Data Pipeline 2

Can we tell if Sana has been making healthy food choices?

Minerva University

CS156: Machine Learning

Prof. Watson

Dec 2, 2024

Table of Contents

Colab Link: CS156-pipeline2.ipynb

Introduction

The question remained unanswered in our previous assignment; about the healthy/unhealthy food choices Sana has been making, while we attempted to classify images using Logistic Regression and SVM, the limited data hindered the models' ability to provide meaningful insights.
This assignment dives deeper to resolve that uncertainty. By expanding the dataset and testing these models again, we aim to provide clearer insights. Additionally, we'll introduce Transfer Learning, a powerful technique that utilizes pre-trained neural networks to tackle data limitations and improve classification accuracy.
Through this two-pronged approach, we aim to not only classify healthy and unhealthy foods more effectively but also compare the performance of traditional models with advanced neural networks to uncover the most reliable solution. All the code can be found in the appendix

Part 1: Enhancing Previous Models with Expanded Data

To expand the dataset, I began by sourcing additional images from publicly available datasets, such as the Food-11 dataset on Kaggle. This dataset offered a variety of food categories, which I mapped to the two main classes: "Healthy" and "Unhealthy." For instance, categories like desserts and noodles were classified as "Unhealthy," while categories like fruits and vegetables were categorized as "Healthy." This initial step significantly expanded the dataset from just 30 images per class to 1,244 images per class, providing a broader representation of food items.

However, despite this improvement, the dataset still lacked sufficient diversity to train robust models effectively. A larger dataset not only enhances the model's ability to generalize but also reduces the risk of overfitting, where the model performs well on the training data but poorly on unseen data. To address this challenge, I turned to **data augmentation**, a technique used to artificially increase the size and diversity of the dataset by applying transformations to existing images.

Data augmentation involved applying transformations such as rotation, flipping, cropping, scaling, and color adjustments to the original images. These techniques simulate variations that the model might encounter in real-world scenarios, such as changes in angle, lighting, or orientation. For instance:

- **Rotation**: Images were rotated by a few degrees to mimic different viewpoints.
- **Flipping**: Horizontal or vertical flips introduced mirrored versions of the images.
- **Cropping**: Random cropping created zoomed-in perspectives.
- **Scaling and Resizing**: Slight changes in size introduced further diversity in visual features.
- **Color Adjustments**: Brightness, contrast, and saturation were altered to simulate varying lighting conditions.

These augmentations effectively expanded the dataset into tens of thousands of unique samples, ensuring the models had sufficient and diverse data to learn from. This enhanced

dataset not only improved the reliability of traditional models like Logistic Regression and SVM but also set the stage for experimenting with advanced methods such as Transfer Learning.

By combining dataset expansion through external sources and augmentation, the assignment significantly elevated the quality and quantity of data, enabling a more robust evaluation of the models.

After augmenting the dataset to improve its diversity, we preprocessed the data to ensure it was compatible with both models we aimed to test: Logistic Regression and Support Vector Machines (SVM). A more detailed explanation of these models, their mathematical foundations, and related concepts is available in the Appendix (- Assignment 1 for a foundational discussion).

Logistic regression

Logistic Regression is one of the simplest classification models, designed to predict a binary outcome by modeling the relationship between the input features and the probability of

belonging to a particular class. Mathematically, the model uses the sigmoid function to map the weighted sum of the inputs to a probability:

$$P(y = 1|x) = \sigma(W \cdot x + b) = \frac{1}{1 + e^{-(W \cdot x + b)}}$$

Where:

- W: Weight vector,
- x: Input features,
- b: Bias term,
- σ: Sigmoid activation function.

While Logistic Regression is a very simple model, it can sometimes perform well on datasets with linearly separable classes. However, given the complexity of our data, we do not expect Logistic Regression to perform particularly well in this scenario. Nonetheless, testing it serves as a useful baseline for comparison with more sophisticated models.

SVM

SVMs are powerful classification models that aim to find the optimal hyperplane that maximizes the margin between classes in the feature space. The decision boundary is determined by the support vectors, which are the data points closest to the hyperplane. For non-linear problems, SVMs use kernel functions to project the data into higher-dimensional spaces where it becomes linearly separable.

The objective of SVM is to solve the following optimization problem:

$$\min \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1, \forall i$$

Where:

- w: Weight vector,
- b: Bias term,
- x_i, y_i: Input features and labels.

We have tested with several kernels like linear, polynomial, rbf to see how they influenced the model’s performance.

A kernel function transforms the input data into a higher-dimensional space, where the classes may become linearly separable. This transformation is done implicitly, meaning the SVM does not compute the transformation explicitly, saving computational resources. Instead, it relies on the **kernel trick**, which calculates the dot product of the transformed features directly.

In essence, kernels determine how the SVM "views" the data:

- A **linear kernel** draws straight boundaries in the original space.
- A **polynomial kernel** introduces flexibility by allowing curved boundaries.
- An **RBF kernel** enables highly adaptive boundaries by mapping the data to a much higher-dimensional space.

SVM is a versatile model that often performs well on moderately complex data. By experimenting with different kernels, we can analyze which approach (linear or non-linear) aligns best with the data structure and provides the highest accuracy.

The results and comparative performance of Logistic Regression and SVM (with different kernels) are discussed in the Part 3- **Model Evaluation** section. The analysis focuses on key metrics such as accuracy and insights drawn from the confusion matrices, which offer a deeper understanding of each model's performance nuances.

Part 2: Experimenting with a new Model

For the next part of the assignment, I focused on 2 more complex model - CNN which is a neural network and Transfer Learning involves reusing a pre-trained neural network.

For image classification tasks, neural networks are particularly powerful because they excel at capturing intricate patterns and features in high-dimensional data, such as the pixel values of an image. They can model non-linear relationships, enabling them to classify data that simpler models like Logistic Regression struggle with, especially when data classes are overlapping or complex.

Unlike traditional models that require handcrafted features, neural networks automatically learn the features most relevant for classification during training. This makes them highly flexible and adaptable across different domains. Moreover, they perform well with large datasets, as their ability to generalize improves with more data. Given the expanded dataset in this assignment, neural networks are an ideal choice for tackling the classification challenge. Their ability to learn hierarchical representations of data—from simple patterns like edges and textures to complex features like shapes and objects—makes them particularly suited for image-based tasks.

CNN

Convolutional Neural Networks (CNNs) are a specialized type of neural network designed to process structured grid-like data, such as images. The CNN learns features like textures, colors, and patterns in food images, distinguishing between healthy items (e.g., fruits, vegetables) and unhealthy ones (e.g., desserts, processed foods).

CNNs are particularly efficient because they use **convolutions**, a mathematical operation that preserves spatial relationships while reducing the number of parameters compared to fully connected layers.

Convolutional Layer

The convolutional layer is the core building block of CNNs. It applies **filters** (also called kernels) to the input image to extract feature maps. Each filter detects specific patterns, such as edges or textures.

$$F(x,y) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K(i,j) \cdot I(x+i, y+j)$$

Where:

- $I(x+i, y+j)$: Input pixel at position $(x+i, y+j)$,
- $K(i,j)$: Kernel weight at position (i,j),
- $F(x,y)$: Value of the feature map at position (x,y).

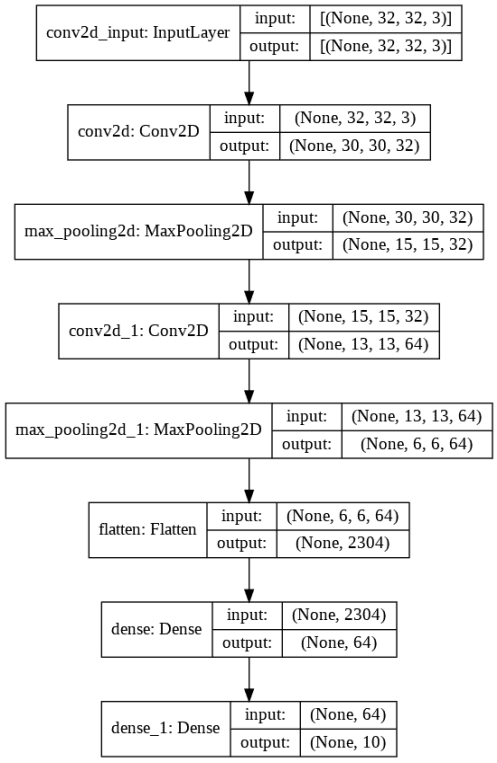
In our model:

- The kernel size is 3×3, meaning each filter covers a 3×3 region of the input image.
- 32 filters are applied in this layer, generating 32 feature maps.
- The input shape specifies the dimensions of the input image: 160×160×3(width, height, RGB channels).

ReLU activation function
introduces non-linearity, enabling the network to learn complex patterns.Is applied after each convolution operation. It is defined as:

$$f(x) = \max(0, x)$$ This ensures that only positive values are passed forward, introducing non-linearity to the model. ReLU allows the CNN to learn complex patterns while reducing the risk of vanishing gradients during training.

Pooling Layers:



Pooling layers reduce the spatial dimensions of the feature maps, helping to decrease computational complexity and mitigate overfitting. They also preserve the most prominent features.

$$F_{\text{pooled}} = \max(F_{ij}), \quad i, j \in \text{pooling window}$$

For my model, pooling layers are applied after each convolutional layer to downsample the feature maps, making the model computationally efficient.

After the convolutional and pooling layers, the feature maps are flattened into a 1D vector, which serves as input to the fully connected layers. - Converts the multi-dimensional feature map into a single vector.

The first Dense layer has 128 neurons with ReLU activation. It combines the extracted features into a high-level representation.

The final Dense layer has 1 neuron with a sigmoid activation for binary classification:

$$P(y = 1|x) = \sigma(W \cdot z + b)$$

Where:

- W: Weights,
- z: Flattened input vector,
- b: Bias.

When we compile the model we have:

Optimizer: Adam

Adam combines the benefits of momentum and adaptive learning rates for efficient optimization. The parameter update rule is:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}$$

Where:

- mt: Exponentially weighted average of gradients,
- vt: Exponentially weighted average of squared gradients,
- η: Learning rate

Loss Function: Binary Cross-Entropy

$$L = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

For binary classification, the loss is defined as:

Where:

- L : Loss function we want to calculate
- yi: True label (0 or 1),

- y^i: Predicted probability.
- N : Number of target samples in the dataset

This loss penalizes incorrect predictions more heavily, encouraging the model to output probabilities close to the true labels.

Transfer Learning

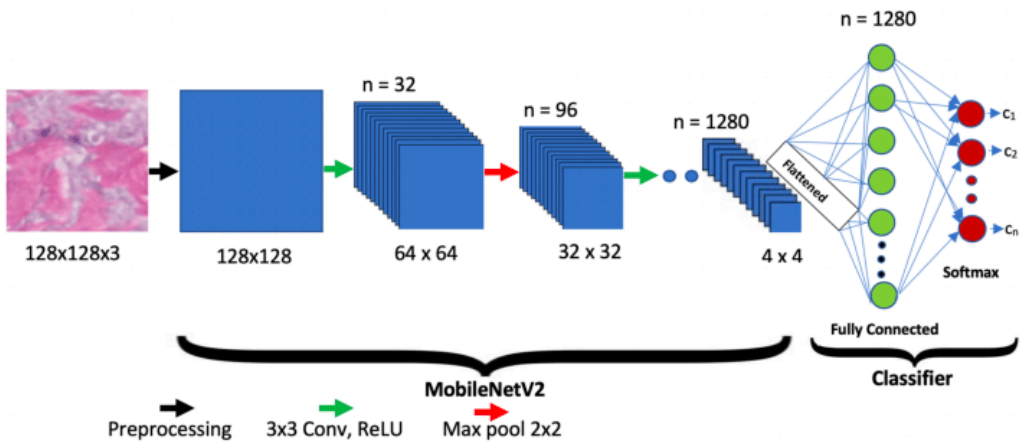
Further, we will explore Transfer Learning, to put it in simpler terms in transfer learning we simply take a model trained on one task is adapted for a different but related task.
It leverages the knowledge learned from large, generic datasets to improve performance on specific, smaller datasets.
There are a lot of pre trained pre-trained models—neural networks that have already been trained on massive datasets like ImageNet. These models serve as feature extractors, as their initial layers capture general patterns such as edges, textures, and shapes, which are transferable across many image-related tasks.

But why should we use Transfer Learning over CNN? Is it more accurate? Well for image classification, Transfer Learning is particularly valuable when the dataset is small or lacks diversity. Instead of training a CNN from scratch, which is computationally expensive and data-intensive, Transfer Learning allows us to fine-tune a pre-trained network for our specific problem. This saves time, reduces resource requirements, and typically results in higher accuracy due to the rich feature representations learned by the pre-trained model.

So for our image classification we will use pre-trained model like MobileNetV2.
MobileNetV2, a popular lightweight neural network, is particularly efficient due to its computational efficiency and accuracy trade-off.

First we did some preprocessing of the data like Resize → Normalize → Augment →Feed into the model.

Now, Lets get into the nitty gritty details and maths behind MobileNetV2 and its role in transfer learning



MobileNetV2 uses depthwise separable convolutions and inverted residuals with linear bottlenecks to efficiently extract features from images.

Feature Extraction

For the Base Model as we can see in the code is used for the feature extraction
Depthwise Separable Convolution: Reduces computation by splitting the convolution operation into two parts:

1. Depthwise Convolution: Applies a single filter per channel.
2. Pointwise Convolution: Combines the channels using 1× 1 convolutions.

$$\text{Standard Convolution: } H \times W \times C_{in} \times C_{out} \times K^2$$

$$\text{Depthwise Separable: } H \times W \times C_{in} \times K^2 + H \times W \times C_{in} \times C_{out}$$

This drastically reduces computation, especially for high-dimensional inputs.

Inverted Residuals: Instead of compressing and expanding feature maps like in traditional ResNets, MobileNetV2 first expands the feature space and then compresses it back.

- Expand: The input is first expanded from input channels to a higher-dimensional feature

$$\text{Expand: } z = \phi(W_{expand} \cdot x)$$

- Depthwise Convolution: Spatial features are extracted using depthwise convolution.

$$\text{Depthwise Conv: } \tilde{z} = D(z)$$

- Linear Bottleneck: The high-dimensional features are projected back to a lower-dimensional space using a linear transformation without non-linear activations.

$$\text{Project: } y = W_{project} \cdot \tilde{z}$$

The key insight behind the linear bottleneck is that non-linearity is avoided in the bottleneck layer to prevent information loss during dimensionality reduction.

