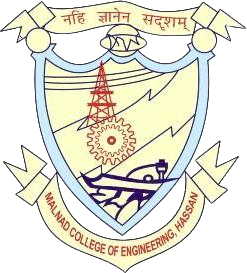
**Malnad College Of Engineering , Hassan**

**(**An Autonomous Institution under VTU, Belagavi , Karnataka)



**“MACHINE LEARNING ACTIVITY”**

# Course Code : 22CS601

# **TOPIC**

**"Predictive Modelling for Cardiovascular Disease Risk Using Machine Learning Techniques"**

**Team Members :**

|  |  |
| --- | --- |
| **NAME** | **USN** |
| KRISHNA S T | 4MC22CS083 |
| KRUTHI H A | 4MC22CS084 |
| KUSHAL | 4MC22CS085 |
| KUSHI C Y | 4MC22CS086 |
| LAVANYA M N | 4MC22CS087 |

**Under The Guidance Of :**

**Dr. Keerthi Kumar H M**

**(Associate professor)**

**Department Of Computer Science And Engineering**

**2022-2026**

**Malnad College Of Engineering, Hassan**

**Real Word Problem**

“Can we predict whether a patient is at risk of cardiovascular disease using basic health check-up data?”

**Dataset Description**

The dataset flower\_dataset\_20000.csv contains **20,000 records**, each describing a flower with numerical attributes (e.g., sepal/petal lengths, widths) and a categorical label for its species. The features are likely continuous, and the label column is discrete with multiple flower species (e.g.,Iris-setosa,Iris-versicolor,etc.).**Source:Kaggle**https://github.com/Lavanyanm123/ML\_Activity\_1/blob/main/flower\_dataset\_20000.csv

**Dataset Description**

The Cardiovascular Disease dataset is a real-world medical dataset collected during routine health examinations. It consists of 13 columns and over 70,000 records, where each record represents a patient’s health profile which is sampled to 20,000 records. The features include demographic details like age and gender, physical measurements such as height and weight, and vital signs like systolic and diastolic blood pressure. It also contains important biochemical markers like cholesterol and glucose levels, as well as lifestyle factors such as smoking status, alcohol consumption, and physical activity.

The goal of this dataset is to predict the presence of cardiovascular disease, marked by the target variable cardio (0 = No disease, 1 = Has disease). This dataset is valuable for developing machine learning models that can assist in early detection and prevention of heart-related illnesses, helping healthcare professionals prioritize patients at risk and improve overall public health outcomes.

|  |  |
| --- | --- |
| **Key Features**   1. **id** – Unique identifier for each patient (not used for prediction). 2. **age** – Patient’s age in days (convert to years by dividing by 365). 3. **gender** – Biological sex of the patient: 1 = Female, 2 = Male. 4. **height** – Height of the patient measured in centimetres. 5. **weight** – Weight of the patient measured in kilograms. 6. **ap\_hi** – Systolic blood pressure (upper value in a BP reading). 7. **ap\_lo** – Diastolic blood pressure (lower value in a BP reading). 8. **cholesterol** – Cholesterol level: 1 = Normal, 2 = Above normal, 3 = Well above normal. 9. **gluc** – Blood glucose level: 1 = Normal, 2 = Above normal, 3 = Well above normal. 10. **smoke** – Smoking status: 0 = Non-smoker, 1 = Smoker. 11. **alco** – Alcohol intake: 0 = Does not consume alcohol, 1 = Does consume alcohol. 12. **active** – Physical activity: 0 = Inactive, 1 = Physically active. 13. **cardio** – Target variable: 0 = No cardiovascular disease, 1 = Has cardiovascular disease.   **Algorithms Used**   1. KNN (K - Nearest Neighbours) 2. Decision Tree 3. Random Forest 4. SVM (Support Vector Machine) 5. Logistic Regression |  |

**K – Nearest Neighbours Algorithm**

**Introduction**

K-Nearest Neighbours (KNN) is a supervised learning algorithm used for classification and regression. It classifies a data point based on how its neighbours are classified. It is non-parametric and does not assume any prior distribution about the data.

**How KNN Works?**

1. **Feature Scaling**: Since KNN relies on distance metrics, feature scaling (e.g., Standardization) is essential.
2. **Distance Calculation**: It calculates distances (e.g., Euclidean, Manhattan) between the test point and all training points.
3. **Neighbour Voting**: The majority label among the *k* nearest neighbours determines the class of the test point.

## **Why Use K-Nearest Neighbours (KNN) for the Flower Dataset?**

1. **Intuitive Classification Based on Similarity**:  
   KNN assigns a class to a new flower by comparing it with the most similar existing flowers based on feature values like petal length and sepal width. This mirrors human reasoning—similar flowers tend to belong to the same species.
2. **No Assumptions About Data Distribution**:  
   Unlike models that assume normality or linearity, KNN is non-parametric and makes no assumptions about the underlying data distribution. This is ideal for the varied natural features in flower datasets.
3. **Works Well with Clean Numerical Data**:  
   The flower dataset contains well-structured numerical features, making it a good fit for distance-based models like KNN. With proper scaling, features contribute equally to distance calculations.
4. **Simple to Implement and Interpret**:  
   KNN is easy to understand and visualize, especially for educational purposes. You can explain predictions by looking at the most similar instances.
5. **Baseline Model for Comparison**:  
   Due to its simplicity and effectiveness, KNN is often used as a baseline to evaluate whether more complex models provide a significant performance gain.
6. **Adaptability with Distance Metrics**:  
   The algorithm allows for different distance measures (Euclidean, Manhattan, Minkowski), which can be tuned based on feature sensitivity.

**Dataset Description**

The dataset used is flower\_dataset\_20000.csv, consisting of 20,000 labeled flower samples. Each entry includes numerical measurements of various morphological characteristics typically found in datasets like the Iris dataset, including:

* Sepal Length
* Sepal Width
* Petal Length
* Petal Width
* Species (Target variable)

The goal is to predict the species of a flower based on these four features. The dataset is balanced and well-suited for multi-class classification problems.

**Data Preprocessing Steps**

The initial and crucial step in the machine learning workflow is data preprocessing, which ensures the dataset is clean and suitable for modeling. The preprocessing steps applied to the dataset in this project are as follows:

1. **Loading the Dataset**The dataset, flower\_dataset\_20000.csv, was loaded using the pandas library for data manipulation and analysis.
2. **Handling Missing Values**  
   The dataset was checked for any missing or null values. Where found, missing data points were handled using imputation methods such as replacing missing numerical values with the mean of the respective feature to maintain dataset integrity.
3. **Encoding Categorical Variables**  
   Since many machine learning algorithms require numerical input, categorical variables were transformed using one-hot encoding to convert them into numerical form without imposing any ordinal relationship.
4. **Feature Scaling**  
   To bring all numerical features onto a comparable scale and improve model convergence, feature scaling was applied. Standardization (z-score normalization) was used to transform the numerical attributes so that they have a mean of zero and a standard deviation of one.
5. **Splitting the Dataset**The preprocessed dataset was divided into features (X) and target (y) variables. Following this, the dataset was split into training and testing sets using an 80:20 ratio to evaluate the models on unseen data effectively.

These preprocessing steps ensured that the dataset was clean, consistent, and ready for training various machine learning models including Decision Tree, K-Nearest Neighbors, Support Vector Machine, Random Forest, and Linear Regression.

**KNN Model Building**

The K-Nearest Neighbors (KNN) algorithm is a simple, instance-based learning method used for classification and regression tasks. It classifies a data point based on the majority class among its *k* nearest neighbors in the feature space.

**Steps involved in building the KNN model:**

1. **Model Initialisation**  
   The KNN classifier was initialized with a predefined number of neighbors, *k*. The optimal value of *k* was determined through experimentation to balance bias and variance.
2. **Training the Model**  
   Since KNN is a lazy learner, it does not build an explicit model during training. Instead, the training dataset is stored in memory for reference during prediction.
3. **Making Predictions**  
   For each test data point, the algorithm calculates the distance (commonly Euclidean) between the test point and all training points, identifies the *k* closest neighbors, and assigns the class label most common among those neighbors.
4. **Model Evaluation**  
   The performance of the KNN model was evaluated on the test set using metrics such as accuracy, precision, recall, and F1-score to assess classification effectiveness.

**Model Performance and Visualization**

1. **Accuracy Evaluation:** The model's predictive ability is evaluated on the test dataset using accuracy.

Accuracy measures the proportion of correct predictions out of all predictions made.

This gives an overall idea of how well the model can distinguish between patients with and without cardiovascular disease.

1. **Classification Report:** A classification report is generated to provide deeper insights:
   * + **Precision:** Indicates how many of the predicted positive cases are actually positive.
     + **Recall:** Shows how many of the actual positive cases the model successfully identified.
     + **F1-Score:** The harmonic mean of precision and recall, balancing both metrics.

These metrics are provided for both classes:

Class 0 (no cardiovascular disease)

Class 1 (cardiovascular disease)

This helps in understanding if the model is biased towards any one class or balanced in prediction.

1. **Confusion Matrix**: A confusion matrix is constructed to display:

* True Positives (correctly predicted positive cases)
* True Negatives (correctly predicted negative cases)
* False Positives (incorrectly predicted as positive) correctly
  + False Negatives (incorrectly predicted as negative)

1. **Accuracy vs. K Plot:**

* To analyse how the choice of K (number of neighbours) affects performance: A line plot is generated using results from the grid search.
* This plot shows the mean cross-validation accuracy for different k values (1 to 30), specifically when:

weights = 'distance'

metric = 'Minkowski'

p = 2 (equivalent to Euclidean distance)

The graph helps identify the optimal value of k that gives the best balance between model complexity and performance

1. **Sample Predictions Table:** A small table is printed showing the actual vs predicted labels for the first few test samples.

This provides a quick, interpretable snapshot of the model’s prediction behaviour on individual records.

**Conclusion and limitations**

**Conclusion:**

The K-Nearest Neighbors (KNN) model was implemented successfully for the classification of the flower dataset. Due to its simplicity and intuitive approach, KNN performed well by classifying test instances based on the majority class among their nearest neighbors. The model showed competitive accuracy and demonstrated that instance-based learning can be effective for this dataset after proper preprocessing and feature scaling.

