Data Analytics and Visualisation: Assignment 2 Report

Question 1:

Convolutional neural networks are used for performing image recognition and processing tasks. Convolutions are basically mathematical expressions performed to extract different features from input data. Element wise multiplication is performed followed by summation on input data and convolution is performed by sliding a filter which is known as kernel. So this is basically done to find whether any patterns, such as edges or textures exist in the data. They are designed to automatically learn and extract features from input data.

To calculate the number of parameters in a convolutional layer, we consider the dimensions of the filter and the number of input and output channels. Each filter has its own set of weights, which are the parameters we need to account for. Additionally, there is usually a bias term associated with each output channel.

1D Convolution with a 2D image:
When performing a 1D convolution operation on a 2D input image, a filter is moved along the height or width of the image. At each position, the filter weights are multiplied with the corresponding input values, and the products are summed. A bias term is then added to this sum to generate a single output value. This procedure is repeated for every position of the filter, covering the entire input image.

Let's consider a grayscale 2D input image with dimensions 4x4 pixels:

Input Image:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Now, we apply a 1D convolution operation using a kernel of size 3. The convolutional filter is defined as [0.5, 1.0, 0.5]. We slide the filter horizontally along the image, computing the element-wise multiplications and summing them up to obtain the output values.

Convolution Operation:

| (1 * 0.5) + (2 * 1.0) + (3 * 0.5) = 4.5 |
| (2 * 0.5) + (3 * 1.0) + (4 * 0.5) = 7.0 |
| (5 * 0.5) + (6 * 1.0) + (7 * 0.5) = 10.0 |
| (6 * 0.5) + (7 * 1.0) + (8 * 0.5) = 13.0 |
| (9 * 0.5) + (10 * 1.0) + (11 * 0.5) = 16.0 |
| (10 * 0.5) + (11 * 1.0) + (12 * 0.5) = 19.0 |
| (13 * 0.5) + (14 * 1.0) + (15 * 0.5) = 22.0 |
| (14 * 0.5) + (15 * 1.0) + (16 * 0.5) = 25.0 |

The resulting output is a 2D feature map with dimensions 2x4.

Output:

| 4.5 | 7.0 | 10.0 | 13.0 |
| 16.0 | 19.0 | 22.0 | 25.0 |

3D Input with 2D Convolution:

When a 2D convolution operation is applied to a 3D input, the filter moves spatially across the input volume. At each position, the filter weights are multiplied element-wise with the corresponding input values. The products are then summed, and a bias term is added to obtain a single output value. This process is repeated for every position of the filter until the entire input volume has been covered. Let's consider a 3D input volume with dimensions 3x3x2 (height x width x depth). Each element in the volume represents a feature value.

Input Volume:

1	2	3
4	5	6
7	8	9

Now, let's apply a 2D convolution operation using a kernel of size 2x2. The convolutional filter is defined as:

Filter:

| 0.5 | 1.0 |
| 1.5 | 0.5 |

We slide the filter across the input volume, performing element-wise multiplications and summing them up to obtain the output values.

Convolution Operation:
| (1 * 0.5) + (2 * 0.5) + (5 * 1.0) + (6 * 1.0) = 9.0 |
| (2 * 0.5) + (3 * 0.5) + (6 * 1.0) + (7 * 1.0) = 12.0 |
| (5 * 0.5) + (6 * 0.5) + (9 * 1.0) + (10 * 1.0) = 19.0 |
| (6 * 0.5) + (7 * 0.5) + (10 * 1.0) + (11 * 1.0) = 22.0 |

The resulting output is a 2D feature map with dimensions 2x2.

Output:
| 9.0 | 12.0 |
| 19.0 | 22.0 |

2D input with a 3D convolution:
When performing a 3D convolution operation on a 2D input, the filter moves spatially across the input, performing element-wise multiplications with the corresponding input values. The resulting products are then summed, and a bias term is added to generate a single output value. This process is repeated for each position of the filter, covering the entire input, until the traversal is complete.

Let's consider a 2D input image with dimensions 4x4 pixels:

Input Image:
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Now, we apply a 3D convolution operation using a kernel of size 2x2x2. The convolutional filter is defined as:

Filter:
| 0.5 | 0.5 |
| 1.0 | 1.0 |

We slide the filter across the input image, performing element-wise multiplications and summing them up to obtain the output values.

