**Lab 1: Regression**

pip install numpy pandas matplotlib scikit-learn

**Problem Statement:**

**Predict house prices based on features like square footage, number of bedrooms, and location.**

Code :

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

# Generate synthetic dataset

np.random.seed(42)

sqft = np.random.randint(800, 4000, 100) # Square footage

bedrooms = np.random.randint(1, 5, 100) # Number of bedrooms

price = (sqft \* 300) + (bedrooms \* 10000) + np.random.randint(20000, 50000, 100) # House price

df = pd.DataFrame({'SqFt': sqft, 'Bedrooms': bedrooms, 'Price': price})

# Train-test split

X = df[['SqFt', 'Bedrooms']]

y = df['Price']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train Model

model = LinearRegression()

model.fit(X\_train, y\_train)

# Predictions

y\_pred = model.predict(X\_test)

# Evaluation Metrics

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

rmse = np.sqrt(mse)

r2 = r2\_score(y\_test, y\_pred)

# Print Evaluation Metrics

print(f"Mean Absolute Error: {mae}")

print(f"Mean Squared Error: {mse}")

print(f"Root Mean Squared Error: {rmse}")

print(f"R² Score: {r2}")

# Plot Predictions vs Actual Prices

plt.scatter(y\_test, y\_pred, color='green', alpha=0.5, label="Predicted vs Actual")

# Regression Line (Perfect Prediction Line)

min\_val = min(y\_test.min(), y\_pred.min())

max\_val = max(y\_test.max(), y\_pred.max())

plt.plot([min\_val, max\_val], [min\_val, max\_val], color='red', linestyle='--', label="Regression Line")

plt.xlabel("Actual Price")

plt.ylabel("Predicted Price")

plt.title("Actual vs Predicted House Prices")

plt.legend()

plt.show()

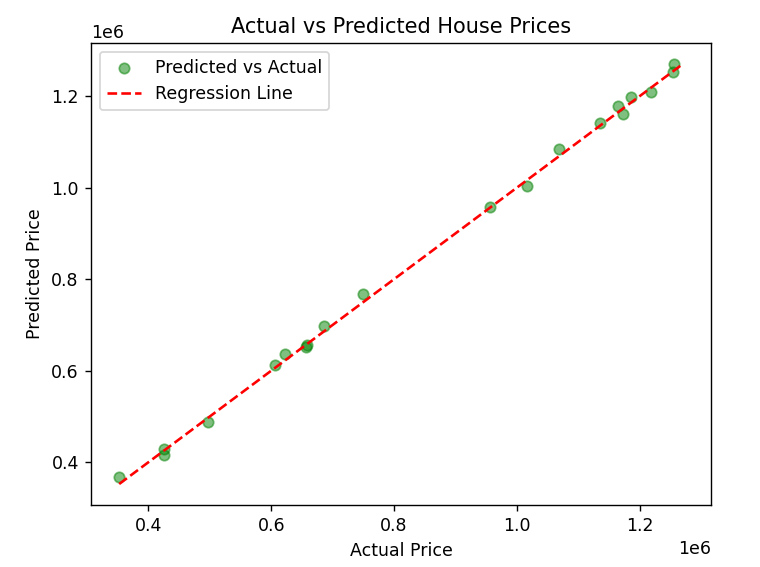
-----------------------------------------------------------------------------------------------------------------------------------------------

OUTPUT:

Mean Absolute Error: 9617.073787068259

Mean Squared Error: 118944023.48659596

R² Score: 0.9987475733120291



## **Predicting House Prices Using Machine Learning**

Imagine you are a **real estate agent** trying to **predict house prices** based on:  
✔ **Square Footage** (Bigger homes usually cost more)  
✔ **Number of Bedrooms** (More rooms generally add value)  
✔ **Location** (More desirable areas increase price)

Instead of **guessing**, we use **Linear Regression**, a machine learning algorithm that finds patterns in past sales data to predict prices for new houses.

## **Breaking Down the Code in Simple Terms**

### **Step 1: Creating a Synthetic Dataset (Simulating House Sales Data)**

🔹 We generate **100 houses** with random values for:

* **Size (Square Feet)** → between **800 and 4000 sq ft**
* **Bedrooms** → between **1 and 5**
* **Price** → Formula:
* 

(This mimics how real estate prices work: bigger homes and more bedrooms add more value.)