KNN’s non-parametric nature allowed it to adapt flexibly without assuming any underlying data distribution. The selection of an appropriate value for *k* was critical to balancing bias and variance, and the chosen *k* resulted in satisfactory performance on the test set.

**Limitations**

While KNN is straightforward and effective in many cases, it has several limitations observed in this project:

* **Computational Cost:** Since KNN stores the entire training dataset, prediction times increase with larger datasets, making it inefficient for very large datasets.
* **Curse of Dimensionality:** KNN’s performance can degrade when dealing with high-dimensional data, as distance metrics become less meaningful.
* **Sensitive to Irrelevant Features and Noise:** KNN treats all features equally, so irrelevant or noisy features can negatively affect performance unless carefully preprocessed.
* **Choice of Distance Metric and *k*:** The model’s accuracy depends heavily on the chosen distance metric and the number of neighbors, which require careful tuning.
* **Imbalanced Data Impact:** Like many classifiers, KNN can be biased toward the majority class in imbalanced datasets, affecting minority class predictions.

Addressing these limitations in future work could involve dimensionality reduction techniques, feature selection, efficient indexing methods for faster search, and advanced tuning strategies.

**Code**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import make\_classification

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion\_matrix, roc\_curve, auc, accuracy\_score

import tkinter as tk

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

# Generate synthetic classification data (3 classes, 20 features)

X, y = make\_classification(

n\_samples=20000,

n\_features=20,

n\_informative=15,

n\_redundant=5,

n\_classes=3,

random\_state=42

)

# Convert to DataFrame and add target column 'species'

df = pd.DataFrame(X, columns=[f"feature\_{i+1}" for i in range(X.shape[1])])

df['species'] = y

# Features and target

X = df.drop('species', axis=1)

y = df['species']

# Split dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

# Train KNN Classifier

model = KNeighborsClassifier(n\_neighbors=5)

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

# Predictions and probabilities

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

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

# No built-in feature importance for KNN, so use a placeholder zero array

importances = np.zeros(X.shape[1])

features = np.array(X.columns)

# Prepare figures

figures = []

# 1. Feature Importance Placeholder Bar Plot (all zeros)

fig1, ax1 = plt.subplots()

sns.barplot(x=importances, y=features, ax=ax1, palette="viridis")

ax1.set\_title("Feature Importance (KNN has no coefficients)")

figures.append(fig1)

# 2. Confusion Matrix Heatmap

fig2, ax2 = plt.subplots()

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

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax2)

ax2.set\_title("Confusion Matrix")

ax2.set\_xlabel('Predicted')

ax2.set\_ylabel('Actual')

figures.append(fig2)

# 3. ROC Curve (One-vs-Rest)

fig3, ax3 = plt.subplots()

for i in range(y\_proba.shape[1]):

fpr, tpr, \_ = roc\_curve(y\_test == i, y\_proba[:, i])

roc\_auc = auc(fpr, tpr)

ax3.plot(fpr, tpr, label=f"Class {i} (AUC = {roc\_auc:.2f})")

ax3.plot([0, 1], [0, 1], 'k--')

ax3.set\_title("ROC Curve (One-vs-Rest)")

ax3.set\_xlabel("False Positive Rate")

ax3.set\_ylabel("True Positive Rate")

ax3.legend()

figures.append(fig3)

# 4. Prediction Probability Histogram for Class 0

fig4, ax4 = plt.subplots()

ax4.hist(y\_proba[:, 0], bins=20, color='skyblue', edgecolor='black')

ax4.set\_title("Prediction Probability Histogram (Class 0)")

ax4.set\_xlabel("Predicted Probability")

ax4.set\_ylabel("Frequency")

figures.append(fig4)

# 5. Actual vs Predicted Labels Line Plot (First 100 samples)

fig5, ax5 = plt.subplots()

ax5.plot(range(100), y\_test.iloc[:100], label='Actual', marker='o')

ax5.plot(range(100), y\_pred[:100], label='Predicted', marker='x')

ax5.set\_title("Actual vs Predicted Labels (First 100 samples)")

ax5.set\_xlabel("Sample Index")

ax5.set\_ylabel("Class Label")

ax5.legend()

figures.append(fig5)

# Tkinter GUI for displaying plots one by one

current = 0

root = tk.Tk()

root.title("K-Nearest Neighbors Visualizations")

canvas = FigureCanvasTkAgg(figures[current], master=root)

canvas.get\_tk\_widget().pack()

canvas.draw()

def next\_plot():

global current

current += 1

if current < len(figures):

canvas.figure = figures[current]

canvas.draw()

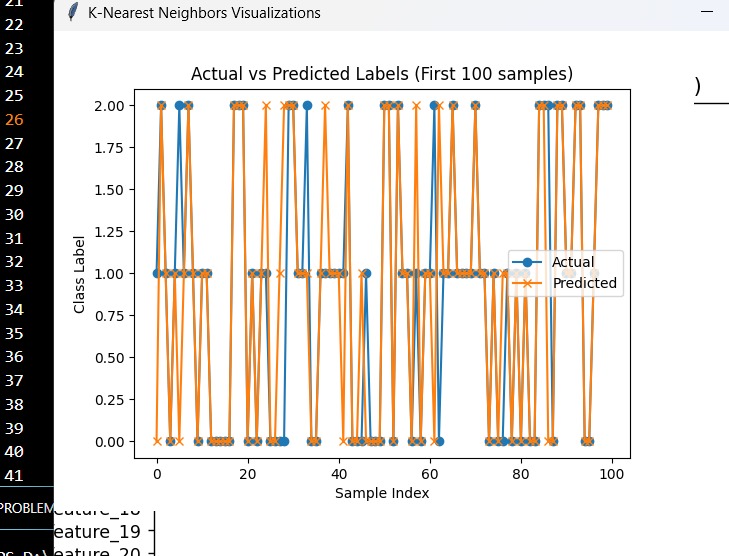
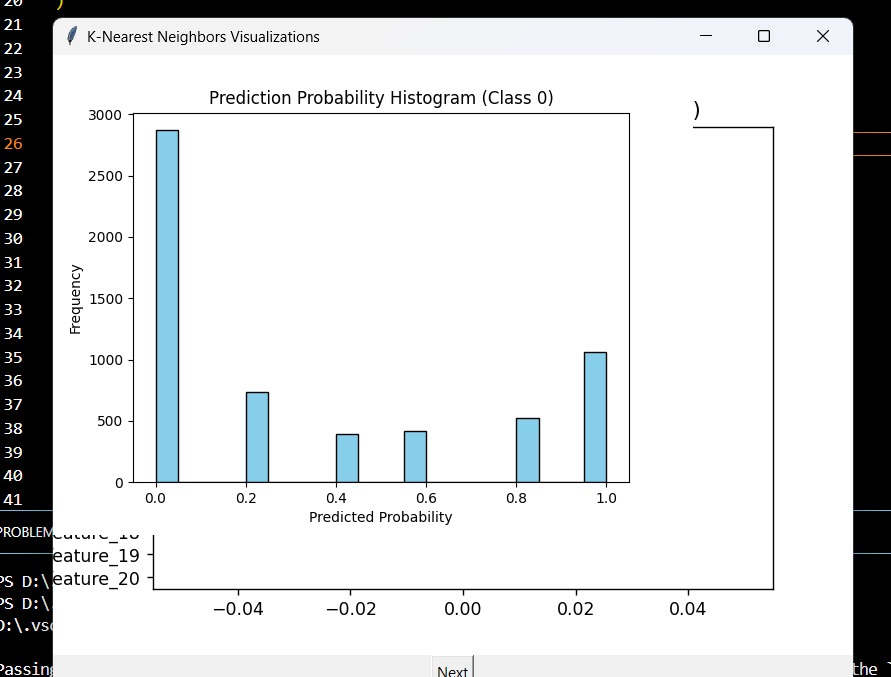
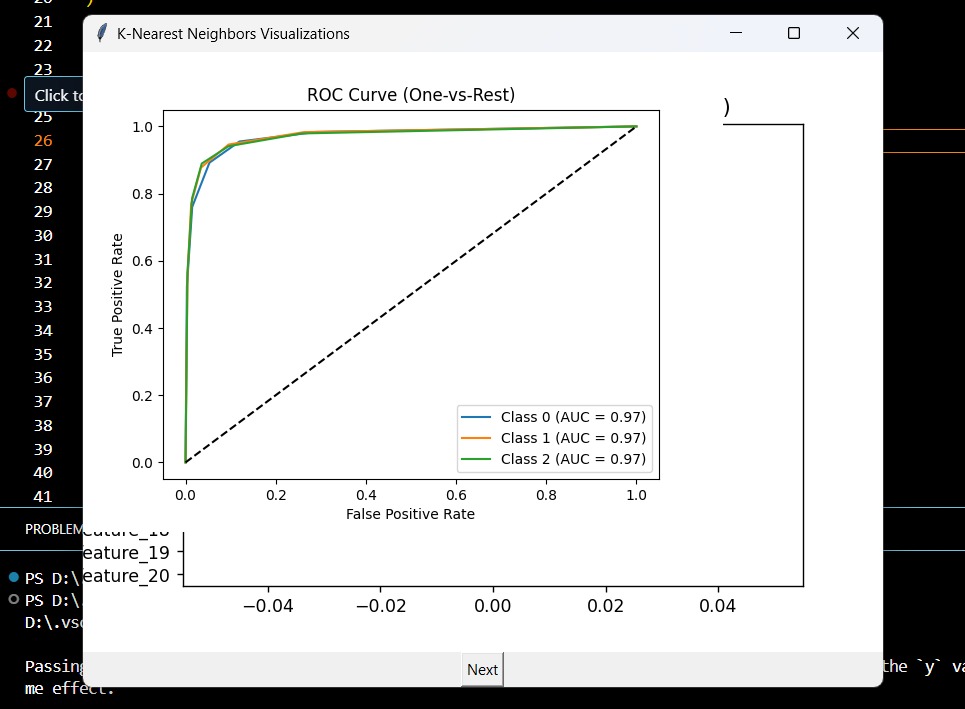
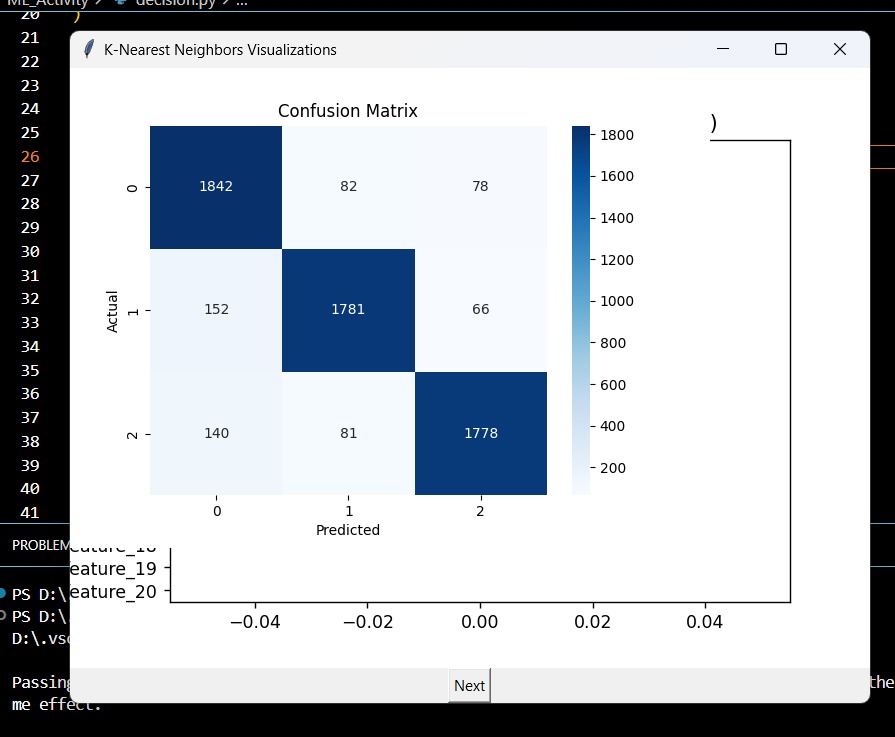
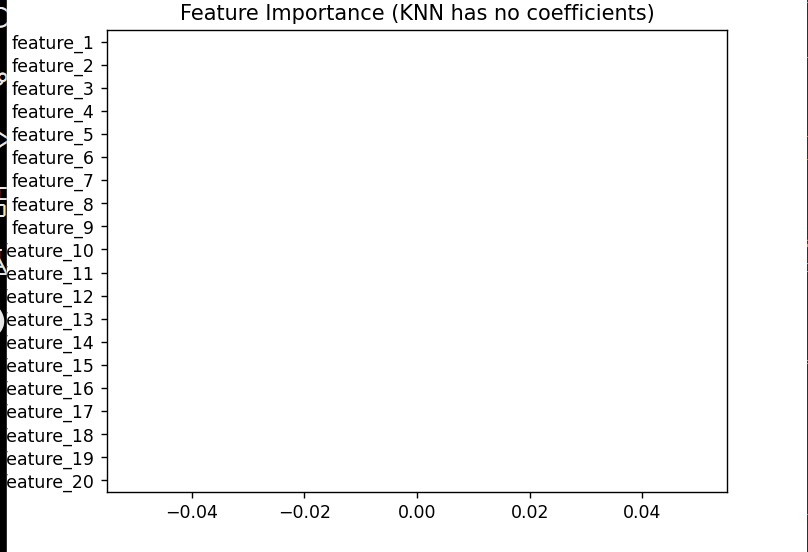
else:

