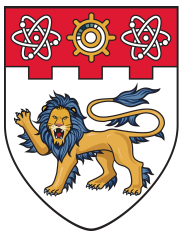


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

MH4510 Statistical Learning and Data Mining

Group: Blush of Dawn

Classification of Blood Cancer Subtypes using Images

Name	Matriculation Number
Alvin Goh Jia Hao	U2140086C
Chong Wei Jun Kelvin	U2140300J
Shannon Lee Wei Kei	U2140284D
Sun Jia Yee Joyce	U2140792E
Sonya Annalise Thong Li Wen	U2140424K

Abstract

Blood cancer remains one of the most pressing concerns globally, and its means of diagnosis is still a challenge to this day. Current image-based research on blood cancer cells primarily focuses on single-cell images often neglecting the spatial features between multiple cells. Our aim is to train models for multi-classification task to classify leukemia subtypes using the Acute Lymphoblastic *Leukemia* (ALL) dataset, which comprises a diverse collection of 3,256 multi-cell blood smear images classed according to their maturity subtypes. In this project, we also proposed a new Convolutional Neural Network (CNN) architecture evaluated multiple machine learning models, including SVM, XGBoost, VGG16, DenseNet-201, and ConvNeXt. We focused on optimizing accuracy while considering the misclassification costs associated with different leukemia stages, and found that ConvNeXt and DenseNet-201 pretrained model combined with our proposed architecture performs the best, with DenseNet-201 topping in performance across accuracy and having minimal misclassification costs.

Contents

1 Introduction

Blood cancer remains one of the most critical forms of cancer worldwide, and has been increasing annually since 1990, reaching a total of 1.3 million in 2019 [1]. Accurate risk assessments are vital for improving treatment outcomes, as early intervention can significantly increase survival rates [2]. Current diagnostic practices, however, including microscopic evaluation and histopathological methods, often rely on subjective visual analysis, which may lead to inconsistencies or delays in diagnosis [3]. As a result, deep learning models, particularly convolutional neural networks (CNNs), for image-based classification offer a promising direction for enhancing accuracy and efficiency in blood cancer diagnostics [4]. Such advancements could revolutionize how early-stage cancers like leukemia are identified and managed, especially by introducing automated tools capable of handling large datasets of blood cell images.

1.1 Literature Review

Many studies in artificial intelligence using images have successfully used convolutional neural networks (CNNs) for classifying types of blood cancer, demonstrating high accuracy levels. For instance, in leukemia classification, researchers have successfully applied CNNs to detect and classify various stages of leukemia cells from peripheral blood smear images with promising accuracy. One study used a CNN model to classify Acute Lymphoblastic Leukemia (ALL) cells from microscopic images, achieving an accuracy of over 90 percent on their test set, demonstrating the potential of CNNs in blood cancer diagnostics [5].

1.2 Gaps in Current Knowledge

While some studies have employed machine learning techniques and artificial neural networks for classification of leukemia using CNN, most have particularly focused on classification associated with single-cell images of blood cancer progression based on images of cell morphology. One particular study comes from research in Iran where a neural network model is used to classify developmental classes of B-Cell leukemia based on multi-cell images [6], but their objectives were limited to only using and developing CNN models. Another study was done recently that adapted ConvNeXt to blood cell classification, but they did not focus on specifically blood cancer subtypes [7]. Despite advancements in blood cancer classification through deep learning, there remains considerable space to explore in classifying the prediction of blood cancer maturity, using multi-cell images that are based on their appearances.

1.3 Research Objectives

The primary research objective of this study is to adapt machine learning methods and compare various pretrained CNN models with modern ConvNeXt architecture for classification tasks, using peripheral smeared blood images. We will first build a feature extraction function to extract features from images and use machine learning models such as Support Vector Machine (SVM) and XGBoost to train on those features for multi-class classification. Subsequently, we move on to propose a new CNN architecture that does both segmentation and classification, training directly on our image dataset for classification tasks, and using pre-trained network architectures such as VGG16, DenseNet-201 and a recent model ConvNeXt, to evaluate which pretrained models are the most optimal for our task and architecture.

2 Dataset and Pre-processing

2.1 Description of Dataset

The models in this study will be using the archived dataset of Acute Lymphoblastic Leukemia (ALL) blood cancer, which provides a large collection of 3,242 peripheral blood smear (PBS) images, organized into two subfolders where one contains the original images captured, and the other one pre-segmented. Within each of these two subfolders contains folders which hold images based on developmental subtypes: Benign, Early Pre-B, Pre-B, and Pro-B ALL. The dataset was prepared in the bone marrow laboratory of Taleqani Hospital in Tehran, Iran.

The dataset consists of image files in JPEG format. Figure 1 shows the folder structures.

```
archive/
|
├── segmented/
|   ├── Benign/
|   ├── Early/
|   ├── Pre/
|   ├── Pro/
├── original/
|   ├── Benign/
|   ├── Early/
|   ├── Pre/
|   ├── Pro/
```

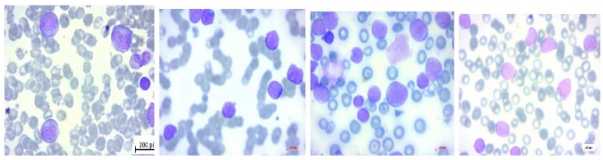
Figure 1: Directory map for the chosen dataset

Figure 2 is a table showing the class names of the developmental subtypes, with their description explaining what each of them means. Benign is considered to be the least aggressive stage amongst all other developments, while Pro is considered to be the most severe and advanced stage of Leukemia B-Cells.

Subclass	Description
Benign	Non-cancerous, healthy cells
Early	Early stages of leukemia
Pre	Pre-stage abnormal cells
Pro	Advanced leukemia cells

Figure 2: Directory map for the chosen dataset

Figure 3 shows some sample images from the original folder of image datasets, each of which were labeled according to their maturity stages.



(a) Benign (b) Early (c) Pre (d) Pro

Figure 3: Directory map for the chosen dataset

2.2 Exploratory Data Analysis

We will be cleaning the data we have, and since our raw data consist mostly of pictures instead of numbers, we will be correcting the resolution of the images and taking appropriate measures during the model training process later on.

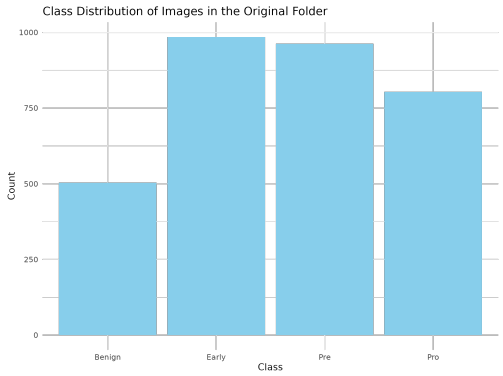


Figure 4: Class Distribution

The class distribution plot in Figure 4 indicates the number of images present in each category. The dataset comprises 3256 images, distributed across the classes as follows: Benign (504 images), Early (985 images), Pre (963 images), and Pro (804 images). The majority of images belong to the Early and Pre class, suggesting a potential imbalance that could affect certain models performance.

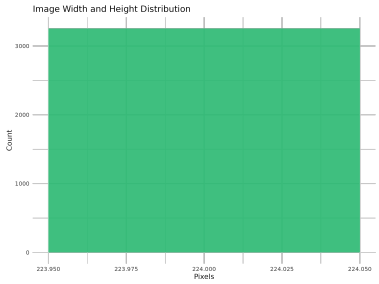


Figure 5: Image dimensions

The analysis of image dimensions as shown in the plot of Figure 5 revealed that all images were consistently sized at 224×224 pixels. This uniformity is advantageous as it simplifies preprocessing steps and aligns with standard input requirements for convolutional neural networks (CNNs). The consistent aspect ratio of 1:1 further ensures that the images maintain their intended visual properties during training, thus reducing potential biases in model predictions.

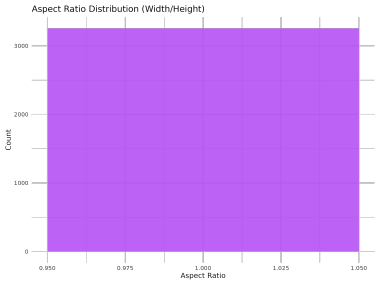


Figure 6: Aspect ratios

The aspect ratio of all images according to Figure 6 was consistently measured at 1.0, affirming that the dataset contains square images. This characteristic is essential for maintaining visual integrity when inputting the images into machine learning models. A uniform aspect ratio eliminates potential distortions that may arise from varying image shapes, thereby facilitating better feature extraction by the models.