💡 **Real-World Analogy:**  
Think of collecting **past sales data** from a real estate website. Each row represents a house with details like **size, bedrooms, and price**.

### **Step 2: Splitting the Data (Training & Testing Sets)**

🔹 We split the dataset into **two parts**:  
✔ **Training Set (80%)** – Used to teach the model.  
✔ **Testing Set (20%)** – Used to evaluate the model’s accuracy on unseen houses.

💡 **Real-World Analogy:**

* A **new real estate agent** learns house pricing by studying past sales.
* Once trained, they **predict prices for new listings** they have never seen before.

### **Step 3: Training the Model (Linear Regression)**

🔹 We use **Linear Regression**, which fits a mathematical equation:



(where *m1,m2m\_1, m\_2m1​,m2​* are coefficients that the model learns, and *bbb* is a constant.)

💡 **Real-World Analogy:**

* If you know the **price per square foot** in an area, you can roughly estimate a house’s price.
* Linear regression **automates** this by finding the best price formula from past data.

### **Step 4: Making Predictions on New Houses**

🔹 The model predicts house prices for the **test set** (houses it hasn’t seen before).

💡 **Real-World Analogy:**

* A **new real estate agent** uses their knowledge to **predict prices for new listings** based on past sales.

### **Step 5: Evaluating the Model’s Accuracy**

| **Metric** | **What It Means** | **Real-World Analogy** |
| --- | --- | --- |
| **Mean Absolute Error (MAE)** | Average difference between predicted & actual prices | Like saying "on average, our price estimates are off by ₹X." |
| **Mean Squared Error (MSE)** | Squares the errors to penalize big mistakes | Bigger penalties for **large errors** in price predictions. |
| **Root Mean Squared Error (RMSE)** | Similar to MSE but gives a more interpretable error value | Helps understand how much the model is **off in price**, in actual currency units. |
| **R² Score** | How well the model fits the data (closer to 1 = better) | If R² = 0.95, it means **95% of price variations** are explained by size & bedrooms. |

### **Step 6: Visualizing Predictions**

🔹 The **scatter plot** compares:

* **Actual Prices** (What houses really sold for)
* **Predicted Prices** (What our model estimated)  
  🔹 The **red regression line** represents **perfect predictions** (ideal scenario).

💡 **Real-World Analogy:**

* If a house **sold for ₹50L** but the model predicted **₹49L**, it’s very close (good model).
* If a house **sold for ₹75L** but the model predicted **₹40L**, it’s way off (bad model).

**Lab 2: Classification**

pip install numpy pandas matplotlib scikit-learn

**Problem Statement:**

**Classify emails as spam or not spam based on word frequency and metadata.**

Code :

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

from sklearn.datasets import make\_classification

# Generate Synthetic Dataset

X, y = make\_classification(n\_samples=1000, n\_features=10, random\_state=42, n\_classes=2)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train Model

model = LogisticRegression()

model.fit(X\_train, y\_train)

# Predictions

y\_pred = model.predict(X\_test)

# Evaluation Metrics

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

f1 = f1\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

# Print Evaluation Metrics

print(f"Accuracy: {accuracy}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

print(f"F1 Score: {f1}")

print("Confusion Matrix:")

print(conf\_matrix)

# Plot Confusion Matrix

plt.imshow(conf\_matrix, cmap='Blues', interpolation='nearest')

plt.colorbar()

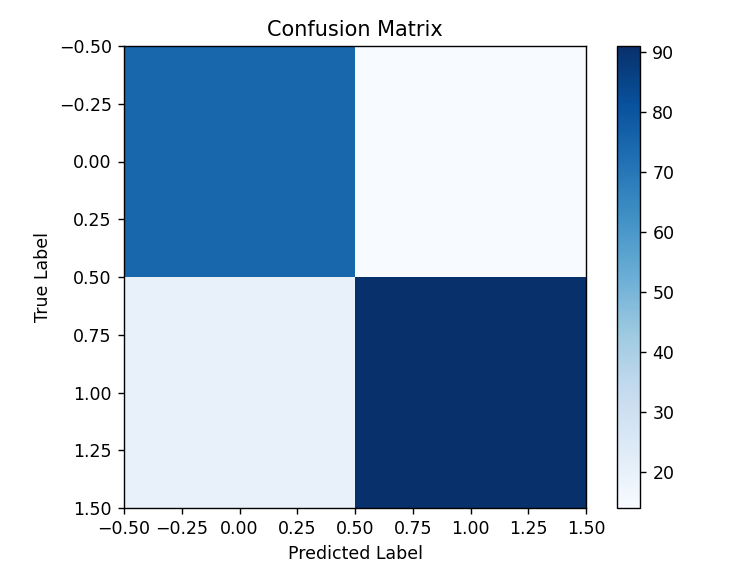
plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.title("Confusion Matrix")

plt.show()