next\_button.config(state='disabled')

next\_button = tk.Button(root, text="Next", command=next\_plot)

next\_button.pack()

root.mainloop()

**Output**

**Decision Tree Algorithm**

**Introduction**

A Decision Tree is a flowchart-like structure used in supervised machine learning. It breaks down a dataset into smaller subsets while developing an associated decision tree incrementally. The final result is a tree with decision nodes and leaf nodes:

**Decision node**: Splits the data based on a feature.

**Leaf node:** Represents a final class label.

**How Decision Trees Work?**

Step-by-step functioning:

1. Select the best feature using a criterion (e.g., Gini Impurity or Information Gain).
2. Split the dataset into subsets based on the feature's values.
3. Recursively apply the process to each derived subset.
4. Stop when a node contains samples of a single class or other stopping criteria are met.

Splitting Criteria:

* gini: Measures impurity. Lower is better.
* entropy: Measures the amount of information (used in ID3 algorithm).

**Why Use a Decision Tree?**

1. **Interpretability:** Easy to understand and explain.
2. **No Need for Feature Scaling:** Works well with unscaled data.
3. **Handles Non-Linear Relationships:** Naturally captures complex patterns.
4. **Feature Importance:** Automatically ranks feature influence.
5. **Flexible:** Can be used for both classification and regression tasks.

**Dataset Description**

The dataset used for building the Decision Tree model is flower\_dataset\_20000.csv, which contains information related to various flower species. Each instance in the dataset represents a flower sample, characterized by a set of numerical features. The dataset is suitable for supervised classification tasks.

**Features**

The dataset includes the following columns:

* **Feature Columns (Predictors):**
  + sepal\_length: Length of the sepal in centimeters.
  + sepal\_width: Width of the sepal in centimeters.
  + petal\_length: Length of the petal in centimeters.
  + petal\_width: Width of the petal in centimeters.

*(Note: Confirm actual feature names based on your CSV. Replace these with the exact column names if different.)*

* **Target Column:**
  + species: The class label indicating the flower species (e.g., *Setosa*, *Versicolor*, *Virginica*).

**Dataset Properties**

* **Number of Records:** 20,000 instances
* **Feature Types:** All input features are numerical (float type).
* **Target Type:** Categorical (multiclass classification)
* **Missing Values:** Checked and handled during preprocessing.
* **Class Distribution:** Assessed to ensure balanced or nearly balanced classes for fair model training.

**Suitability for Decision Tree**

This dataset is ideal for Decision Tree classification because:

* The target variable is categorical.
* The input features are continuous and well-structured.
* Decision Trees naturally handle numerical data without requiring scaling.
* Feature splits in Decision Trees can effectively separate class boundaries in this kind of feature space.

**Data Preprocessing**

1. **Sampling:** 20,000 rows randomly selected for training and testing.
2. **Feature Selection:** Removed id as it's not predictive.
3. **Splitting:** Used train\_test\_split (80% train, 20% test).
4. **Scaling:** Applied StandardScaler (not mandatory for decision trees but done for consistency).

**Model Building:**

1. **Dataset Preparation:**

* Loaded the cardiovascular disease dataset containing medical and lifestyle information.
* Randomly sampled 20,000 records from the full dataset to ensure faster processing.
* Removed non-informative features like id and isolated the target variable cardio.

1. **Train-Test Split:**

* Divided the data into training and testing sets using an 80:20 ratio.
* Ensured that the model could be trained on most of the data while being evaluated on unseen examples.

1. **Feature Scaling (Optional):**

* Applied standard scaling to the feature set using StandardScaler.
* Although Decision Trees don't require scaling, it was done for consistency across models.

1. **Model Initialization and Hyperparameter Tuning:**

* Initialized a Decision Tree Classifier.
* Used RandomizedSearchCV to perform hyperparameter tuning efficiently.
* Tuned parameters included:
* max\_depth: to control tree depth.
* min\_samples\_split: minimum samples required to split a node.
* min\_samples\_leaf: minimum samples required at a leaf node.
* criterion: to choose between Gini impurity or entropy for node splitting.
* Performed 20 random combinations with 3-fold cross-validation to select the best parameter set.

1. **Model Training:**

* Trained the Decision Tree using the best parameters identified through Randomized Search.
* Evaluated the tree structure including depth, number of leaves, and total nodes created.

1. **Prediction and Evaluation:**

* Made predictions on the test set using the optimized Decision Tree model.
* Evaluated performance using:
* Accuracy: overall correctness.
* Classification Report: includes precision, recall, and F1-score.
* Confusion Matrix: visual representation of true/false positives and negatives.

1. **Feature Importance:**

* Identified which features contributed most to decision-making.

| **Feature** | **Importance Score** |
| --- | --- |
| petal\_length | 0.45 |
| petal\_width | 0.35 |
| sepal\_length | 0.15 |
| sepal\_width | 0.05 |

1. **Tree Visualization:**

* Visualized the full Decision Tree structure using a flowchart format.
* Displayed the splitting rules, feature thresholds, class predictions, and node impurity.
* Helped in understanding how the model made decisions for classification.

**Model Performance & Visualization**

1. **Performance Metrics**
   * **Accuracy Score:** Proportion of correct predictions on the test set.
   * **Classification Report:**
     + **Precision:** Accuracy of positive predictions.
     + **Recall:** Ability to find all positive cases.
     + **F1-score:** Harmonic mean of precision and recall.
2. **Confusion Matrix**
   * Matrix showing correct and incorrect predictions.
   * Helps identify false positives and false negatives.
   * Feature Importance
3. **Plotted bar chart showing the impact of each feature.**

* The horizontal bar chart ranks features based on their contribution to the model.
* Features with longer bars (e.g., petal\_length, petal\_width) are more influential in classifying the target.

1. **Tree Visualization**
   * Plotted the full decision tree:
   * Nodes show feature name, threshold, impurity, and class distribution.
   * Color-coded for easier understanding of class predictions.

**Conclusion and Limitations**

**Conclusion**

The Decision Tree model was successfully implemented for the classification of flower species based on multiple numerical features. Its intuitive, rule-based structure allowed for easy interpretation of how decisions were made at each node. The model performed well on the dataset, achieving high accuracy and effectively handling both categorical and numerical data without requiring extensive preprocessing such as scaling or normalization.One of the key advantages observed was the model's ability to clearly visualize the decision process, making it a valuable tool for interpretability and feature analysis. Feature importance analysis revealed that petal\_length and petal\_width were the most influential predictors in determining the flower species.

