Melanoma Skin Cancer Dataset of 10000 Images

The Team Members

Name	*ID*
Farah Mohamed Hegazy	20221452894
Shereen Mohamed Saleh	20221445288
Maryam Wael Mohamed	20221444844
Habiba Yasser Mohamed	20221377183
Sara Mohamed Ahmed	20221380031
Kholoud Waheed Mohamed	20221444910
Habiba Mohamed Atia	20221379966
Shahd Ahmed Karam	20221380283
Nour Ashraf Mohamed	210101067

INTRODUCTION

Melanoma skin cancer is a deadly form of cancer that can significantly impact a person's life. Early detection and accurate classification of melanoma can greatly improve the chances of successful treatment and potentially save lives. This project aims to develop a deep learning model for the accurate classification of melanoma using a dataset containing 10,000 skin cancer images. The dataset consists of 9,600 images for training the model and 1,000 images for evaluating the model's performance.

PROJECT OVERVIEW

Dataset : (https://www.kaggle.com/datasets/hasnainjaved/melanoma-skin-cancer-dataset-of-10000-images/data)

Dataset Description:

The dataset comprises 10,000 images of skin lesions associated with melanoma skin cancer. Each image is labeled with the corresponding melanoma or non-melanoma class. The datasetis divided into two subsets: a training set with 9,600 images and an evaluation set with 1,000images. The images vary in size, color, and texture, representing different types and stages of melanoma. The dataset is carefully curated and annotated by medical professionals toensure accurate labeling and reliable ground truth.

Project Idea:

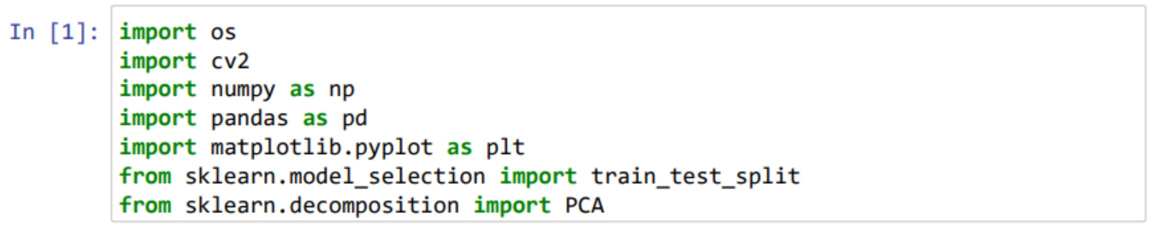
- Develop a deep learning model capable of accurately classifying melanoma skin cancer.
- Train the model using the 9,600 images in the training set.
- Achieve a high classification accuracy, sensitivity, and specificity to ensure reliable melanoma detection.
- Explore different deep learning architectures and techniques to optimize the model's performance.
- Investigate the interpretability of the model to understand the features and patterns contributing to melanoma classification.

Methodology:

- Preprocess the dataset by resizing images, normalizing pixel values, and augmenting the training set to increase its size and diversity.
- Split the dataset into training and evaluation sets according to the provided distribution.
- Experiment with various deep learning architectures such as Convolutional Neural Networks (CNNs), Support Vector Machine (SVM).
- Train the deep learning model using the training set and optimize hyperparameters to improve performance.
- Implement appropriate data augmentation techniques to enhance the model's ability to generalize and handle variations in image appearance.
- Evaluate the model's performance using the evaluation set, calculating metrics such as accuracy, sensitivity, specificity, precision, and F1-score.

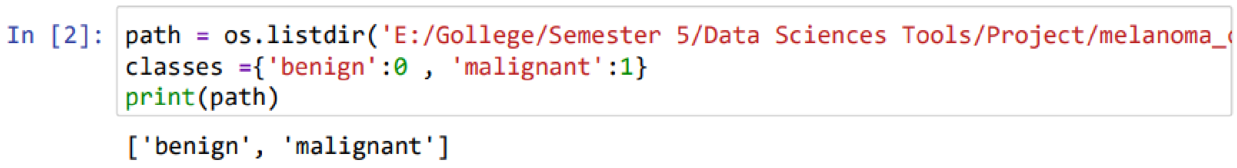
Data Preparation

The code begins by importing necessary libraries and modules such as `os` for operating system related tasks, `cv2` for image processing, `numpy` for numerical computations, `pandas` for data manipulation, `matplotlib.pyplot` for plotting, and specific modules from `sklearn` (Scikit-learn) for machine learning tasks.



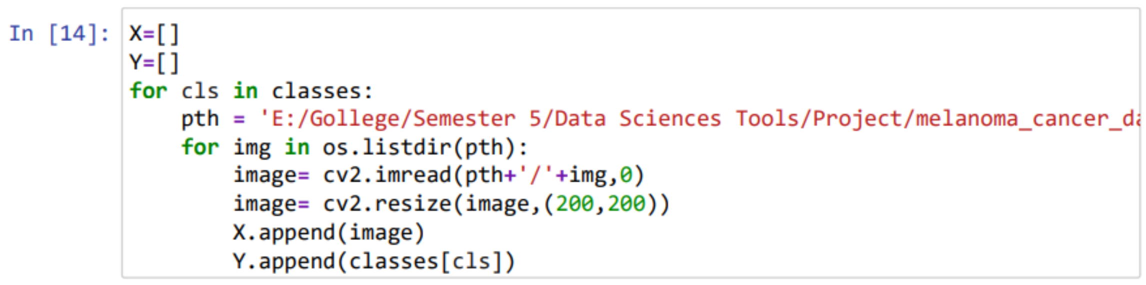
```
In [1]: import os
        import cv2
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from sklearn.model_selection import train_test_split
        from sklearn.decomposition import PCA
```

path line :This line of code attempts to list the files and directories in the specified directory path.
classes line: This line defines a dictionary `classes` where the keys represent the class labels ('benign' and 'malignant') and the values represent the corresponding numeric class labels (0 and 1).

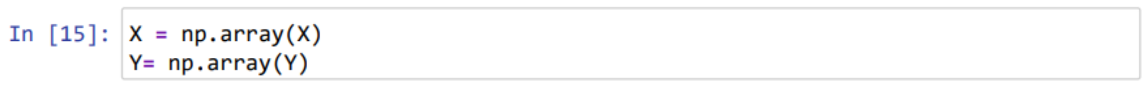


```
In [2]: path = os.listdir('E:/Gollege/Semester 5/Data Sciences Tools/Project/melanoma_
        classes ={'benign':0 , 'malignant':1}
        print(path)

        ['benign', 'malignant']
```

This section of code iterates over the classes defined in `classes`. For each class, it retrieves the list of image files in the specified directory (`pth`). It reads each image using `cv2.imread`, converts it to grayscale (as denoted by `0`), and resizes it to a shape of (200, 200) pixels. The resized image is then appended to the X list, and the corresponding class label is appended to the Y list.
Line 15: These two lines convert the X and Y lists into NumPy arrays for further processing.

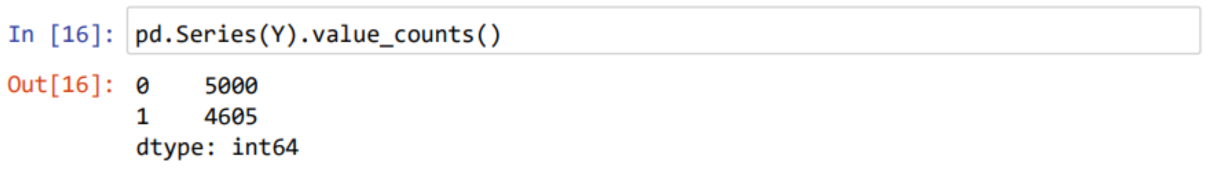


```
In [14]: X=[]
         Y=[]
         for cls in classes:
             pth = 'E:/Gollege/Semester 5/Data Sciences Tools/Project/melanoma_cancer_da
             for img in os.listdir(pth):
                 image= cv2.imread(pth+'/'+img,0)
                 image= cv2.resize(image,(200,200))
                 X.append(image)
                 Y.append(classes[cls])
```



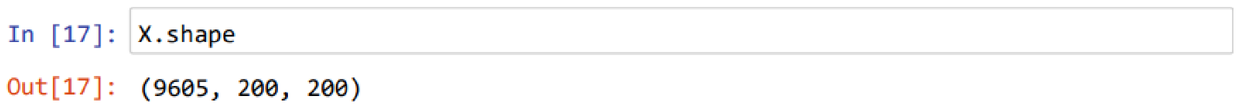
```
In [15]: X = np.array(X)
         Y= np.array(Y)
```

This line creates a pandas Series from the Y array and counts the occurrences of each class label. It provides a count of how many samples belong to each class.