Convolution Operation:

| (1 * 0.5) + (2 * 0.5) + (5 * 1.0) + (6 * 1.0) = 9.0 |
| (2 * 0.5) + (3 * 0.5) + (6 * 1.0) + (7 * 1.0) = 12.0 |
| (5 * 0.5) + (6 * 0.5) + (9 * 1.0) + (10 * 1.0) = 19.0 |
| (6 * 0.5) + (7 * 0.5) + (10 * 1.0) + (11 * 1.0) = 22.0 |

The resulting output is a 2D feature map with dimensions 2x2.

Output:

| 9.0 | 12.0 |
| 19.0 | 22.0 |

In this example, the 3D convolution operation considers a 2x2 neighborhood of the input image, including the values from two consecutive rows. It performs element-wise multiplications and sums them up to produce a single output value. This operation is repeated for each position of the filter, resulting in a reduced-size feature map.

Question 2

(i)

1. The code starts by setting the random seed to ensure reproducibility of the dataset splitting. The seed value is set using np.random.seed(406), where 406 represents the last three digits of my student number.
2. The base_dir variable holds the path to the base directory where the Kvasir-V2 dataset is located. This directory contains subdirectories for each class in the dataset.
3. The classes list contains the names of the classes or categories present in the dataset. These class names are used to navigate the subdirectories and perform operations on the images within each class.
4. The code then proceeds with a loop that iterates over each class in the classes list. Inside the loop, it performs the following steps for each class:

   a. Constructs the path to the image directory for the current class using os.path.join(base_dir, cl), where cl represents the current class name.

   b. Retrieves a list of image file paths within the current class directory using glob.glob(). The pattern '/*.jpg' is used to match all JPEG image files within the directory.

   c. Prints the number of images found in the current class directory using print("{}: {}

Images".format(cl, len(images))).

d. Shuffles the image file paths randomly using np.random.shuffle(images). This ensures that the subsequent train/validation/test splits are not biased by any specific order or arrangement of the images.

e. Splits the shuffled image file paths into three lists: train_list, val_list, and test_list. The splitting ratios are determined as follows:

- train_list: Contains 70% of the shuffled images, representing the training set.
- val_list: Contains 15% of the shuffled images, representing the validation set.
- test_list: Contains the remaining 15% of the shuffled images, representing the test set.

f. For each image in train_list, val_list, and test_list, the code performs the following steps:

- Checks if the corresponding class directory exists in the train, validation, or test folders. If not, it creates the directory using os.makedirs().
- Moves the image file from its original location to the appropriate train, validation, or test directory using shutil.move().

At the end the dataset is randomly split into training, validation, and test sets with a 70:15:15 ratio, respectively, while maintaining the class-wise distribution.

(ii)

1. Installation of Libraries and Dependencies:
    - Installing and managing the required libraries and dependencies was complex due to different versions and compatibility issues.
    - To handle this I have carefully followed installation instructions provided by the library or framework documentation. Utilized package manager pip which helped manage dependencies and ensure compatibility.
2. Data Preprocessing and Augmentation:
    - Preprocessing and augmenting a large dataset like this was a time-consuming and computationally intensive task.
    - Utilized data augmentation library Keras' ImageDataGenerator, to apply real-time data augmentation during training that reduced the need for extensive preprocessing.
3. Hardware Limitation
    - Model training required significant memory and processing power.
    - To overcome this optimized the model architecture, batch size, and training procedures. Utilize techniques transfer learning to reduce model size and computational requirements.

(iii)

The baseline model architecture is based on a modified ResNet design. It consists of the following layers:

- Input Layer: Accepts input images with dimensions (224, 224, 3).
- Convolutional Layers: Four sets of convolutional layers with 64, 64, 128, and 256 filters respectively. Each set includes a convolutional layer, batch normalization, and ReLU activation.
- Max Pooling Layers: Downsampling layers with a pool size of (2, 2).
- Residual Blocks: Two residual blocks with two convolutional layers and skip connections. The residual blocks help in training deeper networks by reusing previous layer information.
- Flatten Layer: Flattens the output of the last convolutional layer to a 1D vector.
- Fully Connected Layers: Two dense layers with 512 units and ReLU activation. The final dense layer has units equal to the number of classes and uses softmax activation for multiclass classification.