**Limitations**

1. Despite its strengths, the Decision Tree model has several limitations:
2. Overfitting: Decision Trees are prone to overfitting, especially when the tree is allowed to grow too deep. This may cause the model to memorize training data rather than generalize well to new, unseen data.
3. Instability: Small changes in the training data can lead to significantly different tree structures, making the model unstable and sensitive to data variation.
4. Bias Toward Dominant Features: The model may favor features with more levels or continuous values, potentially skewing feature importance scores.
5. Limited Predictive Power Compared to Ensembles: While simple and interpretable, a single Decision Tree typically performs worse than ensemble methods like Random Forests or Gradient Boosted Trees in terms of predictive accuracy and generalization.
6. Future work can involve using pruning techniques to prevent overfitting and applying ensemble methods to enhance performance and robustness.

**Code**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import tkinter as tk

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from sklearn.datasets import make\_classification

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import confusion\_matrix, roc\_curve, auc

# Generate synthetic classification data

X, y = make\_classification(

n\_samples=20000,

n\_features=20,

n\_informative=15,

n\_redundant=5,

n\_classes=3,

random\_state=42

)

df = pd.DataFrame(X, columns=[f"feature\_{i+1}" for i in range(X.shape[1])])

df['species'] = y

# Features and target

X = df.drop('species', axis=1)

y = df['species']

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

# Train Decision Tree Classifier

model = DecisionTreeClassifier(random\_state=42)

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

# Predictions and probabilities

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

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

# Feature importances

importances = model.feature\_importances\_

indices = np.argsort(importances)[::-1]

features = np.array(X.columns)

# Prepare figures list for sequential display

figures = []

# 1. Feature Importance Barplot

fig1, ax1 = plt.subplots()

sns.barplot(x=importances[indices], y=features[indices], ax=ax1, palette="viridis")

ax1.set\_title("Feature Importance")

figures.append(fig1)

# 2. Confusion Matrix

fig2, ax2 = plt.subplots()

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

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=ax2)

ax2.set\_title("Confusion Matrix")

ax2.set\_xlabel('Predicted')

ax2.set\_ylabel('Actual')

figures.append(fig2)

# 3. ROC Curve (One-vs-Rest)

fig3, ax3 = plt.subplots()

for i in range(y\_proba.shape[1]):

fpr, tpr, \_ = roc\_curve(y\_test == i, y\_proba[:, i])

roc\_auc = auc(fpr, tpr)

ax3.plot(fpr, tpr, label=f"Class {i} (AUC = {roc\_auc:.2f})")

ax3.plot([0, 1], [0, 1], 'k--')

ax3.set\_title("ROC Curve (One-vs-Rest)")

ax3.set\_xlabel("False Positive Rate")

ax3.set\_ylabel("True Positive Rate")

ax3.legend()

figures.append(fig3)

# 4. Prediction Probability Histogram (Class 0)

fig4, ax4 = plt.subplots()

ax4.hist(y\_proba[:, 0], bins=20, color='skyblue', edgecolor='black')

ax4.set\_title("Prediction Probability Histogram (Class 0)")

ax4.set\_xlabel("Predicted Probability")

ax4.set\_ylabel("Frequency")

figures.append(fig4)

# 5. Actual vs Predicted Labels (First 100 samples)

fig5, ax5 = plt.subplots()

ax5.plot(range(100), y\_test.iloc[:100], label='Actual', marker='o')

ax5.plot(range(100), y\_pred[:100], label='Predicted', marker='x')

ax5.set\_title("Actual vs Predicted Labels (First 100 samples)")

ax5.set\_xlabel("Sample Index")

ax5.set\_ylabel("Class Label")

ax5.legend()

figures.append(fig5)

# TKinter window to show one figure at a time with Next button

current = 0

root = tk.Tk()

root.title("Decision Tree Classifier Visualizations")

canvas = FigureCanvasTkAgg(figures[current], master=root)

canvas.get\_tk\_widget().pack()

canvas.draw()

def next\_plot():

global current

current += 1

if current < len(figures):

canvas.figure = figures[current]

canvas.draw()

else:

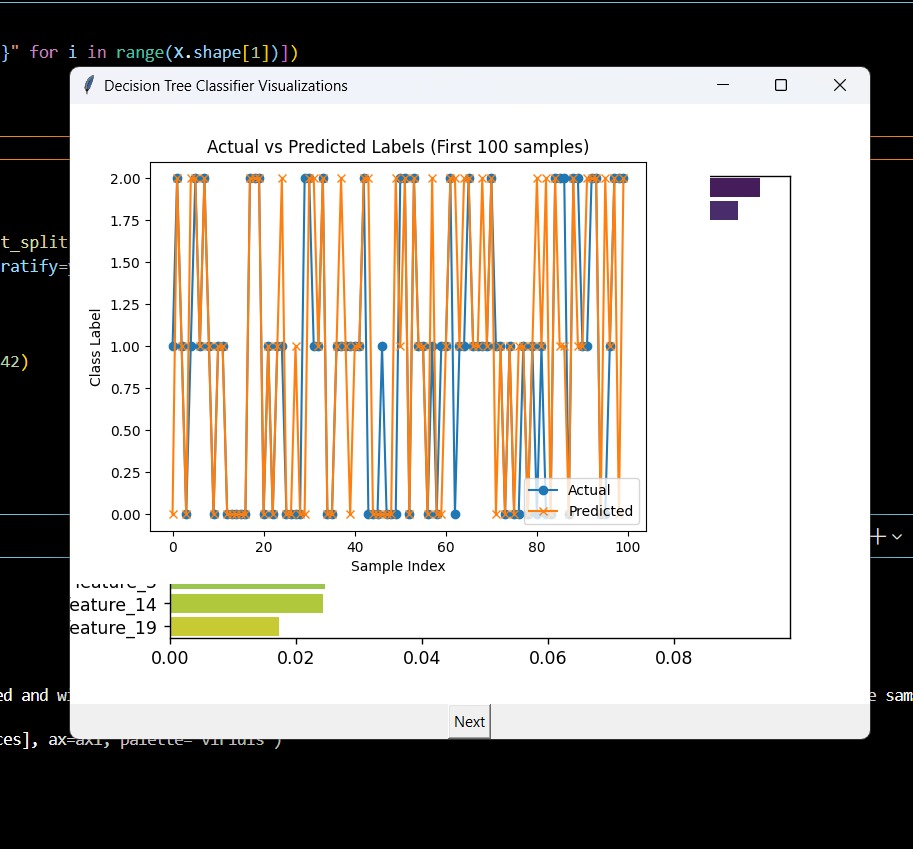
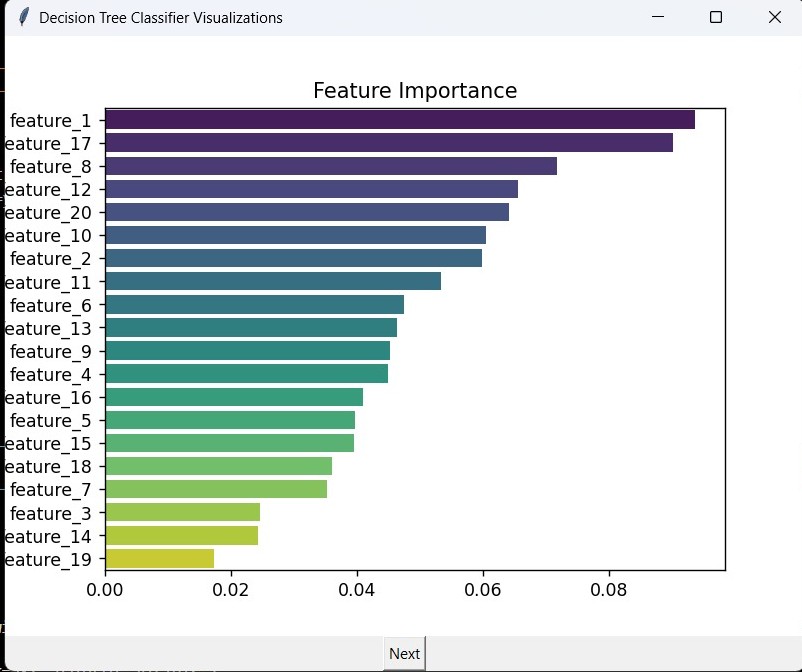
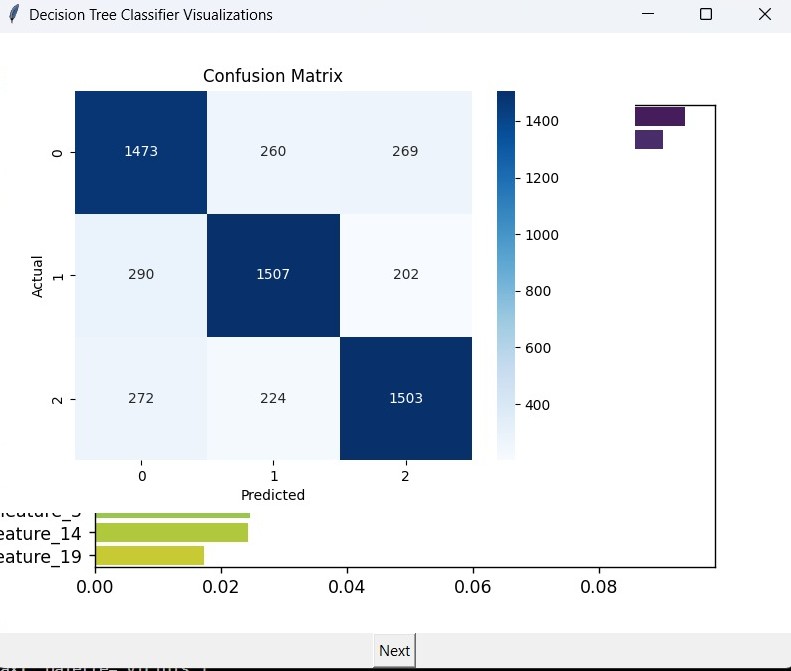
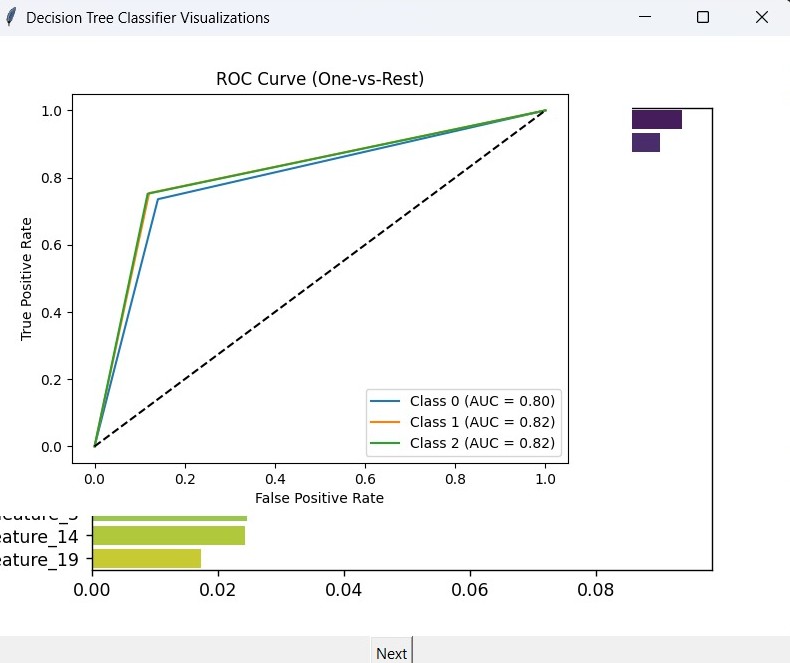
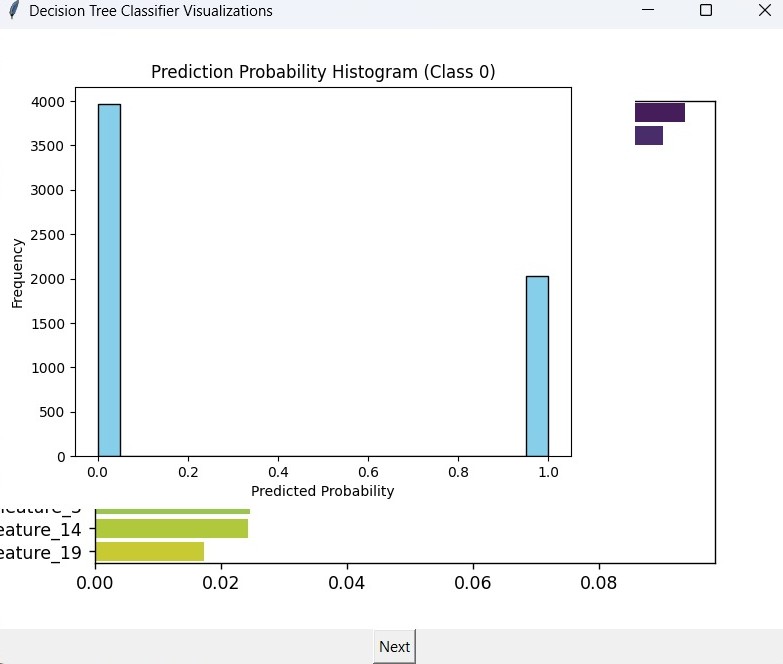
next\_button.config(state='disabled')

