K MEANS

Customer	Age	Amount
C1	20	500
C2	40	1000
C3	30	800
C4	18	300
C5	28	1200
C6	35	1400
C7	45	1800

K-Means (K = 2)

Initial Centroids (Randomly chosen):

- **Z1 = C1 = (20, 500),(x1,y1)**
- **Z2 = C2 = (40, 1000), (x1,y1)**

Step 1: Assign to nearest centroid (Z1 = C1, Z2 = C2)

Formula:

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

Customer	Coordinates	Dist to Z1 (20,500)	Dist to Z2 (40,1000)	Cluster
C1	(20,500)	0	$\sqrt{(20^2 + 500^2)} = 500.40$	Z1
C2	(40,1000)	500.40	0	Z2
C3	(30,800)	$\sqrt{(10^2 + 300^2)} = 300.17$	$\sqrt{(10^2 + 200^2)} = 200.25$	Z2
C4	(18,300)	$\sqrt{(2^2 + 200^2)} = 200.01$	$\sqrt{(22^2 + 700^2)} = 700.34$	Z1
C5	(28,1200)	$\sqrt{(8^2 + 700^2)} = 700.05$	$\sqrt{(12^2 + 200^2)} = 200.36$	Z2
C6	(35,1400)	$\sqrt{(15^2 + 900^2)} = 900.13$	$\sqrt{(5^2 + 400^2)} = 400.03$	Z2
C7	(45,1800)	$\sqrt{(25^2 + 1300^2)}=1300.24$	$\sqrt{(5^2 + 800^2)}=800.01$	Z2

Cluster Assignment after Iteration 1:

- **Cluster 1**: C1, C4
- **Cluster 2**: C2, C3, C5, C6, C7

Step 2: Recalculate Centroids

New Z1 (mean of C1, C4):

- Age = (20 + 18)/2 = 19
- Amount = (500 + 300)/2 = 400
  → Z1 = (19, 400)

New Z2 (mean of C2, C3, C5, C6, C7):

- Age = (40 + 30 + 28 + 35 + 45)/5 = 35.6
- Amount = (1000 + 800 + 1200 + 1400 + 1800)/5 = 1240
  → Z2 = (35.6, 1240)

Step 3: Reassign Points Based on New Centroids

Customer	Coordinates	Dist to Z1 (19,400)	Dist to Z2 (35.6,1240)	Cluster
C1	(20,500)	$\sqrt{(1^2 + 100^2)} = 100.00$	$\sqrt{(15.6^2 + 740^2)} \approx 740.16$	Z1
C2	(40,1000)	$\sqrt{(21^2 + 600^2)} \approx 600.37$	$\sqrt{(4.4^2 + 240^2)} \approx 240.04$	Z2
C3	(30,800)	$\sqrt{(11^2 + 400^2)} \approx 400.15$	$\sqrt{(5.6^2 + 440^2)} \approx 440.04$	Z2
C4	(18,300)	$\sqrt{(1^2 + 100^2)} = 100.00$	$\sqrt{(17.6^2 + 940^2)} \approx 940.16$	Z1
C5	(28,1200)	$\sqrt{(9^2 + 800^2)} \approx 800.05$	$\sqrt{(7.6^2 + 40^2)} \approx 40.72$	Z2
C6	(35,1400)	$\sqrt{(16^2 + 1000^2)} \approx 1000.13$	$\sqrt{(0.6^2 + 160^2)} \approx 160.00$	Z2
C7	(45,1800)	$\sqrt{(26^2 + 1400^2)} \approx 1400.24$	$\sqrt{(9.4^2 + 560^2)} \approx 560.08$	Z2

Final Cluster Assignment (After Convergence):

- **Cluster 1**: C1, C4
- **Cluster 2**: C2, C3, C5, C6, C7

Final Clusters:

Cluster 1 (Low income, young):

- C1: (20, 500)
- C4: (18, 300)

Cluster 2 (Mid-high income, older):

- C2: (40, 1000)

- C3: (30, 800)
- C5: (28, 1200)
- C6: (35, 1400)
- C7: (45, 1800)