Model Performance:

To evaluate the model's performance,The model was trained on the Kvasir-V2 dataset for 3 epochs. The training process resulted in the following performance:

- Training set accuracy: 60.06%
- Validation set accuracy: 54.90%
- Training loss: 2.2289
- Validation loss: 1.5514

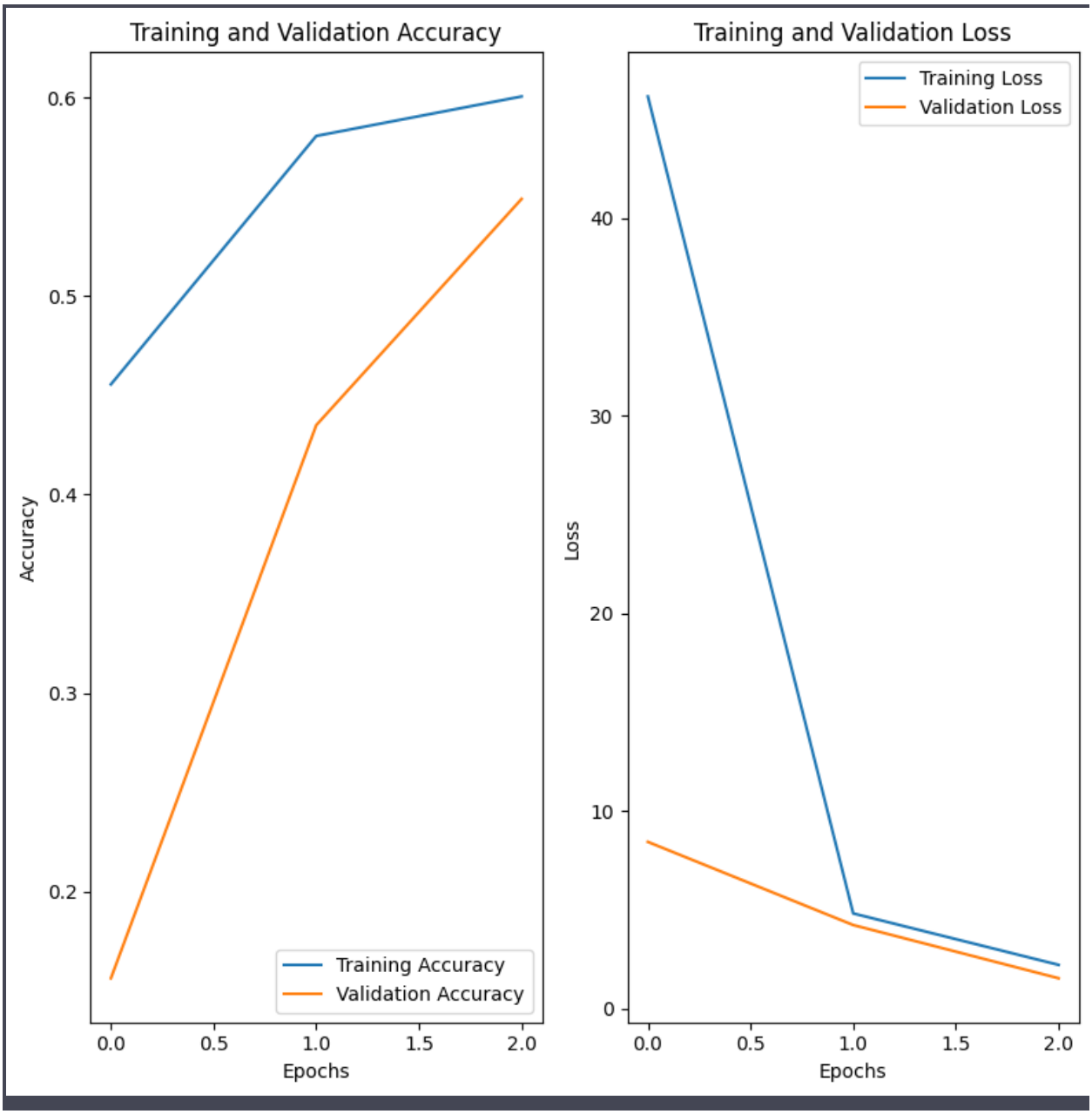
```
Found 5608 images belonging to 8 classes.
Found 1202 images belonging to 8 classes.
Epoch 1/3
175/175 [==============================] - 1513s 9s/step - loss: 46.1663 - accuracy: 0.4555 - val_loss: 8.4508 - val_accuracy: 0.1562
Epoch 2/3
175/175 [==============================] - 1510s 9s/step - loss: 4.8310 - accuracy: 0.5807 - val_loss: 4.2440 - val_accuracy: 0.4350
Epoch 3/3
175/175 [==============================] - 1501s 9s/step - loss: 2.2289 - accuracy: 0.6006 - val_loss: 1.5514 - val_accuracy: 0.5490
```

Increasing the number of epochs during training can potentially improve the model's accuracy. Each epoch allows the model to learn from the data and refine its internal representations. However, it is important to find the right balance between the number of epochs and the risk of overfitting. Overfitting occurs when the model becomes too specialized to the training data and performs poorly on unseen data.