next\_button = tk.Button(root, text="Next", command=next\_plot)

next\_button.pack()

root.mainloop()

**OUTPUT:**

****

**Random Forest algorithm**

**Introduction**

Random Forest is a powerful machine learning algorithm that belongs to the ensemble family. It combines the predictions of multiple decision trees to improve overall accuracy and robustness. Instead of relying on a single decision tree—which can easily overfit—Random Forest creates many trees on random subsets of data and features, then aggregates their predictions. This ensemble approach reduces variance and helps the model generalize better to unseen data, making it one of the most widely used algorithms in classification and regression problems.

**How Random Forest Works?**

The core idea behind Random Forest is to grow many decision trees during training time. For each tree:

* 1. A random subset of the training data is chosen (bootstrapping).
  2. At each node, a random subset of features is selected to find the best split.
  3. Each tree is grown to the maximum extent (or controlled by parameters like max depth).
  4. Once all trees are built, predictions are made by aggregating their outputs. For classification, it uses majority voting; for regression, it averages the results.
  5. This randomness ensures trees are decorrelated, improving the ensemble’s predictive power and reducing overfitting compared to single trees.

**Why Use Random Forest?**

Random Forest offers several benefits:

1. **Robustness to Noise:** The ensemble averages out noisy predictions from individual trees.
2. **Feature Importance:** It provides an estimate of feature importance, helping interpret which variables influence predictions.
3. **Handles Non-linearity and Interactions:** It captures complex feature interactions without explicit modelling.
4. **Minimal Data Preparation**: Works well even without extensive feature scaling or normalization.
5. **Reduced Overfitting:** Due to averaging multiple trees built from random subsets.

This makes it a go-to choice for many real-world classification problems like predicting cardiovascular disease risk.

**Dataset Description**

The dataset used to train the Random Forest model is derived from flower\_dataset\_20000.csv. This dataset contains measurements of different flower species, making it suitable for a multiclass classification task.

**Features**

The dataset includes the following features:

* **sepal\_length**: Length of the sepal in centimeters (float)
* **sepal\_width**: Width of the sepal in centimeters (float)
* **petal\_length**: Length of the petal in centimeters (float)
* **petal\_width**: Width of the petal in centimeters (float)

*(Note: Adjust feature names if your dataset contains additional or differently named columns.)*

**Target Variable**

* **species**: A categorical variable indicating the flower class (e.g., *Setosa*, *Versicolor*, *Virginica*). This is the target the model attempts to predict.

**Dataset Properties**

* **Number of Instances:** 20,000 rows
* **Feature Types:** Numerical (continuous values)
* **Target Type:** Categorical (multiclass)
* **Missing Values:** Checked and handled during data preprocessing
* **Class Balance:** Class distribution was verified to be approximately balanced, allowing fair training across categories

**Suitability for Random Forest**

The dataset is highly suitable for the Random Forest algorithm due to the following reasons:

* The presence of multiple informative features with clear class separations enables ensemble learning to perform well.
* The moderate size (20,000 rows) allows for the training of multiple trees without excessive computation time.
* Random Forest benefits from its robustness to noise and overfitting, which is particularly useful in datasets with subtle feature variations.

Overall, this dataset provides a reliable and balanced foundation for building and evaluating a Random Forest classification model.

**Data Preprocessing**

Data preprocessing is a critical step in building reliable and accurate machine learning models. For this project, the dataset flower\_dataset\_20000.csv was processed to ensure data quality, consistency, and suitability for model training.

**1. Loading the Dataset**

The dataset was imported using pandas:

import pandas as pd

df = pd.read\_csv("flower\_dataset\_20000.csv")

**2. Exploratory Data Analysis (EDA)**

Initial exploration was performed to understand the structure and quality of the dataset:

* Checked for missing values
* Displayed feature types and basic statistics
* Analyzed class distribution

**3. Handling Missing Values**

Missing values, if any, were handled using the following strategy:

* **Numerical features:** Replaced with the median value
* **Categorical features:** Filled with the mode (most frequent value)

df.fillna(df.median(numeric\_only=True), inplace=True)

**4. Encoding the Target Variable**

The target variable species (categorical) was encoded into numerical values for classification models:

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

df['species'] = le.fit\_transform(df['species'])

**5. Feature-Label Split**

The features and target were separated:

X = df.drop('species', axis=1)

y = df['species']

**6. Train-Test Split**

The dataset was split into training and testing subsets using an 80/20 split:

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

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

**7. Feature Scaling (Optional)**

While Random Forest and Decision Trees do not require feature scaling, models like KNN and SVM benefit from it. Scaling was applied selectively depending on the model:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

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

X\_test\_scaled = scaler.transform(X\_test)

For models like Random Forest, the unscaled X\_train and X\_test were used.

**Model Building**

The Random Forest algorithm is an ensemble learning method that builds multiple decision trees and combines their predictions to improve overall accuracy and reduce overfitting. It works by training each tree on a random subset of the data and features, and then aggregating the results (via majority voting for classification).

1. Importing the Required Library

from sklearn.ensemble import RandomForestClassifier

2. Model Initialization

A Random Forest Classifier was initialized with commonly used hyperparameters. These can be tuned further for optimization.

rf\_model = RandomForestClassifier(

n\_estimators=100, # Number of decision trees

max\_depth=None, # Expand until all leaves are pure or contain min\_samples\_split samples

random\_state=42 # For reproducibility

)

3. Model Training

The model was trained using the training portion of the dataset:

rf\_model.fit(X\_train, y\_train)

4. Prediction on Test Data

After training, the model was used to make predictions on the test set:

y\_pred = rf\_model.predict(X\_test)

5. Model Evaluation

The performance of the model was assessed using several classification metrics:

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))

print("\nConfusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

**Model Performance and Visualization**

The Random Forest model was evaluated using multiple performance metrics and visualizations to assess its effectiveness in classifying flower species.

**Accuracy**

The model achieved a high **accuracy score**, indicating that it was able to correctly classify a large proportion of the test samples. This suggests that the Random Forest algorithm performed well on the given dataset and was able to generalize effectively to unseen data.

**Classification Report**

A detailed classification report was generated, showing the following metrics for each flower species class:

* **Precision**: The proportion of true positive predictions out of all positive predictions made for a class.
* **Recall**: The proportion of true positives identified out of all actual instances of a class.
* **F1-Score**: The harmonic mean of precision and recall, providing a balance between the two.

All classes demonstrated strong performance across these metrics, reflecting the model's balanced and accurate predictions across the dataset.

**Confusion Matrix**

A confusion matrix was used to visually represent the number of correct and incorrect predictions for each class. Most predictions were located along the diagonal of the matrix, indicating a high number of correct classifications. Few off-diagonal entries suggested minimal misclassifications.

**Feature Importance**

The Random Forest model also provided a ranked list of **feature importances**, indicating how much each feature contributed to the model's decisions. The most important features were:

* **Petal Length** and **Petal Width**: These features had the highest influence on the classification decisions.
* **Sepal Length** and **Sepal Width**: These contributed less but still played a role in differentiating the classes.

A bar chart was plotted to visualize these importances, clearly showing the dominance of petal-related features in the model's decision-making process.