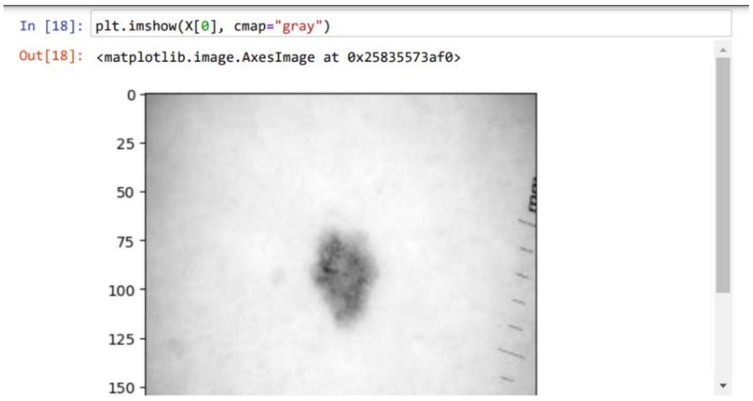
```
In [16]: pd.Series(Y).value_counts()

Out[16]: 0    5000
         1    4605
         dtype: int64
```

This line prints the shape of the X array, which represents the number of samples and the dimensions of each sample.



```
In [17]: X.shape

Out[17]: (9605, 200, 200)
```

This line displays the first image in the X array using `plt.imshow` with a grayscale color map.



```
In [18]: plt.imshow(X[0], cmap="gray")
Out[18]: <matplotlib.image.AxesImage at 0x25835573af0>
```

This line displays the first image in the X array using `plt.imshow` with a grayscale color map.

```
In [19]: X_Updated = X.reshape(len(X),-1)
         X_Updated.shape

Out[19]: (9605, 40000)
```

These lines split the `X_Updated` and Y arrays into training and testing sets using the `train_test_split` function from `sklearn.model_selection`. The `random_state` parameter is set to 10 for reproducibility. The shapes of the resulting arrays `xtrain`, `xtest`, `ytrain`, and `ytest` are printed.

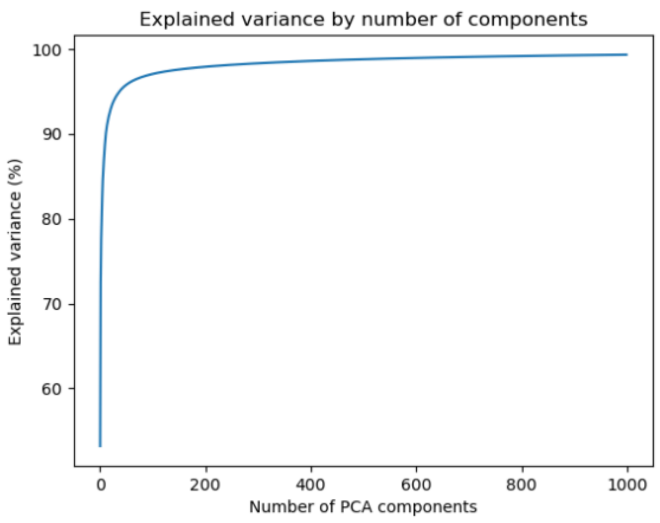
```
In [20]: xtrain,xtest, ytrain,ytest = train_test_split(X_Updated,Y,random_state=10,test

In [21]: xtrain.shape , xtest.shape

Out[21]: ((6723, 40000), (2882, 40000))
```

These lines normalize the pixel values in the `xtrain` and `xtest` array by dividing them by 255, which scales the values between 0 and 1.

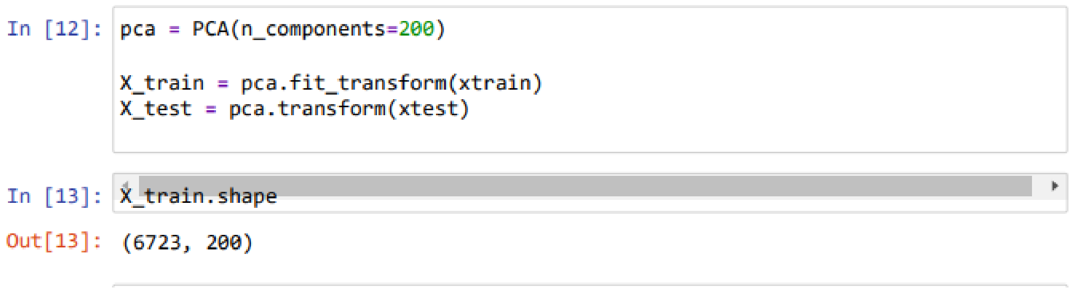
```
In [22]: xtrain = xtrain/255
         xtest = xtest/255
```

These lines perform Principal Component Analysis (PCA) on the `xtrain` data. The PCA class from `sklearn.decomposition` is used with `n_components=1000`, indicating that the algorithm should retain 1000 principal components. The `fit` method is called on the `xtrain` data to fitthe PCA model to the training data. The explained variance ratio for each principal component is computed using the `explained_variance_ratio_` attribute of the PCA model. The component indices and explained variance percentages are stored in the `components` and `explained_variance_percent` variables, respectively.
create a figure and a subplot using `plt.figure` and `fig.add_subplot`. The plot's title, x-axis label, and y-axis label are set using the corresponding methods. Finally, a line plot is created using `ax.plot` to visualize the cumulative sum of the explained variance percentages.

```
In [23]: pca = PCA(n_components=1000).fit(xtrain)

         # Collect the explained variance of each component
         explained_variance = pca.explained_variance_ratio_

         # Component indices
         components = [i for i in range(0, len(explained_variance))]

         # Explained variance in percents
         explained_variance_percent = [100 * i for i in explained_variance]

         fig = plt.figure()
         ax = fig.add_subplot(1, 1, 1)
         ax.set_title('Explained variance by number of components')
         ax.set_ylabel('Explained variance (%)')
         ax.set_xlabel('Number of PCA components')

         # Cumulative sum of the explained variance
         ax.plot(components, np.cumsum(explained_variance_percent))
Out[23]: [<matplotlib.lines.Line2D at 0x258355dc700>]
```



These lines create a new PCA object with `n_components=200` to reduce the dimensionality of the data. The `fit_transform` method is called on the `xtrain` data to perform dimensionality reduction on the training set, and the `transform` method is called on the `xtest` data to transform the testing set. The reduced training data is stored in the `X_train` variable, and the reduced testing data is stored in the `X_test` variable. The shape of the `X_train` array is printed.



```
In [12]: pca = PCA(n_components=200)

         X_train = pca.fit_transform(xtrain)
         X_test = pca.transform(xtest)

In [13]: X_train.shape

Out[13]: (6723, 200)
```

`OVERALL, This code performs the following steps:`

1. Reads and preprocesses images from a specified directory.
2. Splits the data into training and testing sets.
3. Normalizes the pixel values.
4. Applies PCA to reduce the dimensionality of the training and testing data.
5. Plots the explained variance by the number of components.
6. Prints the shape of the reduced training data.

Convolutional Neural Network

What is CNN ?

A CNN, or Convolutional Neural Network, is a type of deep learning algorithm designed for processing and analyzing visual data, such as images and videos. It is particularly effective in tasks like image recognition, object detection, and image classification. CNNs have been successful in various applications, including computer vision, medical image analysis, and autonomous vehicles.

why do we apply CNN on a skin cancer image data ?

