application

```

MacBook-Pro-de-MARVELLOUS: Today marvellous$ python multithreading.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Multithreading
0
  1
  2
  3
  4
  3
  4
MacBook-Pro-de-MARVELLOUS: Today marvellous$ █
  
```

Biological Neural Network & Artificial Neural Network

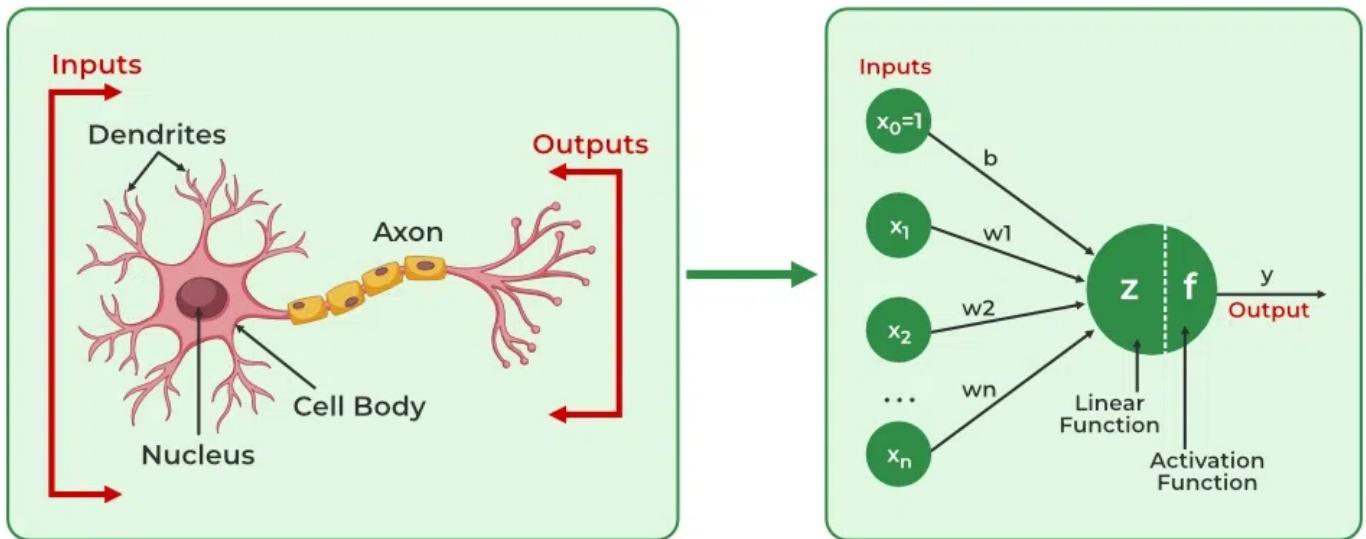


Biological Neural Network (BNN)

- Found in the **human brain and nervous system**.
- Consists of **billions of biological neurons** (brain cells).
- Each neuron is connected to thousands of others via **synapses**.
- Works on **electrochemical signals**.
- **Learning mechanism:** Adjusts the strength of synaptic connections over time (called **synaptic plasticity**).
- Extremely **parallel, adaptive, and energy-efficient**.

Artificial Neural Network (ANN)

- A **mathematical/computational model** inspired by the brain's BNN.
- Consists of **artificial neurons (perceptrons)** arranged in layers.
- Neurons process **numerical data**, not chemical signals.
- **Connections (weights):** Store learned importance of inputs.
- **Learning mechanism:** Adjusts weights using algorithms like **backpropagation** and **gradient descent**.
- Runs on **CPUs/GPUs** with high power consumption.



Biological Neuron (Human Brain)

A **neuron** is the **basic unit of the brain** and nervous system. It processes and transmits information using **electrical and chemical signals**.

Structure

1. **Dendrites** – Branch-like structures that receive signals from other neurons.
2. **Cell Body (Soma)** – Processes the incoming signals.
3. **Axon** – A long extension that transmits signals away to other neurons.
4. **Synapse** – Junction where the axon of one neuron connects to the dendrite of another, using neurotransmitters.

Working

- **Input:** Signals arrive at dendrites.
- **Processing:** Soma integrates signals; if the total signal exceeds a threshold, the neuron “fires.”
- **Output:** Axon carries the electrical impulse to synapses.
- **Learning:** Repeated usage strengthens or weakens synaptic connections (**Synaptic Plasticity**).

Human brain has **~86 billion neurons**, each connecting to thousands of others → massive parallelism.

Artificial Neuron (Perceptron)

An **Artificial Neuron** is a **mathematical model** inspired by biological neurons. It takes inputs, processes them, and produces an output.

Structure

1. **Inputs (x_1, x_2, \dots, x_n):** Features/data points fed into the model.
2. **Weights (w_1, w_2, \dots, w_n):** Each input has an importance factor.
3. **Bias (b):** A constant added to control the activation threshold.
4. **Summation Function:** Weighted sum of inputs.

$$z = (w_1x_1 + w_2x_2 + \dots + w_nx_n) + b$$
5. **Activation Function:** Decides the output of the neuron (fire or not).

Activation Functions (decide non-linearity)

- **Step function:** If $z >$ threshold $\rightarrow 1$ else 0
- **Sigmoid:** Smooth curve between 0 and 1
- **Tanh:** Values between -1 and 1
- **ReLU (Rectified Linear Unit):** $\max(0, z)$ (most popular in DL)

Working

1. Inputs are multiplied by weights.
2. Bias is added.
3. Activation function applied.
4. Output sent to next layer.

This mimics “dendrites + soma + axon” of biological neurons.

Biological vs Artificial Neuron

Aspect	Biological Neuron	Artificial Neuron
Unit	Brain cell	Mathematical function
Inputs	Signals from dendrites	Features (x_1, x_2, \dots)
Weights	Synaptic strength	Numerical weights (w_1, w_2, \dots)
Processing	Soma integrates signals	Weighted sum + bias
Threshold	Fires if strong enough	Activation function
Output	Signal via axon	Predicted value (y)
Learning	Synaptic plasticity	Weight adjustment via algorithms (e.g., backpropagation)

Consider below difference between components of Artificial neuron and Human neuron

Component	Human Neuron	Artificial Neuron
Dendrites	Branch-like extensions that receive signals from other neurons.	Input layer — Receives numerical data from features or previous neurons.
Cell Body (Soma)	Processes incoming signals and determines whether to send a signal forward.	Summation function — Adds up weighted inputs.
Axon	Long projection that transmits electrical impulses to other neurons.	Output value — Sent to next layer's neurons.
Synapse	Connection point where signals are passed chemically to another neuron.	Weights — Determine the strength/importance of the connection.

Object Oriented Paradigms in Python

In object oriented programming there are four concepts as

Encapsulation :

Encapsulation is considered as binding characteristics (Variables) and behaviours (Methods) together. To bind characteristics and behaviour we have to design class.

Consider the below example which contains all types of characteristics and all types of behaviours.

class Employee:

```
CompanyName = "Marvellous"      # Class variable
```

```
def __init__(self, A, B, C):      # Constructor
```

```
    print("Inside constructor")
```

```
    self.Name = A                # Instance variable
```

```
    self.Salary = B               # Instance variable
```

```
    self.City = C                # Instance variable
```

```
def __del__(self):              # Destructor
```

```
    print("Inside destructor")
```

```
def DisplayInfo(self):          # Instance method
```

```
    print("Inside Instance method DisplayInfo")
```

```
    print("Employee Name : "+self.Name)
```

```
    print("Employee Salary : ",self.Salary)
```

```
    print("Employee City : "+self.City)
```

```
@classmethod
```

```
def DisplayCompanyDetails(cls):   # Class method
```

```
    print("Inside Class method DisplayCompanyDetails")
```

```
    print("Company Name : "+cls.CompanyName)
```

```
@staticmethod
```

```
def DisplayCompanyPolicy():
```

```
    print("Inside static method DisplayCompanyPolicy")
```

```

print("All employees are 18+")
print("Working hours are 9 to 6")
print("Holidays : Saturday & Sunday")

def main():
    print("Class variable : "+Employee.CompanyName)
    Employee.DisplayCompanyDetails()

emp1 = Employee('Rahul',15000,'Pune')      # Object creation
emp2 = Employee('Pooja',25000,'Mumbai')    # Object creation

print("Employee Name : "+emp1.Name)
print("Employee Salary : ",emp1.Salary)
print("Employee City : "+emp1.City)
emp2.DisplayInfo()
Employee.DisplayCompanyPolicy()
del emp1
del emp2

if __name__ == "__main__":
    main()
  
```

Abstraction :

Abstraction is considered as hiding something from outsider entity. To achieve abstraction we have to use concept of access modifiers.

◆ Access Modifiers in Python:

- **public:** Accessible from anywhere (default)
- **_protected:** Accessible within class and subclasses
- **__private:** Accessible only within class (name mangled)

class BankAccount:

```

def __init__(self, balance):
    self.__balance = balance # private
  
```

```

def deposit(self, amount):
    self.__balance += amount
def get_balance(self):
    return self.__balance
acc = BankAccount(1000)
acc.deposit(500)
print(acc.get_balance())      # Output: 1500
# print(acc.__balance)       # AttributeError
  
```

Polymorphism :

Polymorphism is defined as single name and multiple behaviour. We can achieve polymorphism using the concept of method overriding.

- Method Overriding: Subclass overrides a method of the parent class.

```
class Parent:
```

```

def show(self):
    print("Parent class")
  
```

```
class Child(Parent):
```

```

def show(self):
    print("Child class")
  
```

```
obj = Child()
```

```
obj.show() # Child class
```

Inheritance :

Inheritance means reusability. By using inheritance one class can acquire all the characteristics and behaviour of another class. Inheritance allows one class to inherit properties and methods from another class.

Types in Python:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        print("Person constructor called")  
  
    def display(self):  
        print(f"Name: {self.name}, Age: {self.age}")  
  
class Student(Person):  
    def __init__(self, name, age, student_id):  
        super().__init__(name, age) # calling parent constructor  
        self.student_id = student_id  
        print("Student constructor called")  
  
    def display(self):  
        super().display() # calling parent display() method  
        print(f"Student ID: {self.student_id}")  
  
# Creating a Student object  
s = Student("Rahul", 21, "ST1024")  
s.display()
```

Python Programming Paradigms

Procedural, Functional, Object-Oriented , Scripting

Python supports **multi-paradigm programming**, allowing us to write the code in Procedural ,Object Oriented or Function or Scripting way.

Procedural Programming

Procedural Programming follows a **step-by-step** approach where code is organized into **procedures or functions**.

Key Features:

- Sequential execution
- Uses functions and global variables
- Simple, good for basic tasks

Example:

```
def Addition(No1, No2):
    return No1 + No2

Ret = Addition(10,11)
print(Ret)
```

Functional Programming

Functional Programming emphasizes using **pure functions**, avoids mutable data, and encourages **function chaining**.

Key Features:

- Stateless functions
- Uses `map()`, `filter()`, `reduce()`
- No side effects

Example:

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared) # [1, 4, 9, 16]
```

Object-Oriented Programming (OOP)

OOP structures code into **classes** and **objects**, focusing on **real-world modeling** using characteristics (attributes) and behavior (methods).

OOP Principles:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Example:

```
class Arithmetic:
    def __init__(self, A, B):
        self.No1 = A
        self.No2 = A

    def Addition(self):
        return self.No1 + self.No2

Obj = Arithmetic(10,11)
Ret = obj.Addition()
print(Ret)
```

Scripting Programming

Scripting involves writing **short programs (scripts)** to automate tasks. Scripts are usually **interpreted**, not compiled.

Key Features:

- Fast to write and execute
- Useful for automation and utility tasks
- Often used for file handling, system operations, automation, etc.

Example: Automating file reading

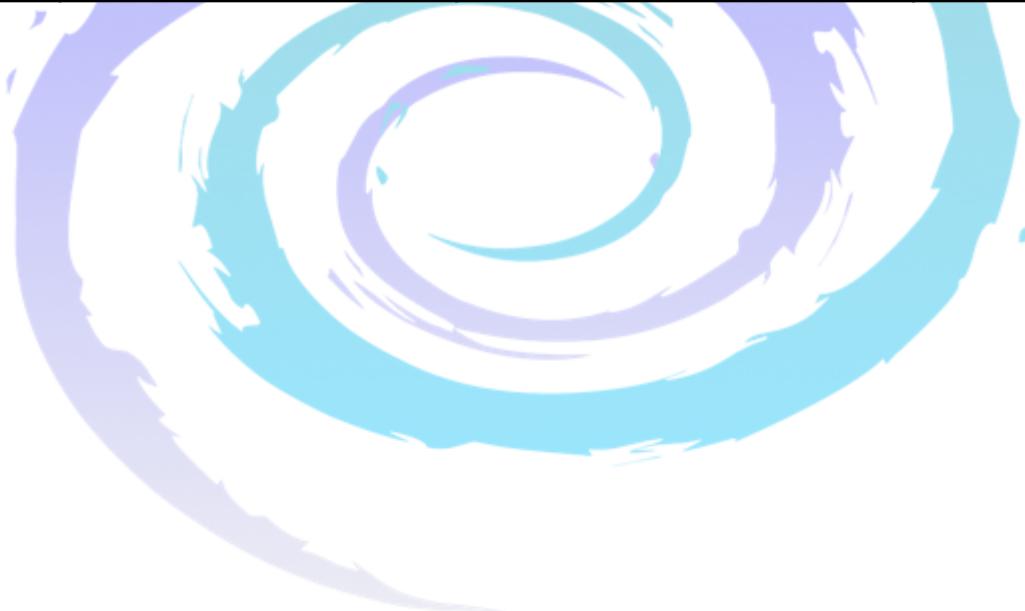
```
with open("Marvellous.txt.txt", "r") as file:
    content = file.read()
    print(content)
```

Common Uses:

- Automation (e.g., renaming files, backups)
- Web scraping (e.g., using `requests` and `BeautifulSoup`)
- Shell scripting (e.g., batch tasks with Python)

Comparison Table

Paradigm	Focus	Use Case	Code Example Type
Procedural	Sequence of steps/functions	Small scripts, utilities	Functions
Functional	Pure functions, immutability	Data transformation, pipelines	<code>map</code> , <code>filter</code> , <code>lambda</code>
OOP	Real-world objects and logic	Scalable apps, reusable code structure	Classes & Objects
Scripting	Quick automation scripts	File operations, task automation	Top-level procedural



OpenCV and CNN in Deep Learning

OpenCV :

- **OpenCV (Open Source Computer Vision Library)** is a powerful library for image processing and computer vision.
- Written in C++ with Python bindings.
- Used in:
 - Image transformations (resize, rotate, filter).
 - Feature detection (edges, corners, faces).
 - Video analysis.
 - Object recognition (integrated with CNN/DL models).

Key Concepts

- **Image Representation:** Stored as a matrix (pixels). Each pixel has values for **BGR** (Blue, Green, Red) in OpenCV.
- **Grayscale Conversion:** Simplifies image → single intensity value (0–255).
- **Edge Detection:** Finds sudden intensity changes (object boundaries).
- **Integration with Deep Learning:** OpenCV provides pre-processing before sending images to CNNs.

CNN (Convolutional Neural Network)

- CNN is a type of deep learning model specialized for images.
- Learns features **automatically** (edges → shapes → objects).
- Key layers:
 - **Convolution Layer:** Extracts features using filters.
 - **Pooling Layer:** Reduces size, keeps important features.
 - **Fully Connected Layer:** Classifies objects.
- Used in:
 - Image classification.
 - Object detection.
 - Face recognition.

Code Explanations

OpenCVPixel.py

Concept: Show image pixel values.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# 1. Read image
img = cv2.imread("sample.jpg")

# 2. Convert to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# 3. Display image + pixel grid
plt.figure(figsize=(10,5))
plt.subplot(1,2,1); plt.imshow(gray, cmap="gray");
plt.title("Grayscale Image"); plt.axis("off")
plt.subplot(1,2,2); plt.imshow(gray, cmap="gray");
plt.colorbar(label="Pixel Value"); plt.title("Pixel Values")
plt.show()
```

Learning:

- How images are stored as pixel grids.
- Visualization of intensity values.

Edge.py

Concept: Detect edges in an image.

```
import cv2

# Read image
img = cv2.imread("sample.jpg", 0) # grayscale

# Canny Edge Detection
edges = cv2.Canny(img, 100, 200)

cv2.imshow("Original", img)
cv2.imshow("Edges", edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Learning:

- Edge detection helps CNN identify **boundaries of objects**.
- OpenCV's **Canny** algorithm works in 3 steps:
 1. Noise reduction.
 2. Gradient calculation.
 3. Detecting edges.

CNNedgeDetection.py

Concept: Using CNN for edge detection/classification.

```

import cv2
import numpy as np
from tensorflow.keras.applications import MobileNetV2,
preprocess_input, decode_predictions

# Load pre-trained CNN
model = MobileNetV2(weights="imagenet")

# Capture from webcam
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    img = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    img_resized = cv2.resize(img, (224, 224))
    x = np.expand_dims(img_resized, axis=0).astype(np.float32)
    x = preprocess_input(x)

    # Prediction
    preds = model.predict(x, verbose=0)
    decoded = decode_predictions(preds, top=1)[0][0]
    label = f"{decoded[1]}: {decoded[2]*100:.1f}%"

    # Show on screen
    cv2.putText(frame, label, (16, 40), cv2.FONT_HERSHEY_SIMPLEX, 1,
(0,255,0), 2)
    cv2.imshow("CNN Classification", frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()

```

Learning:

- Webcam captures **real-time frames**.
- Frames → CNN (MobileNetV2) → Predict object.
- CNN automatically detects **edges** → **shapes** → **object label**.



PIP

PIP Install Packages

- PIP is a recursive acronym that stands for “PIP Installs Packages” or “Preferred Installer Program”.
- It’s a command-line utility that allows you to install, reinstall, or uninstall PyPI packages with a simple and straightforward command: pip.
- That means it’s a tool that allows us to install and manage additional libraries and dependencies that are not distributed as part of the standard library.
- Package management is so important that pip has been included with the Python installer since versions 3.4 for Python 3 and 2.7.9 for Python 2, and it’s used by many Python projects, which makes it an essential tool for every Python developer.

Install PIP

If you’re using Python 2.7.9 (or greater) or Python 3.4 (or greater), then PIP comes installed with Python by default.

If you’re using an older version of Python, you’ll need to use the installation steps below. Otherwise, skip to the bottom to learn how to start using PIP.

Install PIP on Windows (Windows 7, Windows 8.1, and Windows 10)

1. Download get-pip.py to a folder on your computer (<https://bootstrap.pypa.io/get-pip.py>).
2. Open a command prompt and navigate to the folder containing get-pip.py.
3. Run the following command: `python get-pip.py`
4. Pip is now installed.

You can verify that Pip was installed correctly by opening a command prompt and entering the following command:

`pip -V`

How to Install PIP on Linux

If your Linux distro came with Python already installed, you should be able to install PIP using your system’s package manager.

Advanced Package Tool (Python 2.x)

`sudo apt-get install python-pip`

Advanced Package Tool (Python 3.x)

`sudo apt-get install python3-pip`

Pandas Python Library



Pandas is a software library written for the Python programming language for data manipulation and analysis.

In particular, it offers data structures and operations for manipulating numerical tables and time series.

It is free software released under the three-clause BSD license.

The name is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

This is the most important library for Machine Learning because we can deal with our data set by using Pandas.

Install Pandas library using pip as

sudo pip install pandas

Features of Pandas

- DataFrame object for data manipulation with integrated indexing.
- Tools for reading and writing data between in-memory data structures and different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of data sets.
- Label-based slicing, fancy indexing, and subsetting of large data sets.
- Data structure column insertion and deletion.
- Group by engine allowing split-apply-combine operations on data sets.
- Data set merging and joining.
- Hierarchical axis indexing to work with high-dimensional data in a lower-dimensional data structure.
- Time series-functionality: Date range generation[4] and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging.
- Provides data filtration.

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure.

For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, size immutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, ...

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

10	23	56	17	52	61	73	90	26	72
----	----	----	----	----	----	----	----	----	----

DataFrame

DataFrame is a two-dimensional array with heterogeneous data. For example,

Marvellous Infosystems Training Data set

Weight	Pattern	Label
35	Rough	Tennis
47	Rough	Tennis
90	Smooth	Cricket
48	Rough	Tennis
90	Smooth	Cricket
35	Rough	Tennis
92	Smooth	Cricket
35	Rough	Tennis
35	Rough	Tennis
35	Smooth	Cricket
43	Rough	Tennis
110	Smooth	Cricket
35	Rough	Tennis
95	Smooth	Cricket

The table represents the data of a balls.

We are going to user to use this data set in machine learning application.

The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

We consider data frame as our excel sheet.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Panel

Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

We consider panel as excel file which contains multiple excel sheets (Data frame).

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable



Preserving Machine Learning Models on HDD using joblib with Pipeline

When a machine learning model is trained, it learns from data and stores patterns/parameters.

If you close the program, the trained parameters are lost unless you **save the model**. Saving:

- Avoids retraining every time
- Saves computation time
- Enables deployment or sharing

joblib :

- Specially optimized for **large NumPy arrays** (common in ML models)
- Faster than pickle for ML models
- Easy syntax: just **dump** and **load**

Saving a Model to HDD

```
from pathlib import Path
import joblib
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

```
# Configuration details
```

```
ARTIFACTS = Path("artifacts_sample")
ARTIFACTS.mkdir(exist_ok=True)
MODEL_PATH = ARTIFACTS / "iris_pipeline.joblib"
RANDOM_STATE = 42
TEST_SIZE = 0.2
```

```

def main():

    iris = load_iris()

    X = iris.data

    Y = iris.target

    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=TEST_SIZE,
random_state=RANDOM_STATE)

    pipe = Pipeline([
        ("scalar", StandardScaler()),
        ("clf", LogisticRegression(max_iter=1000))
    ])

    pipe.fit(X_train,Y_train)

    Y_pred = pipe.predict(X_test)

    print("Accuracy Score : ",accuracy_score(Y_test,Y_pred))

    joblib.dump(pipe,MODEL_PATH)

if __name__ == "__main__":
    main()
  
```

Loading a Model from HDD

```

from pathlib import Path

import joblib

import numpy as np

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
  
```

Configuration details

```
ARTIFACTS = Path("artifacts_sample")
ARTIFACTS.mkdir(exist_ok=True)
MODEL_PATH = ARTIFACTS / "iris_pipeline.joblib"
RANDOM_STATE = 42
TEST_SIZE = 0.2

def main():
    Labels = ['Setosa','Versicolor','Virginica']

    pipe = joblib.load(MODEL_PATH)
    sample = np.array([[5.1,3.5,1.4,0.2]])
    Y_pred = pipe.predict(sample)[0]
    print("Predicted result is : ",Labels[Y_pred])

if __name__ == "__main__":
    main()
```

Pipeline in Machine Learning

A **Pipeline** is a way to **chain preprocessing steps and a model together** into one object.

Advantages Pipeline :

- Keeps **data transformations & model** in one place
- Ensures the **same preprocessing** is applied during both training and prediction
- Reduces human error in applying transformations
- Makes saving/loading easier (one file instead of many)

Creating and Saving a Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
import joblib
from sklearn.datasets import load_iris

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Create pipeline: Scaling + Logistic Regression
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

# Train pipeline
pipeline.fit(X, y)

# Save pipeline to HDD
joblib.dump(pipeline, 'iris_pipeline.joblib')
print("Pipeline saved as iris_pipeline.joblib")
```

Loading and Using a Pipeline

```
# Load pipeline
loaded_pipeline = joblib.load('iris_pipeline.joblib')

# Predict with pipeline
sample_data = [[5.1, 3.5, 1.4, 0.2]]
prediction = loaded_pipeline.predict(sample_data)

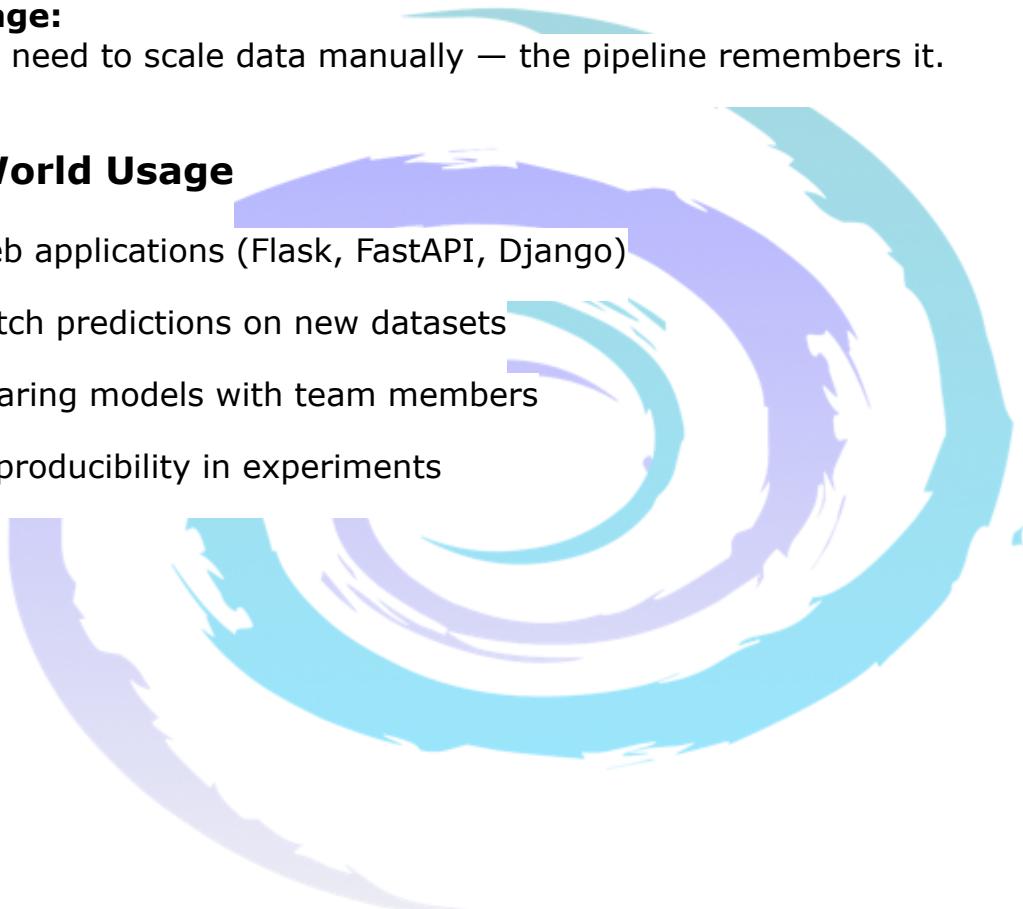
print("Predicted class:", prediction)
```

Advantage:

We don't need to scale data manually — the pipeline remembers it.

Real-World Usage

- Web applications (Flask, FastAPI, Django)
- Batch predictions on new datasets
- Sharing models with team members
- Reproducibility in experiments



Preserving Machine Learning Models on HDD using joblib with Pipeline

When a machine learning model is trained, it learns from data and stores patterns/parameters.

