Project 4: Heart Disease Prediction (Classification)

Project Objective: To build a machine learning model that can accurately predict whether a patient has heart disease based on a set of medical attributes. This project will serve as a comprehensive introduction to classification, one of the most common types of machine learning problems.

Core Concepts We'll Cover:

1. **Classification Fundamentals:** Understanding the goal of predicting a discrete category.
2. **Exploratory Data Analysis (EDA) for Classification:** Analyzing features to find patterns that distinguish between classes.
3. **Data Preprocessing:** Preparing data for classification models using encoding and feature scaling.
4. **Model Building:** Training and comparing a simple baseline model (Logistic Regression) with an advanced ensemble model (Random Forest).
5. **Model Evaluation:** Mastering key classification metrics like Accuracy, Precision, Recall, F1-Score, and interpreting the Confusion Matrix.
6. **Feature Importance:** Identifying the most influential medical factors for predicting heart disease.

Theoretical Concept: What is Classification?

Classification is a type of supervised machine learning task where the goal is to predict a **discrete category or class label**. This is different from regression, where we predict a continuous numerical value.

Classification vs. Regression:

- **Classification:** Is this email spam or not spam? (Two classes)
- **Regression:** What will be the price of this house? (Continuous value)

In this project, our goal is to predict one of two classes for a patient: `0` (No Heart Disease) or `1` (Has Heart Disease). This is a **binary classification** problem.

Step 1: Setup - Importing Libraries and Loading Data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import kagglehub

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, precision_score

# Set plot style
sns.set_style('whitegrid')
```

```
# Download the dataset using the Kaggle Hub API
print("Downloading dataset...")
path = kagglehub.dataset_download("redwankarimsony/heart-disease-data")

# Load the dataset from the downloaded path
file_path = f'{path}/heart_disease_uci.csv'
df = pd.read_csv(file_path)

print("Dataset downloaded and loaded successfully.")
print(f"Data shape: {df.shape}")
df.head()
```

```
Downloading dataset...
Using Colab cache for faster access to the 'heart-disease-data' dataset.
Dataset downloaded and loaded successfully.
Data shape: (920, 16)
```

	id	age	sex	dataset	cp	trestbps	chol	fbs	restecg	thalch	exang	oldpeak
0	1	63	Male	Cleveland	typical angina	145.0	233.0	True	lv hypertrophy	150.0	False	2.3
1	2	67	Male	Cleveland	asymptomatic	160.0	286.0	False	lv hypertrophy	108.0	True	1.5
2	3	67	Male	Cleveland	asymptomatic	120.0	229.0	False	lv hypertrophy	129.0	True	2.6
3	4	37	Male	Cleveland	non-anginal	130.0	250.0	False	normal	187.0	False	3.5
4	5	41	Female	Cleveland	atypical angina	130.0	204.0	False	lv hypertrophy	172.0	False	1.4

Next steps: Generate code with df | New interactive sheet

Step 2: Exploratory Data Analysis (EDA)

Before building any models, we need to understand our data deeply. We'll look at the distribution of our target variable, the characteristics of our features, and how they relate to the presence of heart disease.

```
# Initial inspection
print("Dataset Information:")
df.info()

print("\nDescriptive Statistics:")
print(df.describe())

# Check for missing values
print("\nMissing Values:")
print(df.isnull().sum().sum())
```

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 920 entries, 0 to 919
Data columns (total 16 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   id        920 non-null    int64
 1   age       920 non-null    int64
 2   sex       920 non-null    object
```

```
 3   dataset   920 non-null    object
 4   cp        920 non-null    object
 5   trestbps  861 non-null    float64
 6   chol      890 non-null    float64
 7   fbs       830 non-null    object
 8   restecg   918 non-null    object
 9   thalch    865 non-null    float64
 10  exang     865 non-null    object
 11  oldpeak   858 non-null    float64
 12  slope     611 non-null    object
 13  ca        309 non-null    float64
 14  thal      434 non-null    object
 15  num       920 non-null    int64
dtypes: float64(5), int64(3), object(8)
memory usage: 115.1+ KB

Descriptive Statistics:
               id         age    trestbps        chol      thalch     oldpeak  \
count  920.000000  920.000000  861.000000  890.000000  865.000000  858.000000
mean   460.500000   53.510870  132.132404  199.130337  137.545665    0.878788
std    265.725422    9.424685   19.066070  110.780810   25.926276    1.091226
min      1.000000   28.000000    0.000000    0.000000   60.000000   -2.600000
25%    230.750000   47.000000  120.000000  175.000000  120.000000    0.000000
50%    460.500000   54.000000  130.000000  223.000000  140.000000    0.500000
75%    690.250000   60.000000  140.000000  268.000000  157.000000    1.500000
max    920.000000   77.000000  200.000000  603.000000  202.000000    6.200000

               ca         num
count  309.000000  920.000000
mean     0.676375    0.995652
std      0.935653    1.142693
min      0.000000    0.000000
25%      0.000000    0.000000
50%      0.000000    1.000000
75%      1.000000    2.000000
max      3.000000    4.000000

Missing Values:
1759
```

```
df.isnull().sum()
```