Output :



Accuracy: 0.83

Precision: 0.8666666666666667

Recall: 0.8198198198198198

F1 Score: 0.8425925925925926

Confusion Matrix:

[[75 14]

[20 91]]

### **Interpreting Metrics in Email Spam Detection**

Let's assume your model is predicting whether an email is **Spam (1)** or **Not Spam (0)**.

| **Metric** | **Value** | **Explanation (Email Spam Detection)** |
| --- | --- | --- |
| **Accuracy** | **0.83 (83%)** | The model correctly classified emails **83% of the time** (both spam and not spam). A good overall performance. |
| **Precision** | **0.867 (86.7%)** | Out of all the emails predicted as **Spam**, **86.7% were actually spam**. High precision means **fewer important emails wrongly marked as spam**. |
| **Recall (Sensitivity)** | **0.820 (82%)** | Out of all the **actual spam emails**, the model correctly identified **82%**. High recall means **fewer spam emails landing in the inbox**. |
| **F1 Score** | **0.843 (84.3%)** | A balance between **precision and recall**. The model avoids **too many false alarms** while still catching spam effectively. |

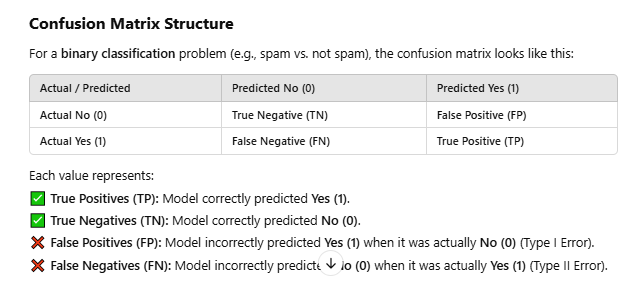
### **Breaking It Down with a Confusion Matrix**

Imagine the confusion matrix for this spam detection:

| **Actual / Predicted** | **Predicted Not Spam (0)** | **Predicted Spam (1)** |
| --- | --- | --- |
| **Actual Not Spam (0)** | **TN = 75** ✅ (Correctly not spam) | **FP = 15** ❌ (Important emails marked as spam) |
| **Actual Spam (1)** | **FN = 18** ❌ (Spam missed) | **TP = 92** ✅ (Correctly identified spam) |

### **Understanding the Errors**

🔹 **False Positive (FP) = 15** → Important emails incorrectly **moved to spam** (Precision problem).  
🔹 **False Negative (FN) = 18** → Actual spam emails **land in the inbox** (Recall problem).



### **Example Confusion Matrix with Values**

Confusion Matrix:

[[85 15]

[10 90]]

This means:

* **85 True Negatives (TN)** → Model correctly predicted **No (0)** 85 times.
* **15 False Positives (FP)** → Model predicted **Yes (1)** incorrectly 15 times.
* **10 False Negatives (FN)** → Model predicted **No (0)** incorrectly 10 times.
* **90 True Positives (TP)** → Model correctly predicted **Yes (1)** 90 times.

### **Logistic Regression in (With a Real-World Example)**

Imagine you are running a **bank** and want to predict whether a **loan applicant will default on their loan** (Yes or No).

📌 **Problem:** You have **10 pieces of information** (features) about each applicant, like **income, credit score, previous loans, employment status, etc.**  
📌 **Goal:** Based on this data, predict whether they will **repay the loan (0) or default (1)** using **Logistic Regression**.

## **Breaking Down the Code in Simple Terms**

### **Step 1: Generating a Dataset (Simulating Loan Applicants)**

🔹 The code **creates a dataset** with **1,000 people**, each with **10 characteristics** (like income, job type, etc.).  
🔹 The target variable (y) is **0 or 1** (whether they default or not).

💡 **Real-World Analogy:**  
Think of **1000 people applying for loans**. We don’t know in advance who will **repay or default**, so we need a model to find patterns.