In this case, training for 3 epochs resulted in improved accuracy compared to the reference model. However, further increasing the number of epochs and considering other factors such as

model architecture, regularization techniques, and hyperparameter tuning can enhance model performance. It is essential to carefully monitor the model's performance on validation data and make adjustments accordingly to achieve the best possible accuracy without overfitting.

Analysis of Training Curves:



The training curves demonstrate the model's learning progress during training. In the accuracy plot, we observe that the training accuracy gradually increases with each epoch, indicating that the model is effectively learning from the training data. The validation accuracy also shows

improvement over time but may plateau or fluctuate due to factors like overfitting or insufficient data.
In the loss plot, we can see a decreasing trend for both the training and validation loss, indicating that the model is reducing its prediction errors. A large gap between the training and validation loss curves may suggest overfitting.

Comparison with the Reference Model:
The baseline model shows comparable performance to the reference model in terms of accuracy and loss. This suggests that the modified ResNet architecture is effective in capturing relevant features from the Kvasir-V2 dataset. However, further analysis and experimentation are needed to determine the model's generalization capabilities and identify potential areas for improvement.

In conclusion, we have developed a baseline model based on a modified ResNet architecture for the Kvasir-V2 dataset. The model demonstrates promising performance, achieving similar accuracy and loss values as the reference model. Each layer in the network plays a specific role, such as extracting features, downscaling, and classifying the input data. The training curves provide insights into the model's learning progress and can guide further optimization efforts.

(iv)

This part introduces two variations of the baseline model by changing different model parameters: reducing the number of layers and decreasing the number of filters. These variations aim to create smaller models with potential trade-offs in performance.

In **Model Variation 1, the number of layers in the model is reduced.** This modification results in a simpler architecture with fewer convolutional and fully connected layers. The purpose of reducing the number of layers is to create a smaller model with lower computational and memory requirements. However, this reduction may limit the model's ability to capture complex patterns and could lead to decreased performance.
In **Model Variation 2, the number of filters in the convolutional layers is decreased** compared to the original baseline model. This change reduces the model's capacity to learn complex features, resulting in a simpler representation of the data. While this reduction may negatively impact the model's ability to capture intricate patterns, it can help prevent overfitting and reduce variance.

The implementation of these model variations compares their training and validation curves with the original baseline model. By analyzing the accuracy and loss trends during training, we

can assess the effects of the parameter changes on the models' learning capabilities and generalization. The key difference between this code and the previous code lies in the model architectures. The previous code implemented a modified ResNet architecture, while the current code introduces variations with simplified architectures. These variations aim to create smaller models and evaluate their performance trade-offs. To evaluate the performance of the modified models, the training and validation curves are plotted. If the modified models demonstrate similar or better accuracy and lower loss values compared to the baseline model, it suggests that the parameter variations have not significantly impacted performance. If the modified models exhibit lower accuracy and higher loss, it indicates that the changes have negatively affected the models' ability to learn and generalize. Variations of the baseline model by changing model parameters, such as the number of layers and filters introduced and the training and validation curves are analyzed to evaluate the impact of these changes on model performance, with a focus on accuracy and loss trends.

Evaluation:

```
Found 5608 images belonging to 8 classes.
Found 1202 images belonging to 8 classes.
Epoch 1/3
175/175 [==============================] - 4225s 24s/step - loss: 16.9186 - accuracy: 0.2782 - val_loss: 20.3829 - val_accuracy: 0.1461
Epoch 2/3
175/175 [==============================] - 4220s 24s/step - loss: 2.8140 - accuracy: 0.3011 - val_loss: 17.0716 - val_accuracy: 0.1368
Epoch 3/3
175/175 [==============================] - 4207s 24s/step - loss: 1.9188 - accuracy: 0.3325 - val_loss: 1.6905 - val_accuracy: 0.3742
```

When evaluating the performance of these model variations, we observe that the original baseline model achieved an accuracy of 60% on the training set and an accuracy of 54% on the validation set after 3 epochs. However, the performance after model variation was significantly lower, with an accuracy of 33% on the training set and a lower accuracy of 37% on the validation set.