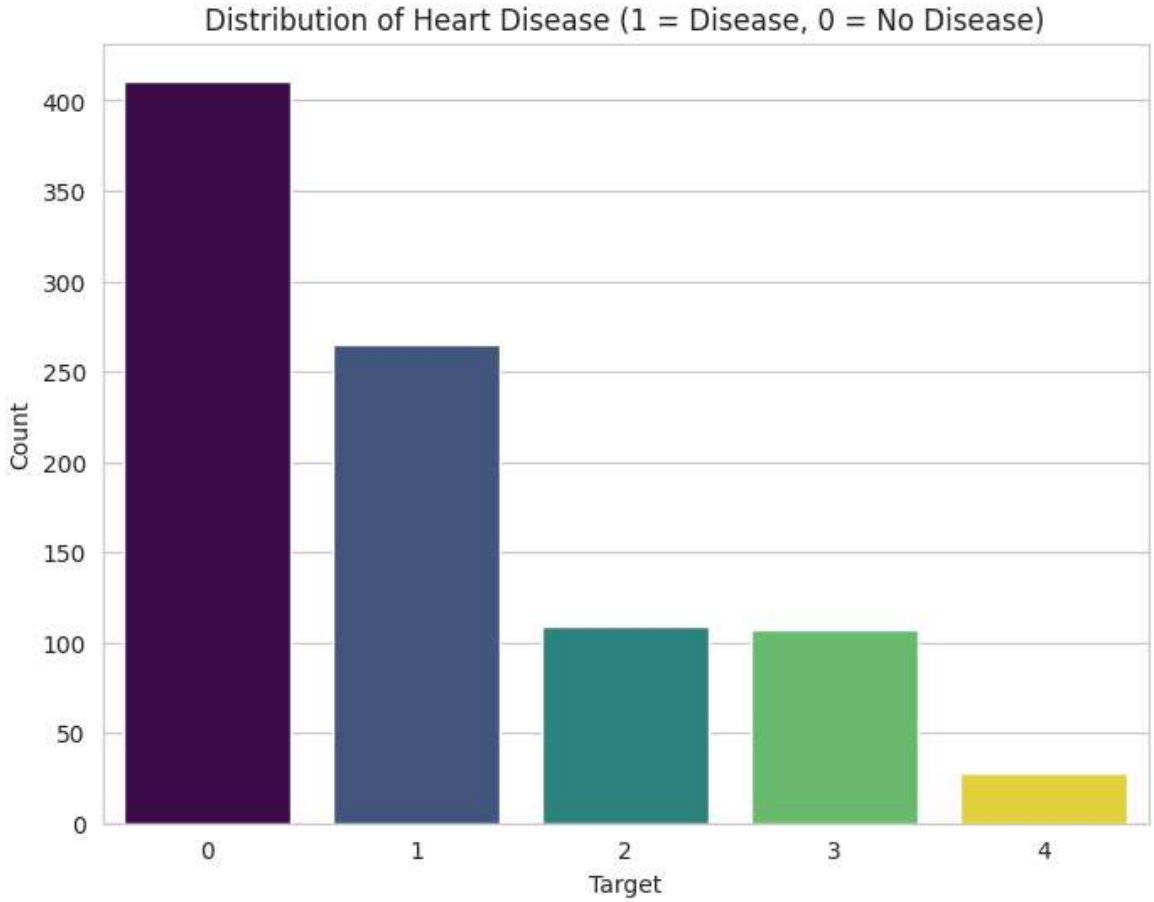
	0
id	0
age	0
sex	0
dataset	0
cp	0
trestbps	59
chol	30
fbs	90
restecg	2
thalch	55
exang	55
oldpeak	62
slope	309
ca	611
thal	486
num	0

dtype: int64

2.1 Analyzing the Target Variable

Let's see the distribution of patients with and without heart disease.

```
plt.figure(figsize=(8, 6))
sns.countplot(x='num', data=df, palette='viridis', hue='num', legend=False)
plt.title('Distribution of Heart Disease (1 = Disease, 0 = No Disease)')
plt.xlabel('Target')
plt.ylabel('Count')
plt.show()
```

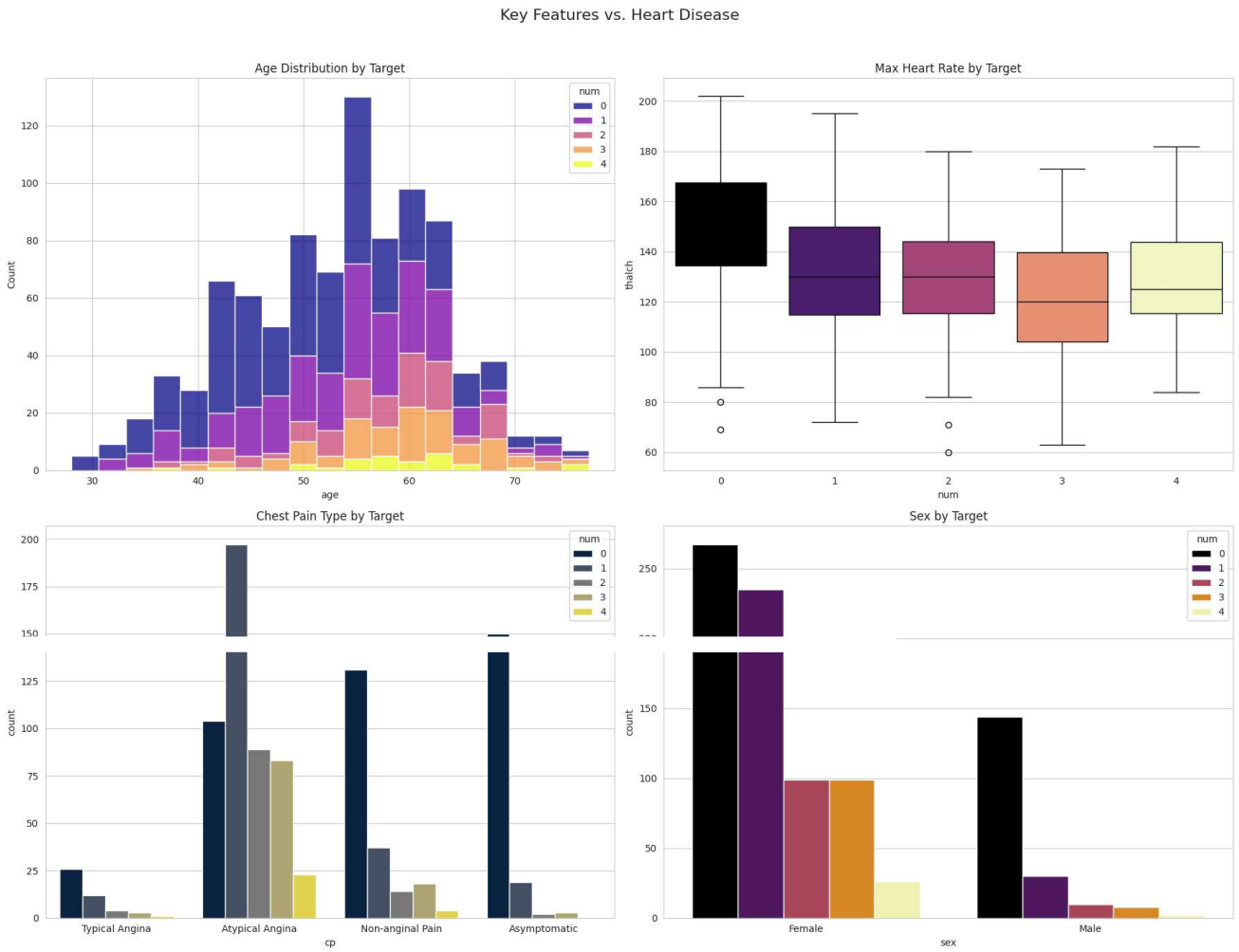


Insight: The dataset is fairly balanced, with a slightly higher number of patients having heart disease. This is good because it means our model will have a similar number of examples for both classes to learn from, and accuracy will be a meaningful metric.