And then for the global Global Average Pooling: The feature map output (5×5×1280) is averaged across its spatial dimensions to produce a single vector:

$$z_{avg} = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} z_{i,j}$$

This reduces the feature map to 1x 1280 simplifying the input for the classification head.

Then we have the Classification Head. A dense layer with a sigmoid activation is added for binary classification:

$$P(y = 1|x) = \sigma(W_{dense} \cdot z_{avg} + b)$$

And the sigmoid function is given by $\sigma(z) = \frac{1}{1 + e^{-z}}$ which ensures the output is between 0 and 1, representing the probability of the image being healthy or unhealthy

Fine Tuning

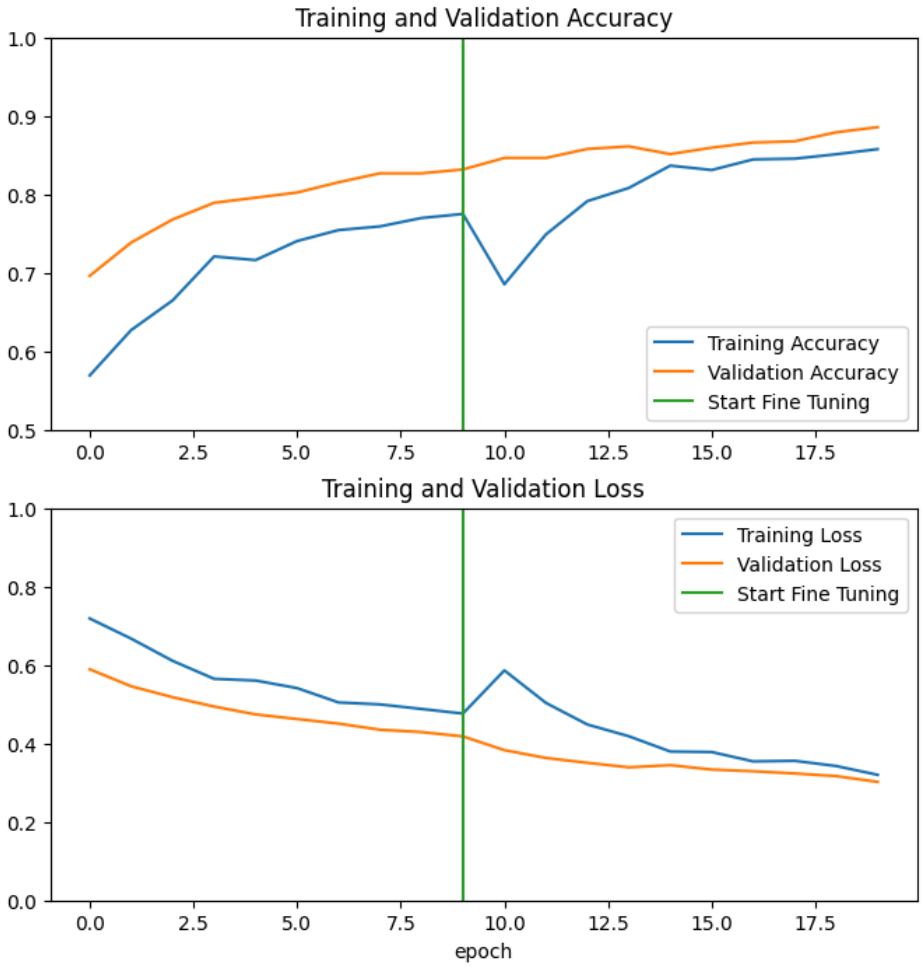
After the initial training of the classification head, the deeper layers of the MobileNetV2 base are unfrozen for fine-tuning. This allows the model to adjust its pre-trained features to better align with the specific task (healthy vs. unhealthy classification).Fine-tuning is typically restricted to a subset of deeper layers (closer to the classification head), as these layers capture more task-specific features. The earlier layers, which capture general features like edges, remain frozen to avoid overfitting.
Unfreezing the Base Model: `base_model.trainable = True` allows the base layers to be updated during backpropagation. Also a subset of layers near the classification head is trained (`fine_tune_at = 100` ensures earlier layers remain frozen).

During this stage, backpropagation updates both:
The classification head (as before).Unfrozen layers of the base model

$$\theta_{\text{base}}^{(t+1)} = \theta_{\text{base}}^{(t)} - \eta_{\text{fine}} \frac{\partial L}{\partial \theta_{\text{base}}}$$

ηfine is a smaller learning rate for the pre-trained layers to avoid large, destabilizing weight updates.

And this allows the deeper pre-trained layers to adapt to the specific task and would Improve performance by learning more task-relevant features while retaining general knowledge from ImageNet.
We can see this from how our model also performed even better after fine tuning.



Part 3: Model Evalution

To facilitate a straightforward comparison among all models, the accuracies for each model have been synthesized below. Accuracy serves as an intuitive metric for identifying which model performs best overall, especially since the dataset is balanced. While accuracy alone cannot fully capture the nuances of performance, it provides a clear starting point for comparison across models.
Additional metrics such as F1 Score, Precision, and Recall to evaluate performance in specific areas of models can be found below in the appendix

Models	**F1 Score**	**Accuracy**
Logistic Regression	0.58	0.56
SVM (kernel = linear)	0.64	0.58
SVM (kernel = rbf)	0.64	0.59
SVM (kernel = poly)	0.64	0.61
CNN	0.70	0.71
Transfer Learning (feature extractor)	0.78	0.83
Transfer Learning (with fine tuning)	*0.86*	*0.88*

The progression of accuracies highlights the critical role of model selection in tackling complex image classification tasks.

- Logistic Regression, as a simple linear model, provides a foundational baseline but is inherently limited in its ability to handle the non-linear patterns and overlapping features present in the dataset. Its inability to process the spatial relationships between pixels makes it unsuitable for image data, as evidenced by its low accuracy of 56%.

- SVM models improve upon this by introducing non-linear kernels, such as the RBF and polynomial kernels, which attempt to map the data into higher-dimensional spaces where separability improves. However, their rigid reliance on fixed feature transformations and lack of hierarchical feature learning keep their performance relatively modest, with accuracies ranging from 58% (linear kernel) to 62% (polynomial kernel). While SVMs capture feature interactions better than Logistic Regression, they still fall short of the capabilities required for the task.

- CNNs bring a transformative leap in performance, achieving an accuracy of 71% by automatically learning hierarchical and spatial features through convolutional layers. These layers extract low-level patterns like edges and textures and combine them into higher-level representations, such as food-specific features that distinguish healthy and unhealthy foods. This ability to leverage spatial relationships and hierarchical structures makes CNNs far superior to traditional models. However, their performance is constrained by the limited size of the dataset and the relatively simple architecture employed, suggesting that deeper or more complex networks could further enhance their accuracy.

- The best results are achieved with Transfer Learning, where the MobileNetV2 model pre-trained on the ImageNet dataset provides a powerful foundation for feature extraction. As a feature extractor, MobileNetV2 leverages its pre-trained knowledge of universal image patterns, resulting in a significant boost in accuracy to 83%. This demonstrates the strength of Transfer Learning in efficiently utilizing large-scale pre-trained features, even when the dataset is relatively small. Fine-tuning the deeper layers of MobileNetV2 further elevates the model's performance to 88% accuracy by adapting its general-purpose features to the specific nuances of the food classification task. This two-stage approach—first leveraging pre-trained features and then fine-tuning for task-specific adaptation—combines the best of both worlds, offering generality and specialization. The fine-tuned Transfer Learning model not only handles the hierarchical and spatial complexities of image data but also demonstrates excellent generalization, making it the most effective and reliable choice for this task. This progression underscores the immense value of pre-trained models and highlights the power of Transfer Learning in solving complex image classification problems with limited data and resources.

Possible Extensions

For my final assignment, I am considering exploring a more creative and challenging approach, such as generating new images based on the current dataset. For example, I could design a system that, when prompted, creates new images of "healthy" or "unhealthy" food. This would allow me to investigate generative models, such as GANs (Generative Adversarial Networks), and delve deeper into the image generation process.

Alternatively, I might experiment with implementing a different model for image classification to compare its performance with the current CNN architecture. This could involve exploring state-of-the-art architectures like Vision Transformers (ViT), providing an opportunity to enhance the accuracy and efficiency of the classification task.

AI Statement:

I used ChatGPT to write code, and comment it. Additionally I have used it to improve my writing and give me feedback on the math and explanation of the different models I have talked about in the assignment.

References

Rajat. (2021, August 13). Transfer learning and the mathematics behind it. *Medium*. https://medium.com/@rajat01221/transfer-learning-and-the-mathematics-behind-it-0988153178aa

Olah, C. (2014, July 8). Conv nets: A modular perspective. *Colah's blog*. http://colah.github.io/posts/2014-07-Conv-Nets-Modular/

Shafkat, I. (2018, June 1). Intuitively understanding convolutions for deep learning. *Towards Data Science*. https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-68ae58248027

Analytics Vidhya. (2023, December). What is MobileNetV2? *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2023/12/what-is-mobilenetv2/

Appendix

Part 1: Code

Part 2: Assignment 1 Pipeline

Setting Up

```
[92] #importing all necessary libraries
     import matplotlib.pyplot as plt
     import numpy as np
     import os
     import tensorflow as tf
     from tensorflow.keras import layers, models
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report, accuracy_score
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.svm import SVC
     from sklearn.model_selection import GridSearchCV
     from tensorflow.keras.applications import VGG16
```

```
[93] #data from drive
     from google.colab import drive
     drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[94] # Main directory containing 'healthy' and 'unhealthy'
     data_dir = '/content/drive/My Drive/cs156-data'
```

```
[95] #paths in the directory
     train_dir = os.path.join(data_dir, 'train')
     validation_dir = os.path.join(data_dir, 'val')
     test_dir = os.path.join(data_dir, 'test')
```

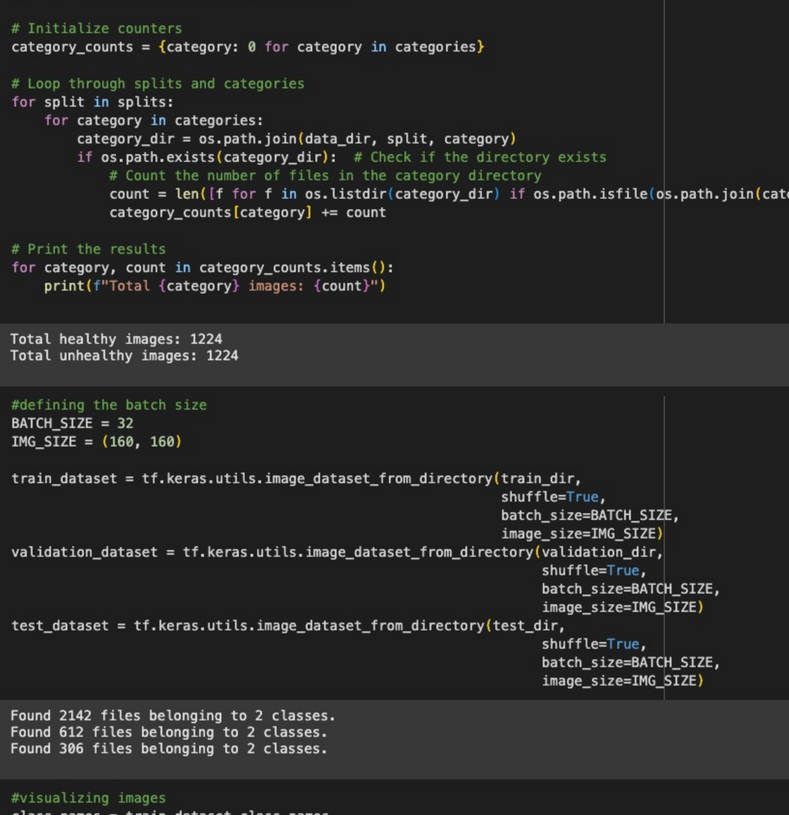
```
[96] import os

     # Define paths to directories
     categories = ["healthy", "unhealthy"]
     splits = ["train", "test", "validation"]

     # Initialize counters
     category_counts = {category: 0 for category in categories}

     # Loop through splits and categories
     for split in splits:
         for category in categories:
             category_dir = os.path.join(data_dir, split, category)
             if os.path.exists(category_dir):  # Check if the directory exists
                 # Count the number of files in the category directory
                 count = len([f for f in os.listdir(category_dir) if os.path.isfile(os.path.join(category_dir, f))])
                 category_counts[category] += count

     # Print the results
     for category, count in category_counts.items():
         print(f"Total {category} images: {count}")
```

```
Total healthy images: 1224
Total unhealthy images: 1224
```

```
[97] #defining the batch size
     BATCH_SIZE = 32
     IMG_SIZE = (160, 160)

     train_dataset = tf.keras.utils.image_dataset_from_directory(train_dir,
                                                                 shuffle=True,
                                                                 batch_size=BATCH_SIZE,
                                                                 image_size=IMG_SIZE)
     validation_dataset = tf.keras.utils.image_dataset_from_directory(validation_dir,
                                                                      shuffle=True,
                                                                      batch_size=BATCH_SIZE,
                                                                      image_size=IMG_SIZE)
     test_dataset = tf.keras.utils.image_dataset_from_directory(test_dir,
                                                                shuffle=True,
                                                                batch_size=BATCH_SIZE,
                                                                image_size=IMG_SIZE)
```

```
Found 2142 files belonging to 2 classes.
Found 612 files belonging to 2 classes.
Found 306 files belonging to 2 classes.
```

```
[7] #visualizing images
    class_names = train_dataset.class_names

    plt.figure(figsize=(10, 10))
    for images, labels in train_dataset.take(1):
      for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

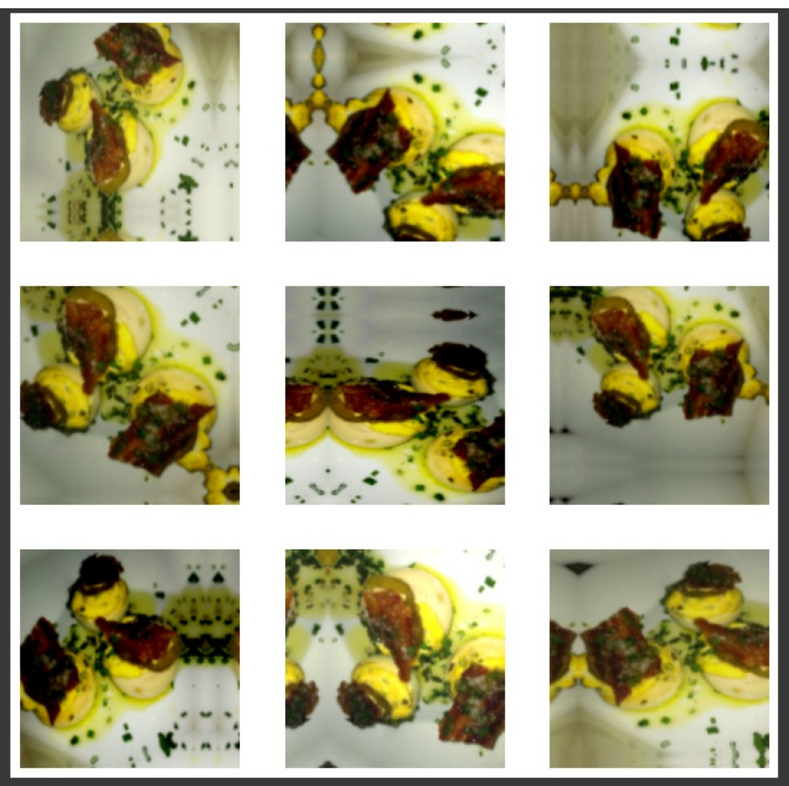


Part 1: Evaluating Previous Models with Data Augmentation

Data Augmentation