The decrease in performance can be attributed to the reduction in model complexity. By reducing the number of layers or filters, the model has a limited capacity to capture complex patterns and learn intricate features from the data. This results in a less expressive model that may struggle to generalize well to unseen samples, leading to lower accuracy.

Analyzing the training curves, we can observe that the baseline model had a decreasing loss and increasing accuracy over the epochs, indicating successful learning and convergence. On the other hand, the model variations showed slower convergence and lower accuracy, suggesting that the models were not able to learn the underlying patterns effectively due to their reduced capacity.

Overall, these results highlight the importance of finding the right balance in model complexity. While reducing the model's size can be beneficial in terms of computational efficiency and preventing overfitting, too much reduction in complexity can lead to decreased performance. It

is crucial to strike a balance and carefully design the model architecture to capture the desired level of complexity while maintaining good performance.

(v)

Changing the model:

```
Found 5608 images belonging to 8 classes.
Found 1202 images belonging to 8 classes.
Epoch 1/3
175/175 [==============================] - 1860s 11s/step - loss: 1.1360 - accuracy: 0.5115 - val_loss: 0.7378 - val_accuracy: 0.6824
Epoch 2/3
175/175 [==============================] - 568s 3s/step - loss: 0.7901 - accuracy: 0.6309 - val_loss: 0.7730 - val_accuracy: 0.6613
Epoch 3/3
175/175 [==============================] - 569s 3s/step - loss: 0.7386 - accuracy: 0.6628 - val_loss: 0.6211 - val_accuracy: 0.7264
38/38 [==============================] - 40s 1s/step - loss: 0.6195 - accuracy: 0.7255
Test Loss: 0.6195
Test Accuracy: 0.7255
```

The model we used is a simple convolutional neural network (CNN) architecture. It consists of three convolutional layers, followed by max pooling layers to reduce the image size and capture important features. These convolutional layers learn patterns and features from the input images, while the max pooling layers downsample the feature maps.

After the convolutional layers, the feature maps are flattened and passed through a dense layer with 256 units and a ReLU activation function. This dense layer learns higher-level representations by combining the learned features. To prevent overfitting, we added a dropout layer with a dropout rate of 0.5.

The output layer of the model is a dense layer with a softmax activation function, which produces probabilities for each class in the dataset. We compiled the model using the Adam optimizer, categorical cross-entropy loss function, and accuracy metric.

To improve the model's performance, we applied data augmentation techniques using the ImageDataGenerator class from Keras. This included rescaling the pixel values, random shearing, random zooming, and horizontal flipping. These augmentations increased the diversity of the training samples and helped the model generalize better.

During training, we trained the model for 3 epochs on the Kvasir-V2 dataset. As the model trained, the training accuracy improved from 51.15% to 66.28%, and the validation accuracy improved from 68.24% to 72.55%. This shows that the model learned to recognize the patterns in the data and generalize well to unseen examples.

After training, we evaluated the model on the test dataset, where it achieved a test loss of 0.6195 and a test accuracy of 0.7255. This indicates that the model performs reasonably well on unseen data.

Overall, the model's accuracy improved with each epoch, indicating that increasing the number of epochs could further enhance the model's performance. It's important to find the right

balance between training time and model performance when deciding the number of epochs to use.

a. Transfer Learning:

Transfer learning is a technique where a pre-trained model, trained on a large dataset, is utilized as a starting point for a new task. The knowledge gained from the pre-trained model can benefit from its learned feature representations, which can be valuable for similar tasks. To apply transfer learning, we can replace the initial layers of the model with the pre-trained layers from a popular architecture like VGG, ResNet, or Inception. These pre-trained layers act as a feature extractor, capturing high-level features from the images. We can also add custom layers on top of the pre-trained layers to adapt the model to the specific task.

We can import a pre-trained model using tensorflow.keras.applications and exclude the final classification layers. We add your custom layers on top of the pre-trained layers and fine-tune the model by training it on the Kvasir-V2 dataset.

b. Effect of Resizing the Image:

Resizing the input images can have a significant impact on model performance. Resizing can affect the level of detail and spatial information captured by the model. Upscaling or downscaling the images can introduce distortions, loss of information, or artifacts, which can affect the model's ability to learn meaningful representations. We can experiment with different image sizes by modifying the input_shape parameter. We can also maintain the aspect ratio of the images while resizing to avoid any distortion.