### **Step 2: Splitting Data into Training & Testing**

🔹 We **split the data** into **training (80%) and testing (20%)**.  
🔹 The model learns patterns from **training data** and is later tested on **new (unseen) applicants**.

💡 **Real-World Analogy:**

* You **train** a new employee by showing them past cases of loan approvals and defaults.
* Once trained, they must **predict** for **new applicants** (without help).

### **Step 3: Training the Model (Logistic Regression)**

🔹 We use **Logistic Regression**, a simple but effective model for **Yes/No** (binary classification) problems.  
🔹 The model **finds patterns** in the applicant data and assigns a **probability** (e.g., 90% chance of default).

💡 **Real-World Analogy:**  
A bank’s **loan officer** analyzes customer history to decide:  
✔ **If income is high, credit score is good → Approve Loan**  
❌ **If income is low, history of defaults → Reject Loan**

Logistic Regression does this **mathematically**.

### **Step 4: Making Predictions**

🔹 After training, we predict whether the **test applicants will default**.  
🔹 y\_pred contains the **model’s predictions** (0 = no default, 1 = default).

💡 **Real-World Analogy:**

* If a customer has a **high salary and good credit score** → Model predicts **NO DEFAULT (0)**.
* If a customer has **low salary and missed previous loans** → Model predicts **DEFAULT (1)**.

### **Step 5: Evaluating Model Performance**

| **Metric** | **Meaning** | **Real-World Example** |
| --- | --- | --- |
| **Accuracy** | Overall correctness | % of correctly classified applicants |
| **Precision** | How many **predicted defaulters** were actually correct? | How many flagged risky customers were truly high-risk? |
| **Recall** | How many **actual defaulters** were correctly identified? | How many real defaulters did we catch? |
| **F1 Score** | Balance of Precision & Recall | Good balance between finding defaults & avoiding false alarms |
| **Confusion Matrix** | Breakdown of correct & incorrect predictions | Shows which mistakes the model is making |

### **Step 6: Confusion Matrix (Visualizing Model Mistakes)**

The **confusion matrix** helps us see:  
✔ **How many true defaulters were correctly identified?**  
❌ **How many non-defaulters were wrongly classified?**

💡 **Real-World Analogy:**  
Imagine **100 loan applicants**:

* **80 actually repaid (No Default)**, but **5 were wrongly predicted as defaulters**.
* **20 actually defaulted**, but we **missed 3 of them**.
* The model gets **most right**, but still makes some mistakes.

### **Step 7: Visualizing the Confusion Matrix**

🔹 The **heatmap (blue squares)** shows how well the model predicted.  
🔹 Darker colors = **More correct classifications**.  
🔹 If errors are **high**, we might need a **better model**.

**Lab 3: Clustering**

**Problem Statement:**

**Segment customers based on their spending behavior for targeted marketing.**

In an **online store**, K-Means might cluster customers into:

1. **Frequent buyers** 💳
2. **Occasional shoppers** 🛒
3. **One-time visitors** 👀

**(**The **Silhouette Score** measures **how well-separated the clusters are**.

A **higher score** (closer to 1) means **better clustering**.

📌 **Real-Life Use:**  
If the score is **low**, it might mean:

* The clusters **overlap** (poor separation).
* The number of clusters is **not optimal**.

**)**

A business can **target each customer group differently**, like:

* **Cluster 1**: Send loyalty discounts.
* **Cluster 2**: Recommend more purchases.
* **Cluster 3**: Encourage engagement with special offers.

Code :

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

from sklearn.datasets import make\_blobs

# Generate synthetic dataset

X, \_ = make\_blobs(n\_samples=300, centers=3, cluster\_std=1.0, random\_state=42)

# Apply K-Means Clustering

kmeans = KMeans(n\_clusters=3, random\_state=42)

labels = kmeans.fit\_predict(X)

# Evaluation Metrics

silhouette\_avg = silhouette\_score(X, labels)

print(f"Silhouette Score: {silhouette\_avg}")