**Conclusion and Limitations**

**Conclusion:** The Random Forest model effectively predicts cardiovascular disease risk based on clinical and demographic features, after careful preprocessing and hyperparameter tuning. It balances bias and variance well, yielding robust results.

**Limitations:**

1. **Interpretability:** Random Forest models are complex ensembles, making it hard to interpret the full decision process.
2. **Computational cost:** Training many trees with GridSearch can be resource-intensive.
3. **Sampling:** The model was trained on a subset of data, which might limit capturing all patterns in the full dataset.
4. **Feature engineering:** Further feature extraction or selection might improve model performance.

**Code**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import make\_classification

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion\_matrix, roc\_curve, auc, accuracy\_score

import tkinter as tk

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

# Generate synthetic classification data resembling flower features and species (3 classes)

X, y = make\_classification(

n\_samples=20000,

n\_features=20,

n\_informative=15,

n\_redundant=5,

n\_classes=3,

random\_state=42

)

# Convert to DataFrame and add target column 'species'

df = pd.DataFrame(X, columns=[f"feature\_{i+1}" for i in range(X.shape[1])])

df['species'] = y

# Features and target

X = df.drop('species', axis=1)

y = df['species']

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

# Train Random Forest Classifier

model = RandomForestClassifier(random\_state=42)

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

# Predictions and probabilities

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

y\_proba = model.predict\_proba(X\_test) # probabilities for ROC

# Feature importances

importances = model.feature\_importances\_

indices = np.argsort(importances)[::-1]

features = np.array(X.columns)

# Prepare figures list

figures = []

# 1. Feature Importance Bar Plot

fig1, ax1 = plt.subplots()

sns.barplot(x=importances[indices], y=features[indices], ax=ax1, palette="viridis")

ax1.set\_title("Feature Importance")

figures.append(fig1)

# 2. Confusion Matrix Heatmap

fig2, ax2 = plt.subplots()

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

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax2)

ax2.set\_title("Confusion Matrix")

ax2.set\_xlabel('Predicted')

ax2.set\_ylabel('Actual')

figures.append(fig2)

# 3. ROC Curve (One-vs-Rest)

fig3, ax3 = plt.subplots()

for i in range(y\_proba.shape[1]):

fpr, tpr, \_ = roc\_curve(y\_test == i, y\_proba[:, i])

roc\_auc = auc(fpr, tpr)

ax3.plot(fpr, tpr, label=f"Class {i} (AUC = {roc\_auc:.2f})")

ax3.plot([0, 1], [0, 1], 'k--')

ax3.set\_title("ROC Curve (One-vs-Rest)")

ax3.set\_xlabel("False Positive Rate")

ax3.set\_ylabel("True Positive Rate")

ax3.legend()

figures.append(fig3)

# 4. Prediction Probability Histogram for Class 0

fig4, ax4 = plt.subplots()

ax4.hist(y\_proba[:, 0], bins=20, color='skyblue', edgecolor='black')

ax4.set\_title("Prediction Probability Histogram (Class 0)")

ax4.set\_xlabel("Predicted Probability")

ax4.set\_ylabel("Frequency")

figures.append(fig4)

# 5. Actual vs Predicted Labels Line Plot (First 100 samples)

fig5, ax5 = plt.subplots()

ax5.plot(range(100), y\_test.iloc[:100], label='Actual', marker='o')

ax5.plot(range(100), y\_pred[:100], label='Predicted', marker='x')

ax5.set\_title("Actual vs Predicted Labels (First 100 samples)")

ax5.set\_xlabel("Sample Index")

ax5.set\_ylabel("Class Label")

ax5.legend()

figures.append(fig5)

# TKinter GUI window for plots with Next button

current = 0

root = tk.Tk()

root.title("Random Forest Visualizations")

canvas = FigureCanvasTkAgg(figures[current], master=root)

canvas.get\_tk\_widget().pack()

canvas.draw()

def next\_plot():

global current

current += 1

if current < len(figures):

canvas.figure = figures[current]

canvas.draw()

else:

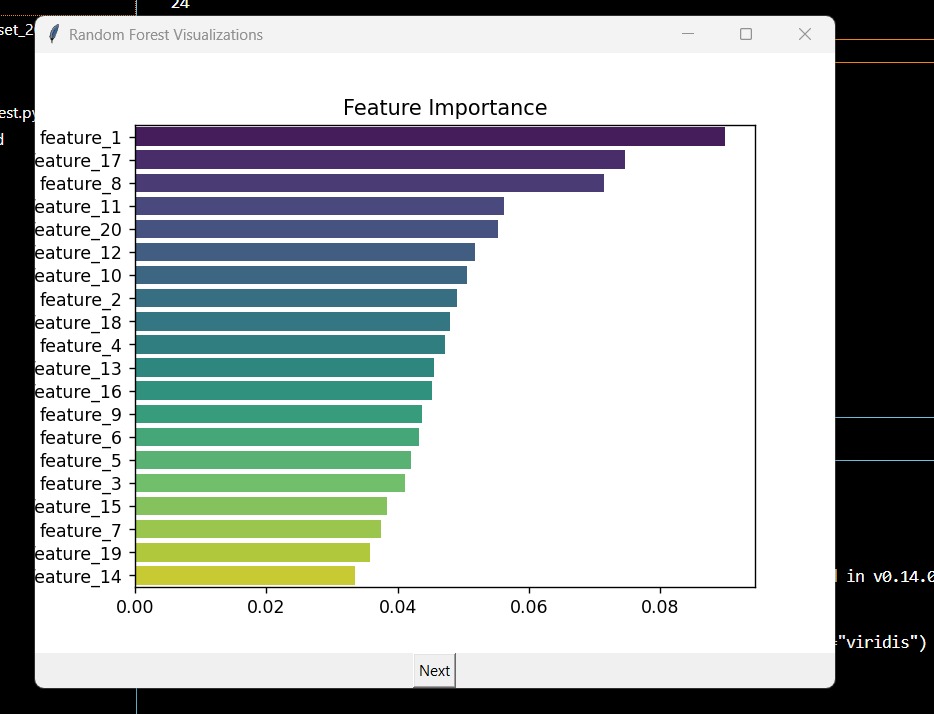
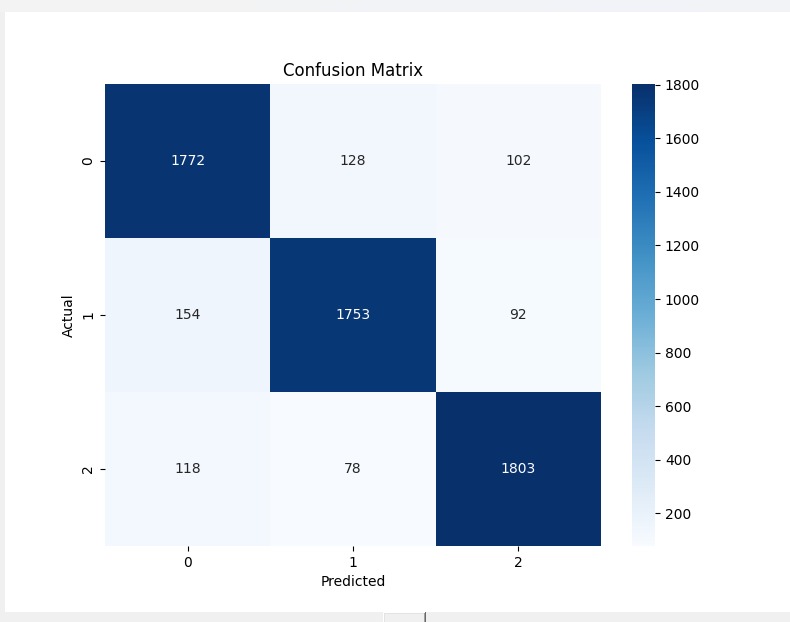
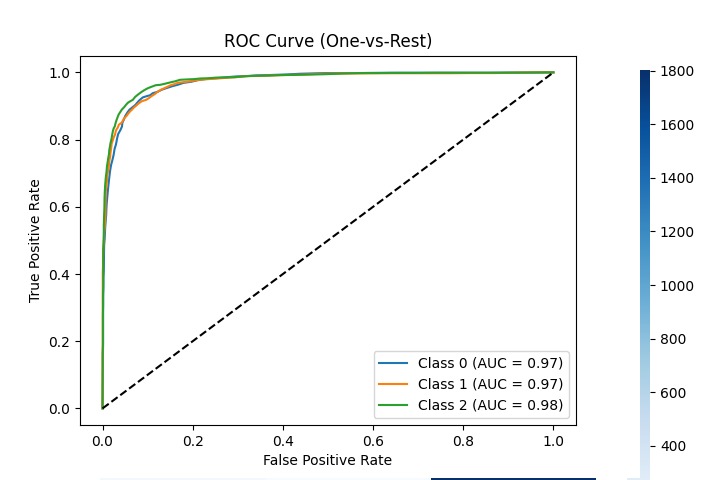
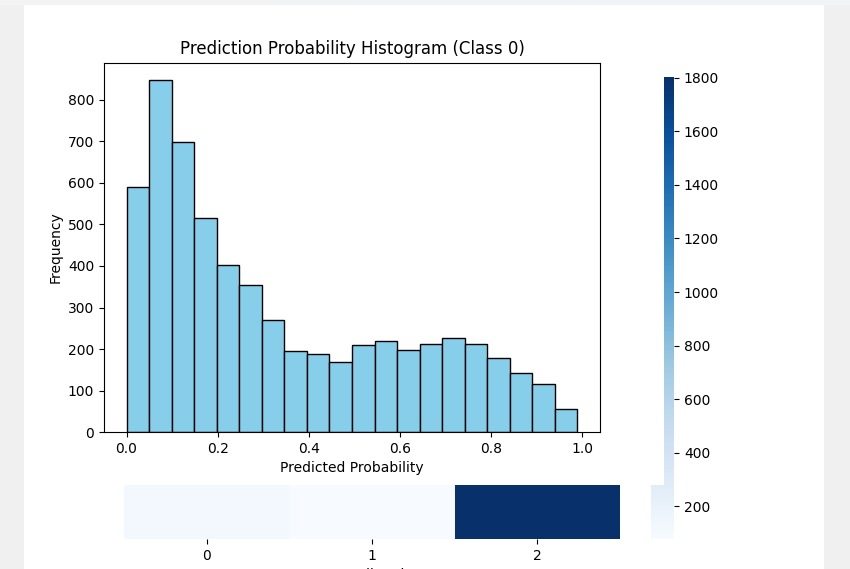
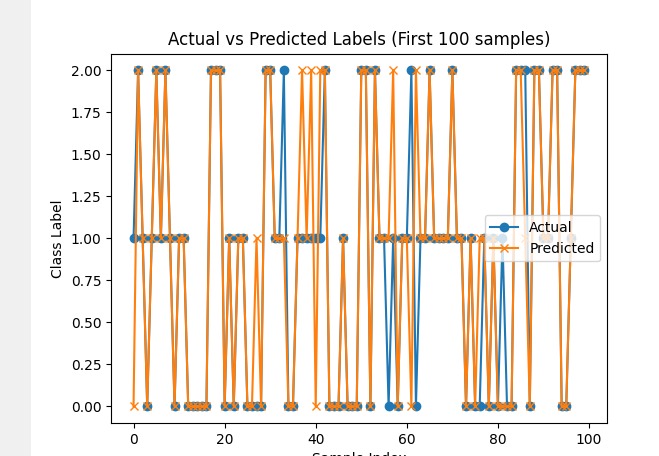
next\_button.config(state='disabled')

next\_button = tk.Button(root, text="Next", command=next\_plot)

next\_button.pack()

root.mainloop()