```
[98] # Define the data augmentation pipeline
     data_augmentation = tf.keras.Sequential([
         # Flip images horizontally and vertically
         tf.keras.layers.RandomFlip('horizontal_and_vertical'),

         # Rotate images randomly by 20% of 360 degrees
         tf.keras.layers.RandomRotation(0.2),

         # Zoom in and out randomly
         tf.keras.layers.RandomZoom(height_factor=(-0.2, 0.2), width_factor=(-0.2, 0.2)),

         # Translate images randomly in height and width directions
         tf.keras.layers.RandomTranslation(height_factor=0.2, width_factor=0.2),

         # Adjust brightness randomly
         tf.keras.layers.RandomBrightness(factor=0.2),

         # Adjust contrast randomly
         tf.keras.layers.RandomContrast(factor=0.2),

         # Add Gaussian noise to images
         tf.keras.layers.GaussianNoise(stddev=0.1)
     ])
```

```
[100] for image, _ in train_dataset.take(1):
        plt.figure(figsize=(10, 10))
        first_image = image[0]
        for i in range(9):
          ax = plt.subplot(3, 3, i + 1)
          augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
          plt.imshow(augmented_image[0] / 255)
          plt.axis('off')
```



#

Logistic Regression

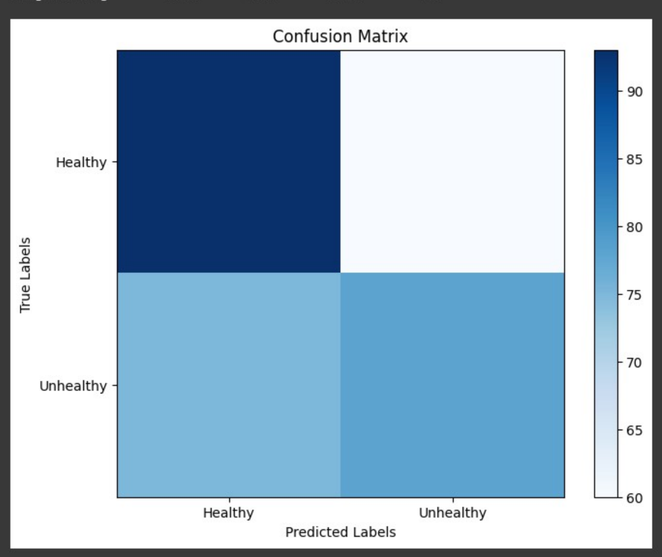
```
[108] # Prepare features and labels
      def preprocess_dataset(dataset, augment=False):
          features = []
          labels = []
          for batch, label_batch in dataset:
              if augment:  # Apply augmentation only for the training dataset
                  batch = data_augmentation(batch)
              for img, label in zip(batch, label_batch):
                  # Flatten and normalize images
                  flattened_img = tf.reshape(img, [-1]) / 255.0
                  features.append(flattened_img.numpy())
                  labels.append(label.numpy())
          return np.array(features), np.array(labels)

      # Preprocess datasets
      X_train, y_train = preprocess_dataset(train_dataset, augment=True)
      X_val, y_val = preprocess_dataset(validation_dataset, augment=False)
      X_test, y_test = preprocess_dataset(test_dataset, augment=False)
```

```
[109] # Train Logistic Regression
      log_reg = LogisticRegression(max_iter=1000)
      log_reg.fit(X_train, y_train)

      # Evaluate the model on the test dataset
      y_pred = log_reg.predict(X_test)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Test Accuracy: {accuracy:.2f}")
```

```
Test Accuracy: 0.56
```

```
[110] report = classification_report(y_test, y_pred)
      print(report)

      # Confusion Matrix
      conf_matrix = confusion_matrix(y_test, y_pred)
      print("Confusion Matrix:")
      print(conf_matrix)

      # Classification Report
      print("Classification Report:")
      print(classification_report(y_test, y_pred))

      # Plot the confusion matrix
      plt.figure(figsize=(8, 6))
      plt.imshow(conf_matrix, cmap='Blues', interpolation='nearest')
      plt.title("Confusion Matrix")
      plt.colorbar()
      plt.xticks(np.arange(2), labels=['Healthy', 'Unhealthy'])
      plt.yticks(np.arange(2), labels=['Healthy', 'Unhealthy'])
      plt.xlabel('Predicted Labels')
      plt.ylabel('True Labels')
      plt.show()
```

```
              precision    recall  f1-score   support

           0       0.55      0.61      0.58       153
           1       0.57      0.51      0.54       153

    accuracy                           0.56       306
   macro avg       0.56      0.56      0.56       306
weighted avg       0.56      0.56      0.56       306

Confusion Matrix:
[[93 60]
 [75 78]]
Classification Report:
              precision    recall  f1-score   support

           0       0.55      0.61      0.58       153
           1       0.57      0.51      0.54       153

    accuracy                           0.56       306
   macro avg       0.56      0.56      0.56       306
weighted avg       0.56      0.56      0.56       306
```



SVM

```
[111] # Scale the data
      scaler = StandardScaler()
      X_train_scaled = scaler.fit_transform(X_train)
      X_test_scaled = scaler.transform(X_test)

      # Initialize the SVM model
      svm_model = SVC(kernel='linear', C=1, gamma = 'scale',max_iter=5000)

      # Train the SVM model
      svm_model.fit(X_train_scaled, y_train)

      # Evaluate the model
      y_pred = svm_model.predict(X_test_scaled)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Test Accuracy with SVM: {accuracy:.2f}")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:297: ConvergenceWarning: Solver terminated early (max_iter=5000).  Consider pre-processing your data with StandardScaler or MinMaxScaler.
  warnings.warn(
Test Accuracy with SVM: 0.58
```

```
[112] #kernel = rbf
      svm_model = SVC(kernel='rbf', C=1, gamma = 'scale',max_iter=5000)

      # Train the SVM model
      svm_model.fit(X_train_scaled, y_train)

      # Evaluate the model
      y_pred = svm_model.predict(X_test_scaled)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Test Accuracy with SVM: {accuracy:.2f}")
```

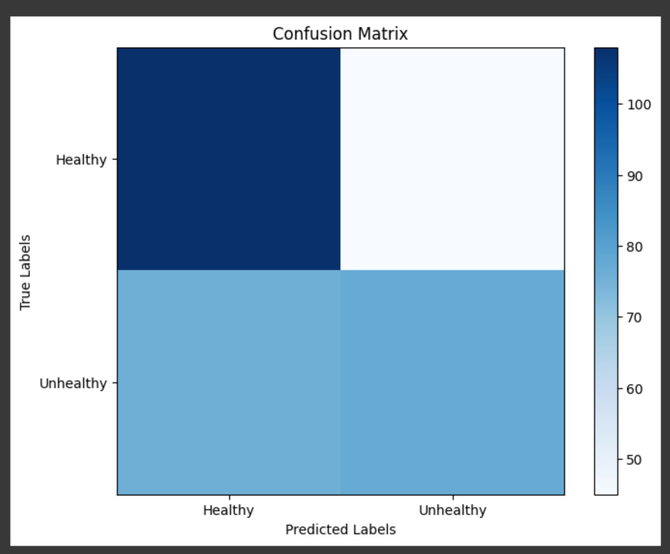
```
Test Accuracy with SVM: 0.59
```

```
[113] #kernel = rbf
      svm_model = SVC(kernel='poly', C=1, gamma = 'scale',max_iter=5000)

      # Train the SVM model
      svm_model.fit(X_train_scaled, y_train)

      # Evaluate the model
      y_pred = svm_model.predict(X_test_scaled)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Test Accuracy with SVM: {accuracy:.2f}")
```

```
Test Accuracy with SVM: 0.60
```

```
[114] report = classification_report(y_test, y_pred)
      print(report)

      # Confusion Matrix
      conf_matrix = confusion_matrix(y_test, y_pred)
      print("Confusion Matrix:")
      print(conf_matrix)

      # Classification Report
      print("Classification Report:")
      print(classification_report(y_test, y_pred))

      # Plot the confusion matrix
      plt.figure(figsize=(8, 6))
      plt.imshow(conf_matrix, cmap='Blues', interpolation='nearest')
      plt.title("Confusion Matrix")
      plt.colorbar()
      plt.xticks(np.arange(2), labels=['Healthy', 'Unhealthy'])
      plt.yticks(np.arange(2), labels=['Healthy', 'Unhealthy'])
      plt.xlabel('Predicted Labels')
      plt.ylabel('True Labels')
      plt.show()
```

```
              precision    recall  f1-score   support

           0       0.59      0.71      0.64       153
           1       0.63      0.50      0.56       153

    accuracy                           0.60       306
   macro avg       0.61      0.60      0.60       306
weighted avg       0.61      0.60      0.60       306

Confusion Matrix:
[[108  45]
 [ 76  77]]
Classification Report:
              precision    recall  f1-score   support

           0       0.59      0.71      0.64       153
           1       0.63      0.50      0.56       153

    accuracy                           0.60       306
   macro avg       0.61      0.60      0.60       306
weighted avg       0.61      0.60      0.60       306
```



Part 2: Neural Network Models

CNN

```
[120] #peprocessing for CNN model
      def preprocess_dataset(dataset, augment=False):
          features = []
          labels = []
          for batch, label_batch in dataset:
              if augment:  # Apply augmentation only for the training dataset
                  batch = data_augmentation(batch)
              # Resize images to (64, 64)
              batch = tf.image.resize(batch, (64, 64))
              # Normalize images to [0, 1]
              batch = batch / 255.0
              features.append(batch)
              labels.append(label_batch)
          return tf.concat(features, axis=0), tf.concat(labels, axis=0)

      X_train, y_train = preprocess_dataset(train_dataset, augment=True)
      X_val, y_val = preprocess_dataset(validation_dataset, augment=False)
      X_test, y_test = preprocess_dataset(test_dataset, augment=False)

      print(f"Training data shape: {X_train.shape}")
      print(f"Validation data shape: {X_val.shape}")
      print(f"Test data shape: {X_test.shape}")
```

```
Training data shape: (2142, 64, 64, 3)
Validation data shape: (612, 64, 64, 3)
Test data shape: (306, 64, 64, 3)
```

```
[121] from tensorflow.keras import layers, models

      # Define the CNN architecture
      cnn_model = models.Sequential([
          layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),  # Updated input shape
          layers.MaxPooling2D((2, 2)),
          layers.Conv2D(64, (3, 3), activation='relu'),
          layers.MaxPooling2D((2, 2)),
          layers.Conv2D(128, (3, 3), activation='relu'),
          layers.MaxPooling2D((2, 2)),
          layers.Flatten(),
          layers.Dense(128, activation='relu'),
          layers.Dropout(0.5),
          layers.Dense(1, activation='sigmoid')
      ])
```

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

```
[123]
      # Display the model architecture
      cnn_model.summary()
```

```
Model: "sequential_9"
```

Layer (type)	Output Shape	Param #
conv2d_24 (Conv2D)	(None, 62, 62, 32)	896
max_pooling2d_24 (MaxPooling2D)	(None, 31, 31, 32)	0
conv2d_25 (Conv2D)	(None, 29, 29, 64)	18,496
max_pooling2d_25 (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_26 (Conv2D)	(None, 12, 12, 128)	73,856
max_pooling2d_26 (MaxPooling2D)	(None, 6, 6, 128)	0
flatten_7 (Flatten)	(None, 4608)	0
dense_16 (Dense)	(None, 128)	589,952
dropout_7 (Dropout)	(None, 128)	0
dense_17 (Dense)	(None, 1)	129

```
Total params: 683,329 (2.61 MB)
Trainable params: 683,329 (2.61 MB)
Non-trainable params: 0 (0.00 B)
```

