**Code 1**

Imports necessary libraries for machine learning and data processing. Loads data from CSV file with the correct delimiter before extracting features and labels from the loaded data, such as color score, symmetry score and blue/white score, while assuming “biopsed” column contains the class label (0 or 1). It then calculates class weights to handle class imbalance and filter out images diagnosed as melanoma for cross-validation as well as defining and training the classifier with adjusted class weights. It also splits the data into training and testing sets using train\_test\_split, with a test size of 20%, stratified by the labels to maintain the class distribution in both sets, before training the classifier using the training data (X\_train and y\_train) via the fit method. Lastly it evaluates the trained classifier using the testing data (X\_test and y\_test) by making predictions (y\_pred) and printing the classification report and confusion matrix and saves the trained classifier.

**Code 2**

Imports necessary libraries for machine learning and data processing. Loads data from CSV file with the correct delimiter before extracting features and labels such as color score, symmetry score, and blue/white score, while assuming “biopsed” column contains the class labels (0 or 1). It the calculates class weight to handle class imbalance an defines a random state for train-test split where it generates a random state each time. The weights are calculated based on the distribution of class labels in the dataset, this random state is used to ensure randomness in splitting the data into training and testing sets. It further splits the data into training and testing sets, with a test size of 20% and defines and trains the classifier with adjusted class weights. Lastly it evaluates the classifier and saves the trained classifier.

This code is similar to the previous but introduces a random state for train-test split to ensure reproducibility. It loads data, preprocesses it, trains a Support Vector Classifier (SVC) model with adjusted class weights to handle imbalance, evaluates the model's performance, and saves the trained model for future use.

**Code 3**

Imports necessary libraries for machine learning and data processing. Defines the function for classification that takes a list of features and a trained classifier as input. It uses the classifier to predict the probabilities of each sample belonging to a particular class. Assuming binary classification, it extracts the probability of class 1. It then loads data from a CSV file and extracts features from the loaded data. The features selected are 'color\_score', 'symmetry\_score', and 'blue\_white\_score'. It loads a trained classifier and uses the loaded classifier to classify each image based on its features, obtaining the probabilities of each sample belonging to class 1. Lastly it prints the probabilities for each image along with its corresponding image path.

**Code 4**

Imports necessary libraries for data manipulation, machine learning, evaluation, visualization, and model serialization before loading data from a CSV file. It then extracts features (X) and labels (y) from the loaded data. Features include 'color\_score', 'symmetry\_score', and 'blue\_white\_score', while the labels are assumed to be stored in the 'diagnostic' column and defines a Support Vector Classifier (SVC) pipeline using make\_pipeline, which includes feature scaling with StandardScaler and the SVC model with probability estimation enabled. Further it defines a cross-validation strategy using Stratified K-Fold (StratifiedKFold) with 5 folds, shuffling the data, and setting a random state to ensure reproducibility before performing cross-validation predictions using cross\_val\_predict, which generates cross-validated estimates for each input data point, returning predicted probabilities for each class using the specified method. It then calculates the mean and standard deviation of the predicted probabilities across all folds and prints the mean probabilities and standard deviations. Lastly it plots a bar chart with error bars representing the mean probabilities for each class, with labels and a title for clarity.

**Code 5**

This code generates learning curves for a Support Vector Classifier (SVC) model using the provided dataset. First it imports necessary libraries for data manipulation, machine learning, and visualization then it loads data from a CSV file before extracting features (X) and labels (y) from the loaded data. Features include 'color\_score', 'symmetry\_score', and 'blue\_white\_score', while the labels are assumed to be stored in the 'diagnostic' column. It splits the data into training and test sets using train\_test\_split, with a test size of 20%, stratified by the labels to maintain the class distribution in both sets. Then it defines a Support Vector Classifier (SVC) pipeline using make\_pipeline, which includes feature scaling with StandardScaler and the SVC model with probability estimation enabled before calculating learning curves using learning\_curve. Learning curves visualize the change in training and validation scores as the number of training examples increases. It uses 5-fold cross-validation and specifies a range of training sizes. Lastly it calculates the mean and standard deviation of training and validation scores across the cross-validation folds before plotting the learning curves using matplotlib.pyplot. The plot includes training and cross-validation scores as a function of the number of training examples. The shaded areas around the curves represent the standard deviation of the scores.

**Code 6**

This code performs the training, evaluation, and serialization of a Decision Tree classifier for a binary classification task. Initially, it loads data from a CSV file and extracts features and labels. Then, it calculates class weights to handle any class imbalance within the dataset. After splitting the data into training and testing sets, it defines a pipeline for the Decision Tree classifier, incorporating feature scaling with StandardScaler. The classifier is trained on the training data with adjusted class weights to mitigate the impact of class imbalance. Subsequently, the trained classifier is evaluated on the test data, generating a classification report and confusion matrix to assess its performance. Finally, the trained classifier is serialized and saved to a file. Overall, this code encapsulates the complete workflow for training and deploying a Decision Tree classifier for binary classification.