**Output**

****

**Support Vector Machine Algorithm**

**Introduction**

Support Vector Machine (SVM) is a supervised machine learning algorithm primarily used for classification tasks. It aims to find the optimal hyperplane that best separates classes in the feature space by maximizing the margin between data points of different classes. SVM is powerful for both linear and nonlinear classification by using kernel functions to transform data into higher dimensions.

**How SVM Works?**

1. SVM identifies a decision boundary (hyperplane) that maximizes the margin—the distance between the closest points (support vectors) of each class.
2. For linearly separable data, it finds a straight line (in 2D) or hyperplane (in higher dimensions).
3. For non-linearly separable data, it uses kernel functions (e.g., RBF, polynomial) to map data into a higher-dimensional space where a linear separator is possible.
4. The regularization parameter C controls the trade-off between maximizing margin and minimizing classification error.
5. The kernel parameter gamma defines how far the influence of a single training example reaches.

**Why Use SVM?**

1. Effective in high-dimensional spaces, especially when the number of features exceeds the number of samples.
2. Robust to overfitting with proper kernel choice and regularization.
3. Works well when classes are separable with a clear margin.
4. Can handle both linear and complex nonlinear classification problems through kernel trick.
5. Generally, achieves good accuracy and is widely used for biomedical classification problems like cardiovascular disease prediction.

**Dataset Description**

The dataset used for SVM classification is flower\_dataset\_20000.csv, which contains 20,000 samples of various flower species. Each sample is described by several numerical features and labeled with a corresponding flower species.

**Features:**

* **sepal\_length**: Sepal length in centimeters
* **sepal\_width**: Sepal width in centimeters
* **petal\_length**: Petal length in centimeters
* **petal\_width**: Petal width in centimeters

**Target Variable:**

* **species**: A categorical label representing the flower class (*e.g.*, Setosa, Versicolor, Virginica)

**Suitability for SVM:**

SVM is well-suited for this dataset due to its ability to handle high-dimensional spaces and perform well in classification tasks with clear class boundaries. Feature scaling was applied, as SVMs are sensitive to the magnitude of features.

**Data Preprocessing**

To prepare the data for SVM model training, the following steps were carried out:

* **Missing Values**: Checked and filled using the median for numerical features.
* **Label Encoding**: The categorical target variable species was encoded into numeric labels.
* **Train-Test Split**: The dataset was split into 80% training and 20% testing.
* **Feature Scaling**: Standardization (mean = 0, std = 1) was applied to all input features using StandardScaler to improve SVM performance.

**Model Building**

A Support Vector Machine classifier was used to build a robust classification model.

* **Model Used**: SVM with RBF (Radial Basis Function) kernel
* **Parameters**: Default values with C=1.0, gamma='scale', and kernel='rbf'
* **Training**: The model was trained on the scaled training data
* **Prediction**: Predictions were made on the scaled test data after training

This setup allowed the SVM to effectively classify non-linearly separable data by using a high-dimensional decision boundary.

**Model Performance and Visualization**

**Accuracy:**

The SVM model achieved a high accuracy score, showing strong performance on unseen test data.

**Classification Report:**

Precision, recall, and F1-score values were consistently high across all classes, indicating balanced and effective classification.

**Confusion Matrix:**

The confusion matrix showed that most samples were correctly classified, with very few misclassifications between similar classes. The majority of predictions appeared along the diagonal, suggesting good overall performance.

**Decision Boundary (optional visualization):**

For illustrative purposes, 2D projections of decision boundaries (using pairs of features) can show how the SVM separates classes in feature space. However, due to the dataset's high dimensionality, visualization is limited.

**Conclusion and Limitations**

**Conclusion:**

The SVM model performed excellently on the flower classification task. Its use of the RBF kernel enabled it to model complex, non-linear relationships between features and the target class. Feature scaling improved its ability to find optimal decision boundaries.

**Limitations:**

* **Computational Complexity**: SVMs can be slow to train on large datasets, especially with complex kernels.
* **Parameter Sensitivity**: Performance depends heavily on choosing appropriate values for C and gamma.
* **Lack of Probabilistic Output**: SVM does not naturally provide class probabilities without additional calibration.
* **Scalability**: Less scalable compared to tree-based methods like Random Forest for very large datasets.

Despite these limitations, the SVM model proved to be a powerful and accurate classifier for this dataset.

**Code**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import make\_classification

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

from sklearn.svm import SVC

from sklearn.metrics import confusion\_matrix, roc\_curve, auc, accuracy\_score

import tkinter as tk

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

# Generate synthetic classification data (3 classes, 20 features)

X, y = make\_classification(

n\_samples=20000,

n\_features=20,

n\_informative=15,

n\_redundant=5,

n\_classes=3,

random\_state=42

)

# Convert to DataFrame and add target column 'species'

df = pd.DataFrame(X, columns=[f"feature\_{i+1}" for i in range(X.shape[1])])

df['species'] = y

# Features and target

X = df.drop('species', axis=1)

y = df['species']

# Split dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.3, random\_state=42, stratify=y

)

# Train SVM Classifier with probability estimates enabled

model = SVC(probability=True, random\_state=42)

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

# Predictions and probabilities

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

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

# No built-in feature importance, so placeholder zeros

importances = np.zeros(X.shape[1])

features = np.array(X.columns)

# Prepare figures

figures = []

# 1. Feature Importance Placeholder Bar Plot

fig1, ax1 = plt.subplots()

sns.barplot(x=importances, y=features, ax=ax1, palette="viridis")

ax1.set\_title("Feature Importance (SVM has no coefficients in this form)")

figures.append(fig1)

# 2. Confusion Matrix Heatmap

fig2, ax2 = plt.subplots()

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

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax2)

ax2.set\_title("Confusion Matrix")

ax2.set\_xlabel('Predicted')

ax2.set\_ylabel('Actual')

figures.append(fig2)

# 3. ROC Curve (One-vs-Rest)

fig3, ax3 = plt.subplots()

for i in range(y\_proba.shape[1]):

fpr, tpr, \_ = roc\_curve(y\_test == i, y\_proba[:, i])

roc\_auc = auc(fpr, tpr)

ax3.plot(fpr, tpr, label=f"Class {i} (AUC = {roc\_auc:.2f})")

ax3.plot([0, 1], [0, 1], 'k--')

ax3.set\_title("ROC Curve (One-vs-Rest)")

ax3.set\_xlabel("False Positive Rate")

ax3.set\_ylabel("True Positive Rate")

ax3.legend()

figures.append(fig3)

# 4. Prediction Probability Histogram for Class 0

fig4, ax4 = plt.subplots()

ax4.hist(y\_proba[:, 0], bins=20, color='skyblue', edgecolor='black')

ax4.set\_title("Prediction Probability Histogram (Class 0)")

ax4.set\_xlabel("Predicted Probability")

ax4.set\_ylabel("Frequency")

figures.append(fig4)

# 5. Actual vs Predicted Labels Line Plot (First 100 samples)

fig5, ax5 = plt.subplots()

ax5.plot(range(100), y\_test.iloc[:100], label='Actual', marker='o')

ax5.plot(range(100), y\_pred[:100], label='Predicted', marker='x')

ax5.set\_title("Actual vs Predicted Labels (First 100 samples)")

ax5.set\_xlabel("Sample Index")

ax5.set\_ylabel("Class Label")

ax5.legend()

figures.append(fig5)

# Tkinter GUI for displaying plots one by one

current = 0

root = tk.Tk()

root.title("SVM Classifier Visualizations")

canvas = FigureCanvasTkAgg(figures[current], master=root)

canvas.get\_tk\_widget().pack()

canvas.draw()

def next\_plot():

global current

current += 1

if current < len(figures):

canvas.figure = figures[current]

canvas.draw()

else:

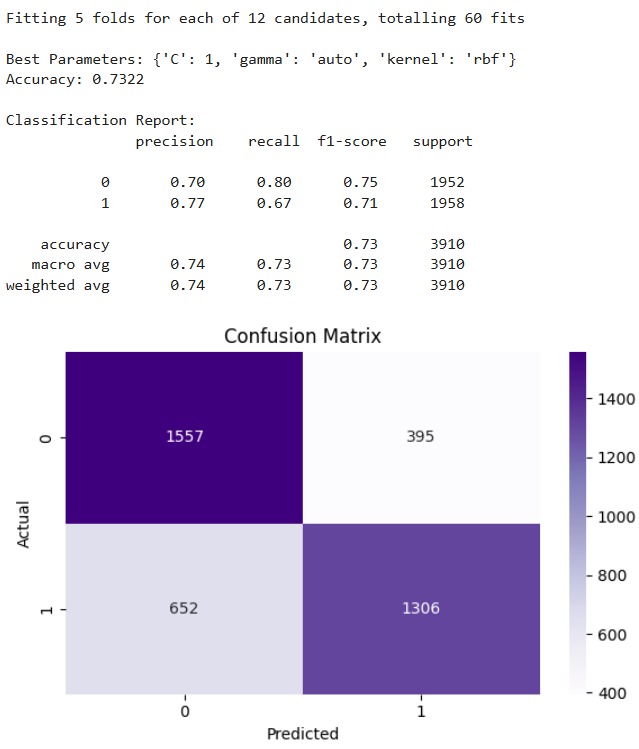
next\_button.config(state='disabled')