```
[124] # Define EarlyStopping
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    monitor='val_loss',    # Monitor validation loss
    patience=5,            # Stop training after 5 epochs with no improvement
    restore_best_weights=True  # Restore model weights from the epoch with the best validation loss
)
```

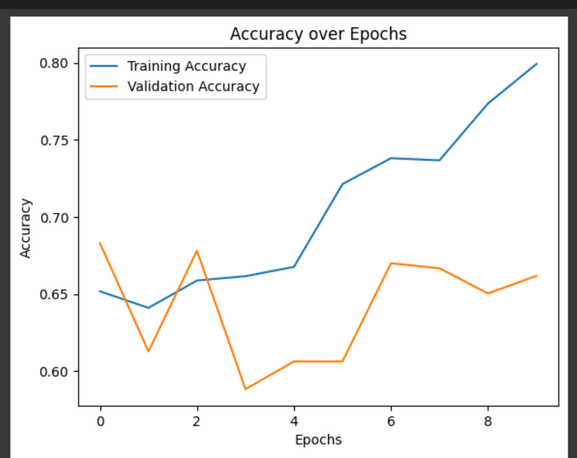
```
[128] # Train the model with EarlyStopping
history = cnn_model.fit(
    X_train, y_train,
    epochs=10,  # Set a high maximum number of epochs
    validation_data=(X_val, y_val),
    batch_size=32,
      # Add EarlyStopping here
)
```

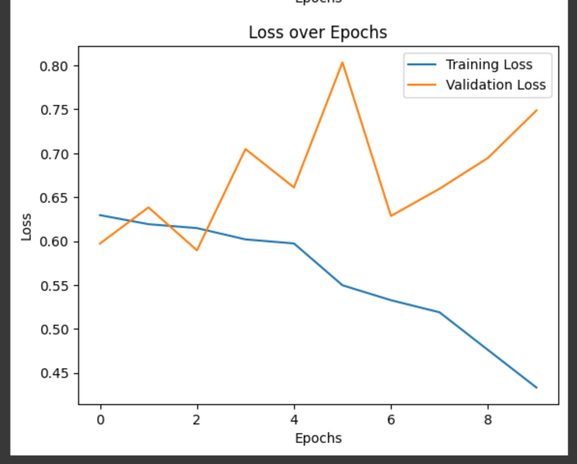
```
Epoch 1/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 22s 327ms/step - accuracy: 0.6521 - loss: 0.6253 - val_accuracy: 0.6830 - val_loss: 0.5972
Epoch 2/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 36s 255ms/step - accuracy: 0.6263 - loss: 0.6279 - val_accuracy: 0.6127 - val_loss: 0.6384
Epoch 3/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 20s 255ms/step - accuracy: 0.6787 - loss: 0.6010 - val_accuracy: 0.6781 - val_loss: 0.5895
Epoch 4/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 20s 256ms/step - accuracy: 0.6714 - loss: 0.5932 - val_accuracy: 0.5882 - val_loss: 0.7048
Epoch 5/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 20s 251ms/step - accuracy: 0.6519 - loss: 0.6111 - val_accuracy: 0.6062 - val_loss: 0.6611
Epoch 6/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 21s 255ms/step - accuracy: 0.7227 - loss: 0.5420 - val_accuracy: 0.6062 - val_loss: 0.8034
Epoch 7/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 17s 256ms/step - accuracy: 0.7340 - loss: 0.5424 - val_accuracy: 0.6699 - val_loss: 0.6286
Epoch 8/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 22s 271ms/step - accuracy: 0.7389 - loss: 0.5202 - val_accuracy: 0.6667 - val_loss: 0.6596
Epoch 9/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 21s 288ms/step - accuracy: 0.7809 - loss: 0.4774 - val_accuracy: 0.6503 - val_loss: 0.6947
Epoch 10/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 18s 253ms/step - accuracy: 0.8067 - loss: 0.4204 - val_accuracy: 0.6618 - val_loss: 0.7487
```

```
[136] # Evaluate on the test data
val_loss, val_accuracy = cnn_model.evaluate(X_test, y_test)
print(f"Accuracy: {val_accuracy:.2f}")

# Generate predictions
y_pred = cnn_model.predict(X_test)
y_pred_classes = (y_pred > 0.5).astype("int32")
```

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 87ms/step - accuracy: 0.6958 - loss: 0.6157
Accuracy: 0.71
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 89ms/step
```

```
[130] import matplotlib.pyplot as plt

# Plot accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

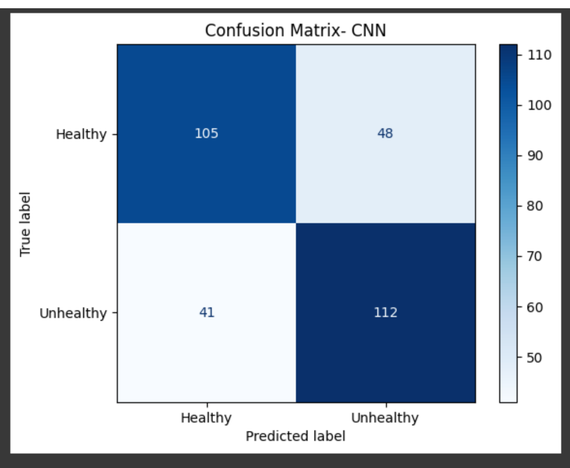





```
[139] # Generate a classification report
report = classification_report(y_test, y_pred_classes, target_names=['Healthy', 'Unhealthy'])
print("Classification Report:")
print(report)
```

```
Classification Report:
              precision    recall  f1-score   support

     Healthy       0.72      0.69      0.70       153
   Unhealthy       0.70      0.73      0.72       153

    accuracy                           0.71       306
   macro avg       0.71      0.71      0.71       306
weighted avg       0.71      0.71      0.71       306
```

```
[138] from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=['Healthy', 'Unhealthy'])
disp.plot(cmap='Blues', values_format='d')
plt.title('Confusion Matrix- CNN')
plt.show()
```



Transfer Learning

```
[12] preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

```
[13] # Create the base model from the pre-trained model MobileNet V2
     IMG_SHAPE = IMG_SIZE + (3,)
     base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                                    include_top=False,
                                                    weights='imagenet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_160_no_top.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
[14] image_batch, label_batch = next(iter(train_dataset))
     feature_batch = base_model(image_batch)
     print(feature_batch.shape)
```

```
(32, 5, 5, 1280)
```

```
[15] base_model.trainable = False
```

```
[140] # Let's take a look at the base model architecture
      base_model.summary()
```

Model: "mobilenetv2_1.00_160"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 160, 160, 3)	0	–
Conv1 (Conv2D)	(None, 80, 80, 32)	864	input_layer_1[0][0]
bn_Conv1 (BatchNormalization)	(None, 80, 80, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 80, 80, 32)	0	bn_Conv1[0][0]
expanded_conv_depthwise (DepthwiseConv2D)	(None, 80, 80, 32)	288	Conv1_relu[0][0]
expanded_conv_depthwise_… (BatchNormalization)	(None, 80, 80, 32)	128	expanded_conv_depthwi…
expanded_conv_depthwise_… (ReLU)	(None, 80, 80, 32)	0	expanded_conv_depthwi…
expanded_conv_project (Conv2D)	(None, 80, 80, 16)	512	expanded_conv_depthwi…
expanded_conv_project_BN (BatchNormalization)	(None, 80, 80, 16)	64	expanded_conv_project…
block_1_expand (Conv2D)	(None, 80, 80, 96)	1,536	expanded_conv_project…
block_1_expand_BN (BatchNormalization)	(None, 80, 80, 96)	384	block_1_expand[0][0]
block_1_expand_relu (ReLU)	(None, 80, 80, 96)	0	block_1_expand_BN[0][…
block_1_pad (ZeroPadding2D)	(None, 81, 81, 96)	0	block_1_expand_relu[0…
block_1_depthwise (DepthwiseConv2D)	(None, 40, 40, 96)	864	block_1_pad[0][0]
block_1_depthwise_BN (BatchNormalization)	(None, 40, 40, 96)	384	block_1_depthwise[0][…
block_1_depthwise_relu (ReLU)	(None, 40, 40, 96)	0	block_1_depthwise_BN[…
block_1_project (Conv2D)	(None, 40, 40, 24)	2,304	block_1_depthwise_rel…
block_1_project_BN (BatchNormalization)	(None, 40, 40, 24)	96	block_1_project[0][0]
block_2_expand (Conv2D)	(None, 40, 40, 144)	3,456	block_1_project_BN[0]…
block_2_expand_BN (BatchNormalization)	(None, 40, 40, 144)	576	block_2_expand[0][0]
block_2_expand_relu (ReLU)	(None, 40, 40, 144)	0	block_2_expand_BN[0][…
block_2_depthwise (DepthwiseConv2D)	(None, 40, 40, 144)	1,296	block_2_expand_relu[0…
block_2_depthwise_BN (BatchNormalization)	(None, 40, 40, 144)	576	block_2_depthwise[0][…
block_2_depthwise_relu (ReLU)	(None, 40, 40, 144)	0	block_2_depthwise_BN[…
block_2_project (Conv2D)	(None, 40, 40, 24)	3,456	block_2_depthwise_rel…
block_2_project_BN (BatchNormalization)	(None, 40, 40, 24)	96	block_2_project[0][0]
block_2_add (Add)	(None, 40, 40, 24)	0	block_1_project_BN[0]… block_2_project_BN[0]…
block_3_expand (Conv2D)	(None, 40, 40, 144)	3,456	block_2_add[0][0]
block_3_expand_BN (BatchNormalization)	(None, 40, 40, 144)	576	block_3_expand[0][0]
block_3_expand_relu (ReLU)	(None, 40, 40, 144)	0	block_3_expand_BN[0][…
block_3_pad (ZeroPadding2D)	(None, 41, 41, 144)	0	block_3_expand_relu[0…
block_3_depthwise (DepthwiseConv2D)	(None, 20, 20, 144)	1,296	block_3_pad[0][0]
block_3_depthwise_BN (BatchNormalization)	(None, 20, 20, 144)	576	block_3_depthwise[0][…
block_3_depthwise_relu (ReLU)	(None, 20, 20, 144)	0	block_3_depthwise_BN[…
block_3_project (Conv2D)	(None, 20, 20, 32)	4,608	block_3_depthwise_rel…
block_3_project_BN (BatchNormalization)	(None, 20, 20, 32)	128	block_3_project[0][0]
block_4_expand (Conv2D)	(None, 20, 20, 192)	6,144	block_3_project_BN[0]…
block_4_expand_BN (BatchNormalization)	(None, 20, 20, 192)	768	block_4_expand[0][0]
block_4_expand_relu (ReLU)	(None, 20, 20, 192)	0	block_4_expand_BN[0][…
block_4_depthwise (DepthwiseConv2D)	(None, 20, 20, 192)	1,728	block_4_expand_relu[0…
block_4_depthwise_BN (BatchNormalization)	(None, 20, 20, 192)	768	block_4_depthwise[0][…
block_4_depthwise_relu (ReLU)	(None, 20, 20, 192)	0	block_4_depthwise_BN[…
block_4_project (Conv2D)	(None, 20, 20, 32)	6,144	block_4_depthwise_rel…
block_4_project_BN (BatchNormalization)	(None, 20, 20, 32)	128	block_4_project[0][0]
block_4_add (Add)	(None, 20, 20, 32)	0	block_3_project_BN[0]… block_4_project_BN[0]…
block_5_expand (Conv2D)	(None, 20, 20, 192)	6,144	block_4_add[0][0]
block_5_expand_BN (BatchNormalization)	(None, 20, 20, 192)	768	block_5_expand[0][0]
block_5_expand_relu (ReLU)	(None, 20, 20, 192)	0	block_5_expand_BN[0][…
block_5_depthwise (DepthwiseConv2D)	(None, 20, 20, 192)	1,728	block_5_expand_relu[0…
block_5_depthwise_BN (BatchNormalization)	(None, 20, 20, 192)	768	block_5_depthwise[0][…
block_5_depthwise_relu (ReLU)	(None, 20, 20, 192)	0	block_5_depthwise_BN[…
block_5_project (Conv2D)	(None, 20, 20, 32)	6,144	block_5_depthwise_rel…
block_5_project_BN (BatchNormalization)	(None, 20, 20, 32)	128	block_5_project[0][0]
block_5_add (Add)	(None, 20, 20, 32)	0	block_4_add[0][0], block_5_project_BN[0]…
block_6_expand (Conv2D)	(None, 20, 20, 192)	6,144	block_5_add[0][0]
block_6_expand_BN (BatchNormalization)	(None, 20, 20, 192)	768	block_6_expand[0][0]
block_6_expand_relu (ReLU)	(None, 20, 20, 192)	0	block_6_expand_BN[0][…