Example 1:-

```
from sklearn.cluster import KMeans

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from matplotlib import pyplot as plt

%matplotlib inline

df = pd.read_csv("income.csv")

df.head()

plt.scatter(df.Age,df['Income($)'])

plt.xlabel('Age')

plt.ylabel('Income($)')

km = KMeans(n_clusters=3)

y_predicted = km.fit_predict(df[['Age','Income($)']])

y_predicted

df['cluster']=y_predicted

df.head()

km.cluster_centers_

df1 = df[df.cluster==0]

df2 = df[df.cluster==1]

df3 = df[df.cluster==2]

plt.scatter(df1.Age,df1['Income($)'],color='green')

plt.scatter(df2.Age,df2['Income($)'],color='red')

plt.scatter(df3.Age,df3['Income($)'],color='black')

plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1]

        ,color='purple',marker='*',label='centroid')

plt.xlabel('Age')
```

```
plt.ylabel('Income ($)')
plt.legend()
scaler = MinMaxScaler()

scaler.fit(df[['Income($)']])
df['Income($)'] = scaler.transform(df[['Income($)']])

scaler.fit(df[['Age']])
df['Age'] = scaler.transform(df[['Age']])
plt.scatter(df.Age,df['Income($)'])
km = KMeans(n_clusters=3)
y_predicted = km.fit_predict(df[['Age','Income($)']])
y_predicted
df['cluster']=y_predicted
df.head()
km.cluster_centers_
df1 = df[df.cluster==0]
df2 = df[df.cluster==1]
df3 = df[df.cluster==2]
plt.scatter(df1.Age,df1['Income($)'],color='green')
plt.scatter(df2.Age,df2['Income($)'],color='red')
plt.scatter(df3.Age,df3['Income($)'],color='black')
plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1],color='purple',marker='*',label=
'centroid')
plt.legend()
```

DBSCAN:

0) Definitions (quick)

- **ε-neighborhood** of a point *p*: all points within distance ≤ ε of *p* (including *p* itself).
- A point is **core** if its ε-neighborhood has **at least `minPts` points** (including itself).
- A point is **border** if it's **within ε of a core** point but **is not core** itself.
- Otherwise it's **noise**.
- Clusters are formed by **expanding from core points**, collecting all points that are **density-reachable**.

We'll use Euclidean distance.

1) Data (2D points)

We'll label points for readability:

Label	(x, y)
A1	(1, 1)
A2	(1, 2)
A3	(2, 1)
A4	(2, 2)
AB (border candidate)	**(3.2, 2)**
B1	(8, 8)
B2	(8, 9)
B3	(9, 8)
B4	(9, 9)
N (noise candidate)	**(5, 5)**

Euclidean distance $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

	A1	A2	A3	A4	AB	B1	B2	B3	B4	N
A1	0.000	1.000	1.000	1.414	2.417	9.220	9.899	9.899	10.630	5.657
A2	1.000	0.000	1.414	1.000	2.200	8.485	9.220	9.220	9.899	5.000

	A1	A2	A3	A4	AB	B1	B2	B3	B4	N
A3	1.000	1.414	0.000	1.000	1.562	8.485	9.220	8.062	8.899	4.243
A4	1.414	1.000	1.000	0.000	1.200	7.810	8.485	8.062	8.899	4.243
AB	2.417	2.200	1.562	1.200	0.000	6.403	7.071	6.403	7.071	2.863
B1	9.220	8.485	8.485	7.810	6.403	0.000	1.000	1.000	1.414	4.243
B2	9.899	9.220	9.220	8.485	7.071	1.000	0.000	1.414	1.000	5.000
B3	9.899	9.220	8.062	8.062	6.403	1.000	1.414	0.000	1.000	5.000
B4	10.630	9.899	8.899	8.899	7.071	1.414	1.000	1.000	0.000	5.657
N	5.657	5.000	4.243	4.243	2.863	4.243	5.000	5.000	5.657	0.000

How to read

- Distance from **A1 to A2** = **1.000**
- Distance from **A4 to AB** = **1.200** ($\leq \varepsilon$=1.5 $\Rightarrow$ AB is reachable from A4)
- Distance from **N to anyone** is large (> 2.8), so N is noise for small ε.