If you close the program, the trained parameters are lost unless you **save the model**. Saving:

- Avoids retraining every time
- Saves computation time
- Enables deployment or sharing

joblib :

- Specially optimized for **large NumPy arrays** (common in ML models)
- Faster than pickle for ML models
- Easy syntax: just **dump** and **load**

Saving a Model to HDD

```
from pathlib import Path
import joblib
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

```
# Configuration details
```

```
ARTIFACTS = Path("artifacts_sample")
ARTIFACTS.mkdir(exist_ok=True)
MODEL_PATH = ARTIFACTS / "iris_pipeline.joblib"
RANDOM_STATE = 42
TEST_SIZE = 0.2
```

```

def main():

    iris = load_iris()

    X = iris.data

    Y = iris.target

    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=TEST_SIZE,
random_state=RANDOM_STATE)

    pipe = Pipeline([
        ("scalar", StandardScaler()),
        ("clf", LogisticRegression(max_iter=1000))
    ])

    pipe.fit(X_train,Y_train)

    Y_pred = pipe.predict(X_test)

    print("Accuracy Score : ",accuracy_score(Y_test,Y_pred))

    joblib.dump(pipe,MODEL_PATH)

if __name__ == "__main__":
    main()
  
```

Loading a Model from HDD

```

from pathlib import Path

import joblib

import numpy as np

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
  
```

Configuration details

```
ARTIFACTS = Path("artifacts_sample")
ARTIFACTS.mkdir(exist_ok=True)
MODEL_PATH = ARTIFACTS / "iris_pipeline.joblib"
RANDOM_STATE = 42
TEST_SIZE = 0.2

def main():
    Labels = ['Setosa','Versicolor','Virginica']

    pipe = joblib.load(MODEL_PATH)
    sample = np.array([[5.1,3.5,1.4,0.2]])
    Y_pred = pipe.predict(sample)[0]
    print("Predicted result is : ",Labels[Y_pred])

if __name__ == "__main__":
    main()
```

Pipeline in Machine Learning

A **Pipeline** is a way to **chain preprocessing steps and a model together** into one object.

Advantages Pipeline :

- Keeps **data transformations & model** in one place
- Ensures the **same preprocessing** is applied during both training and prediction
- Reduces human error in applying transformations
- Makes saving/loading easier (one file instead of many)

Creating and Saving a Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
import joblib
from sklearn.datasets import load_iris

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Create pipeline: Scaling + Logistic Regression
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

# Train pipeline
pipeline.fit(X, y)

# Save pipeline to HDD
joblib.dump(pipeline, 'iris_pipeline.joblib')
print("Pipeline saved as iris_pipeline.joblib")
```

Loading and Using a Pipeline

```
# Load pipeline
loaded_pipeline = joblib.load('iris_pipeline.joblib')

# Predict with pipeline
sample_data = [[5.1, 3.5, 1.4, 0.2]]
prediction = loaded_pipeline.predict(sample_data)

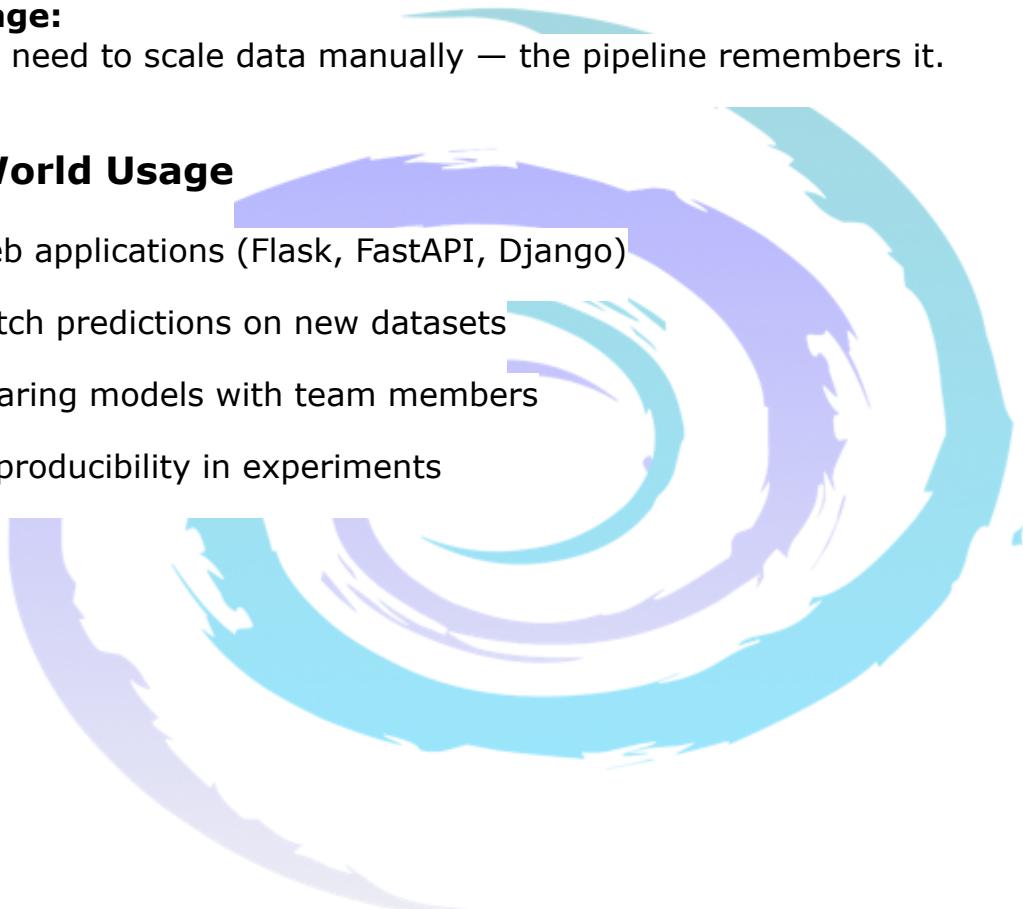
print("Predicted class:", prediction)
```

Advantage:

We don't need to scale data manually — the pipeline remembers it.

Real-World Usage

- Web applications (Flask, FastAPI, Django)
- Batch predictions on new datasets
- Sharing models with team members
- Reproducibility in experiments



Progression of Python Programming Journey

Python Programming

- 1. Introduction & Basics
 - | — History and Features of Python
 - | — Installation (Python + IDE setup)
 - | — First Program ("Jay Ganesh")
 - | — Python Syntax, Indentation, Comments
- 2. Core Data Types
 - | — Numbers (int, float, complex)
 - | — Strings (creation, slicing, formatting)
 - | — Booleans
 - | — Type Casting & Input/Output
- 3. Sequence Data Types
 - | — List
 - | | — Indexing, Slicing
 - | | — Methods (append, insert, pop, sort, etc.)
 - | — Tuple
 - | — Set (union, intersection, difference)
 - | — Dictionary (keys, values, CRUD operations)
- 4. Operators & Expressions
 - | — Arithmetic Operators
 - | — Relational Operators
 - | — Logical Operators
 - | — Membership & Identity Operators
 - | — Operator Precedence
- 5. Control Flow
 - | — Conditional Statements (if, if-else, if-elif-else)
 - | — Loops
 - | | — for loop
 - | | — while loop
 - | | — Nested loops
 - | — break, continue, pass
 - | — Iterators & Generators (basic intro)
- 6. Functions & Modular Programming
 - | — Defining Functions
 - | — Arguments (positional, keyword, default, variable length)
 - | — Return values
 - | — Lambda Functions
 - | — Recursion
 - | — Modules (import, user-defined modules)
 - | — Command-line Arguments
- 7. Strings (Advanced)

- | └─ String Methods
- | └─ String Formatting
- | └─ Regular Expressions (re module)
- └─ 8. File Handling
 - | └─ Opening/Closing Files
 - | └─ Read/Write/Append
 - | └─ File Modes
 - | └─ Exception Handling with Files
- └─ 9. Object-Oriented Programming (OOP)
 - | └─ Classes & Objects
 - | └─ __init__ Constructor
 - | └─ Methods (instance, static, class methods)
 - | └─ Inheritance (single, multiple, multilevel)
 - | └─ Polymorphism & Overriding
 - | └─ Encapsulation & Abstraction
- └─ 10. Exception Handling
 - | └─ try, except
 - | └─ else, finally
 - | └─ Custom Exceptions
 - | └─ Raising Exceptions
- └─ 11. Functional Programming Tools
 - | └─ map(), filter(), reduce()
 - | └─ Decorators (basic & advanced)
 - | └─ Generators & yield
- └─ 12. Libraries & Packages (Essentials)
 - | └─ math, random, datetime
 - | └─ OS and sys
 - | └─ collections
 - | └─ pip & virtual environments
- └─ 13. Python with Data
 - | └─ NumPy (arrays, operations)
 - | └─ Pandas (Series, DataFrames, CSV handling)
 - | └─ Matplotlib (basic plotting)
 - | └─ Introduction to Seaborn
- └─ 14. Advanced Concepts (Optional / Bridge to ML)
 - | └─ Exception handling in real projects
 - | └─ Introduction to Automation (os, shutil, subprocess)
 - | └─ Multithreading & Thread Synchronization
 - | └─ Multiprocessing & Process Pooling
 - | └─ Multicore Programming Concepts
 - | └─ Advanced Decorators (function wrappers)
 - | └─ Intro to ML Libraries (scikit-learn, TensorFlow)

Progression of Our Automations using Python

Python Automations

- 1. Introduction to Automation
 - | — What is Automation?
 - | — Benefits & Use Cases (files, web, emails, reports)
 - | — Required Libraries & Setup (pip, venv, .env for secrets)
- 2. System & File Automations
 - | — Automating File Ops (os, shutil, pathlib)
 - | — Bulk rename/move/copy/delete
 - | — Directory sync & backup scripts (hash compare, timestamps)
 - | — Error handling & retries (try/except, backoff)
- 3. Scheduling & Orchestration
 - | — Periodic Tasks (schedule)
 - | — OS Schedulers
 - | — Long-running daemons & services (systemd, NSSM)
 - | — Dependency chains
- 4. Text & Document Automations
 - | — Bulk search/replace
 - | — CSV/Excel processing
 - | — Automated report generation
- 5. Email Automations
 - | — Send emails (smtplib, yagmail) with HTML & attachments
 - | — Notification systems (success/failure alerts, daily summaries)
- 6. OS & Process Automations
 - | — Run system commands (subprocess.run, capture output)
 - | — System monitoring (CPU, memory, disk, net) with psutil
 - | — Process info: list processes, name, PID, CPU%, RSS/Memory (psutil)
 - | — Start/stop/restart processes safely
 - | — Log file monitoring & tailing
- 7. Concurrency for Automation
 - | — Multithreading (I/O-bound tasks: downloads, API calls)
 - | — Multiprocessing (CPU-bound: parsing, compression, transforms)
 - | — Process pools & queues (concurrent.futures, multiprocessing.Pool)
- 8. Logging, Auditing & Maintenance
 - | — logging basics (levels, formatters, handlers)

Progression of our Data Science Journey

Artificial Intelligence (AI)

| — 1. Data Science

| | — Data Collection

| | | — Databases (SQL, NoSQL)

| | | — APIs, Web Scraping

| | | — Streaming Data (Kafka, IoT devices)

| | | — Open Datasets (Kaggle, UCI Repository)

| |

| | — Data Cleaning (ETL)

| | | — Handling Missing Values

| | | — Outlier Detection & Treatment

| | | — Data Transformation (scaling, encoding)

| | | — Feature Engineering

| |

| | — Exploratory Data Analysis (EDA)

| | | — Descriptive Statistics

| | | — Correlation Analysis

| | | — Hypothesis Testing

| | | — Data Summaries & Patterns

| |

| | — Data Visualization

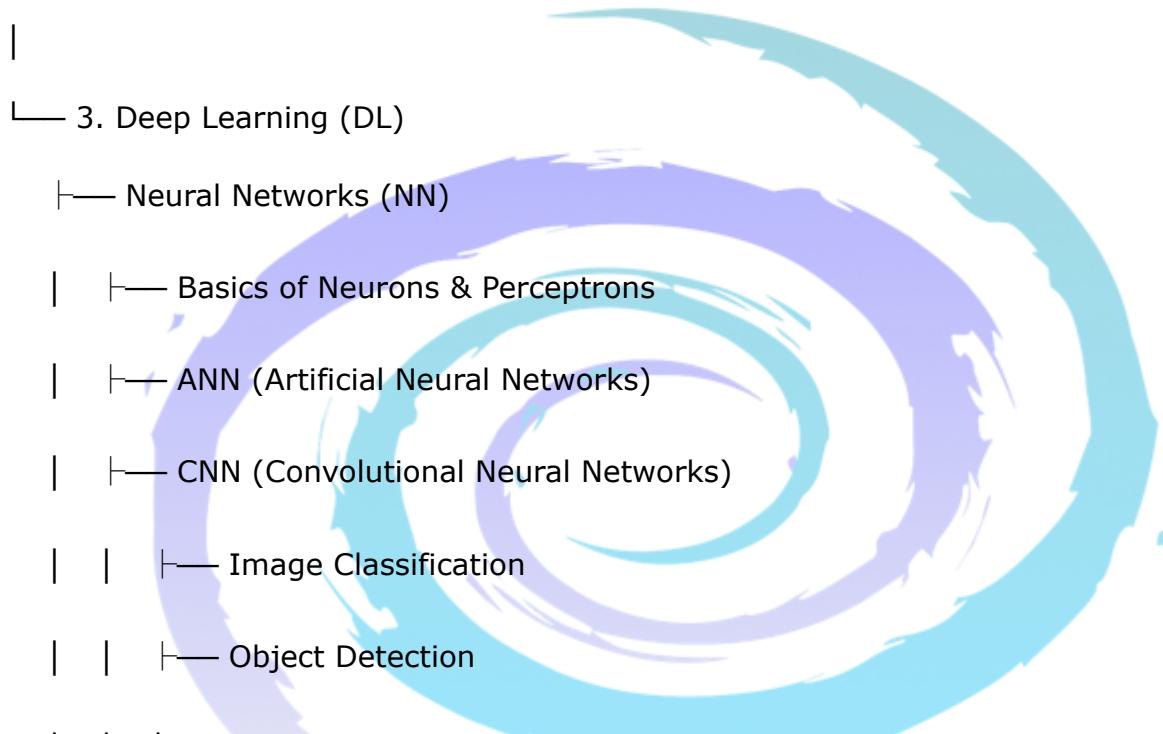
| | | — Charts (Bar, Pie, Line, Scatter)

| | | — Heatmaps & Histograms

- | | └— Dashboards (Tableau, Power BI)
- | | └— Python Libraries (Matplotlib, Seaborn, Plotly)
- | |
- | └— Applied AI + Statistics
- | └— Probability & Distributions
- | └— Decision Making with Data
- |
- | └— 2. Machine Learning (ML)

- | └— Supervised Learning
- | | └— Regression
 - | | | └— Linear Regression
 - | | | └— Logistic Regression
 - | | | └— Regularization
- | | └— Classification
 - | | | └— K-Nearest Neighbors (KNN)
 - | | | └— Support Vector Machines (SVM)
 - | | | └— Decision Trees
 - | | | └— Random Forests
- | |
- | | └— Ensemble Methods
 - | | | └— Bagging
 - | | | └— Boosting
- | |

- | └— Unsupervised Learning
 - | | └— Clustering
 - | | | └— K-Means
 - | | └— DBSCAN
 - | | └— Dimensionality Reduction
 - | | | └— PCA (Principal Component Analysis)



└─ Transformers (Attention-based Architecture)

| └─ Key Concepts

| | └─ Self-Attention

| | └─ Multi-Head Attention

| | └─ Positional Encoding

| └─ Encoder/Decoder Blocks

| └─ Model Types

| | └─ Encoder-only (BERT → Text Understanding)

| | └─ Decoder-only (GPT → Text Generation)

| └─ Encoder-Decoder (T5, BART → Summarization, Translation)

└─ LLMs (Large Language Models)

| └─ Pretraining

| | └─ Next Word Prediction

| | └─ Masked Language Modeling

| └─ Fine-tuning

| | └─ Domain Adaptation

| | └─ Instruction Tuning

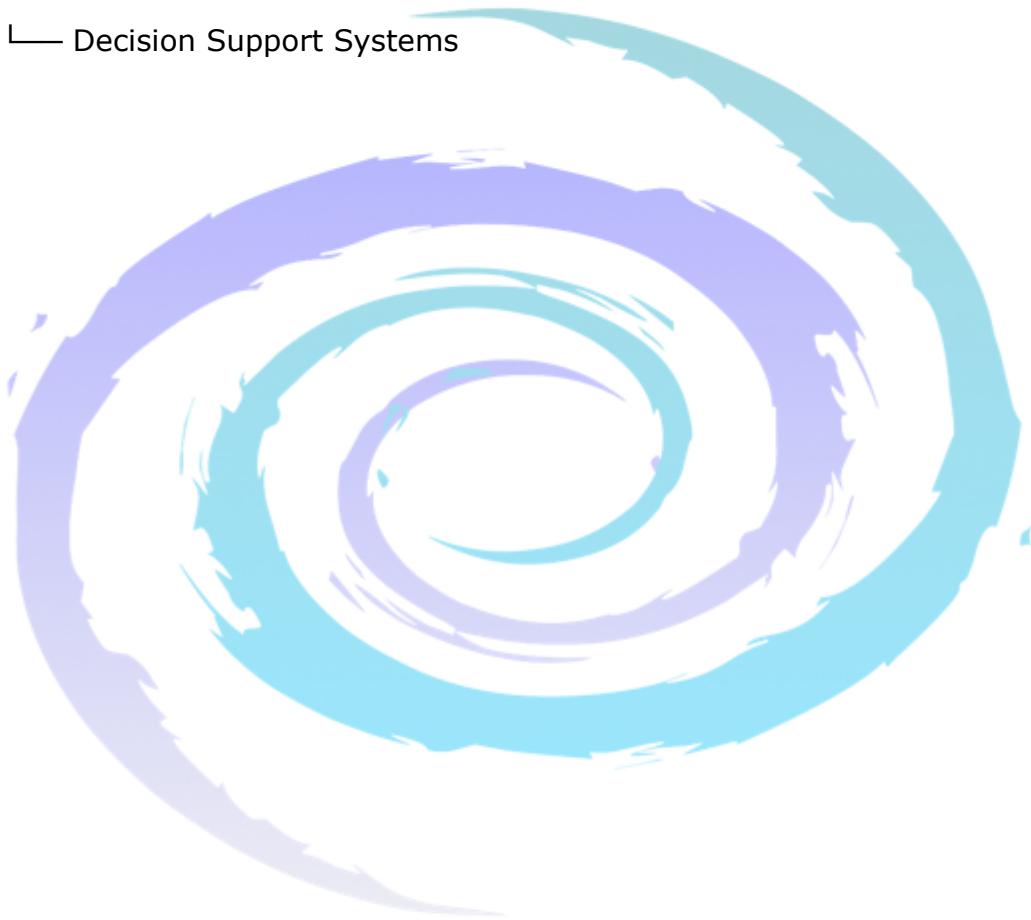
| └─ RLHF (Reinforcement Learning with Human Feedback)

| └─ Multimodal LLMs

| | └─ Text + Image (GPT-4V, Gemini)

| | └─ Text + Audio (Whisper + LLMs)

- | | └ Text + Video (emerging research)
- | └ Applications
 - | └ Chatbots (ChatGPT, Claude, Gemini)
 - | └ Code Generation (Copilot, Code Llama)
 - | └ Content Generation (blogs, emails, reports)
 - | └ Summarization / Translation
- | └ Decision Support Systems



Artificial Intelligence (AI)

- **Definition:** Making machines that can think, reason, and act like humans.
- **Relation:** The **umbrella field** under which everything else falls.

Data Science

- **Definition:** Extracting knowledge and insights from data using AI + statistics.
- **Relation:** Uses AI/ML tools + statistics for solving real-world problems.

Sub-steps:

- **Data Collection** → Gathering raw data (sensors, websites, databases).
- **Data Cleaning (ETL)** → Removing errors, filling missing values, transforming formats.
- **EDA (Exploratory Data Analysis)** → Finding trends and patterns in data.
- **Data Visualization** → Graphs, dashboards to make data easy to understand.
- **Applied AI + Statistics** → Using ML/AI methods for predictions/decisions.

Machine Learning (ML)

- **Definition:** A method in AI where machines **learn patterns from data** without being explicitly programmed.
- **Relation:** A **subset of AI**, forms the core of modern AI systems.

Types:

- **Supervised Learning** → Learning from labeled data.
 - **Regression** → Predict numbers (e.g., house price).
 - **Classification** → Predict categories (spam/not spam).
 - **Ensemble Methods** → Combine many models for better accuracy (Random Forests, XGBoost).
- **Unsupervised Learning** → No labels, find hidden patterns.
 - **Clustering** → Group similar items (customer segmentation).
 - **Dimensionality Reduction** → Compress data but keep info (PCA, t-SNE).
- **Reinforcement Learning (RL)** → Learn by trial and error, with rewards/punishments.
 - **Q-Learning, DQN, Policy Gradient** → Used in games, robotics.

- **Feature Engineering / Evaluation** → Selecting good inputs & testing models.

Deep Learning (DL)

- **Definition:** A subfield of ML that uses **neural networks with many layers**.
- **Relation:** Advanced ML → powerful for images, speech, text.

Neural Networks:

- **ANN (Artificial Neural Networks)** → Basic model (input → hidden → output).
- **CNN (Convolutional Neural Networks)** → Best for images, detect features like edges/faces.
- **RNN (Recurrent Neural Networks)** → Best for sequential data like text/speech.
 - **LSTM (Long Short-Term Memory)** → RNN with better memory.
 - **GRU (Gated Recurrent Unit)** → Lighter alternative to LSTM.
- **Autoencoders** → Compress data, detect anomalies.

Other Architectures:

- **GANs (Generative Adversarial Networks)** → Two models compete (Generator creates, Discriminator judges). Used in fake image generation (DeepFakes).
- **Transformers** → Use **self-attention**, process whole sequences in parallel.
 - **Encoder-only (BERT)** → Understanding text.
 - **Decoder-only (GPT)** → Generating text.
 - **Encoder–Decoder (T5, BART)** → Translation, summarization.

LLMs (Large Language Models)

- **Definition:** Very large Transformer models trained on massive text data.
- **Relation: Built using Transformers**, scaled up with billions of parameters.

Steps:

- **Pretraining** → Learn general language patterns (predict missing/next word).
- **Fine-tuning** → Specialize for tasks (medical chatbot, coding assistant).
- **RLHF (Reinforcement Learning with Human Feedback)** → Align with human expectations (safe, polite).

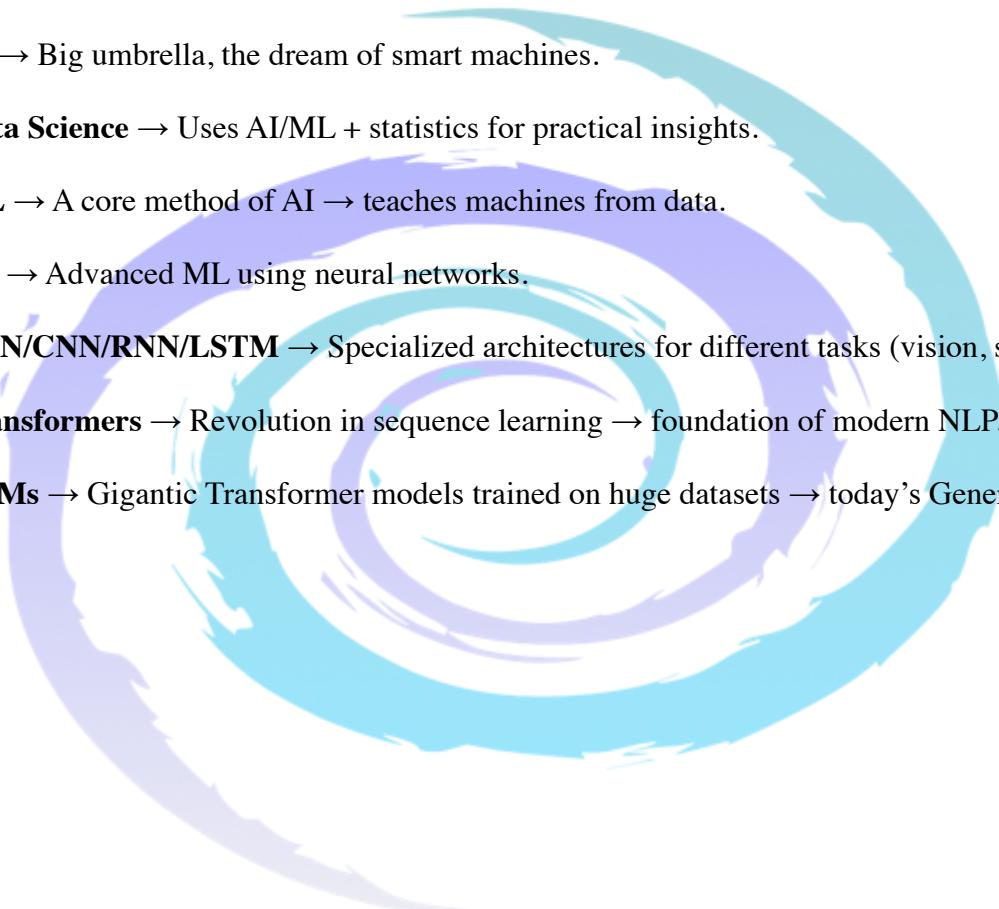
- **Multimodal LLMs** → Handle text + image + audio (e.g., GPT-4, Gemini).

Applications:

- **Chatbots** → ChatGPT, Claude.
- **Code Generation** → Copilot, Code Llama.
- **Summarization/Translation** → Summarize articles, translate languages.

Relations in One Flow

1. **AI** → Big umbrella, the dream of smart machines.
2. **Data Science** → Uses AI/ML + statistics for practical insights.
3. **ML** → A core method of AI → teaches machines from data.
4. **DL** → Advanced ML using neural networks.
5. **ANN/CNN/RNN/LSTM** → Specialized architectures for different tasks (vision, sequences).
6. **Transformers** → Revolution in sequence learning → foundation of modern NLP.
7. **LLMs** → Gigantic Transformer models trained on huge datasets → today's Generative AI.