block_6_pad (ZeroPadding2D)	(None, 21, 21, 192)	0	block_6_expand_relu[0…
block_6_depthwise (DepthwiseConv2D)	(None, 10, 10, 192)	1,728	block_6_pad[0][0]
block_6_depthwise_BN (BatchNormalization)	(None, 10, 10, 192)	768	block_6_depthwise[0][…
block_6_depthwise_relu (ReLU)	(None, 10, 10, 192)	0	block_6_depthwise_BN[…
block_6_project (Conv2D)	(None, 10, 10, 64)	12,288	block_6_depthwise_rel…
block_6_project_BN (BatchNormalization)	(None, 10, 10, 64)	256	block_6_project[0][0]
block_7_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_6_project_BN[0]…
block_7_expand_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_7_expand[0][0]
block_7_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_7_expand_BN[0][…
block_7_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_7_expand_relu[0…
block_7_depthwise_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_7_depthwise[0][…
block_7_depthwise_relu (ReLU)	(None, 10, 10, 384)	0	block_7_depthwise_BN[…
block_7_project (Conv2D)	(None, 10, 10, 64)	24,576	block_7_depthwise_rel…
block_7_project_BN (BatchNormalization)	(None, 10, 10, 64)	256	block_7_project[0][0]
block_7_add (Add)	(None, 10, 10, 64)	0	block_6_project_BN[0]… block_7_project_BN[0]…
block_8_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_7_add[0][0]
block_8_expand_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_8_expand[0][0]
block_8_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_8_expand_BN[0][…
block_8_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_8_expand_relu[0…
block_8_depthwise_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_8_depthwise[0][…
block_8_depthwise_relu (ReLU)	(None, 10, 10, 384)	0	block_8_depthwise_BN[…
block_8_project (Conv2D)	(None, 10, 10, 64)	24,576	block_8_depthwise_rel…
block_8_project_BN (BatchNormalization)	(None, 10, 10, 64)	256	block_8_project[0][0]
block_8_add (Add)	(None, 10, 10, 64)	0	block_7_add[0][0], block_8_project_BN[0]…
block_9_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_8_add[0][0]
block_9_expand_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_9_expand[0][0]
block_9_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_9_expand_BN[0][…
block_9_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_9_expand_relu[0…
block_9_depthwise_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_9_depthwise[0][…
block_9_depthwise_relu (ReLU)	(None, 10, 10, 384)	0	block_9_depthwise_BN[…
block_9_project (Conv2D)	(None, 10, 10, 64)	24,576	block_9_depthwise_rel…
block_9_project_BN (BatchNormalization)	(None, 10, 10, 64)	256	block_9_project[0][0]
block_9_add (Add)	(None, 10, 10, 64)	0	block_8_add[0][0], block_9_project_BN[0]…
block_10_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_9_add[0][0]
block_10_expand_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_10_expand[0][0]
block_10_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_10_expand_BN[0]…
block_10_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_10_expand_relu[…
block_10_depthwise_BN (BatchNormalization)	(None, 10, 10, 384)	1,536	block_10_depthwise[0]…
block_10_depthwise_relu (ReLU)	(None, 10, 10, 384)	0	block_10_depthwise_BN…
block_10_project (Conv2D)	(None, 10, 10, 96)	36,864	block_10_depthwise_re…
block_10_project_BN (BatchNormalization)	(None, 10, 10, 96)	384	block_10_project[0][0]
block_11_expand (Conv2D)	(None, 10, 10, 576)	55,296	block_10_project_BN[0…
block_11_expand_BN (BatchNormalization)	(None, 10, 10, 576)	2,304	block_11_expand[0][0]
block_11_expand_relu (ReLU)	(None, 10, 10, 576)	0	block_11_expand_BN[0]…
block_11_depthwise (DepthwiseConv2D)	(None, 10, 10, 576)	5,184	block_11_expand_relu[…
block_11_depthwise_BN (BatchNormalization)	(None, 10, 10, 576)	2,304	block_11_depthwise[0]…
block_11_depthwise_relu (ReLU)	(None, 10, 10, 576)	0	block_11_depthwise_BN…
block_11_project (Conv2D)	(None, 10, 10, 96)	55,296	block_11_depthwise_re…
block_11_project_BN (BatchNormalization)	(None, 10, 10, 96)	384	block_11_project[0][0]
block_11_add (Add)	(None, 10, 10, 96)	0	block_10_project_BN[0… block_11_project_BN[0…
block_12_expand (Conv2D)	(None, 10, 10, 576)	55,296	block_11_add[0][0]
block_12_expand_BN (BatchNormalization)	(None, 10, 10, 576)	2,304	block_12_expand[0][0]
block_12_expand_relu (ReLU)	(None, 10, 10, 576)	0	block_12_expand_BN[0]…
block_12_depthwise (DepthwiseConv2D)	(None, 10, 10, 576)	5,184	block_12_expand_relu[…
block_12_depthwise_BN (BatchNormalization)	(None, 10, 10, 576)	2,304	block_12_depthwise[0]…
block_12_depthwise_relu (ReLU)	(None, 10, 10, 576)	0	block_12_depthwise_BN…
block_12_project (Conv2D)	(None, 10, 10, 96)	55,296	block_12_depthwise_re…
block_12_project_BN (BatchNormalization)	(None, 10, 10, 96)	384	block_12_project[0][0]
block_12_add (Add)	(None, 10, 10, 96)	0	block_11_add[0][0], block_12_project_BN[0…
block_13_expand (Conv2D)	(None, 10, 10, 576)	55,296	block_12_add[0][0]
block_13_expand_BN (BatchNormalization)	(None, 10, 10, 576)	2,304	block_13_expand[0][0]
block_13_expand_relu (ReLU)	(None, 10, 10, 576)	0	block_13_expand_BN[0]…
block_13_pad (ZeroPadding2D)	(None, 11, 11, 576)	0	block_13_expand_relu[…
block_13_depthwise (DepthwiseConv2D)	(None, 5, 5, 576)	5,184	block_13_pad[0][0]
block_13_depthwise_BN (BatchNormalization)	(None, 5, 5, 576)	2,304	block_13_depthwise[0]…
block_13_depthwise_relu (ReLU)	(None, 5, 5, 576)	0	block_13_depthwise_BN…
block_13_project (Conv2D)	(None, 5, 5, 160)	92,160	block_13_depthwise_re…
block_13_project_BN (BatchNormalization)	(None, 5, 5, 160)	640	block_13_project[0][0]
block_14_expand (Conv2D)	(None, 5, 5, 960)	153,600	block_13_project_BN[0…
block_14_expand_BN (BatchNormalization)	(None, 5, 5, 960)	3,840	block_14_expand[0][0]
block_14_expand_relu (ReLU)	(None, 5, 5, 960)	0	block_14_expand_BN[0]…
block_14_depthwise (DepthwiseConv2D)	(None, 5, 5, 960)	8,640	block_14_expand_relu[…
block_14_depthwise_BN (BatchNormalization)	(None, 5, 5, 960)	3,840	block_14_depthwise[0]…
block_14_depthwise_relu (ReLU)	(None, 5, 5, 960)	0	block_14_depthwise_BN…
block_14_project (Conv2D)	(None, 5, 5, 160)	153,600	block_14_depthwise_re…
block_14_project_BN (BatchNormalization)	(None, 5, 5, 160)	640	block_14_project[0][0]
block_14_add (Add)	(None, 5, 5, 160)	0	block_13_project_BN[0… block_14_project_BN[0…
block_15_expand (Conv2D)	(None, 5, 5, 960)	153,600	block_14_add[0][0]
block_15_expand_BN (BatchNormalization)	(None, 5, 5, 960)	3,840	block_15_expand[0][0]
block_15_expand_relu (ReLU)	(None, 5, 5, 960)	0	block_15_expand_BN[0]…
block_15_depthwise (DepthwiseConv2D)	(None, 5, 5, 960)	8,640	block_15_expand_relu[…

block_15_depthwise_BN (BatchNormalization)	(None, 5, 5, 960)	3,840	block_15_depthwise[0]…
block_15_depthwise_relu (ReLU)	(None, 5, 5, 960)	0	block_15_depthwise_BN…
block_15_project (Conv2D)	(None, 5, 5, 160)	153,600	block_15_depthwise_re…
block_15_project_BN (BatchNormalization)	(None, 5, 5, 160)	640	block_15_project[0][0]
block_15_add (Add)	(None, 5, 5, 160)	0	block_14_add[0][0], block_15_project_BN[0]…
block_16_expand (Conv2D)	(None, 5, 5, 960)	153,600	block_15_add[0][0]
block_16_expand_BN (BatchNormalization)	(None, 5, 5, 960)	3,840	block_16_expand[0][0]
block_16_expand_relu (ReLU)	(None, 5, 5, 960)	0	block_16_expand_BN[0]…
block_16_depthwise (DepthwiseConv2D)	(None, 5, 5, 960)	8,640	block_16_expand_relu[…
block_16_depthwise_BN (BatchNormalization)	(None, 5, 5, 960)	3,840	block_16_depthwise[0]…
block_16_depthwise_relu (ReLU)	(None, 5, 5, 960)	0	block_16_depthwise_BN…
block_16_project (Conv2D)	(None, 5, 5, 320)	307,200	block_16_depthwise_re…
block_16_project_BN (BatchNormalization)	(None, 5, 5, 320)	1,280	block_16_project[0][0]
Conv_1 (Conv2D)	(None, 5, 5, 1280)	409,600	block_16_project_BN[0…
Conv_1_bn (BatchNormalization)	(None, 5, 5, 1280)	5,120	Conv_1[0][0]
out_relu (ReLU)	(None, 5, 5, 1280)	0	Conv_1_bn[0][0]

Total params: 2,257,984 (8.61 MB)
Trainable params: 1,861,440 (7.10 MB)
Non-trainable params: 396,544 (1.51 MB)

```
[44] global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
     feature_batch_average = global_average_layer(feature_batch)
     print(feature_batch_average.shape)
```

```
(32, 1280)
```

```
[45] prediction_layer = tf.keras.layers.Dense(1, activation='sigmoid')
     prediction_batch = prediction_layer(feature_batch_average)
     print(prediction_batch.shape)
```

```
(32, 1)
```

```
[46] inputs = tf.keras.Input(shape=(160, 160, 3))
     x = data_augmentation(inputs)
     x = preprocess_input(x)
     x = base_model(x, training=False)
     x = global_average_layer(x)
     x = tf.keras.layers.Dropout(0.2)(x)
     outputs = prediction_layer(x)
     model = tf.keras.Model(inputs, outputs)
```

```
[47] base_learning_rate = 0.0001
     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=base_learning_rate),
                   loss=tf.keras.losses.BinaryCrossentropy(),
                   metrics=[tf.keras.metrics.BinaryAccuracy(threshold=0.5, name='accuracy')])
```

```
[48] initial_epochs = 10

     loss0, accuracy0 = model.evaluate(validation_dataset)
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 17s 717ms/step - accuracy: 0.5308 - loss: 0.9686
```

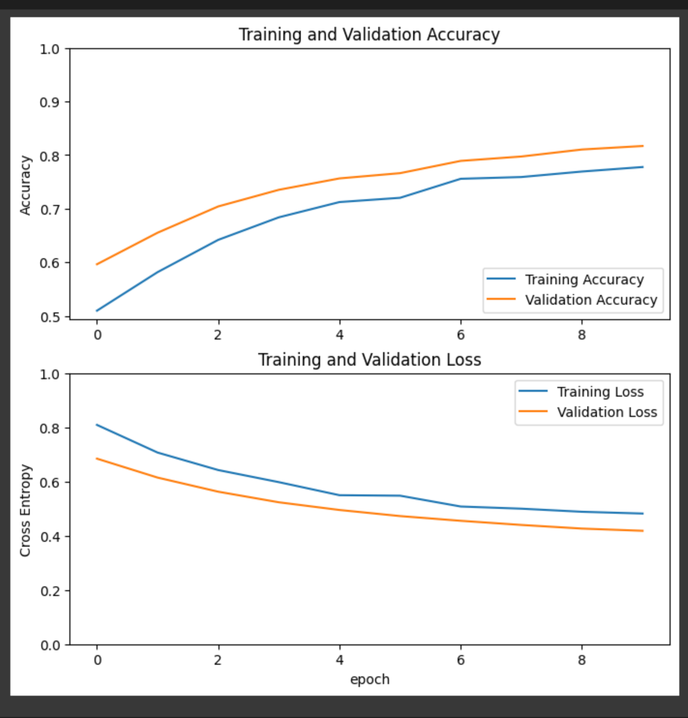
```
[49] print("initial loss: {:.2f}".format(loss0))
     print("initial accuracy: {:.2f}".format(accuracy0))
```

```
initial loss: 0.94
initial accuracy: 0.53
```

```
[23] history = model.fit(train_dataset,
                         epochs=initial_epochs,
                         validation_data=validation_dataset)
```

```
Epoch 1/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 234s 3s/step - accuracy: 0.4818 - loss: 0.8532 - val_accuracy: 0.5964 - val_loss: 0.6845
Epoch 2/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 98s 1s/step - accuracy: 0.5601 - loss: 0.7329 - val_accuracy: 0.6552 - val_loss: 0.6146
Epoch 3/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 147s 2s/step - accuracy: 0.6326 - loss: 0.6565 - val_accuracy: 0.7042 - val_loss: 0.5625
Epoch 4/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 140s 2s/step - accuracy: 0.6789 - loss: 0.6026 - val_accuracy: 0.7353 - val_loss: 0.5234
Epoch 5/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 137s 1s/step - accuracy: 0.7042 - loss: 0.5591 - val_accuracy: 0.7565 - val_loss: 0.4950
Epoch 6/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 144s 1s/step - accuracy: 0.7222 - loss: 0.5451 - val_accuracy: 0.7663 - val_loss: 0.4725
Epoch 7/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 160s 2s/step - accuracy: 0.7574 - loss: 0.5164 - val_accuracy: 0.7892 - val_loss: 0.4551
Epoch 8/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 115s 1s/step - accuracy: 0.7713 - loss: 0.4926 - val_accuracy: 0.7974 - val_loss: 0.4398
Epoch 9/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 147s 1s/step - accuracy: 0.7794 - loss: 0.4857 - val_accuracy: 0.8105 - val_loss: 0.4264
Epoch 10/10
67/67 ━━━━━━━━━━━━━━━━━━━━ 98s 1s/step - accuracy: 0.7832 - loss: 0.4812 - val_accuracy: 0.8170 - val_loss: 0.4182
```

```
[24] acc = history.history['accuracy']
     val_acc = history.history['val_accuracy']

     loss = history.history['loss']
     val_loss = history.history['val_loss']

     plt.figure(figsize=(8, 8))
     plt.subplot(2, 1, 1)
     plt.plot(acc, label='Training Accuracy')
     plt.plot(val_acc, label='Validation Accuracy')
     plt.legend(loc='lower right')
     plt.ylabel('Accuracy')
     plt.ylim([min(plt.ylim()),1])
     plt.title('Training and Validation Accuracy')

     plt.subplot(2, 1, 2)
     plt.plot(loss, label='Training Loss')
     plt.plot(val_loss, label='Validation Loss')
     plt.legend(loc='upper right')
     plt.ylabel('Cross Entropy')
     plt.ylim([0,1.0])
     plt.title('Training and Validation Loss')
     plt.xlabel('epoch')
     plt.show()
```




```
[50] base_model.trainable = True
```

```
[51] # Let's take a look to see how many layers are in the base model
     print("Number of layers in the base model: ", len(base_model.layers))

     # Fine-tune from this layer onwards
     fine_tune_at = 100

     # Freeze all the layers before the `fine_tune_at` layer
     for layer in base_model.layers[:fine_tune_at]:
       layer.trainable = False
```

```
Number of layers in the base model:  154
```

```
[52] model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                   optimizer = tf.keras.optimizers.RMSprop(learning_rate=base_learning_rate/10),
                   metrics=[tf.keras.metrics.BinaryAccuracy(threshold=0.5, name='accuracy')])
```

```
[53] model.summary()
```

```
Model: "functional_2"
```

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 160, 160, 3)	0
sequential (Sequential)	(None, 160, 160, 3)	0
true_divide_1 (TrueDivide)	(None, 160, 160, 3)	0
subtract_1 (Subtract)	(None, 160, 160, 3)	0
mobilenetv2_1.00_160 (Functional)	(None, 5, 5, 1280)	2,257,984
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 1280)	0
dropout_1 (Dropout)	(None, 1280)	0
dense_1 (Dense)	(None, 1)	1,281

```
Total params: 2,259,265 (8.62 MB)
Trainable params: 1,862,721 (7.11 MB)
Non-trainable params: 396,544 (1.51 MB)
```

```
[54] len(model.trainable_variables)
```