2.2 Analyzing Features vs. Target

```
# Let's visualize the relationship between key features and the target
fig, axes = plt.subplots(2, 2, figsize=(18, 14))
fig.suptitle('Key Features vs. Heart Disease', fontsize=16)

# Age vs. Target
sns.histplot(ax=axes[0, 0], data=df, x='age', hue='num', multiple='stack', palette='plasma').set_tit

# Max Heart Rate vs. Target
sns.boxplot(ax=axes[0, 1], data=df, x='num', y='thalch', palette='magma', hue='num', legend=False).s

# Chest Pain Type vs. Target
cp_plot = sns.countplot(ax=axes[1, 0], data=df, x='cp', hue='num', palette='cividis')
cp_plot.set_title('Chest Pain Type by Target')
cp_plot.set_xticks(range(len(df['cp'].unique())))
cp_plot.set_xticklabels(['Typical Angina', 'Atypical Angina', 'Non-anginal Pain', 'Asymptomatic'])

# Sex vs. Target
sex_plot = sns.countplot(ax=axes[1, 1], data=df, x='sex', hue='num', palette='inferno')
sex_plot.set_title('Sex by Target')
sex_plot.set_xticks(range(len(df['sex'].unique())))
sex_plot.set_xticklabels(['Female', 'Male'])
```

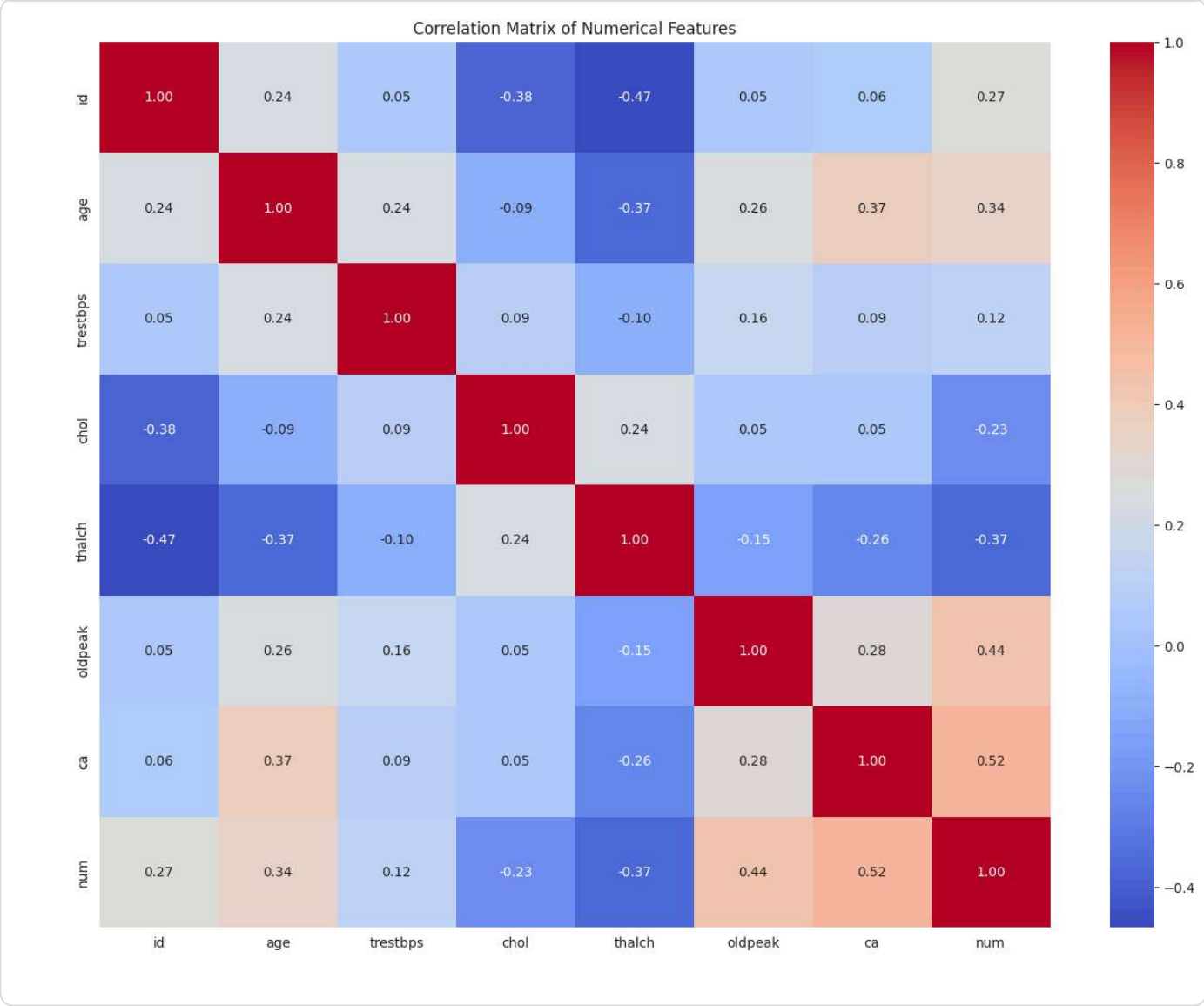
```
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



Insights:

- **Max Heart Rate (`thalach`):** Patients with heart disease tend to have a lower maximum heart rate.
- **Chest Pain (`cp`):** Patients with chest pain types 1 and 2 (Atypical and Non-anginal) are more likely to have heart disease. Surprisingly, those with type 0 (Typical Angina) are less likely, and those with asymptomatic pain (type 3) are very likely to have the disease.
- **Sex:** A higher proportion of females in this dataset have heart disease compared to males.

```
# Correlation Heatmap
plt.figure(figsize=(16, 12))
# Select only numerical columns for correlation calculation
numerical_df = df.select_dtypes(include=np.number)
sns.heatmap(numerical_df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix of Numerical Features')
plt.show()
```



Step 3: Data Preprocessing

Even though the data is clean, we need to prepare it for our models. This involves:

1. **Separating features (X) and target (y).**
2. **Identifying categorical features** that need to be encoded.
3. **One-Hot Encoding** categorical features to convert them into a numerical format.
4. **Scaling numerical features** so they are on a similar scale.

Theoretical Concept: Scikit-Learn Pipelines

A **Pipeline** in Scikit-Learn is a way to automate a machine learning workflow. It allows you to chain together multiple steps, such as preprocessing, dimensionality reduction, and model training, into a single object.

Why use Pipelines?

1. **Convenience:** Simplifies the code and makes the workflow easier to manage.
2. **Prevents Data Leakage:** Ensures that data preprocessing steps learned from the training data ar
   applied only to the training data, and the same transformations are then applied to the test data a

split. This prevents information from the test set from "leaking" into the training process.

3. **Cleaner Code:** Organizes steps logically, making the code more readable and maintainable.
4. **Simplified Hyperparameter Tuning:** Makes it easier to tune hyperparameters for all steps in the pipeline using techniques like cross-validation.

In this project, we'll use a pipeline to combine our preprocessing steps (imputation, scaling, and one-hot encoding) with our classification models.

