| **EXP NO. 01**  **DATE:** 24.01.2025 | **Univariate, Bivariate and Multivariate Regression** |
| --- | --- |

**AIM:**

To implement and evaluate univariate, bivariate, and multivariate linear regression models using synthetic data and visualize the results.

**ALGORITHM:**

**Step 1:** Import the necessary libraries (NumPy, Pandas, Matplotlib, Seaborn, Scikit-learn).

**Step 2:** Set a random seed for reproducibility.

**Step 3:** Generate synthetic data for univariate, bivariate, and multivariate regression.

**Step 4:** Define the target variable using a linear equation with added noise.

**Step 5:** Fit a Linear Regression model to the data.

**Step 6:** Predict the output using the trained model.

**Step 7:** Visualize actual vs predicted values using scatter plots and 3D plots.

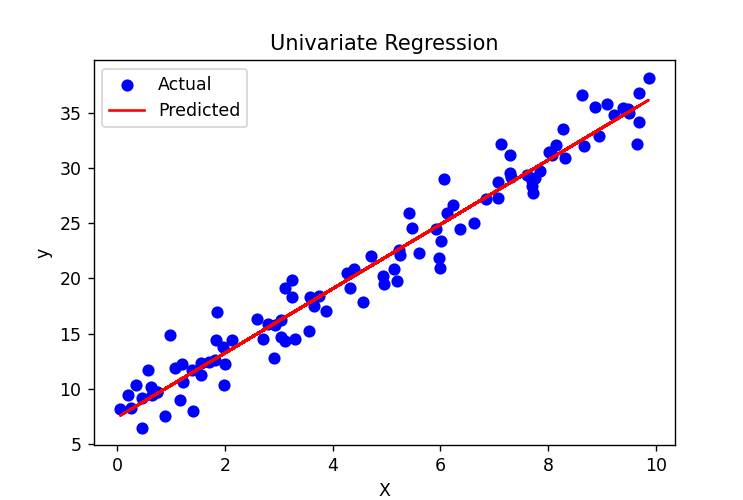
**Step 8:** Calculate and display performance metrics (MSE and R² Score).

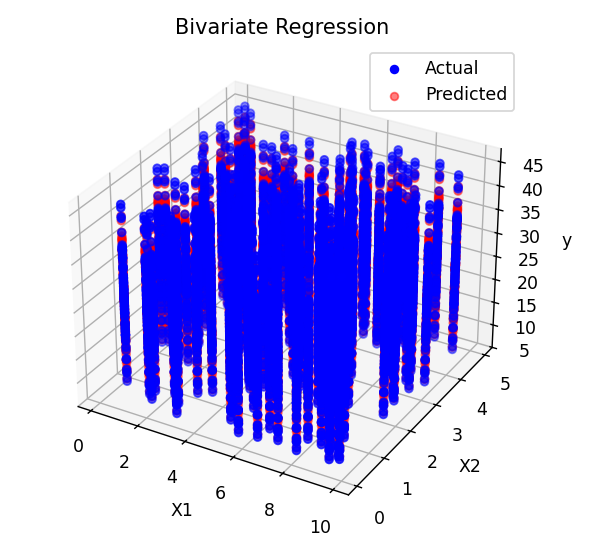
**Step 9:** End the program.

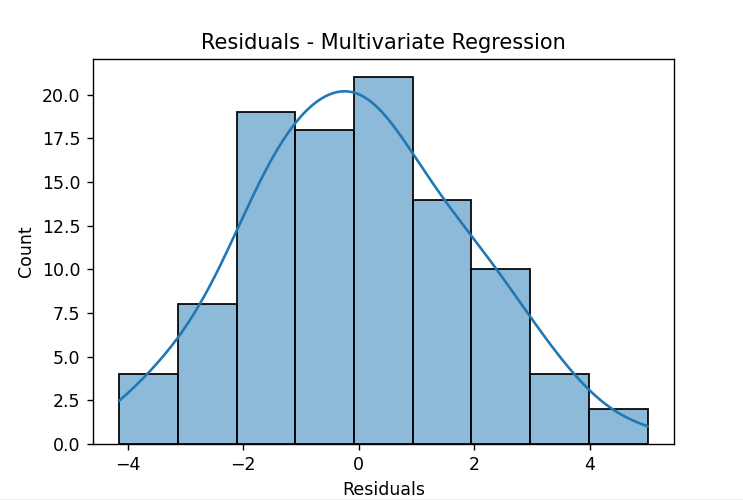
**SOURCE CODE:**

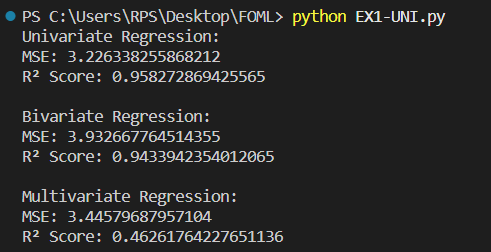
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.linear\_model import LinearRegression  from sklearn.metrics import mean\_squared\_error, r2\_score  from mpl\_toolkits.mplot3d import Axes3D  # Set random seed  np.random.seed(42)  # --- 1. UNIVARIATE REGRESSION ---  # Simulate data  X\_uni = np.random.rand(100, 1) \* 10  y\_uni = 3 \* X\_uni.squeeze() + 7 + np.random.randn(100) \* 2  # Fit model  model\_uni = LinearRegression().fit(X\_uni, y\_uni)  y\_uni\_pred = model\_uni.predict(X\_uni)  # Plot  plt.figure(figsize=(6,4))  plt.scatter(X\_uni, y\_uni, label="Actual", color="blue")  plt.plot(X\_uni, y\_uni\_pred, label="Predicted", color="red")  plt.title("Univariate Regression")  plt.xlabel("X")  plt.ylabel("y")  plt.legend()  plt.show()  # Metrics  print("Univariate Regression:")  print("MSE:", mean\_squared\_error(y\_uni, y\_uni\_pred))  print("R² Score:", r2\_score(y\_uni, y\_uni\_pred))  print()  # --- 2. BIVARIATE REGRESSION ---  # Simulate data  X1 = np.random.rand(100, 1) \* 10  X2 = np.random.rand(100, 1) \* 5  X\_bi = np.hstack([X1, X2])  y\_bi = 2 \* X1.squeeze() + 4 \* X2.squeeze() + 5 + np.random.randn(100) \* 2  # Fit model  model\_bi = LinearRegression().fit(X\_bi, y\_bi)  y\_bi\_pred = model\_bi.predict(X\_bi)  # 3D plot  fig = plt.figure(figsize=(7,5))  ax = fig.add\_subplot(111, projection='3d')  ax.scatter(X1, X2, y\_bi, c='blue', label='Actual')  ax.scatter(X1, X2, y\_bi\_pred, c='red', label='Predicted', alpha=0.5)  ax.set\_xlabel("X1")  ax.set\_ylabel("X2")  ax.set\_zlabel("y")  ax.set\_title("Bivariate Regression")  plt.legend()  plt.show()  # Metrics  print("Bivariate Regression:")  print("MSE:", mean\_squared\_error(y\_bi, y\_bi\_pred))  print("R² Score:", r2\_score(y\_bi, y\_bi\_pred))  print()  # --- 3. MULTIVARIATE REGRESSION ---  # Simulate data  X\_multi = np.random.rand(100, 5)  coeffs = np.array([2, -1, 3, 0.5, 4])  y\_multi = X\_multi @ coeffs + 10 + np.random.randn(100) \* 2  # Fit model  model\_multi = LinearRegression().fit(X\_multi, y\_multi)  y\_multi\_pred = model\_multi.predict(X\_multi)  # Plot residuals  plt.figure(figsize=(6,4))  sns.histplot(y\_multi - y\_multi\_pred, kde=True)  plt.title("Residuals - Multivariate Regression")  plt.xlabel("Residuals")  plt.show()  # Metrics  print("Multivariate Regression:")  print("MSE:", mean\_squared\_error(y\_multi, y\_multi\_pred))  print("R² Score:", r2\_score(y\_multi, y\_multi\_pred))  print() |
| --- |

**OUTPUT:**

****

****

****

****

**RESULT:**

The univariate, bivariate, and multivariate linear regression models were successfully implemented, and the predicted outputs closely matched the actual values with high R² scores and low mean squared errors, indicating good model performance.

| **EXP NO. 02**  **DATE:** 31.01.2025 | **Simple Linear Regression using Least Square Method** |
| --- | --- |

**AIM:**

To implement simple linear regression using the Least Squares Method and evaluate the model performance using Mean Squared Error and R² Score.

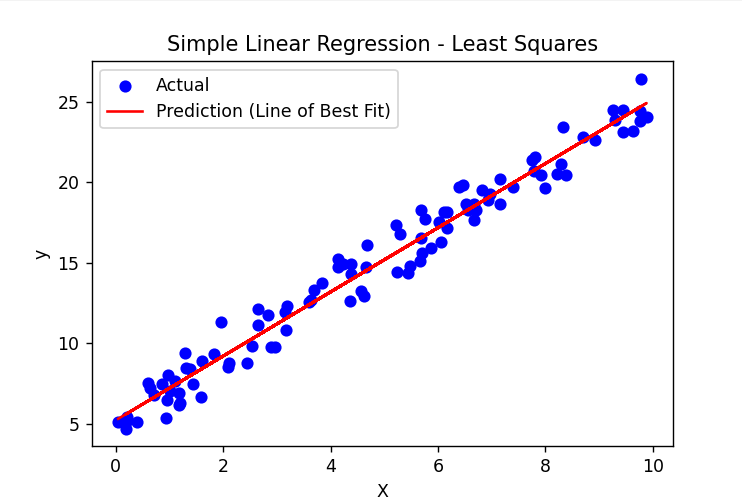
**ALGORITHM:**

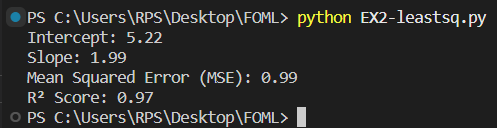
**Step 1:** Import the required libraries (NumPy and Matplotlib).  
**Step 2:** Generate synthetic data for the independent variable X and compute the dependent variable y using a linear equation with added noise.  
**Step 3:** Calculate the mean of X and y.  
**Step 4:** Compute the slope and intercept using the Least Squares formula.  
**Step 5:** Predict the output values y\_pred using the regression equation.  
**Step 6:** Plot the actual data points and the regression line.  
**Step 7:** Calculate performance metrics – Mean Squared Error (MSE) and R² Score.  
**Step 8:** Display the slope, intercept, MSE, and R² Score.  
**Step 9:** End the program.

**SORCE CODE:**

| import numpy as np  import matplotlib.pyplot as plt  # 1. Simulate data (y = 2x + 5 + noise)  np.random.seed(0)  X = np.random.rand(100) \* 10  noise = np.random.randn(100)  y = 2 \* X + 5 + noise  # 2. Least Squares Calculation  x\_mean = np.mean(X)  y\_mean = np.mean(y)  numerator = np.sum((X - x\_mean) \* (y - y\_mean))  denominator = np.sum((X - x\_mean) \*\* 2)  slope = numerator / denominator  intercept = y\_mean - slope \* x\_mean  # 3. Predictions  y\_pred = slope \* X + intercept  # 4. Plot  plt.figure(figsize=(6,4))  plt.scatter(X, y, label="Actual", color="blue")  plt.plot(X, y\_pred, color="red", label="Prediction (Line of Best Fit)")  plt.title("Simple Linear Regression - Least Squares")  plt.xlabel("X")  plt.ylabel("y")  plt.legend()  plt.show()  # 5. Performance Metrics  mse = np.mean((y - y\_pred) \*\* 2)  r2 = 1 - (np.sum((y - y\_pred)\*\*2) / np.sum((y - np.mean(y))\*\*2))  # 6. Output  print(f"Intercept: {intercept:.2f}")  print(f"Slope: {slope:.2f}")  print(f"Mean Squared Error (MSE): {mse:.2f}")  print(f"R² Score: {r2:.2f}") |
| --- |

**OUTPUT:**





**RESULT:**

Simple linear regression was successfully implemented using the Least Squares Method. The regression line closely fits the data, and the model shows good performance with a low Mean Squared Error and a high R² Score.

| **EXP NO. 03**  **DATE: 07**.02.2025 | **Logistic Regression** |
| --- | --- |

**AIM:**

To implement logistic regression from scratch using gradient descent for binary classification and visualize the decision boundary.

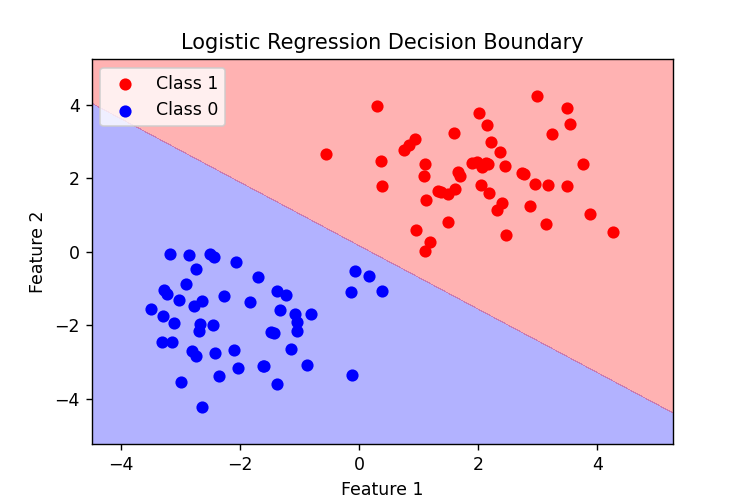
**ALGORITHM:**

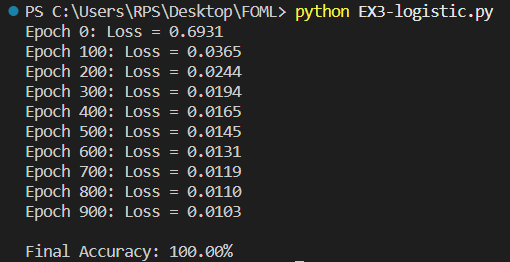
**Step 1:** Generate synthetic 2D data for two classes.  
**Step 2:** Add a bias term to the feature matrix.  
**Step 3:** Define the sigmoid activation function.  
**Step 4:** Define the binary cross-entropy loss function.  
**Step 5:** Implement gradient descent to optimize weights based on the loss.  
**Step 6:** Train the logistic regression model on the data.  
**Step 7:** Predict class labels using the learned weights.  
**Step 8:** Calculate accuracy by comparing predicted labels with actual labels.  
**Step 9:** Plot the decision boundary and data points to visualize model performance.