Applying Convolutional Neural Networks (CNNs) to skin cancer image data has proven to be a valuable and effective approach for several reasons:

1. **Image Features and Patterns:** Skin cancer images often contain intricate patterns and subtle features that are crucial for accurate diagnosis. CNNs are adept at learning hierarchical features from data, automatically extracting relevant patterns at different levels of abstraction. This makes CNNs well-suited for capturing the complex textures and structures present in skin cancer images.

2. **Translation Invariance:** CNNs exhibit translation invariance, meaning they can recognize patterns regardless of their position in the image. This property is crucial in medical imaging, where the precise location of a lesion or anomaly may vary from one image to another. CNNs' ability to detect features irrespective of their spatial location is beneficial for identifying skin cancer across diverse images.

3. **Parameter Sharing:** CNNs utilize parameter sharing in convolutional layers, which helps in reducing the number of parameters in the network. This is particularly advantageous for medical image datasets, which are often limited in size. The shared weights enable the network to generalize well to new examples, even with a relatively small amount of training data.

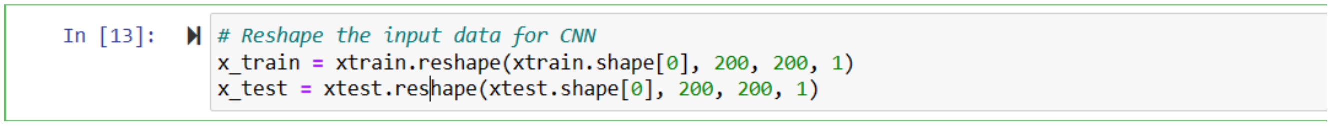
4. **Local Connectivity:** The local connectivity in CNNs allows them to focus on small, local regions of the input image, facilitating the detection of specific features. In skin cancer diagnosis, local features such as irregular borders, color variations, and asymmetry are important indicators. CNNs can learn to recognize these features through the use of convolutional filters.

5. **Transfer Learning:** CNNs trained on large image datasets (e.g., ImageNet) can be fine-tuned for specific tasks, including skin cancer classification. Transfer learning leverages the knowledge gained from a broader dataset and adapts it to a more specific domain, often leading to improved performance, especially when limited labeled data is available for the target task.

6. **Automated Diagnosis:** CNNs have the potential to automate the diagnosis of skin cancer, providing a quick and objective analysis of medical images. This can be particularly beneficial for early detection, allowing for timely intervention and treatment.

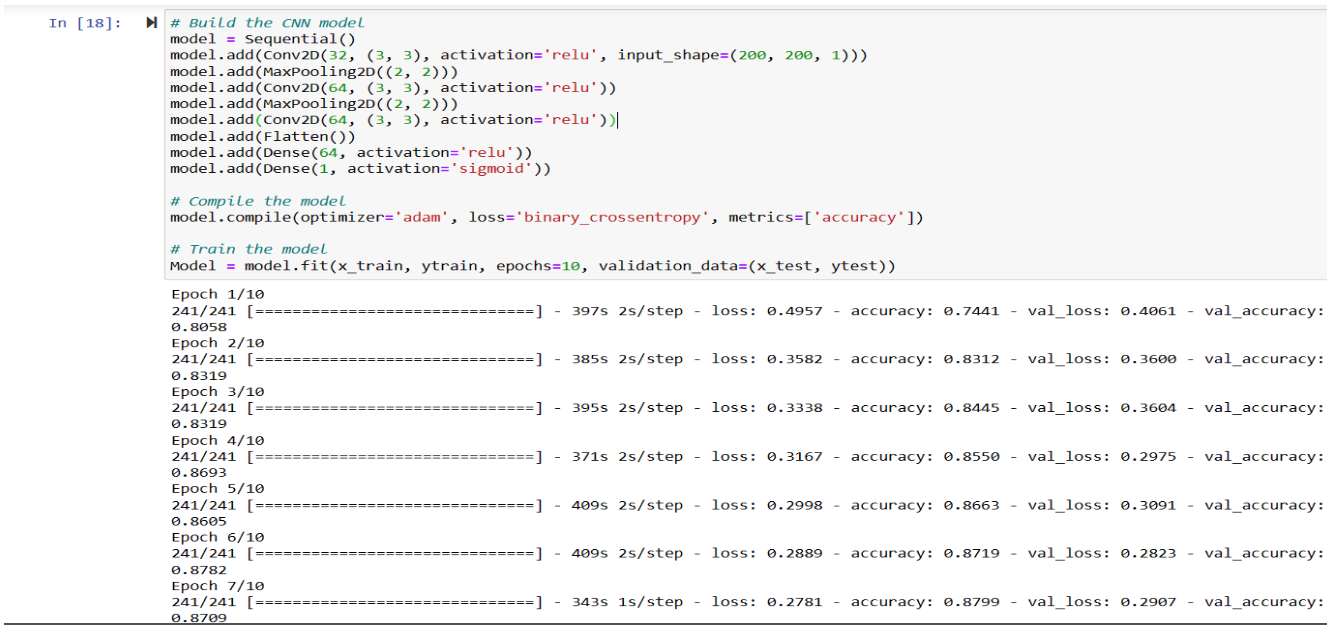
- ## Code explanation:-

The code reshapes the training and testing data for a CNN model by converting them into a four-dimensional array with dimensions (xtrain.shape[0], 200, 200, 1), where xtrain.shape[0] represents the number of samples and 200x200 represents the width and height of each sample with a single channel.



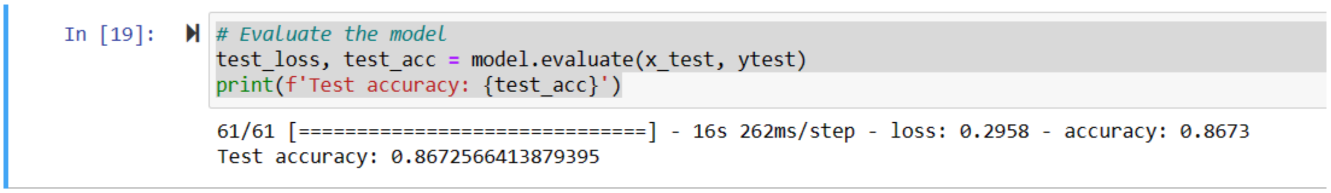
```
In [13]:  # Reshape the input data for CNN
          x_train = xtrain.reshape(xtrain.shape[0], 200, 200, 1)
          x_test = xtest.reshape(xtest.shape[0], 200, 200, 1)
```

The code builds a CNN model using Keras with TensorFlow backend. It consists of three convolutional layers with increasing filter sizes, ReLU activation, and max pooling. The flattened output is then passed through two fully connected layers with ReLU activation, and a final output layer with sigmoid activation. The model is compiled with Adam optimizer and binary cross-entropy loss. It is trained for 10 epochs using the training data
(`x_train` and `ytrain`), and the validation data (`x_test` and `ytest`) is used to evaluate the model's performance.