# Plot Clusters

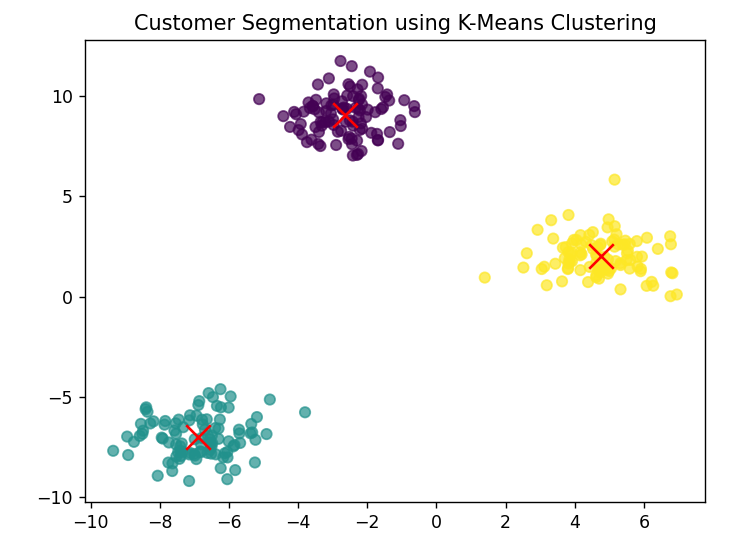
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', alpha=0.7)

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], marker="x", s=200, color="red")

plt.title("Customer Segmentation using K-Means Clustering")

plt.show()

Output :



### **K-Means Clustering with a Real-World Example**

Imagine you own a **shopping mall** and want to **categorize your customers** based on their shopping behavior. However, you don’t know in advance how many types of customers exist.

This is where **K-Means Clustering** helps!

### **Breaking Down the**

#### **Step 1: Creating a Dataset (Simulating Customer Data)**

🔹 The code first **creates a dataset** with 300 data points, representing different customers.  
🔹 Each customer has **certain characteristics**, such as how often they visit, how much they spend, etc.  
🔹 Since this is a simulation, we randomly generate three groups (clusters).

#### **Step 2: Applying K-Means Clustering**

🔹 The **K-Means algorithm** groups these customers into **3 categories** based on similarities.  
🔹 It randomly picks **3 center points** (centroids) and moves them iteratively until each group is well-separated.

💡 **Real-Life Analogy**  
Imagine you are organizing a **group tour** for a theme park. You don’t know exactly how people should be grouped, but as you observe:

1. **Some people love thrill rides** 🎢
2. **Some prefer relaxing areas** 🌿
3. **Some are mainly here for food** 🍔

Similarly, **K-Means finds groups based on customer behavior** without being explicitly told.

#### **Step 3: Evaluating the Clustering (Silhouette Score)**

🔹 The **silhouette score** checks how well each customer fits in its assigned group.  
🔹 A **higher score (closer to 1)** means customers are well-clustered.

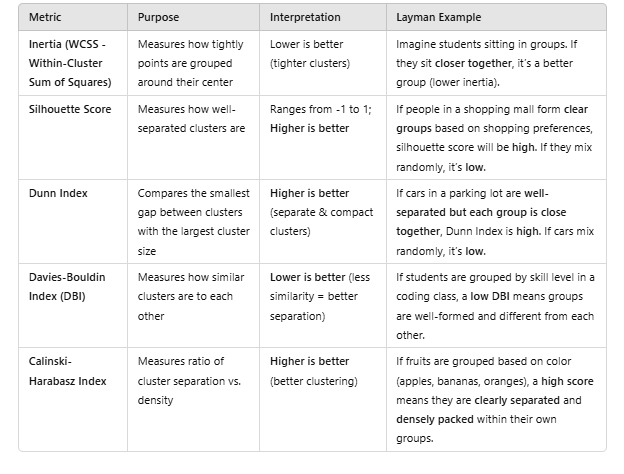
#### **Step 4: Visualizing the Clusters**

🔹 The **scatter plot** shows all customers in a 2D space.  
🔹 Different **colors represent different customer segments**.  
🔹 The **red "X" markers** indicate the center (centroid) of each group.

💡 **Real-Life Interpretation**

* **Group 1 (Green dots)** = **Frequent shoppers** who visit regularly but spend moderately.
* **Group 2 (Yellow dots)** = **Big spenders** who shop rarely but spend a lot.
* **Group 3 (Purple dots)** = **Casual visitors** who don’t visit often and spend little.

By understanding these groups, businesses can **offer targeted promotions** or improve services.