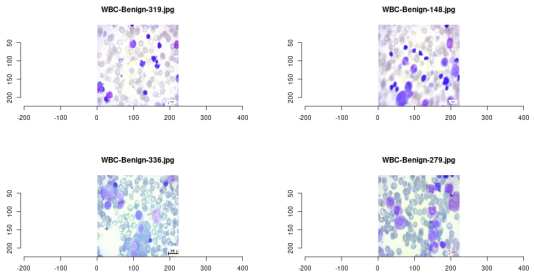


Figure 7: Benign

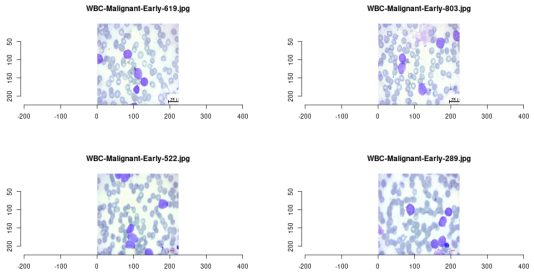


Figure 8: Early

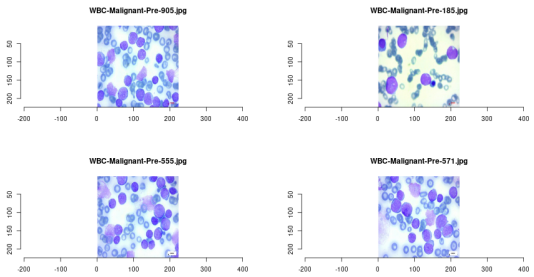


Figure 9: Pre

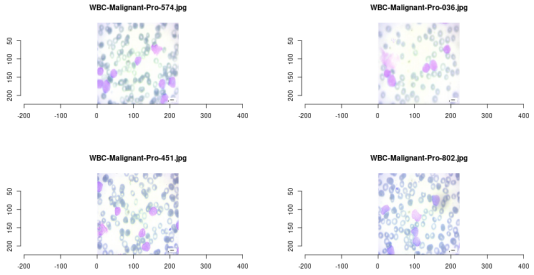


Figure 10: Pro

To further understand the dataset, a selection of sample images from each class was visually inspected. The images displayed clear differentiation between classes especially between Early and Pre, as well as Pro which shows a sudden lightness in colour tone of the cells, reflecting variations in cellular morphology that could be critical for accurate classification. This visual assessment is crucial for confirming the relevance of the dataset to the research objectives and the potential for successful application in automated diagnostic systems. From the exploratory data analysis of the image data and resolution, the only aspect we need to focus on is rebalancing the datas scarcity and unequal sizes in each class via augmentation techniques.

3 Feature Engineering

The loading of dataset process begins by defining the full dataset of image paths and their corresponding class labels. It first gathers all image file paths from the directory /kaggle/input/leukemia-images/Original for the four defined classes ("Benign", "Early", "Pre", "Pro") and stores them in the file_paths vector. Each images class is recorded in the class_labels vector. This initial step ensures that the dataset is categorized and ready for splitting. Figure 14 below shows the first 6 rows of the table output for the full dataframe before splitting.

A data.frame: 6 × 2

	image_filename	class
	<chr>	<chr>
1	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-001.jpg	Benign
2	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-002.jpg	Benign
3	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-003.jpg	Benign
4	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-004.jpg	Benign
5	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-005.jpg	Benign
6	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-006.jpg	Benign

Figure 11: Full dataset

We call this DataFrame full_df, which associates each image path with its corresponding class. A numeric representation of each class in the column class_numeric is also generated by converting the class labels into numeric indices (0 for Benign, 1 for Early, 2 for Pre, and 3 for Pro). This numeric conversion is important because machine learning models typically require numerical data for training. The class labels are mapped to integers using the factor function, which is then adjusted to start from 0 (using as.numeric(factor(...)) - 1).

A data.frame: 6 × 3

	image_filename	class	class_numeric
	<chr>	<chr>	<dbl>
1618	/kaggle/input/leukemia-images/Original/Pre/WBC-Malignant-Pre-129.jpg	Pre	2
1098	/kaggle/input/leukemia-images/Original/Early/WBC-Malignant-Early-594.jpg	Early	1
3176	/kaggle/input/leukemia-images/Original/Pro/WBC-Malignant-Pro-724.jpg	Pro	3
104	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-104.jpg	Benign	0
514	/kaggle/input/leukemia-images/Original/Early/WBC-Malignant-Early-010.jpg	Early	1
3161	/kaggle/input/leukemia-images/Original/Pro/WBC-Malignant-Pro-709.jpg	Pro	3

Figure 12: Shuffled train dataset

The dataset is then split into training, validation, and test sets using a stratified approach. This ensures that the class distribution is maintained in each subset. First, a random 64% of the data is allocated to the training set using createDataPartition from the caret library. The remaining 36% is set aside for the validation and test sets. After this, the remaining data (36%) is further divided, with 20% designated for validation and 16% for testing. Only the training data is specifically shuffled before creating the validation set.

A data.frame: 6 × 3

	image_filename	class	class_numeric
	<chr>	<chr>	<dbl>
6	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-006.jpg	Benign	0
8	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-008.jpg	Benign	0
15	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-015.jpg	Benign	0
19	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-019.jpg	Benign	0
21	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-021.jpg	Benign	0
28	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-028.jpg	Benign	0

Figure 13: Validation dataset

A data.frame: 6 × 3

	image_filename	class	class_numeric
	<chr>	<chr>	<dbl>
1	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-001.jpg	Benign	0
3	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-003.jpg	Benign	0
9	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-009.jpg	Benign	0
12	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-012.jpg	Benign	0
17	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-017.jpg	Benign	0
18	/kaggle/input/leukemia-images/Original/Benign/WBC-Benign-018.jpg	Benign	0

Figure 14: Test dataset

By keeping the validation and test sets in their original order, the model is tested in a consistent, replicable manner, ensuring that the evaluation reflects the model's generalization capabilities rather than its ability to adapt to random data orderings. In terms of how the trained neural network model will be used in diagnosis, users will only need to input an image of blood cancer cells into the model and they will receive a classification result of one of the 4 subtypes as an output.

3.1 Segmented and Original Images for Different Model Trainings

The machine learning methods we will be using for this project are SVM and XGBoost, while the neural networks models we will be using are CNN pretrained models and a new architecture model ConvNeXt. Since these models thrive on rich feature extraction such as colour and intensity, images from the Original folder in the dataset will be chosen to train them. This is because the original images likely retain natural colour variations and contextual information that enhance feature diversity, allowing the model to use informative colour and texture features.

By extracting features like colour histograms and texture descriptors from the original images, SVM and XGBoost can differentiate between classes based on these informative characteristics more effectively compared to the pre-segmented images in the Segmented folder. For CNN models, we will not rely on existing segmented image dataset to train our neural network models. Instead, we propose a new architecture that uses U-Net to do the segmentation for us.

3.2 Feature Extraction for Machine Learning Models

Feature extraction is a fundamental step in image processing for machine learning pipelines. It transforms raw data into structured information, enhancing models' abilities to recognize patterns and make accurate predictions [8]. This process is essential for models like Support Vector Machines (SVMs) and XGboost. Below, we provide a mathematical overview of key feature extraction techniques.

Color Histogram Features

Color histograms represent the distribution of color intensities within an image. For an RGB image, each pixel is defined by red, green, and blue intensities. A histogram is created for each channel, where intensity values are split into bins, and the frequency of pixels in each bin is recorded.
For a color channel C with intensity values I_j divided into n bins, the histogram for bin i is defined as:

$$\text{hist}(C, i) = \frac{\sum_j \mathbb{1}(I_j \in \text{bin}_i)}{\text{Total number of pixels}}$$

where $\mathbb{1}(I_j \in \text{bin}_i)$ is an indicator function that counts if pixel I_j falls into bin i [9]. By normalizing, this feature becomes size-invariant, allowing direct comparison across images. Color histograms capture color distribution compactly, simplifying classification.

Texture Features

Texture features capture patterns in pixel intensities, describing an images surface characteristics (Haralick et. al, 2007). Key descriptors include:

Contrast

Contrast measures pixel intensity variation and is calculated as the variance σ^2 around the mean intensity μ:

$$\text{contrast} = \frac{1}{N}\sum_{j=1}^{N}(I_j - \mu)^2$$

where I_j is the intensity of the j-th pixel, N is the total number of pixels, and μ is the mean intensity.

Skewness

Skewness indicates asymmetry in intensity distribution. Positive skew suggests more light pixels, while negative skew indicates more dark pixels:

$$\text{skewness} = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{I_j - \mu}{\sigma}\right)^3$$

Kurtosis

Kurtosis captures the tailedness of intensity values, indicating how often extreme values occur:

$$\text{kurtosis} = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{I_j - \mu}{\sigma}\right)^4 - 3$$

These texture features summarize the statistical distribution of intensities, aiding models by highlighting the complexity or smoothness of surfaces.In the kurtosis formula, the subtraction of 3 serves to "normalize" the kurtosis relative to the normal distribution, which has a kurtosis of 3. By subtracting 3, the resulting measure is known as excess kurtosis

Edge Detection Features

Edge detection identifies boundaries between regions in an image. Using the Sobel operator, intensity gradients are approximated across x- and y-axes [10]:

$$\text{edge magnitude} = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

where $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ represent intensity changes along the horizontal and vertical directions, respectively. The mean and standard deviation of gradients provide models with a compact measure of image edges, which help in recognizing structural boundaries.

Shape Features

Shape features describe an objects geometric properties in an image, often derived after binary segmentation [11]. These descriptors include area, perimeter and circularity.

Area (A) is the total pixel count in the object, representing size:

$$A = \sum_j \mathbb{1}(I_j \in \text{object pixels})$$

Perimeter (P) measures the length of the boundary around the object, capturing contour details. Circularity indicates roundness, comparing the shape to a circle:

$$\text{circularity} = \frac{4\pi A}{P^2}$$

These features allow models to distinguish shapes by providing roundness or elongation dimensions, useful for object detection tasks.

Thus, feature extraction reduces high-dimensional pixel data into compact representations that capture essential image characteristics.