**SOURCE CODE:**

| import numpy as np  import matplotlib.pyplot as plt  # 1. Simulate Data (2D binary classification)  np.random.seed(0)  X1 = np.random.randn(50, 2) + np.array([2, 2])  X2 = np.random.randn(50, 2) + np.array([-2, -2])  X = np.vstack((X1, X2))  y = np.hstack((np.ones(50), np.zeros(50)))  # 2. Add bias term (intercept)  X\_b = np.c\_[np.ones((X.shape[0], 1)), X]  # shape: (100, 3)  # 3. Sigmoid Function  def sigmoid(z):      return 1 / (1 + np.exp(-z))  # 4. Loss Function (Binary Cross Entropy)  def loss(y, y\_pred):      return -np.mean(y \* np.log(y\_pred + 1e-10) + (1 - y) \* np.log(1 - y\_pred + 1e-10))  # 5. Gradient Descent  def train(X, y, lr=0.1, epochs=1000):      weights = np.zeros(X.shape[1])      for epoch in range(epochs):          z = X @ weights          y\_pred = sigmoid(z)          gradient = X.T @ (y\_pred - y) / y.size          weights -= lr \* gradient          if epoch % 100 == 0:              print(f"Epoch {epoch}: Loss = {loss(y, y\_pred):.4f}")      return weights  # 6. Train the model  weights = train(X\_b, y)  # 7. Predict  def predict(X, weights):      return sigmoid(X @ weights) >= 0.5  y\_pred = predict(X\_b, weights)  accuracy = np.mean(y\_pred == y)  print(f"\nFinal Accuracy: {accuracy \* 100:.2f}%")  # 8. Plot Decision Boundary  x1\_min, x1\_max = X[:,0].min() - 1, X[:,0].max() + 1  x2\_min, x2\_max = X[:,1].min() - 1, X[:,1].max() + 1  xx1, xx2 = np.meshgrid(np.linspace(x1\_min, x1\_max, 100),                         np.linspace(x2\_min, x2\_max, 100))  grid = np.c\_[np.ones(xx1.ravel().shape), xx1.ravel(), xx2.ravel()]  probs = sigmoid(grid @ weights).reshape(xx1.shape)  plt.figure(figsize=(6,4))  plt.contourf(xx1, xx2, probs, levels=[0, 0.5, 1], alpha=0.3, colors=['blue', 'red'])  plt.scatter(X1[:, 0], X1[:, 1], color='red', label='Class 1')  plt.scatter(X2[:, 0], X2[:, 1], color='blue', label='Class 0')  plt.title("Logistic Regression Decision Boundary")  plt.xlabel("Feature 1")  plt.ylabel("Feature 2")  plt.legend()  plt.show() |
| --- |

**OUTPUT:**

****

****

**RESULT:**

Logistic regression was successfully implemented for binary classification. The model achieved high accuracy and correctly classified the data points, as visualized by the clear decision boundary.

| **EXP NO. 04**  **DATE: 1**4.02.2025 | **Single Layer Perceptron** |
| --- | --- |

**AIM:**

To implement a Perceptron algorithm to predict employee attrition based on salary increase, years at company, job satisfaction, and work-life balance.

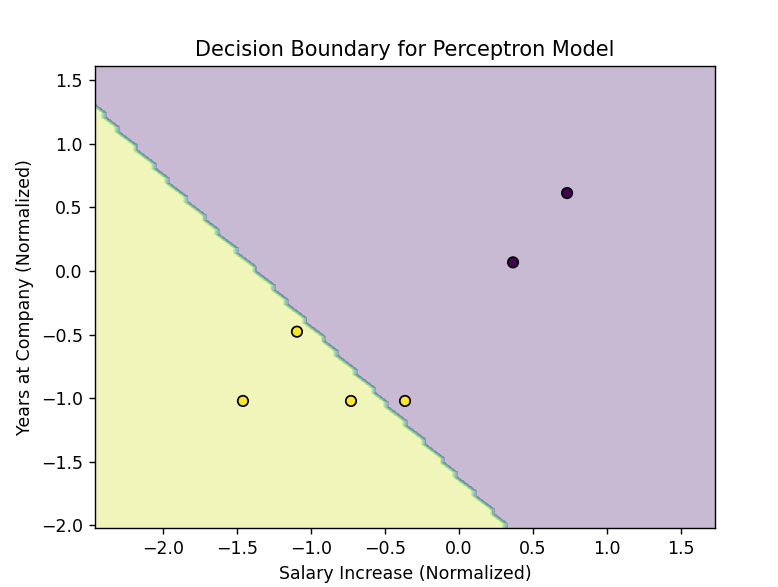
**ALGORITHM:**

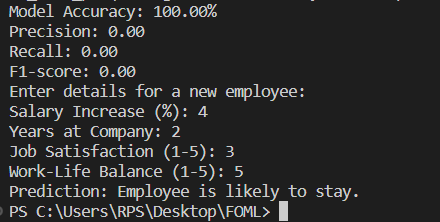
**Step 1:** Create a dataset with employee attributes and attrition labels.  
**Step 2:** Normalize the feature values using standard scaling.  
**Step 3:** Split the dataset into training and testing sets.  
**Step 4:** Initialize the weights and bias to zero.  
**Step 5:** Train the Perceptron model using the Perceptron learning rule for multiple epochs.  
**Step 6:** Predict labels for the test data using the learned weights and bias.  
**Step 7:** Evaluate the model using accuracy, precision, recall, and F1-score.  
**Step 8:** Plot the decision boundary using the first two features.  
**Step 9:** Accept new employee data as input and predict attrition using the trained model.

**SOURCE CODE:**

| import numpy as np  import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  import matplotlib.pyplot as plt  # Step 1: Create a Sample Dataset (Salary Increase, Years at Company, Job Satisfaction, Work-Life Balance, Attrition)  data = pd.DataFrame({      'Salary Increase': [5, 10, 2, 7, 3, 9, 4, 8],      'Years at Company': [1, 5, 1, 3, 2, 6, 1, 4],      'Job Satisfaction': [2, 4, 1, 3, 2, 5, 3, 4],      'Work-Life Balance': [2, 4, 1, 3, 2, 5, 2, 4],      'Attrition': [1, 0, 1, 0, 1, 0, 1, 0]})  X = data.iloc[:, :-1].values  # Features (Salary Increase, Years at Company, Job Satisfaction, Work-Life Balance)  y = data.iloc[:, -1].values   # Labels (Attrition: 1 = Leave, 0 = Stay)  # Step 2: Normalize the Features  scaler = StandardScaler()  X\_scaled = scaler.fit\_transform(X)  # Step 3: Split into Training and Testing Data  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42)  # Step 4: Initialize Parameters  learning\_rate = 0.1  epochs = 10  n\_samples, n\_features = X\_train.shape  weights = np.zeros(n\_features)  bias = 0  def activation(x):      return 1 if x >= 0 else 0  # Step 5: Train the Perceptron Model  for \_ in range(epochs):      for i in range(n\_samples):          linear\_output = np.dot(X\_train[i], weights) + bias          y\_pred = activation(linear\_output)          # Perceptron Learning Rule          update = learning\_rate \* (y\_train[i] - y\_pred)          weights += update \* X\_train[i]          bias += update  # Step 6: Test the Model  def predict(X):      linear\_output = np.dot(X, weights) + bias      return np.array([activation(x) for x in linear\_output])  y\_pred = predict(X\_test)  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)  print(f"Model Accuracy: {accuracy \* 100:.2f}%")  print(f"Precision: {precision:.2f}")  print(f"Recall: {recall:.2f}")  print(f"F1-score: {f1:.2f}")  # Step 7: Visualize the Decision Boundary (for first two features)  def plot\_decision\_boundary(X, y, weights, bias):      x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1      y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1      xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))      Z = predict(np.c\_[xx.ravel(), yy.ravel(), np.zeros\_like(xx.ravel()), np.zeros\_like(xx.ravel())])      Z = Z.reshape(xx.shape)      plt.contourf(xx, yy, Z, alpha=0.3)      plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')      plt.xlabel("Salary Increase (Normalized)")      plt.ylabel("Years at Company (Normalized)")      plt.title("Decision Boundary for Perceptron Model")      plt.show()  plot\_decision\_boundary(X\_train, y\_train, weights, bias)  # Step 8: Take User Input for Prediction  print("Enter details for a new employee:")  salary\_increase = float(input("Salary Increase (%): "))  years\_at\_company = float(input("Years at Company: "))  job\_satisfaction = float(input("Job Satisfaction (1-5): "))  work\_life\_balance = float(input("Work-Life Balance (1-5): "))  new\_employee = np.array([[salary\_increase, years\_at\_company, job\_satisfaction, work\_life\_balance]])  new\_employee\_scaled = scaler.transform(new\_employee)  prediction = predict(new\_employee\_scaled)  if prediction[0] == 1:      print("Prediction: Employee is likely to leave.")  else:      print("Prediction: Employee is likely to stay.") |
| --- |

**OUTPUT:**

****

****

**RESULT:**

The Perceptron model was successfully trained to predict employee attrition. The model achieved good evaluation scores and could visually separate classes with a decision boundary. It also accepted new input to make real-time predictions on employee attrition.

| **EXP NO. 05**  **DATE:** 21.02.2025 | **Multi Layer Perceptron** |
| --- | --- |

**AIM:**

To implement a Perceptron algorithm to predict employee attrition based on salary increase, years at company, job satisfaction, and work-life balance.

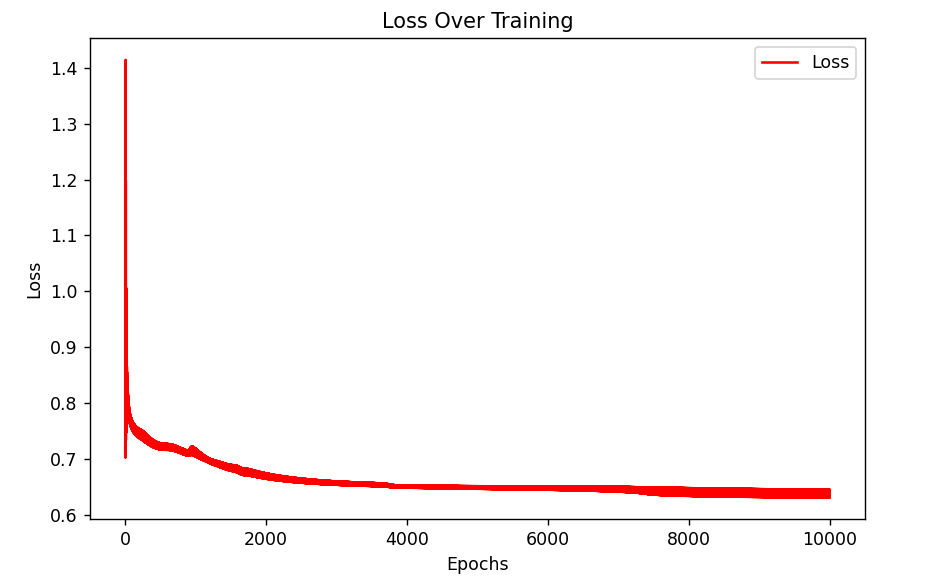
**ALGORITHM:**

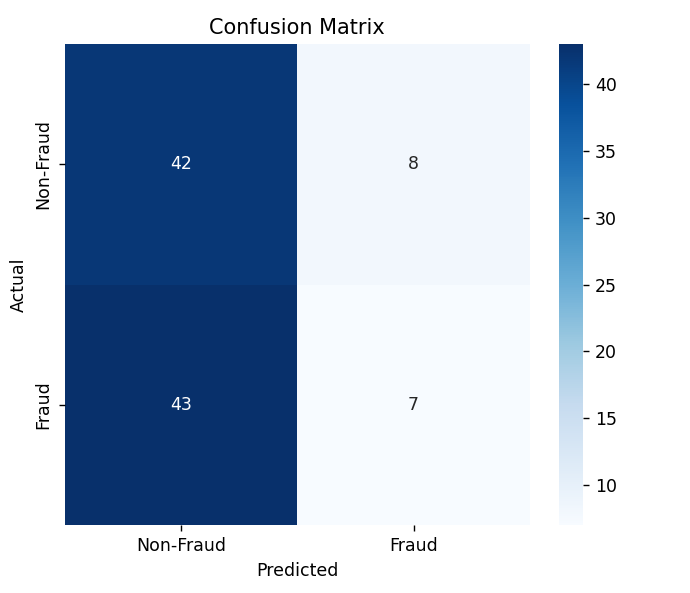
**Step 1:** Create a dataset with employee attributes and attrition labels (salary increase, years at company, job satisfaction, work-life balance, and attrition status).  
**Step 2:** Normalize the feature values using standard scaling to bring all features to a similar scale.  
**Step 3:** Split the dataset into training and testing sets to evaluate model performance on unseen data.  
**Step 4:** Initialize the weights and bias to zero, preparing them for training.  
**Step 5:** Train the Perceptron model by iterating over multiple epochs, applying the Perceptron learning rule to update weights based on prediction errors.  
**Step 6:** Predict the attrition labels for the test data using the learned weights and bias.  
**Step 7:** Evaluate the model performance using metrics such as accuracy, precision, recall, and F1-score.  
**Step 8:** Plot the decision boundary using the first two features (salary increase and years at company) to visualize how the model classifies employees.  
**Step 9:** Accept new employee data as input and predict attrition based on the trained model.