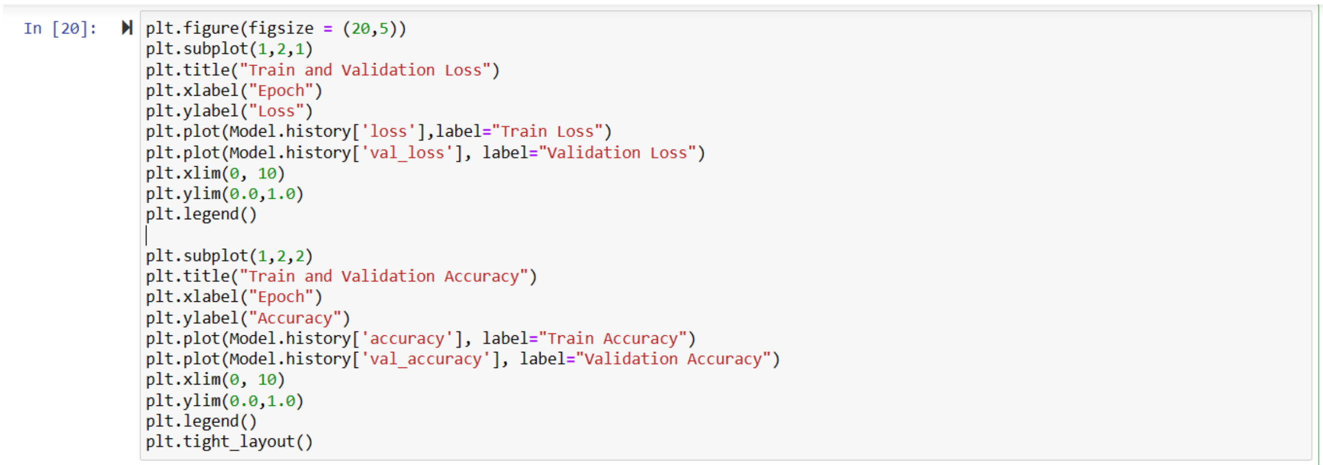
```
In [18]:  # Build the CNN model
          model = Sequential()
          model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(200, 200, 1)))
          model.add(MaxPooling2D((2, 2)))
          model.add(Conv2D(64, (3, 3), activation='relu'))
          model.add(MaxPooling2D((2, 2)))
          model.add(Conv2D(64, (3, 3), activation='relu'))
          model.add(Flatten())
          model.add(Dense(64, activation='relu'))
          model.add(Dense(1, activation='sigmoid'))

          # Compile the model
          model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

          # Train the model
          Model = model.fit(x_train, ytrain, epochs=10, validation_data=(x_test, ytest))

          Epoch 1/10
          241/241 [==============================] - 397s 2s/step - loss: 0.4957 - accuracy: 0.7441 - val_loss: 0.4061 - val_accuracy:
          0.8058
          Epoch 2/10
          241/241 [==============================] - 385s 2s/step - loss: 0.3582 - accuracy: 0.8312 - val_loss: 0.3600 - val_accuracy:
          0.8319
          Epoch 3/10
          241/241 [==============================] - 395s 2s/step - loss: 0.3338 - accuracy: 0.8445 - val_loss: 0.3604 - val_accuracy:
          0.8319
          Epoch 4/10
          241/241 [==============================] - 371s 2s/step - loss: 0.3167 - accuracy: 0.8550 - val_loss: 0.2975 - val_accuracy:
          0.8693
          Epoch 5/10
          241/241 [==============================] - 409s 2s/step - loss: 0.2998 - accuracy: 0.8663 - val_loss: 0.3091 - val_accuracy:
          0.8605
          Epoch 6/10
          241/241 [==============================] - 409s 2s/step - loss: 0.2889 - accuracy: 0.8719 - val_loss: 0.2823 - val_accuracy:
          0.8782
          Epoch 7/10
          241/241 [==============================] - 343s 1s/step - loss: 0.2781 - accuracy: 0.8799 - val_loss: 0.2907 - val_accuracy:
          0.8709
```

The code evaluates the trained model using the testing data (`x_test` and `ytest`). It computes the test loss and accuracy of the model on the testing dataset. The test accuracy is then printed to the console using f-string formatting.

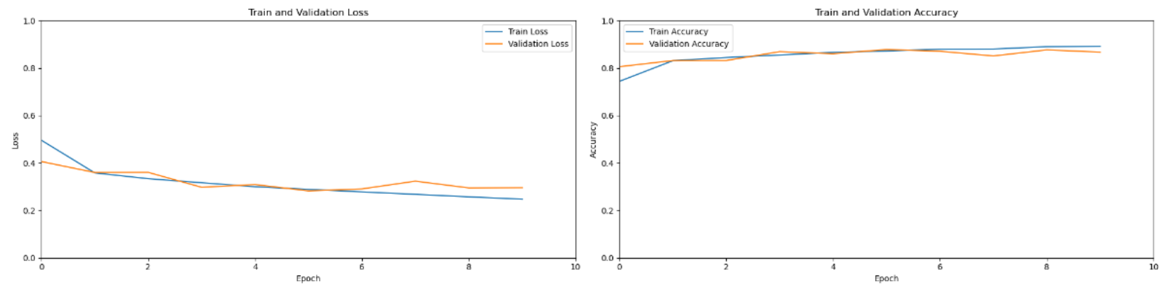


```
In [19]:  # Evaluate the model
          test_loss, test_acc = model.evaluate(x_test, ytest)
          print(f'Test accuracy: {test_acc}')

          61/61 [==============================] - 16s 262ms/step - loss: 0.2958 - accuracy: 0.8673
          Test accuracy: 0.8672566413879395
```

The code creates a figure with two subplots using matplotlib. The first subplot displays the training and validation loss over the epochs, while the second subplot shows the training and validation accuracy. The plots are limited to show epochs 0 to 10 and the y-axis range is set between 0.0 and 1.0. Legends are added to differentiate the lines. The `plt.tight_layout()` command is used to improve the spacing between subplots.

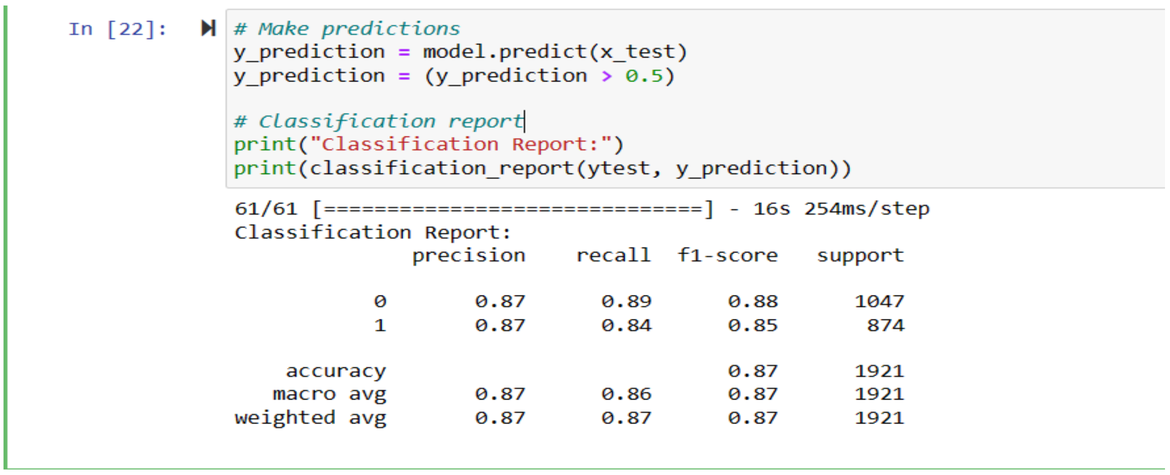


```
In [20]:  plt.figure(figsize = (20,5))
          plt.subplot(1,2,1)
          plt.title("Train and Validation Loss")
          plt.xlabel("Epoch")
          plt.ylabel("Loss")
          plt.plot(Model.history['loss'],label="Train Loss")
          plt.plot(Model.history['val_loss'], label="Validation Loss")
          plt.xlim(0, 10)
          plt.ylim(0.0,1.0)
          plt.legend()

          plt.subplot(1,2,2)
          plt.title("Train and Validation Accuracy")
          plt.xlabel("Epoch")
          plt.ylabel("Accuracy")
          plt.plot(Model.history['accuracy'], label="Train Accuracy")
          plt.plot(Model.history['val_accuracy'], label="Validation Accuracy")
          plt.xlim(0, 10)
          plt.ylim(0.0,1.0)
          plt.legend()
          plt.tight_layout()
```



The model makes predictions on the testing data using the predict method. The predicted probabilities are thresholded at 0.5 to obtain the predicted classes (y_pred).

The classification report (classification_report) provides metrics such as precision, recall, F1-score, and support for each class. Precision represents the proportion of correctly predicted positive samples out of all samples predicted as positive, while recall represents the proportion of correctly predicted positive samples out of all actual positive samples. The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both measures.