RNN Character Prediction Application

```

import numpy as np

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.utils import to_categorical

def MarvellousCharcterPredictions():

  # 1) Prepare toy text dataset

  text = "hellohellohellohello"  # repeating pattern
  chars = sorted(list(set(text)))  # unique characters
  char_to_int = {c:i for i,c in enumerate(chars)}
  int_to_char = {i:c for i,c in enumerate(chars)}

  print("Unique characters:", chars)

  # 2) Convert text to integer sequence

  encoded = [char_to_int[c] for c in text]

  # 3) Create input-output pairs (sequence length = 3)

  seq_length = 3
  X, y = [], []
  for i in range(len(encoded) - seq_length):
    X.append(encoded[i:i+seq_length])  # e.g. "hel"
    y.append(encoded[i+seq_length])    # next char e.g. "l"

  X = np.array(X)
  y = np.array(y)

  # One-hot encode output

```

```

y = to_categorical(y, num_classes=len(chars))

# Reshape X for RNN (samples, timesteps, features)
X = X.reshape((X.shape[0], X.shape[1], 1))

# 4) Build RNN model
model = Sequential()
model.add(SimpleRNN(16, activation="tanh", input_shape=(seq_length, 1)))
model.add(Dense(len(chars), activation="softmax"))

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

# 5) Train
model.fit(X, y, epochs=300, verbose=0)

# 6) Test prediction
test_input = ["h", "e", "l"] # we expect model to predict "l"
encoded_input = np.array([[char_to_int[c] for c in test_input]]).reshape((1, seq_length, 1))

pred = model.predict(encoded_input, verbose=0)
predicted_char = int_to_char[np.argmax(pred)]

print("Input:", test_input, "-> Predicted next char:", predicted_char)

def main():
    MarvellousCharcterPredictions()

if __name__ == "__main__":
    main()
  
```

Explanation of above application

- **Task:** Next-character prediction.
- **Granularity: Character** level.
- **Model:** A simple recurrent neural network (**SIMPLERNN**) with one hidden layer (16 units) + a softmax output over the unique characters in the text.

Data preparation

Raw text and vocabulary

```
text = "hellohellohellohello"
chars = sorted(list(set(text))) # unique characters
char_to_int = {c:i for i,c in enumerate(chars)}
int_to_char = {i:c for i,c in enumerate(chars)}
```

- **text length = 20** characters ("hello" × 4).
- Unique characters in "hello" = {h, e, l, o} → after **sorted(...)**: ['e', 'h', 'l', 'o'].
- Example mapping (because of alphabetical sort):
 - e→0, h→1, l→2, o→3
 (Important: mapping depends on the **sorted order**, not the order of first appearance.)

Integer encoding the text

```
encoded = [char_to_int[c] for c in text]
```

- Converts each char to its integer id.
- This is **not** one-hot; it's just integer indices (inputs will be reshaped to numeric features of size 1; Keras casts to float internally).

Sliding window to make (X, y) pairs

```
seq_length = 3
X, y = [], []
for i in range(len(encoded) - seq_length):
    X.append(encoded[i:i+seq_length]) # 3 chars as input
    y.append(encoded[i+seq_length]) # 4th char as label
• With length 20 and seq_length=3, you get 17 training samples (from positions 0..16).
• A few concrete examples from the actual text:
  ◦ indices 0–2: h e l → next is l
```

- indices 1–3: **e l l** → next is **o**
- indices 2–4: **l l o** → next is **h**
- indices 3–5: **l o h** → next is **e**
- ...and so on (a repeating cycle from the “hello” pattern).

One-hot encode the targets

```
y = to_categorical(y, num_classes=len(chars))
```

- **len(chars)=4**, so **y** becomes shape **(17, 4)**.
- Reason: we are doing **multi-class classification** (which of the 4 characters comes next?), so the label must be a one-hot vector.

Reshape inputs for RNN

```
X = np.array(X) # (17, 3)
X = X.reshape((X.shape[0], X.shape[1], 1)) # (samples, timesteps,
features) → (17, 3, 1)
```

- **Timesteps = 3** (we feed 3 characters in order).
- **Features = 1** (each timestep is a single numeric feature: the integer id).
Note: In larger setups, we would typically use an **Embedding** or **one-hot inputs** instead of raw integers to avoid implying any “ordinal” meaning to ids.

Model architecture

```
model = Sequential()
model.add(SimpleRNN(16, activation="tanh", input_shape=(seq_length,
1)))
model.add(Dense(len(chars), activation="softmax"))
SimpleRNN(16, activation="tanh")
```

- **Recurrent layer** with 16 hidden units.
- Processes the sequence (3 timesteps) and returns the **last hidden state** (a vector of length 16).
- **Why tanh?** Values range in [-1, 1], which helps stabilize recurrent dynamics. (ReLU is less common for vanilla RNNs due to exploding activations; LSTM/GRU are more robust alternatives.)

Parameter count (good to teach!):

- For SimpleRNN with **U** units and input dim **D**:
 - Weights = **U * (U + D + 1)** (kernel DxU, recurrent UxU, bias U)

- Here $\mathbf{U=16, D=1} \rightarrow 16 * (16+1+1) = 16 * 18 = 288$ parameters.

Dense(4, activation="softmax")

- Maps the 16-dim hidden state to 4 logits (one per character).
- **Softmax** converts logits to a probability distribution over the 4 characters.

Parameter count:

- Dense weights: $16 * 4 = 64$
- Bias: 4
- Total Dense params = **68**

Grand total = 288 (RNN) + 68 (Dense) = **356** parameters.

Loss, optimizer, metrics

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

- **Loss:** `categorical_crossentropy` is the standard for one-hot multi-class classification. If \mathbf{p} is the predicted probability vector and \mathbf{y} is one-hot, the loss simplifies to $-\log(p_{\text{trueClass}})$.
- **Optimizer:** `adam` is a robust default that adapts learning rates per parameter.
- **Metric:** `accuracy` counts correct argmax predictions.

Training

```
model.fit(X, y, epochs=300, verbose=0)
```

- Trains on the 17 examples for **300 epochs**.
- With such a tiny dataset, the model will **overfit** (which is totally fine for a didactic demo—our goal is to make it memorize the pattern).
- No validation split is used here (again fine for a toy example).

Inference (predicting the next character)

```
test_input = [ "h", "e", "l" ]           # we expect "l" next in the
pattern
encoded_input = np.array([[char_to_int[c] for c in
test_input]]).reshape((1, seq_length, 1))
```

```
pred = model.predict(encoded_input, verbose=0) # shape (1, 4)
```

```
predicted_char = int_to_char[np.argmax(pred)]
```

- `encoded_input` shape is **(1, 3, 1)** → one sample, 3 timesteps, 1 feature each.

- `pred` is a **(1, 4)** probability vector.

- `np.argmax(pred)` gives the predicted class id; map back via `int_to_char`.

- For “**h e l**”, the next character in the repeating pattern is indeed “**l**”.



RNN, LSTM vs Transformers

Basic Concept of RNN (Recurrent Neural Network)

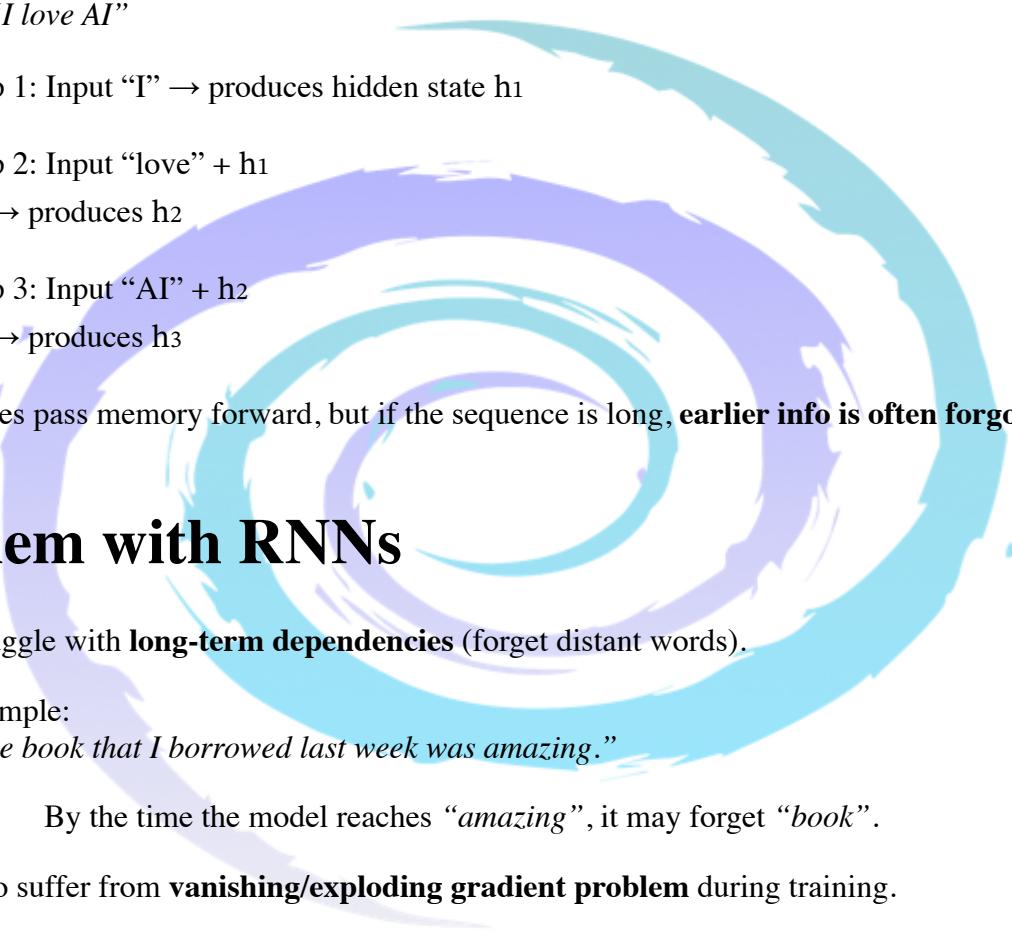
- A neural network designed to process **sequential data** (text, speech, time series).
 - Takes input **one step at a time**.
 - Maintains a **hidden state (memory)** that carries info from previous steps.

Example:

Sentence: “*I love AI*”

- Step 1: Input “I” → produces hidden state h_1
- Step 2: Input “love” + h_1
 h_1 → produces h_2
- Step 3: Input “AI” + h_2
 h_2 → produces h_3

Hidden states pass memory forward, but if the sequence is long, **earlier info is often forgotten**.



Problem with RNNs

- Struggle with **long-term dependencies** (forget distant words).
- Example:
“The book that I borrowed last week was amazing.”
 - By the time the model reaches “*amazing*”, it may forget “*book*”.
- Also suffer from **vanishing/exploding gradient problem** during training.

Basic Concept of LSTM (Long Short-Term Memory)

- An advanced RNN architecture (1997).
- Introduced **gates** to control memory flow:
 - **Forget Gate:** Decides what to throw away.
 - **Input Gate:** Decides what new info to store.
 - **Output Gate:** Decides what to send as output.

This helps LSTMs remember **longer sequences** compared to vanilla RNNs.

Example:

Sentence: “*The concert I attended in London last summer was unforgettable.*”

- LSTM can connect “concert” ↔ “unforgettable” better than RNN, because it manages memory with gates.

Problems with RNNs (and LSTMs/GRUs)

RNNs process sequences **one word at a time** → like reading slowly word by word.

This caused several issues:

1. Sequential Processing = Slow

- RNNs can't process words in parallel.
- Example: Sentence “*I love Transformers*”
 - RNN reads → “I” → then “love” → then “Transformers”.
- For long texts (books, code, big datasets), this is **very slow**.

Transformers process **all words at once** → **parallelism** = faster training.

2. Long-Term Dependencies Problem

- RNNs often **forget context** when sentences get long.
- Example:
“The book that I borrowed from the library last week because it had such good reviews was amazing.”
 - RNN may lose track of “book” by the time it reaches “amazing.”

Transformers use **self-attention** → any word can directly connect to any other word, no matter how far.

3. Vanishing/Exploding Gradients

- When training RNNs on long sequences, gradients (learning signals) either shrink (vanish) or blow up (explode).
- This makes training unstable.
- LSTMs/GRUs helped, but only partially.

Transformers fix this with **attention + residual connections**, no gradient vanishing issues.

4. Contextual Understanding

- RNN reads left → right (or bidirectional in Bi-LSTM).
- Still limited in capturing **global context**.
- Example: Word “*bank*”:
 - Sentence: “He sat by the **bank** of the river.”
 - RNN struggles unless trained deeply.
 - Transformer attention → directly links “bank” to “river”.

5. Scalability

- RNNs don't scale well to very large data/models.
- Transformers scale beautifully → that's why we have **GPT-3 (175B params)**, **GPT-4**, **BERT**, etc.

Advantages of Transformers

Parallel processing (train much faster on GPUs).

Global context via self-attention (long-distance relationships).

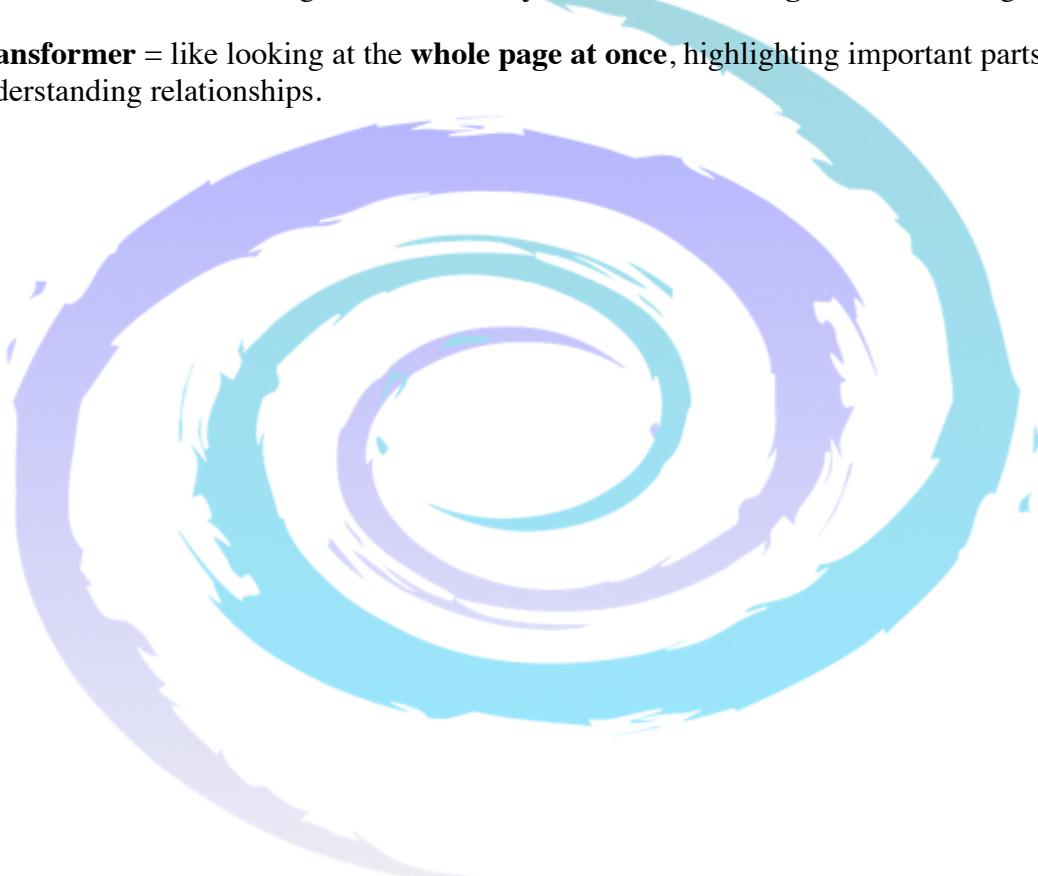
Scalable (works for billions of parameters).

Versatile (works for text, images, audio, even DNA).

State-of-the-art performance → replaced RNNs in NLP.

Simple Analogy

- **RNN/LSTM** = like reading a novel **word by word with one finger**, remembering as best as you can.
- **Transformer** = like looking at the **whole page at once**, highlighting important parts, and instantly understanding relationships.



Breast Cancer Classification using Random Forest & Pipeline

This project demonstrates an **end-to-end machine learning pipeline** for predicting **breast cancer diagnosis** (Benign vs Malignant) using the **Breast Cancer Wisconsin dataset**.

It follows **industrial best practices** by:

- Automating preprocessing with **Pipelines**
- Handling missing values using **SimpleImputer**
- Using **Random Forest Classifier** as a robust baseline
- Saving & loading trained models using **Joblib**
- Providing **data visualization** for interpretability

Dependencies

Install the required Python packages before running the project:

```
pip install pandas numpy matplotlib scikit-learn joblib
```

Dataset Information

Source: UCI Machine Learning Repository - Breast Cancer Wisconsin Dataset

Features (10):

1. ClumpThickness
2. UniformityCellSize
3. UniformityCellShape
4. MarginalAdhesion
5. SingleEpithelialCellSize
6. BareNuclei
7. BlandChromatin
8. NormalNucleoli
9. Mitoses

Target:

- **2 = Benign**
- **4 = Malignant**

Workflow

Data Preparation

- Convert raw `.data` file to `.csv` with **headers**
- Replace **missing values (?)** with `NaN`
- Ensure numeric type conversion for all feature columns

Train-Test Split

- Split into **70% train** and **30% test**
- Use **stratified sampling** to preserve class balance

Pipeline Construction

- **Step 1:** `SimpleImputer(strategy="median")` for missing value handling
- **Step 2:** `RandomForestClassifier` with 300 estimators for classification

Model Training & Evaluation

- **Metrics:** Accuracy, Confusion Matrix, Classification Report
- **Feature Importance Plot:** Shows most influential features

Model Saving & Loading

- Save model with **Joblib**
- Load model for future predictions without retraining

Running the Project

Prepare CSV (Only Once)

```
from breast_cancer_pipeline import data_file_to_csv
data_file_to_csv()
```

Train & Evaluate Model

```
python breast_cancer_pipeline.py
```

Expected output:

```
Train Accuracy :: 1.0
Test Accuracy :: 0.97
Classification Report:
      precision    recall   f1-score   support
...
Confusion Matrix:
[[...]]
Model saved to bc_rf_pipeline.joblib
Loaded model prediction for first test sample: 2
```

Visualizations

- Feature Importance (Random Forest)
- Confusion Matrix with Matplotlib

Model Storage

- Model is saved as **bc_rf_pipeline.joblib**
- Can be loaded anytime for prediction without retraining:

```
from breast_cancer_pipeline import load_model
model = load_model("bc_rf_pipeline.joblib")
```

Sample Prediction

```
sample = test_x.iloc[[0]]
pred = model.predict(sample)
print("Prediction:", pred[0]) # 2 (Benign) or 4 (Malignant)
```

Author

Piyush Manohar Khairnar

 Date: 09/08/2025

Recurrent Neural Networks (RNN)

- **Recurrent Neural Networks (RNNs)** are a class of neural networks designed for **sequential data** (text, speech, time-series, video frames).
- Unlike **Feedforward Neural Networks (FNNs)** and **Convolutional Neural Networks (CNNs)**, RNNs have a **memory mechanism** that helps them remember **previous inputs**.
- They are widely used in **Natural Language Processing (NLP)**, **speech recognition**, **music generation**, and **time-series forecasting**.

Sequence Data

- Data that has **order** and **context matters**.
- Example: A sentence → “*I love AI*” → the word “love” depends on “I”.

Hidden State

- At each time step, the RNN maintains a **hidden state (h_t)**.
- This hidden state captures **information from previous inputs**.
- Formula:

$$h_t = f(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

where:

- h_t = hidden state at time t
- x_t = input at time t
- W_h, W_x = weight matrices
- f = activation function (usually tanh/ReLU)

Timestep

- One **unit of time** in sequence processing.
- Example: For text input, each word is a timestep.

Input Feature

- The dimension of each input.
- Example: Word embeddings (vector representation of words) → each word is represented as a vector of numbers.

Output

- RNN can produce:
 - **Many-to-One:** Sentiment analysis (entire sentence → one label).
 - **Many-to-Many:** Language translation (sequence → sequence).

Variants of RNN

- **Vanilla RNN** → basic RNN, suffers from vanishing gradient problem.
- **LSTM (Long Short-Term Memory)** → solves long-term dependency issue with gates (input, forget, output).
- **GRU (Gated Recurrent Unit)** → simpler than LSTM but effective.

Uses in Development

- **NLP:** Sentiment analysis, machine translation, text generation.
- **Speech Recognition:** Convert audio sequence to text.
- **Time-Series Forecasting:** Predict stock prices, weather data.
- **Music Generation:** Learn sequences of notes.

Simple Python Code Example

Here's a basic RNN example with Keras to demonstrate sequential input:

```

import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Embedding

# Sample: Sequence classification
# Suppose we have 10 sequences, each with 5 timesteps, vocabulary
size = 20
x = np.random.randint(20, size=(10, 5))    # Input sequences (10
samples, 5 timesteps)
y = np.random.randint(2, size=(10, 1))      # Binary output (labels)

# Build RNN model
model = Sequential()
model.add(Embedding(input_dim=20, output_dim=8, input_length=5))  # Word embedding
model.add(SimpleRNN(16, activation='tanh'))                      # RNN layer
model.add(Dense(1, activation='sigmoid'))                         # Output layer

# Compile model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train
model.fit(x, y, epochs=5, verbose=1)

# Prediction
print("Sample Prediction:", model.predict(x[:1]))

```

Explanation of Code

1. **Embedding Layer:** Converts integer-encoded words → dense vectors.
2. **SimpleRNN Layer:** Processes sequences one timestep at a time, remembers past states.
3. **Dense Layer:** Outputs final classification.
4. **Prediction:** Outputs probability (0–1) for binary class.

Sequence Data :

- **Sequence Data** = Data where **order matters**.
- Each element depends on the **previous element(s)**.
- Examples:
 -  Text: A sentence → meaning depends on word order.
 - “*I love AI*” ≠ “*AI love I*”
 -  Audio: A speech signal → each sound depends on previous sounds.
 -  Time-series: Stock price today depends on yesterday’s price.

In **normal feedforward neural networks**, inputs are **independent**. But in **sequence tasks**, we need **context**.

How RNN Handles Sequence Data

Step-by-Step Process

1. Input sequence is fed **one element at a time** (word by word, timestep by timestep).
2. At each timestep t , RNN takes:
 - Current input (x_t)
 - Previous hidden state (h_{t-1}) (memory from past)
 - Combines them to compute **new hidden state** (h_t)
3. Hidden state (h_t) = acts like **short-term memory**.
4. At the end, RNN can output:
 - Final result (e.g., sentiment of whole sentence)
 - Or sequence of outputs (e.g., translation word-by-word).

Memory Mechanism in RNN

- **Hidden State** = Memory cell that carries **past information**.
- Formula:

- $$h_t = f(W_x x_t + W_h h_{t-1} + b)$$

- where:

- x_t = current input
- h_{t-1} = previous memory
- h_t = new memory
- f = activation function (tanh, ReLU)

Example

Sentence: “**I love AI**”

- At timestep 1: input = “I”, memory starts forming.
- At timestep 2: input = “love”, RNN remembers “I” + “love”.
- At timestep 3: input = “AI”, RNN remembers “I love AI”.

Thus, RNN builds meaning step-by-step.

Simple Visualization

```

x1 → [RNN Cell] → h1
x2 → [RNN Cell] → h2
x3 → [RNN Cell] → h3 → output
  
```

- Each **RNN cell** passes **memory (h)** forward.
- Like a **chain** of connected cells.

Small Python Example to Demonstrate Memory

Here's a **toy RNN implementation** to show how it processes a sequence:

```
import numpy as np

# Example sentence: "I love AI" (encoded as numbers)
sequence = [1, 2, 3]    # Let's say 1=I, 2=love, 3=AI

# Initialize weights and hidden state
Wx, Wh, b = 0.5, 0.8, 0.1    # random chosen values
h = 0    # initial hidden state

print("Processing sequence step by step:")

for t, x in enumerate(sequence):
    h = np.tanh(Wx * x + Wh * h + b)    # memory update
    print(f"Timestep {t+1} | Input={x} | Hidden State={h:.4f}")
```

Output Explanation

- At **timestep 1**, RNN only sees “I” → stores memory.
- At **timestep 2**, it sees “love” + remembers “I”.
- At **timestep 3**, it sees “AI” + remembers “I love”.

This demonstrates **sequence + memory flow**.

Why This Matters in Development

- RNNs are powerful because they **don't forget the past immediately**.
- Example applications:
 - **Chatbot** → remembers previous words in conversation.
 - **Stock predictor** → remembers yesterday's price.
 - **Music generator** → remembers last few notes.

Advantages & Limitations

Advantages:

- Good for sequential data.
- Captures context (history).

Limitations:

- **Vanishing Gradient** → hard to remember long-term dependencies.
- Slow for very long sequences.
- Replaced in modern NLP by **Transformers** (e.g., **BERT**, **GPT**, **ChatGPT**).

Summary

- RNNs are neural networks for **sequential tasks**.
- They maintain **memory** using hidden states.
- Widely used in **NLP, speech, and forecasting**.
- Advanced versions: **LSTM & GRU**.
- Modern replacements: **Transformers**.

Recursion in Python

Iteration :

Iteration is when we execute a block of code multiple times using loops like:

- `for` loop
- `while` loop

Example: Print numbers from 1 to 5 using iteration

```
for i in range(1, 6):
    print(i)
```

Output:

```
1
2
3
4
5
```

Recursion :

Recursion is when a **function calls itself** to solve smaller instances of the same problem.

Each recursive call should bring us closer to a **base case** (stopping condition).

General Syntax of Recursion:

```
def recursive_function(parameters):
    if base_condition:
        return some_value
    else:
        return recursive_function(modified_parameters)
```

Important Components of Recursion

1. **Base Case** – stopping condition
2. **Recursive Case** – function calling itself

Without a **base case**, recursion causes **infinite calls** and eventually a **stack overflow**.

Example 1: Factorial of a Number

Problem:

Factorial of $n = n \times (n-1) \times (n-2) \times \dots \times 1$
 e.g., `factorial(4) = 4 * 3 * 2 * 1 = 24`

Recursive Code:

```
Fact = 1
```

```
def Factorial(no):
```

```
    global Fact
```

```
    if(no >= 1):
```

```
        Fact = Fact * no
```

```
        no = no - 1
```

```
        Factorial(no)
```

```
    return Fact
```

Recursion vs Iteration – Code Comparison

Problem: Print 1 to N

Using Iteration:

```
def print_numbers_iter(n):
    for i in range(1, n+1):
        print(i)
```

Using Recursion:

```
def print_numbers_rec(n):
    if n == 0:
        return
    print_numbers_rec(n - 1)
    print(n)
```

Iteration uses a loop.

Recursion uses **call stack**.

Recursion Limit

Python has a limit to how deep recursion can go.