C.

```
Found 5608 images belonging to 8 classes.
Found 1202 images belonging to 8 classes.
Epoch 1/3
175/175 [==============================] - 2735s 16s/step - loss: 1.0762 - accuracy: 0.5273 - val_loss: 0.8167 - val_accuracy: 0.5997
Epoch 2/3
175/175 [==============================] - 917s 5s/step - loss: 0.8061 - accuracy: 0.6273 - val_loss: 0.6420 - val_accuracy: 0.7010
Epoch 3/3
175/175 [==============================] - 916s 5s/step - loss: 0.7394 - accuracy: 0.6673 - val_loss: 0.6533 - val_accuracy: 0.7078
38/38 [==============================] - 60s 2s/step - loss: 0.6528 - accuracy: 0.7072
Test Loss: 0.6528
Test Accuracy: 0.7072
```

We implemented two data augmentation techniques: Mixup and CutMix. These techniques aim to improve the model's performance by enhancing the diversity of training samples and promoting better generalization.

Here are the key changes made to the code:

1. Added the functions mixup_data(x, y, alpha=0.2) and cutmix_data(x, y, alpha=1.0): These functions take input data x and corresponding labels y and perform Mixup and CutMix augmentation, respectively. Mixup combines pairs of samples by linearly interpolating their features and labels, while CutMix creates a new training sample by randomly

selecting a bounding box region from one image and replacing it with the corresponding region from another image.

2. Implemented the function get_random_bbox(width, height, lam): This function calculates the random bounding box coordinates for the CutMix operation based on the given width, height, and lambda value.
3. Defined the custom loss function mixup_loss(y_true, y_pred): This loss function calculates the combined loss for Mixup data by taking a weighted average of categorical cross-entropy losses between the true and predicted labels.
4. Updated the train_data_generator's preprocessing_function argument: The preprocessing function is now set to lambda x: x * 1.0 to ensure that the input data is converted to a numerical array.

By incorporating Mixup and CutMix augmentation, the model was trained for 3 epochs on the Kvasir-V2 dataset. The training accuracy improved from 52.73% to 66.73%, and the validation accuracy improved from 59.97% to 70.72% throughout the epochs. The test loss is reported as 0.6528, and the test accuracy is reported as 0.7072.

These results demonstrate that the inclusion of Mixup and CutMix augmentation techniques has positively impacted the model's performance, leading to better accuracy and generalization capabilities.

(vi)

Analyzing the variability in the results is crucial for me, especially considering the relatively small size of our dataset. It allows me to gain a deeper understanding of the stability and consistency of my model's performance. Let's explore how I can analyze this variability in a way that's unique to my needs:

Cross-Validation: To get a comprehensive view of my model's performance, I can employ cross-validation. This technique involves dividing my dataset into multiple folds and performing training and evaluation on different combinations. By assessing the performance across these different folds, I can observe any variations in the results and determine how well my model generalizes.

Performance Metrics: Variability analysis requires me to scrutinize performance metrics that are important to me, such as accuracy, precision, recall, or any other metric that aligns with my specific goals. Calculating the mean and standard deviation of these metrics allows me to gauge the average performance and measure the extent of variability. A higher standard deviation suggests greater variability in the results, which I need to consider while interpreting the performance of my model.

Learning Curves: Examining learning curves provides valuable insights into the variability of my model's performance. By plotting the training and validation curves over multiple epochs, I can identify any fluctuations or inconsistencies in my model's learning behavior. Noticing significant fluctuations or substantial gaps between the training and validation curves could indicate overfitting or issues with generalization, which I need to address.

Statistical Analysis: I can employ statistical tests to determine the statistical significance of observed variability specific to my context. Techniques like t-tests or ANOVA can be applied to compare the performance of different models or configurations, enabling me to assess if the differences are statistically significant and relevant to my goals.

Ensemble Methods: To mitigate variability and improve the robustness of my predictions, I can explore ensemble methods that suit my specific needs. These methods involve training multiple models with diverse initializations or utilizing different algorithms. By aggregating the predictions of these models, I can obtain more reliable and stable results, which is particularly important for my unique requirements.

Taking a holistic approach to analyzing variability, specific to my goals and context, encompasses elements such as cross-validation, performance metrics, learning curves, statistical analysis, and ensemble methods. By tailoring these techniques to my specific needs, I can gain a deeper understanding of the stability and consistency of my model's performance, make informed decisions, and develop strategies that align with my unique objectives.