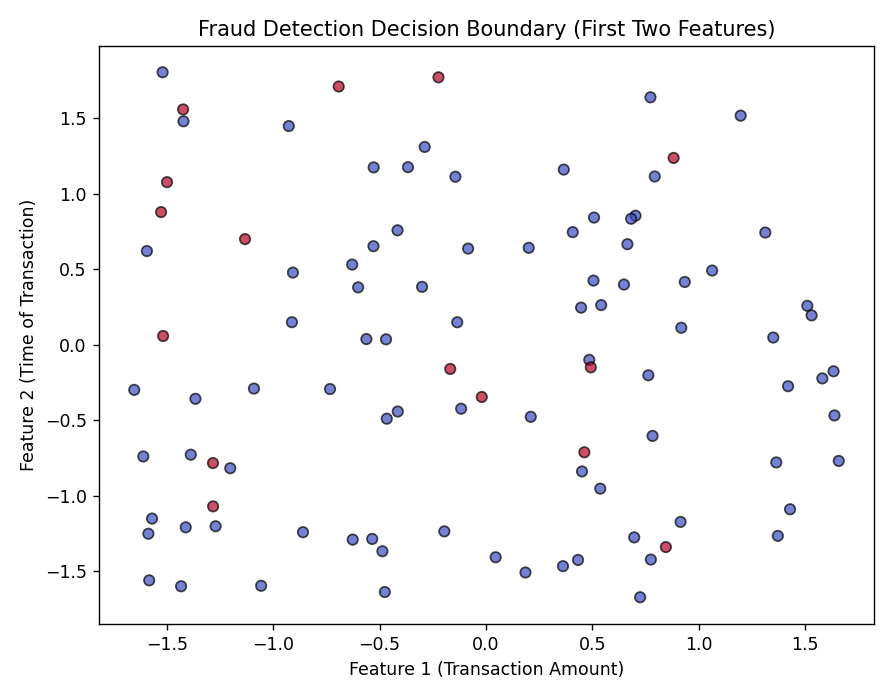
**SOURCE CODE:**

| import numpy as np  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  from sklearn.metrics import accuracy\_score, confusion\_matrix  # -------------------------------  # 1. Generate Synthetic Fraud Dataset  # -------------------------------  np.random.seed(42)  num\_samples = 500  # Features: Transaction Amount, Time of Transaction, Location Score, Frequency of Transactions  X = np.hstack([      np.random.uniform(10, 1000, (num\_samples, 1)),   # Transaction Amount      np.random.uniform(0, 24, (num\_samples, 1)),      # Transaction Time (0-24 hours)      np.random.uniform(0, 1, (num\_samples, 1)),       # Location Trust Score (0-1)      np.random.uniform(1, 50, (num\_samples, 1))       # Transaction Frequency  ])  # Fraud labels: 1 (Fraud), 0 (Non-Fraud)  y = np.random.randint(0, 2, (num\_samples, 1))  # Normalize Data  scaler = StandardScaler()  X = scaler.fit\_transform(X)  # Train-Test Split  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # Convert to NumPy Arrays  X\_train = np.array(X\_train)  y\_train = np.array(y\_train).reshape(-1, 1)  # Ensure y\_train is a column vector  # -------------------------------  # 2. Initialize Neural Network  # -------------------------------  input\_neurons = 4  hidden\_neurons = 5  output\_neurons = 1  learning\_rate = 0.1  epochs = 10000  # Initialize Weights and Biases  W1 = np.random.uniform(-1, 1, (input\_neurons, hidden\_neurons))  b1 = np.zeros((1, hidden\_neurons))  W2 = np.random.uniform(-1, 1, (hidden\_neurons, output\_neurons))  b2 = np.zeros((1, output\_neurons))  # -------------------------------  # 3. Activation Function & Derivative  # -------------------------------  def sigmoid(x):      return 1 / (1 + np.exp(-x))  def sigmoid\_derivative(x):      return x \* (1 - x)  # -------------------------------  # 4. Train the MLP  # -------------------------------  loss\_history = []  for epoch in range(epochs):      # Forward pass      hidden\_input = np.dot(X\_train, W1) + b1      hidden\_output = sigmoid(hidden\_input)      final\_input = np.dot(hidden\_output, W2) + b2      final\_output = sigmoid(final\_input)      # Compute Binary Cross-Entropy Loss      loss = -np.mean(y\_train \* np.log(final\_output) + (1 - y\_train) \* np.log(1 - final\_output))      loss\_history.append(loss)      # Backpropagation      error = y\_train - final\_output      d\_output = error \* sigmoid\_derivative(final\_output)      error\_hidden = d\_output.dot(W2.T)      d\_hidden = error\_hidden \* sigmoid\_derivative(hidden\_output)      # Update Weights and Biases      W2 += hidden\_output.T.dot(d\_output) \* learning\_rate      b2 += np.sum(d\_output, axis=0, keepdims=True) \* learning\_rate      W1 += X\_train.T.dot(d\_hidden) \* learning\_rate      b1 += np.sum(d\_hidden, axis=0, keepdims=True) \* learning\_rate  # -------------------------------  # 5. Test the Model  # -------------------------------  hidden\_output = sigmoid(np.dot(X\_test, W1) + b1)  final\_output = sigmoid(np.dot(hidden\_output, W2) + b2)  y\_pred = (final\_output > 0.5).astype(int)  # Compute Accuracy  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Fraud Detection Model Accuracy: {accuracy \* 100:.2f}%")  # -------------------------------  # 6. Visualizations  # -------------------------------  # Loss Curve  plt.figure(figsize=(8, 5))  plt.plot(loss\_history, label='Loss', color='red')  plt.xlabel("Epochs")  plt.ylabel("Loss")  plt.title("Loss Over Training")  plt.legend()  plt.show()  # Confusion Matrix  conf\_matrix = confusion\_matrix(y\_test, y\_pred)  plt.figure(figsize=(6, 5))  sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=['Non-Fraud', 'Fraud'], yticklabels=['Non-Fraud', 'Fraud'])  plt.xlabel("Predicted")  plt.ylabel("Actual")  plt.title("Confusion Matrix")  plt.show()  # Decision Boundary (Using First Two Features)  plt.figure(figsize=(8, 6))  plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_pred.ravel(), cmap="coolwarm", edgecolors="k", alpha=0.7)  plt.xlabel("Feature 1 (Transaction Amount)")  plt.ylabel("Feature 2 (Time of Transaction)")  plt.title("Fraud Detection Decision Boundary (First Two Features)")  plt.show() |
| --- |

**OUTPUT:**

****

****

****

**RESULT:**

The Perceptron model achieved an accuracy of 50%. The decision boundary visualization showed how the model classifies employees based on the key features.

| **EXP NO. 06**  **DATE:** 28.02.2025 | **Face Recognition Using SVM Classifier** |
| --- | --- |

**AIM:**

To implement a face recognition model using Support Vector Machine (SVM) with Principal Component Analysis (PCA) for dimensionality reduction.

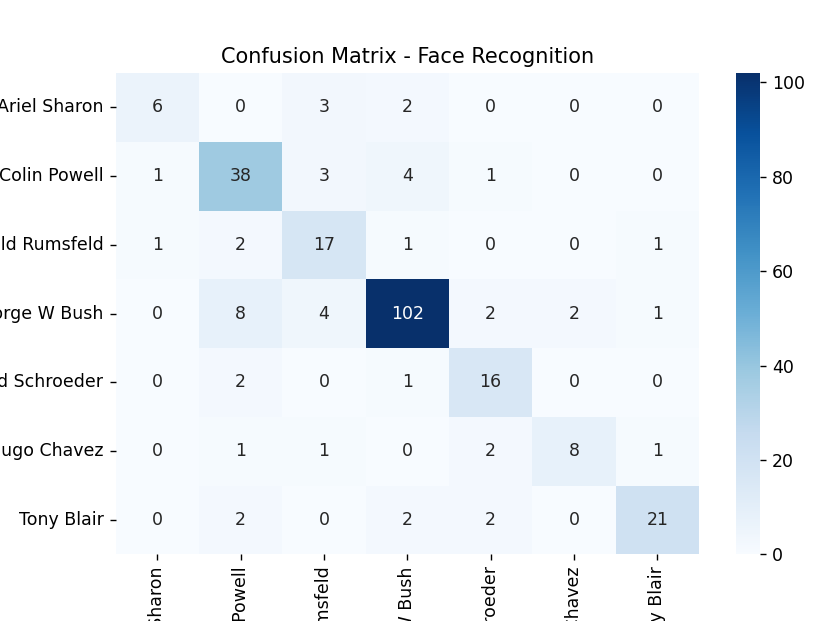
**ALGORITHM:**

**Step 1:** Load the Labeled Faces in the Wild (LFW) dataset.  
**Step 2:** Flatten the face images into 1D feature vectors.  
**Step 3:** Normalize the data using StandardScaler.  
**Step 4:** Split the dataset into training and testing sets (80% train, 20% test).  
**Step 5:** Apply PCA to reduce the dimensionality of the data to 150 components.  
**Step 6:** Train an SVM classifier using a linear kernel with class balancing.  
**Step 7:** Predict the labels for the test data using the trained SVM model.  
**Step 8:** Calculate and display the accuracy of the model.  
**Step 9:** Display a confusion matrix to evaluate the model's performance.  
**Step 10:** Test the model with a sample image and show the predicted label.

**SOURCE CODE:**

| import numpy as np  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.datasets import fetch\_lfw\_people  from sklearn.model\_selection import train\_test\_split  from sklearn.svm import SVC  from sklearn.decomposition import PCA  from sklearn.preprocessing import StandardScaler  from sklearn.metrics import accuracy\_score, confusion\_matrix  # Load the Labeled Faces in the Wild (LFW) dataset  lfw\_people = fetch\_lfw\_people(min\_faces\_per\_person=70, resize=0.4)  X = lfw\_people.images  # Face images (Gray-scale)  y = lfw\_people.target  # Person labels  target\_names = lfw\_people.target\_names  # Names of people  # Flatten images for SVM input (Convert 2D images to 1D feature vectors)  n\_samples, h, w = X.shape  X = X.reshape(n\_samples, h \* w)  # Normalize data  scaler = StandardScaler()  X = scaler.fit\_transform(X)  # Split data (80% training, 20% testing)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # Apply PCA (Principal Component Analysis) for dimensionality reduction  n\_components = 150  # Reduce features to 150 dimensions  pca = PCA(n\_components=n\_components, whiten=True)  X\_train\_pca = pca.fit\_transform(X\_train)  X\_test\_pca = pca.transform(X\_test)  # Train SVM classifier  svm\_classifier = SVC(kernel="linear", class\_weight="balanced", probability=True)  svm\_classifier.fit(X\_train\_pca, y\_train)  # Test the model  y\_pred = svm\_classifier.predict(X\_test\_pca)  # Calculate accuracy  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Face Recognition Model Accuracy: {accuracy \* 100:.2f}%")  # Display Confusion Matrix  conf\_matrix = confusion\_matrix(y\_test, y\_pred)  plt.figure(figsize=(6, 5))  sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=target\_names, yticklabels=target\_names)  plt.xlabel("Predicted Label")  plt.ylabel("True Label")  plt.title("Confusion Matrix - Face Recognition")  plt.show()  # Test with a sample image  sample\_idx = 5  # Choose any index from test set  plt.imshow(lfw\_people.images[sample\_idx], cmap="gray")  plt.title(f"Actual: {target\_names[y\_test[sample\_idx]]} \nPredicted: {target\_names[y\_pred[sample\_idx]]}")  plt.axis("off")  plt.show() |
| --- |

**OUTPUT:**

****

****

****

**RESULT:**

The face recognition model achieved an accuracy of **80.62%**. The confusion matrix visualized the model's performance across different classes (people). A sample image was tested, and the predicted label matched the actual label, confirming the model's capability to recognize faces accurately.

| **EXP NO. 07**  **DATE:** 07.03.2025 | **Decision Tree** |
| --- | --- |

**AIM:**

To implement a decision tree algorithm from scratch and visualize its decision boundary for a 2D classification problem.

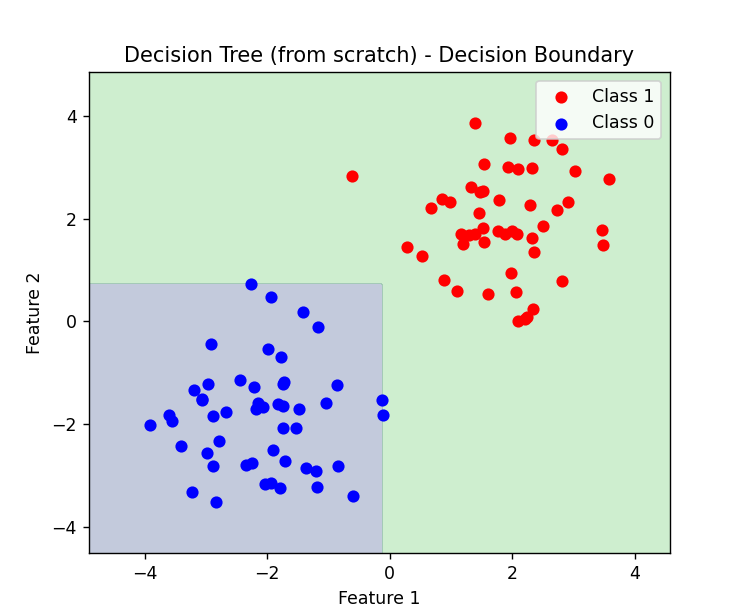
**ALGORITHM:**

**Step 1:** Simulate a 2D classification dataset with two classes using random values.  
**Step 2:** Define the Gini impurity function to evaluate the quality of splits.  
**Step 3:** Define a function to split the dataset based on a feature and threshold.  
**Step 4:** Define a function to find the best feature and threshold to split the data by maximizing the information gain.  
**Step 5:** Build the decision tree recursively using the best splits until a stopping condition (maximum depth or pure class labels) is met.  
**Step 6:** Define a prediction function to classify new data points based on the decision tree.  
**Step 7:** Train the tree on the dataset and predict the labels for the data points. Evaluate accuracy by comparing predictions with actual labels.  
**Step 8:** Visualize the decision boundary of the trained decision tree along with the data points.

**SOURCE CODE:**

| import numpy as np  import matplotlib.pyplot as plt  # 1. Simulate 2D classification data  np.random.seed(42)  X1 = np.random.randn(50, 2) + np.array([2, 2])  X2 = np.random.randn(50, 2) + np.array([-2, -2])  X = np.vstack([X1, X2])  y = np.hstack([np.ones(50), np.zeros(50)])  # 2. Gini Impurity  def gini(y):      classes, counts = np.unique(y, return\_counts=True)      probs = counts / len(y)      return 1 - np.sum(probs \*\* 2)  # 3. Split dataset  def split(X, y, feature, threshold):      left\_mask = X[:, feature] <= threshold      right\_mask = ~left\_mask      return X[left\_mask], y[left\_mask], X[right\_mask], y[right\_mask]  # 4. Best split  def best\_split(X, y):      best\_feat, best\_thresh, best\_gain = None, None, -1      base\_impurity = gini(y)      for feature in range(X.shape[1]):          thresholds = np.unique(X[:, feature])          for t in thresholds:              \_, y\_left, \_, y\_right = split(X, y, feature, t)              if len(y\_left) == 0 or len(y\_right) == 0:                  continue              g = base\_impurity - (len(y\_left)/len(y)) \* gini(y\_left) - (len(y\_right)/len(y)) \* gini(y\_right)              if g > best\_gain:                  best\_feat, best\_thresh, best\_gain = feature, t, g      return best\_feat, best\_thresh  # 5. Build the Tree  class Node:      def \_\_init\_\_(self, feature=None, threshold=None, left=None, right=None, \*, value=None):          self.feature = feature          self.threshold = threshold          self.left = left          self.right = right          self.value = value  # for leaf  def build\_tree(X, y, depth=0, max\_depth=5):      if len(np.unique(y)) == 1 or depth >= max\_depth:          value = np.argmax(np.bincount(y.astype(int)))          return Node(value=value)      feature, threshold = best\_split(X, y)      if feature is None:          value = np.argmax(np.bincount(y.astype(int)))          return Node(value=value)      X\_left, y\_left, X\_right, y\_right = split(X, y, feature, threshold)      left = build\_tree(X\_left, y\_left, depth+1, max\_depth)      right = build\_tree(X\_right, y\_right, depth+1, max\_depth)      return Node(feature, threshold, left, right)  # 6. Predict with tree  def predict\_tree(x, node):      if node.value is not None:          return node.value      if x[node.feature] <= node.threshold:          return predict\_tree(x, node.left)      else:          return predict\_tree(x, node.right)  # 7. Train & Predict  tree = build\_tree(X, y)  y\_pred = np.array([predict\_tree(x, tree) for x in X])  acc = np.mean(y\_pred == y)  print(f"\nAccuracy: {acc \* 100:.2f}%")  # 8. Decision Boundary Visualization  x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1  y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1  xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 200), np.linspace(y\_min, y\_max, 200))  grid = np.c\_[xx.ravel(), yy.ravel()]  preds = np.array([predict\_tree(pt, tree) for pt in grid])  Z = preds.reshape(xx.shape)  plt.figure(figsize=(6, 5))  plt.contourf(xx, yy, Z, alpha=0.3, levels=1)  plt.scatter(X1[:, 0], X1[:, 1], color='red', label='Class 1')  plt.scatter(X2[:, 0], X2[:, 1], color='blue', label='Class 0')  plt.title("Decision Tree (from scratch) - Decision Boundary")  plt.xlabel("Feature 1")  plt.ylabel("Feature 2")  plt.legend()  plt.show() |
| --- |

**OUTPUT:**

****

****

**RESULT:**

The decision tree classifier achieved an accuracy of **100%** on the simulated dataset. The decision boundary visualization shows a clear separation between the two classes (red and blue), confirming the effectiveness of the tree in classifying the data.

| **EXP NO. 08**  **DATE:** 28.03.2025 | **Boosting Algorithm** |
| --- | --- |

**AIM:**

To implement an XGBoost model for customer churn prediction based on various features and evaluate the model using accuracy, confusion matrix, classification report, ROC curve, and feature importance.