Check limit:

```
import sys
print(sys.getrecursionlimit()) # usually 1000
```

To increase:

```
sys.setrecursionlimit(2000) # use with caution
```

Why and When to Use Recursion

Use when:

- Problem can be broken into similar subproblems
- Structure resembles tree or hierarchy
- Backtracking or divide & conquer is involved

Avoid when:

- Stack depth is large
- Iteration is simpler and more efficient

Consider below application which demonstrate concept of Recursion

```
print("---- Marvellous Infosystems by Piyush Khairnar----")
print("Demonstration of Scope of Recursion")
import sys
print(" Maximum number of recursive call are {} in python".format(sys.getrecursionlimit()))
# Changing recursion limit
sys.setrecursionlimit(1500)
print(" Maximum number of recursive call are {} in python".format(sys.getrecursionlimit()))
# Recursive function which goes into infinite recursive calls
def fun():
    print("Inside fun")
    fun()
i = 1
# Recursive function which performs recursive calls 10 times
def gun():
    global i
    if(i<=10):
        print(i)
        i+=1
        gun()
gun()
def fact(no):
    if(no == 0):
        return 1
```

```
return no * fact(no-1)

value = 5
ret = fact(value)
print("Factorial of {} is {}".format(value,ret))
```

Output of Above application

```
MacBook-Pro-de-MARVELLOUS: Today marvellous$ python Recursion.py
---- Marvellous Infosystems by Piyush Khairnar-----
Demonstration of Scope of Recursion
Maximum number of recursive call are 1000 in python
Maximum number of recursive call are 1500 in python
1
2
3
4
5
6
7
8
9
10
Factorial of 5 is 120
MacBook-Pro-de-MARVELLOUS: Today marvellous$ █
```



Relationship Between

Deep Learning, RNN, NLP,

Transformers, LLM & GenAI

1. Deep Learning – The Foundation

- **Definition:** Subfield of Machine Learning using **neural networks with multiple layers**.
- **Goal:** Learn **complex patterns** from large amounts of data (images, text, audio).
- **Architectures in Deep Learning:**
 - **FNN (Feedforward NN)** → Basic networks.
 - **CNN (Convolutional NN)** → Best for images.
 - **RNN (Recurrent NN)** → Best for sequences (text, speech, time-series).
 - **Transformers** → Best for long sequences, modern NLP.

Deep Learning = Parent field. Everything else (RNN, Transformers, LLM, GenAI) grows from here.

2. NLP – Application Area of Deep Learning

- **Definition:** Natural Language Processing = Teaching machines to **understand and generate human language**.
- **Why Deep Learning?**
 - Classical NLP (rule-based, statistical) struggled with complexity.
 - Deep Learning allowed **automatic feature extraction** + handling of complex language structures.

NLP = **Problem Domain** (language).

Deep Learning = **Tools** (RNN, Transformers) used to solve NLP problems.

3. RNN – First Breakthrough for NLP

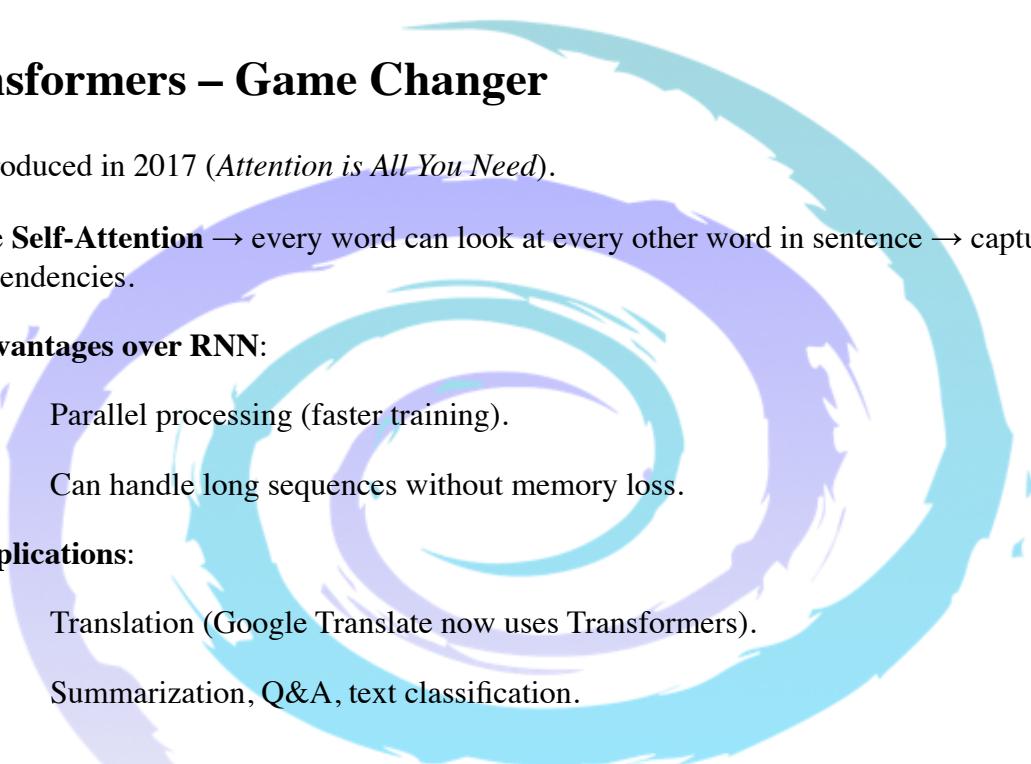
- **RNN (Recurrent Neural Networks):**
 - Designed to handle **sequence data** (sentences, speech).
 - Maintains **memory (hidden state)** → understands context of words.

- Variants: **LSTM, GRU** → better long-term memory.
- **Applications in NLP:**
 - Sentiment analysis
 - Machine translation (before Transformers)
 - Speech recognition

Mapping:

NLP tasks → handled by RNN (first DL success in language).
 But: RNNs struggled with **long texts** (vanishing gradient).

4. Transformers – Game Changer

- 
- Introduced in 2017 (*Attention is All You Need*).
 - Use **Self-Attention** → every word can look at every other word in sentence → captures long-range dependencies.
 - **Advantages over RNN:**
 - Parallel processing (faster training).
 - Can handle long sequences without memory loss.
 - **Applications:**
 - Translation (Google Translate now uses Transformers).
 - Summarization, Q&A, text classification.

Mapping:

Transformers replaced RNNs in NLP → became the **new standard**.

5. LLM – Large Language Models

- **Definition:** A Transformer model trained on **massive datasets** (billions/trillions of words).
- **Examples:** GPT (OpenAI), BERT, PaLM, LLaMA.
- **Characteristics:**
 - Learns grammar, facts, reasoning.
 - Can generate **coherent, human-like text**.
- **Why “Large”?**
 - Billions of parameters → store knowledge about world.

Mapping:

LLMs = **Large-scale implementation of Transformers** applied to NLP.

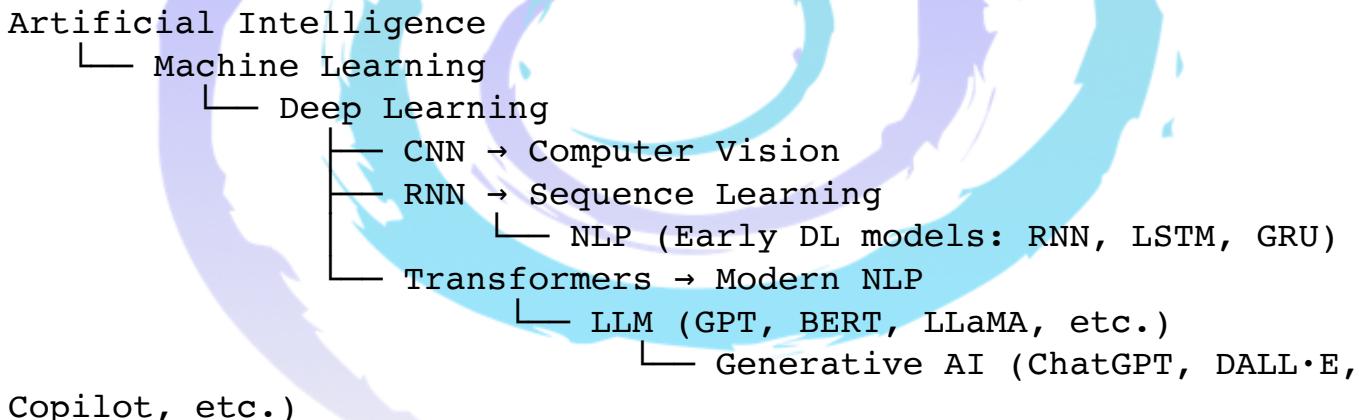
6. Generative AI (GenAI)

- **Definition:** AI that can **generate new content** (text, images, music, code).
- **LLMs in GenAI:**
 - ChatGPT (text generation)
 - Code assistants (Copilot)
 - Multi-modal models (text → image, text → video)
- **Beyond text:**
 - Diffusion Models → Images (DALL·E, Stable Diffusion)
 - Audio generation (MusicLM)

Mapping:

GenAI = Application layer that uses **LLMs (text)**, **Diffusion models (images)**, **GANs/VAEs (music/video)**.

Full Hierarchical Flow



8. Simple Analogy that we discussed in session

- **Deep Learning** = Toolbox.
- **RNN** = Old hammer for sequences (good, but limited).
- **Transformers** = New electric drill (powerful & scalable).
- **LLM** = A mega machine built using Transformers.
- **GenAI** = Products we create using that mega machine (ChatGPT, DALL·E).

Summary Mapping

Concept	Role	Example
Deep Learning	Parent field, provides architectures	ANN, CNN, RNN, Transformers
NLP	Application domain (language)	Translation, Chatbots
RNN	Early DL architecture for NLP	Sentiment analysis, speech recognition
Transformers	Modern DL architecture for NLP	BERT, GPT
LLM	Large-scale Transformer trained on huge text	GPT-4, LLaMA
GenAI	AI that generates new content	ChatGPT, Copilot, DALL·E



Scheduling in Python

- Schedule is in-process scheduler for periodic jobs that use the builder pattern for configuration.
- Schedule lets you run Python functions (or any other callable) periodically at pre-determined intervals using a simple, human-friendly syntax.
- Schedule Library is used to schedule a task at a particular time every day or a particular day of a week.
- We can also set time in 24 hours format that when a task should run.
- Basically, Schedule Library matches your systems time to that of scheduled time set by you.
- Once the scheduled time and system time matches the job function (command function that is scheduled) is called.

Before using schedule library we have to install it as

Installation of Schedule module

pip install schedule

Classes and methods from schedule library

schedule.Scheduler class

schedule.every(interval=1)

Calls every on the default scheduler instance.

Schedule a new periodic job.

schedule.run_pending()

Calls run_pending on the default scheduler instance.

Run all jobs that are scheduled to run.

schedule.run_all(delay_seconds=0)

Calls run_all on the default scheduler instance.

Run all jobs regardless if they are scheduled to run or not.

schedule.idle_seconds()

Calls idle_seconds on the default scheduler instance.

schedule.next_run()

Calls next_run on the default scheduler instance.

Datetime when the next job should run.

schedule.cancel_job(job)

Calls cancel_job on the default scheduler instance.

Delete a scheduled job.

schedule.Job(interval, scheduler=None) class

A periodic job as used by Scheduler.

There are two parameters as

interval: A quantity of a certain time unit

scheduler: The Scheduler instance that this job will register itself with once it has been fully configured in Job.do().

Basic methods for Schedule.job

at(time_str)

Schedule the job every day at a specific time.

Calling this is only valid for jobs scheduled to run every N day(s).

Parameters: time_str – A string in XX:YY format.

Returns: The invoked job instance

do(job_func, *args, **kwargs)

Specifies the job_func that should be called every time the job runs.

Any additional arguments are passed on to job_func when the job runs.

Parameters: job_func – The function to be scheduled

Returns: The invoked job instance

run()

Run the job and immediately reschedule it.

Returns: The return value returned by the job_func

to/latest)

Schedule the job to run at an irregular (randomized) interval.

For example, every(A).to(B).seconds executes the job function every N seconds such that A <= N <= B.

Selection in Python

Selection :

Selection or Decision Making in programming refers to the ability of a program to execute **certain blocks of code based on specific conditions**.

Python provides the following selection structures:

1. `if` statement
2. `if-else` statement
3. `if-elif-else` ladder
4. Nested `if` statements
5. Use of logical operators in conditions

`if` Statement

Syntax:

```
if condition:
    # block of code (executed if condition is True)
```

Example:

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

Key Point:

- If the condition is `True`, the block executes.
- If the condition is `False`, nothing happens.

`if-else` Statement

Syntax:

```
if condition:
    # block A
else:
    # block B
```

Example:

```
num = 5
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

- Executes **either** the `if` block or the `else` block, never both.

if-elif-else Ladder

Use when you need to evaluate **multiple conditions**.

```
if condition1:
    # block A
elif condition2:
    # block B
elif condition3:
    # block C
else:
    # block D
```

Example:

```
score = 82
if score >= 90:
    print("Grade A")
elif score >= 75:
    print("Grade B")
elif score >= 60:
    print("Grade C")
else:
    print("Fail")
```

- Checks top-down; first `True` condition wins.
- Only **one block** is executed.

Nested **if** Statements

An **if** statement **inside another if** statement.

Example:

python

```
x = 20
if x > 10:
    if x < 30:
        print("x is between 10 and 30")
    else:
        print("x is greater than or equal to 30")
```

- Useful for **complex decision trees**
- Maintain proper **indentation**

Logical Operators

Used to combine multiple conditions inside **if**.

Operator	Description	Example
and	Both must be True	<code>x > 5 and x < 10</code>
or	At least one True	<code>x < 0 or x > 100</code>
not	Reverses condition	<code>not(x == 5)</code>

Sentiment Analysis with a SimpleRNN

```

import numpy as np

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

def MarvellousSentimentAnalysis():

# 1) Tiny dataset (Positive = 1, Negative = 0)

sentences = [
    "I love this movie",
    "This film was great",
    "What a fantastic experience",
    "I really enjoyed it",
    "Absolutely wonderful acting",
    "I hate this movie",
    "This film was terrible",
    "What a bad experience",
    "I really disliked it",
    "Absolutely horrible acting"
]

print("Input dataset : ")
print(sentences)

labels = [1,1,1,1,1, 0,0,0,0,0] # 1=Positive, 0=Negative

# 2) Tokenize text → convert words to integers

tokenizer = Tokenizer(num_words=50) # keep top 50 words
tokenizer.fit_on_texts(sentences)

X = tokenizer.texts_to_sequences(sentences)

```

```

print("Word Index:", tokenizer.word_index) # show vocabulary

# 3) Pad sequences → same length input
maxlen = 5
X = pad_sequences(X, maxlen=maxlen)
y = np.array(labels)

# 4) Build simple RNN model
model = Sequential()
model.add(Embedding(input_dim=50, output_dim=8, input_length=maxlen)) # word embeddings
model.add(SimpleRNN(8, activation="tanh"))
model.add(Dense(1, activation="sigmoid")) # binary output
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

# 5) Train
model.fit(X, y, epochs=30, verbose=0)

# 6) Test on new examples
test_sentences = ["I enjoyed this film", "I hated this film"]
test_seq = tokenizer.texts_to_sequences(test_sentences)
test_seq = pad_sequences(test_seq, maxlen=maxlen)
pred = model.predict(test_seq)
for s, p in zip(test_sentences, pred):
    print(f"Sentence: {s} -> Sentiment: { 'Positive' if p>0.5 else 'Negative' }")

def main():
    MarvellousSentimentAnalysis()

if __name__ == "__main__":
    main()
  
```

Explanation of above application

A tiny **binary sentiment classifier** (Positive = 1, Negative = 0) that reads a short sentence and predicts its sentiment using an **Embedding** → **SimpleRNN** → **Dense(sigmoid)** network.

Dataset & Labels

```

sentences = [
    "I love this movie", "This film was great",
    "What a fantastic experience", "I really enjoyed it",
    "Absolutely wonderful acting",
    "I hate this movie", "This film was terrible",
    "What a bad experience", "I really disliked it",
    "Absolutely horrible acting"
]
labels = [1,1,1,1,1, 0,0,0,0,0]
  • 10 short sentences (5 positive, 5 negative).
  • Balanced labels → avoids class-imbalance issues for this demo.

```

Tokenization (Text → Integer Sequences)

```

tokenizer = Tokenizer(num_words=50)      # keep top 50 words (by
frequency)
tokenizer.fit_on_texts(sentences)
X = tokenizer.texts_to_sequences(sentences)
print("Word Index:", tokenizer.word_index)
  • Tokenizer builds a vocabulary from the training texts.
  • num_words=50 means: at model time we will keep only the 50 most frequent tokens (IDs 1 .. 49 effectively; 0 is reserved for padding). Tokens with IDs  $\geq 50$  are treated as out-of-vocabulary if you later set an oov_token.
  • texts_to_sequences maps each word to its integer ID.

```

Example: If `tokenizer.word_index` contains `{"i":1, "this":2, "film":3, "was":4, "great":5, ...}`, then

- "I love this movie" → [1, <id(love)>, 2, <id(movie)>].

Padding (Uniform Sequence Length)

```
maxlen = 5
```

```
X = pad_sequences(X, maxlen=maxlen)
```

```
y = np.array(labels)
```

- RNNs expect **equal-length** sequences. `pad_sequences` left-pads with zeros to make each sequence length = `maxlen` (here 5).
- Shape after padding: `X.shape == (10, 5)` (10 sentences, 5 tokens each). Zeros are **padding token**.

Why 5? It's arbitrary and small for demo. Longer sentences get **truncated** (from the start by default). You can control truncation/padding side with `padding='post'`, `truncating='post'`.

Model Architecture

```
model = Sequential()
model.add(Embedding(input_dim=50, output_dim=8, input_length=maxlen))
model.add(SimpleRNN(8, activation="tanh"))
model.add(Dense(1, activation="sigmoid"))
```

Embedding layer

- **Purpose:** Learn a dense vector for each word ID (distributional semantics).
- **Shapes:**
 - Input: `(batch, timesteps=5)` of integer IDs
 - Output: `(batch, timesteps=5, embed_dim=8)` (each token → 8-D vector)
- **Parameters:** `input_dim * output_dim = 50 * 8 = 400` trainable params.

Important: `input_dim` must be \geq the maximum token index your tokenizer will produce (bounded by `num_words`). With `num_words=50`, `input_dim=50` is consistent.

SimpleRNN layer

- **Purpose:** Read the sequence of embeddings and produce a final hidden state summarizing the sentence.
- **Config:** `SimpleRNN(8, activation="tanh")`
- **Input dim D = 8** (embedding size), **units U = 8**.
- **Output shape:** `(batch, 8)` (last hidden state).
- **Parameters:** $U * (U + D + 1) \rightarrow 8 * (8 + 8 + 1) = 8 * 17 = **136**$.

Why `tanh`? Classic for vanilla RNNs. For more complex text, prefer **GRU/LSTM** to handle longer dependencies.

Dense output layer

- **Purpose:** Binary classification (positive vs negative).
- **Activation:** `sigmoid` → outputs probability p
- **Params:** weights $8 \times 1 = 8$ + bias $1 \rightarrow 9$.

Total params: 400 (Embedding) + 136 (RNN) + 9 (Dense) = **545**.

Compile & Train

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

```
model.fit(X, y, epochs=30, verbose=0)
```

- **Loss:** `binary_crossentropy` fits a sigmoid output for binary labels.
- **Optimizer:** Adam is a solid default.
- **Metrics:** Accuracy for quick feedback.

With such a tiny dataset and 30 epochs, the model will likely **memorize**.

Inference (Predict on New Sentences)

```
test_sentences = [ "I enjoyed this film", "I hated this film" ]
test_seq = tokenizer.texts_to_sequences(test_sentences)
test_seq = pad_sequences(test_seq, maxlen=maxlen)
pred = model.predict(test_seq)
for s, p in zip(test_sentences, pred):
    print(f"Sentence: {s} -> Sentiment:", "Positive" if p>0.5 else "Negative")
  • The same tokenizer is used at inference to ensure word-to-ID mapping consistency.
```

- Pad to `maxlen=5`.

- Apply `sigmoid` scores and threshold at **0.5**:

- $p > 0.5 \rightarrow$ Positive; else Negative.

End-to-End Shapes

- **Training:**

- $X: (10, 5) \rightarrow$ Embedding $\rightarrow (10, 5, 8) \rightarrow$ SimpleRNN $\rightarrow (10, 8) \rightarrow$ Dense $\rightarrow (10, 1)$

- **Inference:**

- `test_seq: (2, 5) → (2, 5, 8) → (2, 8) → (2, 1)`



Application 6

Supervised Machine Learning

User Defined K Nearest Neighbour Algorithm

- In this application we create our own algorithm for classified machine learning.
- We create our own K Nearest Neighbour algorithm.
- For user defined algorithm we design one class named as MarvellousKNN.
- This class contains 3 methods as fit, predict, closest method.
- There is one naked method euc() which calculate distance between two points using Euclidean distance algorithm.
- fit() method initialises training data and its targets inside class.
- predict() method creates one array as prediction which stores shortest distance between all test data and training data elements.
- predict() method calls closest method which returns the shortest distance.

Consider below characteristics of Machine Learning Application :

Classifier :	User Defined K Nearest Neighbour
DataSet :	Iris Dataset
Features :	Sepal Width, Sepal Length, Petal Width, Petal Length
Labels :	Versicolor, Setosa , Virginica
Training Dataset :	75 Entries
Testing Dataset :	75 Entries

```

1 from sklearn import tree
2 from scipy.spatial import distance
3 from sklearn.datasets import load_iris
4 from sklearn.metrics import accuracy_score
5 from sklearn.model_selection import train_test_split
6
7 def euc(a,b):
8     return distance.euclidean(a,b)
9
10 class MarvellousKNN():
11     def fit(self,TrainingData,TrainingTarget):
12         self.TrainingData = TrainingData
13         self.TrainingTarget = TrainingTarget
14
15     def predict(self,TestData):
16         predictions = []
17         for row in TestData:
18             label = self.closest(row)
19             predictions.append(label)
20
21         return predictions
  
```

```
22 def closest(self,row):
23     bestdistance = euc(row,self.TrainingData[0])
24     bestindex = 0
25     for i in range(1,len(self.TrainingData)):
26         dist = euc(row,self.TrainingData[i])
27         if dist < bestdistance:
28             bestdistance = dist
29             bestindex = i
30     return self.TrainingTarget[bestindex]
31
32 def MarvellousKNeighbor():
33     border = "-"*50
34
35     iris = load_iris()
36
37     data = iris.data
38     target = iris.target
39
40     print(border)
41     print("Actual data set")
42     print(border)
43
44     for i in range(len(iris.target)):
45         print("ID: %d, Label %s, Feature : %s" % (i,iris.data[i],iris.target[i]))
46     print("Size of Actual data set %d"%(i+1))
47
48     data_train, data_test, target_train, target_test = train_test_split(data,target,test_size=0.5)
49
50
51     print(border)
52     print("Training data set")
53     print(border)
54     for i in range(len(data_train)):
55         print("ID: %d, Label %s, Feature : %s" % (i,data_train[i],target_train[i]))
56     print("Size of Training data set %d"%(i+1))
57
58     print(border)
59     print("Test data set")
60     print(border)
61     for i in range(len(data_test)):
62         print("ID: %d, Label %s, Feature : %s" % (i,data_test[i],target_test[i]))
63     print("Size of Test data set %d"%(i+1))
64     print(border)
65
66     classifier = MarvellousKNN()
67
68     classifier.fit(data_train,target_train)
69
70     predictions = classifier.predict(data_test)
71
72     Accuracy = accuracy_score(target_test,predictions)
73
74     return Accuracy
75
76 def main():
77
78     Accuracy = MarvellousKNeighbor()
79     print("Accuracy of classification algorithm with K Neighbor classifier is",Accuracy*100,"%")
80
81 if __name__ == "__main__":
82     main()
83
```

Output of above Application :

```

(base) MacBook-Pro-de-MARVELLOUS:iris marvellous$ python3 MarvellousClassifier.py
-----
Actual data set
-----
ID: 0, Label: [5.1 3.5 1.4 0.2], Feature: 0
ID: 1, Label: [4.9 3.0 1.4 0.2], Feature: 0
ID: 2, Label: [4.7 3.2 1.3 0.2], Feature: 0
ID: 3, Label: [4.6 3.1 1.5 0.2], Feature: 0
ID: 4, Label: [5.0 3.6 1.4 0.2], Feature: 0
ID: 5, Label: [5.4 3.9 1.7 0.4], Feature: 0
ID: 6, Label: [4.6 3.4 1.4 0.3], Feature: 0
ID: 7, Label: [5.0 3.4 1.5 0.2], Feature: 0
ID: 8, Label: [4.4 2.9 1.4 0.2], Feature: 0
ID: 9, Label: [4.9 3.1 1.5 0.1], Feature: 0
ID: 10, Label: [5.4 3.7 1.5 0.2], Feature: 0
ID: 11, Label: [4.8 3.4 1.6 0.2], Feature: 0
ID: 12, Label: [4.8 3.0 1.4 0.1], Feature: 0
ID: 13, Label: [4.3 3.0 1.1 0.1], Feature: 0
ID: 14, Label: [5.8 4.0 1.2 0.2], Feature: 0
ID: 146, Label: [6.3 2.5 5.0 1.9], Feature: 2
ID: 147, Label: [6.5 3.0 5.2 2.0], Feature: 2
ID: 148, Label: [6.2 3.4 5.4 2.3], Feature: 2
ID: 149, Label: [5.9 3.0 5.1 1.8], Feature: 2
Size of Actual data set 150
-----
Training data set
-----
ID: 0, Label: [5.4 3.0 4.5 1.5], Feature: 1
ID: 1, Label: [5.4 3.7 1.5 0.2], Feature: 0
ID: 2, Label: [6.0 2.7 5.1 1.6], Feature: 1
ID: 3, Label: [5.0 3.5 1.6 0.6], Feature: 0
ID: 4, Label: [6.9 3.1 4.9 1.5], Feature: 1
ID: 5, Label: [5.0 2.0 3.5 1.0], Feature: 1
ID: 6, Label: [5.5 4.2 1.4 0.2], Feature: 0
ID: 7, Label: [5.5 3.5 1.3 0.2], Feature: 0
ID: 8, Label: [4.6 3.6 1.0 0.2], Feature: 0
ID: 9, Label: [4.9 3.1 1.5 0.2], Feature: 0
ID: 10, Label: [6.4 2.9 4.3 1.3], Feature: 1

```

```
ID: 72, Label: [ 5.  3. 6  1. 4  0. 2], Feature: 0
ID: 73, Label: [ 5. 7  2. 9  4. 2  1. 3], Feature: 1
ID: 74, Label: [ 5.  3. 4  1. 6  0. 4], Feature: 0
Size of Training data set 75
```

Test data set