3.3 Segmentation

Segmentation in image classification refers to the process of dividing an image into meaningful regions, or segments, based on shared attributes like color, texture, or intensity. This technique is important because it allows for more accurate and efficient analysis by isolating relevant features that aid in object detection and classification. For example, in tasks such as medical imaging or autonomous driving, segmentation helps identify specific areas of interest, like tumors or pedestrians, reducing noise and improving the performance of classification models. By focusing on distinct regions rather than raw images, segmentation allows machine learning models to process information more effectively, leading to better accuracy in predictions [12]. We will explain more in our proposed architecture where we use U-Net for the segmentation process.

Figure 15: Before and after segmentation of Bening and Early images

3.4 Data Augmentation

Data augmentation is a crucial technique employed in this study to enhance the performance of the convolutional neural network (CNN) model for blood cancer cell classification tasks. Given the imbalanced distribution of the dataset, which consists of 3,256 images across four classes: Benign (504 images), Early (985 images), Pre (963 images), and Pro (804 images), data augmentation serves to artificially increase the diversity of the training set [13]. We will be using the Original images version of the dataset to illustrate the augmentation.

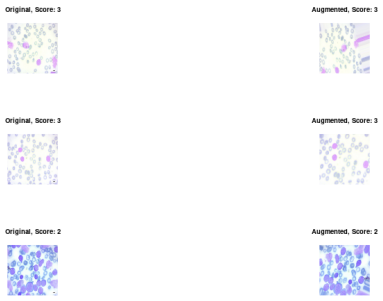


Figure 16: Before and after augmentation

Figure 16 involves a series of transformations applied to the original images, including random rotations (up to 30 degrees), width and height shifts (up to 20 percent of the total dimensions), shear transformations, and zooming (up to 20 percent). Additionally, horizontal flipping is implemented to introduce variations in image orientation. Each of these transformations helps to create multiple variations of the same image, thereby improving the model's ability to generalise and reducing the risk of overfitting.

By rescaling pixel values to a range of [0, 1] and utilising a validation split of 20 percent, the augmentation pipeline not only enhances the model against class imbalance but also aids in improving the overall accuracy of predictions based on their shapes and characteristics. This is particularly useful for the classes with fewer images (e.g., Benign), as it allows the model to learn from a more diverse set of examples without needing additional data.

Data generation in data augmentation involves creating modified versions of existing data samples to increase the diversity and size of the dataset, enhancing model training process and reducing overfitting. This process includes applying a range of transformations to the original data while preserving its underlying patterns and labels. For image data, common augmentations include rotations, flips, cropping, scaling, brightness adjustments, and introducing noise. For text, it might involve paraphrasing, synonym replacement, or shuffling words. By synthetically expanding the dataset, data augmentation helps machine learning models generalize better to unseen scenarios without requiring additional labeled data.

3.5 Stratified Sampling

Stratified sampling is a technique used to ensure that each subgroup (or "stratum") within a dataset is proportionally represented in the sample [14]. In this method, the population is divided into distinct, non-overlapping subgroups based on a specific characteristic, such as class labels. A sample is then drawn from each subgroup, with the sample size from each stratum determined proportionally to its size in the overall dataset. This approach is valuable in datasets where certain classes or groups may be underrepresented, especially such as the Benign class in our dataset. We will apply this to all the model trainings in this project.

3.6 Class Weighting

The class weighting technique serves as an effective strategy to address the unequal representation of classes in the context of classification. This methodology is especially pertinent in scenarios where certain classes are underrepresented, leading to potential bias in the model's learning process.

Let C represent the total number of unique classes in the dataset, and let n_i denote the number of instances in class i, where i0,1,2,,C1. The goal of class weighting is to assign a weight w_i to each class i such that the learning algorithm compensates for the underrepresentation of minority classes. The class weight w_i can be computed using the following formula:

$$w_i = \frac{N}{C \cdot n_i}$$

where:

- w_i is the weight for a sample belonging to class i,
- N is the total number of instances in the dataset,
- C is the total number of unique classes,
- n_i is the number of instances in class i.

This equation serves a dual purpose: it ensures that classes with fewer instances are assigned higher weights, thereby increasing their importance during the training process. As a result, the model is penalised more heavily for misclassifying samples from minority classes, directing the learning algorithm to allocate more attention to these classes. By quantifying the significance of each class through the assigned weights, the model can learn more effectively from all available instances, thereby enhancing its predictive capabilities.

To incorporate the calculated class weights, the loss function of a model is modified to give more importance to certain samples based on their calculated weights. The weighted version of the loss function is:

$$\mathcal{L}_{\text{weighted}} = \frac{1}{N} \sum_{i=1}^{N} w_i \cdot L_{\text{original}}$$

where:

- w_i is the weight assigned to the i-th sample,
- L_{original} is the original loss function of the model.

Classes with larger weights contribute more to the loss. This is typically the case for underrepresented classes, which are given more importance during model training. Conversely, samples with smaller weights contribute less to the loss. This is often the case for overrepresented classes, where their contributions are minimized to avoid dominating the models learning process.

Thus, the weighted loss function ensures that the model is trained to focus more on samples that are critical for improving the performance on minority classes (or other underrepresented samples), while not overfitting to the majority classes. During training, the model aims to minimize this weighted loss function. The gradient descent optimization will adjust the models parameters to reduce the weighted error across all samples in the dataset, making the model more balanced in its predictions across different classes.

4 Evaluating Misclassification Costs

The cost of misclassification in the context of classifying blood cancer subtypes is significant, particularly due to the medical nature of the task. The impact of incorrectly classifying one class as another can have serious consequences, such as giving wrong treatment to the patients or disregarding serious existing conditions. Below is how we perceived the severity for each misclassification.

On the other hand, misclassifying early conditions as benign is a critical error. Such a mistake could lead to a delay in necessary interventions during the early stages of a condition, at a time when treatment is most effective and outcomes are more favorable. We give this a score of 10.

Misclassifying an "Early" stage as a "Pre" stage might still lead to relatively less severe consequences, such as slight misinterpretations of risk, which could justify a moderate cost of 15. However, misclassifying "Early" as "Pro" could have more serious ramifications, such as unnecessarily aggressive treatments or a failure to intervene early enough, which would justify a higher cost of 25.

The most severe misclassification occurs when pre or pro conditions are misclassified as benign. This error could result in overlooking advanced or aggressive conditions that require immediate attention. As a result, patient outcomes could worsen due to a lack of timely treatment. We give this a score of 20 for misclassifying pre, 30 for pro.

Another possible error is when pre is misclassified as pro, or vice versa. Although this is less severe than misclassifying a condition as benign, it still carries risks. It could lead to inappropriate treatment protocols, such as overtreatment or undertreatment, depending on the incorrect progression stage. We give this score a 10.

With these in mind, we formulate a cost matrix that assigns a unit cost to each misclassification according to their severity:

$$\text{Cost Matrix} = \begin{bmatrix} 0 & 10 & 20 & 30 \\ 10 & 0 & 15 & 25 \\ 20 & 15 & 0 & 10 \\ 30 & 25 & 10 & 0 \end{bmatrix}$$

Here, diagonal values represent correct classifications (cost = 0), and off-diagonal values represent misclassification costs. For the simplicity of this project, we assume that the cost of misclassifying one class with the other is equal as vice versa.

Using the confusion matrix, the total misclassification cost can be calculated by multiplying the frequency of each type of misclassification by its corresponding cost in the cost matrix and summing all costs:

$$\text{Total Cost} = \sum_{i \neq j} \text{ConfusionMatrix}[i, j] \times \text{CostMatrix}[i, j]$$

This metric aggregates the costs of all misclassifications, providing an overall measure of how costly the errors made by the model are.

To normalize the total misclassification cost, the average cost per sample is calculated by dividing the total cost by the total number of samples. This metric provides a more practical evaluation by considering the cost of misclassification on an individual basis, making it easier to compare the model's performance across different datasets or models of varying sizes:

$$\text{Average Cost Per Sample} = \frac{\text{Total Cost}}{\text{Total Number of Samples}}$$

Rather than relying on traditional accuracy metrics, we incorporating a cost-sensitive accuracy which takes into the account of each misclassification cost:

$$\text{Cost-Sensitive Accuracy} = 1 - \frac{\text{Total Misclassification Cost}}{\text{Maximum Possible Cost}}$$

Where the maximum possible cost is the sum of all misclassifications if every sample were misclassified.

The cost-benefit ratio is another important evaluation tool, comparing the benefits of correct classifications with the penalties of misclassifications:

$$\text{Cost-Benefit Ratio} = \frac{\text{Correct Classification Benefit}}{\text{Misclassification Cost}}$$

We can assign a monetary or impact value to correct classifications based on domain knowledge.

Adjust the F1-score for each class by incorporating the misclassification costs:

$$\text{Weighted F1-Score} = \sum_i (F1_i \times W_i)$$

Where $W_i = \frac{1}{\text{CostMatrix}[i,j]}$ is the weight based on misclassification cost.

These formulas will be used to evaluate the cost of misclassification of every chosen model in this project, and will also be used as metrics for comparison between the models.

5 Machine Learning Models

In this section, we will introduce and train chosen models of machine learning for our classification task. Descriptions for each model and their respective techniques of model tuning and adjustments, together with their training and test error results are also included.