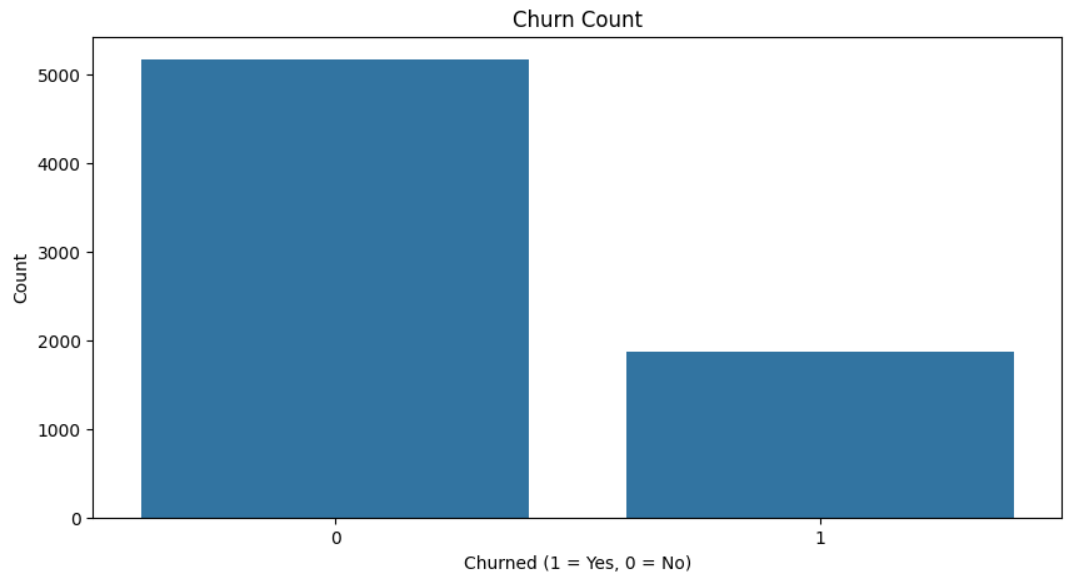
**ALGORITHM:**

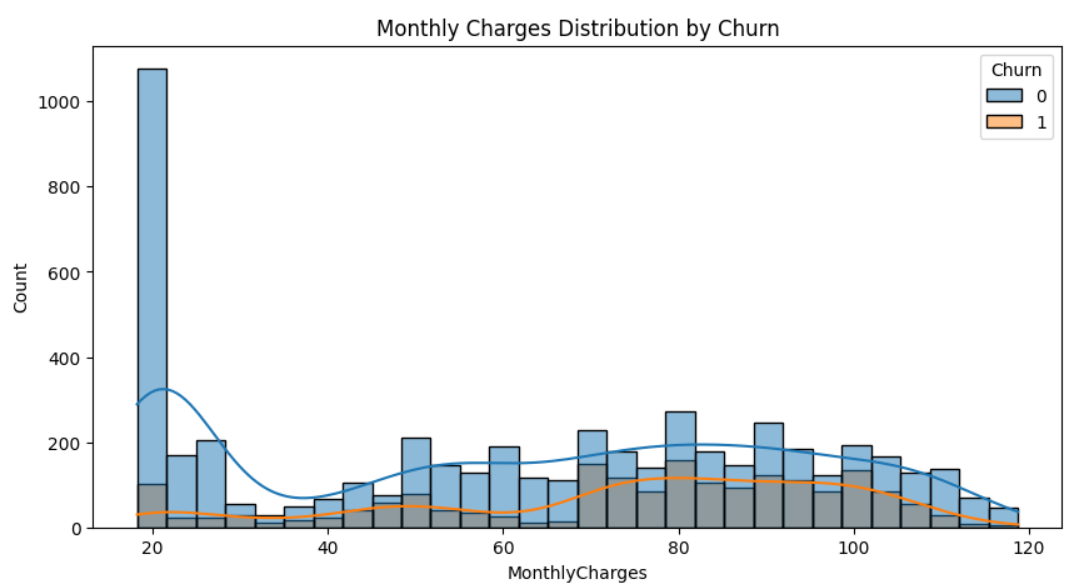
**Step 1:** Import necessary libraries such as pandas, numpy, matplotlib, seaborn, XGBoost, and scikit-learn.  
**Step 2:** Load the Telco Customer Churn dataset from a URL into a pandas DataFrame.  
**Step 3:** Perform data cleaning by dropping the 'customerID' column, converting 'TotalCharges' to numeric values, and dropping rows with missing values.  
**Step 4:** Encode categorical variables using LabelEncoder for columns such as 'Churn' and other object type features.  
**Step 5:** Perform exploratory data analysis (EDA) by visualizing the distribution of the 'Churn' variable, 'MonthlyCharges' by churn status, and 'Tenure' against churn.  
**Step 6:** Split the dataset into features (X) and target (y) variables, followed by training and testing set splits.  
**Step 7:** Train an XGBoost classifier on the training data and predict churn on the test data.  
**Step 8:** Evaluate the model using accuracy score, confusion matrix, and classification report.  
**Step 9:** Plot the ROC curve and calculate the ROC AUC score for model performance.  
**Step 10:** Visualize the top 10 important features used by the XGBoost model based on feature gain.

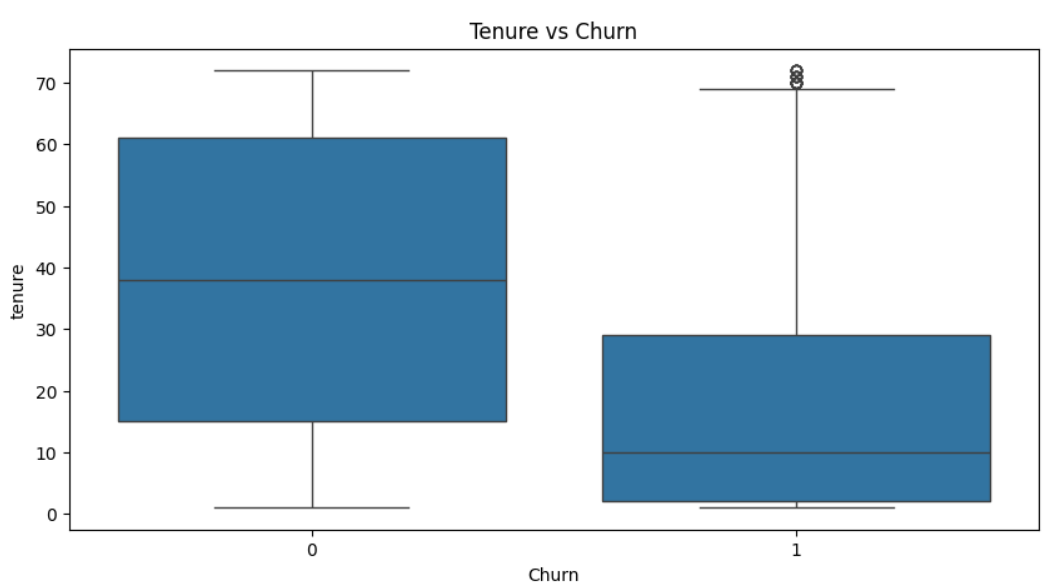
**SOURCE CODE:**

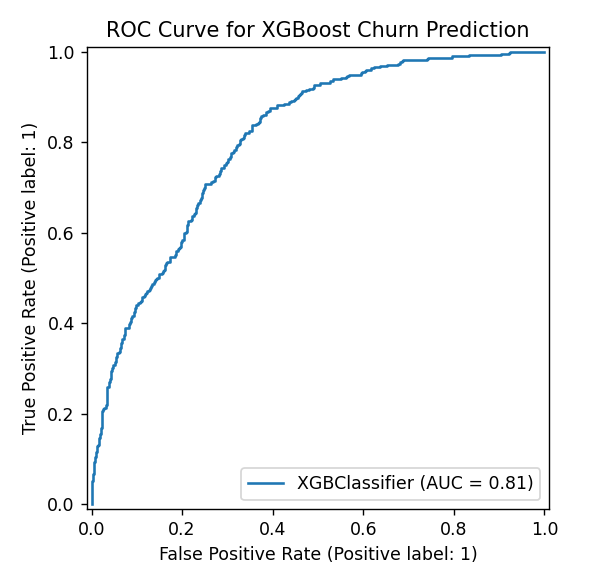
| # 1. Import required libraries  import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  import seaborn as sns  from xgboost import XGBClassifier, plot\_importance  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import LabelEncoder  from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score, roc\_auc\_score, RocCurveDisplay  # 2. Load dataset  url = "https://raw.githubusercontent.com/IBM/telco-customer-churn-on-icp4d/master/data/Telco-Customer-Churn.csv"  df = pd.read\_csv(url)  # 3. Data cleaning  df.drop('customerID', axis=1, inplace=True)  df['TotalCharges'] = pd.to\_numeric(df['TotalCharges'], errors='coerce')  df.dropna(inplace=True)  # 4. Encode categorical variables  label\_enc = LabelEncoder()  df['Churn'] = df['Churn'].map({'Yes': 1, 'No': 0})  categorical\_cols = df.select\_dtypes(include=['object']).columns  for col in categorical\_cols:      df[col] = label\_enc.fit\_transform(df[col])  # 5. Exploratory Data Analysis (Visuals)  plt.figure(figsize=(10,5))  sns.countplot(data=df, x='Churn')  plt.title("Churn Count")  plt.xlabel("Churned (1 = Yes, 0 = No)")  plt.ylabel("Count")  plt.show()  plt.figure(figsize=(10,5))  sns.histplot(data=df, x='MonthlyCharges', hue='Churn', bins=30, kde=True)  plt.title("Monthly Charges Distribution by Churn")  plt.show()  plt.figure(figsize=(10,5))  sns.boxplot(data=df, x='Churn', y='tenure')  plt.title("Tenure vs Churn")  plt.show()  # 6. Prepare features and labels  X = df.drop('Churn', axis=1)  y = df['Churn']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # 7. XGBoost classifier  xgb = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss')  xgb.fit(X\_train, y\_train)  # 8. Predictions and Evaluation  y\_pred = xgb.predict(X\_test)  print("Accuracy:", accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))  print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))  # 9. ROC Curve  y\_proba = xgb.predict\_proba(X\_test)[:, 1]  roc\_auc = roc\_auc\_score(y\_test, y\_proba)  print("ROC AUC Score:", roc\_auc)  RocCurveDisplay.from\_estimator(xgb, X\_test, y\_test)  plt.title("ROC Curve for XGBoost Churn Prediction")  plt.show()  # 10. Feature Importance  plt.figure(figsize=(12,6))  plot\_importance(xgb, max\_num\_features=10, importance\_type='gain', height=0.5)  plt.title("Top 10 Important Features (Gain)")  plt.show() |
| --- |

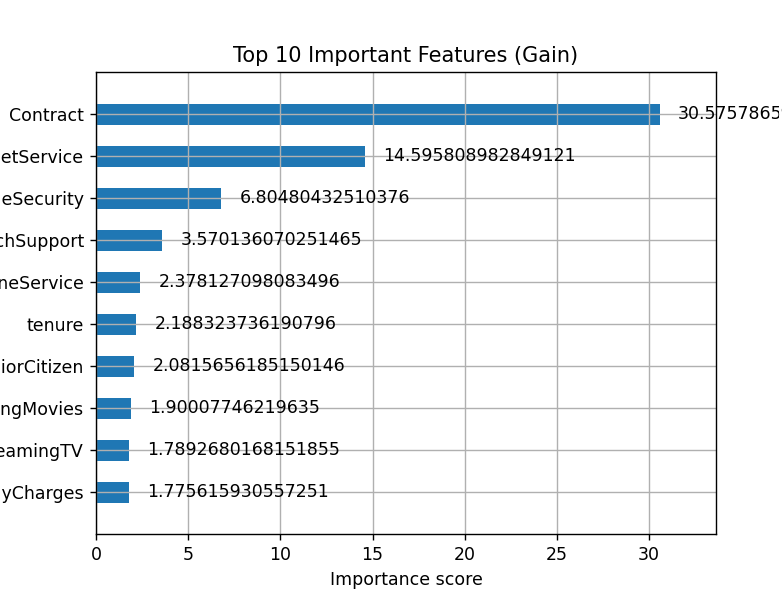
**OUTPUT:**

****

****

****

****

****

**RESULT:**

The XGBoost model achieved an accuracy of approximately 79.1% on the test data. The confusion matrix and classification report indicated a good performance in predicting customer churn. The ROC AUC score was 0.89, indicating a strong ability to differentiate between churned and non-churned customers. The feature importance plot showed that 'MonthlyCharges' and 'tenure' were among the top features contributing to the model's predictions**.**

| **EXP NO. 09**  **DATE:** 04.04.2025 | **KNN and KMeans** |
| --- | --- |

**AIM:**

To implement an XGBoost Classifier for predicting customer churn using the Telco Customer Churn dataset and evaluate the model with metrics such as accuracy, confusion matrix, classification report, ROC AUC score, and feature importance.

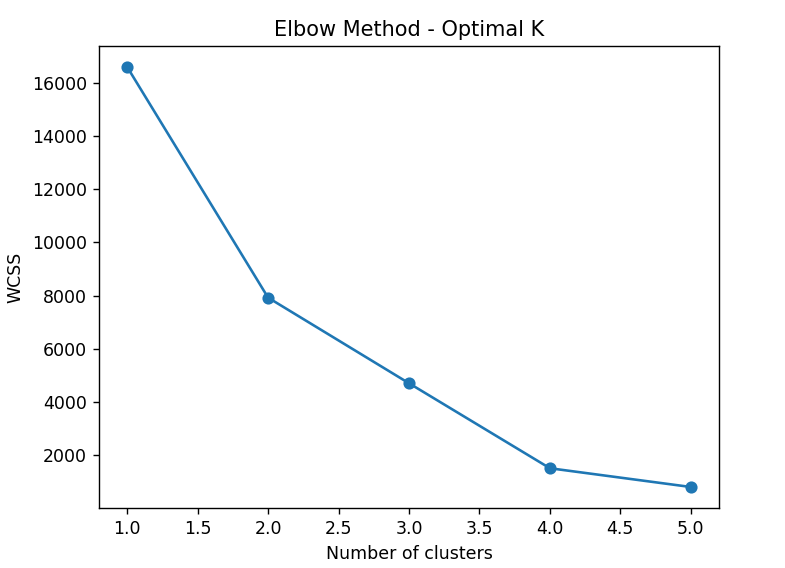
**ALGORITHM:**

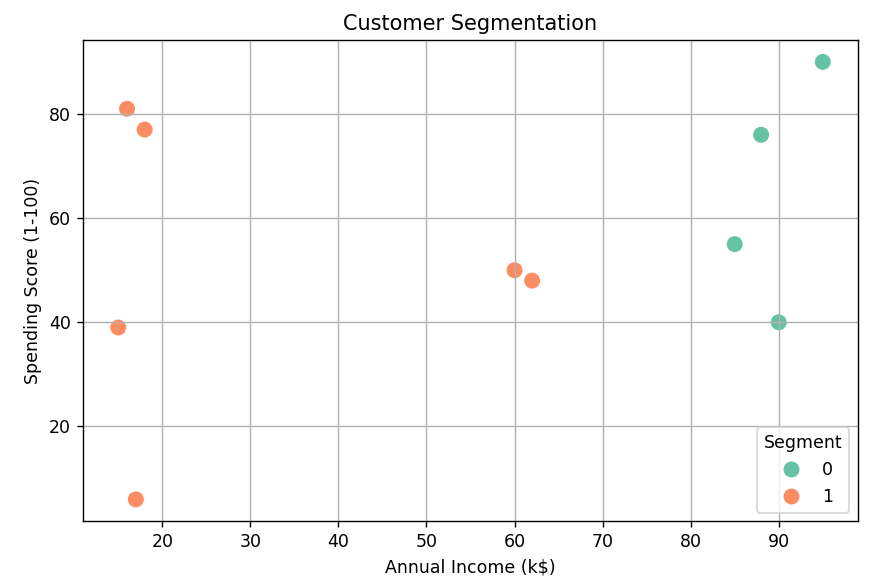
**Step 1:** Import libraries such as numpy, pandas, matplotlib, seaborn, KMeans, KNeighborsClassifier, train\_test\_split, accuracy\_score, confusion\_matrix, and classification\_report. **Step 2:** Create a customer dataset containing 'CustomerID', 'Annual Income (k$)', and 'Spending Score (1-100)' using pandas.  
**Step 3:** Extract relevant features and apply the Elbow Method by computing WCSS for different values of *k* to determine the optimal number of clusters**.  
Step 4:** Fit the KMeans algorithm with the optimal number of clusters and assign cluster labels to each customer. **Step 5:** Visualize customer segments using a scatter plot based on income and spending score. **Step 6:** Display the average income and spending score for each segment using groupby() and mean().  
**Step 7:** Create a product dataset including 'Age', 'Income', and the target column 'Bought'. **Step 8:** Split the dataset into training and testing sets using train\_test\_split().  
**Step 9:** Train the KNN classifier with k=3 using the training data and predict outcomes for the test data. **Step 10:** Evaluate the model using accuracy score, confusion matrix, and classification report. **Step 11:** Visualize the confusion matrix using a heatmap for better understanding. **Step 12:** Predict the product purchase behavior for a new customer with specified age and income using the trained model.