DBSCAN CODE:

Example 1: -

```
import numpy as np
from sklearn.cluster import DBSCAN
points = {
   "A1": (1, 1),
   "A2": (1, 2),
   "A3": (2, 1),
   "A4": (2, 2),
   "AB": (3.2, 2),  # border candidate
   "B1": (8, 8),
   "B2": (8, 9),
   "B3": (9, 8),
   "B4": (9, 9),
```

```
    "N":  (5, 5),    # noise candidate
}

labels = list(points.keys())
X = np.array([points[k] for k in labels])

# eps=1.5, min_samples=3; min_samples includes the point itself
db = DBSCAN(eps=1.5, min_samples=3, metric="euclidean").fit(X)
cluster_ids = db.labels_  # -1 = noise

out = {name: cid for name, cid in zip(labels, cluster_ids)}
print(out)
```

DBSCAN CODE:

Example 2:-

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np

# Load dataset
iris = load_iris()
X = iris.data
y_true = iris.target  # Actual species labels (0, 1, 2)

# Put into DataFrame for readability
df = pd.DataFrame(X, columns=iris.feature_names)
df['species'] = y_true
df.head()
```

```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# Step 1: Feature scaling (important for DBSCAN because it uses distance)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 2: Create and fit DBSCAN
dbscan = DBSCAN(eps=0.6, min_samples=5, metric='euclidean')
dbscan.fit(X_scaled)

# Step 3: Get cluster labels (-1 means noise)
labels = dbscan.labels_
df['cluster'] = labels
df.head(10)
n_clusters = len(set(labels) - {-1})
n_noise = list(labels).count(-1)

print(f"Number of clusters found: {n_clusters}")
print(f"Number of noise points: {n_noise}")

pd.crosstab(df['cluster'], df['species'])
```

Chart Explanation: -

```
import matplotlib.pyplot as plt

# Simple scatter plot of clusters
plt.figure(figsize=(6, 4))
```

```
# Plot points, coloring them by cluster label
plt.scatter(
   X_scaled[:, 2],   # Petal length (scaled)
   X_scaled[:, 3],   # Petal width (scaled)
   c=labels,        # Color by cluster ID
   cmap='Spectral',  # Colormap
   s=50,            # Marker size
   edgecolors='k'    # Black outline for visibility
)

plt.xlabel('Petal length (scaled)')
plt.ylabel('Petal width (scaled)')
plt.title('DBSCAN Clustering (Easy Visualization)')
plt.colorbar(label='Cluster ID')  # Show color legend
plt.show()
```

Hierarchical Clustering

```
# hierarchical_prediction.py

import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv('income.csv')

# Select numeric columns
```

```
X = df.select_dtypes(include=['float64', 'int64'])
if 'ID' in X.columns:
    X = X.drop(columns=['ID'])

# Scale data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply Agglomerative Clustering
model = AgglomerativeClustering(n_clusters=3, linkage='ward', metric='euclidean')
labels = model.fit_predict(X_scaled)

# Add cluster column to dataframe
df['Cluster'] = labels

# Show results
print(df.head())

# Optional: 2D Visualization (if only 2 columns)
if X_scaled.shape[1] == 2:
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels, cmap='rainbow')
    plt.title("Agglomerative Clustering Result")
    plt.xlabel(X.columns[0])
    plt.ylabel(X.columns[1])
    plt.grid(True)
    plt.show()
df.head(5)
```

Q-Learning

- Q-Learning is an **off-policy** algorithm that learns the optimal policy by always considering the *best future action* (greedy).

SARSA

- SARSA is an **on-policy** algorithm that learns the policy it actually follows (including exploration).
- It updates Q-values using the *action actually taken* in the next state.

General Story (Easy Language)

- **States** = Product customer is buying (Mobile, Laptop, Camera).
- **Actions** = What extra item you suggest (Headphone, Cover, Charger, etc.).
- **Reward** = How much customer liked the suggestion (5 = very good, 1 = not much).
- **Goal** = Learn the **best recommendation** for each product.
- So it updates Q-value by looking at **maximum future value**.

SARSA

⚡ Example: If someone buys **Mobile**, and you suggest **Headphone**:

- Reward = 5
- Update Q(Mobile, Headphone) using **max future guess**.
- Plus assume next time we will pick the action we actually tried (even if random).
- So it updates Q-value by looking at **the real next action chosen**.

⚡ Example: If someone buys **Laptop**, and you suggest **Mouse**:

- Reward = 4
- Next time maybe you randomly suggested **Bag**.
- Update Q(Laptop, Mouse) using the value of **Bag**, not the best item.

Aspect	Q-Learning	SARSA
Thinking style	"Always assume I'll be smart next time (pick best)."	"Assume I'll do what I actually did (maybe random)."
Behavior	More aggressive (directly tries to learn optimal).	Safer, learns actual policy (includes mistakes).
Example	Suggesting Mobile → Always think "Headphone is best, so future = Headphone."	Suggesting Mobile → If you tried Cover next time, update using Cover, even if it's not best.

Aspect	SARSA (On-policy)	Q-Learning (Off-policy)
Next action used	Action actually taken (A')	Best possible action (max)
Policy type	On-policy	Off-policy
Behavior	Safer, accounts for exploration	More aggressive, assumes greedy future
Convergence	Converges to **optimal policy** if ε → 0 gradually	Converges faster to optimal policy
Exploration effect	Directly reflected	Ignored in update (learns as if always greedy)

Q-LEARNING