5.1 Support Vector Machines

Support Vector Machines (SVM) are a class of supervised learning models commonly used for classification and regression tasks. The fundamental idea behind SVM is to find a hyperplane (or decision boundary) that best separates data points of different classes in a high-dimensional feature space. In the case of classification tasks, such as in our project, SVM can predict continuous values by fitting a hyperplane that minimizes the error margin while maintaining a balance between model complexity and accuracy.

Given a dataset $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^n$ are feature vectors and $y_i \in \{-1, 1\}$ are class labels, SVM seeks to maximize the margin M between the decision boundary and the nearest data points, known as the support vectors. The decision function is given by:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b,$$

where $\mathbf{w}$ is the weight vector and b is the bias term. The optimization problem for a soft-margin SVM is:

$$\min_{\mathbf{w},b,\xi} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i,$$

subject to the constraints:

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0,$$

where ξ_i are slack variables that allow for misclassification, and $C > 0$ is a regularization parameter that controls the trade-off between margin maximization and misclassification penalties.

When data is not linearly separable, the SVM uses a kernel trick to map the input features to a higher-dimensional space, where linear separation becomes possible. The Radial Basis Function (RBF) kernel is given by:

$$K(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

where:

- $K(x, y)$: The value of the kernel function between the two data points x and y.
- x, y: The input data points (vectors) in the feature space.
- $\|x - y\|^2$: The squared Euclidean distance between x and y, which measures how far apart the points are in the feature space.
- σ: The bandwidth parameter (also known as the kernel parameter) that controls the spread or smoothness of the kernel. It defines the width of the Gaussian function. Smaller values of σ make the kernel more sensitive to local variations, while larger values make it smoother.

The dual form of the optimization problem becomes [15]:

$$\max_{\alpha} \quad \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j),$$

subject to the constraints:

$$\sum_{i=1}^{N} \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C.$$

Here, α_i are the Lagrange multipliers, and $K(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function.

To find the optimal values of the regularization parameter C and the kernel parameter σ, a grid search with cross-validation is used. The parameter C determines the trade-off between achieving low error on the training data and maximizing the margin. A high C prioritizes minimizing classification errors, which may lead to overfitting, while a low C favors larger margins, potentially leading to underfitting. The parameter σ controls the influence of individual data points. A smaller σ produces a smoother decision boundary, while a larger σ allows the model to capture more detailed variations in the data.

In the case of class imbalance, we assign weights to the classes based on their frequencies. The weight for each class is calculated with the same formula as given in section 3.6.

The performance of the model is evaluated using metrics such as accuracy, precision, recall, and the confusion matrix. The decision function $f(\mathbf{x}) = \mathbf{w}^\top \Phi(\mathbf{x}) + b$, where $\Phi(\mathbf{x})$ is the nonlinear mapping induced by the RBF kernel, determines the predicted class. Cross-validation is used to assess the accuracy of the model by averaging performance over multiple data splits.

5.1.1 Training process and tuning

The training process first begins by setting up the environment and loading libraries such as keras, tensorflow, and EBImage for image processing and machine learning tasks. Image data is loaded from a directory structure, where each subdirectory represents a class. The code organizes the data into a structured dataframe, assigns numeric labels to classes, and splits the dataset into training, validation, and test sets using a stratified approach to maintain class distribution. Class weights are computed to address potential class imbalance issues.

A data.frame: 6 × 64

	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	0.03163656	2.6236003	3.1068061	1.5554515	0.59113280	0.1519493	-0.1634713	-0.3242494	-0.50210272
2	-0.06400671	-0.2602499	-0.3348147	-0.4120166	-0.46574645	-0.5371078	-0.6452848	-0.6382741	-0.09371064
3	-0.25529325	-0.2883850	-0.3505538	-0.4146364	-0.46873620	-0.5939824	-0.7390973	-0.8789369	-0.94534665
4	0.22292310	-0.1195743	-0.1721771	-0.1395576	-0.02176747	0.8902248	2.9614860	3.2212775	1.79365058
5	-0.25529325	-0.2602499	-0.3453074	-0.3989176	-0.29234052	0.2481986	2.2455966	2.0868520	0.49340883
6	-0.25529325	-0.2883850	-0.3505538	-0.4146364	-0.46873620	-0.5939824	-0.7390973	-0.8798315	-0.95696393

Figure 17: Extracted and filtered train features

Images are resized and processed to generate features, including color histograms, texture descriptors, edge intensity, and shape features derived from segmentation. The feature extraction function supports validation to ensure consistent output dimensions across samples. The script also analyzes and resolves inconsistencies in feature lengths by padding or truncating as necessary. Features are standardized to facilitate effective model training, and near-zero variance (NZV) features are identified and excluded to prevent model overfitting or inefficiency. The remaining features are used to build training, validation, and test datasets, which are ready for SVM training. Figure 16 shows the train features after extraction, standardization and filtered of NZV.

The SVM model undergoes hyperparameter tuning using a radial basis function (RBF) kernel. A grid search is conducted over a range of values from $10^{-}3$ to 10^{3} for C (cost) and σ (RBF kernel parameter) via 5-fold cross-validation. Class weights are applied to handle class imbalances during training. The script selects the best model based on validation accuracy for each fold and evaluates its performance on the test set. Subsequently, we report the summary statistics for training and validation accuracy across folds. Selected optimal hyperparameters were then used to retrain the final model. The trained final model is then evaluated on all datasets (training, validation, test), with confusion matrix for its performance on test set generated for detailed insights.

5.1.2 Test and train results

Cross-validation results:

	fold	train_accuracy	valid_accuracy	sigma	C
Accuracy	1	0.9807692	0.8865031	0.01	10
Accuracy1	2	0.9976134	0.8742331	0.01	100
Accuracy2	3	0.9952038	0.8895706	0.01	100
Accuracy3	4	1.0000000	0.8757669	0.01	1000
Accuracy4	5	0.9976077	0.9064417	0.01	100

Figure 18: Cross validation results for SVM

From the cross-validation results in figure 18, the most optimal hyperparameters are σ = 0.01 and C =100 is the 5-th fold having validation accuracy of 0.9064.

	Reference			
Prediction	Benign	Early Pre		Pro
Benign	70	5	1	3
Early	10	150	3	0
Pre	0	2	148	0
Pro	0	0	1	125

Overall Statistics

Accuracy	: 0.9517
95% CI	: (0.9296, 0.9685)
No Information Rate	: 0.3031
P-Value [Acc > NIR]	: < 2.2e-16
Kappa	: 0.9344

Figure 19: Confusion matrix for SVM

The confusion matrix reveals the models predictions across four classesBenign, Early, Pre, and Pro. It shows strong performance, with the majority of predictions falling along the diagonal (correct classifications). For instance, 70 out of 79 Benign cases and 150 out of 163 Early cases were correctly classified, although there is occasional confusion, such as Benign misclassified as Early (5 cases) and Early as Benign (10 cases). Similarly, "Pre" and "Pro" classes have minimal misclassifications (3 combined). With an overall accuracy of 95.17% and a Kappa score of 0.9344, the model demonstrates excellent performance and balanced classification, making it highly reliable for practical use.

Class-Specific Statistics:

	Sensitivity	Specificity	Pos Pred Value	Neg Pred Value	Precision
Class: Benign	0.8750000	0.9794521	0.8860759	0.9772210	0.8860759
Class: Early	0.9554140	0.9639889	0.9202454	0.9802817	0.9202454
Class: Pre	0.9673203	0.9945205	0.9866667	0.9864130	0.9866667
Class: Pro	0.9765625	0.9974359	0.9920635	0.9923469	0.9920635

	Recall	F1	Prevalence	Detection Rate
Class: Benign	0.8750000	0.8805031	0.1544402	0.1351351
Class: Early	0.9554140	0.9375000	0.3030888	0.2895753
Class: Pre	0.9673203	0.9768977	0.2953668	0.2857143
Class: Pro	0.9765625	0.9842520	0.2471042	0.2413127

	Detection Prevalence	Balanced Accuracy
Class: Benign	0.1525097	0.9272260
Class: Early	0.3146718	0.9597015
Class: Pre	0.2895753	0.9809204
Class: Pro	0.2432432	0.9869992

Figure 20: Class statistics by SVM

The model demonstrates strong performance, with a cross-validated mean training accuracy of $99.42\% \pm 0.77\%$ and validation accuracy of $88.65\% \pm 1.30\%$, indicating good generalization. On the test set, it achieved an impressive overall accuracy of 95.17% (95% CI: $92.96\%, 96.85\%$) and a Kappa score of 0.9344, reflecting high agreement with actual labels. Class-specific metrics show balanced performance, with sensitivities ranging from 87.50% (Benign) to 97.66% (Pro) and balanced accuracies exceeding 92% for all classes. Precision and recall are particularly high for "Pre" and "Pro," demonstrating strong classification for these categories. The confusion matrix reveals occasional misclassifications, primarily between "Benign" and "Early," but these are infrequent. The model generalizes well, with a small gap between training accuracy (99.71%) and test accuracy (95.17%), suggesting minimal overfitting.

5.1.3 Misclassification cost results

Cost-Weighted Misclassification Matrix:

	Benign	Early	Pre	Pro
Benign	0	50	20	90
Early	100	0	45	0
Pre	0	30	0	0
Pro	0	0	10	0

Total Misclassification Cost: 345

Average Cost Per Sample: 0.6660232

Cost-Sensitive Accuracy: 0.9878905

Weighted F1-score 0.9447882

Figure 21: SVM misclassification cost results

The model demonstrates strong cost-sensitive performance, with a low Average Cost Per Sample (0.666) and high Cost-Sensitive Accuracy (0.988), indicating effective minimization of penalties from misclassifications. The Total Misclassification Cost (345) is reasonable given the dataset size and complexity, while the Weighted F1-Score (0.945) highlights a balanced ability to accurately classify samples across all classes while considering the cost of errors. These metrics collectively suggest the model is well-suited for scenarios where minimizing misclassification costs is critical.