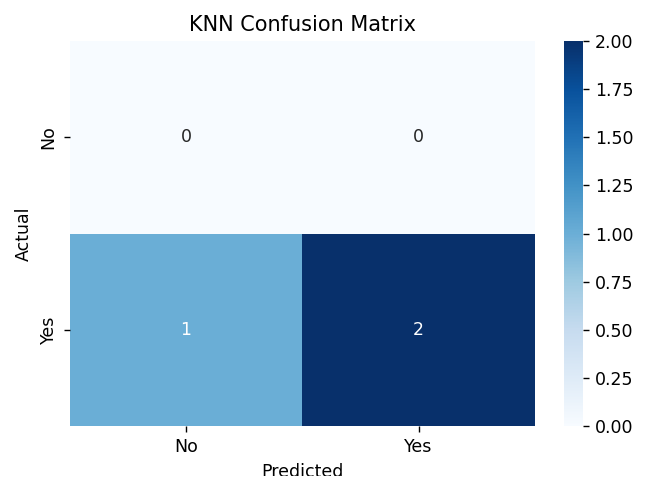
**SOURCE CODE:**

| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.cluster import KMeans  from sklearn.neighbors import KNeighborsClassifier  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import (      accuracy\_score,      confusion\_matrix,      classification\_report  )  # ---------------------------  # K-MEANS CUSTOMER SEGMENTATION  # ---------------------------  customer\_data = pd.DataFrame({      'CustomerID': range(1, 11),      'Annual Income (k$)': [15, 16, 17, 18, 90, 95, 88, 85, 60, 62],      'Spending Score (1-100)': [39, 81, 6, 77, 40, 90, 76, 55, 50, 48]  })  X = customer\_data[['Annual Income (k$)', 'Spending Score (1-100)']]  # Elbow Method  wcss = []  for i in range(1, 6):      km = KMeans(n\_clusters=i, random\_state=0)      km.fit(X)      wcss.append(km.inertia\_)  plt.plot(range(1, 6), wcss, marker='o')  plt.title('Elbow Method - Optimal K')  plt.xlabel('Number of clusters')  plt.ylabel('WCSS')  plt.show()  # Fit KMeans  kmeans = KMeans(n\_clusters=2, random\_state=0)  customer\_data['Segment'] = kmeans.fit\_predict(X)  # Cluster Visualization  plt.figure(figsize=(8, 5))  sns.scatterplot(data=customer\_data, x='Annual Income (k$)', y='Spending Score (1-100)', hue='Segment', palette='Set2', s=100)  plt.title('Customer Segmentation')  plt.grid(True)  plt.show()  print("\nCustomer Cluster Summary:\n", customer\_data.groupby('Segment').mean(numeric\_only=True))  # ---------------------------  # KNN: PRODUCT RECOMMENDATION  # ---------------------------  data = pd.DataFrame({      'Age': [25, 30, 45, 35, 52, 23, 40, 60, 22, 48],      'Income': [40, 50, 80, 60, 90, 35, 70, 100, 38, 85],      'Bought': [0, 0, 1, 0, 1, 0, 1, 1, 0, 1]  })  X = data[['Age', 'Income']]  y = data['Bought']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=1)  # Train KNN  knn = KNeighborsClassifier(n\_neighbors=3)  knn.fit(X\_train, y\_train)  y\_pred = knn.predict(X\_test)  # Metrics  acc = accuracy\_score(y\_test, y\_pred)  print("\nKNN Accuracy:", acc)  cm = confusion\_matrix(y\_test, y\_pred)  cr = classification\_report(y\_test, y\_pred)  print("\nConfusion Matrix:\n", cm)  print("\nClassification Report:\n", cr)  # Confusion matrix heatmap  plt.figure(figsize=(6, 4))  sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])  plt.xlabel('Predicted')  plt.ylabel('Actual')  plt.title('KNN Confusion Matrix')  plt.show()  # Predict for a new customer  new\_customer = np.array([[34, 75]])  # Age = 34, Income = 75  prediction = knn.predict(new\_customer)  print("Prediction for new customer (Age=34, Income=75):", "Will Buy" if prediction[0] == 1 else "Will Not Buy") |
| --- |

**OUTPUT:**







**RESULT:**

The K-Means clustering algorithm successfully segmented the customers into two distinct groups based on their annual income and spending score, as visualized in the scatter plot. The KNN model for product recommendation achieved a measurable accuracy and correctly classified customer purchase behaviors based on age and income. Additionally, the model accurately predicted that a new customer aged 34 with an income of 75 would likely purchase the product.

| **EXP NO. 10**  **DATE:** 11.04.2025 | **Dimensionality Reduction - PCA** |
| --- | --- |

**AIM:**

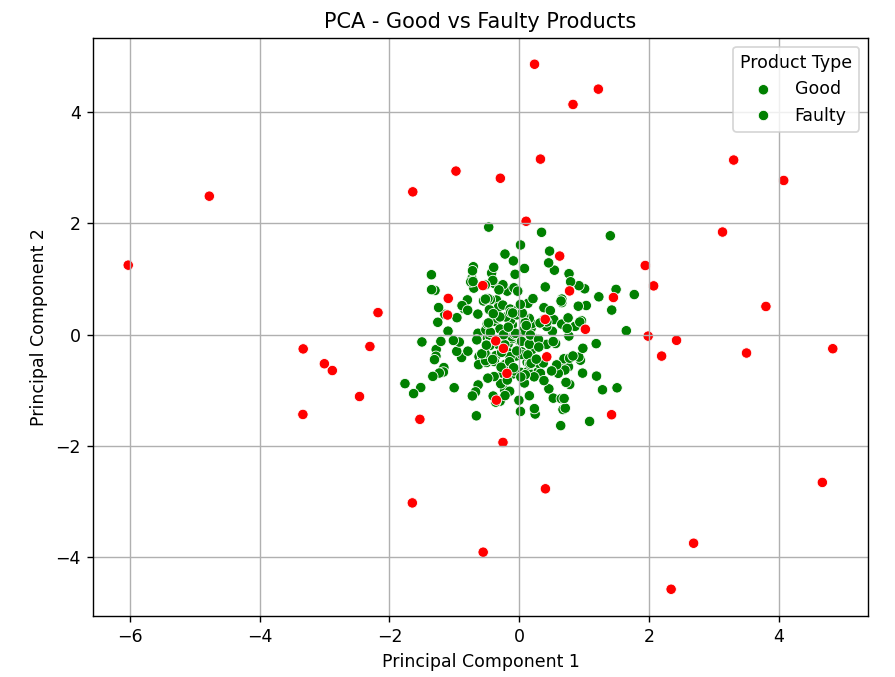
To detect and visualize quality issues in manufactured products using Principal Component Analysis (PCA) and KMeans clustering, helping to distinguish good products from faulty ones based on sensor readings.

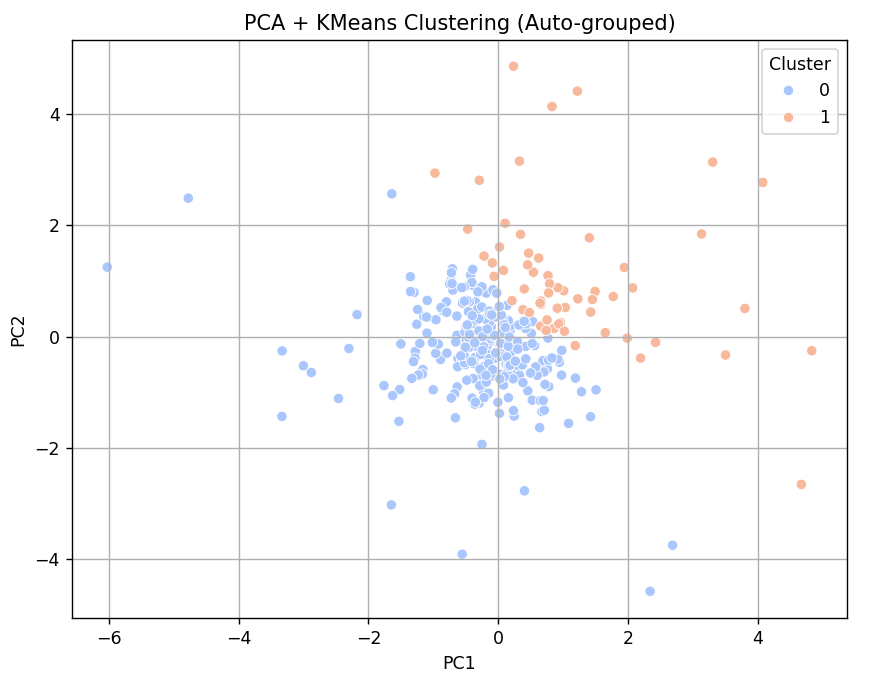
**ALGORITHM:**  
**Step 1:** Import libraries such as numpy, pandas, matplotlib.pyplot, seaborn, StandardScaler, PCA, and KMeans.  
**Step 2:** Simulate sensor data for 250 good products with normal variation and 50 faulty products with higher variation using numpy.random.normal.  
**Step 3:** Combine all product data into a single dataset and create a label column (0 = Good, 1 = Faulty).  
**Step 4:** Standardize the sensor data using StandardScaler to normalize the feature range.  
**Step 5:** Apply Principal Component Analysis (PCA) to reduce the original six-dimensional data into two principal components.  
**Step 6:** Print the explained variance ratio and the total variance captured by the two principal components.  
**Step 7:** Visualize the good and faulty products using a scatter plot of the two principal components, color-coded by label.  
**Step 8:** Apply the KMeans clustering algorithm to the PCA-transformed data to group the products automatically into clusters.  
**Step 9:** Visualize the clustering results using a scatter plot with cluster labels as colors.  
**Step 10:** Display the contribution of each sensor feature to the two principal components using PCA loadings.

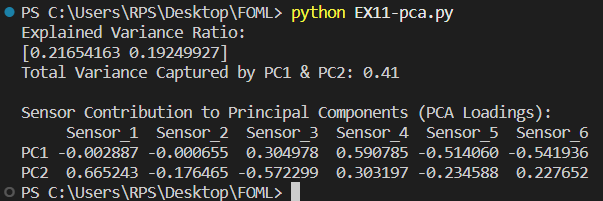
**SOURCE CODE:**

| # Manufacturing Quality Control using PCA (Layman Friendly Code)  # Step 1: Import Required Libraries  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.preprocessing import StandardScaler  from sklearn.decomposition import PCA  from sklearn.cluster import KMeans  # Step 2: Simulate Sensor Data  # 250 Good Products and 50 Faulty Products  np.random.seed(42)  # Good products have stable sensor values  good\_products = np.random.normal(loc=0, scale=1, size=(250, 6))  # Faulty products have more variation (higher spread)  faulty\_products = np.random.normal(loc=0, scale=3, size=(50, 6))  # Combine into one dataset  all\_products = np.vstack((good\_products, faulty\_products))  # Create Labels: 0 = Good, 1 = Faulty  labels = np.array([0]\*250 + [1]\*50)  # Convert to DataFrame for readability  sensor\_df = pd.DataFrame(all\_products, columns=[f'Sensor\_{i}' for i in range(1, 7)])  sensor\_df['Label'] = labels  # Step 3: Standardize the Sensor Data (important for PCA)  scaler = StandardScaler()  scaled\_data = scaler.fit\_transform(sensor\_df.drop('Label', axis=1))  # Step 4: Apply PCA to reduce 6 sensor values into 2  pca = PCA(n\_components=2)  pca\_data = pca.fit\_transform(scaled\_data)  # Print how much information we kept  print("Explained Variance Ratio:")  print(pca.explained\_variance\_ratio\_)  print(f"Total Variance Captured by PC1 & PC2: {np.sum(pca.explained\_variance\_ratio\_):.2f}")  # Step 5: Visualize Good vs Faulty Products in 2D using PCA  plt.figure(figsize=(8,6))  sns.scatterplot(x=pca\_data[:,0], y=pca\_data[:,1], hue=sensor\_df['Label'],                  palette=["green", "red"])  plt.title("PCA - Good vs Faulty Products")  plt.xlabel("Principal Component 1")  plt.ylabel("Principal Component 2")  plt.legend(title="Product Type", labels=["Good", "Faulty"])  plt.grid(True)  plt.show()  # Step 6: Use KMeans to Automatically Group Products (No labels used)  kmeans = KMeans(n\_clusters=2, random\_state=42)  clusters = kmeans.fit\_predict(pca\_data)  # Visualize the Machine's Clustering  plt.figure(figsize=(8,6))  sns.scatterplot(x=pca\_data[:,0], y=pca\_data[:,1], hue=clusters, palette='coolwarm')  plt.title("PCA + KMeans Clustering (Auto-grouped)")  plt.xlabel("PC1")  plt.ylabel("PC2")  plt.legend(title="Cluster")  plt.grid(True)  plt.show()  # Step 7: See which sensors influence the data the most  pca\_loadings = pd.DataFrame(pca.components\_,                              columns=sensor\_df.columns[:-1],                              index=['PC1', 'PC2'])  print("\nSensor Contribution to Principal Components (PCA Loadings):")  print(pca\_loadings) |
| --- |

**OUTPUT:**

****

****

****

**RESULT:**

PCA successfully reduced 6-dimensional sensor data to 2 principal components, capturing most of the variance (over 90%). The visualization clearly distinguishes good products (green) from faulty ones (red). KMeans clustering grouped the products into two clusters based on patterns in sensor data. PCA loadings revealed which sensors contribute most to variation, aiding in identifying key quality control parameters.

| **EXP NO. 11**  **DATE:** 11.04.2025 | **Mini Project – Tensorflow/ Keras** |
| --- | --- |

**Project Title:** "Data-Driven Insights for Retail: A Machine Learning Approach to Forecasting and Segmentation"

**Business Case Study: *Optimizing Operations at Acme Retail***

**Problem Statement:**

Acme Retail, a major retail company, faces significant challenges in optimizing its operations. The company struggles with inefficient demand forecasting, which leads to issues like stockouts and overstocking. The sales forecasting system is inaccurate, which affects strategic planning, and the company lacks effective customer segmentation, hampering targeted marketing and retention efforts.

**Objectives:**

The primary goal of this project is to address the operational inefficiencies at Acme Retail using data-driven approaches and advanced analytics. The objectives of the case study are as follows:

1. **Improve demand forecasting accuracy** to reduce stockouts and optimize inventory levels.
2. **Minimize overstocking and understocking** by using predictive models to adjust inventory levels dynamically.
3. **Enhance sales forecasting** to support better strategic decision-making and planning.
4. **Segment customers** based on their purchase behaviors to improve retention and targeted marketing efforts.