```
import numpy as np
import random

# States and Actions
states = ["Mobile", "Laptop", "Camera"]
actions = {
    "Mobile": ["Headphone", "Cover", "Charger"],
    "Laptop": ["Mouse", "Bag", "Charger"],
    "Camera": ["Tripod", "Lens", "Bag"]
}

# Q-table (State x Action)
Q = {state: {action: 0 for action in acts} for state, acts in actions.items()}

# Rewards (simulated customer behavior)
rewards = {
    ("Mobile", "Headphone"): 5,
    ("Mobile", "Cover"): 3,
```

```
    ("Mobile", "Charger"): 1,
    ("Laptop", "Mouse"): 4,
    ("Laptop", "Bag"): 2,
    ("Laptop", "Charger"): 1,
    ("Camera", "Lens"): 5,
    ("Camera", "Tripod"): 3,
    ("Camera", "Bag"): 2
}

# Learning Parameters
alpha = 0.1   # learning rate
gamma = 0.9   # discount factor
episodes = 1000

# Training
for _ in range(episodes):
    state = random.choice(states)
    action = random.choice(actions[state])

    reward = rewards.get((state, action), 0)

    # Update Q-value
    Q[state][action] = Q[state][action] + alpha * (reward + gamma * max(Q[state].values()) - Q[state][action])

# Final Policy (Best action for each state)
policy = {state: max(acts, key=acts.get) for state, acts in Q.items()}

print("Learned Policy (Best Recommendation):")
print(policy)
```

```
old_value = Q[state][action]            # Purana Q-value
future_value = max(Q[state].values())    # Future me sabse acha action ka Q-value
target = reward + gamma * future_value   # Naya expected value (reward + future reward)

# Update rule
Q[state][action] = old_value + alpha * (target - old_value)
```

SARSA

```
import numpy as np
import random

# States and Actions
states = ["Mobile", "Laptop", "Camera"]
actions = {
    "Mobile": ["Headphone", "Cover", "Charger"],
    "Laptop": ["Mouse", "Bag", "Charger"],
    "Camera": ["Tripod", "Lens", "Bag"]
}

# Q-table (State x Action)
Q = {state: {action: 0 for action in acts} for state, acts in actions.items()}

# Rewards (simulated customer behavior)
rewards = {
    ("Mobile", "Headphone"): 5,
    ("Mobile", "Cover"): 3,
```

```
    ("Mobile", "Charger"): 1,
    ("Laptop", "Mouse"): 4,
    ("Laptop", "Bag"): 2,
    ("Laptop", "Charger"): 1,
    ("Camera", "Lens"): 5,
    ("Camera", "Tripod"): 3,
    ("Camera", "Bag"): 2
}

# Learning Parameters
alpha = 0.1   # learning rate
gamma = 0.9   # discount factor
episodes = 1000

# Training (SARSA)
for _ in range(episodes):
    state = random.choice(states)
    action = random.choice(actions[state])

    reward = rewards.get((state, action), 0)

    # pick next_action (actually taken, not max)
    next_action = random.choice(actions[state])

    # SARSA update
    Q[state][action] = Q[state][action] + alpha * (
        reward + gamma * Q[state][next_action] - Q[state][action]
    )
```

```
# Final Policy (Best action for each state)
policy = {state: max(acts, key=acts.get) for state, acts in Q.items()}

print("Learned Policy (Best Recommendation):")
print(policy)
```

APRIORI ALGO

```
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

# Sample transactions (each row = customer basket)
dataset = [
    ['Milk', 'Bread', 'Butter'],
    ['Bread', 'Butter'],
    ['Milk', 'Bread'],
    ['Milk', 'Bread', 'Butter', 'Jam'],
    ['Bread', 'Jam'],
    ['Milk', 'Butter']
]