5.2 XGBoost

XGBoost is a gradient boosting algorithm that can deliver higher speeds and performance when handling structured and tabular data. Unlike other Random Forest, XGBoost forms the tree one at a time, allowing the new tree to correct the error produced by the previous tree, thereby improving the prediction of the model [16]. XGBoost has built-in regularization techniques such as L-1 / L-2 regularization to prevent overfitting which produces a model that generalizes well on unseen data [17]. XGBoost is also able to train models with large datasets efficiently and quickly due to the utilization of parallel and distributed computing [18].

XGBoost incorporates regularization in its objective function and takes the form

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_j \Omega(f_k), \quad \text{where} \quad \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

The first term l is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target y_i. As multiple labels are used to classify the images, we will select l to be the multi-class cross-entropy loss function. The second term Ω penalizes the complexity of the model. Inside it, T is the number of leaf nodes and $\|w\|^2$ is the squared sum of the leaf prediction values [19]. There are 2 hyperparameters to prevent overfitting of the model: γ penalizes the number of leaf nodes and λ is the L-2 regularization parameters for the leaf predicted values. When these hyperparameters are set to zero, the objective falls back to the traditional gradient tree boosting.

The model is trained in an additive manner. We let $\hat{y}_i^{(t)}$ be the prediction of the i-th instance at the t-th iteration, and add $f_t(x_i)$ to minimize the objective function, therby giving us

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

It will be difficult for us to solve the objective directly so we apply 2nd order Taylor approximation to it and get

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \left[l(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t)$$

$$\text{where} \quad g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}) \quad \text{and} \quad h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}).$$

Since $l(y_i, \hat{y}^{(t-1)})$ is a constant, we can remove it and simplifying the modified objective further, we arrive at the following equation that will give us our optimal value.

$$\tilde{L}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{\left(\sum_{i \in I_j} g_i\right)^2}{\sum_{i \in I_j} h_i} + \gamma T$$

g_i and h_i are known as the first and second order gradient statistics on the loss function. Alternatively, they are called gradient and hessian respectively [20].

5.2.1 Training process and tuning

In our project for classifying images of leukemia using XGBoost, the pipeline begins by loading necessary libraries such as xgboost, caret, dplyr, and imager, among others, for tasks like image processing, feature extraction, and model evaluation. The dataset, which consists of images from four classes (Benign, Early, Pre, Pro), is loaded and structured by first gathering the file paths for images of each class and associating labels to them. It ensures that the dataset is organized into a training, validation, and test set (64%, 20%, and 16%, respectively). Class weights are calculated based on the sample sizes of each class, and the dataset is shuffled to ensure randomness.

The feature extraction process is crucial in converting image data into a format suitable for machine learning models. We defined a function, "extract_features", which takes an image path, processes the image (resizes it, converts to grayscale, extracts color RGB histograms, calculates texture features like contrast, skewness, and kurtosis, and performs edge detection), and computes shape features like area, perimeter and circularity based on image segmentation. Additionally, the function get_feature_params is used to analyze the feature lengths across the training data, ensuring consistent extraction. The feature extraction pipeline is validated and adjusted to handle inconsistent feature lengths by padding or truncating features as needed, ensuring that all features have a uniform length.

Once the features are extracted, they are processed and combined with labels to create a final dataset that is ready for model training. Features are extracted for the training, validation, and test datasets, and any "near-zero variance" (NZV) features are removed to improve the model's performance.

The xgboost model is then trained using the filtered dataset, with parameters optimized for multi-class classification. Early stopping is used during training to avoid overfitting.

We will be using xgboost in 'xgboost' package to train our model. This pacakge consists of the following parameters.

1. **nrounds:** the maximum number of iterations
2. **eta:** the learning rate; The higher the eta, the faster the computation
3. **max.depth:** the depth of the tree; The higher the depth of the tree, the higher the chance of overfitting
4. **subsample:** the number of samples (observations) supplied to a tree
5. **colsample_bytree:** the number of features (variables) supplied to a tree
6. **min_child_weight:** minimum weight needed to create a new node in the tree

After training, predictions are made on the test dataset, and a confusion matrix is generated to evaluate the model's performance. The trained model is saved for future use.

5.2.2 Test and train results

```
Confusion Matrix and Statistics

          Reference
Prediction Benign Early Pre Pro
    Benign     66     2   0   0
    Early      10   153   6   0
    Pre         1     2 144   0
    Pro         3     0   3 128

Overall Statistics

               Accuracy : 0.9479
                 95% CI : (0.9251, 0.9654)
    No Information Rate : 0.3031
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9289

 Mcnemar's Test P-Value : NA

Final Test Accuracy: 0.9478764


Final Train Accuracy: 0.9947267
```

Figure 22: Confusion Matrix of XGBoost

In our model, we set nrounds = 1000, eta = 0.1, max.depth = 6, subsample = 0.8, colsample_bytree = 0.8 and min_child_weight = 1. This gives us a training accuracy of 0.9947267 and test accuracy of 0.9478764. As both accuracies are close to each other, it suggests that the model is not overfitted. From the confusion matrix in Figure 43, we can deduce the following observations. XGBoost is able to classify all 'Pro' cases correctly. It is able to do sufficiently well in 'Early' and 'Pre' cases. However, it struggles to classify 'Benign' cases correctly, which is evident in the class having the highest number of false positives among the 4. This has serious consequences for both medical practioners and patients. Patient who have no illness are given medication by a practioner, who has come out with the wrong diagnosis. This could harm the patient's body in the long run. Legal action could be taken against the practioner for the wrong diagnosis, resulting in a lose-lose situation for both parties.

```
Class-Specific Statistics:

             Sensitivity Specificity Pos Pred Value Neg Pred Value Precision
Class: Benign   0.8250000   0.9954338      0.9705882      0.9688889 0.9705882
Class: Early    0.9745223   0.9556787      0.9053254      0.9885387 0.9053254
Class: Pre      0.9411765   0.9917808      0.9795918      0.9757412 0.9795918
Class: Pro      1.0000000   0.9846154      0.9552239      1.0000000 0.9552239
                  Recall        F1 Prevalence Detection Rate
Class: Benign 0.8250000 0.8918919  0.1544402      0.1274131
Class: Early  0.9745223 0.9386503  0.3030888      0.2953668
Class: Pre    0.9411765 0.9600000  0.2953668      0.2779923
Class: Pro    1.0000000 0.9770992  0.2471042      0.2471042
             Detection Prevalence Balanced Accuracy
Class: Benign            0.1312741         0.9102169
Class: Early             0.3262548         0.9651005
Class: Pre               0.2837838         0.9664786
Class: Pro               0.2586873         0.9923077
```

Figure 23: Overall statistics

Nevertheless, XGBoost appears to handle class imbalance well, with high sensitivity, specificity, and balanced accuracy. From Figure 18, the balanced accuracy, which accounts for class imbalances, is also high, particularly for the "Pro" class (99.23%), and other classes show strong results as well (Early: 96.51%, Pre:

96.65%, Benign: 91.02%). Sensitivity and specificity are generally excellent, with the "Pro" class achieving perfect sensitivity (100%) and high specificity (98.46%), while "Benign" class also performs well with 82.5% sensitivity and 99.54% specificity. Precision is good, particularly for the "Pre" (97.96%) and "Benign" (97.06%) classes, though the "Early" class has slightly lower precision (90.53%) and sensitivity (97.45%), suggesting that some "Early" instances may be misclassified as "Benign". The model also excels in Negative Predictive Value, especially for the "Pro" class (100%). The F1 scores, which balance precision and recall, are also high, with "Pro" achieving the highest score (0.9771).

5.2.3 Misclassification cost results

Cost-Weighted Misclassification Matrix:

	Benign	Early	Pre	Pro
Benign	0	20	0	0
Early	100	0	90	0
Pre	20	30	0	0
Pro	90	0	30	0

Total Misclassification Cost: 380

Average Cost Per Sample: 0.7335907

Cost-Sensitive Accuracy: 0.986662

Weighted F1-score 0.9419104

Figure 24: Misclassification cost results, including the cost weighted confusion matrix

The Total misclassification Cost of this model is 380 which gives the overall financial impact of incorrect predictions, calculated by multiplying the frequency of misclassifications by their respective penalties. The average cost per sample of 0.7335907 normalizes this total cost by the number of samples, providing a clearer picture of the cost per misclassified instance. Cost-sensitive accuracy of 0.986662 adjusts the traditional accuracy by factoring in the severity of errors, highlighting the model's ability to minimize costly misclassifications.

Therefore, even though the Benign class is less prevalent, the model maintains strong performance across all classes. This suggests that XGBoost is well-suited for imbalanced datasets and is handling the class distribution in a balanced way.