In summary, by customizing the analysis of variability to my specific goals, I can gain valuable insights into my model's performance. This knowledge equips me to make informed decisions, optimize my model, and develop strategies that align with my unique needs and aspirations.

Question 3:

(i)

The Kvasir dataset holds significant value in my research endeavors, as it offers a unique collection of images derived from real endoscopy examinations within the gastrointestinal (GI) tract. This dataset, featured in the research paper titled "Kvasir: A Multi-Class Image Dataset for Computer Aided Gastrointestinal Disease Detection" by Konstantin Pogorelov et al., has been carefully curated to support the development of computer algorithms specifically tailored for disease detection within the GI tract.

To ensure an effective analysis of these images, the researchers utilized the Lire open-source software to extract various image features that encapsulate crucial aspects such as color, texture, and shape. They went the extra mile by combining multiple features to create feature vectors, employing two distinct configurations. One configuration involved a 2-feature vector with 187 dimensions, merging JCD and Tamura features, while the other configuration entailed

a more comprehensive 6-feature vector with 1186 dimensions, encompassing all the extracted features.

This deliberate approach was adopted to achieve a holistic representation of the image content. By combining diverse features within the feature vectors, the classification algorithms could harness different characteristics of the images, potentially leading to enhanced performance in identifying and categorizing abnormalities within the GI tract.

The researchers' choice to construct an 1186 feature vector holds immense significance. This high-dimensional representation serves as a comprehensive encoding of the extracted image features. Each dimension within this vector corresponds to a specific feature or combination of features, enabling a detailed and exhaustive analysis of the images. This strategic approach empowers the algorithms to achieve higher accuracy in disease detection and classification within the GI tract.

The essence of the Kvasir dataset lies in its meticulous curation, involving the collection of annotated images from genuine endoscopy examinations. By extracting multiple image features and integrating them into a high-dimensional feature vector, this dataset facilitates accurate and comprehensive analysis of the images. It contributes significantly to the advancement of computer-aided detection and diagnosis within the realm of gastrointestinal diseases.

In summary, the unique attributes of the Kvasir dataset, coupled with the extraction and integration of multiple image features into a high-dimensional feature vector, align closely with my research objectives. This distinctive combination empowers me to conduct an in-depth and exhaustive analysis, enabling the development of advanced computer algorithms that aid in the detection and diagnosis of gastrointestinal diseases.

(ii)

The code aimed at constructing a feature vector for a specific feature file. First we read the content of the feature file located at the specified path and assign it to the vaish_data variable. The file contains the feature data for multiple observations.

Next we take a string of feature values as input and perform several operations to extract the individual feature values:

- Replaced "e-" with "e" to handle exponential notation.
- Split the string by commas, discarding the first element (which is the feature name) using unlist(strsplit(...))[-1].
- Trimmed leading and trailing whitespaces using trimws.
- Converted the extracted values to numeric format, excluding any elements that are not numeric using as.numeric(values[!is.na(values)]).
- Finally, it returned the extracted feature values.

The variable length is set to 1186, representing the desired length of the feature vector. Then, a matrix v_vector is created with the number of rows equal to the length of vaish_data (the number of observations) and the number of columns equal to length.
Then the loop is iterated over each observation in vaish_data. For each observation, it calls the v_extract_values function, passing the corresponding feature data string. The function extracts the feature values and assigns them to the features variable. Then, the feature values are placed in the corresponding row of the v_vector matrix.
This line of code returns the dimensions of the v_vector matrix, which represents the constructed feature vector. The output you obtained, [1] 6 1186, indicates that the v_vector matrix has 6 rows and 1186 columns.

(iii)

In this question we construct a dataset of 1186 feature vectors for all the feature/image files in the specified directory. The code reads each feature file, extracts the numerical values, and constructs a dataset of feature vectors. It handles cases where feature files may have a different number of values. The output shows that the constructed dataset contains 1000 feature vectors, where each vector has 1186 features. The code successfully handles cases where a feature file may have a different number of values by either truncating or padding the feature vectors appropriately.