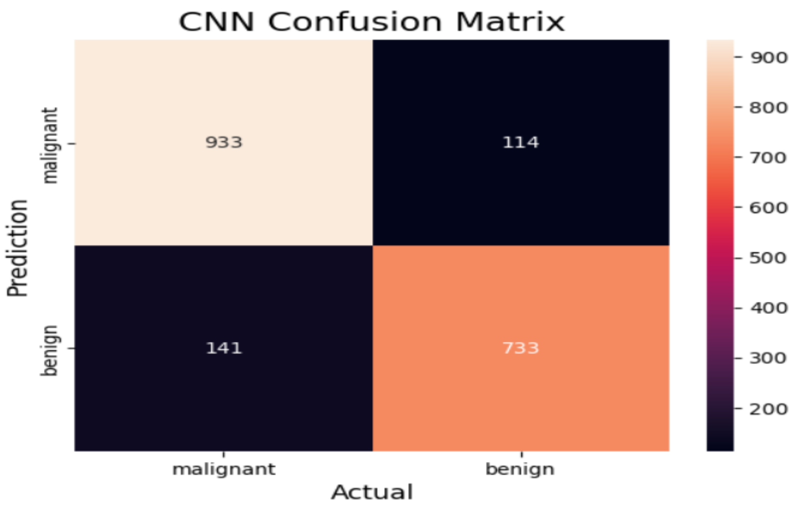
```
In [22]:  # Make predictions
          y_prediction = model.predict(x_test)
          y_prediction = (y_prediction > 0.5)

          # Classification report
          print("Classification Report:")
          print(classification_report(ytest, y_prediction))

61/61 [==============================] - 16s 254ms/step
Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.89      0.88      1047
           1       0.87      0.84      0.85       874

    accuracy                           0.87      1921
   macro avg       0.87      0.86      0.87      1921
weighted avg       0.87      0.87      0.87      1921
```

The confusion matrix (confusion_matrix) provides a tabular representation of the model's predictions compared to the actual labels. It shows the count of true positives, true negatives, false positives, and false negatives.

```
In [23]:  # Confusion matrix
          print("Confusion Matrix:")
          print(confusion_matrix(ytest, y_prediction))

          Confusion Matrix:
          [[933 114]
           [141 733]]
```

The provided code segment utilizes the seaborn library to create a heatmap representation of the confusion matrix for a classification model. The confusion matrix is constructed by comparing the actual labels (ytest) with the predicted labels (y_prediction). The heatmap includes annotations, displaying the counts in each cell. The x-axis and y-axis labels correspond to the classes ('malignant' and 'benign'). The resulting plot offers a visual depiction of the model's performance in terms of correctly and incorrectly classified instances.

```
In [24]:  #Plot the confusion matrix.
          sns.heatmap(confusion_matrix(ytest, y_prediction),annot=True,fmt='g',
          xticklabels=['malignant', 'benign'],
          yticklabels=['malignant', 'benign'])
          plt.ylabel('Prediction',fontsize=13)
          plt.xlabel('Actual',fontsize=13)
          plt.title('CNN Confusion Matrix',fontsize=17)
          plt.show()
```

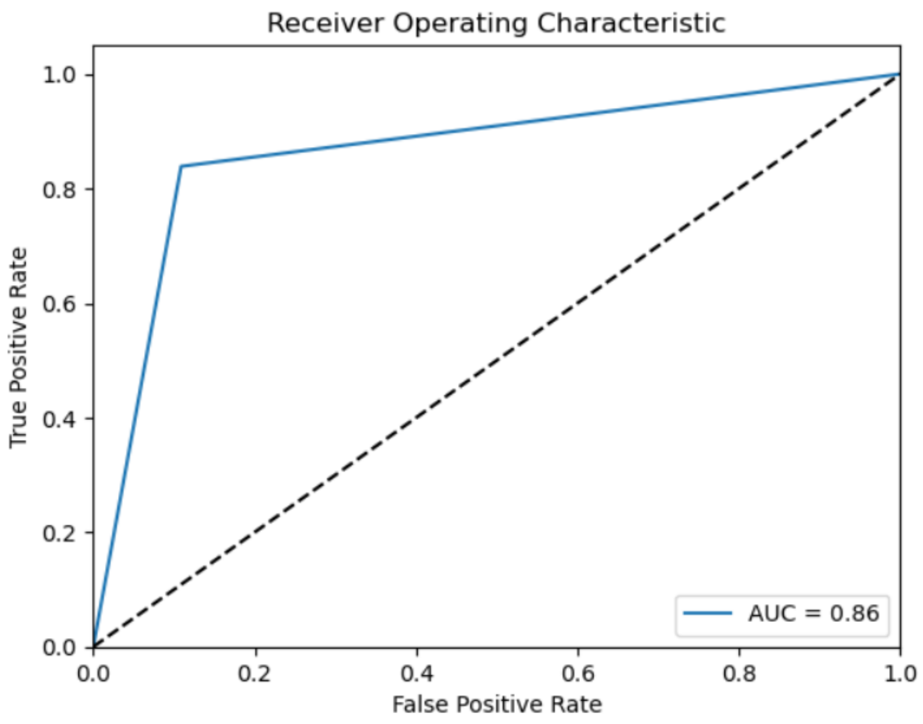


The Receiver Operating Characteristic (ROC) curve is computed using the roc_curve function, which plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) at various threshold values. The area under the ROC curve (AUC) is also calculated using the roc_auc_score function. A higher AUC indicates better performance of the model in distinguishing between the classes.

```
In [25]:  # ROC curve
          fpr, tpr, thresholds = roc_curve(ytest, y_prediction)
          auc = roc_auc_score(ytest, y_prediction)
```

The ROC curve is plotted using Matplotlib. The plot shows the ROC curve along with a diagonal reference line (dashed red line), representing a random classifier. The AUC value is displayed in the legend.

```
In [26]:  # Plot ROC curve
          plt.figure()
          plt.plot(fpr, tpr, label=f'AUC = {auc:.2f}')
          plt.plot([0, 1], [0, 1], 'k--')
          plt.xlim([0.0, 1.0])
          plt.ylim([0.0, 1.05])
          plt.xlabel('False Positive Rate')
          plt.ylabel('True Positive Rate')
          plt.title('Receiver Operating Characteristic')
          plt.legend(loc='lower right')
          plt.show()
```



SUPORT VECTOR MACHINE (SVM)

What is SVM ?

is a popular machine learning algorithm used for both classification and regression tasks. It is particularly effective in solving complex problems with high-dimensional data and has been widely used in various fields such as image recognition, text classification, and bioinformatics.

The main idea behind SVM is to find an optimal hyperplane that separates the data points of different classes with the maximum margin. The hyperplane is defined as the decision boundary that maximizes the distance between the closest data points of different classes, known as support vectors. By maximizing the margin, SVM aims to achieve better generalization and robustness in classifying new, unseen data.

why do we apply SVM on a skin cancer image data ?

1. Non-linearity in image data: Skin cancer image data often exhibits complex patterns and non-linear relationships between the image features and the corresponding labels (benign or malignant). SVM's ability to handle non-linear data through kernel functions makes it suitable for capturing intricate patterns in skin cancer images.

2. High-dimensional feature space: Skin cancer images typically contain a large number of pixels, resulting in high-dimensional feature spaces. SVM's effectiveness in high-dimensional spaces allows it to handle the rich and detailed information present in these images.

3. Robustness against overfitting: SVM's margin-based approach helps prevent overfitting by finding a decision boundary that maximizes the margin between different classes. This robustness is crucial when working with skin cancer image data, as the model needs to generalize well to unseen images.

4. Support for binary classification: SVM is a binary classification algorithm, which aligns with the task of classifying skin cancer images into two categories: benign and malignant. By choosing appropriate training data and labels, SVM can accurately distinguish between healthy and cancerous skin lesions.

5. Interpretability: SVM provides insights into the decision-making process by identifying support vectors, which are the data points closest to the decision boundary. These support vectors can help dermatologists and researchers understand the specific features or characteristics that contribute to the classification of skin cancer.

6. Well-established performance: SVM has been extensively studied and applied in various domains, including medical image analysis. Its performance has been demonstrated in skin cancer classification tasks, achieving high accuracy and sensitivity rates.