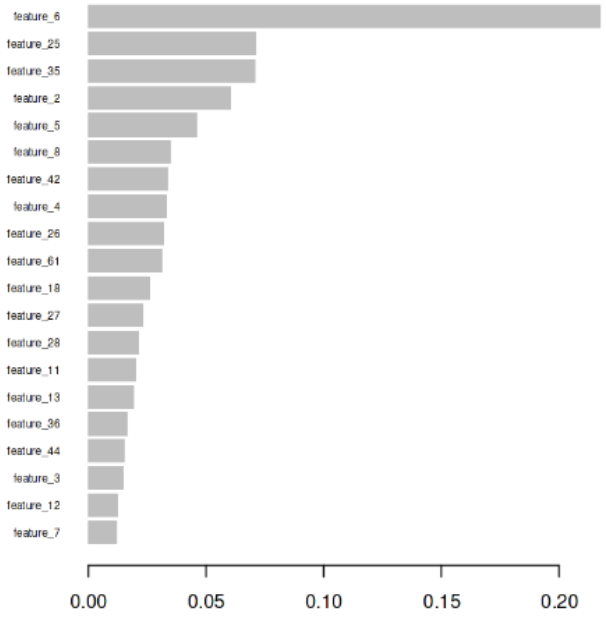


Figure 25: Feature importance of XGBoost

Finally, the top 20 important features are visualized in Figure 45. The top 5 features that are of high importance are feature_6, feature_25, feature_35, feature_2 and feature_5.

6 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of deep learning model designed primarily for processing structured grid-like data, such as images. It is inspired by the human visual system, where different layers in the network learn hierarchical patterns and features from raw pixel data [21]. CNNs are particularly effective in image recognition, object detection, and segmentation tasks because they automatically learn and refine the features needed for these tasks during training, making them highly efficient for visual pattern recognition.

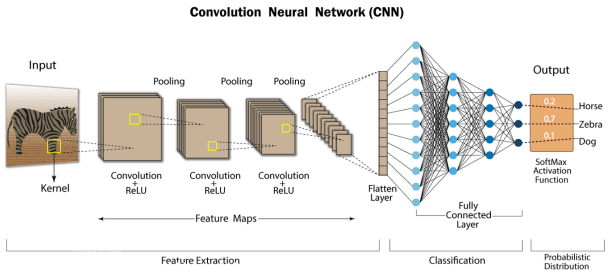


Figure 26: Convolutional Neural Network diagram

A CNN typically consists of multiple layers: convolutional layers, which apply filters to detect various features like edges, textures, and shapes; activation functions, usually ReLU (Rectified Linear Unit), that introduce non-linearity to the network; pooling layers, which downsample feature maps to reduce spatial dimensions and computational load while retaining important information; and fully connected layers as shown in Figure X.X, which interpret the extracted features for classification or other tasks.

6.1 Convolutional Layer

A convolutional layer in a Convolutional Neural Network (CNN) is designed to detect specific features in input data by applying a mathematical operation called convolution. In this layer, a small matrix of weights, known as a filter or kernel, slides over the input image (or previous layers output) to produce feature maps that highlight important patterns like edges, textures, or shapes. The convolution operation itself involves element-wise multiplication between the filter and a corresponding section of the input, followed by summing these products.Mathematically, the convolution operation for a given position can be described as:

$$Z(i,j) = \sum_{n=1}^{n} \sum_{m=1}^{m} X(i+m, j+n) \times K(m,n)$$

where:

- $Z(i,j)$ is the output at position (i,j),
- $X(i+m, j+n)$ is the pixel value from the input image region,
- $K(m,n)$ is the values of the filter.

The values of m and n are determined by the filters dimensions, and the filter is shifted across the input image by a fixed step size known as the stride. This operation captures spatial hierarchies in the input, enabling the network to learn patterns that represent different parts of an object or texture in the image. Multiple filters are usually applied within a convolutional layer, producing different feature maps that capture diverse aspects of the input data, which are then stacked together and passed to subsequent layers. Through training, these filters learn to identify increasingly complex features, making convolutional layers a crucial component for feature extraction in CNNs.

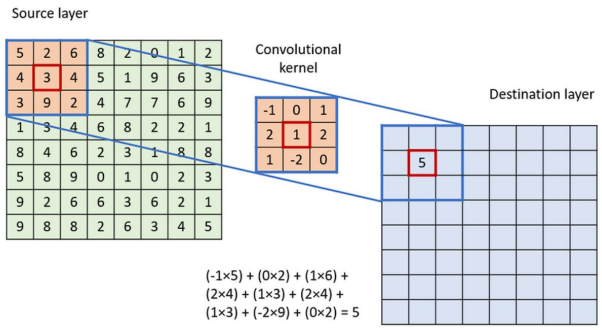


Figure 27: Convolutional operation example

Figure 27 illustrates a two-dimensional convolution operation, where a 3×3 kernel (filter) is applied to a 7×7 input matrix (source layer) to produce an output matrix (destination layer). At each step, a 3×3 sub-matrix is selected from the input matrix, and an element-wise multiplication is performed between the sub-matrix and the kernel. The resulting values are summed to produce a single value, which is placed in the corresponding position of the output matrix. For example, given the sub-matrix

$$\begin{bmatrix} 5 & 2 & 6 \\ 4 & 3 & 4 \\ 3 & 9 & 2 \end{bmatrix}$$

and the kernel

$$\begin{bmatrix} -1 & 0 & 1 \\ 2 & 1 & 2 \\ 1 & -2 & 0 \end{bmatrix},$$

the computation is

$$(-1 \cdot 5) + (0 \cdot 2) + (1 \cdot 6) + (2 \cdot 4) + (1 \cdot 3) + (2 \cdot 4) + (1 \cdot 3) + (-2 \cdot 9) + (0 \cdot 2) = 5.$$

This value is placed in the output matrix. The kernel slides across the input matrix, repeating this process for all valid positions, forming the complete output matrix.

6.2 Pooling Layer

A pooling layer in a Convolutional Neural Network (CNN) is a downsampling operation that reduces the spatial dimensions (height and width) of the input feature map while retaining the most essential information. The primary goal of pooling is to decrease the computational load, reduce the number of parameters, and help the network generalize better by making it more invariant to small translations, rotations, or distortions in the input. The most commonly used pooling methods are max pooling and average pooling. In max pooling, a window or filter slides over the feature map and selects the maximum value within each region (typically 2x2 or 3x3).

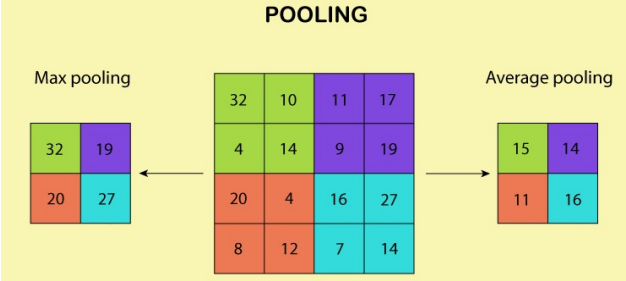


Figure 28: Average and max pooling example

In average pooling, it computes the average value within the same window. By reducing the spatial size, pooling helps make the model more efficient and less prone to overfitting, while also allowing it to focus on the most prominent features. Pooling layers are usually applied after convolutional layers to summarise the extracted features and progressively reduce the dimensionality of the feature maps, thus capturing more abstract patterns as the network deepens.

6.3 Proposed Architecture

6.3.1 Differences between Research Paper's Architecture and Our Proposed Architecture

The proposed architecture that we are introducing is based on the research paper that was done on the Acute Lymphoblastic Leukemia (ALL) image classification dataset that we are currently using [6] . Our architecture will be slightly different, in which it will incorporate a U-Net-like architecture into the segmentation block and a chosen pretrained model for the feature extraction block. The use of U-Net functionalities is a feature absent in the research paper.

For the classification block, the research paper employs two dense sequential blocks with LeakyReLU and ReLU activations, batch normalization, and dropout, culminating in a dense layer with four neurons and Softmax activation for multi-class classification. In contrast, our architecture includes more regularization with L2 penalties, higher dropout rates (0.7), and directly incorporates segmentation-based features into its dense layers, making the classification block more specialized towards leukemia cell images. Finally, while both approaches uses transfer learning, the research paper focuses on fine-tuning the CNNs, while we combined this with segmentation-informed learning.

Below is a table on the summary of key differences between the research paper's and our proposed architecture:

Aspect	Research paper	Our project
Feature Extraction	10 pretrained CNNs (e.g., DenseNet, ResNet, VGG).	Custom U-Net + 3 pretrained CNNs
Segmentation	Preprocessing step, features combined later.	Integrated into the pipeline with U-Net.
Classification Block	Two dense blocks with specific activations.	Dense layers with segmentation + pretrained CNN features and high dropout.
Regularization	Dropout, LeakyReLU, batch normalization.	L2 regularization, higher dropout rates (0.7).
Final Output	Softmax activation for 4 classes.	Softmax activation for the number of classes in `full_df`.

Table 1: Summary of key differences

6.3.2 Segmentation Block

Our architecture starts off with a purple segmentation layer for cancer cell shape identification based on their distinct purple colouration.
The U-Net architecture is a convolutional neural network originally developed for biomedical image segmentation tasks. It has a unique "U" shape formed by a contracting path and an expansive path. In the contracting path, high-level features are captured by successive convolutional and pooling layers, reducing the spatial dimensions while increasing feature depth.