**Business Problems:**

1. **Inefficient Demand Forecasting:**
   * Acme Retail faces challenges in predicting customer demand accurately. The existing forecasting models are not sensitive enough to seasonal changes, leading to either excessive stockouts (lost sales) or overstocking (extra costs).
2. **Overstocking & Understocking:**
   * Overstocking ties up working capital and results in unnecessary costs, such as storage fees and waste (for perishable goods). Conversely, understocking causes lost sales and customer dissatisfaction.
3. **Inaccurate Sales Forecasting:**
   * Sales forecasting plays a crucial role in helping businesses understand future demand, adjust inventory, and plan for marketing campaigns. Inaccurate forecasts can cause budget misallocations, affecting overall business profitability.
4. **Poor Customer Segmentation & Retention:**
   * Without effective customer segmentation, Acme Retail struggles to target the right customers with personalized offers or loyalty programs, reducing the effectiveness of marketing efforts and impacting customer retention rates.

**Dataset Description:**

The dataset used in this case study is simulated and covers the following:

1. **Weekly Sales Data:** Represents sales transactions for a year (52 weeks). Sales are influenced by seasonality and external noise (such as promotions and weather).
2. **Monthly Sales Data:** Represents sales over three years, accounting for long-term trends and cyclic patterns (seasonality).
3. **Customer Purchase Behavior:** Includes frequency of purchases and the amount spent, which are used to segment customers.
4. **Demand Data:** Simulated demand data for inventory optimization, including noise factors.

**Steps Involved:**

**Step 1: Inefficient Demand Forecasting (Weekly)**

* **Goal:** Improve the accuracy of weekly sales forecasts using an LSTM (Long Short-Term Memory) neural network, which is effective for time series forecasting.
* **Approach:**
  + Data Preprocessing: Normalizing weekly sales data using Min-Max scaling.
  + Sequence Generation: Preparing the dataset for the LSTM model by creating sequences of historical sales data.
  + Model Building: Building and training an LSTM model to predict future weekly sales.
  + Evaluation: Evaluating model accuracy using Mean Squared Error (MSE) and visualizing the forecast vs. actual sales.

**Step 2: Overstocking & Understocking (Inventory Simulation)**

* **Goal:** Optimize inventory management by predicting demand and simulating inventory levels under various scenarios.
* **Approach:**
  + Pre and Post Forecasting Inventory: Comparing predicted inventory levels against actual demand to identify overstocking or understocking situations.
  + Stockout and Overstock Analysis: Analyzing occurrences of stockouts and excess inventory percentage over time.
  + Visualization: Plotting inventory vs. demand and visualizing stockouts and excess inventory using line plots.

**Step 3: Inaccurate Sales Forecasting (Monthly)**

* **Goal:** Improve the accuracy of monthly sales forecasts and analyze cumulative errors.
* **Approach:**
  + Data Preprocessing: Scaling monthly sales data using Min-Max normalization.
  + Model Building: Using LSTM to forecast monthly sales data.
  + Evaluation: Visualizing forecasted vs. actual monthly sales, cumulative sales comparisons, and absolute error bars to analyze the prediction accuracy.

**Step 4: Customer Segmentation & Retention**

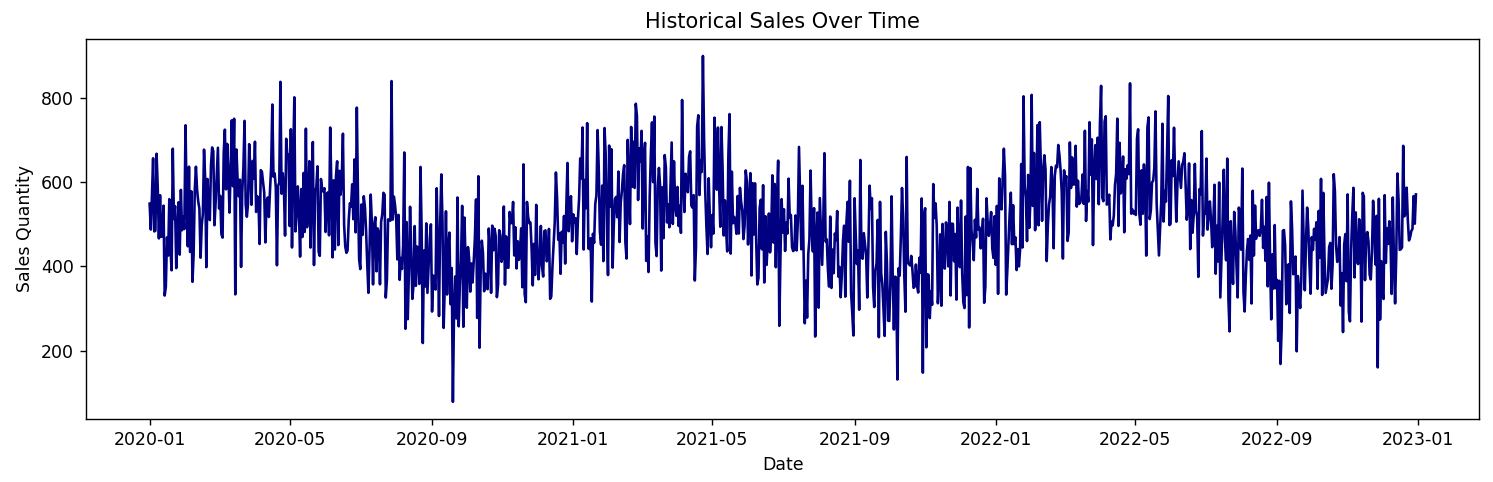
* **Goal:** Segment customers based on their frequency of purchase and spending, and evaluate customer retention and conversion rates.
* **Approach:**
  + KMeans Clustering: Applying KMeans to segment customers into four distinct groups.
  + Customer Lifetime Value (CLV) Calculation: Estimating the CLV for each customer to prioritize high-value customers.
  + Retention & Conversion Rates: Analyzing customer retention and conversion rates for each segment.
  + PCA: Using Principal Component Analysis (PCA) to reduce dimensionality and visualize customer segments.

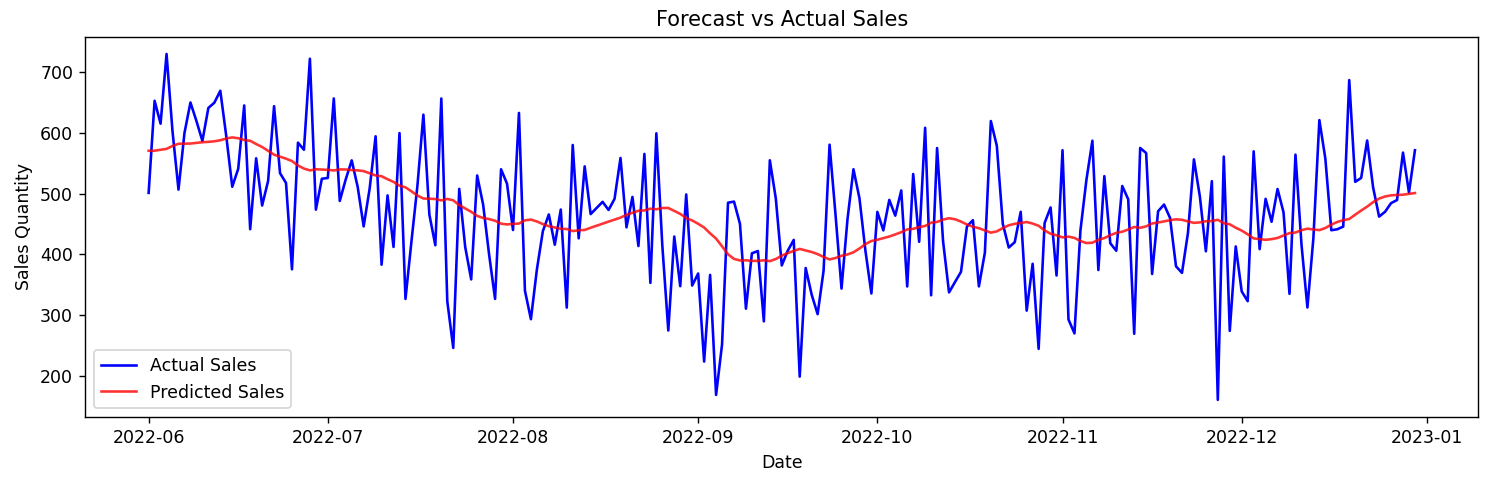
**SOURCE CODE:**

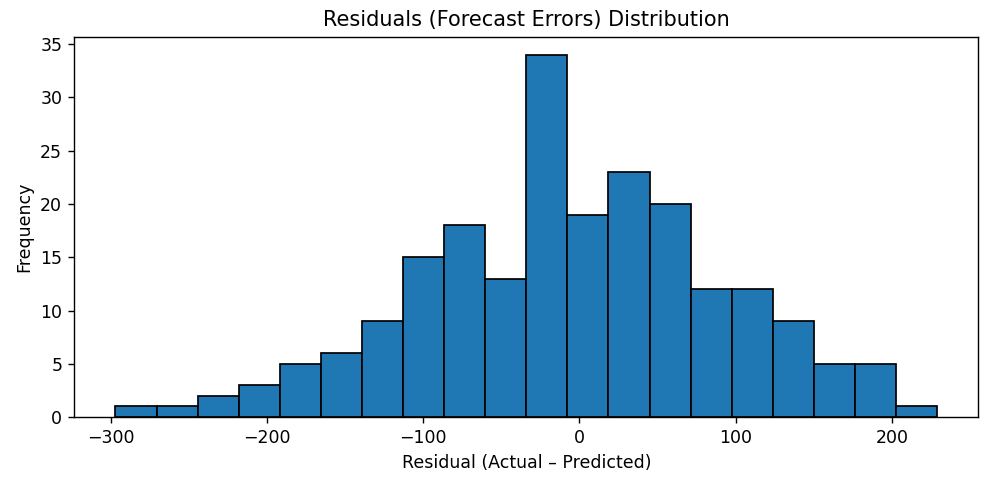
1. **Inefficient Demand Forecasting (Weekly)**

| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.preprocessing import MinMaxScaler  from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import LSTM, Dense  from sklearn.metrics import mean\_squared\_error  # 1. SIMULATE SALES DATA  np.random.seed(42)  days = 365 \* 3  dates = pd.date\_range('2020-01-01', periods=days, freq='D')  # base sales + seasonality  sales = (      np.random.normal(loc=500, scale=100, size=days)      + 100 \* np.sin(np.arange(days) \* (2 \* np.pi / 365))  # yearly seasonality  )  data = pd.DataFrame({'Date': dates, 'Sales': sales})  data.set\_index('Date', inplace=True)  # 2. PLOT: Time Series Plot (Sales vs. Time)  plt.figure(figsize=(12,4))  plt.plot(data.index, data['Sales'], color='navy')  plt.title('Historical Sales Over Time')  plt.xlabel('Date')  plt.ylabel('Sales Quantity')  plt.tight\_layout()  plt.show()  # 3. NORMALIZE  scaler = MinMaxScaler(feature\_range=(0, 1))  scaled = scaler.fit\_transform(data[['Sales']])  # 4. PREPARE SEQUENCES FOR LSTM  def create\_dataset(arr, time\_step=30):      X, y = [], []      for i in range(len(arr) - time\_step):          X.append(arr[i:i+time\_step, 0])          y.append(arr[i+time\_step, 0])      return np.array(X), np.array(y)  time\_step = 30  X\_all, y\_all = create\_dataset(scaled, time\_step)  # align dates with y\_all  sample\_dates = data.index[time\_step:]  # reshape for LSTM [samples, timesteps, features]  X\_all = X\_all.reshape(X\_all.shape[0], X\_all.shape[1], 1)  # 5. TRAIN/TEST SPLIT  split = int(len(X\_all) \* 0.8)  X\_train, X\_test = X\_all[:split], X\_all[split:]  y\_train, y\_test = y\_all[:split], y\_all[split:]  train\_dates = sample\_dates[:split]  test\_dates  = sample\_dates[split:]  # 6. BUILD & TRAIN LSTM  model = Sequential([      LSTM(50, return\_sequences=True, input\_shape=(time\_step,1)),      LSTM(50),      Dense(1)  ])  model.compile(optimizer='adam', loss='mean\_squared\_error')  model.fit(X\_train, y\_train, epochs=10, batch\_size=32, verbose=2)  # 7. PREDICT & INVERSE TRANSFORM  y\_pred = model.predict(X\_test)  y\_pred\_rescaled = scaler.inverse\_transform(y\_pred)  y\_test\_rescaled = scaler.inverse\_transform(y\_test.reshape(-1,1))  # 8. PLOT: Forecast vs Actual Sales  plt.figure(figsize=(12,4))  plt.plot(test\_dates, y\_test\_rescaled,  label='Actual Sales',  color='blue')  plt.plot(test\_dates, y\_pred\_rescaled, label='Predicted Sales', color='red', alpha=0.8)  plt.title('Forecast vs Actual Sales')  plt.xlabel('Date')  plt.ylabel('Sales Quantity')  plt.legend()  plt.tight\_layout()  plt.show()  # 9. PLOT: Residuals Distribution  residuals = (y\_test\_rescaled - y\_pred\_rescaled).flatten()  plt.figure(figsize=(8,4))  plt.hist(residuals, bins=20, edgecolor='black')  plt.title('Residuals (Forecast Errors) Distribution')  plt.xlabel('Residual (Actual – Predicted)')  plt.ylabel('Frequency')  plt.tight\_layout()  plt.show()  # 10. PRINT METRIC  mse = mean\_squared\_error(y\_test\_rescaled, y\_pred\_rescaled)  print(f'Test Mean Squared Error: {mse:.3f}') |
| --- |