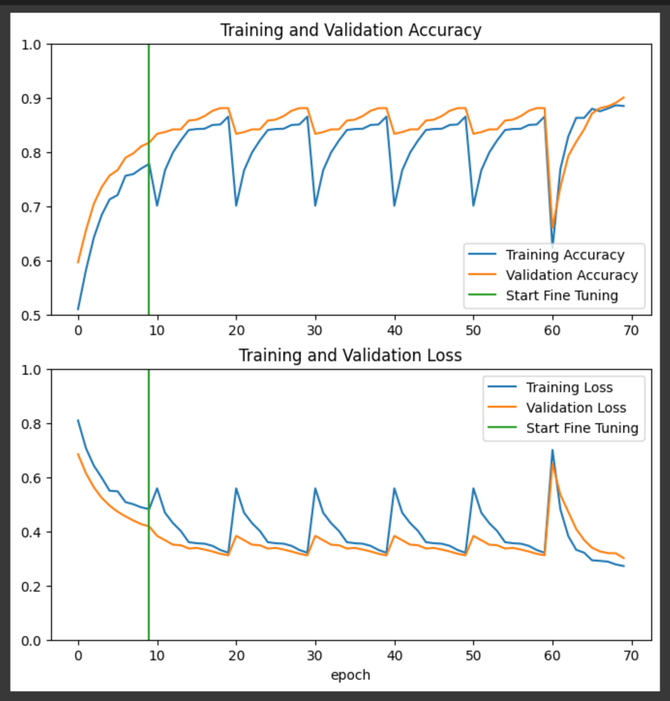
```
56
```

```
[55] fine_tune_epochs = 10
     total_epochs =  initial_epochs + fine_tune_epochs

     history_fine = model.fit(train_dataset,
                              epochs=total_epochs,
                              initial_epoch=len(history.epoch),
                              validation_data=validation_dataset)
```

```
Epoch 11/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 142s 2s/step - accuracy: 0.5893 - loss: 0.7709 - val_accuracy: 0.6601 - val_loss: 0.6495
Epoch 12/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 140s 2s/step - accuracy: 0.7585 - loss: 0.4848 - val_accuracy: 0.7353 - val_loss: 0.5353
Epoch 13/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 140s 2s/step - accuracy: 0.8413 - loss: 0.3709 - val_accuracy: 0.7925 - val_loss: 0.4718
Epoch 14/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 145s 2s/step - accuracy: 0.8645 - loss: 0.3310 - val_accuracy: 0.8186 - val_loss: 0.4085
Epoch 15/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 143s 2s/step - accuracy: 0.8633 - loss: 0.3159 - val_accuracy: 0.8415 - val_loss: 0.3679
Epoch 16/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 140s 2s/step - accuracy: 0.8769 - loss: 0.2831 - val_accuracy: 0.8709 - val_loss: 0.3397
Epoch 17/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 137s 2s/step - accuracy: 0.8715 - loss: 0.2902 - val_accuracy: 0.8807 - val_loss: 0.3259
Epoch 18/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 144s 2s/step - accuracy: 0.8772 - loss: 0.2971 - val_accuracy: 0.8840 - val_loss: 0.3196
Epoch 19/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 144s 2s/step - accuracy: 0.8844 - loss: 0.2768 - val_accuracy: 0.8905 - val_loss: 0.3189
Epoch 20/20
67/67 ━━━━━━━━━━━━━━━━━━━━ 145s 2s/step - accuracy: 0.8887 - loss: 0.2754 - val_accuracy: 0.9003 - val_loss: 0.3014
```

```
[56] acc += history_fine.history['accuracy']
     val_acc += history_fine.history['val_accuracy']

     loss += history_fine.history['loss']
     val_loss += history_fine.history['val_loss']
```

```
[58] plt.figure(figsize=(8, 8))
     plt.subplot(2, 1, 1)
     plt.plot(acc, label='Training Accuracy')
     plt.plot(val_acc, label='Validation Accuracy')
     plt.ylim([0.5, 1])
     plt.plot([initial_epochs-1,initial_epochs-1],
              plt.ylim(), label='Start Fine Tuning')
     plt.legend(loc='lower right')
     plt.title('Training and Validation Accuracy')

     plt.subplot(2, 1, 2)
     plt.plot(loss, label='Training Loss')
     plt.plot(val_loss, label='Validation Loss')
     plt.ylim([0, 1.0])
     plt.plot([initial_epochs-1,initial_epochs-1],
             plt.ylim(), label='Start Fine Tuning')
     plt.legend(loc='upper right')
     plt.title('Training and Validation Loss')
     plt.xlabel('epoch')
     plt.show()
```




```
[59] loss, accuracy = model.evaluate(test_dataset)
     print('Test accuracy :', accuracy)
```

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 58s 6s/step - accuracy: 0.8904 - loss: 0.2567
Test accuracy : 0.8856208920478821
```

```
[63] # Retrieve a batch of images from the test set
     image_batch, label_batch = test_dataset.as_numpy_iterator().next()
     predictions = model.predict_on_batch(image_batch).flatten()
     predictions = tf.where(predictions < 0.5, 0, 1)

     print('Predictions:\n', predictions.numpy())
     print('Labels:\n', label_batch)

     plt.figure(figsize=(10, 10))
     for i in range(9):
       ax = plt.subplot(3, 3, i + 1)
       plt.imshow(image_batch[i].astype("uint8"))
       plt.title(class_names[predictions[i]])
       plt.axis("off")
```

```
Predictions:
 [0 1 1 1 0 0 1 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1 0 0 1]
Labels:
 [0 1 1 1 0 0 1 1 1 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 0 0 0 1 1 1 1 1]
```

Has Sana been making healthy food choices?

Data Collection

Lately, I've been finding myself eating out a lot, trying different restaurants and cuisines. It's always fun to explore new places, but as I was scrolling through my phone gallery one day, I realized I had taken quite a few photos of my meals whenever I eat out, which provided a mix of food types, from fresh salads and smoothie bowls to burgers and pizzas. This got me thinking—am I making healthy food choices or indulging a bit too much in processed foods?

To investigate this, I decided to classify my meal images into two categories: healthy and unhealthy. For this task, I manually curated a dataset of 60 images. This dataset consists of 30 images labeled as "healthy" and 30 labeled as "unhealthy."

The images were labeled based on the visual content of the food. If a meal visibly contained more vegetables, fruits, or whole foods, it was classified as "healthy." On the other hand, if the food appeared heavily processed—like fast food, pastries, or fried dishes—it was labeled as "unhealthy." The labeling process was subjective but guided by general nutritional knowledge about the benefits of whole, plant-based foods versus processed and high-calorie foods.

With 30 images in each category, I aimed to build a classification model that could help me understand whether my meals were leaning more toward health-conscious choices or indulgent treats.

Data Pipeline

[33]:
```
from IPython.display import Image, display
image_path = 'pipeline.png'
# Display the image
display(Image(filename=image_path))
```

Converting Data to Python

[34]:
```
# Import standard libraries for file handling and numerical operations
import os
import numpy as np

# Import image processing library for handling image files
from PIL import Image

# Define the base data directory where your images are stored
data_dir = '/Users/sana/cs156-ml-assignment-1/data'

# Define paths to 'healthy' and 'unhealthy' image folders
healthy_dir = os.path.join(data_dir, 'healthy')
unhealthy_dir = os.path.join(data_dir, 'unhealthy')

# Set the desired image size to resize all images to a uniform format
image_size = (64, 64)

# Function to load and process images from a given folder
def load_images_from_folder(folder, label):
    images = []  # List to store processed image arrays
    labels = []  # List to store corresponding labels (0 for healthy, 1 for unhealthy)

    # Define valid image file extensions that will be processed
    valid_extensions = ('.jpg', '.jpeg', '.png',)

    # Loop through all files in the folder
    for filename in os.listdir(folder):
        # Check if the file has a valid image extension
        if filename.lower().endswith(valid_extensions):
            # Create the full file path to the image
            img_path = os.path.join(folder, filename)
            try:
                # Open the image using PIL
                with Image.open(img_path) as img:
                    img = img.convert('RGB')  # Convert image to RGB format
                    img = img.resize(image_size)  # Resize image to predefined dimensions
                    img_array = np.array(img)  # Convert the image to a numpy array
                    img_array = img_array / 255.0  # Normalize pixel values to the range [0,1]
                    images.append(img_array)  # Append processed image array to the list
                    labels.append(label)  # Append corresponding label to the labels list
            except Exception as e:
                # Handle exceptions, such as unsupported file formats or corrupted images
                print("Error loading image {}: {}".format(img_path, e))

    # Return two lists: one with images and one with labels
    return images, labels


# Load images and assign labels (0 for healthy, 1 for unhealthy)
healthy_images, healthy_labels = load_images_from_folder(healthy_dir, label=0)
unhealthy_images, unhealthy_labels = load_images_from_folder(unhealthy_dir, label=1)

# Combine the healthy and unhealthy images and labels into a single dataset
images = healthy_images + unhealthy_images
labels = healthy_labels + unhealthy_labels

# Convert the lists to numpy arrays for easier manipulation during model training
X = np.array(images)  # Feature data (image arrays)
y = np.array(labels)  # Target labels (0 or 1)
```

Cleaning and Pre Processing

Initially, when gathering the images from the dataset, I carefully performed a manual selection process to ensure that the images were high-quality and relevant to the task of classifying healthy vs. unhealthy food. Any images that included **extraneous elements**—such as humans, distracting backgrounds, or objects that could confuse the model—were removed. This was important to maintain a focus on the food itself, as including irrelevant elements could have led to misleading classifications. Additionally, any **blurry** or poorly captured images that might obscure important visual features of the food were excluded to avoid affecting the accuracy of the classification model.

Once the dataset was cleaned and only **clear, focused images of food** were retained, the next step in the image preprocessing workflow was to standardize the image sizes. All the images were resized to **64x64 pixels** to ensure consistency across the dataset. This resizing ensures that the machine learning model can process the images uniformly, as varying image sizes could disrupt feature extraction and model performance.

After resizing, the images were converted from their original 3D format (which consists of `height`, `width`, and `channels`—representing the pixel dimensions and color channels) to a **2D format**. Specifically, the line `X_flat = X.reshape(num_samples, -1)` was used to flatten each image into a **1D array** of pixel values. This transformation is necessary because models like Logistic Regression and SVM require **2D input** where each row represents a single image, and each column corresponds to a pixel value (or feature). By flattening the images, we transformed the data into a format that is compatible with these models, making it easier to process and classify.

These models require flattened data where each pixel is treated as a feature, and this step effectively reshapes our image data into the required format. And as color is crucial for distinguishing between healthy and unhealthy food (for example, if the color of fresh veggies vs. processed food is important), we stick with the RGB images and do not convert it into grayscale.

[35]:
```
#Flatten the image
num_samples = X.shape[0]
X_flat = X.reshape(num_samples, -1)
```

[36]:
```
# Print data shapes
print("X shape:", X.shape)
print("X_flat shape:", X_flat.shape)
print("y shape:", y.shape)
```

```
X shape: (60, 64, 64, 3)
X_flat shape: (60, 12288)
y shape: (60,)
```

[37]:
```
# Display samples of healthy and unhealthy food
import matplotlib.pyplot as plt

# Number of samples to display per class
num_samples_per_class = 4

# Get indices of healthy and unhealthy samples
healthy_indices = np.where(y == 0)[0]
unhealthy_indices = np.where(y == 1)[0]

# Select samples
healthy_samples = X[healthy_indices[:num_samples_per_class]]
unhealthy_samples = X[unhealthy_indices[:num_samples_per_class]]

# Create a figure with subplots
fig, axes = plt.subplots(nrows=2, ncols=num_samples_per_class, figsize=(15, 6))
plt.subplots_adjust(wspace=0.1, hspace=0.2)

# Display healthy food images in the first row
for i in range(num_samples_per_class):
    img = healthy_samples[i].reshape(image_size[1], image_size[0], 3)
    axes[0, i].imshow(img)
    axes[0, i].set_title('Healthy')
    axes[0, i].axis('off')

# Display unhealthy food images in the second row
for i in range(num_samples_per_class):
    img = unhealthy_samples[i].reshape(image_size[1], image_size[0], 3)
    axes[1, i].imshow(img)
    axes[1, i].set_title('Unhealthy')
    axes[1, i].axis('off')

# Show the plot
plt.tight_layout()
plt.show()
```

[38]:
```
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
# Use an 80/20 split: 80% training data, 20% testing data
X_train, X_test, y_train, y_test = train_test_split(
    X_flat, y, test_size=0.2, random_state=42, stratify=y
)
print("Training data shape:", X_train.shape)
print("Training labels shape:", y_train.shape)
print("Testing data shape:", X_test.shape)
print("Testing labels shape:", y_test.shape)
```

```
Training data shape: (48, 12288)
Training labels shape: (48,)
Testing data shape: (12, 12288)
Testing labels shape: (12,)
```

[39]:
```
# Count the number of samples per class
unique, counts = np.unique(y, return_counts=True)
class_distribution = dict(zip(unique, counts))
print("Class distribution:", class_distribution)
```

```
Class distribution: {0: 30, 1: 30}
```

[40]:
```
# Compute overall mean and standard deviation
mean_pixel_value = np.mean(X_flat) #indicates the average brightness of the images
std_pixel_value = np.std(X_flat) #shows the contrast level in the images
print(f"Overall Mean Pixel Value: {mean_pixel_value:.4f}")
print(f"Overall Standard Deviation of Pixel Values: {std_pixel_value:.4f}")
```

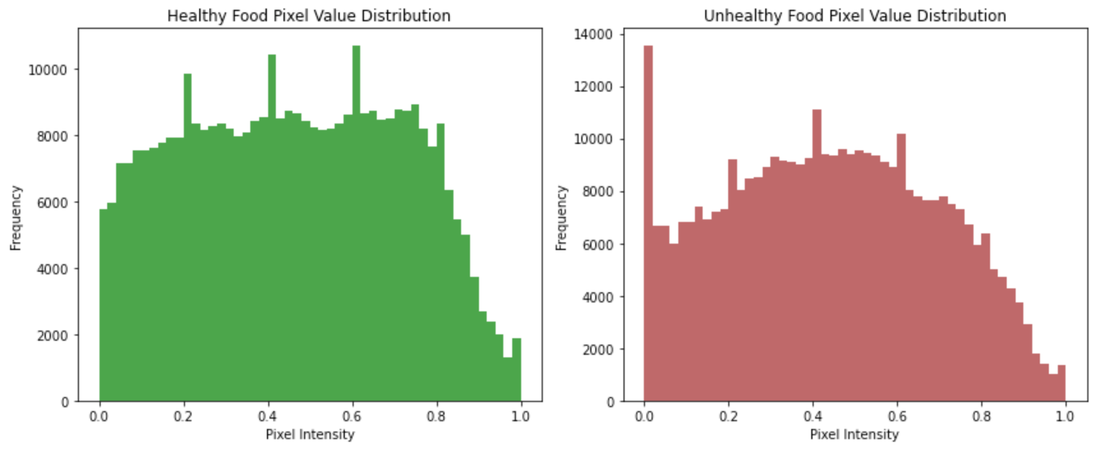
```
Overall Mean Pixel Value: 0.4513
Overall Standard Deviation of Pixel Values: 0.2535
```

[41]:
```
# Healthy class statistics
healthy_pixels = X_flat[y == 0]
mean_healthy = np.mean(healthy_pixels)
std_healthy = np.std(healthy_pixels)