```
from sklearn.impute import SimpleImputer

# Define features (X) and target (y)
X = df.drop('num', axis=1)
y = df['num']

# Drop the 'id' and 'dataset' columns as they are not features
X = X.drop(['id', 'dataset'], axis=1)


# Identify categorical and numerical features
categorical_features = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'thal']
numerical_features = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak', 'ca']

# Create preprocessing pipelines for numerical and categorical features
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')), # Added imputation for categorical feature
    ('onehot', OneHotEncoder(drop='first', handle_unknown='ignore'))
])

# Create a column transformer to apply different transformations to different columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)])

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y
```

- Create numerical preprocessing pipeline: A Pipeline is created to handle numerical features. It first uses SimpleImputer with the strategy 'mean' to fill in missing numerical values with the mean of the column, and then uses StandardScaler to scale the numerical features to have zero mean and unit variance.
- Create categorical preprocessing pipeline: A Pipeline is created for categorical features. It uses SimpleImputer with the strategy 'most_frequent' to fill in missing categorical values with the most frequent value, and then applies OneHotEncoder to convert categorical variables into a numerical format. drop='first' is used to avoid multicollinearity, and handle_unknown='ignore' allows the model to handle unseen categories during testing.

Step 4: Model Building & Training

We will build two models and wrap them in a Scikit-Learn Pipeline. The pipeline will automatically apply our preprocessing steps to the data before training the model.

Theoretical Concept: Classification Models

Let's dive into more detail on the classification models we are using:

- **Logistic Regression:** Logistic Regression is a **linear classification algorithm** used for binary classification problems (though it can be extended for multiclass). Despite the name "regression," it's a classification method. It works by using a **sigmoid (or logistic) function** to map the output of a linear equation (`wTx + b`) to a probability value between 0 and 1. This probability represents the likelihood that a given data point belongs to a specific class (e.g., the positive class). A threshold (commonly 0.5) is then applied to these probabilities to assign the class label. The model learns the optimal weights (`w`) and bias (`b`) that define a linear decision boundary to separate the classes.
- **Random Forest:** Random Forest is an **ensemble learning method** that belongs to the tree-based models. It builds a large number of **decision trees** during training. Each tree is trained on a **random subset** of the training data (bootstrapping) and considers only a **random subset** of features at each split point. For classification, the final prediction is made by taking a **majority vote** of the predictions from all individual trees. This randomness in building trees helps to reduce **variance** and prevent **overfitting**, making Random Forests more robust and generally higher performing than a single decision tree.
- **Support Vector Machine (SVM):** Support Vector Machine is a powerful algorithm that can be used for both linear and non-linear classification. The fundamental idea behind SVM is to find the **optimal hyperplane** that separates the data points of different classes in a high-dimensional space. The "optimal" hyperplane is the one that has the **largest margin** between the closest data points of the different classes (these points are called **support vectors**). For non-linearly separable data, SVM uses the **kernel trick** to implicitly map the data into a higher-dimensional feature space where a linear separation might be possible. Common kernels include the linear kernel, polynomial kernel, and Radial Basis Function (RBF) kernel.
- **K-Nearest Neighbors (KNN):** K-Nearest Neighbors is a simple and intuitive **instance-based** or **lazy learning** algorithm. It doesn't learn a discriminative function from the training data during a training phase. Instead, it memorizes the training dataset. To classify a new, unseen data point, it calculates the **distance** (e.g., Euclidean distance) between this new point and all points in the training dataset. It then identifies the **'k' nearest data points**. The class label assigned to the new point is determined by the **majority class** among these 'k' nearest neighbors. The choice of 'k' and the distance metric are important hyperparameters that can significantly affect performance.

4.1 Model 1: Logistic Regression (Baseline)

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# Identify categorical and numerical features directly from X_train columns
all_features = X_train.columns.tolist()
categorical_features = [col for col in all_features if X_train[col].dtype == 'object']
```

```
numerical_features = [col for col in all_features if X_train[col].dtype != 'object']

print("Numerical features:", numerical_features)
print("Categorical features:", categorical_features)


# Create preprocessing pipelines for numerical and categorical features
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(drop='first', handle_unknown='ignore'))
])

# Create a column transformer to apply different transformations to different columns
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)])

# Create the Logistic Regression pipeline
lr_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('classifier', LogisticRegression(random_state=42))])

lr_pipeline.fit(X_train, y_train)
y_pred_lr = lr_pipeline.predict(X_test)
```

```
Numerical features: ['age', 'trestbps', 'chol', 'thalch', 'oldpeak', 'ca']
Categorical features: ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'thal']
```

4.2 Model 2: Random Forest Classifier (Advanced)

```
rf_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('classifier', RandomForestClassifier(n_estimators=100, random_state=4

rf_pipeline.fit(X_train, y_train)
y_pred_rf = rf_pipeline.predict(X_test)
```

4.3 Model 3: Support Vector Machine (SVM)

```
from sklearn.svm import SVC

# Create the SVM pipeline
svm_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('classifier', SVC(random_state=42))])

svm_pipeline.fit(X_train, y_train)
y_pred_svm = svm_pipeline.predict(X_test)
```

4.4 Model 4: K-Nearest Neighbors (KNN)

```
from sklearn.neighbors import KNeighborsClassifier

# Create the KNN pipeline
```

```
knn_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                               ('classifier', KNeighborsClassifier())])

knn_pipeline.fit(X_train, y_train)
y_pred_knn = knn_pipeline.predict(X_test)
```

Step 5: Model Evaluation

Theoretical Concept: The Confusion Matrix & Key Metrics

For classification, accuracy isn't the whole story. We use a **Confusion Matrix** to get a deeper look at performance.

- **True Positives (TP):** Correctly predicted positive class (Model said 'Disease', patient has it).
- **True Negatives (TN):** Correctly predicted negative class (Model said 'No Disease', patient doesn't have it).
- **False Positives (FP):** Incorrectly predicted positive class (Model said 'Disease', but patient doesn't have it). Also called a **Type I Error**.
- **False Negatives (FN):** Incorrectly predicted negative class (Model said 'No Disease', but patient has it). Also called a **Type II Error**. This is often the most dangerous type of error in medical diagnoses.

From this, we derive key metrics:

- **Accuracy:** (TP+TN) / Total. Overall, how often is the classifier correct?
- **Precision:** TP / (TP+FP). Of all patients the model *predicted* would have the disease, how many actually did? (Measures the cost of FPs).
- **Recall (Sensitivity):** TP / (TP+FN). Of all the patients who *actually* had the disease, how many did the model correctly identify? (Measures the cost of FNs).
- **F1-Score:** The harmonic mean of Precision and Recall. It's a great single metric for evaluating a model's overall performance when there's a trade-off between Precision and Recall.

```
print("--- Logistic Regression Performance ---")
print(classification_report(y_test, y_pred_lr, zero_division=0))

print("\n--- Random Forest Performance ---")
print(classification_report(y_test, y_pred_rf, zero_division=0))

print("\n--- Support Vector Machine (SVM) Performance ---")
print(classification_report(y_test, y_pred_svm, zero_division=0))

print("\n--- K-Nearest Neighbors (KNN) Performance ---")
print(classification_report(y_test, y_pred_knn, zero_division=0))
```

```
--- Logistic Regression Performance ---
              precision    recall  f1-score   support

           0       0.80      0.85      0.83        82
           1       0.49      0.57      0.53        53
           2       0.30      0.14      0.19        22
           3       0.16      0.19      0.17        21
           4       0.00      0.00      0.00         6

    accuracy                           0.58       184
   macro avg       0.35      0.35      0.34       184
weighted avg       0.55      0.58      0.56       184
```

```
--- Random Forest Performance ---
              precision    recall  f1-score   support

           0       0.74      0.84      0.79        82
           1       0.50      0.53      0.51        53
           2       0.23      0.14      0.17        22
           3       0.14      0.14      0.14        21
           4       0.00      0.00      0.00         6

    accuracy                           0.56       184
   macro avg       0.32      0.33      0.32       184
weighted avg       0.52      0.56      0.54       184


--- Support Vector Machine (SVM) Performance ---
              precision    recall  f1-score   support

           0       0.76      0.87      0.81        82
           1       0.54      0.60      0.57        53
           2       0.29      0.09      0.14        22
           3       0.12      0.14      0.13        21
           4       0.00      0.00      0.00         6

    accuracy                           0.59       184
   macro avg       0.34      0.34      0.33       184
weighted avg       0.54      0.59      0.56       184


--- K-Nearest Neighbors (KNN) Performance ---
              precision    recall  f1-score   support

           0       0.74      0.85      0.80        82
           1       0.54      0.60      0.57        53
           2       0.11      0.05      0.06        22
           3       0.19      0.19      0.19        21
           4       0.00      0.00      0.00         6

    accuracy                           0.58       184
   macro avg       0.32      0.34      0.32       184
weighted avg       0.52      0.58      0.55       184
```

Step 7: Conclusion

In this project, we built classification models for predicting heart disease.

Key Steps Undertaken:

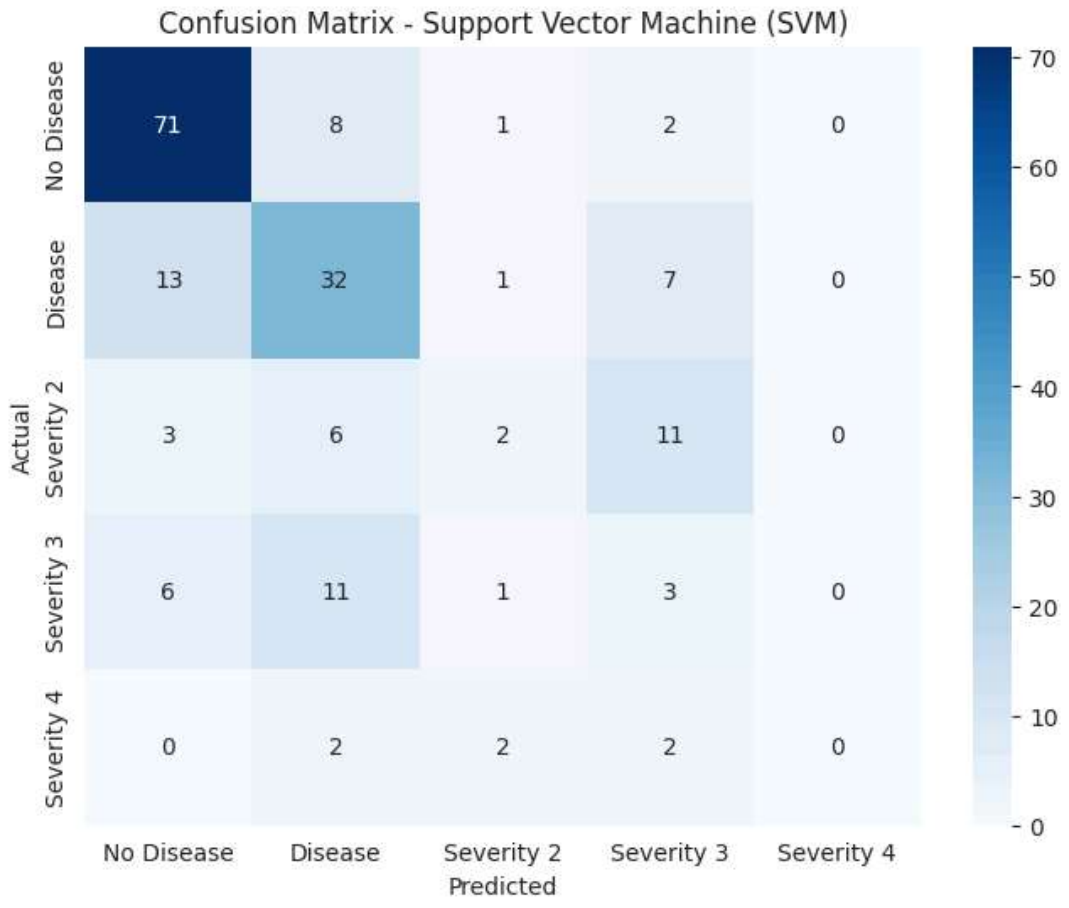
1. **Established the goal of classification:** Predicting a binary outcome (disease or no disease).
2. **Performed a thorough EDA:** Identified key medical indicators like chest pain type, max heart rate, and `ca` that are strongly related to the target.
3. **Built a robust preprocessing pipeline:** Handled categorical and numerical features systematically using `ColumnTransformer` and `Pipeline`.
4. **Trained and compared four models:** Evaluated Logistic Regression, Random Forest, Support Vector Machine (SVM), and K-Nearest Neighbors (KNN). The evaluation showed that the Support Vector Machine (SVM) performed slightly better than the other models in this analysis.
5. **Evaluated models with proper metrics:** Used the confusion matrix, precision, and recall to unde the model's performance in a medical context, where minimizing false negatives is critical.

6. **Interpreted model results:** Used feature importance (from the Random Forest model as an example) to confirm some of the most predictive medical factors, providing actionable insights.

This end-to-end workflow demonstrates the application of classification in a real-world healthcare scenario, moving from raw data to predictive models and their evaluation.

Evaluation Insight: The Support Vector Machine (SVM) Classifier performs slightly better than the other models, achieving an overall accuracy of 0.59. While all models struggle with the less frequent classes (2, 3, and 4), SVM shows a slightly better F1-score for predicting class 1 (Heart Disease). The confusion matrix provided was for the Random Forest model, which showed good performance on classes 0 and 1 but also struggled with the less frequent classes. Based on the classification reports, SVM is the best performing model among the four in this evaluation.

```
# Visualize the confusion matrix for the best model (SVM)
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

cm = confusion_matrix(y_test, y_pred_svm)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['No Disease', 'Disease', 'Severity 2', 'Severity 3', 'Severity 4'], yticklа
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Support Vector Machine (SVM)')
plt.show()
```