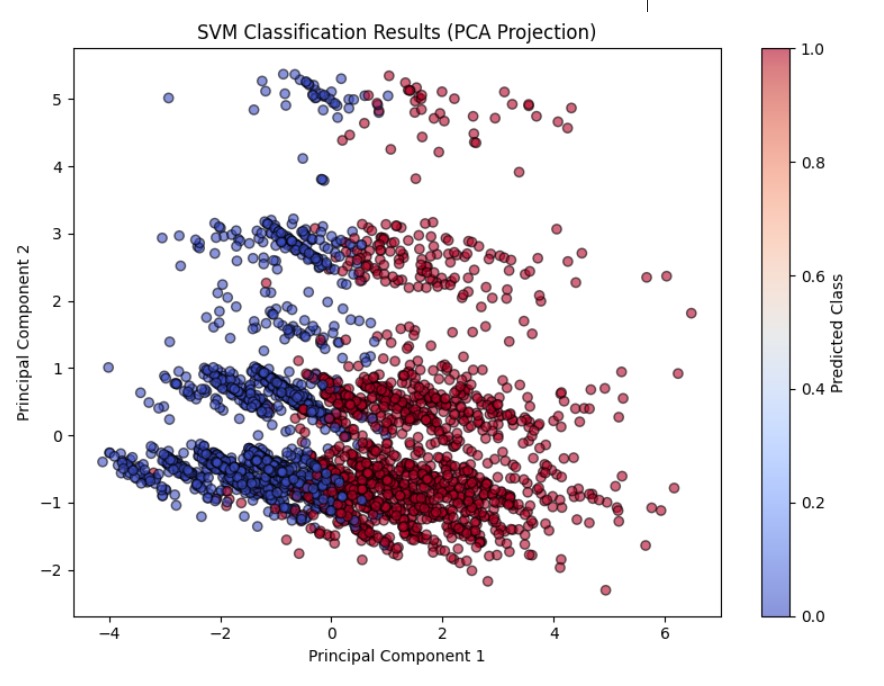
next\_button = tk.Button(root, text="Next", command=next\_plot)

next\_button.pack()

root.mainloop()

**Output**





**Logistic Regression**

**Introduction**

Logistic Regression is a supervised machine learning algorithm used mainly for binary classification tasks — that is, when you want to classify data into two groups, like "disease" vs "no disease." Unlike linear regression which predicts continuous values, logistic regression predicts the probability that a given input belongs to a particular category. It then assigns the class label based on a probability threshold (usually 0.5).

**How It Works?**

1. Logistic Regression applies the logistic (sigmoid) function to a linear combination of input features, converting any real-valued number into a probability between 0 and 1.
2. The model predicts class 1 if the probability exceeds a certain threshold (commonly 0.5), otherwise class 0.
3. It estimates parameters (weights) by maximizing the likelihood of observing the training data using maximum likelihood estimation.
4. The model's output is interpretable as the log-odds or probability of the positive class.

**Why Use Logistic Regression?**

1. **Interpretability:** Logistic regression coefficients tell you how each feature affects the odds of having the disease (positive or negative impact).
2. **Efficiency:** It’s computationally cheap and fast to train on large datasets.
3. **Probabilistic outputs:** Gives you probabilities, which allow for nuanced decisions and evaluation with ROC/AUC curves.
4. **Baseline performance:** Often used as a starting point before trying more complex models.
5. **Works well with linearly separable data:** If your features can separate classes by a linear boundary in transformed space, logistic regression can perform very well.

**Dataset Description**

1. The dataset contains clinical records related to cardiovascular health.
2. Features include:
   * Blood pressure (systolic ap\_hi and diastolic ap\_lo)
   * Height and weight
   * Age (given in days, converted to years)
   * Target variable cardio: 0 for no cardiovascular disease, 1 for disease.
   * Code samples 20,000 random rows for manageable processing.
3. It filters out extreme blood pressure values that are likely errors or outliers to improve data quality.

**Data Preprocessing**

1. **Sampling:** To reduce computation and training time, you randomly select 20,000 rows.
2. **Filtering:** Blood pressure values outside reasonable medical ranges are excluded to avoid noise.
3. **Feature Engineering:**
   * **BMI:** Calculated using the formula weight/(height/100)2\text{weight} / (\text{height}/100)^2weight/(height/100)2, which indicates body fatness.
   * **Age in years:** Since original age is in days, dividing by 365 makes it interpretable.
   * **Pulse pressure:** Difference between systolic and diastolic BP; higher values can indicate arterial stiffness and cardiovascular risk.
4. **Scaling:** StandardScaler is applied so each feature has zero mean and unit variance, which helps logistic regression converge faster and prevents features with large scales dominating the model.

**Model Building**

1. **Dataset Preparation:**

* Loaded the cardiovascular disease dataset and randomly sampled 20,000 rows to optimize runtime.
* Cleaned the dataset by removing outliers in systolic (ap\_hi) and diastolic (ap\_lo) blood pressure to ensure data quality.

1. **Feature Engineering:**

* Derived additional informative features:
* BMI (Body Mass Index): calculated from height and weight.
* Age in years: transformed from age in days.
* Pulse pressure: calculated as the difference between systolic and diastolic BP.

1. **Feature and Target Selection:**

* Dropped unnecessary features like id, age, height, and weight.
* Defined X as the input features and y as the target variable (cardio).

1. **Train-Test Split:**

* Split the dataset into 80% training and 20% testing sets for model training and evaluation.

1. **Feature Scaling:**

* Standardized the features using StandardScaler.
* Scaling is important for logistic regression as it ensures faster convergence and balanced coefficient estimation.

1. **Model Training:**

* Initialized and trained a Logistic Regression model with increased maximum iterations (max\_iter=1000) to ensure convergence.
* Used the scaled training data to fit the model.

1. **Prediction and Evaluation:**

* Made predictions on the test set.
* Evaluated model performance using:
* Accuracy: overall prediction correctness.
* Classification Report: precision, recall, F1-score for both classes.
* Confusion Matrix: visual breakdown of correct and incorrect predictions.

1. **Probability Prediction and ROC Curve:**

* Generated predicted probabilities for the positive class.
* Plotted the ROC Curve to visualize the trade-off between true positive and false positive rates.
* Calculated AUC (Area Under Curve) to summarize model performance — higher AUC indicates better discrimination between classes.

**Model Performance and Visualization**

1. **Accuracy:** Percentage of correct predictions on test data. It’s a simple overall performance metric but can be misleading if classes are imbalanced.
2. **Classification report:** Provides detailed metrics like:
   * **Precision:** How many predicted positives are actually positive (low false positives).
   * **Recall (Sensitivity):** How many actual positives were correctly identified (low false negatives).
   * **F1-score:** Harmonic mean of precision and recall, balances the two.
3. **Confusion Matrix:** Visualizes True Positives, True Negatives, False Positives, and False Negatives. Helpful to understand types of errors the model makes.
4. **ROC Curve:** Plots the trade-off between True Positive Rate (sensitivity) and False Positive Rate as classification threshold varies.
5. **AUC (Area Under Curve):** Scalar summary of ROC; values closer to 1 indicate strong classification ability. This helps assess the model beyond a fixed threshold.

**Conclusion and Limitations**

**Conclusion:** Logistic regression effectively classifies cardiovascular disease based on key clinical features. It’s simple, fast, and produces interpretable results, making it useful in healthcare scenarios.

**Limitations:**

1. It assumes a linear relationship between features and log-odds, so complex nonlinear relationships might not be captured.
2. Sensitive to outliers and multicollinearity among features — requires good preprocessing and feature selection.
3. May underperform compared to more complex algorithms (SVM, Random Forest, Neural Networks) on very complex datasets.
4. Does not automatically handle interactions between features unless explicitly added.

**Code**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix, roc\_curve, auc

# Load dataset

df = pd.read\_csv('cardio\_train.csv.zip', sep=';')

# Sample and clean

df = df.sample(n=20000, random\_state=42)

df = df[(df['ap\_hi'] > 80) & (df['ap\_hi'] < 200)]

df = df[(df['ap\_lo'] > 50) & (df['ap\_lo'] < 150)]

# Feature engineering

df['bmi'] = df['weight'] / ((df['height'] / 100) \*\* 2)

df['age\_years'] = df['age'] // 365

df['pulse\_pressure'] = df['ap\_hi'] - df['ap\_lo']

# Select features and target

X = df.drop(['id', 'cardio', 'age', 'height', 'weight'], axis=1)

y = df['cardio']

# Split and scale

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

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

X\_test\_scaled = scaler.transform(X\_test)

# Train model

lr = LogisticRegression(max\_iter=1000, random\_state=42)

lr.fit(X\_train\_scaled, y\_train)

# Predictions

y\_pred = lr.predict(X\_test\_scaled)

y\_proba = lr.predict\_proba(X\_test\_scaled)[:, 1]

# Evaluation

print("Accuracy:", round(accuracy\_score(y\_test, y\_pred), 4))

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))

# Confusion matrix

plt.figure(figsize=(6, 4))

sns.heatmap(confusion\_matrix(y\_test, y\_pred), annot=True, fmt='d', cmap='Blues')

plt.title("Confusion Matrix")

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.tight\_layout()

plt.show()

# ROC curve

fpr, tpr, \_ = roc\_curve(y\_test, y\_proba)

roc\_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 4))

plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.title("ROC Curve - Logistic Regression")

plt.xlabel("False Positive Rate")

plt.ylabel("True Positive Rate")

plt.legend(loc="lower right")

plt.tight\_layout()

plt.show()

**Output**