# Unhealthy class statistics
unhealthy_pixels = X_flat[y == 1]
mean_unhealthy = np.mean(unhealthy_pixels)
std_unhealthy = np.std(unhealthy_pixels)

print(f"Healthy - Mean: {mean_healthy:.4f}, Std Dev: {std_healthy:.4f}")
print(f"Unhealthy - Mean: {mean_unhealthy:.4f}, Std Dev: {std_unhealthy:.4f}")
```

```
Healthy - Mean: 0.4616, Std Dev: 0.2553
Unhealthy - Mean: 0.4411, Std Dev: 0.2513
```

Differences Between Classes: Slight differences in mean pixel values indicate that unhealthy food images are, on average, slightly brighter or have different color distributions.

[42]:
```
plt.figure(figsize=(12, 5))

# Assuming you have separate datasets for healthy and unhealthy food
healthy_pixels_flat = healthy_pixels.reshape(-1)  # Flatten healthy food images
unhealthy_pixels_flat = unhealthy_pixels.reshape(-1)  # Flatten unhealthy food images

# Healthy pixels histogram
plt.subplot(1, 2, 1)
plt.hist(healthy_pixels_flat, bins=50, color='green', alpha=0.7)
plt.title('Healthy Food Pixel Value Distribution')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

# Unhealthy pixels histogram
plt.subplot(1, 2, 2)
plt.hist(unhealthy_pixels_flat, bins=50, color='brown', alpha=0.7)
plt.title('Unhealthy Food Pixel Value Distribution')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



- Healthy Food Histogram (green): The pixel values appear to be more evenly distributed across the intensity range, with a slight skew towards the middle. This suggests that healthy food images might contain a wide variety of colors and brightness levels, likely due to the natural variance in fruits, vegetables, and other healthy items.
- Unhealthy Food Histogram (brown): The distribution is more skewed towards lower intensities. This suggests that unhealthy food images might contain more darker pixels, which is often the case with processed, fried, or packaged food items.

While pixel intensity histograms provide a basic view, they don't capture enough detailed information about the structure and patterns in the images. Extracting RGB color histograms for each image can help capture color distribution differences in more detail, which can significantly improve the performance of the classification model.

Feature Engineering

[43]:
```python
def extract_rgb_histogram(image, bins=32, normalize=True):
    """
    Extracts RGB color histograms from an image.

    Parameters:
    - image (numpy array): The input image in RGB format.
    - bins (int): Number of bins for the histogram.
    - normalize (bool): Whether to normalize the histogram.

    Returns:
    - hist_features (numpy array): Concatenated histogram features.
    """
    # Compute the histogram for each color channel
    hist_r, _ = np.histogram(image[:, :, 0], bins=bins, range=(0, 1))
    hist_g, _ = np.histogram(image[:, :, 1], bins=bins, range=(0, 1))
    hist_b, _ = np.histogram(image[:, :, 2], bins=bins, range=(0, 1))

    # Concatenate the histograms into a single feature vector
    hist_features = np.concatenate((hist_r, hist_g, hist_b))

    if normalize:
        # Normalize the histogram
        hist_features = hist_features.astype('float32')
        hist_sum = np.sum(hist_features)
        if hist_sum != 0:
            hist_features /= hist_sum

    return hist_features
```

[44]:
```python
# Apply the histogram extraction to all images
def extract_features_from_dataset(X):
    features = []
    for img in X:
        # Since images are normalized to [0, 1], no need to divide by 255
        hist = extract_rgb_histogram(img, bins=32, normalize=True)
        features.append(hist)
    return np.array(features)
```

[45]:
```python
# Extract histogram features for training and testing sets
X_train_hist = extract_features_from_dataset(X_train.reshape(-1, image_size[1], image_size[0], 3))
X_test_hist = extract_features_from_dataset(X_test.reshape(-1, image_size[1], image_size[0], 3))


# Output the shapes of the resulting datasets
print("Histogram features shape (training):", X_train_hist.shape)
print("Histogram features shape (testing):", X_test_hist.shape)
```

```
Histogram features shape (training): (48, 96)
Histogram features shape (testing): (12, 96)
```

Model Selection

To classify the images as either healthy or unhealthy food, we have divided the dataset into training and testing sets, ensuring that our models can be properly evaluated for their generalization to unseen data. For the classification task, we will focus on three key models.

- First, we'll implement **Basic Logistic Regression** using the flattened pixel values as input features. This model will serve as a baseline, providing insight into how well a simple linear classifier can differentiate between healthy and unhealthy foods based solely on raw pixel data.
- Next, we'll enhance the feature set by applying **Logistic Regression with RGB features**, where we extract and use color histograms from the Red, Green, and Blue channels of each image. This approach allows us to capture color distribution differences between healthy and unhealthy foods, making the model more informed and potentially improving classification performance.
- Finally, we will implement **Support Vector Machines (SVM)** to classify the images. SVM is a powerful classifier that works well when we need to find a hyperplane that separates the classes in a high-dimensional space.

By testing these three models, we aim to compare their performances and select the one best suited for our image classification task.

1. Logistic Regression

Logistic Regression is a linear classifier that models the probability of a binary outcome using the **logistic function**. Below is a step-by-step breakdown of its mathematical formulation:

Mathematical Model:

- Logistic regression predicts the probability that a given input belongs to class (y = 1) (e.g., "unhealthy"):

$$P(y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}}$$

Where:

- $P(y = 1|X)$ is the probability of the image being "unhealthy" given features (X).
- β_0 is the intercept (bias term).
- $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients (weights) associated with each feature $x_1, x_2, \ldots, x_n$).

This function is known as the **sigmoid function** or **logistic function**, which maps the weighted sum of the input features to a value between 0 and 1, representing a probability (check figure below).

$$\sigma(z) = \frac{1}{1 + e^{-}z}$$

Training:

- During training, **maximum likelihood estimation (MLE)** is used to find the best parameters β by maximizing the likelihood of the observed data (healthy vs. unhealthy labels) given the feature values X.
- The **log-likelihood function** for logistic regression is:

$$L(\beta) = \sum_{i=1}^{m} \left[y^{(i)} \log(P(y = 1|X^{(i)})) + (1 - y^{(i)}) \log(1 - P(y = 1|X^{(i)})) \right]$$

Where:

- $y^{(i)}$ is the true label for the i-th image (1 for unhealthy, 0 for healthy).
- $P(y = 1|X^{(i)})$ is the predicted probability for the i-th image.
- To maximize the log-likelihood function, we use **gradient ascent**. The process involves:

  1. **Initialization**: Start with small random values for the parameters β.
  2. **Compute the Gradient**: Calculate the gradient of the log-likelihood function with respect to each parameter.
  3. **Update Parameters**: Adjust each parameter β_j using the update rule:

  $$\beta_j := \beta_j + \alpha \cdot \frac{\partial L}{\partial \beta_j}$$

  where α is the learning rate, controlling the size of the updates.

  4. **Iteration**: Repeat the gradient calculation and parameter updates for a specified number of iterations or until convergence.
- The objective is to find the parameters β that **maximize** this log-likelihood function, leading to the highest possible agreement between the predicted probabilities and the true labels. leading to the highest possible agreement between the predicted probabilities and the true labels.

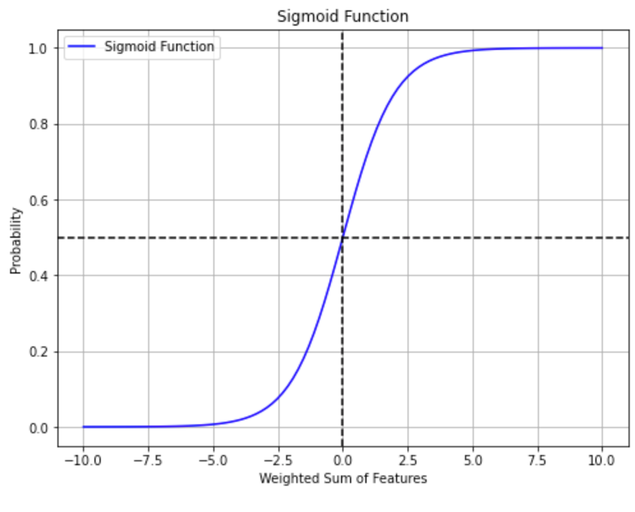
Prediction:

- After training, logistic regression predicts class 1 ("unhealthy") if the probability (P(y=1 | X) > 0.5), and class 0 ("healthy") otherwise.

Raw Pixels vs RGB

- The features could be **raw pixel values** (as in the basic logistic regression) or **RGB histograms** (in the feature-engineered model).
- Logistic regression assumes a **linear relationship** between the input features and the log-odds of the outcome. In the case of raw pixel values, the model tries to linearly separate healthy and unhealthy food images based on their pixel intensities. In the RGB histogram model, it separates the classes based on the distribution of color intensities.

[47]:
```
# A visualization of the sigmoid function to help think more intuitively
import numpy as np
import matplotlib.pyplot as plt

# Generate data for the sigmoid function
values = np.linspace(-10, 10, 100)
sigmoid = 1 / (1 + np.exp(-values))

# Plot the sigmoid function
plt.figure(figsize=(8, 6))
plt.plot(values, sigmoid, label='Sigmoid Function', color='blue')
plt.title('Sigmoid Function')
plt.xlabel('Weighted Sum of Features')
plt.ylabel('Probability')
plt.grid(True)
plt.axhline(0.5, color='black', linestyle='--')
plt.axvline(0, color='black', linestyle='--')
plt.legend()
plt.show()
```



2. SVM

Support Vector Machine (SVM) is a powerful classification algorithm that finds the optimal hyperplane to separate classes in a dataset. In the context of classifying **healthy vs. unhealthy food images**, SVM aims to separate the two categories of food based on the input features **RGB histograms**).

The key idea behind SVM is to find the **hyperplane** that best separates the two classes while maximizing the **margin** between the hyperplane and the closest points from either class. These closest points are called **support vectors**.

Mathematical Model

For a linearly separable dataset, the SVM algorithm seeks to find a **hyperplane** that separates the healthy (class 0) and unhealthy (class 1) images.

The equation of a hyperplane in (n)-dimensional space is:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

Where:

- $\mathbf{w}$ is the weight vector (normal to the hyperplane),
- $\mathbf{x}$ is the input feature vector (e.g., raw pixel values or RGB histograms of the image),
- b is the bias (intercept).

The objective is to find $\mathbf{w}$ and b such that the margin between the two classes is maximized.

Maximizing the Margin

The margin is defined as the distance between the hyperplane and the closest data points from each class (these are the support vectors). The distance between a point (\mathbf{x}) and the hyperplane is given by:

$$\frac{|\mathbf{w} \cdot \mathbf{x} + b|}{\|\mathbf{w}\|}$$

For linearly separable data, the SVM algorithm solves the following optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

Subject to the constraint that all data points are correctly classified:

$$y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \forall i$$

Where:

- $y^{(i)}$ is the label for the i-th image $+1$ for unhealthy, -1 for healthy),
- $\mathbf{x}^{(i)}$ is the feature vector for the i-th image.

The goal is to maximize the margin while ensuring that all data points are correctly classified.

Prediction

Once the optimal hyperplane is found, the classification of a new image $mathbf{x}$ is determined by the sign of the decision function:

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

- If $f(\mathbf{x}) > 0$, the image is classified as "unhealthy" (class 1),
- If $f(\mathbf{x}) \leq 0$, the image is classified as "healthy" (class 0).

SVM Algorithm (Pseudocode)

Below is a pseudocode representation of the SVM algorithm for binary classification (healthy vs. unhealthy food images).

```
Input: Training data (X_train, y_train), Regularization parameter (C), Kernel function (K)
Output: Weight vector (w), Bias (b)
```

```
1. Initialize weight vector w and bias b to small random values.
2. Solve the optimization problem to find w and b:
   a. Maximize the margin by minimizing:
      (1/2) * ||w||^2
   b. Subject to the constraints:
      y^(i) * (w . x^(i) + b) >= 1 - ξ^(i), for each training sample i.
   c. Include the regularization term (C) to penalize misclassified samples.
3. For a new input image x:
   a. Compute the decision function:
      f(x) = w . x + b
   b. Predict the label:
      - If f(x) > 0, classify as "unhealthy" (class 1).
      - If f(x) <= 0, classify as "healthy" (class 0).
4. Return the optimal w and b.
```

Training the model

Now is the time to train the models, we will also use cross validation for all the models to evaluate the generalization of performance of the models by dividing it into 5 equally sized folds. The model is trained on 4 of these folds and evaluated on the 1 remaining fold. This process is repeated 5 times, each time using a different fold as the validation set and the other 4 folds as the training set. calculates the mean accuracy across the 5 folds, which gives an overall performance estimate of the model. This average score provides a better understanding of how well the model is likely to perform on unseen data, as it reflects the model's ability to generalize across different subsets of the training data and ensuring that the data isnt overfitting.