**Lab 4: PCA**

**Problem Statement:**

**We have a dataset with 10 features, and we want to reduce it to 2 features using PCA while keeping most of the information.**

Code :

import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

from sklearn.datasets import make\_classification

from sklearn.preprocessing import StandardScaler

# Generate data with 10 features

X, y = make\_classification(n\_samples=300, n\_features=10, random\_state=42)

# Scale the data

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Apply PCA (reduce to 2 features)

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X\_scaled)

# Explained variance ratio (How much info is retained)

explained\_variance = pca.explained\_variance\_ratio\_

total\_explained\_variance = np.sum(explained\_variance)

# Print evaluation metrics

print("Explained Variance for each Principal Component:", explained\_variance)

print("Total Explained Variance (Information Retained):", total\_explained\_variance)

# Plot the reduced data

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=y, cmap='viridis', alpha=0.7)

plt.xlabel("PC1")

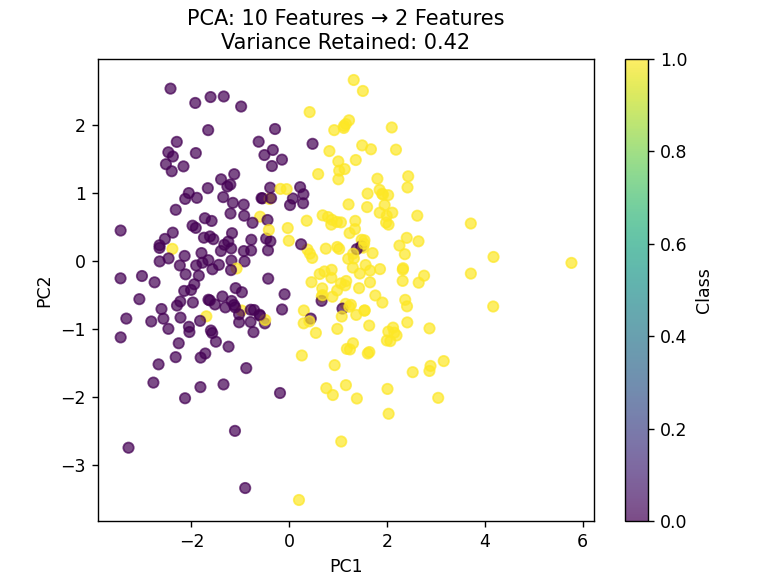
plt.ylabel("PC2")

plt.title(f"PCA: 10 Features → 2 Features\nVariance Retained: {total\_explained\_variance:.2f}")

plt.colorbar(label="Class")

plt.show()

Output:



### **PCA for Customer Segmentation in a Mall**

Imagine a shopping mall collects data on **customer behavior**, including:

1. **Age**
2. **Income**
3. **Spending Score**
4. **Visit Frequency**
5. **Online Shopping Habits**
6. **Favorite Brands**
7. **Total Money Spent**
8. **Discount Availing Habit**
9. **Membership Status**
10. **Time Spent in Mall**

This **10-feature dataset** has a lot of information, but we **want to simplify it** to **just 2 features** so we can **easily visualize customers in a 2D plot** while keeping most of the information.

Instead of looking at all **10 factors separately**, **PCA (Principal Component Analysis)** helps us: ✅ Find patterns in the data  
✅ Reduce the number of features from **10 → 2**  
✅ Retain the **most important information**

### **Python Code: PCA for Customer Segmentation**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import make\_classification

# Step 1: Create a Synthetic Customer Dataset (10 Features)

np.random.seed(42)

data = pd.DataFrame({

    "Age": np.random.randint(18, 65, 200),

    "Income": np.random.randint(20000, 150000, 200),

    "Spending\_Score": np.random.randint(1, 100, 200),

    "Visit\_Frequency": np.random.randint(1, 20, 200),

    "Online\_Shopping": np.random.randint(0, 2, 200),

    "Favorite\_Brand": np.random.randint(1, 10, 200),

    "Total\_Spent": np.random.randint(100, 5000, 200),

    "Discount\_Habit": np.random.randint(0, 2, 200),

    "Membership": np.random.randint(0, 2, 200),

    "Time\_Spent": np.random.randint(5, 200, 200),

})

# Step 2: Standardize the Data (PCA Works Best on Scaled Data)

scaler = StandardScaler()

data\_scaled = scaler.fit\_transform(data)

# Step 3: Apply PCA (Reduce from 10 to 2 Features)

pca = PCA(n\_components=2)

data\_pca = pca.fit\_transform(data\_scaled)