Skin cancer is a prevalent form of cancer that can be potentially life-threatening if not detected and treated early. Machine learning techniques, such as Support Vector Machines (SVM), can aid in the classification of skin cancer based on variousfeatures. This report presents the results of an SVM model applied to a skin cancerdataset and evaluates its performance using accuracy and F1-score metrics.

- ## DATA SCALING :

Before training the SVM model, the features were standardized using the StandardScaler() function. This standardization process ensures that all features have a mean of 0 and a standard deviation of 1, which is crucial for SVMs. The training set was scaled using the fit_transform() method of the scaler, while the test set was transformed using the transform() method.

- ## SVM Model Training :

The SVM model was initialized with a linear kernel and a regularization parameter (C) value of 1.0. The model was then trained using the training set. The SVM algorithm learns to classify skin cancer cases based on the provided features.

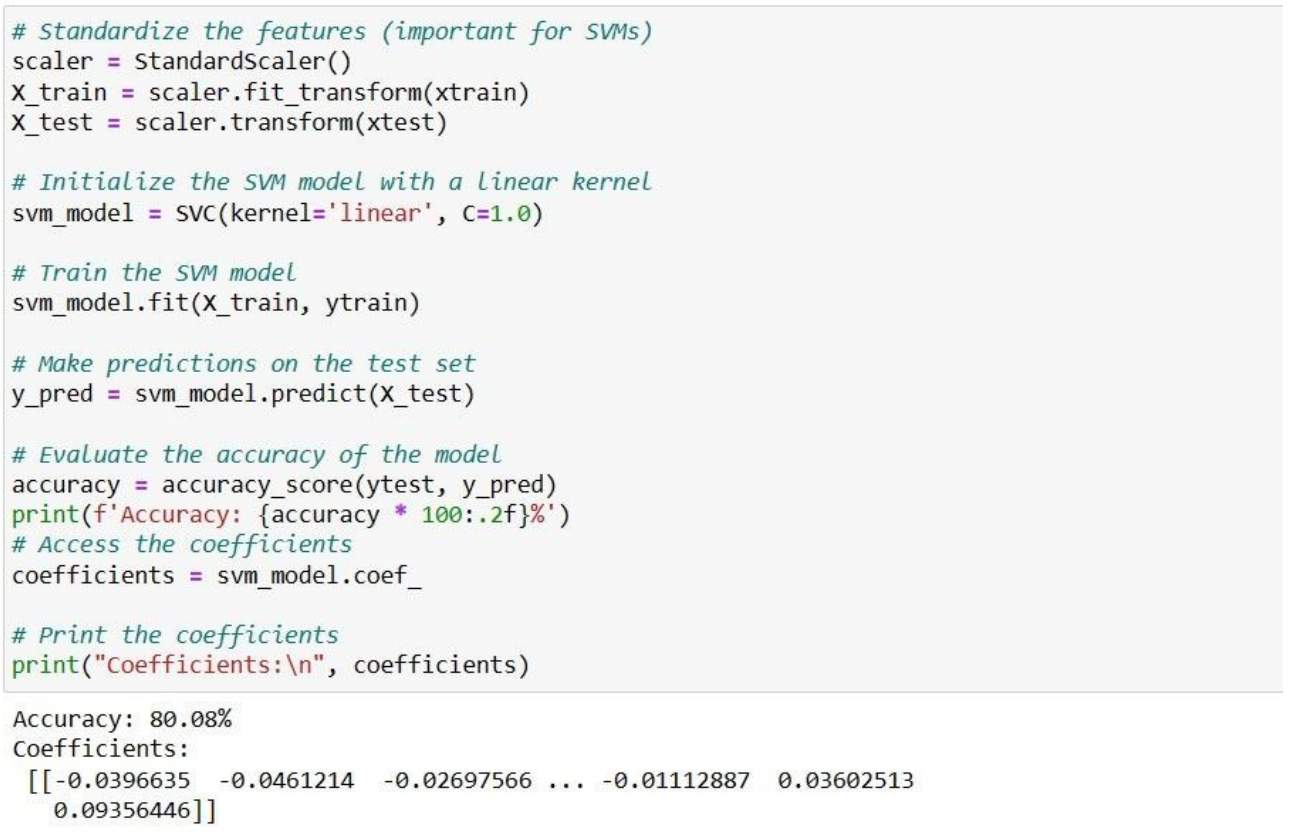
- ## Model Evaluation

Once the model was trained, predictions were made on the test set using the trained SVM model. The accuracy of the model was computed by comparing the predicted labels (y_pred) with the true labels (ytest). The accuracy of the model was determined to be 80.08%, indicating that the model correctly predicted the skin cancer cases in the test set with an 80.08% accuracy rate.

- ## Coefficients Analysis

The Coefficients represent the importance of each feature in the SVM model. Negative Coefficients indicate that a decrease in that feature's value leads to a higher likelihood of skin cancer, while positive Coefficients indicate the opposite. The obtained Coefficients were as follows:

```
[-0.0396635 -0.0461214 -0.02697566 ... -0.01112887 0.03602513 0.09356446]
```



```python
# Standardize the features (important for SVMs)
scaler = StandardScaler()
X_train = scaler.fit_transform(xtrain)
X_test = scaler.transform(xtest)

# Initialize the SVM model with a linear kernel
svm_model = SVC(kernel='linear', C=1.0)

# Train the SVM model
svm_model.fit(X_train, ytrain)

# Make predictions on the test set
y_pred = svm_model.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(ytest, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
# Access the coefficients
coefficients = svm_model.coef_

# Print the coefficients
print("Coefficients:\n", coefficients)
```

```
Accuracy: 80.08%
Coefficients:
 [[-0.0396635  -0.0461214  -0.02697566 ... -0.01112887  0.03602513
   0.09356446]]
```

Precision, Recall, F1-Score, and Support

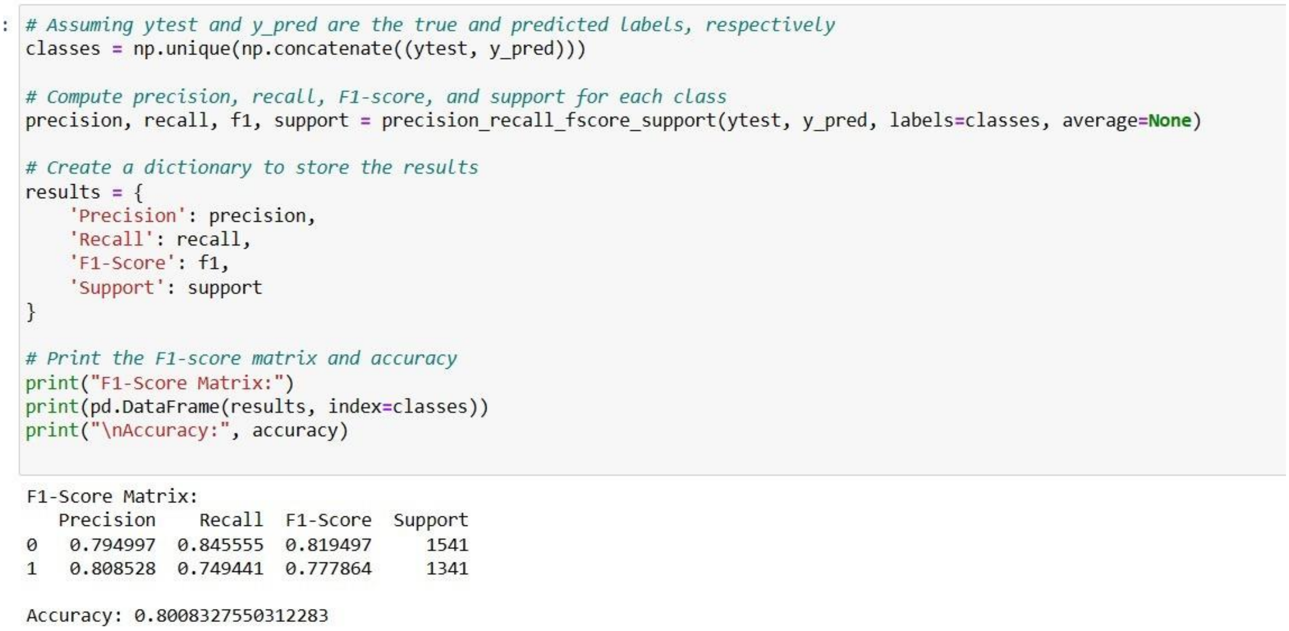
To further evaluate the performance of the SVM model, precision, recall, F1-score, and support metrics were computed for each class (0 and 1) using the precision_recall_fscore_support() function. The precision metric measures the proportion of correctly predicted positive cases out of all predicted positive cases. Recall, also known as sensitivity or true positive rate, measures the proportion of correctly predicted positive cases out of all actual positive cases. The F1-score is the harmonic mean of precision and recall, providing a balanced measure of the model's performance. Support represents the number of instances in each class. The computed metrics for each class were as follows:

Class 0:

- Precision: 0.794997
- Recall: 0.845555
- F1-Score: 0.819497
- Support: 1541

CLASS 1 :

- Precision: 0.808528
- Recall: 0.749441
- F1-Score: 0.777864
- Support: 1341



```
# Assuming ytest and y_pred are the true and predicted labels, respectively
classes = np.unique(np.concatenate((ytest, y_pred)))

# Compute precision, recall, F1-score, and support for each class
precision, recall, f1, support = precision_recall_fscore_support(ytest, y_pred, labels=classes, average=None)

# Create a dictionary to store the results
results = {
    'Precision': precision,
    'Recall': recall,
    'F1-Score': f1,
    'Support': support
}

# Print the F1-score matrix and accuracy
print("F1-Score Matrix:")
print(pd.DataFrame(results, index=classes))
print("\nAccuracy:", accuracy)
```

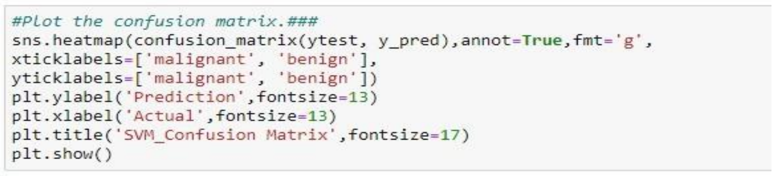
```
F1-Score Matrix:
   Precision    Recall  F1-Score  Support
0   0.794997  0.845555  0.819497     1541
1   0.808528  0.749441  0.777864     1341

Accuracy: 0.8008327550312283
```

- confusion matrix

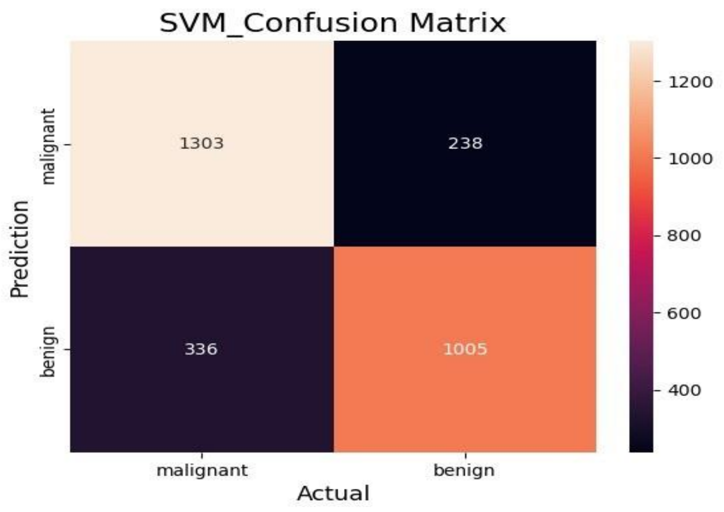
confusion matrix is a table that summarizes the performance of a classification model

The diagonal cells of the matrix show the number of tumors that were correctly classified. In this case, 1200 malignant tumors were correctly classified as malignant, and 1005 benign tumors were correctly classified as benign.

The off-diagonal cells of the matrix show the number of tumors that were misclassified. In this case, 238 malignant tumors were incorrectly classified as benign, and 336 benign tumors were incorrectly classified as malignant



```
#Plot the confusion matrix.###
sns.heatmap(confusion_matrix(ytest, y_pred),annot=True,fmt='g',
xticklabels=['malignant', 'benign'],
yticklabels=['malignant', 'benign'])
plt.ylabel('Prediction',fontsize=13)
plt.xlabel('Actual',fontsize=13)
plt.title('SVM_Confusion Matrix',fontsize=17)
plt.show()
```

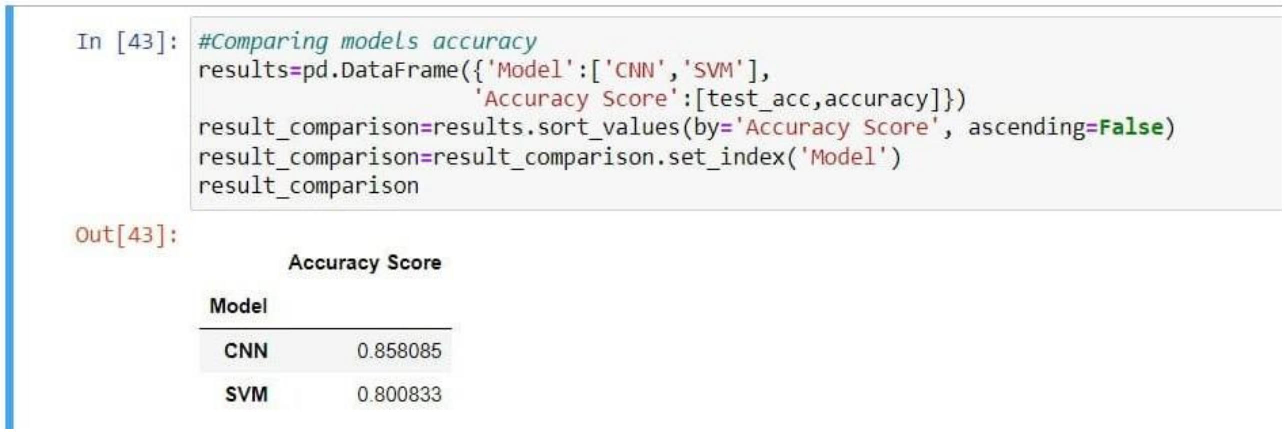


- ## Conclusion

  In conclusion, the SVM model achieved an accuracy of 80.08% in classifying skin cancer cases. The precision, recall, and F1-score metrics were also calculated for each class, providing insights into the model's performance for specific categories. These results demonstrate the potential of using machine learning techniques, such as SVM, for skin cancer classification, allowing for early detection and timely intervention.

MODEL COMPARISON

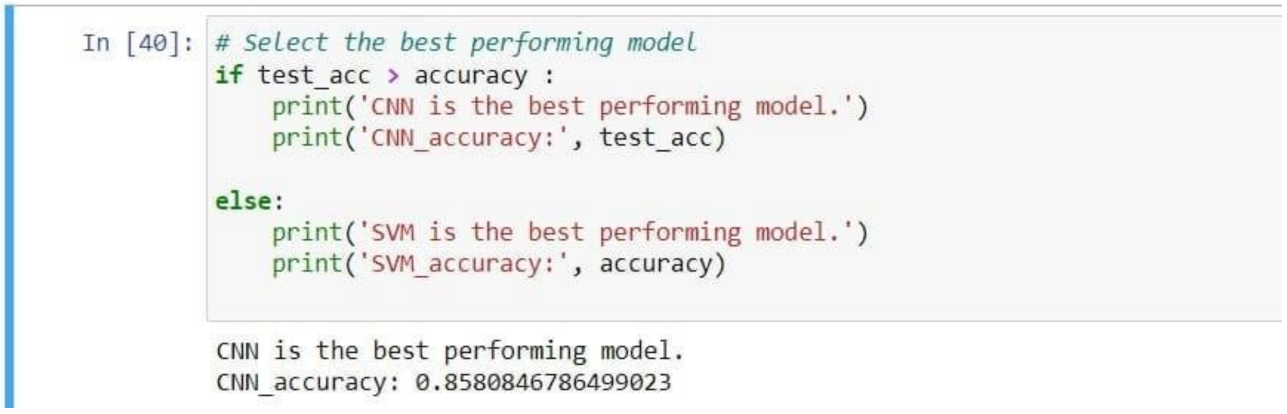
This code is an effective way to compare the performance of two models based on their accuracy scores, allowing users to quickly identify which model achieved higher accuracy in the task at hand.



```
In [43]: #Comparing models accuracy
         results=pd.DataFrame({'Model':['CNN','SVM'],
                               'Accuracy Score':[test_acc,accuracy]})
         result_comparison=results.sort_values(by='Accuracy Score', ascending=False)
         result_comparison=result_comparison.set_index('Model')
         result_comparison

Out[43]:
```