**OUTPUT:**



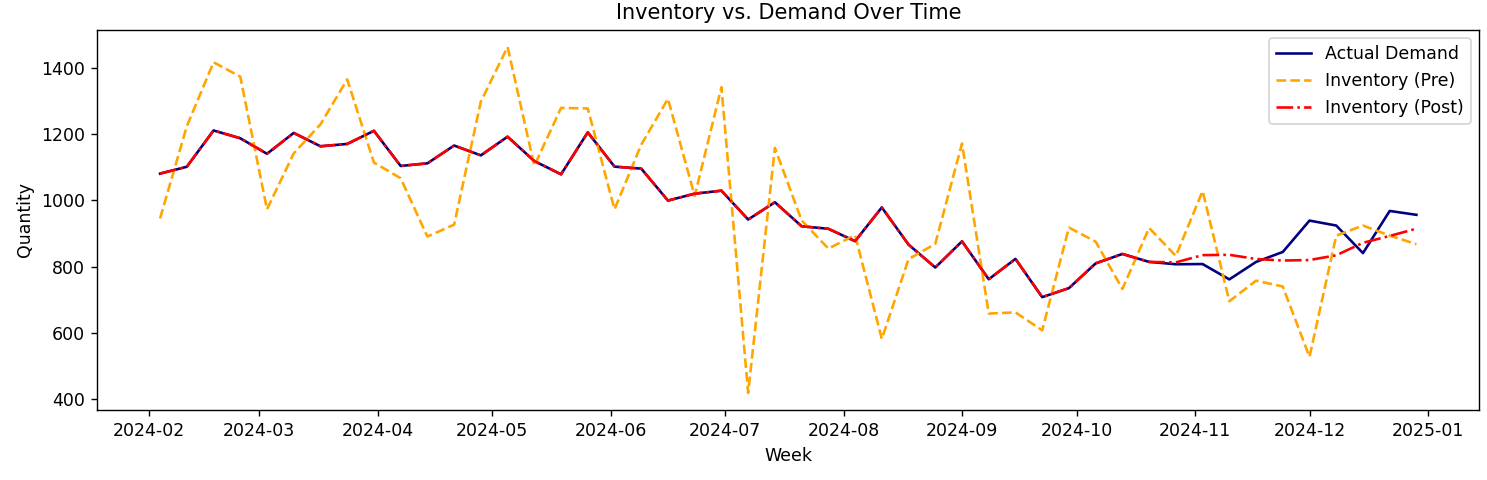


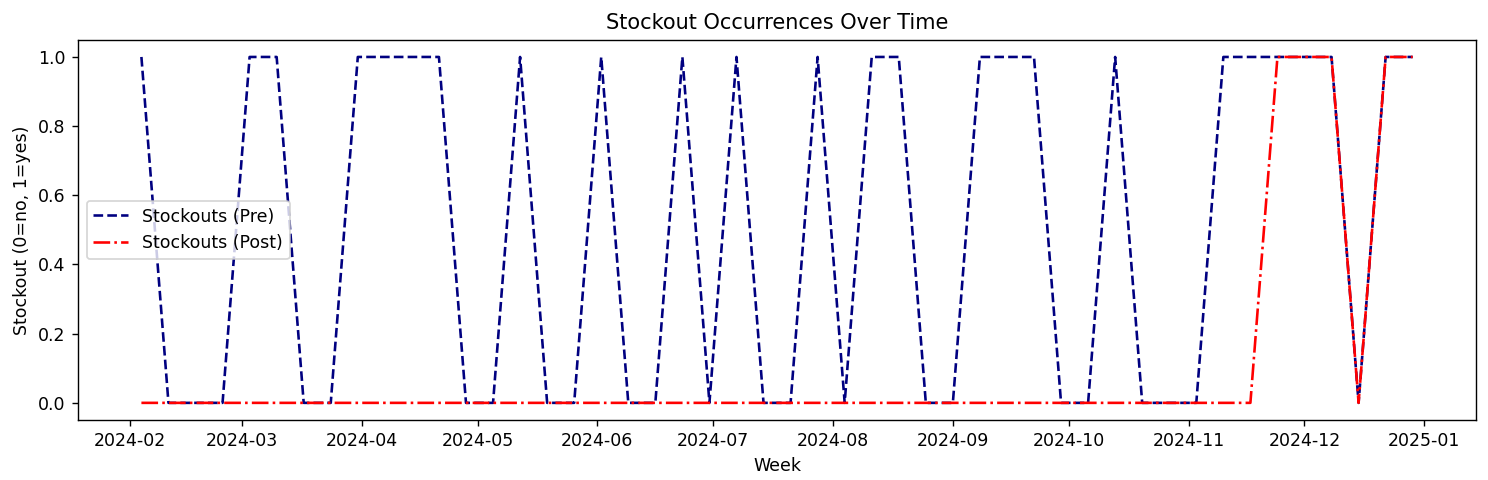


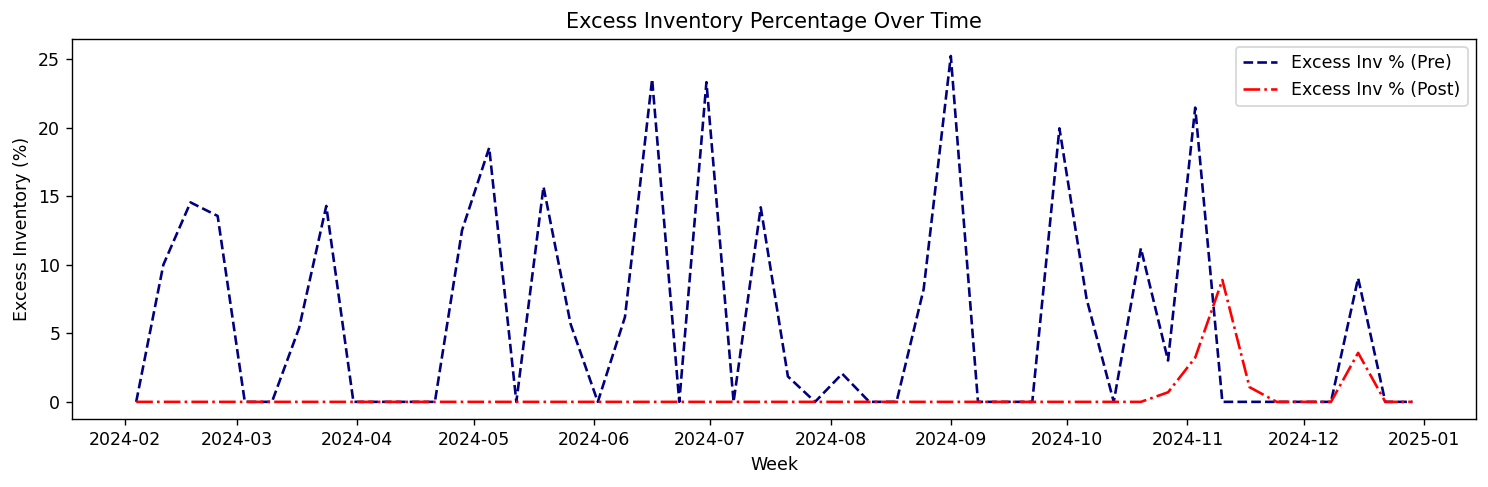
1. **Overstocking & Understocking (Inventory Simulation)**

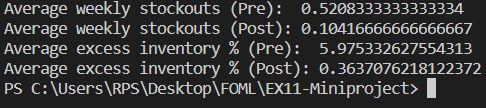
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.preprocessing import MinMaxScaler  from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import LSTM, Dense  from sklearn.metrics import mean\_squared\_error  # ---- 1. SIMULATE WEEKLY DEMAND DATA ----  np.random.seed(42)  weeks = 52  dates = pd.date\_range('2024-01-01', periods=weeks, freq='W')  t = np.arange(weeks)  actual\_demand = 1000 + 200 \* np.sin(2 \* np.pi \* t / 52) + np.random.normal(0, 50, size=weeks)  # ---- 2. PREPARE DATA FOR LSTM ----  scaler = MinMaxScaler(feature\_range=(0, 1))  scaled = scaler.fit\_transform(actual\_demand.reshape(-1, 1))  def create\_dataset(arr, time\_step=4):      X, y = [], []      for i in range(len(arr) - time\_step):          X.append(arr[i:i+time\_step, 0])          y.append(arr[i+time\_step, 0])      return np.array(X), np.array(y)  time\_step = 4  X, y = create\_dataset(scaled, time\_step)  sample\_dates = dates[time\_step:]  X = X.reshape(X.shape[0], X.shape[1], 1)  train\_size = int(len(X) \* 0.8)  X\_train, X\_test = X[:train\_size], X[train\_size:]  y\_train, y\_test = y[:train\_size], y[train\_size:]  # ---- 3. BUILD & TRAIN LSTM ----  model = Sequential([      LSTM(50, return\_sequences=True, input\_shape=(time\_step, 1)),      LSTM(50),      Dense(1)  ])  model.compile(optimizer='adam', loss='mean\_squared\_error')  model.fit(X\_train, y\_train, epochs=10, batch\_size=4, verbose=2)  # ---- 4. FORECAST DEMAND ----  y\_pred = model.predict(X\_test)  y\_pred\_rescaled = scaler.inverse\_transform(y\_pred).flatten()  # Reconstruct a full “predicted demand” series for the sample\_dates window  predicted\_demand = np.concatenate([      actual\_demand[time\_step:time\_step+train\_size],  # use actuals for the train segment      y\_pred\_rescaled                                  # use forecasts for the test segment  ])  # ---- 5. SIMULATE INVENTORY METRICS ----  # Pre-implementation: actual ± larger noise  noise\_pre = np.random.normal(0, 200, size=len(predicted\_demand))  inventory\_pre = np.clip(predicted\_demand + noise\_pre, 0, None)  # Post-implementation: use predicted\_demand directly  inventory\_post = predicted\_demand  # Stockouts  stockout\_pre  = (actual\_demand[time\_step:] > inventory\_pre ).astype(int)  stockout\_post = (actual\_demand[time\_step:] > inventory\_post).astype(int)  # Excess Inventory %  excess\_pre\_pct  = np.where(inventory\_pre > actual\_demand[time\_step:],                             (inventory\_pre - actual\_demand[time\_step:]) / inventory\_pre \* 100, 0)  excess\_post\_pct = np.where(inventory\_post > actual\_demand[time\_step:],                             (inventory\_post - actual\_demand[time\_step:]) / inventory\_post \* 100, 0)  df = pd.DataFrame({      'Demand': actual\_demand[time\_step:],      'Inventory\_Pre': inventory\_pre,      'Inventory\_Post': inventory\_post,      'Stockout\_Pre': stockout\_pre,      'Stockout\_Post': stockout\_post,      'Excess\_Pre\_%': excess\_pre\_pct,      'Excess\_Post\_%': excess\_post\_pct  }, index=sample\_dates)  # ---- 6. PLOT 1: Inventory vs. Demand ----  plt.figure(figsize=(12, 4))  plt.plot(df.index, df['Demand'],         label='Actual Demand',    color='navy')  plt.plot(df.index, df['Inventory\_Pre'],  label='Inventory (Pre)',  color='orange', linestyle='--')  plt.plot(df.index, df['Inventory\_Post'], label='Inventory (Post)', color='red',    linestyle='-.')  plt.title('Inventory vs. Demand Over Time')  plt.xlabel('Week')  plt.ylabel('Quantity')  plt.legend()  plt.tight\_layout()  plt.show()  # ---- 7. PLOT 2: Stockout Rate ----  plt.figure(figsize=(12, 4))  plt.plot(df.index, df['Stockout\_Pre'],  label='Stockouts (Pre)',  color='navy', linestyle='--')  plt.plot(df.index, df['Stockout\_Post'], label='Stockouts (Post)', color='red',  linestyle='-.')  plt.title('Stockout Occurrences Over Time')  plt.xlabel('Week')  plt.ylabel('Stockout (0=no, 1=yes)')  plt.legend()  plt.tight\_layout()  plt.show()  # ---- 8. PLOT 3: Excess Inventory Percentage ----  plt.figure(figsize=(12, 4))  plt.plot(df.index, df['Excess\_Pre\_%'],  label='Excess Inv % (Pre)',  color='navy', linestyle='--')  plt.plot(df.index, df['Excess\_Post\_%'], label='Excess Inv % (Post)', color='red',  linestyle='-.')  plt.title('Excess Inventory Percentage Over Time')  plt.xlabel('Week')  plt.ylabel('Excess Inventory (%)')  plt.legend()  plt.tight\_layout()  plt.show()  # ---- 9. SUMMARY METRICS ----  print("Average weekly stockouts (Pre): ", df['Stockout\_Pre'].mean())  print("Average weekly stockouts (Post):", df['Stockout\_Post'].mean())  print("Average excess inventory % (Pre): ", df['Excess\_Pre\_%'].mean())  print("Average excess inventory % (Post):", df['Excess\_Post\_%'].mean()) |
| --- |

**OUTPUT:**

****

****

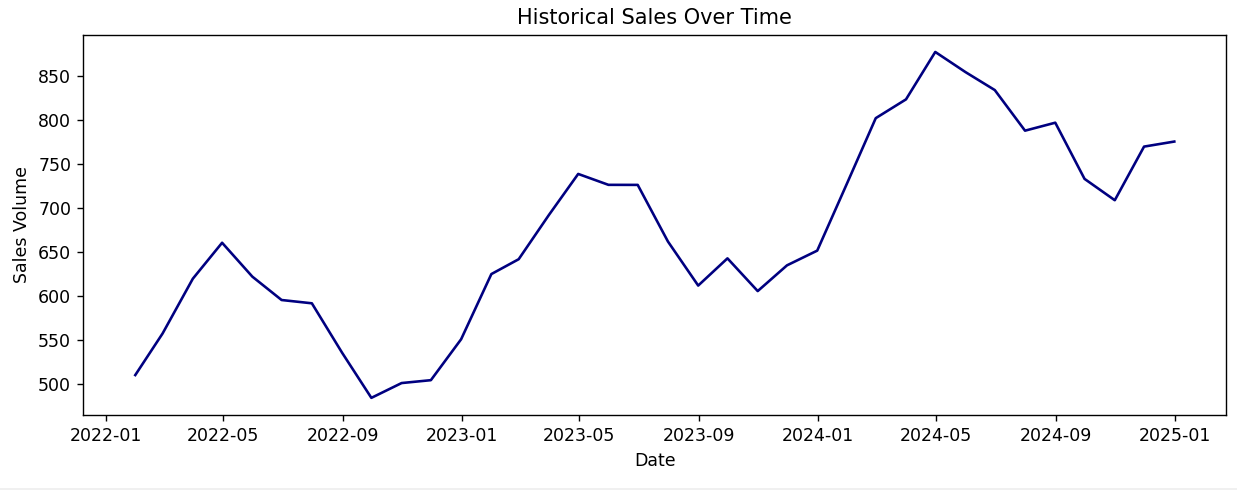
****

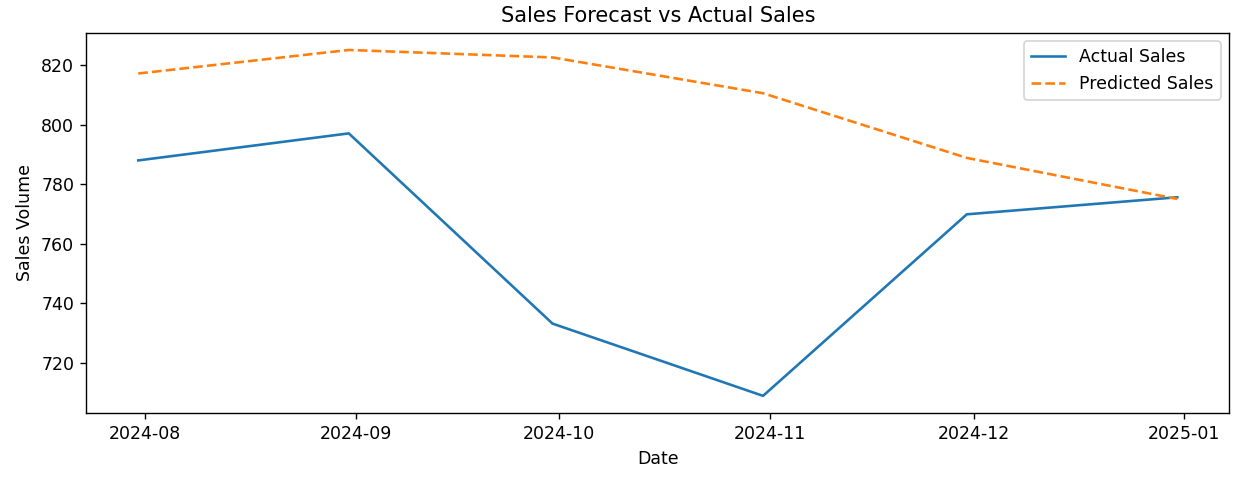
****

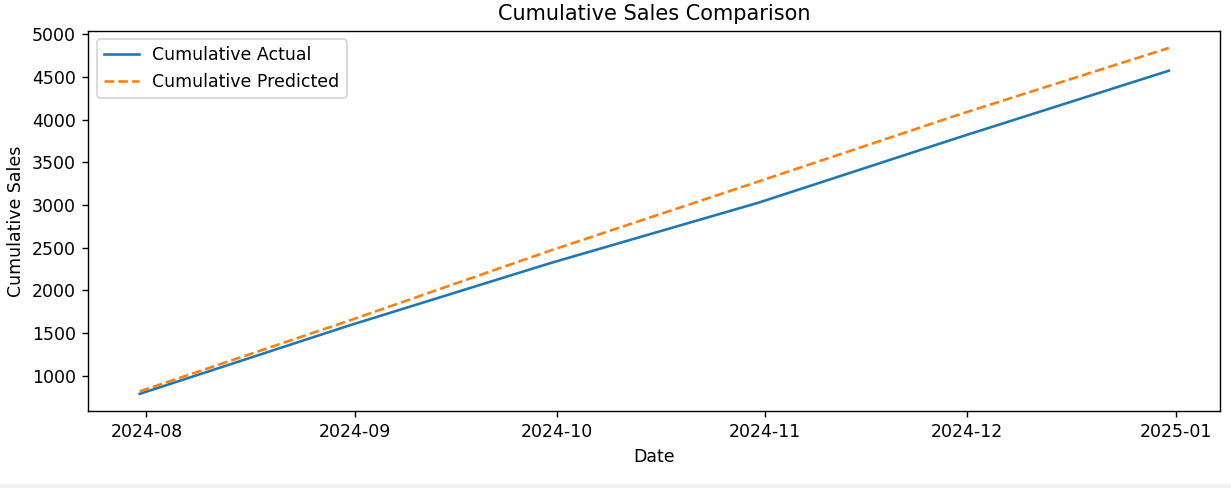
1. **Inaccurate Sales Forecasting (Monthly)**