# Convert into one-hot encoded DataFrame
from mlxtend.preprocessing import TransactionEncoder
te = TransactionEncoder()
te_array = te.fit(dataset).transform(dataset)

df = pd.DataFrame(te_array, columns=te.columns_)
```

```
print(df)
# Find frequent itemsets with min_support=0.5 (50% transactions)
frequent_itemsets = apriori(df, min_support=0.5, use_colnames=True)
print(frequent_itemsets)

# Generate rules from frequent itemsets
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.0)
print(rules[['antecedents','consequents','support','confidence','lift']])
```

Use an algorithm to balance exploration (trying both versions) winrate pattern

```
import random

# 2 coins with different win rates
coin_A_winrate = 0.5   # 50% win
coin_B_winrate = 0.7   # 70% win

epsilon = 0.1  # 10% exploration
wins_A, wins_B = 0, 0

for i in range(100):
    # epsilon-greedy choice
    if random.random() < epsilon:
        choice = random.choice(["A", "B"])  # explore
    else:
        choice = "A" if wins_A > wins_B else "B"  # exploit

    # simulate toss
    if choice == "A" and random.random() < coin_A_winrate:
```

```
        wins_A += 1
    elif choice == "B" and random.random() < coin_B_winrate:
        wins_B += 1

print("Wins from Coin A:", wins_A)
print("Wins from Coin B:", wins_B)
```

ANN (MLPClassifier)

```
# Step 1: Import Libraries
from sklearn.neural_network import MLPClassifier
import numpy as np

# Step 2: Create Simple Dataset
# Format: [[Marks, Attendance], ...]
X = np.array([
    [85, 90],   # Pass
    [90, 95],   # Pass
    [45, 50],   # Fail
    [30, 40],   # Fail
    [75, 80],   # Pass
    [55, 60],   # Fail
    [88, 85],   # Pass
    [40, 35]    # Fail
])

# Labels: 1 = Pass, 0 = Fail
y = np.array([1, 1, 0, 0, 1, 0, 1, 0])
```

```
# Step 3: Create & Train ANN Model
model = MLPClassifier(
    hidden_layer_sizes=(4,),   # 1 hidden layer with 4 neurons
    activation='relu',         # Activation function
    max_iter=1000,             # Max training steps
    random_state=42            # For same result every time
)

model.fit(X, y)

# Step 4: Predict for New Student
new_student = [[70, 75]]  # Marks=70, Attendance=75
prediction = model.predict(new_student)

# Step 5: Show Result
if prediction[0] == 1:
    print("✅ Student will PASS!")
else:
    print("❌ Student will FAIL!")
```

ANN (MLPRegressor)

Example 1:-

```
# Simple ANN for Regression without scaling
import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

```
# 1) Tiny synthetic product dataset
X = np.array([
   [10, 5, 200],
   [12, 6, 220],
   [8,  2, 150],
   [20, 10,300],
   [15, 8, 260],
   [7,  1, 120],
   [18, 9, 280],
   [9,  3, 170],
   [11, 4, 190],
   [14, 7, 250]
], dtype=float)

y = np.array([120, 140, 80, 260, 190, 60, 230, 95, 130, 170], dtype=float)

# 2) Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3) Create & train ANN without scaling
model = MLPRegressor(
   hidden_layer_sizes=(8,4),
   activation='relu',
   solver='adam',  # sgd//
   learning_rate_init=0.01,
   max_iter=2000,       # more iterations may be needed without scaling
   random_state=42
)
d
```

```
model.fit(X_train, y_train)

# 4) Evaluate
y_pred = model.predict(X_test)
print("Test MSE:", mean_squared_error(y_test, y_pred))
print("Test R2 :", r2_score(y_test, y_pred))

# 5) Predict new product directly
new_product = np.array([[13, 5, 210]])
pred_sales = model.predict(new_product)[0]
print(f"Predicted sales: {pred_sales:.1f} units")
```

ANN (MLPRegressor)