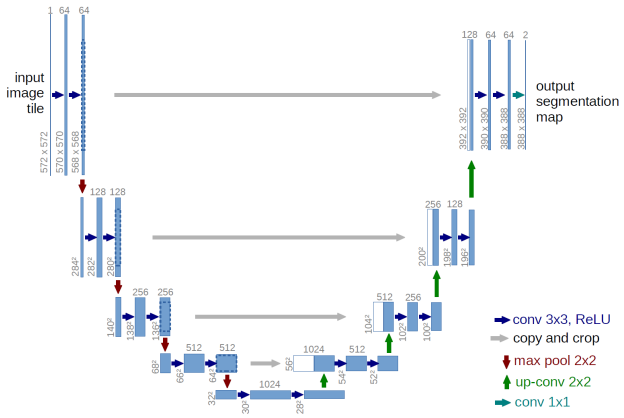


Figure 29: Average and max pooling example

Conversely, the expansive path uses transposed convolutions to upsample these features, reconstructing spatial information while simultaneously combining it with corresponding features from the contracting path through skip connections. These skip connections help retain important spatial context lost during downsampling, which improves the network's ability to accurately segment complex images. This design makes U-Net especially effective in tasks that require precise localization and boundary delineation [12].

6.3.3 Architecture Overview

Figure 30: Proposed architecture flowchart, using ConvNeXt as an example for pretrained model

This model architecture integrates a pretrained base model with a U-Net-inspired segmentation block to perform segmentation and classification tasks. The pretrained model, initialized with ImageNet weights and excluding the top classification layers (include_top = FALSE), acts as a feature extractor, capturing high-level image features.

The architecture starts with a functional layer that processes an input image to segment purple regions corresponding to lymphoblasts, using a color threshold in the HSV (Hue, Saturation, Value) color space with specified lower and upper bounds for the hue, saturation, and value channels. The input image is then converted from RGB to HSV format using TensorFlow's rgb_to_hsv function. A binary mask is created by checking if each pixel in the image falls within the defined purple color range for each of the HSV channels.

This mask is then cast to a float32 type and expanded to match the original image's dimensions. The mask is finally applied to the original image by multiplying the RGB values by the mask, segregating the purple regions from the rest of the image, producing a segmented image where only the purple areas (lymphoblasts) are retained.

The segmentation block is based on the U-Net architecture, widely used in image segmentation. The encoder (contracting path) progressively extracts abstract features through convolutional layers followed by max-pooling. The encoder blocks have 32, 64, and 128 filters in conv1, conv2, and conv3, respectively, with max-pooling reducing spatial resolution. At the bottleneck, two convolutional layers with 256 filters learn abstract features before the decoder begins. The decoder (expansive path) upsamples features using transposed convolutional layers and incorporates skip connections from corresponding encoder layers to retain spatial information. Filters in the decoder decrease from 128 to 64 and 32, refining features at each step and reconstructing the spatial resolution for pixel-wise segmentation.

The pretrained base model also extracts features passed through a global average pooling layer, reducing spatial dimensions to a single feature vector. This vector is flattened and concatenated with the segmentation block's output, combining segmentation and pretrained features. The combined features are processed by the classification block, which consists of fully connected layers with 512, 256, and 128 units, each followed by batch normalization, dropout (0.7), and L2 regularization. The final fully connected layer is a softmax layer with four units (Benign, Early, Pre, Pro), producing class probabilities.

This architecture will be used consistently across all pretrained models in this project.

6.4 VGG16 Architecture

6.4.1 Model description

VGG16 is a popular CNN pre-trained model created by Visual Geometry Group (VGG) primarily used for image classification. It has a total of 16 layers, 13 of which are convolutional and the remaining 3 are fully connected. It utilises 3x3 convolutional filters throughout the network and for down-sampling, has 2x2 max-pooling layers. Our CNN model is built on this and we use a transfer learning approach using VGG16 with the pretrained weights from the Imagenet dataset [22].

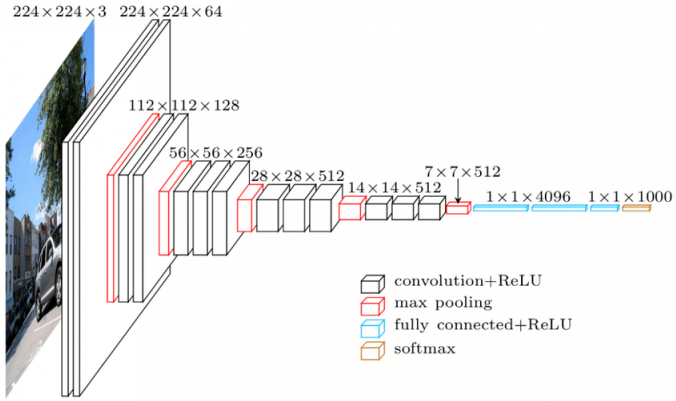


Figure 31: VGG16 architecture diagram

6.4.2 Training process and tuning

Layer (Type)	Output Shape	Param #	Connected to	Trainable
input_2 (InputLayer)	(None, 224, 224, 3)	0	–	Y
tf.image.rgb_to_hsv (TFOpLambda)	(None, 224, 224, 3)	0	input_2[0][0]	Y
tf.__operators__.getitem (SlicingOpLambda)	(None, 224, 224)	0	tf.image.rgb_to_hsv[0][0]	Y
conv2d (Conv2D)	(None, 224, 224, 32)	896	tf.math.multiply[0][0]	Y
conv2d_1 (Conv2D)	(None, 224, 224, 32)	9248	conv2d[0][0]	Y
max_pooling2d (MaxPooling2D)	(None, 112, 112, 32)	0	conv2d_1[0][0]	Y
conv2d_2 (Conv2D)	(None, 112, 112, 64)	18496	max_pooling2d[0][0]	Y
conv2d_3 (Conv2D)	(None, 112, 112, 64)	36928	conv2d_2[0][0]	Y
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 64)	0	conv2d_3[0][0]	Y
conv2d_4 (Conv2D)	(None, 56, 56, 128)	73856	max_pooling2d_1[0][0]	Y
conv2d_5 (Conv2D)	(None, 56, 56, 128)	147584	conv2d_4[0][0]	Y
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 128)	0	conv2d_5[0][0]	Y
conv2d_6 (Conv2D)	(None, 28, 28, 256)	295168	max_pooling2d_2[0][0]	Y
conv2d_7 (Conv2D)	(None, 28, 28, 256)	590080	conv2d_6[0][0]	Y
conv2d_transpose (Conv2DTranspose)	(None, 56, 56, 128)	295040	conv2d_7[0][0]	Y
concatenate (Concatenate)	(None, 56, 56, 256)	0	conv2d_transpose[0][0], conv2d_5[0][0]	Y
dense (Dense)	(None, 512)	205783552	concatenate_3[0][0]	Y
batch_normalization (BatchNormalization)	(None, 512)	2048	dense[0][0]	Y
dropout (Dropout)	(None, 512)	0	batch_normalization[0][0]	Y
dense_3 (Dense)	(None, 4)	516	dropout_2[0][0]	Y

Table 2: Model architecture using VGG16 as pre-trained model

The training process begins by organizing the dataset for model training, starting with the preparation of file paths and corresponding class labels. The dataset is split into training (64%), validation (20%), and test (16%) sets using stratified sampling to ensure each class is proportionally represented. After the initial split, the training set (train_df) is further shuffled randomly. Hyperparameter tuning focuses on optimizing key parameters for model performance. The initial learning rate is set to 1×10^{-5} for stable convergence. Dense layers are regularized with L_2 regularization (0.01) and dropout (0.7) to prevent overfitting. Early stopping mechanism is also implemented with a patience of 8, and using a learning rate plateau scheduler of patience 6, which reduces the learning rate by a factor of 0.1 if the validation loss stalls for 6 epochs. These callbacks help optimize training and prevent overfitting.

Data augmentation, including rotations, shifts, and zooms, is applied only to the training set. Class weights are adjusted based on class distribution to address imbalance, using the same formula from section 3.6. The model is trained for 220 epochs and evaluated using categorical cross-entropy loss and accuracy.

6.4.3 Test and train results

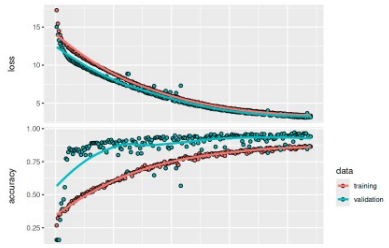


Figure 32: VGG16 loss and accuracy plot

The plots show training (red) and validation (blue) performance over epochs. Training loss decreases steadily, while validation loss stabilizes, indicating limited improvement in generalization. Interestingly, validation accuracy surpasses training accuracy at times, likely due to regularization techniques like dropout or data augmentation. While the model generalizes relatively well, the plateau in validation accuracy suggests potential for further fine-tuning.

```
Confusion Matrix and Statistics

          Reference
Prediction Benign Early Pre Pro
    Benign     68     0   0   0
    Early       8   156  10   0
    Pre         0     0 142   0
    Pro         4     1   1 128

Overall Statistics

               Accuracy : 0.9537
                 95% CI : (0.9318, 0.9701)
    No Information Rate : 0.3031
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9368

 Mcnemar's Test P-Value : NA

      Final Training Loss: 3.34835

      Final Validation Loss: 3.34835

      Final Train Accuracy: 0.8636806

      Final Validation Accuracy: 0.940625


        Test Accuracy: 0.953125
```

Figure 33: Confusion Matrix of VGG16

The VGG16 model has demonstrated strong performance on this dataset, with a test accuracy of 95.31% and a test loss of 3.088493, indicating good generalization to unseen data. The confusion matrix shows that the model has correctly classified a majority of the images, particularly in the "Benign" and "Pro" classes, achieving perfect specificity in both. However, some misclassifications are observed in the "Early" and "Pre" classes, with "Early" images being misclassified as "Benign" and "Pre" images being misclassified as "Early."