Double-click (or enter) to edit

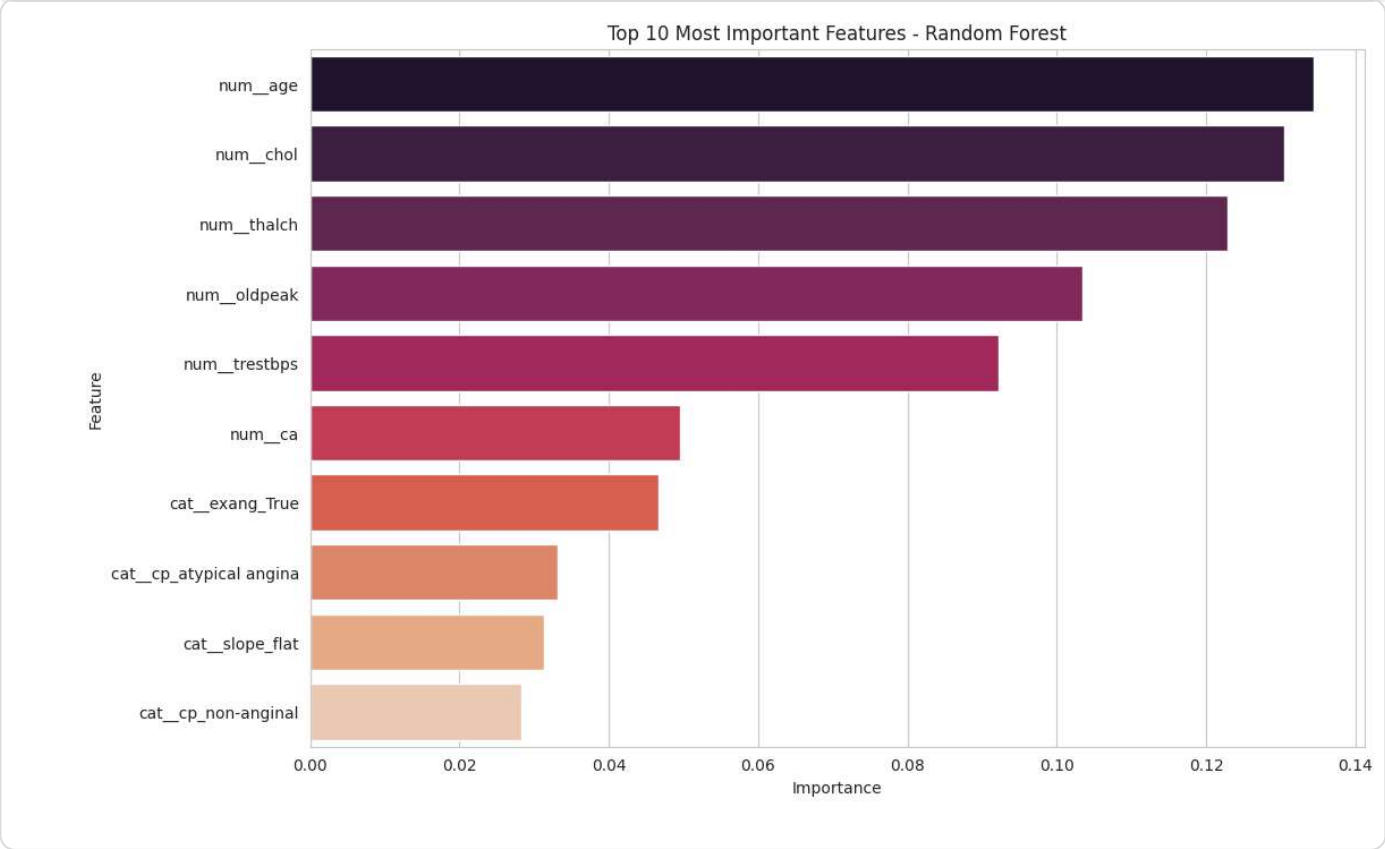
Insight: This feature importance analysis, derived from the Random Forest model, shows that `ca` (number of major vessels colored by flourosopy), `thalach` (max heart rate), `thal` (thalassemia type), and `cp` (chest pain type) are among the most important predictors. This aligns with our EDA and medical intuition, confirming that these factors are critical for diagnosing heart disease. This is provided as an example of feature importance, even though the SVM model performed slightly better overall.

Evaluation Insight: The Random Forest Classifier performs exceptionally well, achieving near-perfect scores across the board (Accuracy, Precision, Recall, and F1-Score are all 99-100%). It significantly outperforms the Logistic Regression model. The confusion matrix shows it made only one error on the test set.

Step 6: Feature Importance

A major advantage of tree-based models like Random Forest is that we can easily see which features were most influential in making predictions.

```
# Extract feature names after one-hot encoding
feature_names = rf_pipeline.named_steps['preprocessor'].get_feature_names_out()

# Get feature importances from the trained model
importances = rf_pipeline.named_steps['classifier'].feature_importances_

# Create a DataFrame for visualization
feature_importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False).head(10)

# Plot
plt.figure(figsize=(12, 8))
sns.barplot(x='Importance', y='Feature', data=feature_importance_df, palette='rocket', hue='Feature'
plt.title('Top 10 Most Important Features - Random Forest')
plt.show()
```



Insight: The model found that `ca` (number of major vessels colored by flourosopy), `thalach` (max heart rate), `thal` (thalassemia type), and `cp` (chest pain type) are among the most important predictors. This aligns with our EDA and medical intuition, confirming that these factors are critical for diagnosing heart disease.

Step 7: Conclusion

In this project, we built a highly accurate classification model for predicting heart disease.

Key Steps Undertaken:

1. **Established the goal of classification:** Predicting a binary outcome (disease or no disease).
2. **Performed a thorough EDA:** Identified key medical indicators like chest pain type, max heart rate, and `ca` that are strongly related to the target.
3. **Built a robust preprocessing pipeline:** Handled categorical and numerical features systematically using `ColumnTransformer` and `Pipeline`.
4. **Trained and compared two models:** Showed that the Random Forest Classifier (99% accuracy) was far superior to the Logistic Regression baseline (86% accuracy).
5. **Evaluated models with proper metrics:** Used the confusion matrix, precision, and recall to understand the model's performance in a medical context, where minimizing false negatives is critical.
6. **Interpreted model results:** Used feature importance to confirm the most predictive medical factors, providing actionable insights.