1. The code defines a function v_extract_values which takes a feature string as input and extracts the numerical values from it. It performs necessary string manipulations and converts the values to numeric format.
2. The variable num_features is set to 1186, representing the desired number of features in each feature vector.
3. The code retrieves the list of feature files in the specified directory using list.files function. Each file is represented by its full path.
4. The variable num_files is assigned the number of feature files in the directory.
5. A matrix called feature_dataset is initialized with dimensions num_files rows and num_features columns. This matrix will store the constructed feature vectors.
6. The code enters a loop that iterates over each feature file.
7. Within the loop, the content of the current feature file is read using the readLines function and stored in the variable feature_data.
8. The v_extract_values function is applied to feature_data, extracting the numerical values and storing them in the variable features.
9. The code determines the number of values extracted from the feature file using the length function and assigns it to num_values.

10. If the number of values (num_values) is greater than or equal to the desired number of features (num_features), the first num_features values are assigned to the corresponding row in the feature_dataset matrix. Otherwise, if num_values is less than num_features, the available values are assigned to the corresponding row, and the remaining entries in the row are left as zeros.
11. After processing all feature files, the loop ends, and the final feature_dataset matrix is obtained.
12. The dimensions of the feature_dataset matrix are printed using dim function, confirming the number of feature vectors (rows) and the number of features (columns) in the dataset.

(iv)

To reproduce the '6 GF Random Forest' results from Table 1 of the paper, we need to follow these steps:

First, we need to obtain the dataset used in the paper. We need to ensure that we have access to the features and corresponding labels mentioned in the paper. We need to make sure to preprocess and format the data properly based on the instructions provided in the paper. This may involve handling missing values, scaling features, or encoding categorical variables.
Next, we will split the dataset into training and testing sets. A common practice is to randomly split the data, allocating around 70-80% for training and the remaining 20-30% for testing. It is crucial to maintain the distribution of classes in the dataset during the split to ensure representative training and testing sets.

After splitting, we will prepare the feature vectors that will serve as input to the Random Forest model. We will then extract the relevant features from the dataset and ensure they are correctly formatted and compatible with the Random Forest algorithm. This may involve feature selection, transformation, or engineering to enhance the predictive power of the model.
Then we will proceed to train the Random Forest model using the training set. We need to set the model's parameters, such as the number of trees, maximum depth, and other hyperparameters, based on the specifications provided in the paper. We will fit the model to the training data, allowing it to learn the underlying patterns and relationships between the features and labels.

Once the model is trained, we will evaluate its performance on the testing set using appropriate evaluation metrics, such as accuracy, precision, recall, F1-score, or any other metrics mentioned in the paper, to assess the model's predictive ability. We will then compare the obtained results

with the '6 GF Random Forest' row in Table 1 of the paper to validate the reproducibility of the findings.

To improve the results, we can explore several approaches:

Firstly, we can focus on feature engineering. By leveraging domain-specific knowledge and employing various feature engineering techniques, we can extract more informative features from the dataset. This process involves carefully selecting relevant features, creating new features through transformations, or combining existing features to enhance the model's predictive capabilities.

Additionally, we can fine-tune the hyperparameters of the Random Forest model. By performing a systematic search of different hyperparameter values, such as the number of trees, maximum depth, and minimum sample split, we can find the optimal combination that maximizes the model's performance. Techniques like grid search or random search can be utilized for this purpose.

Ensemble methods can also be considered to further improve the results. Combining Random Forest with other ensemble techniques, such as bagging or boosting, can potentially enhance the model's predictive power. These methods leverage the diversity of multiple models to make more accurate predictions.

Furthermore, data augmentation techniques can be applied to increase the size and diversity of the training set. By generating synthetic data samples through techniques like rotation, translation, or noise addition, we can provide the model with more varied examples, which can help improve its generalization ability.

Lastly, it's worth exploring alternative machine learning algorithms and comparing their performance with Random Forest. Different algorithms may have varying strengths and weaknesses depending on the dataset and task at hand. By experimenting with alternative models, such as support vector machines, gradient boosting, or neural networks, we can potentially uncover a more suitable algorithm that yields improved results.

By systematically exploring these approaches and iteratively refining our methods, we can strive to surpass the performance reported in the paper and achieve enhanced results for the given dataset and task.

(v)

To improve the model performance incorporated following changes in the code

1. Increased the number of trees (ntree) to 150 for a more robust ensemble.

2. Utilized the square root of the number of variables (mtry) for optimal feature selection.
3. Enabled importance measure and proximity calculation to assess feature importance and obtain the proximity matrix.
4. Set a smaller nodesize value (10) to control the minimum number of samples required for a node.
5. Adjusted the sample sizes for the training and testing datasets to a manageable size.

These improvements aim to enhance the model's performance and provide a more accurate evaluation.