**Code 7**

This code performs probability estimation using a trained Decision Tree classifier for a binary classification task. It starts by loading data from a CSV file and extracting the relevant features. Then, it loads a pre-trained Decision Tree classifier from a file. Subsequently, it defines a function, classify\_features, to predict probabilities using the trained classifier for each input feature set. Specifically, it extracts the probability of class 1, assuming binary classification. Finally, it iterates through the data, classifies each image based on its features, and prints the probability of each image being classified as class 1 (e.g., the probability of melanoma). This code enables the estimation of probabilities for each class label using the trained Decision Tree classifier, facilitating more nuanced analysis beyond simple binary classification outcomes.

**Code 8**

This code conducts cross-validated probability estimation using a Decision Tree classifier for a binary classification task. Initially, it loads data from a CSV file and extracts features and labels. Then, it defines a pipeline for the Decision Tree classifier, including feature scaling with StandardScaler. Next, it sets up a cross-validation strategy using Stratified K-Fold with 5 splits, ensuring class balance in each fold and fixing the random state for reproducibility. The code then performs cross-validated predictions of class probabilities using the cross\_val\_predict function, specifying method='predict\_proba' to obtain probability estimates. Afterward, it calculates the mean and standard deviation of the predicted probabilities across all folds. Finally, it plots a bar chart with error bars representing the mean probabilities for each class label, providing insight into the model's confidence in its predictions. Overall, this code facilitates the assessment of uncertainty and confidence in the Decision Tree classifier's probability estimates through cross-validation and visualization of mean probabilities with error bars.

**Code 9**

This code conducts a learning curve analysis for a Decision Tree classifier, aiming to evaluate the model's performance as the size of the training data increases. Initially, it loads data from a CSV file and extracts features and labels. Then, it splits the data into training and test sets using a fixed random state for reproducibility and maintaining class balance across splits. The code defines a pipeline for the Decision Tree classifier, incorporating feature scaling with StandardScaler and setting class weights to handle imbalance. Next, it calculates learning curves using the learning\_curve function, specifying a range of training sizes and 5-fold cross-validation. It computes the mean and standard deviation of training and validation scores across different training sizes. Finally, it visualizes the learning curves using Matplotlib, plotting the training and cross-validation scores as a function of the number of training examples, with shaded areas representing the standard deviation around the mean scores. This analysis aids in understanding how model performance evolves with varying amounts of training data, helping to identify potential issues such as overfitting or underfitting.

**Code 10**

This code implements a K-Nearest Neighbors (KNN) classifier for a binary classification task. It begins by loading data from a CSV file and extracting features and labels. Then, it calculates class weights to address any imbalance in the dataset. After splitting the data into training and test sets, it defines a pipeline for the KNN classifier, including feature scaling with StandardScaler. The classifier is trained on the training data with adjusted class weights to account for imbalance. Subsequently, it evaluates the classifier's performance on the test data, generating a classification report and confusion matrix to assess its accuracy and other metrics. Finally, it saves the trained classifier to a file, enabling its reuse without the need for retraining. Overall, this code encapsulates the complete workflow for training, evaluating, and saving a KNN classifier for binary classification.

**Code 11**

This code performs probability estimation using a pre-trained K-Nearest Neighbors (KNN) classifier for a binary classification task. It first loads data from a CSV file and extracts the relevant features. Then, it defines a function, classify\_features, to predict probabilities using the trained classifier for each input feature set, assuming binary classification and outputting the probability of class 1. After loading the pre-trained KNN classifier from a file, it uses this classifier to predict probabilities for each image based on its features. Finally, it iterates through the data, classifies each image, and prints the probability of each image being classified as class 1 (e.g., the probability of melanoma). This code enables the estimation of probabilities for each class label using the pre-trained KNN classifier, facilitating more nuanced analysis beyond simple binary classification outcomes.

**Code 12**

This code snippet focuses on evaluating the performance and uncertainty estimation of a K-Nearest Neighbors (KNN) classifier for a binary classification task. Initially, it loads data from a CSV file and separates features and labels. Following this, it sets up a pipeline for the KNN classifier, incorporating feature scaling with StandardScaler. Subsequently, it defines a cross-validation strategy using Stratified K-Fold with 5 splits to ensure balanced class distribution in each fold. The code then leverages cross\_val\_predict to perform cross-validated predictions of class probabilities, employing the method='predict\_proba' parameter to obtain probability estimates. After computing the mean and standard deviation of the predicted probabilities across all folds, it prints these statistics. Lastly, it visualizes the mean probabilities with error bars representing the standard deviations using Matplotlib, providing insight into the classifier's confidence in its predictions for each class label. Overall, this code offers a comprehensive assessment of the KNN classifier's performance and uncertainty through cross-validation and visualization of probability estimates.