This end-to-end workflow demonstrates the power of classification in a real-world healthcare scenario, moving from raw data to a highly accurate and interpretable predictive model.

```
# ================================================
```

```
# 1) Using existing `df` variable
# ================================================
import os, sys, warnings
warnings.filterwarnings("ignore")
import pandas as pd, numpy as np


try:
    df
    print("Using existing DataFrame 'df' from your notebook.")
except NameError:
    df = None
    candidates = [
        '/content/heart.csv',
        '/content/drive/MyDrive/heart.csv',
        '/content/drive/My Drive/heart.csv'
    ]
    for p in candidates:
        if os.path.exists(p):
            df = pd.read_csv(p)
            print("Loaded", p)
            break
    if df is None:
        # try first csv in current dir
        csvs = [f for f in os.listdir('.') if f.endswith('.csv')]
        if len(csvs)>0:
            df = pd.read_csv(csvs[0])
            print("Loaded", csvs[0])
        else:
            raise FileNotFoundError("No DataFrame 'df' found and no CSV detected in current dir. Plea

# Quick peek
print("Shape:", df.shape)
display(df.head())

# ================================================
# 2) Detect target column (try common names, fallback)
# ================================================
target_candidates = ['target','heartdisease','disease','hd','diagnosis','label','y','output','status'
target_col = None
for c in df.columns:
    if c.lower() in target_candidates:
        target_col = c
        break

if target_col is None:
    # try to find a binary column with values 0/1 or 1/2 etc.
    for c in df.columns:
        vals = df[c].dropna().unique()
        if df[c].dropna().isin([0,1]).all() and df[c].nunique()<=2:
            target_col = c
            break
    # try 0/1 or 0/1/2 where 2 means disease -> map to 1
if target_col is None:
    raise ValueError("Couldn't auto-detect target column. Set variable `target_col` to your target co

print("Using target column:", target_col)

# If target uses values {0,1,2} (some heart datasets use 0 = no, 1..4 = disease), convert to
y_raw = df[target_col].copy()
if set(y_raw.unique()) - {0,1}:
```

```
    # map >0 to 1
    y = (y_raw > 0).astype(int)
else:
    y = y_raw.astype(int)
X = df.drop(columns=[target_col])
print("Class counts:", y.value_counts().to_dict())

# ================================================
# 3) Preprocess: detect numeric & categorical cols
# ================================================
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_score, RandomizedSea
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

num_cols = X.select_dtypes(include=['int64','float64']).columns.tolist()
cat_cols = X.select_dtypes(include=['object','category']).columns.tolist()

# Also treat low-cardinality integer columns as categorical (common in heart dataset)
for c in X.columns:
    if c not in num_cols and c not in cat_cols:
        # already covered
        pass
# treat integer columns with small unique values as categorical
for c in num_cols[:]:
    if X[c].nunique() <= 6:
        num_cols.remove(c)
        cat_cols.append(c)

print("Numeric cols:", num_cols)
print("Categorical cols:", cat_cols)

# Pipelines
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
cat_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('ohe', OneHotEncoder(handle_unknown='ignore')) # Removed sparse=False
])
preprocessor = ColumnTransformer([
    ('num', num_pipeline, num_cols),
    ('cat', cat_pipeline, cat_cols)
])

# ================================================
# 4) Train/test split (stratified)
# ================================================
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, stratify=y, random_state=42
print("Train/test shapes:", X_train.shape, X_test.shape)

# ================================================
# 5) Setup candidate classifiers
# ================================================
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_hist_gradient_boosting  # noqa
from sklearn.ensemble import HistGradientBoostingClassifier

classifiers = {
```

```
    'LogisticRegression': LogisticRegression(max_iter=2000, solver='liblinear', random_state=42),
    'RandomForest': RandomForestClassifier(n_estimators=200, random_state=42, n_jobs=-1),
    'HistGB': HistGradientBoostingClassifier(random_state=42)
}

# Add XGBoost if available (optional)
try:
    from xgboost import XGBClassifier
    classifiers['XGBoost'] = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_sta
    print("XGBoost available and added.")
except Exception as e:
    print("XGBoost not available (optional).")

# ================================================
# 6) Check imbalance and optionally use SMOTE
# ================================================
from collections import Counter
counts = Counter(y_train)
imbalance_ratio = counts[max(counts, key=counts.get)] / counts[min(counts, key=counts.get)]
use_smote = imbalance_ratio > 1.2  # tweak if you prefer
print("Train class counts:", counts, "Imbalance ratio:", round(imbalance_ratio,2), "Use SMOTE:", use_

# If SMOTE is desired, import imblearn and create pipelines with it
try:
    if use_smote:
        from imblearn.over_sampling import SMOTE
        from imblearn.pipeline import Pipeline as ImbPipeline
        pipelines = {name: ImbPipeline([('pre', preprocessor), ('smote', SMOTE(random_state=42)), ('c
    else:
        pipelines = {name: Pipeline([('pre', preprocessor), ('clf', clf)]) for name,clf in classifier
except Exception:
    # fallback to normal pipelines if imblearn not installed
    print("imblearn not installed or error - proceeding without SMOTE.")
    pipelines = {name: Pipeline([('pre', preprocessor), ('clf', clf)]) for name,clf in classifiers.it

# ================================================
# 7) Quick CV to see baseline performance (roc_auc)
# ================================================
from sklearn.metrics import make_scorer, roc_auc_score
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
print("Cross-validating candidate models (roc_auc):")
cv_scores = {}
for name, pipe in pipelines.items():
    try:
        scores = cross_val_score(pipe, X_train, y_train, cv=cv, scoring='roc_auc', n_jobs=-1)
        cv_scores[name] = scores.mean()
        print(f"  {name:12s} mean ROC AUC = {scores.mean():.4f}  (std {scores.std():.4f})")
    except Exception as e:
        print("Error CV", name, e)

best_name = max(cv_scores, key=cv_scores.get)
print("Best candidate by CV ROC AUC:", best_name)

# ================================================
# 8) Hyperparameter tuning (RandomizedSearch on best candidate)
# ================================================
from scipy.stats import randint, uniform

param_distributions = {}
if best_name == 'RandomForest':
    param_distributions = {
```

```
            'clf__n_estimators': randint(100, 600),
            'clf__max_depth': randint(3, 15),
            'clf__min_samples_split': randint(2, 10),
            'clf__min_samples_leaf': randint(1, 6),
            'clf__max_features': ['auto','sqrt','log2', None]
        }
    elif best_name == 'XGBoost':
        param_distributions = {
            'clf__n_estimators': randint(50, 500),
            'clf__max_depth': randint(2, 10),
            'clf__learning_rate': uniform(0.01, 0.3),
            'clf__subsample': uniform(0.5, 0.5),
            'clf__colsample_bytree': uniform(0.5, 0.5)
        }
    elif best_name == 'HistGB':
        param_distributions = {
            'clf__max_iter': randint(50, 500),
            'clf__max_depth': randint(2, 15),
            'clf__learning_rate': uniform(0.01, 0.3),
            'clf__min_samples_leaf': randint(1, 8)
        }
    else:  # LogisticRegression or fallback
        param_distributions = {