Example 2:-

```
# Diwali Sales Prediction using ANN (MLPRegressor)

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error, r2_score

# 1) Load dataset
df = pd.read_csv("diwali.csv")
print("Dataset shape:", df.shape)
print(df.head(2))

# 2) Select useful columns (features + target)
features = ["Gender", "Age", "Marital_Status", "Occupation", "Orders"]
```

```
target = "Amount"
data = df[features + [target]].copy()

# 3) Encode categorical column (Gender, Occupation)
le_gender = LabelEncoder()
data["Gender"] = le_gender.fit_transform(data["Gender"])   # M/F → 0/1

le_occ = LabelEncoder()
data["Occupation"] = le_occ.fit_transform(data["Occupation"])

# 4) Features (X) and target (y)
X = data.drop(columns=[target])
y = data[target]

# 5) Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 6) Scale features
scaler = StandardScaler().fit(X_train)
X_train_s = scaler.transform(X_train)
X_test_s = scaler.transform(X_test)

# 7) Build and train ANN model
model = MLPRegressor(
    hidden_layer_sizes=(16, 8),  # 2 hidden layers
    activation="relu",
    solver="adam",
    learning_rate_init=0.01,
    max_iter=1000,
```

```
    random_state=42
)
model.fit(X_train_s, y_train)

# 8) Evaluate
y_pred = model.predict(X_test_s)
print("\nTest MSE:", mean_squared_error(y_test, y_pred))
print("Test R2 :", r2_score(y_test, y_pred))

# 9) Predict for a sample customer (your given row)
sample = pd.DataFrame([[ "F", 28, 0, "Healthcare", 1 ]],
                      columns=["Gender", "Age", "Marital_Status", "Occupation", "Orders"])

# Encode like before
sample["Gender"] = le_gender.transform(sample["Gender"])
sample["Occupation"] = le_occ.transform(sample["Occupation"])

# Scale and predict
sample_scaled = scaler.transform(sample)
pred_amount = model.predict(sample_scaled)[0]

print("\nPredicted Diwali Sales Amount:", round(pred_amount, 2))
```

MDP

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

# Define the states
states = ['Sunny', 'Cloudy', 'Rainy']
```

```
P = np.array([
    [0.6, 0.3, 0.1],  # From Sunny//
    [0.2, 0.5, 0.3],  # From Cloudy
    [0.1, 0.4, 0.5]   # From Rainy
])

# Initial state distribution
initial_distribution = np.array([0.3, 0.5, 0.2])

# Simulate weather for a number of days
num_days = 100
state_sequence = []

# Start with initial distribution
current_state_probs = initial_distribution.copy()

for day in range(num_days):
    # Choose next state based on current probabilities
    next_state = np.random.choice(states, p=current_state_probs)

    state_sequence.append(next_state)

    # Update state probabilities for next day
    current_state_probs = np.dot(current_state_probs, P)

# Print first few days
print("Weather sequence (first 10 days):")
print(state_sequence[:10])
```

```
# Plot the state probabilities over time
plt.figure(figsize=(10, 6))
for i, state in enumerate(states):
    state_probs = [0] * num_days
    temp_prob = initial_distribution[i]
    for j in range(num_days):
        state_probs[j] = temp_prob
        temp_prob = np.dot(temp_prob, P[i])
pass  # We'll fix this below

# Properly compute state probabilities over time
state_probs_over_time = np.zeros((num_days, len(states)))
state_probs_over_time[0] = initial_distribution

for t in range(1, num_days):
    state_probs_over_time[t] = np.dot(state_probs_over_time[t-1], P)

# Plot
plt.figure(figsize=(10, 6))
for i, state in enumerate(states):
    plt.plot(range(num_days), state_probs_over_time[:, i], label=state)

plt.title("State Probabilities Over Time")
plt.xlabel("Day")
plt.ylabel("Probability")
plt.legend()
plt.grid(True)
plt.show()
```

Sentiment Analysis (POSITIVE / NEGATIVE)

```
from transformers import pipeline

# Load GPT-2 (small version)
generator = pipeline("text-generation", model="gpt2")

# Generate text
prompt = "The future of AI is"
outputs = generator(prompt, max_length=50, num_return_sequences=1)

print(outputs[0]['generated_text'])
classifier = pipeline("sentiment-analysis",
                      model="distilbert-base-uncased-finetuned-sst-2-english")

texts = [
    "I love this movie!",
    "This is the worst thing ever.",
    "It's okay, not great but not bad.",
    "I was thinking, in todays lecture, teach LLM or will teach tomorrow",
    "I will teach LLm tomorrow",
    "I was thinking to beat people when they ask me about my wedding",
    "Hemil has got new job congratulations to him but he was absent in he class due to fever to he was not able to give answer in interview "
]