```
ID: 0, Label: [ 6. 1  2. 6  5. 6  1. 4], Feature: 2
ID: 1, Label: [ 6. 2  3. 4  5. 4  2. 3], Feature: 2
ID: 2, Label: [ 4. 8  3. 4  1. 9  0. 2], Feature: 0
ID: 3, Label: [ 5. 4  3. 9  1. 3  0. 4], Feature: 0
ID: 4, Label: [ 5. 7  2. 6  3. 5  1. ], Feature: 1
ID: 5, Label: [ 4. 8  3.  1. 4  0. 1], Feature: 0
ID: 6, Label: [ 5. 5  2. 6  4. 4  1. 2], Feature: 1
ID: 7, Label: [ 7. 2  3.  5. 8  1. 6], Feature: 2
ID: 8, Label: [ 4. 6  3. 4  1. 4  0. 3], Feature: 0
ID: 9, Label: [ 5. 3  3. 7  1. 5  0. 2], Feature: 0
ID: 10, Label: [ 5. 1  3. 3  1. 7  0. 5], Feature: 0
ID: 11, Label: [ 6. 5  3.  5. 2  2. ], Feature: 2
ID: 62, Label: [ 6. 9  3. 2  5. 7  2. 3], Feature: 2
ID: 63, Label: [ 7. 2  3. 2  6.  1. 8], Feature: 2
ID: 64, Label: [ 5. 8  2. 7  5. 1  1. 9], Feature: 2
ID: 65, Label: [ 4. 6  3. 2  1. 4  0. 2], Feature: 0
ID: 66, Label: [ 6. 7  2. 5  5. 8  1. 8], Feature: 2
ID: 67, Label: [ 5. 1  3. 8  1. 9  0. 4], Feature: 0
ID: 68, Label: [ 7. 1  3.  5. 9  2. 1], Feature: 2
ID: 69, Label: [ 4. 3  3.  1. 1  0. 1], Feature: 0
ID: 70, Label: [ 5. 8  2. 7  5. 1  1. 9], Feature: 2
ID: 71, Label: [ 4. 5  2. 3  1. 3  0. 3], Feature: 0
ID: 72, Label: [ 5. 9  3.  5. 1  1. 8], Feature: 2
ID: 73, Label: [ 4. 6  3. 1  1. 5  0. 2], Feature: 0
ID: 74, Label: [ 7. 7  2. 6  6. 9  2. 3], Feature: 2
Size of Test data set 75
```

Accuracy of classification algorithm with K Neighbour classifier is 97.33333333333334 %
(base) MacBook-Pro-de-MARVELLOUS:iris marvellous\$ █

Supervised Machine Learning

Supervised Machine Learning :

Supervised Machine Learning is a type of machine learning where the model is trained on a **labeled dataset** — which means each input (X) is associated with a correct output (Y). The algorithm learns the relationship between inputs and outputs and uses it to **predict outcomes for new, unseen data**.

Example:

If you're training a model to classify emails as *Spam* or *Not Spam*, you feed it emails with the correct labels. The model learns the pattern and then can predict future emails.

Types of Supervised Learning

Supervised learning is broadly categorized into:

1. **Classification**
2. **Regression**

1. Classification

In classification, the output is **categorical** (discrete labels or classes).

Examples:

- Email Spam Detection (Spam / Not Spam)
- Disease Diagnosis (Positive / Negative)
- Handwriting Recognition (Digits 0–9)

Popular Algorithms:

Algorithm	Description
Logistic Regression	Used for binary and multi-class classification. Despite its name, it's a classification method.
K-Nearest Neighbors (KNN)	Classifies a data point based on the majority class among its 'K' nearest neighbors.
Decision Tree	A tree-like model of decisions, used for classification and regression.
Random Forest	An ensemble of decision trees, improves accuracy and reduces overfitting.
Support Vector Machine (SVM)	Finds the best boundary (hyperplane) that separates classes.

2. Regression

In regression, the output is **continuous** (numeric values).

✓ Examples:

- Predicting house prices
- Forecasting stock market trends
- Estimating car mileage

Popular Algorithms:

Algorithm	Description
Linear Regression	Models the relationship between dependent and independent variables with a straight line.
Polynomial Regression	Fits a polynomial curve instead of a straight line for non-linear relationships.

Supervised Learning Flow

1. Collect labeled data
2. Split into training and testing sets
3. Choose appropriate algorithm
4. Train the model on training data
5. Evaluate the model on test data (using metrics like accuracy, RMSE, etc.)
6. Use model for prediction on new data

Evaluation Metrics

Classification:

- Accuracy
- Precision
- Recall
- F1 Score
- Confusion Matrix

Regression:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)

- Root Mean Squared Error (RMSE)
- R² Score (Coefficient of Determination)



Application 5

Supervised Machine Learning

Accuracy Calculation with Decision Tree & K Nearest Neighbour

- In this application we train iris data set with Decision Tree algorithm and K Nearest Neighbour algorithm.
- We divide iris data set into two equal parts as training data and test data.
- We apply training on training data and predict the result on test data.
- We calculate accuracy of both the algorithms by applying of test data.

Consider below characteristics of Machine Learning Application :

Classifier :	Decision Tree & K Nearest Neighbour
DataSet :	Iris Dataset
Features :	Sepal Width, Sepal Length, Petal Width, Petal Length
Labels :	Versicolor, Setosa , Virginica
Training Dataset :	75 Entries
Testing Dataset :	75 Entries

```

1 from sklearn import tree
2 from sklearn.datasets import load_iris
3 from sklearn.metrics import accuracy_score
4 from sklearn.neighbors import KNeighborsClassifier
5 from sklearn.model_selection import train_test_split
6
7 def MarvellousCalculateAccuracyDecisionTree():
8     iris = load_iris()
9
10    data = iris.data
11    target = iris.target
12
13    data_train, data_test, target_train, target_test = train_test_split(data,target,test_size=0.5)
14
15    classifier = tree.DecisionTreeClassifier()
16
17    classifier.fit(data_train,target_train)
18
19    predictions = classifier.predict(data_test)
20
21    Accuracy = accuracy_score(target_test,predictions)
22
23    return Accuracy
24
  
```

```

25
26 def MarvellousCalculateAccuracyKNeighbor():
27   iris = load_iris()
28
29   data = iris.data
30   target = iris.target
31
32   data_train, data_test, target_train, target_test = train_test_split(data,target,test_size=0.5)
33
34   classifier = KNeighborsClassifier()
35
36   classifier.fit(data_train,target_train)
37
38   predictions = classifier.predict(data_test)
39
40   Accuracy = accuracy_score(target_test,predictions)
41
42   return Accuracy
43
44 def main():
45   Accuracy = MarvellousCalculateAccuracyDecisionTree()
46   print("Accuracy of classification algorithm with Decision Tree classifier is",Accuracy*100,"%")
47
48   Accuracy = MarvellousCalculateAccuracyKNeighbor()
49   print("Accuracy of classification algorithm with K Neighbor classifier is",Accuracy*100,"%")
50
51 if __name__ == "__main__":
52   main()
53

```

Output of above application

```

(base) MacBook-Pro-de-MARVELLOUS:iris marvellous$ python3 irisAccuracy.py
Accuracy of classification algorithm with Decision Tree classifier is 94.66666666666667 %
Accuracy of classification algorithm with K Neighbor classifier is 97.33333333333334 %
(base) MacBook-Pro-de-MARVELLOUS:iris marvellous$ █

```

Accuracy with Decision Tree algorithm is 94% and with KNN is 97%.

Note : Accuracy may vary.

Supervised VS Unsupervised Machine Learning

Supervised learning:

Supervised learning is the learning of the model where with input variable (say, x) and an output variable (say, Y) and an algorithm to map the input to the output.

That is, $Y = f(X)$

Why supervised learning?

The basic aim is to approximate the mapping function(mentioned above) so well that when there is a new input data (x) then the corresponding output variable can be predicted.

It is called supervised learning because the process of learning (from the training dataset) can be thought of as a teacher who is supervising the entire learning process. Thus, the “learning algorithm” iteratively makes predictions on the training data and is corrected by the “teacher”, and the learning stops when the algorithm achieves an acceptable level of performance(or the desired accuracy).

Example of Supervised Learning



Suppose there is a basket which is filled with some fresh fruits, the task is to arrange the same type of fruits at one place.

Also, suppose that the fruits are apple, banana, cherry, grape.

Suppose one already knows from their previous work (or experience) that, the shape of each and every fruit present in the basket so, it is easy for them to arrange the same type of fruits in one place.

Here, the previous work is called as training data in Data Mining terminology. So, it learns the things from the training data. This is because it has a response variable which says y that if some fruit has so and so features then it is grape, and similarly for each and every fruit.

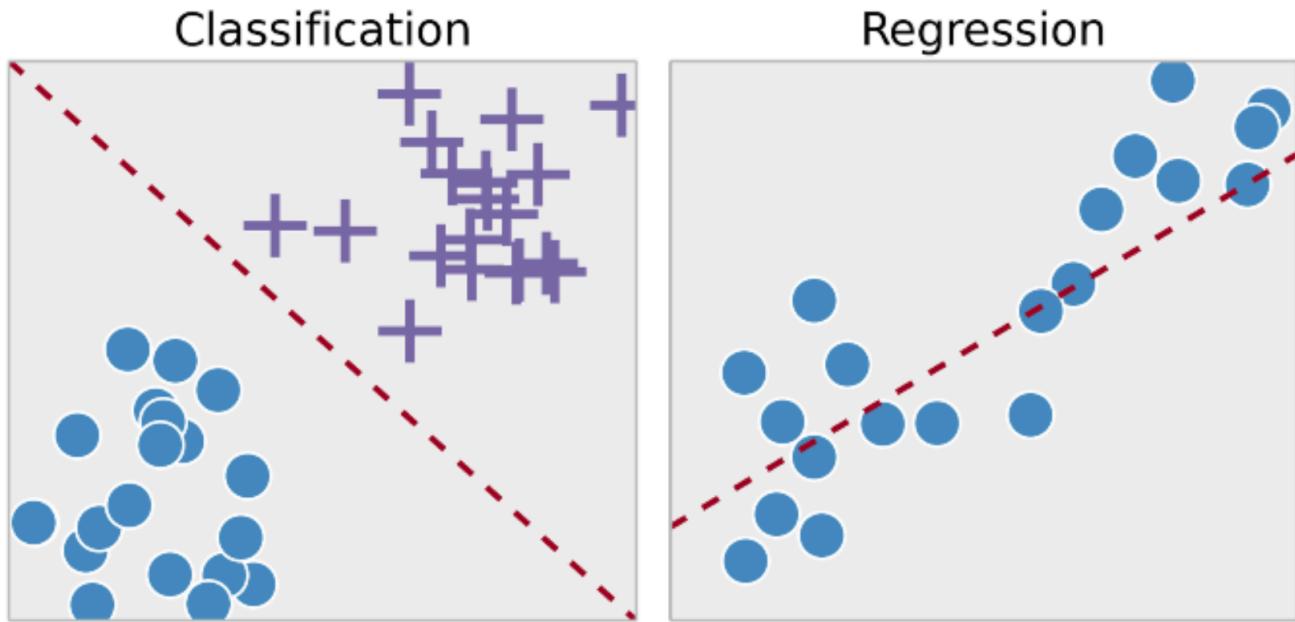
This type of information is deciphered from the data that is used to train the model.

This type of learning is called Supervised Learning.

Such problems are listed under classical Classification Tasks.

Types of Supervised Machine Learning Techniques

Regression:



Regression technique predicts a single output value using training data.

Example: You can use regression to predict the house price from training data. The input variables will be locality, size of a house, etc.

Classification:

Classification means to group the output inside a class. If the algorithm tries to label input into two distinct classes, it is called binary classification. Selecting between more than two classes is referred to as multiclass classification.

Example: Determining whether or not someone will be a defaulter of the loan.

Strengths: Outputs always have a probabilistic interpretation, and the algorithm can be regularized to avoid overfitting.

Weaknesses: Logistic regression may underperform when there are multiple or non-linear decision boundaries. This method is not flexible, so it does not capture more complex relationships.

Unsupervised Learning:

Unsupervised learning is where only the input data (say, X) is present and no corresponding output variable is there.

Why Unsupervised Learning?

The main aim of Unsupervised learning is to model the distribution in the data in order to learn more about the data.

It is called so, because there is no correct answer and there is no such teacher(unlike supervised learning). Algorithms are left to their own devices to discover and present the interesting structure in the data.

Example of Unsupervised Learning



Again, Suppose there is a basket and it is filled with some fresh fruits. The task is to arrange the same type of fruits at one place.

This time there is no information about those fruits beforehand, its the first time that the fruits are being seen or discovered

So how to group similar fruits without any prior knowledge about those.

First, any physical characteristic of a particular fruit is selected. Suppose color.

Then the fruits are arranged on the basis of the color. The groups will be something as shown below:

RED COLOR GROUP: apples & cherry fruits.

GREEN COLOR GROUP: bananas & grapes.

So now, take another physical character say, size, so now the groups will be something like this.

RED COLOR AND BIG SIZE: apple.

RED COLOR AND SMALL SIZE: cherry fruits.

GREEN COLOR AND BIG SIZE: bananas.

GREEN COLOR AND SMALL SIZE: grapes.

Here, there is no need to know or learn anything beforehand. That means, no train data and no response variable. This type of learning is known as Unsupervised Learning.

Types of Unsupervised Machine Learning Techniques

Unsupervised learning problems further grouped into clustering and association problems.

Clustering

Clustering is an important concept when it comes to unsupervised learning. It mainly



sample



Cluster/group

deals with finding a structure or pattern in a collection of uncategorized data. Clustering algorithms will process your data and find natural clusters(groups) if they exist in the data. You can also modify how many clusters your algorithms should identify. It allows you to adjust the granularity of these groups.

Association

Association rules allow you to establish associations amongst data objects inside large databases. This unsupervised technique is about discovering exciting relationships between variables in large databases. For example, people that buy a new home most likely to buy new furniture.

Other Examples:

- A subgroup of cancer patients grouped by their gene expression measurements
- Groups of shopper based on their browsing and purchasing histories
- Movie group by the rating given by movies viewers

Difference between Supervised & Unsupervised Machine Learning

Parameters	Supervised machine learning technique	Unsupervised machine learning technique
Process	In a supervised learning model, input and output variables will be given.	In unsupervised learning model, only input data will be given
Input Data	Algorithms are trained using labeled data.	Algorithms are used against data which is not labeled
Algorithms Used	Support vector machine, Neural network, Linear and logistics regression, random forest, and Classification trees.	Unsupervised algorithms can be divided into different categories: like Cluster algorithms, K-means, Hierarchical clustering, etc.
Computational Complexity	Supervised learning is a simpler method.	Unsupervised learning is computationally complex
Use of Data	Supervised learning model uses training data to learn a link between the input and the outputs.	Unsupervised learning does not use output data.
Accuracy of Results	Highly accurate and trustworthy method.	Less accurate and trustworthy method.
Real Time Learning	Learning method takes place offline.	Learning method takes place in real time.
Number of Classes	Number of classes is known.	Number of classes is not known.
Main Drawback	Classifying big data can be a real challenge in Supervised Learning.	You cannot get precise information regarding data sorting, and the output as data used in unsupervised learning is labeled and not known.

Summary

- In Supervised learning, you train the machine using data which is well "labeled."
- Unsupervised learning is a machine learning technique, where you do not need to supervise the model.
- Supervised learning allows you to collect data or produce a data output from the previous experience.
- Unsupervised machine learning helps you to finds all kind of unknown patterns in data.
- For example, you will able to determine the time taken to reach back home based on weather condition, Times of the day and holiday.
- For example, Baby can identify other dogs based on past supervised learning.
- Regression and Classification are two types of supervised machine learning techniques.
- Clustering and Association are two types of Unsupervised learning.
- In a supervised learning model, input and output variables will be given while with unsupervised learning model, only input data will be given.

Support Vector Machine

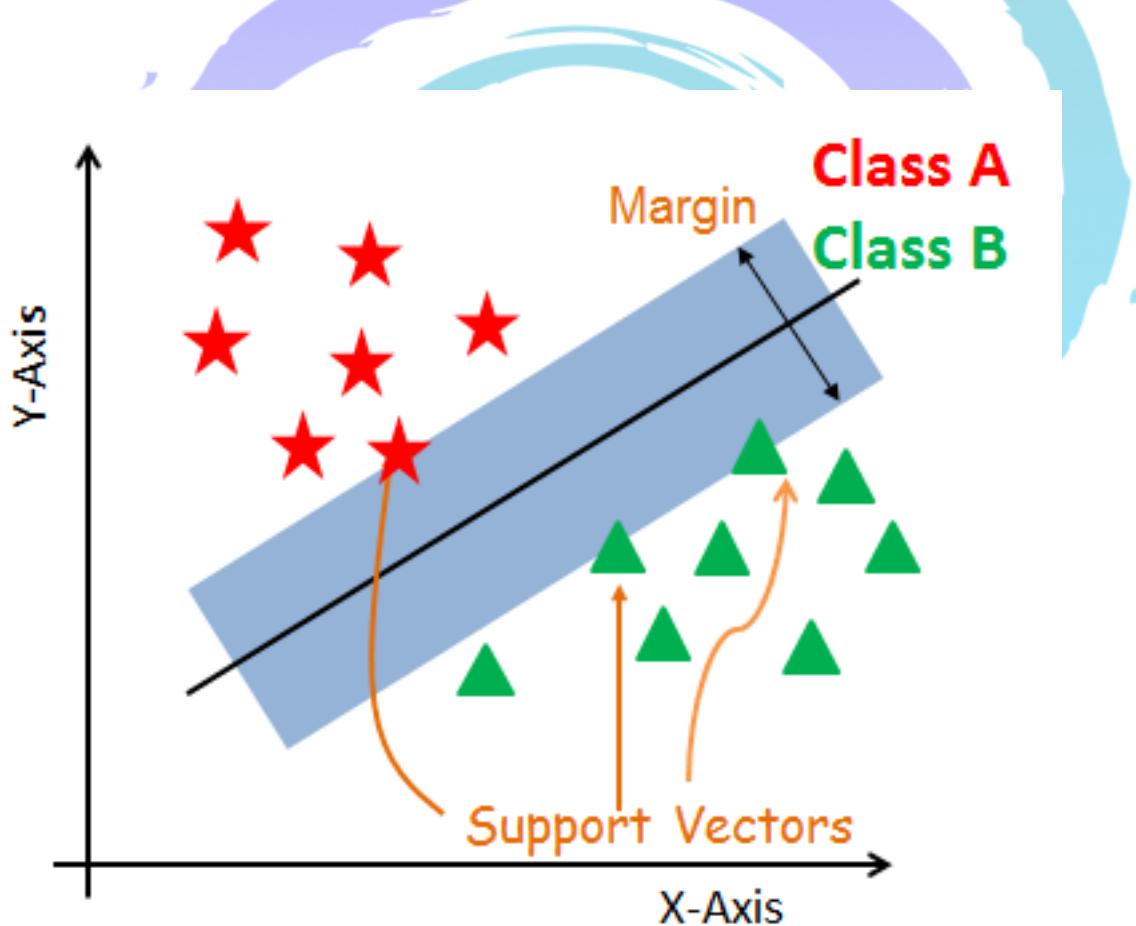
SVM offers very high accuracy compared to other classifiers such as logistic regression, and decision trees. It is known for its kernel trick to handle nonlinear input spaces. It is used in a variety of applications such as face detection, intrusion detection, classification of emails, news articles and web pages, classification of genes, and handwriting recognition.

Support Vector Machines

Support Vector Machines is considered to be a classification approach. It can easily handle multiple continuous and categorical variables. SVM constructs a hyperplane in multidimensional space to separate different classes.

SVM generates optimal hyperplane in an iterative manner, which is used to minimize an error.

The core idea of SVM is to find a maximum marginal hyperplane(MMH) that best divides the dataset into classes.



Support Vectors

Support vectors are the data points, which are closest to the hyperplane. These points will define the separating line better by calculating margins. These points are more relevant to the construction of the classifier.

Hyperplane

A hyperplane is a decision plane which separates between a set of objects having different class memberships.

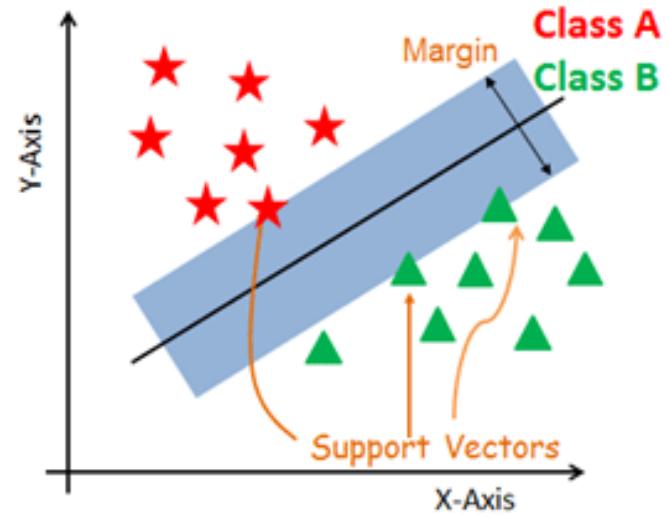
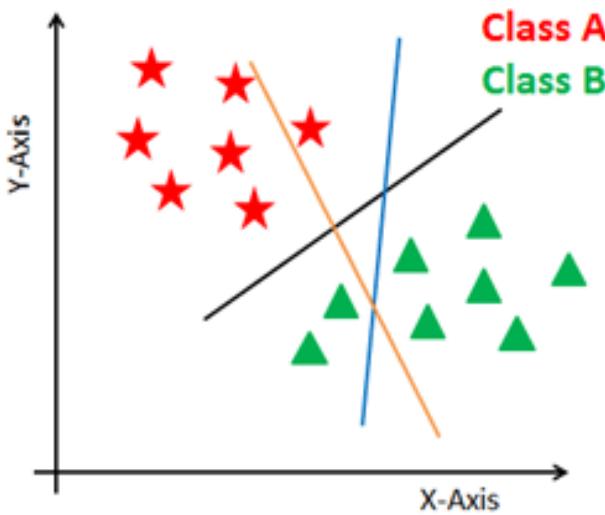
Margin

A margin is a gap between the two lines on the closest class points. This is calculated as the perpendicular distance from the line to support vectors or closest points. If the margin is larger in between the classes, then it is considered a good margin, a smaller margin is a bad margin.

How does SVM work?

The main objective is to segregate the given dataset in the best possible way. The distance between the either nearest points is known as the margin. The objective is to select a hyperplane with the maximum possible margin between support vectors in the given dataset. SVM searches for the maximum marginal hyperplane in the following steps:

1. Generate hyperplanes which segregates the classes in the best way. Left-hand side figure showing three hyperplanes black, blue and orange. Here, the blue and orange have higher classification error, but the black is separating the two classes correctly.
2. Select the right hyperplane with the maximum segregation from the either nearest data points as shown in the right-hand side figure.



Breast Cancer Dataset with Support Vector Machine

```

1 from sklearn import datasets
2 from sklearn.model_selection import train_test_split
3 from sklearn import svm
4 from sklearn import metrics
5
6 def MarvellousSVM():
7     #Load dataset
8     cancer = datasets.load_breast_cancer()
9
10    # print the names of the 13 features
11    print("Features of the cancer dataset : ", cancer.feature_names)
12
13    # print the label type of cancer('malignant' 'benign')
14    print("Labels of the cancer dataset : ", cancer.target_names)
15
16    # print data(feature)shape
17    print("Shape of dataset is : ",cancer.data.shape)
18
19    # print the cancer data features (top 5 records)
20    print("First 5 records are : ")
21    print(cancer.data[0:5])
22
23    # print the cancer labels (0:malignant, 1:benign)
24    print("Target of dataset : ", cancer.target)
25
26    # Split dataset into training set and test set
27    X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
28                                                       test_size=0.3,random_state=109) # 70% training and 30% test
29
30    #Create a svm Classifier
31    clf = svm.SVC(kernel='linear') # Linear Kernel
32
33    #Train the model using the training sets
34    clf.fit(X_train, y_train)
35
36    #Predict the response for test dataset
37    y_pred = clf.predict(X_test)
38
39    # Model Accuracy: how often is the classifier correct?
40    print("Accuracy of the model is : ",metrics.accuracy_score(y_test, y_pred)*100)
41
42 def main():
43     print("_____ Marvellous Support Vector Machine _____")
44
45     MarvellousSVM()
46
47 if __name__ == "__main__":
48     main()

```

User Defined Functions

There are multiple syntactical ways in which we can design user defined functions.

Consider below application which demonstrate different concepts that we can apply while defining user defined functions

```
print("---- Marvellous Infosystems by Piyush Khairnar----")
```

```
print("Demonstration of Advanced Functions")
```

Function which accepts nothing and return nothing

```
def Marvellous1():
    print("Inside Marvellous1")
```

Function which accepts value and return nothing

```
def Marvellous2(value):
    print("Inside Marvellous2")
    print("Accepted value is ",value)
```

Function which accepts value and return value

```
def Marvellous3(value):
    print("Inside Marvellous3")
    print("Accepted value is ",value)
    return value+1
```

Function which accepts multiple values and return multiple values

```
def Marvellous4(value1, value2):
    print("Inside Marvellous4")
    add = value1 + value2
    sub = value1 - value2
    return add,sub
```

Function which calls another function which is defined outside it.

```
def Marvellous5():
    print("Inside Marvellous5")
    Marvellous1()
```

Function which contains another nested function defined in it.