Model 1: Basic Logistic Regression

[48]:
```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

# Initialize the Logistic Regression model
log_reg_basic = LogisticRegression(max_iter=1000)

# Perform 5-fold cross-validation on the training set to evaluate performance
cv_scores_basic = cross_val_score(log_reg_basic, X_train, y_train, cv=5)

# Print cross-validation scores and their mean
print("Mean cross-validation score (Basic Logistic Regression):", cv_scores_basic.mean())

# Train the Logistic Regression model on the entire training data (final training)
log_reg_basic.fit(X_train, y_train)

# Predict on the test set
y_pred_basic = log_reg_basic.predict(X_test)
```

```
Mean cross-validation score (Basic Logistic Regression): 0.48
```

Model 2: Logistic Regression with RGB

[49]:
```python
# Initialize the Logistic Regression model
log_reg_hist = LogisticRegression(max_iter=1000)

# Perform 5-fold cross-validation on the training set to evaluate performance
cv_scores_hist = cross_val_score(log_reg_hist, X_train_hist, y_train, cv=5)

# Print cross-validation scores and mean
print("Mean cross-validation score (Feature-Engineered Logistic Regression):", cv_scores_hist.mean())

# Train the Logistic Regression model on the entire training data (final training)
log_reg_hist.fit(X_train_hist, y_train)

# Predict on the test set
y_pred_hist = log_reg_hist.predict(X_test_hist)
```

```
Mean cross-validation score (Feature-Engineered Logistic Regression): 0.5777777777777778
```

Model 3: SVM with HyperParameter Tuning

For the SVM model with a lot of parameters, we will go one step ahead and also do hyperparameter tuning. With GridSearchCV we systematically will evaluate all possible combinations of these hyperparameters through 5-fold cross-validation, identifying the combination that yielded the highest cross-validation accuracy. This meticulous search not only optimizes the model's ability to generalize to unseen data but also mitigates the risk of overfitting by selecting parameters, like class weight, kernel etc that cab help balance bias and variance effectively.

[50]:
```python
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC


# Scale the data (SVM is sensitive to feature scales)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_hist)
X_test_scaled = scaler.transform(X_test_hist)

# Train a basic Linear SVM model (for comparison)
linear_svm = SVC(kernel='linear', C=0.1, class_weight='balanced')
linear_svm.fit(X_train_scaled, y_train)

#  Predict on the test set and evaluate the basic linear SVM
y_pred_linear = linear_svm.predict(X_test_scaled)

#  Hyperparameter tuning for SVM using GridSearchCV
# Define the parameter grid for GridSearch
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100, 1000],  # Regularization parameter
    'kernel': ['linear', 'rbf'],          # Linear and RBF kernels
    'gamma': ['scale', 'auto']            # For RBF kernel (only applicable to non-linear)
}

# Initialize GridSearchCV with SVM and the parameter grid
grid_search = GridSearchCV(SVC(), param_grid, cv=5)

# Fit the GridSearchCV on the training data
grid_search.fit(X_train_scaled, y_train)

# Print the best parameters and the best cross-validation score
print("Best Parameters (C, kernel, gamma):", grid_search.best_params_)
print("Best Cross-Validation Accuracy:", grid_search.best_score_)

#  Use the best model found by GridSearchCV to predict on the test set
best_svm = grid_search.best_estimator_
y_pred_best = best_svm.predict(X_test_scaled)
```

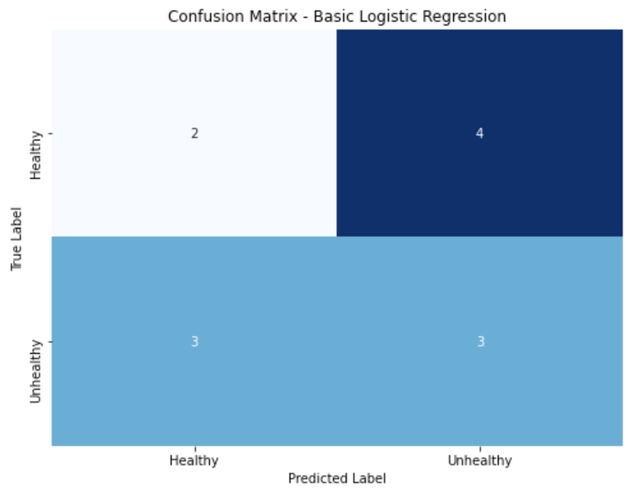
```
Best Parameters (C, kernel, gamma): {'C': 1, 'gamma': 'scale', 'kernel': 'linear'}
Best Cross-Validation Accuracy: 0.711111111111111
```

Predictions and Performance Metrics

[51]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Visualizing Performance Metrics - Confusion Matrices

def plot_confusion_matrix(cm, title, labels):
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=labels, yticklabels=labels)
    plt.title(title)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

# For Basic Logistic Regression
print("\033[1m\nPerformance Metrics for Logistic Regression (Raw Pixel Data):\033[0m")
accuracy_basic = accuracy_score(y_test, y_pred_basic)
print(f"Accuracy: {accuracy_basic}")
print("Classification Report:")
print(classification_report(y_test, y_pred_basic))
print("Mean cross-validation score (Basic Logistic Regression):", cv_scores_basic.mean())
cm_basic = confusion_matrix(y_test, y_pred_basic)
plot_confusion_matrix(cm_basic, 'Confusion Matrix - Basic Logistic Regression', ['Healthy', 'Unhealthy'])

# For Logistic Regression with RGB Histogram Features
print("\033[1m\nPerformance Metrics for Logistic Regression (RGB Histogram Features):\033[0m")
accuracy_hist = accuracy_score(y_test, y_pred_hist)
print(f"Accuracy: {accuracy_hist}")
print("Classification Report:")
print(classification_report(y_test, y_pred_hist))
print("Mean cross-validation score (Feature-Engineered Logistic Regression):", cv_scores_hist.mean())
cm_hist = confusion_matrix(y_test, y_pred_hist)
plot_confusion_matrix(cm_hist, 'Confusion Matrix - Logistic Regression (RGB Histogram Features)', ['Healthy', 'Unhealthy'])

# For SVM with Best Parameters
print("\033[1m\nPerformance Metrics for SVM ( best parameters):\033[0m")
accuracy_svm = accuracy_score(y_test, y_pred_best)
print(f"Accuracy: {accuracy_svm}")
print("Classification Report:")
print(classification_report(y_test, y_pred_best))
print("Best Cross-Validation Accuracy:", grid_search.best_score_)
cm_svm = confusion_matrix(y_test, y_pred_best)
plot_confusion_matrix(cm_svm, 'Confusion Matrix -  SVM Model', ['Healthy', 'Unhealthy'])

plt.tight_layout()
plt.show()
```

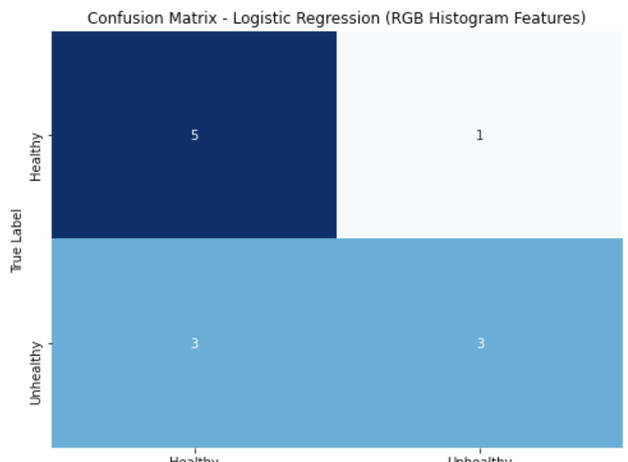
Performance Metrics for Logistic Regression (Raw Pixel Data):

```
Accuracy: 0.4166666666666667
Classification Report:
              precision    recall  f1-score   support

           0       0.40      0.33      0.36         6
           1       0.43      0.50      0.46         6

    accuracy                           0.42        12
   macro avg       0.41      0.42      0.41        12
weighted avg       0.41      0.42      0.41        12

Mean cross-validation score (Basic Logistic Regression): 0.48
```



Performance Metrics for Logistic Regression (RGB Histogram Features):

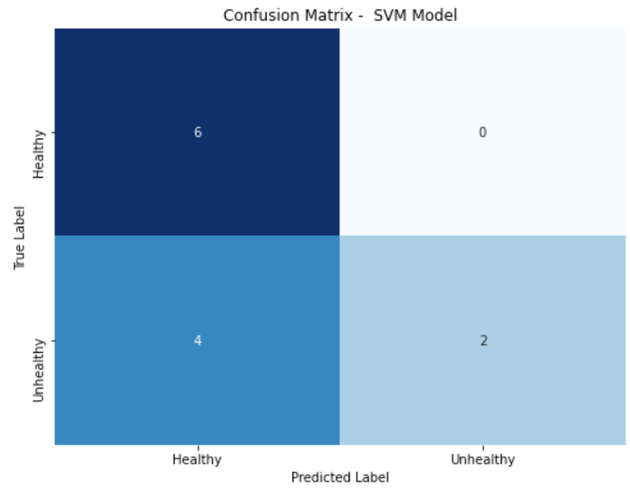
```
Accuracy: 0.6666666666666666
Classification Report:
              precision    recall  f1-score   support

           0       0.62      0.83      0.71         6
           1       0.75      0.50      0.60         6

    accuracy                           0.67        12
   macro avg       0.69      0.67      0.66        12
weighted avg       0.69      0.67      0.66        12

Mean cross-validation score (Feature-Engineered Logistic Regression): 0.5777777777777778
```



Performance Metrics for SVM (best parameters):

```
Accuracy: 0.6666666666666666
Classification Report:
              precision    recall  f1-score   support

           0       0.60      1.00      0.75         6
           1       1.00      0.33      0.50         6

    accuracy                           0.67        12
   macro avg       0.80      0.67      0.62        12
weighted avg       0.80      0.67      0.62        12
```

```
Best Cross-Validation Accuracy: 0.711111111111111
```




```
<Figure size 432x288 with 0 Axes>
```

- Basic Logistic Regression (Raw Pixel Values) The Basic Logistic Regression model yielded an accuracy of 41.67% with a cross-validation score of 48%. The confusion matrix shows that the model struggled to separate healthy and unhealthy images, classifying many healthy foods incorrectly as unhealthy. This is because raw pixel data does not provide meaningful abstractions for the model to differentiate between the two classes. Logistic Regression, being a linear model, struggles with high-dimensional, noisy data like pixel values, especially without any feature engineering. In this case, the model's inability to effectively process raw image data led to frequent misclassifications, highlighting the limitations of using pixel values without additional context.
- Logistic Regression with RGB Histogram Features The Logistic Regression model with RGB Histogram Features performed significantly better, achieving an accuracy of 66.67% with a cross-validation score of 57.78%. RGB histograms reduced the data's complexity by capturing the color distribution in the images, which is a critical distinguishing factor between healthy (label 0) and unhealthy (label 1) foods. As seen in the confusion matrix, the model excelled at classifying healthy foods (83% recall) but had a harder time with unhealthy foods. This suggests that while color features from RGB histograms helped the model, color alone might not fully capture the characteristics of unhealthy foods, which may rely on texture or other visual attributes.
- SVM with RGB Histogram Features and Hyperparameter Tuning The SVM model using RGB histograms, and further optimized through hyperparameter tuning, matched Logistic Regression's accuracy at 66.67% but had a higher cross-validation score of 71.11%, indicating better generalization. However, the confusion matrix reveals a bias toward classifying healthy foods. The SVM model correctly classified all healthy foods (100% recall) but struggled with unhealthy foods (only 33% recall). This imbalance likely arose because SVM is sensitive to the structure of the data, and with a small dataset, it overfit to patterns in the unhealthy class. While the hyperparameters helped optimize the decision boundary, the model was still prone to favoring the healthy class over healthy foods. SVM typically shines with larger datasets, where it can fully explore the margins and create a strong hyperplane that separates classes and given the limited dataset it did not provide enough variation Color is certainly a distinguishing factor between healthy and unhealthy food, but color alone may not capture other important factors, such as texture, shape, or even subtle visual cues that differentiate processed food from fresh produce.Without richer, more complex features (like texture, edges, or deep learning-based representations), the SVM may not be able to fully leverage its power, leading to it overfitting to the simple patterns found

Conclusion and Future Recommendations

Throughout this project, We found that Logistic Regression with RGB Histogram Features struck the best balance between performance, generalization, and interpretability for classifying images as healthy or unhealthy. The model consistently performed well for both classes, maintaining a solid balance between precision and recall, which was crucial for ensuring that neither healthy nor unhealthy images were overlooked. I appreciated how Logistic Regression provided clear and understandable insights into how different color distributions influenced the classifications, making the decision-making process transparent and reliable. On the other hand, while Support Vector Machine (SVM) showed slightly better generalization in cross-validation, I noticed it tended to favor healthy images more, which made it less dependable for our dataset where accurate classification of both categories was equally important but SVM still stands as a strong model not perfect for this small dataset. One key thing was adding features improved accuracy significantly, additionally, relying solely on RGB histograms meant that some spatial and textural information in the images might have been missed, potentially limiting the model's ability to fully distinguish between classes.

To overcome these limitations, I believe that enhancing our feature set with texture and shape descriptors could provide a more comprehensive understanding of the images and also improve SVM's accuracy because it works better on complex features. Moreover, expanding and diversifying our dataset through data augmentation techniques would likely improve both model's robustness and reduce the risk of overfitting.

I also looked up different models for image classification and integrating Convolutional Neural Networks (CNNs) into our approach could make it robust. CNNs are powerful for image analysis as they can automatically learn and extract complex features from the data, potentially offering higher accuracy and better generalization without the need for extensive manual feature engineering. By combining these strategies—advanced feature engineering, dataset expansion, and exploring deep learning models like CNNs—we can significantly enhance our classification system, making it more accurate and reliable for distinguishing between healthy and unhealthy images.

References

Verma, H. (2020, May 30). The math behind logistic regression. Medium. https://medium.com/analytics-vidhya/the-math-behind-logistic-regression-c2f04ca27bca

Nitesh, A. (2020, July 12). Math behind support vector machine (SVM). Medium. https://ankitnitjsr13.medium.com/math-behind-support-vector-machine-svm-5e7376d0ee4d

Cote, D. (2019, November 21). Demonstrating the power of feature engineering: Part I. Medium. https://medium.com/@dave.cote.msc/demonstrating-the-power-of-feature-engineering-part-i-7d5c0222d249

Benner, J. (2020, August 6). Cross-Validation and Hyperparameter Tuning: How to Optimise your Machine Learning Model. Towards Data Science. https://towardsdatascience.com/cross-validation-and-hyperparameter-tuning-how-to-optimise-your-machine-learning-model13f005af9d7d