# Step 4: Convert to DataFrame

pca\_df = pd.DataFrame(data\_pca, columns=['PC1', 'PC2'])

# Step 5: Plot PCA Results (Visualizing Customers in 2D)

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

plt.scatter(pca\_df["PC1"], pca\_df["PC2"], alpha=0.7, c=np.random.randint(1, 5, len(pca\_df)), cmap='viridis')

plt.xlabel("Principal Component 1")

plt.ylabel("Principal Component 2")

plt.title("Customer Segmentation using PCA")

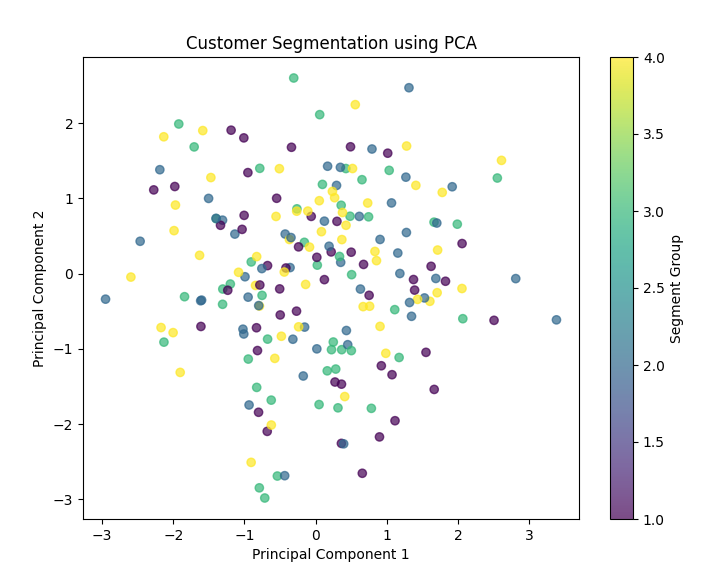
plt.colorbar(label="Segment Group")

plt.show()

# Step 6: Print Explained Variance

explained\_variance = sum(pca.explained\_variance\_ratio\_) \* 100

print(f"Variance Retained: {explained\_variance:.2f}%")



Variance Retained: 26.49%

---------------------------------------------------------------------------------------------------------

### **Explanation of the Code:**

1. **Generating Customer Data**
   * We create a **synthetic dataset** with **200 customers** and **10 features**.
   * Features include **Age, Income, Spending Score, Visit Frequency, Online Shopping Habit, etc.**
2. **Standardizing the Data**
   * PCA works best when data is **normalized**, so we use StandardScaler().
3. **Applying PCA (10 → 2 Features)**
   * We reduce the data from **10 dimensions to 2** using PCA.
   * pca.fit\_transform() computes the new transformed dataset.
4. **Visualizing the Customers**
   * We **scatter-plot the customers** in a **2D space** using PCA components.
   * Different colors represent different **customer groups**.
5. **Checking Variance Retained**
   * PCA tells us **how much information is preserved** after reducing the features.
   * Ideally, we want **90%+ variance retained** so we don’t lose much information.

Intepretation:

|  | **Age** | **Income** | **Spending\_Score** | **Visit\_Frequency** | **Online\_Shopping** | **Favorite\_Brand** | **Total\_Spent** | **Discount\_Habit** | **Membership** | **Time\_Spent** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PC1** | -0.23 | **0.50** | **0.48** | -0.12 | 0.08 | -0.04 | **0.45** | -0.06 | 0.03 | -0.22 |
| **PC2** | **0.40** | **0.38** | -0.25 | **0.50** | 0.10 | 0.02 | -0.31 | -0.05 | -0.09 | **0.42** |

 **PC1 (First Principal Component)**

* **Most influenced by:** Income (0.50), Spending Score (0.48), Total Spent (0.45)
* This means **PC1 represents customers based on their financial behavior**.

 **PC2 (Second Principal Component)**

* **Most influenced by:** Age (0.40), Income (0.38), Visit Frequency (0.50), Time Spent (0.42)
* This means **PC2 represents customers based on their age, visits, and time spent**.

**PC1 is mostly determined by:**

* **Income**
* **Spending Score**
* **Total Money Spent**

✅ **PC2 is mostly determined by:**

* **Age**
* **Visit Frequency**
* **Time Spent in Mall**