results = classifier(texts)
for text, result in zip(texts, results):
    print(f"Text: {text} → {result['label']} ({result['score']:.2f})")
```

token counting

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

text = "Hello, I am learning about BERT!"
tokens = tokenizer.tokenize(text)
ids = tokenizer.convert_tokens_to_ids(tokens)

print("Text:", text)
print("Tokens:", tokens)
print("Token IDs:", ids)
print("Back to text:", tokenizer.decode(ids))
```

zero-shot classification approach

Example 1:-

```
# multi-label zero-shot classification
from transformers import pipeline

zero = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")

text = "Yoga improves physical health and reduces mental stress."
labels = ["fitness", "mental health", "technology", "finance"]

result = zero(text, labels, multi_label=True)
```

```
print(result)
```

zero-shot classification approach

Example 2:-

```
# topic detection using zero-shot classification
from transformers import pipeline

classifier = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")

headline = "Stock markets rally as inflation drops to a 2-year low."
topics = ["finance", "sports", "politics", "environment"]

output = classifier(headline, topics)
print(output)
```

Sentiment Analysis (POSITIVE / NEGATIVE) through RNN

```
# --- Bare-minimum LSTM classifier (toy example) ---
import re, torch, torch.nn as nn, torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence

# 1) Tiny dataset: sentence -> label (1=positive, 0=negative)
data = [
  ("I love it", 1),
  ("This is great", 1),
  ("Amazing film", 1),
```

```
  ("Bad movie", 0),
  ("I hate it", 0),
  ("Terrible", 0),
]

# 2) Build tiny vocab
def tok(s): return re.findall(r"\b\w+\b", s.lower())
vocab = {"<pad>":0, "<unk>":1}
for s,_ in data:
    for t in tok(s):
        if t not in vocab: vocab[t] = len(vocab)

# 3) Simple Dataset (returns tensor ids, label)
class TinyDS(Dataset):
    def __init__(self, data, vocab):
        self.samples = []
        for s,label in data:
            ids = [vocab.get(t, vocab["<unk>"]) for t in tok(s)]
            self.samples.append((torch.tensor(ids,dtype=torch.long), torch.tensor(label)))
    def __len__(self): return len(self.samples)
    def __getitem__(self,i): return self.samples[i]

def collate(batch):
    seqs = [b[0] for b in batch]
    labels = torch.stack([b[1] for b in batch])
    lengths = torch.tensor([len(s) for s in seqs])
    padded = pad_sequence(seqs, batch_first=True, padding_value=vocab["<pad>"])
    return padded, lengths, labels

ds = TinyDS(data, vocab)
loader = DataLoader(ds, batch_size=2, shuffle=True, collate_fn=collate)
```

```
# 4) Tiny LSTM model
class SmallLSTM(nn.Module):
    def __init__(self, vocab_size, emb=8, hid=16):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, emb, padding_idx=vocab["<pad>"])
        self.lstm = nn.LSTM(emb, hid, batch_first=True)
        self.fc = nn.Linear(hid, 2)
    def forward(self, x, lengths):
        e = self.embed(x)                    # (B,L,emb)
        out, (hn,cn) = self.lstm(e)          # hn: (1,B,hid)
        last = hn[-1]                        # (B,hid)
        return self.fc(last)

model = SmallLSTM(len(vocab))
opt = torch.optim.Adam(model.parameters(), lr=0.01)
crit = nn.CrossEntropyLoss()

# 5) Train a few epochs
for epoch in range(20):
    tot, corr = 0.0, 0
    for x, lengths, y in loader:
        logits = model(x, lengths)
        loss = crit(logits, y)
        opt.zero_grad(); loss.backward(); opt.step()
        tot += loss.item()
        preds = logits.argmax(dim=1)
        corr += (preds == y).sum().item()
    if (epoch+1)%5==0:
        print(f"Epoch {epoch+1}, loss={tot/len(loader):.3f}, acc={corr/len(ds):.2f}")
```

```
# 6) Predict helper
def predict(text):
    ids = [vocab.get(t, vocab["<unk>"]) for t in tok(text)]
    x = torch.tensor([ids], dtype=torch.long)
    with torch.no_grad():
        logits = model(x, torch.tensor([len(ids)]))
        probs = F.softmax(logits[0], dim=0)
        lab = "POS" if probs.argmax().item()==1 else "NEG"
    print(text, "->", lab, f"(pos_prob={probs[1]:.2f})")

# Try some sentences
predict("I enjoyed this")
predict("What a terrible movie")
predict("Good film")
```