```
def Marvellous6():
    print("Inside Marvellous6")
    def InnerFun():
        print("Inside InnerFun")
    InnerFun()
```

Function calls for above functions

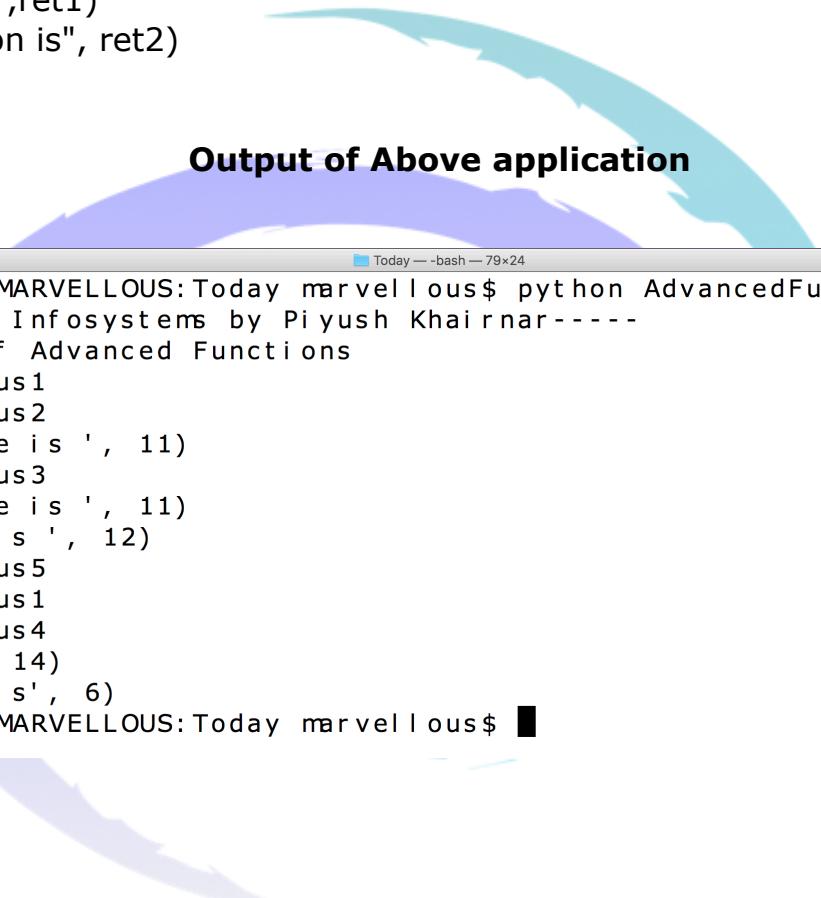
```
no = 11
```

```
Marvellous1()  
Marvellous2(no)  
ret = Marvellous3(no)  
print("Return value is ", ret)
```

```
Marvellous5()
```

```
ret1,ret2 = Marvellous4(10,4)  
print("Addition is",ret1)  
print("Substraction is", ret2)
```

Output of Above application



```
MacBook-Pro-de-MARVELLOUS: Today marvellous$ python AdvancedFunction.py  
---- Marvellous Infosystems by Piyush Khairnar-----  
Demonstration of Advanced Functions  
Inside Marvellous1  
Inside Marvellous2  
('Accepted value is ', 11)  
Inside Marvellous3  
('Accepted value is ', 11)  
('Return value is ', 12)  
Inside Marvellous5  
Inside Marvellous1  
Inside Marvellous4  
('Addition is', 14)  
('Substraction is', 6)  
MacBook-Pro-de-MARVELLOUS: Today marvellous$ █
```

TensorFlow Applications for Deep Learning

- **TensorFlow** is an open-source deep learning framework developed by Google.
- It provides tools for **tensor computations**, **automatic differentiation**, and **GPU/TPU acceleration**.
- Core components:
 - **Tensors** → Multidimensional arrays (like NumPy arrays but optimized for GPU/TPU).
 - **Operations (Ops)** → Functions that manipulate tensors.
 - **Graph Execution** and **Eager Execution** → Modes of running code.
 - **GradientTape** → Automatic differentiation engine.

Consider the below application programs to understand important concepts of TensorFlow

Tensor Constants

Constants are fixed values in the computation graph.

 **Program Example:**

```
import tensorflow as tf

# Create constant tensors
a = tf.constant(5)
b = tf.constant(3)

print("Tensor a:", a)
print("Tensor b:", b)
```

 **Explanation:**

- `tf.constant(value)` creates an **immutable tensor**.
- `a` and `b` are **0-D tensors (scalars)**.

Tensor Arithmetic Operations

TensorFlow supports element-wise operations like addition, subtraction, multiplication, and division.

 **Program Example:**

```
import tensorflow as tf

a = tf.constant(5)
b = tf.constant(3)
```

```

add = tf.add(a, b)
sub = tf.subtract(a, b)
mul = tf.multiply(a, b)
div = tf.divide(a, b)

print("a + b =", add.numpy())
print("a - b =", sub.numpy())
print("a * b =", mul.numpy())
print("a / b =", div.numpy())
  
```

 **Explanation:**

- `tf.add, tf.subtract, tf.multiply, tf.divide` → perform basic math.
- `.numpy()` converts tensor result into NumPy value (works in **Eager Execution** mode).

Tensor Operations with Matrices

Tensors extend beyond scalars to **2D matrices** and **ND arrays**.

 **Program Example:**

```

import tensorflow as tf

# Define matrices
A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
B = tf.constant([[5, 6], [7, 8]], dtype=tf.float32)

matmul = tf.matmul(A, B)          # Matrix multiplication
transpose = tf.transpose(A)      # Transpose

print("Matrix A:\n", A.numpy())
print("Matrix B:\n", B.numpy())
print("A x B:\n", matmul.numpy())
print("Transpose of A:\n", transpose.numpy())
  
```

 **Explanation:**

- `tf.matmul(A, B)` → Performs **matrix multiplication**.
- `tf.transpose(A)` → Switches rows and columns.
- TensorFlow automatically handles **linear algebra operations** used in deep learning (like weights \times inputs).

Broadcasting in TensorFlow

Broadcasting allows operations between tensors of different shapes.

📌 Program Example:

```
import tensorflow as tf

C = tf.constant([1, 2, 3], dtype=tf.float32)
D = tf.constant(2.0)

broadcast_add = C + D
broadcast_mul = C * D

print("C:", C.numpy())
print("C + D:", broadcast_add.numpy())
print("C * D:", broadcast_mul.numpy())
```

🔍 Explanation:

- Tensor [1, 2, 3] and scalar 2.0 are **broadcasted**.
- Rule: TensorFlow expands dimensions automatically if compatible.
- Useful in **neural networks** when applying bias terms across batches.

Automatic Differentiation with GradientTape

Deep learning requires gradient computation for optimization.

📌 Program Example:

```
import tensorflow as tf

x = tf.Variable(3.0)

with tf.GradientTape() as tape:
    y = x**2 + 2*x + 1    # Function y

grad = tape.gradient(y, x)

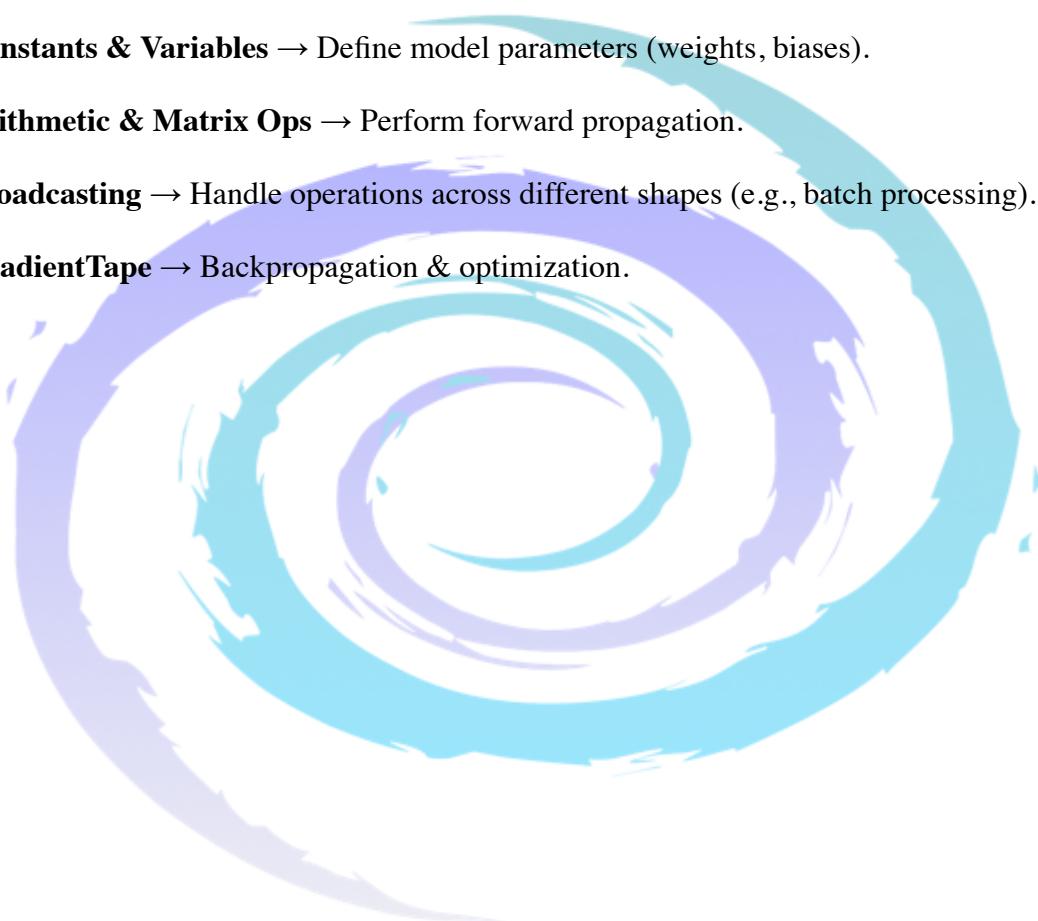
print("Function: y = x^2 + 2x + 1 at x=3")
print("Gradient dy/dx =", grad.numpy())    # Expected: 8
```

Explanation:

- `tf.Variable` → trainable parameter.
- `tf.GradientTape` → records operations for automatic differentiation.
- `tape.gradient(y, x)` → computes derivative (dy/dx).
- Essential for **backpropagation** in neural networks.

Relevance to Deep Learning

- **Constants & Variables** → Define model parameters (weights, biases).
- **Arithmetic & Matrix Ops** → Perform forward propagation.
- **Broadcasting** → Handle operations across different shapes (e.g., batch processing).
- **GradientTape** → Backpropagation & optimization.



TensorFlow for Deep Learning

TensorFlow is an open-source deep learning framework developed by Google.

It provides tools for **tensor computations**, **automatic differentiation**, and **GPU/TPU acceleration**.

Core components:

- **Tensors** → Multidimensional arrays (like NumPy arrays but optimized for GPU/TPU).
- **Operations (Ops)** → Functions that manipulate tensors.
- **Graph Execution** and **Eager Execution** → Modes of running code.
- **GradientTape** → Automatic differentiation engine.
- A **Tensor** is a **multidimensional array** (generalization of scalars, vectors, and matrices).
- TensorFlow represents all data as tensors, making it the **core data structure**.

Tensor Ranks (Dimensions):

1. **0-D Tensor (Scalar)** → Single number
Example: `tf.constant(5)`
`5`
2. **1-D Tensor (Vector)** → Ordered list of numbers
Example: `tf.constant([1, 2, 3])`
`[1, 2, 3]`
3. **2-D Tensor (Matrix)** → Table of numbers
Example:
`[[1, 2],`
`[3, 4]]`
4. **3-D Tensor (Cube / Volume)** → Stack of matrices
Example: Images with channels (RGB).
5. **n-D Tensor** → Higher dimensions (e.g., a batch of images, each with multiple channels).

Tensor Properties:

- **Shape** → Dimensions of tensor (e.g., `[3, 2]` means 3 rows, 2 columns).
- **Rank** → Number of dimensions (scalar = 0, vector = 1, matrix = 2, etc.).
- **Data type (dtype)** → e.g., `float32`, `int32`.

◆ Tensors are Important in Deep Learning

Deep learning models are essentially **mathematical functions** that transform input tensors into output tensors using **layers**.

- **Input Data Representation:**

- Images → 4D tensors (`batch_size, height, width, channels`).
- Text → 2D/3D tensors (`batch_size, sequence_length, embedding_dim`).
- Time series → 3D tensors (`batch_size, timesteps, features`).

- **Weights & Biases:**

- Each layer has trainable tensors for weights and biases.
- Example: Dense (fully connected) layer with 128 inputs and 64 outputs has a **weight tensor of shape [128, 64]**.

- **Forward Propagation:**

- Tensors flow through layers (matrix multiplication, activation functions).
- Example:

$$Y = f(X \cdot W + b)$$
 - X = input tensor,
 - W = weight tensor,
 - b = bias tensor.

- **Backpropagation (Training):**

- GradientTape computes **gradients of loss wrt tensors (weights & biases)**.
- Optimizers (SGD, Adam) update the tensors to minimize loss.

◆ Relevance of Tensors in Neural Networks

1. Data Storage:

- All input/output and intermediate results are stored as tensors.

2. Matrix Computations:

- Neural networks rely heavily on **matrix multiplications** (tensor × tensor).
- GPUs/TPUs optimize tensor operations massively for speed.

3. Broadcasting:

- Bias vectors applied across a batch of inputs.
- Example: Adding a 1D tensor (bias) to a 2D tensor (matrix of activations).

4. Automatic Differentiation:

- Gradients wrt tensor values are essential for training.

5. Real Applications:

- **Computer Vision:** Images as tensors (Convolutional Neural Networks).
- **NLP:** Sentences converted to tensor embeddings (Transformers).
- **Speech Recognition:** Audio signals represented as tensors.
- **Reinforcement Learning:** States, actions, and rewards stored as tensors.

◆ Example Flow in Deep Learning with Tensors

👉 Suppose we build a simple image classifier:

- Input: Batch of 32 images, each 28×28 grayscale → Tensor shape [32 , 28 , 28 , 1].
- Layer 1: Convolution filter ($3 \times 3 \times 1 \times 32$) → Extracts features.
- Layer 2: Dense Layer (weights [6272, 128]) → Flattened features to hidden neurons.
- Output: Dense Layer (weights [128, 10]) → Probability tensor for 10 classes (digits 0–9).

At each step:

- **Forward pass:** Tensors flow through layers.
- **Backward pass:** Tensor gradients update weights.

Time-Series Next-Value Prediction with a SimpleRNN

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
```

1) Reproducibility

```
np.random.seed(42)
tf.random.set_seed(42)
```

2) Create a longer sequence (0..199)

```
data = np.arange(0, 200, dtype=np.float32)
```

3) Normalize to [0, 1] to help training

```
max_val = data.max()
data_norm = data / max_val
```

4) Make sliding windows: use 'window' numbers to predict the next one

```
window = 5 # timesteps
```

```
X, y = [], []
```

```
for i in range(len(data_norm) - window):
```

```
    X.append(data_norm[i:i+window]) # e.g. [0,1,2,3,4] (normalized)
```

```
    y.append(data_norm[i+window]) # next number (normalized)
```

```
X = np.array(X)[..., np.newaxis] # shape: (samples, timesteps, features=1)
```

```
y = np.array(y)
```

```
print("X shape:", X.shape) # (samples, 5, 1)
```

5) Train / test split

```
split = int(0.8 * len(X))
```

```
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

# 6) Build a tiny RNN
model = Sequential([
    SimpleRNN(32, activation="tanh", input_shape=(window, 1)),
    Dense(1)
])
```

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.005),
              loss="mse")
```

```
# 7) Train
```

```
history = model.fit(X_train, y_train,
                     epochs=300, batch_size=32,
                     verbose=0,
                     validation_data=(X_test, y_test))
```

```
# 8) Helper to predict next number for a given (unnormalized) sequence
```

```
def predict_next(seq):
    seq = np.asarray(seq, dtype=np.float32)
    assert len(seq) == window, f"Need {window} numbers"
    seq_norm = (seq / max_val).reshape(1, window, 1)
    pred_norm = model.predict(seq_norm, verbose=0)[0, 0]
    return pred_norm * max_val # de-normalize back
```

```
# 9) Demo predictions
```

```
tests = [
    [95, 96, 97, 98, 99],      # expect ~100
    [7, 8, 9, 10, 11],        # expect ~12
    [100, 101, 102, 103, 104] # expect ~105
```

]

for t in tests:

```
print(f"Input: {t} -> Predicted next: {round(predict_next(t))}")
```



Explanation of above application

- **Task:** Given the last **5 numbers** in a numeric sequence, predict the **next number**.
- **Data:** A simple increasing sequence $0, 1, 2, \dots, 199$.
- **Model:** A tiny **RNN regression** model: `SimpleRNN(32, tanh) → Dense(1)`.
- **Learning target:** Real value (next number), so we use **MSE** loss (regression).

Imports & Reproducibility

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

np.random.seed(42)
tf.random.set_seed(42)
  • Sets fixed seeds so training runs are repeatable (weights init & batching become deterministic).
```

Data: Create & Normalize

```
data = np.arange(0, 200, dtype=np.float32) # [0, 1, 2, ..., 199]
max_val = data.max() # 199.0
data_norm = data / max_val # scale to [0, 1]
  • We generate a monotonic sequence of length 200.

  • Normalization helps training (stabilizes gradients; keeps values in a compact range).
    Here: x_norm = x / 199 → all inputs/targets lie in  $[0, 1]$ .
```

Windowing (Supervised samples from a sequence)

```
window = 5 # timesteps
X, y = [], []
for i in range(len(data_norm) - window):
    X.append(data_norm[i:i+window]) # past 5 values
    y.append(data_norm[i+window]) # next value
X = np.array(X)[..., np.newaxis] # (samples, timesteps=5,
features=1)
y = np.array(y)
```

Why: RNNs expect **sequences of timesteps**. We slide a window of length 5 across the series:

- For positions $i = 0..194$:

- Input $X[i] = [x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}]$
- Target $y[i] = x_{i+5}$
- Shapes:
 - Number of samples = $200 - 5 = 195$
 - X shape = **(195, 5, 1)** → (samples, timesteps, features)
(feature dimension is 1 because each timestep is a single scalar)
 - y shape = **(195,)**

This turns a 1D series into a supervised learning dataset for **one-step-ahead forecasting**.

Train / Test Split

```
split = int(0.8 * len(X)) # 156
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]
```

- We respect **time order** by taking the **first 80%** as training, **last 20%** as test.
This mimics real forecasting: train on the past, test on later points.
- Result:
 - $X_train: (156, 5, 1), y_train: (156,)$
 - $X_test: (39, 5, 1), y_test: (39,)$

Model Architecture

```
model = Sequential([
    SimpleRNN(32, activation="tanh", input_shape=(window, 1)),
    Dense(1)
])
```

- **SimpleRNN(32, tanh):** Processes the 5 steps and outputs the final hidden state (size 32).
 - **tanh** keeps hidden values in **[-1, 1]**, which is common for vanilla RNNs.
- **Dense(1):** Maps the 32-D hidden state to **one real number** (the next value).

Parameter count

- For **SimpleRNN(U, input_dim=D)**, params = $U * (U + D + 1)$
Here $U=32, D=1 \rightarrow 32 * (32+1+1) = 32 * 34 = **1088**$
- For **Dense(1)** after a 32-D input: weights $32 * 1 = 32$ + bias 1 → **33**
- **Total = $1088 + 33 = 1121$ parameters**

Compile (Loss & Optimizer)

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.005)
,
    loss="mse")
```

- **Loss:** mse (mean squared error) — standard for regression.
- **Optimizer:** Adam with a slightly higher LR (0.005) to learn quickly on this simple task.

Train

```
history = model.fit(
    X_train, y_train,
    epochs=300, batch_size=32,
    verbose=0,
    validation_data=(X_test, y_test))
```

- **Epochs=300** on a very easy signal is fine; the model will quickly learn the near-linear mapping.
- We track **validation loss** to ensure it generalizes to later timesteps.

Prediction Helper (De-normalize output)

```
def predict_next(seq):
    seq = np.asarray(seq, dtype=np.float32)
    assert len(seq) == window, f"Need {window} numbers"
    seq_norm = (seq / max_val).reshape(1, window, 1)      # (1, 5, 1)
    pred_norm = model.predict(seq_norm, verbose=0)[0, 0]
    return pred_norm * max_val                            # back to
```

original scale

- **Input:** an **unnormalized** list of 5 numbers (e.g., [95, 96, 97, 98, 99]).
- We **normalize** to match training scale, predict a **normalized** next value, then **de-normalize**.
- Shapes at inference: (1, 5, 1) → model → (1, 1) → scalar.

Demo Predictions

```
tests = [
    [95, 96, 97, 98, 99],      # expect ~100
    [7, 8, 9, 10, 11],        # expect ~12
    [100, 101, 102, 103, 104] # expect ~105
```

]

```
for t in tests:
```

```
    print(f"Input: {t} -> Predicted next: {round(predict_next(t))} ")
```

- Because the underlying process is **almost linear** ($\text{next} \approx \text{previous} + 1$), a small RNN learns to output $\approx \text{last} + 1$.
- Rounded outputs should be close to 100, 12, 105 respectively.



Transformers

- A **deep learning architecture** introduced in 2017 by Google in the paper “*Attention is All You Need*”.
- Designed for **sequence data** (like text, speech, DNA).
- Became the backbone of **NLP (Natural Language Processing)** and now extends to **vision, speech, and multimodal AI**.

Transformers replaced older sequence models like **RNNs** and **LSTMs** because they are faster, more scalable, and capture long-range dependencies better.



Attention Is All You Need

Ashish Vaswani*
 Google Brain
avaswani@google.com

Noam Shazeer*
 Google Brain
noam@google.com

Niki Parmar*
 Google Research
nikip@google.com

Jakob Uszkoreit*
 Google Research
usz@google.com

Llion Jones*
 Google Research
llion@google.com

Aidan N. Gomez* [†]
 University of Toronto
aidan@cs.toronto.edu

Lukasz Kaiser*
 Google Brain
lukasz Kaiser@google.com

Ilia Polosukhin* [‡]
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

1 Introduction

Recurrent neural networks, long short-term memory [13] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and

^{*}Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

[†]Work performed while at Google Brain.

[‡]Work performed while at Google Research.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

Piyush Khaire - 7588945488

आम्ही Technical संस्कार करतो !!!

©Marvellous Infosystems

Page 1

arXiv:1706.03762v5 [cs.CL] 6 Dec 2017

Transformer Architecture

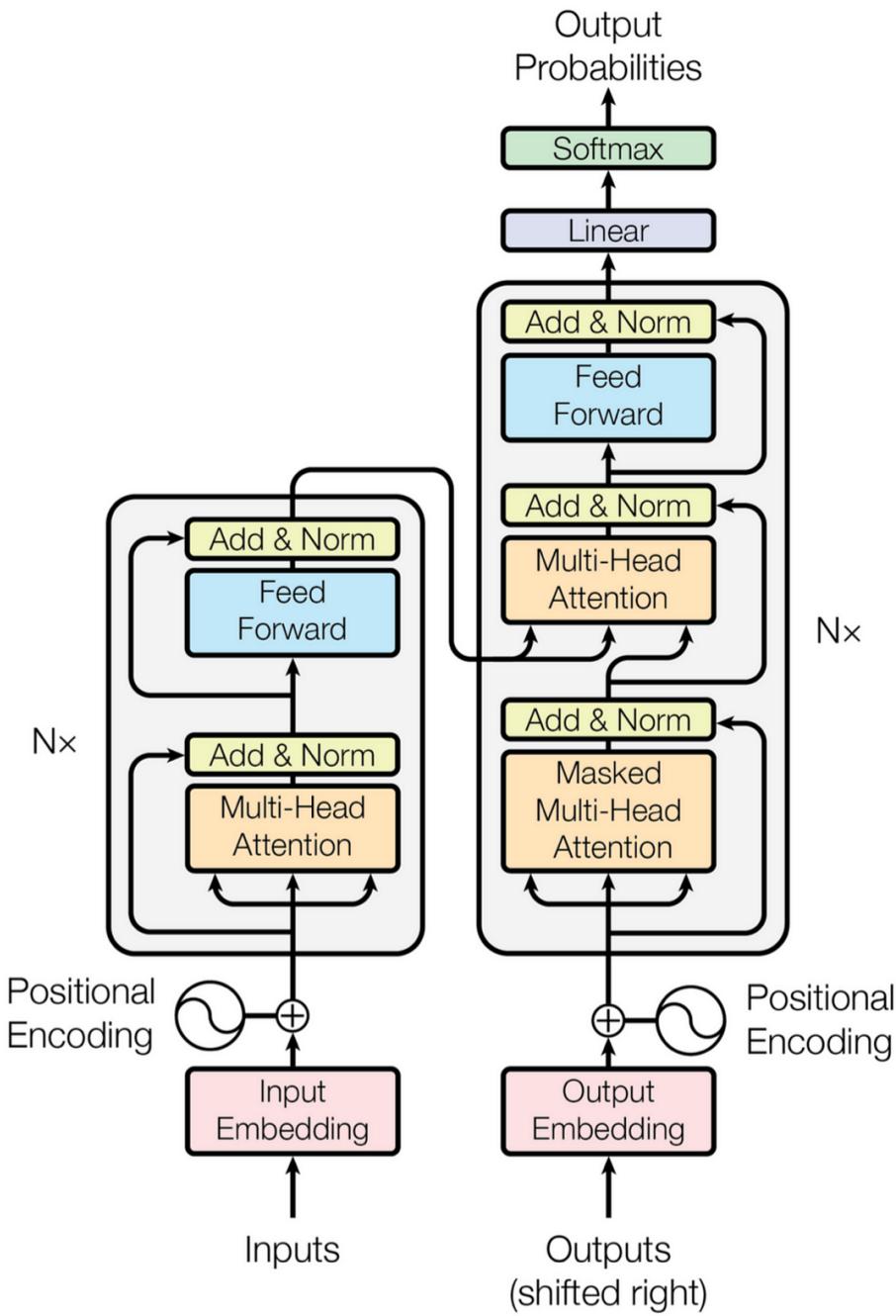


Figure 1: The Transformer - model architecture.

1. Input Embeddings

- Neural networks can't understand text directly. They need **numbers**.
- So each word (or sub-word/token) is converted into a **vector** (embedding).

Example:

Sentence: “*The cat sat*”

- "The" → [0.12, -0.98, 0.45, ...]
- "cat" → [0.67, 0.03, -0.45, ...]
- "sat" → [-0.22, 0.91, 0.11, ...]

These vectors capture meaning — “cat” will be closer to “dog” than to “banana”.

2. Positional Encoding

- Unlike RNNs, transformers **don't know word order**.
- They only see a bag of vectors.
- To fix this, we add **positional encodings** (learned vectors).

Example:

Sentence: “*I love Python*”

- Without positions: could be shuffled → same meaning for model
- With positions:
 - “I” → Position 1
 - “love” → Position 2
 - “Python” → Position 3

Now the model knows “I love Python” ≠ “Python love I”.

3. Attention Mechanism

- Attention helps the model decide **which words to focus on** when processing each word.

Example:

Sentence: “*The animal didn't cross the street because it was too wide.*”

- What does “**it**” refer to?
- Attention mechanism finds “**street**” is more related to “it” than “animal”.

This is how Transformers solve ambiguity.

4. Self-Attention

- Each word looks at **all other words** in the same sentence and updates its meaning.

Example:

Sentence: “*She saw a jaguar.*”

- Without self-attention: “jaguar” is unclear (car? animal?).
- With self-attention:
 - If sentence continues: “*She saw a jaguar in the zoo*” → attention links “jaguar” with “zoo” → animal.
 - If sentence continues: “*She saw a jaguar on the highway*” → attention links “jaguar” with “highway” → car.

Context decides meaning.

5. Multi-Head Attention

- Instead of one attention view, we use **multiple heads**.
- Each head learns a different relationship.

Example:

Sentence: “*The cat sat on the mat.*”

- Head 1: Focuses on **subject-object** (“cat” ↔ “sat”)
- Head 2: Focuses on **preposition** (“sat” ↔ “on”)
- Head 3: Focuses on **location** (“on” ↔ “mat”)

Like multiple students highlighting different parts of a sentence, then combining notes.

6. Feed-Forward Network

- After attention, each word’s vector passes through a **small neural network** (same for every word).
- This refines the representation.

Think of it as polishing the understanding of each word after looking at the context.

7. Layer Normalization + Residual Connections

- To make training stable and efficient:
 - **Residual connections:** Skip connections so information flows easily.
 - **Layer norm:** Normalizes values for faster learning.

Prevents the model from “forgetting” original word meaning after transformations.

8. Stacking Layers

- One layer captures **basic context**.
- Multiple layers stacked = **deep understanding**.

Example:

Sentence: “*The bank approved the loan.*”

- Lower layers: figure out simple word meanings.
- Higher layers: decide **bank = financial institution**, not river bank.

9. Encoder vs Decoder

- **Encoder:** Only reads and understands input. (Used in BERT)
- **Decoder:** Generates words one by one (used in GPT).
- **Encoder–Decoder:** Used for translation (input English → output French).

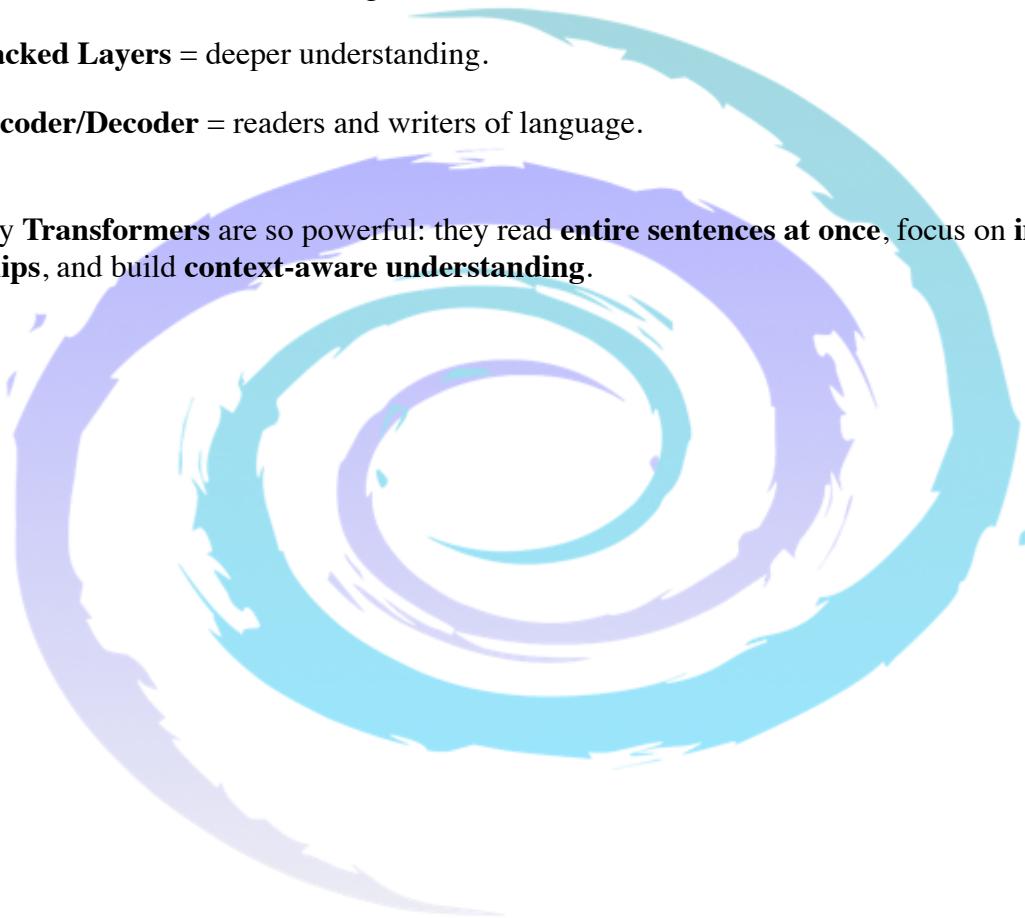
Example (Translation):

- Encoder: “I love dogs” → creates context representation.
- Decoder: “J’aime les chiens” → generates French translation.

Summary

- **Embeddings** = numbers for words.
- **Positional Encoding** = word order.
- **Attention** = what matters most.
- **Self-Attention** = words looking at each other.
- **Multi-Head Attention** = many perspectives at once.
- **Feed Forward** = refine meaning.
- **Stacked Layers** = deeper understanding.
- **Encoder/Decoder** = readers and writers of language.

This is why **Transformers** are so powerful: they read **entire sentences at once**, focus on **important relationships**, and build **context-aware understanding**.



Clustering using K-Mean algorithm

K-means clustering is a clustering algorithm that aims to partition n observations into k clusters.

There are 3 steps:

Step 1:

Initialisation – K initial “means” (centroids) are generated at random

Step 2:

Assignment – K clusters are created by associating each observation with the nearest centroid

Step 3:

Update – The centroid of the clusters becomes the new mean

Assignment and Update are repeated iteratively until convergence

The end result is that the sum of squared errors is minimised between points and their respective centroids.