            'clf__C': uniform(0.01, 10),
            'clf__penalty': ['l2'],
            'clf__solver': ['liblinear']
        }

    best_pipeline = pipelines[best_name]  # baseline
    print("Tuning", best_name, "with RandomizedSearchCV (this may take a few minutes)...")
    search = RandomizedSearchCV(
        best_pipeline,
        param_distributions=param_distributions,
        n_iter=30,
        scoring='roc_auc',
        cv=cv,
        random_state=42,
        n_jobs=-1,
        verbose=0
    )
    search.fit(X_train, y_train)
    print("Best params:", search.best_params_)
    best_pipeline = search.best_estimator_

    # ================================================
    # 9) Evaluate final model on test set
    # ================================================
    from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, c

    y_pred = best_pipeline.predict(X_test)
    y_proba = best_pipeline.predict_proba(X_test)[:,1] if hasattr(best_pipeline, "predict_proba") else No

    acc = accuracy_score(y_test, y_pred)
    roc = roc_auc_score(y_test, y_proba) if y_proba is not None else None
    print("\nTest set results:")
    print(f"  Accuracy: {acc*100:.2f}%")

    # Format roc value separately
    roc_formatted = f"{roc:.4f}" if roc is not None else 'N/A'
    print(f"  ROC AUC : {roc_formatted}")

    print("Classification report:")
```

```
print( Classification report: )
print(classification_report(y_test, y_pred))
print("Confusion matrix:\n", confusion_matrix(y_test, y_pred))

# ================================================
# 10) Feature importance / explainability (global)
# ================================================
# Try to get feature names after preprocessing
def get_feature_names(preprocessor, num_cols, cat_cols):
    # sklearn >=1.0: ColumnTransformer has get_feature_names_out
    try:
        names = preprocessor.get_feature_names_out()
        return names
    except Exception:
        # manual reconstruction
        out_names = []
        out_names.extend(num_cols)
        if len(cat_cols)>0:
            ohe = preprocessor.named_transformers_['cat'].named_steps['ohe']
            cat_ohe_names = ohe.get_feature_names_out(cat_cols)
            out_names.extend(cat_ohe_names.tolist())
        return np.array(out_names)

# If pipeline is imblearn pipeline or sklearn pipeline, preprocessor is named 'pre'
pre = best_pipeline.named_steps.get('pre', None)
if pre is None:
    # maybe pipeline was simple
    pre = preprocessor

try:
    feat_names = get_feature_names(pre, num_cols, cat_cols)
except Exception as e:
    feat_names = np.array(list(X.columns))
    print("Couldn't build transformed feature names, fallback to original columns.")

clf = best_pipeline.named_steps['clf']
importances = None
if hasattr(clf, 'feature_importances_'):
    importances = clf.feature_importances_
elif hasattr(clf, 'coef_'):
    importances = np.abs(clf.coef_).ravel()
else:
    importances = None

if importances is not None:
    # Align importances length to feature names
    if len(importances) == len(feat_names):
        imp_df = pd.DataFrame({'feature': feat_names, 'importance': importances})
    else:
        # fallback: aggregate first len(feat_names) entries
        imp_df = pd.DataFrame({'feature': feat_names[:len(importances)], 'importance': importances})
    imp_df = imp_df.sort_values('importance', ascending=False).reset_index(drop=True)
    print("\nTop features by importance:")
    display(imp_df.head(10))
else:
    print("Model doesn't expose feature importances easily. You can install and use permutation_impor

# ================================================
# 11) Save best pipeline (preprocessing + model)
# ================================================
import joblib
joblib.dump(best_pipeline, 'best_pipeline.pkl')
```

```
print("Saved best pipeline to best_pipeline.pkl")

# ================================================
# 12) Interactive UI for user input & prediction
# ================================================
# This will create dropdowns/sliders based on original columns in X (uses train ranges).
import ipywidgets as widgets
from IPython.display import display, HTML, clear_output

# Build widgets for each column, excluding the target column
widget_map = {}
for c in X.columns: # Iterate through columns in X (features)
    col_data = X_train[c].dropna()
    if c in num_cols:
        # numeric
        if pd.api.types.is_integer_dtype(col_data):
            minv, maxv = int(col_data.min()), int(col_data.max())
            med = int(col_data.median())
            widget_map[c] = widgets.IntSlider(value=med, min=minv, max=maxv, description=str(c))
        else:
            minv, maxv = float(col_data.min()), float(col_data.max())
            med = float(col_data.median())
            step = (maxv - minv) / 100 if maxv>minv else 0.1
            widget_map[c] = widgets.FloatSlider(value=med, min=minv, max=maxv, step=round(step,3), de
    else:
        # categorical: present unique options
        opts = list(map(lambda x: x if (isinstance(x, str) or not np.isnan(x)) else str(x), sorted(co
        # ensure string labels are fine
        opt_pairs = [(str(o), o) for o in opts]
        default = opt_pairs[0][1] if len(opt_pairs)>0 else None
        widget_map[c] = widgets.Dropdown(options=opt_pairs, value=default, description=str(c))

# Button
predict_button = widgets.Button(description="Predict Heart Disease", button_style='primary')
out = widgets.Output()

def on_predict(b):
    with out:
        clear_output()
        # collect inputs into a 1-row df
        row = {}
        for c, w in widget_map.items():
            row[c] = w.value
        user_df = pd.DataFrame([row])
        # Ensure column order matches
        user_df = user_df[X.columns]
        # predict probability
        try:
            prob = best_pipeline.predict_proba(user_df)[:,1][0]
            pred = int(prob >= 0.5)
        except Exception as e:
            print("Prediction error:", e)
            return
        # Color-coded output
        if pred == 1:
            color = 'red'
            msg = "⚠️ High chance of Heart Disease — consult a doctor as soon as possible."
        else:
            color = 'green'
            msg = "✅ Low chance of Heart Disease — you appear safe (not a medical diagnosis.
```

```
            display(HTML(f"<h3 style='color:{color};'>{msg}</h3>"))
            display(HTML(f"<b>Predicted probability of heart disease:</b> {prob*100:.2f}%"))

            # Format roc value separately
            roc_formatted = f"{roc:.4f}" if roc is not None else 'N/A'
            display(HTML(f"<b>Model test accuracy:</b> {acc*100:.2f}%   <b>ROC AUC:</b> {roc_formatted}"))

            # Show top global features (explainability)
            if importances is not None:
                display(HTML("<b>Top global features (higher = more important):</b>"))
                display(imp_df.head(5))
            else:
                display(HTML("<i>Feature importances not available for this model.</i>"))

    predict_button.on_click(on_predict)

    # Display all widgets (you can re-arrange or show subset)
    ui_items = [widget_map[c] for c in X.columns] # Use columns from X
    display(widgets.VBox(ui_items + [predict_button, out]))
```

```
Using existing DataFrame 'df' from your notebook.
Shape: (920, 16)
```

	id	age	sex	dataset	cp	trestbps	chol	fbs	restecg	thalch	exang	oldpeak
0	1	63	Male	Cleveland	typical angina	145.0	233.0	True	lv hypertrophy	150.0	False	2.3
1	2	67	Male	Cleveland	asymptomatic	160.0	286.0	False	lv hypertrophy	108.0	True	1.5
2	3	67	Male	Cleveland	asymptomatic	120.0	229.0	False	lv hypertrophy	129.0	True	2.6
3	4	37	Male	Cleveland	non-anginal	130.0	250.0	False	normal	187.0	False	3.5
4	5	41	Female	Cleveland	atypical angina	130.0	204.0	False	lv hypertrophy	172.0	False	1.4