**Code 13**

This code snippet is dedicated to assessing the learning behavior of a K-Nearest Neighbors (KNN) classifier through learning curve analysis. It starts by loading data from a CSV file and segregating features and labels. Subsequently, it splits the data into training and test sets using a fixed random state for reproducibility and maintaining class balance across splits. The code then constructs a pipeline for the KNN classifier, incorporating feature scaling with StandardScaler. Following this, it calculates learning curves using the learning\_curve function, specifying a range of training sizes and 5-fold cross-validation. It computes the mean and standard deviation of training and validation scores across different training sizes. Finally, it plots the learning curves using Matplotlib, showcasing the training and cross-validation scores as a function of the number of training examples. The shaded areas around the mean scores denote the standard deviation, aiding in understanding the variance in the model's performance with varying training data sizes. Overall, this analysis provides valuable insights into the KNN classifier's behavior concerning training data size, facilitating the assessment of model complexity and generalization capabilities.

**Code 14**

This code snippet is dedicated to training and evaluating a Random Forest classifier for a binary classification task. Initially, it loads data from a CSV file and separates features and labels. Then, it computes class weights to address any class imbalance within the dataset. Following this, it splits the data into training and test sets, ensuring class balance in each split. The code constructs a pipeline for the Random Forest classifier, integrating feature scaling with StandardScaler and incorporating the computed class weights to handle imbalance. Subsequently, it trains the classifier on the training data. After training, it evaluates the classifier's performance on the test data, generating a classification report and confusion matrix to assess its accuracy and other metrics. Finally, it saves the trained classifier to a file, facilitating its reuse without retraining. Overall, this code encapsulates the complete workflow for training, evaluating, and saving a Random Forest classifier for binary classification.

**Code 15**

This code segment focuses on utilizing a pre-trained Random Forest classifier to predict probabilities for a binary classification task. Initially, it loads data from a CSV file and extracts the relevant features. Subsequently, it loads the pre-trained Random Forest classifier from a file. The code then defines a function, classify\_features, to predict probabilities using the loaded classifier for each input feature set, assuming binary classification and outputting the probability of class 1. Following this, it iterates through the data, classifies each image using the Random Forest classifier, and prints the probability of each image being classified as class 1 (e.g., the probability of melanoma). Overall, this code enables the estimation of probabilities for each class label using the pre-trained Random Forest classifier, facilitating more nuanced analysis beyond simple binary classification outcomes.

**Code 16**

This code snippet conducts a cross-validated evaluation of a Random Forest classifier's performance for a binary classification task. Initially, it loads data from a CSV file and separates features and labels. Then, it defines a pipeline for the Random Forest classifier, incorporating feature scaling with StandardScaler. Next, it specifies a Stratified K-Fold cross-validation strategy with 5 folds to ensure balanced class distributions in each fold. The code then performs cross-validation predictions, obtaining probability estimates for each class label. Afterward, it calculates the mean and standard deviation of these predictions for each class. Subsequently, it prints the mean probabilities and standard deviations. Finally, it plots a grouped bar chart displaying the mean probabilities for both classes along with error bars representing the standard deviations, providing insights into the classifier's predictive performance across different classes. This analysis aids in understanding the uncertainty and variability associated with the classifier's predictions.

**Code 17**

This code snippet performs a learning curve analysis for a Random Forest classifier applied to a binary classification task. Initially, it loads data from a CSV file and separates features and labels. Then, it splits the data into training and test sets, ensuring stratification to preserve the class distribution. Next, it defines a pipeline for the Random Forest classifier, incorporating feature scaling with StandardScaler and balanced class weights. The code proceeds to calculate learning curves, which represent how the classifier's performance changes with varying training set sizes. It computes mean and standard deviation of training and cross-validation scores across different training set sizes. Finally, it plots the learning curves, illustrating the training and cross-validation scores against the number of training examples, enabling the assessment of model performance and potential overfitting or underfitting issues.

**Code 18**

This code is designed to train a classifier, specifically a Support Vector Machine (SVM), using generated images. Initially, it loads data from a CSV file containing image features and labels. Then, it handles class imbalance by computing class weights. The code proceeds to split the data into training and testing sets, applying Synthetic Minority Over-sampling Technique (SMOTE) to the training data to address class imbalance further. Data augmentation is performed using Keras' ImageDataGenerator, applying transformations like rotation, shifting, shearing, zooming, and horizontal flipping to the resampled training data. The classifier pipeline, including standard scaling and SVM, is defined and trained using the adjusted class weights and augmented data batches. After training, the classifier's performance is evaluated on the test set, with a classification report and confusion matrix printed. Finally, the trained classifier is saved for future use.