```
#-----
import copy
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
#-----

df = pd.DataFrame({
    'x': [12, 20, 28, 18, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53, 55, 61,
64, 69, 72],
    'y': [39, 36, 30, 52, 54, 46, 55, 59, 63, 70, 66, 63, 58, 23, 14, 8, 19,
7, 24]
})

print("Step 1: Initialisation – K initial “means” (centroids) are generated at random");

print("-----");
print("Data set for training");
print("-----");
print(df);
print("-----");
np.random.seed(200)
k = 3
# centroids[i] = [x, y]
```

```

centroids = {
    i+1: [np.random.randint(0, 80), np.random.randint(0, 80)]
    for i in range(k)
}
print("-----");
print("Random centroid generated");
print(centroids);
print("-----");

fig = plt.figure(figsize=(5, 5))
plt.scatter(df['x'], df['y'], color='k')

colmap = {1: 'r', 2: 'g', 3: 'b'}
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])

plt.title("Marvellous : Dataset with random centroid");
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.show()

#-----
# Assignment – K clusters are created by associating each observation with the
# nearest centroid

def assignment(df, centroids):

    for i in centroids.keys():
        # sqrt((x1 - x2)^2 - (y1 - y2)^2)
        df['distance_from_{}'.format(i)] = (
            np.sqrt(
                (df['x'] - centroids[i][0]) ** 2
                + (df['y'] - centroids[i][1]) ** 2
            )
        )

    centroid_distance_cols = ['distance_from_{}'.format(i) for i in
    centroids.keys()]

    df['closest'] = df.loc[:, centroid_distance_cols].idxmin(axis=1)

    df['closest'] = df['closest'].map(lambda x: int(x.strip('distance_from_')))

    df['color'] = df['closest'].map(lambda x: colmap[x])
    return df
  
```

```

print("Step 2 : Assignment - K clusters are created by associating each
observation with the nearest centroid");

print("Before assignment dataset");
print(df)
df = assignment(df, centroids)

print("First centroid : Red");
print("Second centroid : Green");
print("Third centroid : Blue");

print("After assignment dataset");
print(df)

fig = plt.figure(figsize=(5, 5))
plt.scatter(df['x'], df['y'], color=df['color'], alpha=0.5, edgecolor='k')
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.title("Marvellous : Dataset with clustering & random centroid");
plt.show()

# -----
old_centroids = copy.deepcopy(centroids)
print("Step 3:Update - The centroid of the clusters becomes the new mean
Assignment and Update are repeated iteratively until convergence");

def update(k):
    print("Old values of centroids");
    print(k);

    for i in centroids.keys():
        centroids[i][0] = np.mean(df[df['closest'] == i]['x'])
        centroids[i][1] = np.mean(df[df['closest'] == i]['y'])

    print("New values of centroids");
    print(k);
    return k

centroids = update(centroids)

fig = plt.figure(figsize=(5, 5))
ax = plt.axes()
plt.scatter(df['x'], df['y'], color=df['color'], alpha=0.5, edgecolor='k')
for i in centroids.keys():

```

```

plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)

for i in old_centroids.keys():
    old_x = old_centroids[i][0]
    old_y = old_centroids[i][1]
    dx = (centroids[i][0] - old_centroids[i][0]) * 0.75
    dy = (centroids[i][1] - old_centroids[i][1]) * 0.75
    ax.arrow(old_x, old_y, dx, dy, head_width=2, head_length=3, fc=colmap[i],
ec=colmap[i])

plt.title("Marvellous : Dataset with clustering and updated centroids");
plt.show()

#-----
## Repeat Assignment Stage
print("Before assignment dataset");
print(df)
df = assignment(df, centroids)
print("After assignment dataset");
print(df)

# Plot results
fig = plt.figure(figsize=(5, 5))
plt.scatter(df['x'], df['y'], color=df['color'], alpha=0.5, edgecolor='k')
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.title("Marvellous : Dataset with clustering and updated centroids");
plt.show()

# Continue until all assigned categories don't change any more
while True:
    closest_centroids = df['closest'].copy(deep=True)
    centroids = update(centroids)
    print("Before assignment dataset");
    print(df)
    df = assignment(df, centroids)
    print("After assignment dataset");
    print(df)
    if closest_centroids.equals(df['closest']):
        break

print("Final values of centroids");
print(centroids);

```

```
fig = plt.figure(figsize=(5, 5))
plt.scatter(df['x'], df['y'], color=df['color'], alpha=0.5, edgecolor='k')
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.title("Marvellous : Final dataset with set centroids");
plt.show()
```



Output of above application :

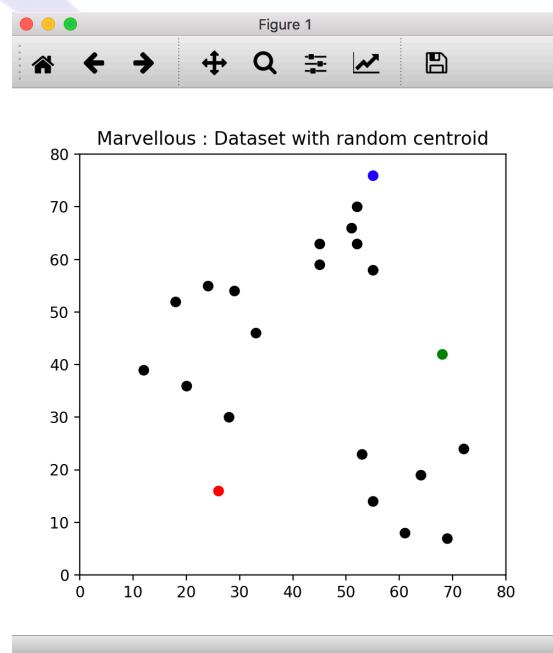
Step 1: Initialisation – K initial “means” (centroids) are generated at random

Data set for training

	x	y
0	12	39
1	20	36
2	28	30
3	18	52
4	29	54
5	33	46
6	24	55
7	45	59
8	45	63
9	52	70
10	51	66
11	52	63
12	55	58
13	53	23
14	55	14
15	61	8
16	64	19
17	69	7
18	72	24

Random centroid generated

{1: [26, 16], 2: [68, 42], 3: [55, 76]}



Step 2 : Assignment – K clusters are created by associating each observation with the nearest centroid

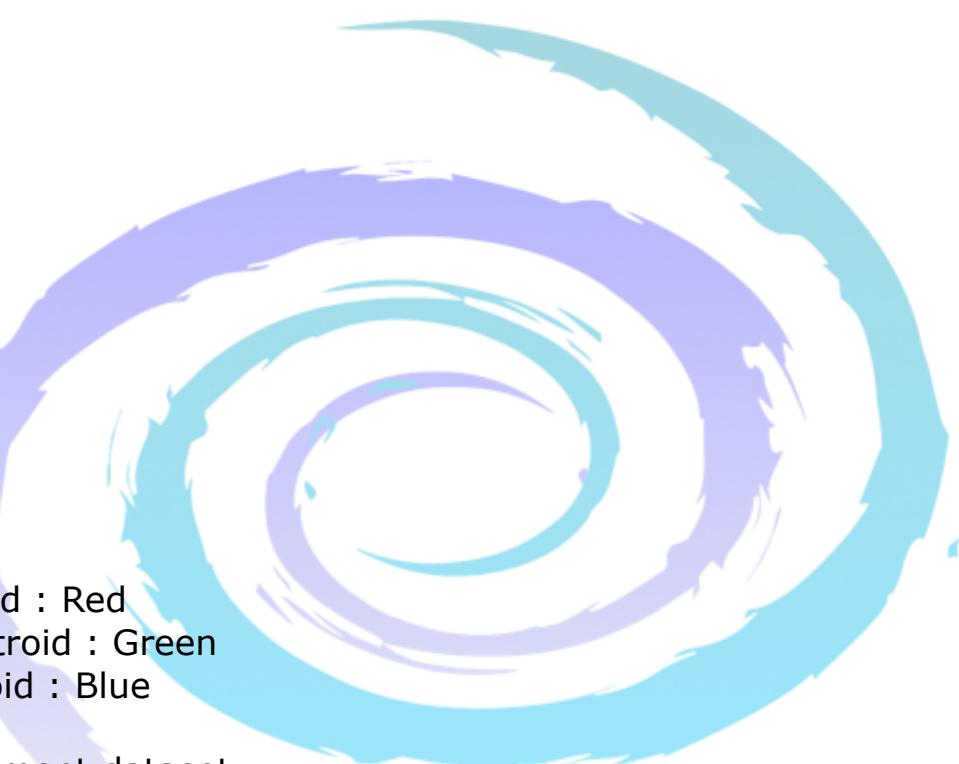
Before assignment dataset

	x	y
0	12	39
1	20	36
2	28	30
3	18	52
4	29	54
5	33	46
6	24	55
7	45	59
8	45	63
9	52	70
10	51	66
11	52	63
12	55	58
13	53	23
14	55	14
15	61	8
16	64	19
17	69	7
18	72	24

First centroid : Red

Second centroid : Green

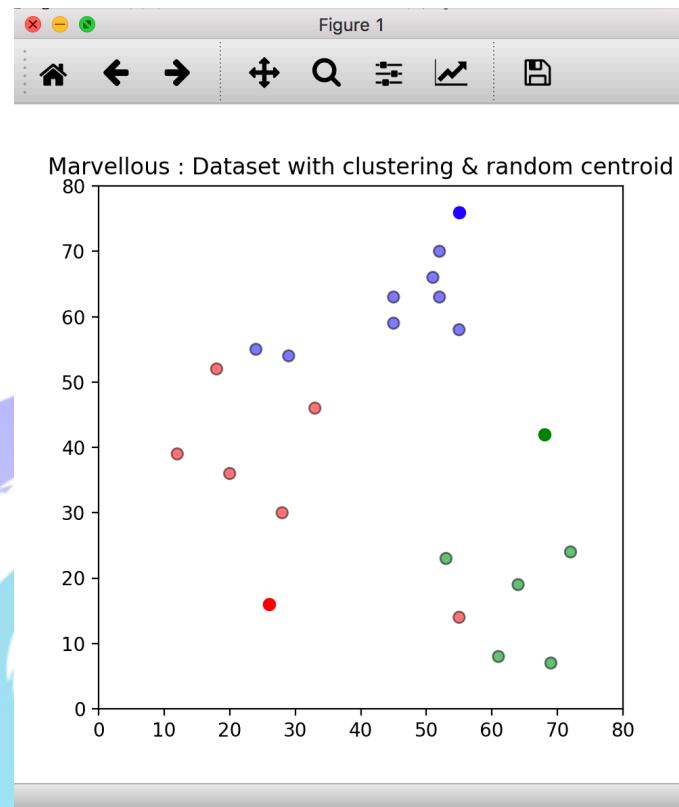
Third centroid : Blue



After assignment dataset

	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	26.925824	56.080300	56.727418	1	r
1	20	36	20.880613	48.373546	53.150729	1	r
2	28	30	14.142136	41.761226	53.338541	1	r
3	18	52	36.878178	50.990195	44.102154	1	r
4	29	54	38.118237	40.804412	34.058773	3	b
5	33	46	30.805844	35.227830	37.202150	1	r
6	24	55	39.051248	45.880279	37.443290	3	b
7	45	59	47.010637	28.600699	19.723083	3	b
8	45	63	50.695167	31.144823	16.401219	3	b
9	52	70	59.933296	32.249031	6.708204	3	b
10	51	66	55.901699	29.410882	10.770330	3	b
11	52	63	53.712196	26.400758	13.341664	3	b
12	55	58	51.039201	20.615528	18.000000	3	b
13	53	23	27.892651	24.207437	53.037722	2	g
14	55	14	29.068884	30.870698	62.000000	1	r
15	61	8	35.902646	34.713110	68.264193	2	g

16 64 19	38.118237	23.345235	57.706152	2	g
17 69 7	43.931765	35.014283	70.405966	2	g
18 72 24	46.690470	18.439089	54.708317	2	g



Step 3: Update – The centroid of the clusters becomes the new mean Assignment and Update are repeated iteratively until convergence

Old values of centroids

{1: [26, 16], 2: [68, 42], 3: [55, 76]}

New values of centroids

{1: [27.666666666666668, 36.166666666666664], 2: [63.8, 16.2], 3: [44.125, 61.0]}

Note : New centroids are mean of the generated X and Y coordinates of clustering members. We can use additions of distances to decide the error rate.

Example :

X for centroid 1 : $(12+20+28+18+55) / 6 = 27.66$

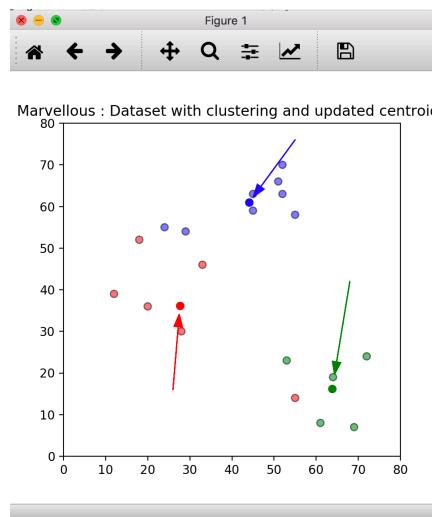
Y for centroid 2 : $(39+36+30+52+14) / 6 = 36.16$

Before assignment dataset

	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	26.925824	56.080300	56.727418	1	r
1	20	36	20.880613	48.373546	53.150729	1	r
2	28	30	14.142136	41.761226	53.338541	1	r
3	18	52	36.878178	50.990195	44.102154	1	r
4	29	54	38.118237	40.804412	34.058773	3	b
5	33	46	30.805844	35.227830	37.202150	1	r
6	24	55	39.051248	45.880279	37.443290	3	b
7	45	59	47.010637	28.600699	19.723083	3	b
8	45	63	50.695167	31.144823	16.401219	3	b
9	52	70	59.933296	32.249031	6.708204	3	b
10	51	66	55.901699	29.410882	10.770330	3	b
11	52	63	53.712196	26.400758	13.341664	3	b
12	55	58	51.039201	20.615528	18.000000	3	b
13	53	23	27.892651	24.207437	53.037722	2	g
14	55	14	29.068884	30.870698	62.000000	1	r
15	61	8	35.902646	34.713110	68.264193	2	g
16	64	19	38.118237	23.345235	57.706152	2	g
17	69	7	43.931765	35.014283	70.405966	2	g
18	72	24	46.690470	18.439089	54.708317	2	g

After assignment dataset

	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	15.920811	56.595760	38.936045	1	r
1	20	36	7.668478	48.067453	34.742130	1	r
2	28	30	6.175669	38.367695	34.943034	1	r
3	18	52	18.550981	58.131575	27.631786	1	r
4	29	54	17.883108	51.379763	16.666302	3	b
5	33	46	11.186549	42.856505	18.675268	1	r
6	24	55	19.186946	55.583091	21.000372	1	r
7	45	59	28.667151	46.746979	2.183031	3	b
8	45	63	31.944831	50.434909	2.183031	3	b
9	52	70	41.674999	55.078853	11.958914	3	b
10	51	66	37.874427	51.418674	8.500919	3	b
11	52	63	36.223458	48.264687	8.125000	3	b
12	55	58	34.982932	42.716273	11.281207	3	b
13	53	23	28.550637	12.762445	39.022630	2	g
14	55	14	35.191934	9.070832	48.241742	2	g
15	61	8	43.640259	8.664872	55.621629	2	g
16	64	19	40.184643	2.807134	46.465209	2	g
17	69	7	50.587932	10.567876	59.453895	2	g
18	72	24	45.972516	11.317243	46.325108	2	g



Old values of centroids

```
{1: [27.666666666666668, 36.166666666666664], 2: [63.8, 16.2], 3: [44.125, 61.0]}
```

New values of centroids

```
{1: [22.5, 43.0], 2: [62.333333333333336, 15.833333333333334], 3: [47.0, 61.857142857142854]}
```

Before assignment dataset

	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	15.920811	56.595760	38.936045	1	r
1	20	36	7.668478	48.067453	34.742130	1	r
2	28	30	6.175669	38.367695	34.943034	1	r
3	18	52	18.550981	58.131575	27.631786	1	r
4	29	54	17.883108	51.379763	16.666302	3	b
5	33	46	11.186549	42.856505	18.675268	1	r
6	24	55	19.186946	55.583091	21.000372	1	r
7	45	59	28.667151	46.746979	2.183031	3	b
8	45	63	31.944831	50.434909	2.183031	3	b
9	52	70	41.674999	55.078853	11.958914	3	b
10	51	66	37.874427	51.418674	8.500919	3	b
11	52	63	36.223458	48.264687	8.125000	3	b
12	55	58	34.982932	42.716273	11.281207	3	b
13	53	23	28.550637	12.762445	39.022630	2	g
14	55	14	35.191934	9.070832	48.241742	2	g
15	61	8	43.640259	8.664872	55.621629	2	g
16	64	19	40.184643	2.807134	46.465209	2	g
17	69	7	50.587932	10.567876	59.453895	2	g
18	72	24	45.972516	11.317243	46.325108	2	g

After assignment dataset

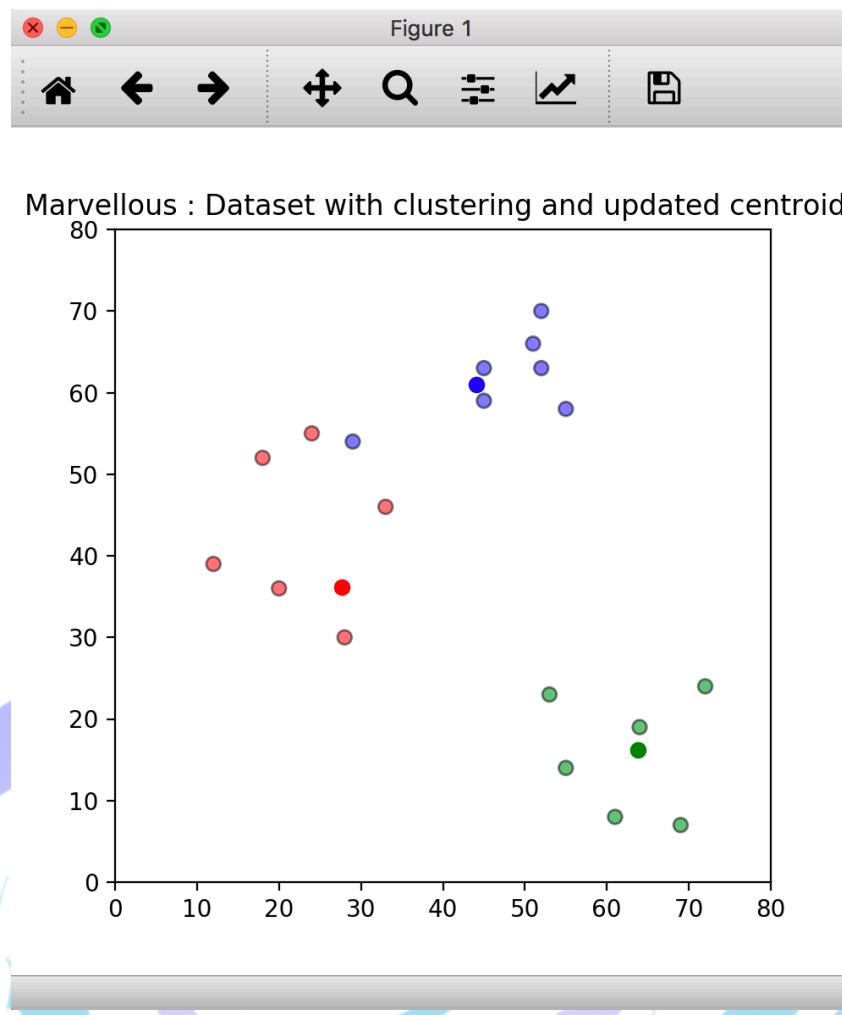
	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	11.236103	55.408834	41.802500	1	r
1	20	36	7.433034	46.891423	37.384380	1	r
2	28	30	14.115594	37.141247	37.092823	1	r
3	18	52	10.062306	57.214266	30.629451	1	r
4	29	54	12.776932	50.673519	19.640130	1	r
5	33	46	10.920165	42.076980	21.152990	1	r
6	24	55	12.093387	54.803943	24.000425	1	r
7	45	59	27.608875	46.516723	3.487587	3	b
8	45	63	30.103986	50.250760	2.303502	3	b
9	52	70	39.990624	55.143500	9.555424	3	b
10	51	66	36.623080	51.430914	5.758756	3	b
11	52	63	35.640567	48.285321	5.128949	3	b
12	55	58	35.794553	42.799598	8.881303	3	b
13	53	23	36.472592	11.767422	39.317649	2	g
14	55	14	43.557433	7.559027	48.521193	2	g
15	61	8	52.031241	7.945998	55.647029	2	g
16	64	19	47.940067	3.578485	46.105690	2	g
17	69	7	58.806887	11.066717	59.104197	2	g
18	72	24	53.021222	12.654600	45.366984	2	g

Old values of centroids

{1: [22.5, 43.0], 2: [62.33333333333336, 15.83333333333334], 3: [47.0, 61.857142857142854]}

New values of centroids

{1: [23.428571428571427, 44.57142857142857], 2: [62.33333333333336, 15.83333333333334], 3: [50.0, 63.16666666666664]}



Before assignment dataset

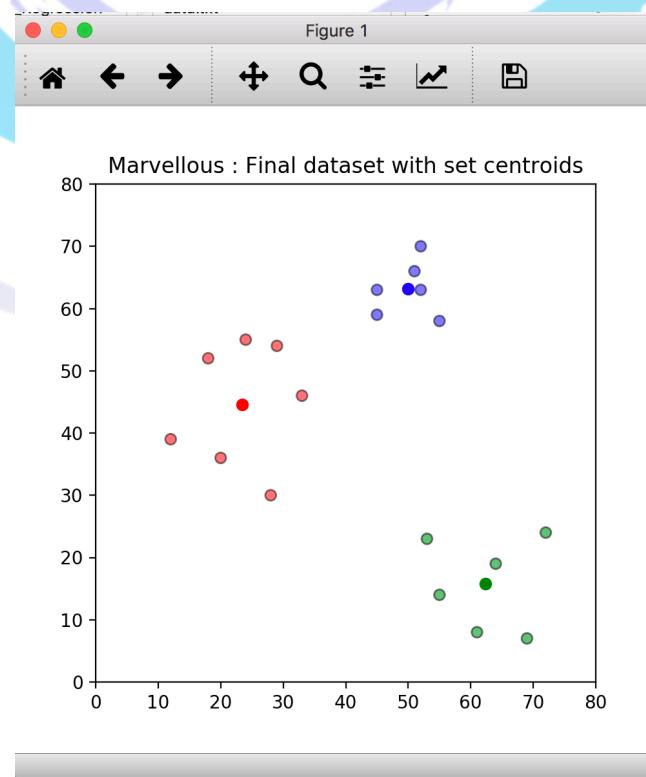
	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	11.236103	55.408834	41.802500	1	r
1	20	36	7.433034	46.891423	37.384380	1	r
2	28	30	14.115594	37.141247	37.092823	1	r
3	18	52	10.062306	57.214266	30.629451	1	r
4	29	54	12.776932	50.673519	19.640130	1	r
5	33	46	10.920165	42.076980	21.152990	1	r
6	24	55	12.093387	54.803943	24.000425	1	r
7	45	59	27.608875	46.516723	3.487587	3	b
8	45	63	30.103986	50.250760	2.303502	3	b
9	52	70	39.990624	55.143500	9.555424	3	b
10	51	66	36.623080	51.430914	5.758756	3	b
11	52	63	35.640567	48.285321	5.128949	3	b
12	55	58	35.794553	42.799598	8.881303	3	b
13	53	23	36.472592	11.767422	39.317649	2	g
14	55	14	43.557433	7.559027	48.521193	2	g
15	61	8	52.031241	7.945998	55.647029	2	g
16	64	19	47.940067	3.578485	46.105690	2	g
17	69	7	58.806887	11.066717	59.104197	2	g
18	72	24	53.021222	12.654600	45.366984	2	g

After assignment dataset

	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	12.714286	55.408834	45.033629	1	r
1	20	36	9.231711	46.891423	40.472556	1	r
2	28	30	15.271689	37.141247	39.799846	1	r
3	18	52	9.200710	57.214266	33.892395	1	r
4	29	54	10.951656	50.673519	22.913485	1	r
5	33	46	9.677451	42.076980	24.159769	1	r
6	24	55	10.444215	54.803943	27.252421	1	r
7	45	59	25.952075	46.516723	6.508541	3	b
8	45	63	28.371443	50.250760	5.002777	3	b
9	52	70	38.248383	55.143500	7.120003	3	b
10	51	66	34.919441	51.430914	3.004626	3	b
11	52	63	33.999100	48.285321	2.006932	3	b
12	55	58	34.308623	42.799598	7.189885	3	b
13	53	23	36.603223	11.767422	40.278544	2	g
14	55	14	43.947325	7.559027	49.420250	2	g
15	61	8	52.431685	7.945998	56.252654	2	g
16	64	19	47.957677	3.578485	46.332434	2	g
17	69	7	59.062402	11.066717	59.293292	2	g
18	72	24	52.748150	12.654600	44.922464	2	g

Final values of centroids

```
{1: [23.428571428571427, 44.57142857142857], 2: [62.333333333333336, 15.833333333333334], 3: [50.0, 63.166666666666664]}
```



Clustering using K-Mean algorithm

In this case study we are using Iris dataset with K-Mean algorithm from sklearn.