Model	Accuracy Score
CNN	0.858085
SVM	0.800833

This code segment selects the best performing model between a CNN (Convolutional Neural Network) and an SVM (Support Vector Machine) based on their accuracy scores.



```
In [40]: # Select the best performing model
         if test_acc > accuracy :
             print('CNN is the best performing model.')
             print('CNN_accuracy:', test_acc)

         else:
             print('SVM is the best performing model.')
             print('SVM_accuracy:', accuracy)

         CNN is the best performing model.
         CNN_accuracy: 0.8580846786499023
```

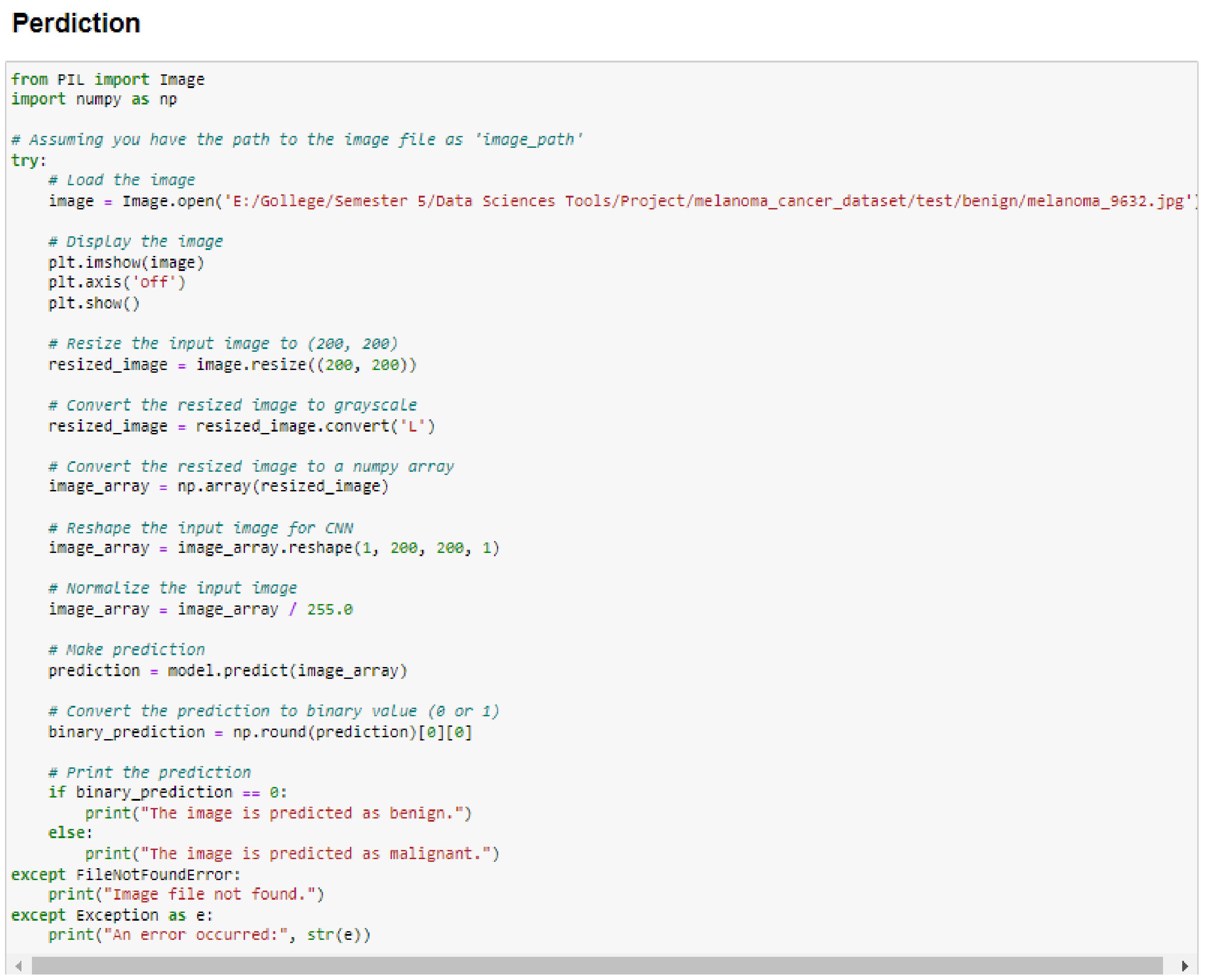
Prediction:

Because the CNN model is the best we used it to predict the image if begin or malignant

First   Loading and displaying the image

Second Preprocessing the image

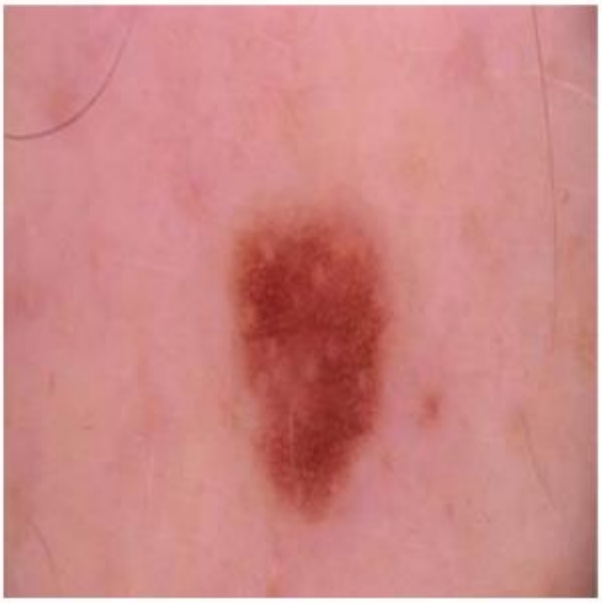
Third   Making the prediction



Perdiction

```
from PIL import Image
import numpy as np

# Assuming you have the path to the image file as 'image_path'
try:
    # Load the image
    image = Image.open('E:/Gollege/Semester 5/Data Sciences Tools/Project/melanoma_cancer_dataset/test/benign/melanoma_9632.jpg')

    # Display the image
    plt.imshow(image)
    plt.axis('off')
    plt.show()

    # Resize the input image to (200, 200)
    resized_image = image.resize((200, 200))

    # Convert the resized image to grayscale
    resized_image = resized_image.convert('L')

    # Convert the resized image to a numpy array
    image_array = np.array(resized_image)

    # Reshape the input image for CNN
    image_array = image_array.reshape(1, 200, 200, 1)

    # Normalize the input image
    image_array = image_array / 255.0

    # Make prediction
    prediction = model.predict(image_array)

    # Convert the prediction to binary value (0 or 1)
    binary_prediction = np.round(prediction)[0][0]

    # Print the prediction
    if binary_prediction == 0:
        print("The image is predicted as benign.")
    else:
        print("The image is predicted as malignant.")
except FileNotFoundError:
    print("Image file not found.")
except Exception as e:
    print("An error occurred:", str(e))
```

The Output:



```
1/1 [==============================] - 1s 784ms/step
The image is predicted as benign.
```