Sentiment Analysis (POSITIVE / NEGATIVE) through distilbert Model

```
# lower-level: tokenization -> model -> softmax -> label
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch.nn.functional as F

model_name = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
model.eval()

text = "The new course on ML is excellent and practical."
inputs = tokenizer(text, return_tensors="pt")
```

```
with torch.no_grad():
    logits = model(**inputs).logits          # shape (1, num_labels)
probs = F.softmax(logits, dim=-1).squeeze().tolist()
labels = model.config.id2label           # {0: 'NEGATIVE', 1: 'POSITIVE'}
print("Probs:", {labels[i]: probs[i] for i in range(len(probs))})
```

Text to text model

```
# translate using T5 (text-to-text). Good for many tasks via prompt.
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

model_name = "t5-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
model.eval()

prompt = "translate English to French: I love learning AI"
inputs = tokenizer(prompt, return_tensors="pt")
# generate() performs autoregressive decoding
outputs = model.generate(**inputs, max_length=40)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Text to text model through gpt-2

```
from transformers import pipeline

# Load GPT-2 model
generator = pipeline("text-generation", model="gpt2")

print("🤖 AI Text Generator Bot Started!")
print("Type 'exit' to stop.\n")
```

```
while True:
    # User input prompt
    prompt = input("You: ")

    if prompt.lower() in ["exit", "quit", "stop"]:
        print("Bot: Chatbot stopped.")
        break

    # Generate text
    outputs = generator(
        prompt,
        max_length=100,
        num_return_sequences=1,
        do_sample=True,
        temperature=0.8,
        top_k=50,
        top_p=0.95
    )

    # Output
    print("\nBot:", outputs[0]['generated_text'])
    print("-" * 60)
```

Text to text model through openAI

Example1:-

```
from openai import OpenAI
client = OpenAI(api_key="sk-proj-roll_name")

response = client.chat.completions.create(
    model="gpt-4o-mini",
```

```
    messages=[
        {"role": "user", "content": "Write a short greeting."}
    ]
)

# correct way to get text
print(response.choices[0].message.content)
```

Text to text model through openAI

Example2:-

```
from openai import OpenAI
client = OpenAI(api_key="sk-proj-roll_name")

prompt = """Machine Learning is a powerful technology that enables computers to learn patterns
from data without being explicitly programmed. Instead of manually writing rules,
the system improves its performance automatically as more data becomes available.
This ability makes ML useful for building predictive and intelligent applications.
It is widely used across industries—finance relies on ML for fraud detection and
risk prediction, while healthcare uses it for diagnosing diseases and analysing medical images.
Retail businesses apply ML for recommendation systems and understanding customer behaviour,
 and automation benefits through smart bots and predictive maintenance. Overall,
 Machine Learning has become a core technology that drives many modern AI systems."""

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": prompt}]
)

# safer: get the message object then its content attribute
msg = response.choices[0].message
print("Raw message object:", msg)       # optional, to inspect
```

```
print("\nResult:\n", msg.content)
```

Hugging face image generation model

```
from huggingface_hub import InferenceClient
from IPython.display import display

client = InferenceClient(
    provider="nebius",
    api_key="hf_rollno_name"
)

# Generate image (returns PIL.Image)
image = client.text_to_image(
    "Astronaut riding a horse",
    model="black-forest-labs/FLUX.1-dev",
)

# Save image to disk
 image.save("flux_output.png")
 print("Image saved as flux_output.png")

#Display image in Jupyter cell
display(image)
```

Image generation model through OpenAI

```
import base64
from openai import OpenAI
```

```
client = OpenAI(api_key="sk-proj-_roll_name")

# user prompt
prompt = input("Enter image description: ")

response = client.images.generate(
    model="gpt-image-1",
    prompt=prompt,
    size="1024x1024"
)

# image ko base64 se extract
image_base64 = response.data[0].b64_json

# base64 → bytes
image_bytes = base64.b64decode(image_base64)

# save image
filename = "generated_image.png"
with open(filename, "wb") as f:
    f.write(image_bytes)

print(f"\nImage saved as: {filename}")
```