```
#importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

#importing the Iris dataset with pandas
dataset = pd.read_csv('iris.csv')
x = dataset.iloc[:, [0, 1, 2, 3]].values

#Finding the optimum number of clusters for k-means classification
from sklearn.cluster import KMeans
wcss = []

for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)

#Plotting the results onto a line graph, allowing us to observe 'The elbow'
plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS') #within cluster sum of squares
plt.show()

#Applying kmeans to the dataset / Creating the kmeans classifier
kmeans = KMeans(n_clusters = 3, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 0)
y_kmeans = kmeans.fit_predict(x)

#Visualising the clusters
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Iris-setosa')

plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Iris-versicolour')

plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Iris-virginica')

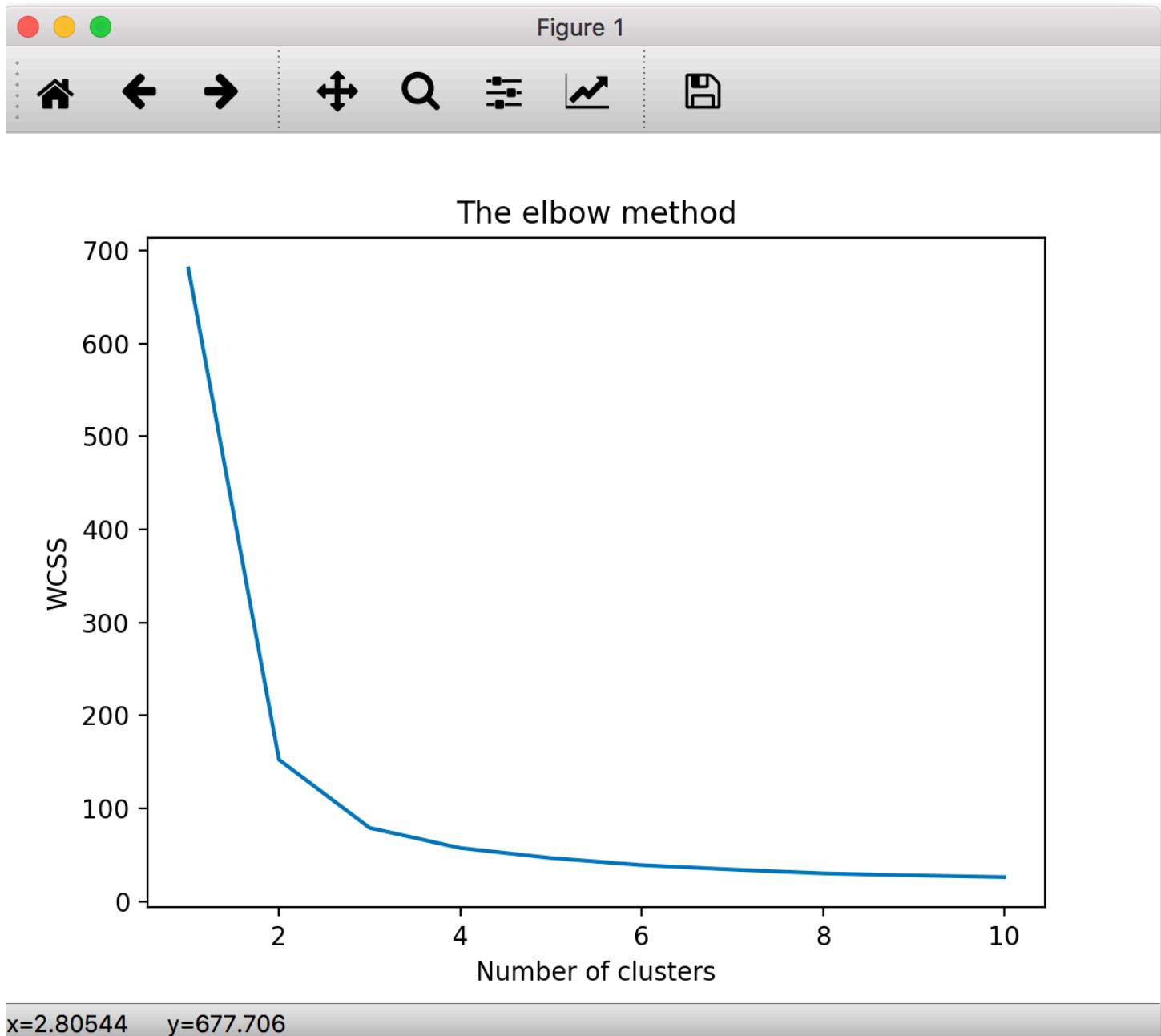
#Plotting the centroids of the clusters
```

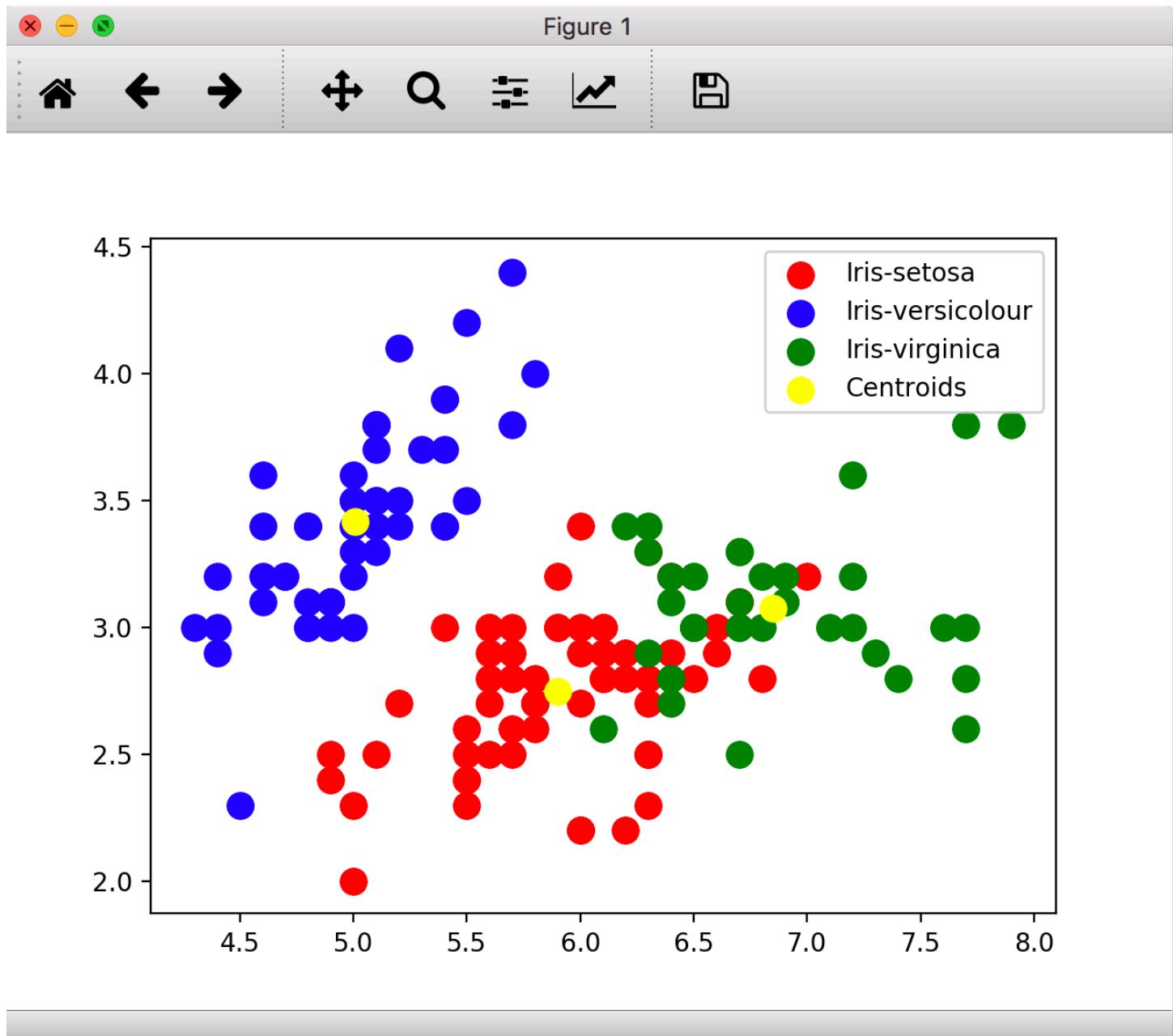
```
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,1], s = 100,  
c = 'yellow', label = 'Centroids')
```

```
plt.legend()
```

```
plt.show()
```

Output of above application :





Clustering using K-Mean algorithm

In this case study we are generating the data set at run time randomly and apply user defined K-Mean algorithm.

```

import numpy as np
import pandas as pd
from copy import deepcopy
from matplotlib import pyplot as plt

def MarvellousKMean():
    print("____");
    # Set three centers, the model should predict similar results
    center_1 = np.array([1,1])
    print(center_1)
    print("____");
    center_2 = np.array([5,5])
    print(center_2)
    print("____");
    center_3 = np.array([8,1])
    print(center_3)
    print("____");
    # Generate random data and center it to the three centers
    data_1 = np.random.randn(7, 2) + center_1
    print("Elements of first cluster with size"+str(len(data_1)))
    print(data_1)
    print("____");
    data_2 = np.random.randn(7,2) + center_2
    print("Elements of second cluster with size"+str(len(data_2)))
    print(data_2)
    print("____");
    data_3 = np.random.randn(7,2) + center_3
    print("Elements of third cluster with size"+str(len(data_3)))
    print(data_3)
    print("____");
    data = np.concatenate((data_1, data_2, data_3), axis = 0)
    print("Size of complete data set"+str(len(data)))
    print(data);
    print("____");
    plt.scatter(data[:,0], data[:,1], s=7)
    plt.title('Marvellous Infosystems : Input Dataset')
    plt.show()
    print("____");
    # Number of clusters
    k = 3

    # Number of training data
  
```

```

n = data.shape[0]
print("Total number of elements are",n)
print("_____");
# Number of features in the data
c = data.shape[1]
print("Total number of features are",c)
print("_____");

# Generate random centers, here we use sigma and mean to ensure it
represent the whole data
mean = np.mean(data, axis = 0)
print("Value of mean",mean)
print("_____");
# Calculate standard deviation
std = np.std(data, axis = 0)
print("Value of std",std)
print("_____");
centers = np.random.randn(k,c)*std + mean
print("Random points are",centers)
print("_____");

# Plot the data and the centers generated as random
plt.scatter(data[:,0], data[:,1], c='r', s=7)
plt.scatter(centers[:,0], centers[:,1], marker='*', c='g', s=150)
plt.title('Marvellous Infosystems : Input Dataset with random centroid *')
plt.show()
print("_____");

centers_old = np.zeros(centers.shape) # to store old centers
centers_new = deepcopy(centers) # Store new centers

print("Values of old centroids")
print(centers_old)
print("_____");

print("Values of new centroids")
print(centers_new)
print("_____");

data.shape
clusters = np.zeros(n)
distances = np.zeros((n,k))

print("Initial distances are")
print(distances)
print("_____");

error = np.linalg.norm(centers_new - centers_old)
print("value of error is ",error);
# When, after an update, the estimate of that center stays the same, exit loop
  
```

```

while error != 0:
    print("value of error is ",error);
    # Measure the distance to every center
    print("Measure the distance to every center")
    for i in range(k):
        print("Iteration number ",i)
        distances[:,i] = np.linalg.norm(data - centers[i], axis=1)

    # Assign all training data to closest center
    clusters = np.argmin(distances, axis = 1)

    centers_old = deepcopy(centers_new)

    # Calculate mean for every cluster and update the center
    for i in range(k):
        centers_new[i] = np.mean(data[clusters == i], axis=0)
    error = np.linalg.norm(centers_new - centers_old)

# end of while
centers_new

# Plot the data and the centers generated as random
plt.scatter(data[:,0], data[:,1], s=7)
plt.scatter(centers_new[:,0], centers_new[:,1], marker='*', c='g', s=150)
plt.title('Marvellous Infosystems : Final data with Centroid')
plt.show()

def main():
    print("---- Marvellous Infosystems by Piyush Khairnar----")
    print("Unsuervised Machine Learning")
    print("Clustering using K Mean Algorithm")
    MarvellousKMean()

if __name__ == "__main__":
    main()
  
```

Output of above application :

---- Marvellous Infosystems by Piyush Khairnar---

Unsuervised Machine Learning
Clustering using K Mean Algorithm

[1 1]

[5 5]

[8 1]

Elements of first cluster with size7

```
[[ 0.76455047  0.1057342 ]
 [ 0.57413358  1.07954625]
 [-0.14432049 -0.52217058]
 [ 1.31367866  1.05880002]
 [ 0.85572402  1.87967134]
 [ 1.28881943  0.68895633]
 [ 4.12989909  1.80041537]]
```

Elements of second cluster with size7

```
[[5.34096455 4.5997487 ]
 [4.06005167 6.22535108]
 [5.42174515 4.17713132]
 [3.36119862 3.88051506]
 [4.4704562 4.57687409]
 [5.37072318 4.87360285]
 [5.76651027 5.92478869]]
```

Elements of third cluster with size7

```
[[ 8.29482744  1.36251521]
 [ 8.20703246  1.29568945]
 [ 8.06619888 -0.50865465]
 [ 8.94870609  3.58106472]
 [ 8.25722023 -0.9248197 ]
 [ 7.05976628  0.26199046]
 [ 7.88080164  1.17437363]]
```

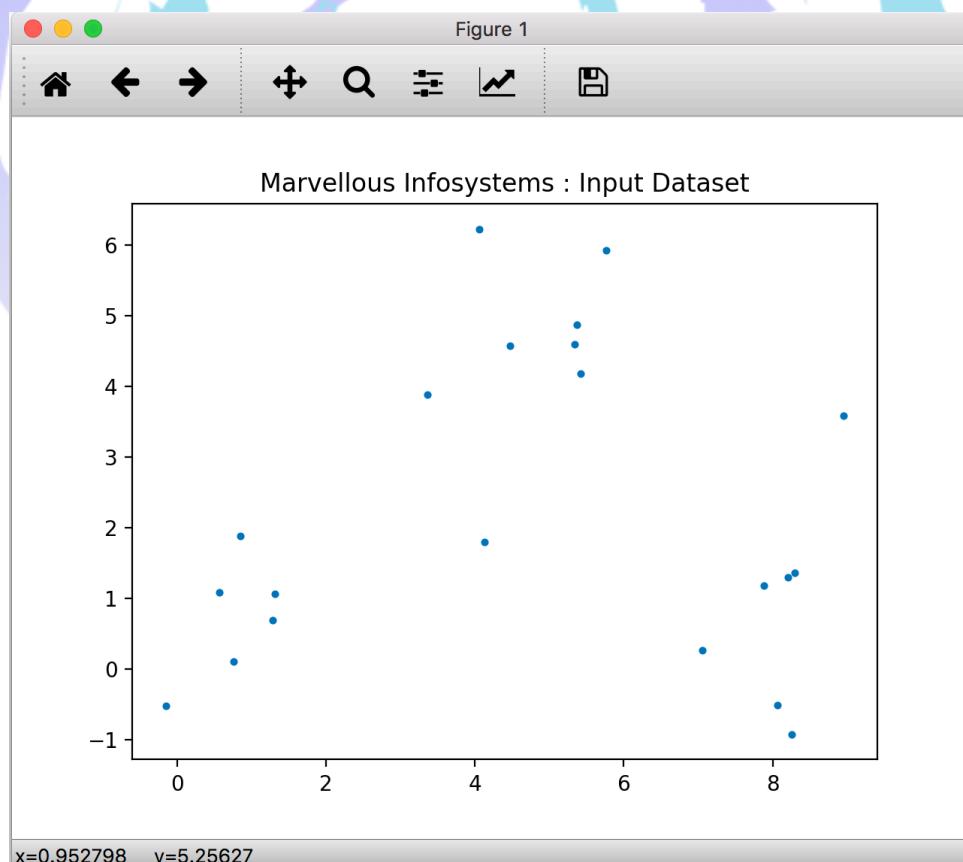
Size of complete data set21

```
[[ 0.76455047  0.1057342 ]
 [ 0.57413358  1.07954625]
 [-0.14432049 -0.52217058]
 [ 1.31367866  1.05880002]
 [ 0.85572402  1.87967134]
 [ 1.28881943  0.68895633]]
```

```
[ 4.12989909  1.80041537]
[ 5.34096455  4.5997487 ]
[ 4.06005167  6.22535108]
[ 5.42174515  4.17713132]
[ 3.36119862  3.88051506]
[ 4.4704562   4.57687409]
[ 5.37072318  4.87360285]
[ 5.76651027  5.92478869]
[ 8.29482744  1.36251521]
[ 8.20703246  1.29568945]
[ 8.06619888 -0.50865465]
[ 8.94870609  3.58106472]
[ 8.25722023 -0.9248197 ]
[ 7.05976628  0.26199046]
[ 7.88080164  1.17437363]]
```

Total number of elements are 21

Total number of features are 2



Value of mean [4.72803273 2.21862495]

Value of std [2.94300319 2.15577759]

Random points are [[9.68991885 2.78162963]

[6.17016879 1.5050397]

[5.29720489 0.21581884]]

Values of old centroids

[[0. 0.]

[0. 0.]

[0. 0.]]

Values of new centroids

[[9.68991885 2.78162963]

[6.17016879 1.5050397]

[5.29720489 0.21581884]]

Initial distances are

[[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

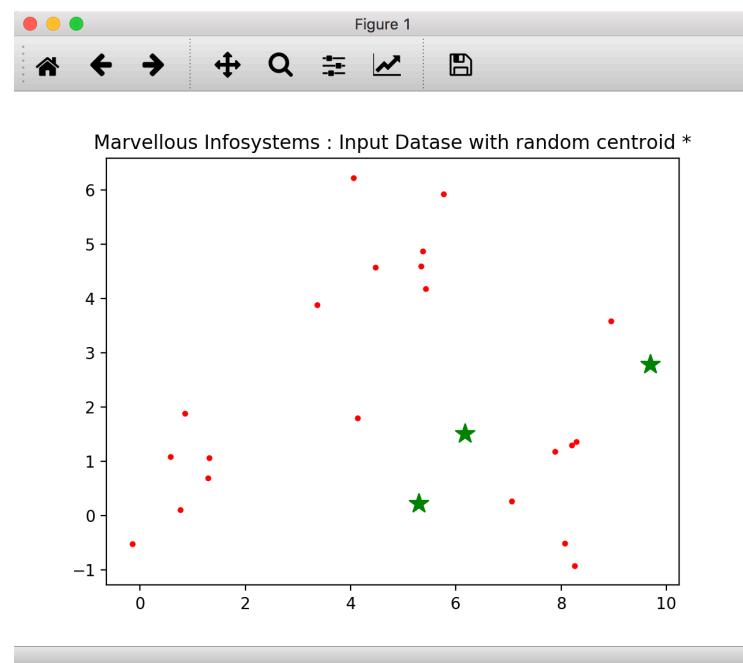
[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]]



value of error is 13.04128351259481
 value of error is 13.04128351259481

Measure the distance to every center

Iteration number 0

Iteration number 1

Iteration number 2

value of error is 3.847400069439621

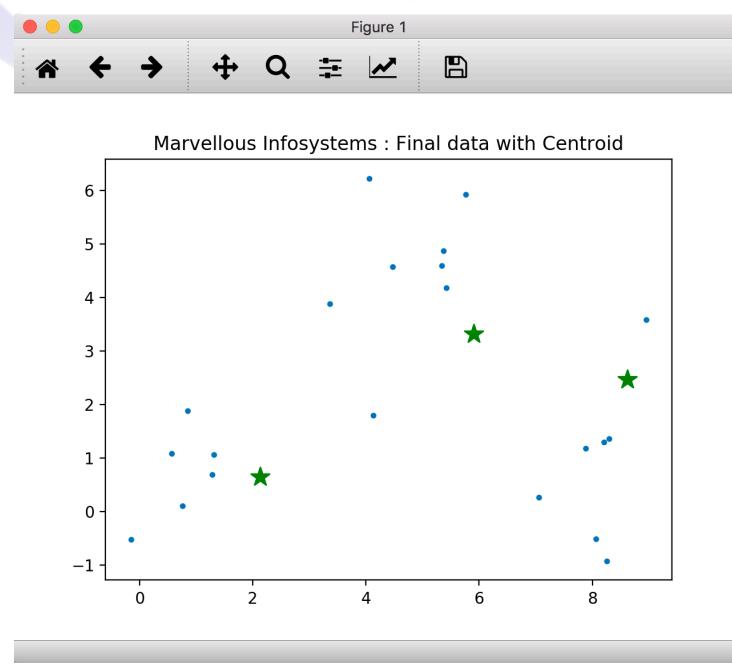
Measure the distance to every center

Iteration number 0

Iteration number 1

Iteration number

2



K-Nearest Neighbors (KNN)

KNN (K-Nearest Neighbors) is a simple, supervised machine learning algorithm used for both classification and regression tasks. However, it is more widely used for classification problems.

KNN works by:

- **Storing all available data** (no model training),
- And classifying a new data point based on **majority class of its K nearest neighbors** (for classification),
- Or **average of K nearest neighbor values** (for regression).

Step-by-Step Working of KNN:

1. **Choose the number K** (the number of neighbors to consider).
2. **Calculate the distance** between the new data point and all existing data points using a distance metric (typically Euclidean distance).
3. **Sort the distances** in ascending order and identify the K closest neighbors.
4. **Classify or Predict:**
 - For **classification**: Use **majority vote** among K nearest points.
 - For **regression**: Take the **mean (average)** of values of the K nearest points.

Mathematical Formula

◆ Euclidean Distance:

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

For n-dimensions:

$$\text{Distance} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Problem Statement:

We want to classify a new point P (3 , 3) using **K = 3** based on the following training data:

Point	Coordinates (x, y)	Class
A	(1, 2)	Red
B	(2, 3)	Red
C	(3, 1)	Blue
D	(6, 5)	Blue

Step 1: Calculate Euclidean Distance

The **Euclidean distance formula** is:

Calculate distances between point P (3 , 3) and each training point:

- A(1,2): $\sqrt{(3-1)^2 + (3-2)^2} = \sqrt{4 + 1} = \sqrt{5} \approx 2.24$
- B(2,3): $\sqrt{(3-2)^2 + (3-3)^2} = \sqrt{1 + 0} = \sqrt{1} = 1.00$
- C(3,1): $\sqrt{(3-3)^2 + (3-1)^2} = \sqrt{0 + 4} = \sqrt{4} = 2.00$
- D(6,5): $\sqrt{(3-6)^2 + (3-5)^2} = \sqrt{9 + 4} = \sqrt{13} \approx 3.61$

Step 2: Sort and Select K Nearest Neighbors

Sort the distances in ascending order and pick **K = 3** nearest:

1. B (1.00) → Red
2. C (2.00) → Blue
3. A (2.24) → Red

Step 3: Majority Voting

From top 3 neighbors:

- **Red:** 2 votes (B, A)
- **Blue:** 1 vote (C)

Final **Predicted Class** for point P(3,3): **Red**

Code Explanation

```

import numpy as np
import math

# Step 1: Euclidean distance function
def EucDistance(P1, P2):
    return math.sqrt((P1['x'] - P2['x'])**2 + (P1['y'] - P2['y'])**2)

# KNN Classification function
def MarvellousKNN():
    line = "-"*50

    # Step 0: Training data
    data = [
        {'point': 'A', 'x': 1, 'y': 2, 'label': 'Red'},
        {'point': 'B', 'x': 2, 'y': 3, 'label': 'Red'},
        {'point': 'C', 'x': 3, 'y': 1, 'label': 'Blue'},
        {'point': 'D', 'x': 6, 'y': 5, 'label': 'Blue'}
    ]
    print(line)
    print("Training data set:")
    for i in data:
        print(i)
    print(line)

    # New point to classify
    new_point = {'x': 3, 'y': 3}

    # Step 1: Calculate distances
    for d in data:
        d['distance'] = EucDistance(d, new_point)

    print("Calculated Distances:")
    for d in data:
        print(d)
    print(line)

    # Step 2: Sort by distance
    sorted_data = sorted(data, key=lambda item: item['distance'])

    print("Sorted Data by Distance:")
    for d in sorted_data:
        print(d)
    print(line)
  
```

```
# Step 3: Select top K = 3 neighbors
K = 3
k_nearest = sorted_data[:K]

# Step 4: Majority voting
red = sum(1 for i in k_nearest if i['label'] == 'Red')
blue = sum(1 for i in k_nearest if i['label'] == 'Blue')

print(f"Red votes = {red}, Blue votes = {blue}")
if red > blue:
    print("Predicted class = Red")
else:
    print("Predicted class = Blue")

# Call the function
MarvellousKNN()
```