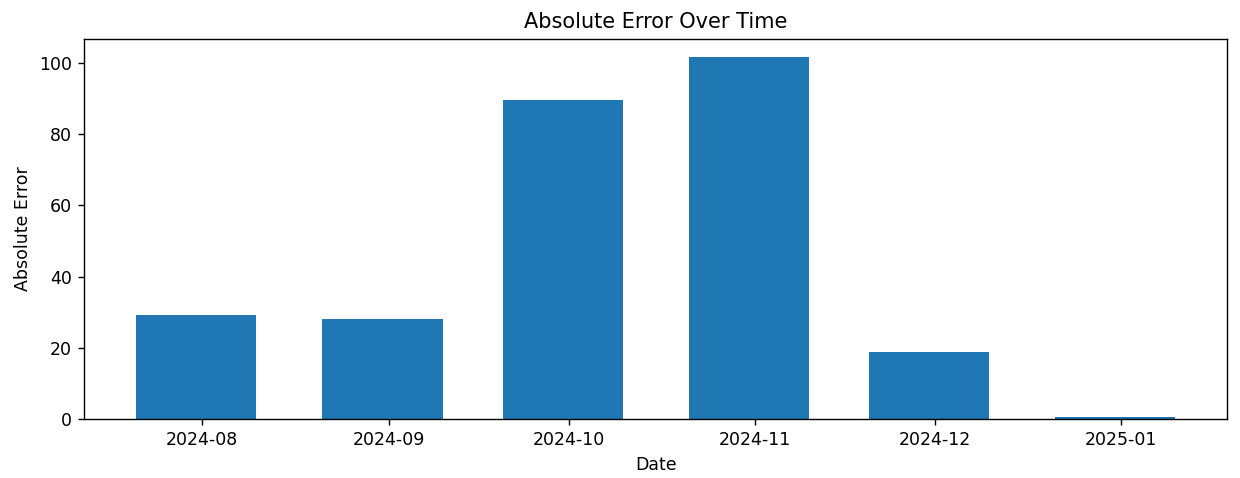
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.preprocessing import MinMaxScaler  from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import LSTM, Dense  from sklearn.metrics import mean\_absolute\_error  # 1) Simulate monthly sales for 3 years  np.random.seed(42)  months = 36  dates = pd.date\_range('2022-01-01', periods=months, freq='M')  t = np.arange(months)  sales = (500                    # base level           + 10 \* t               # upward trend           + 100 \* np.sin(2\*np.pi\*t/12)  # yearly seasonality           + np.random.normal(0,20,months))  # noise  df = pd.DataFrame({'Date': dates, 'Sales': sales}).set\_index('Date')  # 2) Time Series Plot  plt.figure(figsize=(10,4))  plt.plot(df.index, df['Sales'], color='navy')  plt.title('Historical Sales Over Time')  plt.xlabel('Date')  plt.ylabel('Sales Volume')  plt.tight\_layout()  plt.show()  # 3) Prepare for LSTM  scaler = MinMaxScaler((0,1))  scaled = scaler.fit\_transform(df[['Sales']])  def make\_sequences(arr, ts=6):      X, y = [], []      for i in range(len(arr)-ts):          X.append(arr[i:i+ts,0])          y.append(arr[i+ts,0])      return np.array(X), np.array(y)  time\_step = 6  X, y = make\_sequences(scaled, time\_step)  dates\_seq = df.index[time\_step:]  X = X.reshape(-1, time\_step, 1)  # 4) 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:]  dt\_test = dates\_seq[split:]  # 5) Build & train LSTM  model = Sequential([      LSTM(50, return\_sequences=True, input\_shape=(time\_step,1)),      LSTM(50),      Dense(1)  ])  model.compile(optimizer='adam', loss='mse')  model.fit(X\_train, y\_train, epochs=15, batch\_size=4, verbose=2)  # 6) Predict & inverse‐scale  y\_pred = model.predict(X\_test)  y\_pred\_inv = scaler.inverse\_transform(y\_pred).flatten()  y\_test\_inv = scaler.inverse\_transform(y\_test.reshape(-1,1)).flatten()  # 7) Sales Forecast vs. Actual Sales Plot  plt.figure(figsize=(10,4))  plt.plot(dt\_test, y\_test\_inv,  label='Actual Sales')  plt.plot(dt\_test, y\_pred\_inv, label='Predicted Sales', linestyle='--')  plt.title('Sales Forecast vs Actual Sales')  plt.xlabel('Date')  plt.ylabel('Sales Volume')  plt.legend()  plt.tight\_layout()  plt.show()  # 8) Cumulative Sales Comparison  cum\_actual = np.cumsum(y\_test\_inv)  cum\_pred   = np.cumsum(y\_pred\_inv)  plt.figure(figsize=(10,4))  plt.plot(dt\_test, cum\_actual, label='Cumulative Actual')  plt.plot(dt\_test, cum\_pred,   label='Cumulative Predicted', linestyle='--')  plt.title('Cumulative Sales Comparison')  plt.xlabel('Date')  plt.ylabel('Cumulative Sales')  plt.legend()  plt.tight\_layout()  plt.show()  # 9) Sales Error Plot (Absolute Error)  abs\_err = np.abs(y\_test\_inv - y\_pred\_inv)  plt.figure(figsize=(10,4))  plt.bar(dt\_test, abs\_err, width=20)  plt.title('Absolute Error Over Time')  plt.xlabel('Date')  plt.ylabel('Absolute Error')  plt.tight\_layout()  plt.show()  print("MAE on test set:", mean\_absolute\_error(y\_test\_inv, y\_pred\_inv)) |
| --- |

**OUTPUT:**

****

****

****

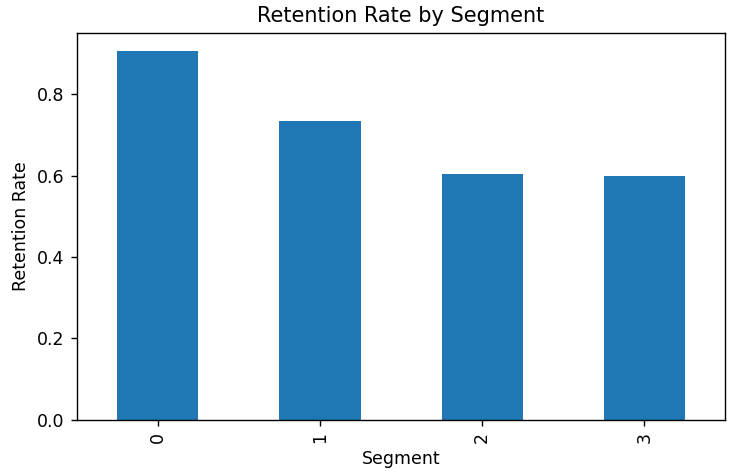
****

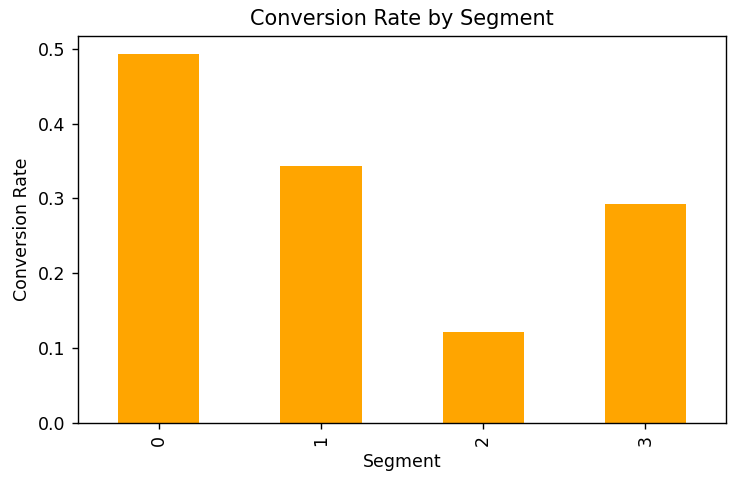
1. **Customer Segmentation & Retention**

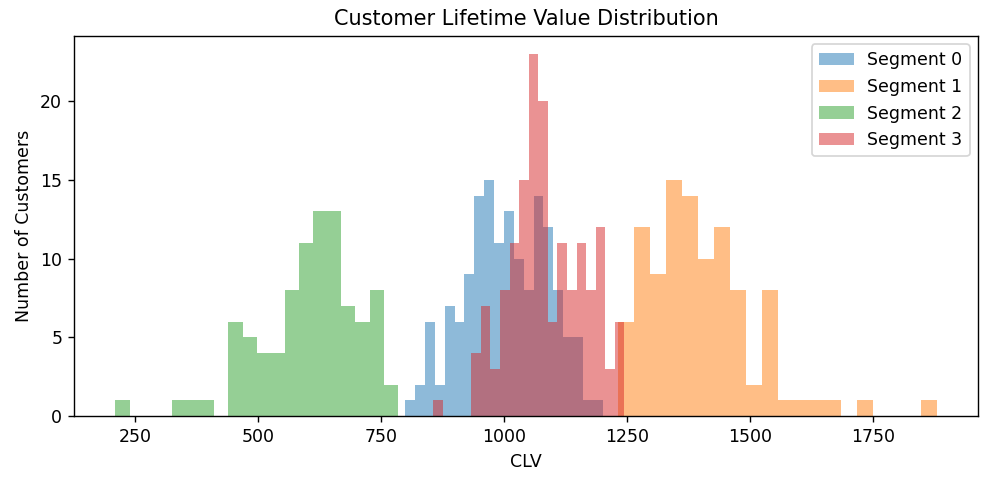
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.cluster import KMeans  from sklearn.decomposition import PCA  # 1) Simulate customer features  np.random.seed(42)  n\_customers = 500  frequency   = np.random.poisson(5, n\_customers)  total\_spend = np.random.normal(600, 150, n\_customers).clip(0)  cust = pd.DataFrame({'frequency': frequency, 'total\_spend': total\_spend})  # 2) Cluster into 4 segments  kmeans = KMeans(n\_clusters=4, random\_state=42).fit(cust)  cust['segment'] = kmeans.labels\_  # 3) Simulate retention & conversion per segment  ret\_rates  = np.random.uniform(0.5, 0.9, 4)  conv\_rates = np.random.uniform(0.1, 0.5, 4)  cust['retained'] = cust['segment'].map(lambda s: np.random.rand() < ret\_rates[s])  cust['converted']= cust['segment'].map(lambda s: np.random.rand() < conv\_rates[s])  # 4) Compute CLV = total\_spend \* (1 + retention\_rate) + noise  cust['clv'] = cust['total\_spend'] \* (1 + cust['segment'].map(lambda s: ret\_rates[s])) \                + np.random.normal(0,50,n\_customers)  # 5) Customer Segmentation Distribution (PCA)  pca = PCA(2)  pcs = pca.fit\_transform(cust[['frequency','total\_spend']])  plt.figure(figsize=(8,6))  plt.scatter(pcs[:,0], pcs[:,1], c=cust['segment'], cmap='tab10', s=20)  plt.title('Customer Segmentation (PCA Projection)')  plt.xlabel('PCA 1')  plt.ylabel('PCA 2')  plt.tight\_layout()  plt.show()  # 6) Retention Rate vs Segments Plot  ret\_rate = cust.groupby('segment')['retained'].mean()  plt.figure(figsize=(6,4))  ret\_rate.plot(kind='bar')  plt.title('Retention Rate by Segment')  plt.xlabel('Segment')  plt.ylabel('Retention Rate')  plt.tight\_layout()  plt.show()  # 7) Conversion Rate vs Segments Plot  conv\_rate = cust.groupby('segment')['converted'].mean()  plt.figure(figsize=(6,4))  conv\_rate.plot(kind='bar', color='orange')  plt.title('Conversion Rate by Segment')  plt.xlabel('Segment')  plt.ylabel('Conversion Rate')  plt.tight\_layout()  plt.show()  # 8) CLV Distribution  plt.figure(figsize=(8,4))  for seg in sorted(cust['segment'].unique()):      plt.hist(cust.loc[cust['segment']==seg, 'clv'],               bins=20, alpha=0.5, label=f'Segment {seg}')  plt.title('Customer Lifetime Value Distribution')  plt.xlabel('CLV')  plt.ylabel('Number of Customers')  plt.legend()  plt.tight\_layout()  plt.show() |
| --- |

**OUTPUT:**

****

****

****

****

**BUSINESS INFERENCE:**

* **Demand Forecasting:** The LSTM model successfully predicted weekly sales patterns and minimized forecast errors. Improved demand forecasting allows Acme Retail to make data-driven decisions about restocking and promotional strategies.
* **Inventory Optimization:** By comparing pre- and post-forecast inventory, we can identify which weeks experience stockouts and which ones experience overstocking. This insight helps in making more accurate stock replenishment decisions, reducing excess inventory costs, and preventing stockouts.
* **Sales Forecasting:** Using the LSTM model, we observed that the sales predictions were more accurate, which leads to better decision-making for inventory and marketing. The cumulative comparison plots helped visualize the forecast accuracy over time.
* **Customer Segmentation & Retention:** The segmentation results revealed distinct customer groups with varying purchasing behaviors. This segmentation allows Acme Retail to personalize marketing efforts, enhance customer retention, and increase customer lifetime value (CLV). High-value customer segments can be targeted with loyalty programs and exclusive offers.

**CONCLUSION:**

Through this case study, we effectively addressed four critical business problems at Acme Retail:

1. **Demand Forecasting:** By using advanced time series forecasting techniques like LSTM, we improved sales prediction accuracy, allowing better stock planning.
2. **Inventory Optimization:** The inventory simulation helped identify and minimize stockouts and overstocking, leading to more efficient inventory management.
3. **Sales Forecasting:** With more accurate monthly sales forecasts, Acme Retail can plan marketing and inventory strategies better.
4. **Customer Segmentation:** By segmenting customers effectively, Acme Retail can tailor marketing strategies to each segment, improving retention and CLV.

These improvements, powered by data-driven insights, contribute to operational efficiency, cost reduction, and enhanced customer satisfaction. Acme Retail can now optimize its supply chain, improve its forecasting systems, and implement targeted marketing strategies to drive profitability and growth.