The test accuracy (95.37%) is also slightly higher than the train accuracy (86.36%), which shows that there is no signs of overfitting.

```
Class-Specific Statistics:

             Sensitivity Specificity Pos Pred Value Neg Pred Value Precision
Class: Benign   0.8500000   1.0000000      1.0000000      0.9733333 1.0000000
Class: Early    0.9936306   0.9501385      0.8965517      0.9970930 0.8965517
Class: Pre      0.9281046   1.0000000      1.0000000      0.9707447 1.0000000
Class: Pro      1.0000000   0.9846154      0.9552239      1.0000000 0.9552239
                  Recall        F1 Prevalence Detection Rate
Class: Benign 0.8500000 0.9189189  0.1544402      0.1312741
Class: Early  0.9936306 0.9425982  0.3030888      0.3011583
Class: Pre    0.9281046 0.9627119  0.2953668      0.2741313
Class: Pro    1.0000000 0.9770992  0.2471042      0.2471042
             Detection Prevalence Balanced Accuracy
Class: Benign            0.1312741         0.9250000
Class: Early             0.3359073         0.9718845
Class: Pre               0.2741313         0.9640523
Class: Pro               0.2586873         0.9923077
```

Figure 34: VGG16 class specific statistics

The "Benign" class has slightly lower sensitivity (85%) but perfect precision, meaning the model is accurate when predicting Benign but may miss some true Benign cases. The "Early" class has the highest sensitivity (99.36%) but lower precision (89.66%), indicating that the model may falsely predict "Early" in some cases. For the "Pre" class, both sensitivity and specificity are good, and the "Pro" class shows excellent performance with 100% sensitivity and strong precision (95.52%). The model's F1 scores reflect these findings, with the "Pro" class achieving the highest F1 score (97.71%). Thus, with the balanced accuracy, the model is performing exceptionally well in terms of handling class imbalancces, with only minor misclassifications occurring between certain classes.

6.4.4 Misclassification cost

```
Cost-Weighted Misclassification Matrix:

       Benign Early Pre Pro
Benign      0     0   0   0
Early      80     0 150   0
Pre         0     0   0   0
Pro       120    25  10   0

Total Misclassification Cost: 385

Average Cost Per Sample: 0.7432432

Cost-Sensitive Accuracy: 0.9864865

Weighted F1-score 0.9503321
```

Figure 35: VGG16 misclassification cost

The cost-weighted misclassification results highlight the model's strong performance, with a cost-sensitive accuracy of 98.65% and a weighted F1-score of 0.9503. The total misclassification cost of 385 and an average cost per sample of 0.743 reflect the impact of high-cost errors, particularly in misclassifying "Early" and "Pro" classes. While the model effectively minimizes costly mistakes, further fine-tuning with cost-sensitive methods could further reduce the misclassification impact.

6.5 DenseNet-201

6.5.1 Model description

DenseNet-201 is a deep convolutional neural network architecture that builds on the concept of dense connections between layers. Unlike traditional networks, where each layer receives input from only the previous one, DenseNet connects each layer to every other layer in a feed-forward manner. This results in a highly efficient network where each layer has direct access to the gradients from the loss function, enhancing the flow of information and reducing the risk of vanishing gradients. DenseNet-201, specifically, is a version of DenseNet with 201 layers, designed to perform exceptionally well in image classification tasks by leveraging its dense connectivity to improve feature reuse and reduce the number of parameters compared to traditional deep networks [23].

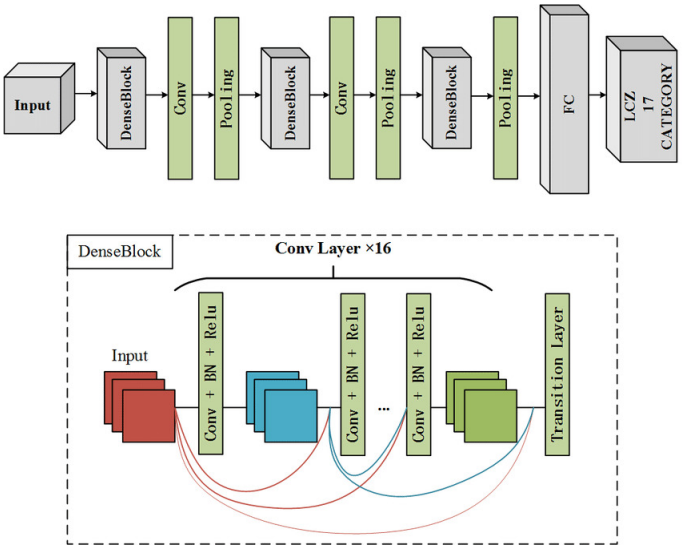


Figure 36: DenseNet-201 diagram

Each dense block is composed of multiple convolutional layers, where each layer receives input from all previous layers within the same block, as well as outputs feature maps that are passed to the subsequent layers. The key feature of DenseNet-201 is its "dense connectivity" pattern, where each layer is connected to every other layer in a feed-forward fashion. This ensures that each layer has access to all previously learned features, enhancing feature reuse and improving gradient flow during training. Between dense blocks, transition layers are used to reduce the feature map size and the number of channels, helping to control the computational complexity. DenseNet-201 is typically characterized by having fewer parameters than traditional deep networks because of the efficient feature reuse. It performs particularly well in tasks like image classification and object recognition by leveraging the dense connections to extract rich, multi-level features from input data.

6.5.2 Training process and tuning

The splitting and shuffling of dataset remains the same as all the other models in this project. For hyperparameter tuning, the learning rate is set to 1×10^{-4} for a quicker convergence. Dense layers are regularized with L_2 regularization (0.01) and dropout (0.7) to prevent overfitting. Unlike VGG16 which requires a higher patience for stable convergence, DenseNet-201 is able to train smoothly with a early stopping patience of 5 and and using a learning rate scheduler of patience 3, which reduces the learning rate by a factor of 0.1 if the validation loss stalls for 3 epochs. Data augmentation, including rotations, shifts, and zooms, are similarly only applied to the training set. The model is also trained for 220 epochs and evaluated using categorical cross-entropy loss and accuracy.

Layer (type)	Output Shape	Param #	Connected to	Trainable
input_2 (InputLayer)	(None, 224, 224, 3)	0	[]	Y
tf.image.rgb_to_hsv (TFOpLambda)	(None, 224, 224, 3)	0	[input_2[0][0]]	Y
tf.__operators__.get_item (SlicingOpLambda)	(None, 224, 224)	0	[tf.image.rgb_to_hsv[0][0]]	Y
tf.__operators__.get_item_1 (SlicingOpLambda)	(None, 224, 224)	0	[tf.image.rgb_to_hsv[0][0]]	Y
tf.math.greater_equal (TFOpLambda)	(None, 224, 224)	0	[tf.__operators__.get_item[0][0]]	Y
tf.math.less_equal (TFOpLambda)	(None, 224, 224)	0	[tf.__operators__.get_item_1[0][0]]	Y
tf.__operators__.get_item_2 (SlicingOpLambda)	(None, 224, 224)	0	[tf.image.rgb_to_hsv[0][0]]	Y
tf.math.logical_and (TFOpLambda)	(None, 224, 224)	0	[tf.math.greater_equal[0][0], tf.math.less_equal[0][0]]	Y
tf.math.greater_equal_1 (TFOpLambda)	(None, 224, 224)	0	[tf.__operators__.get_item_2[0][0]]	Y
tf.__operators__.get_item_3 (SlicingOpLambda)	(None, 224, 224)	0	[tf.image.rgb_to_hsv[0][0]]	Y
tf.math.logical_and_1 (TFOpLambda)	(None, 224, 224)	0	[tf.math.logical_and[0][0], tf.math.greater_equal_1[0][0]]	Y
tf.math.greater_equal_2 (TFOpLambda)	(None, 224, 224)	0	[tf.__operators__.get_item_3[0][0]]	Y
tf.math.logical_and_2 (TFOpLambda)	(None, 224, 224)	0	[tf.math.logical_and_1[0][0], tf.math.greater_equal_2[0][0]]	Y
tf.cast (TFOpLambda)	(None, 224, 224)	0	[tf.math.logical_and_2[0][0]]	Y
tf.expand_dims (TFOpLambda)	(None, 224, 224, 1)	0	[tf.cast[0][0]]	Y
tf.tile (TFOpLambda)	(None, 224, 224, 3)	0	[tf.expand_dims[0][0]]	Y
tf.math.multiply (TFOpLambda)	(None, 224, 224, 3)	0	[input_2[0][0], tf.tile[0][0]]	Y
conv2d (Conv2D)	(None, 224, 224, 32)	896	[tf.math.multiply[0][0]]	Y
conv2d_1 (Conv2D)	(None, 224, 224, 32)	9248	[conv2d[0][0]]	Y
max_pooling2d (MaxPooling2D)	(None, 112, 112, 32)	0	[conv2d_1[0][0]]	Y
conv2d_2 (Conv2D)	(None, 112, 112, 64)	18496	[max_pooling2d[0][0]]	Y
conv2d_3 (Conv2D)	(None, 112, 112, 64)	36928	[conv2d_2[0][0]]	Y
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 64)	0	[conv2d_3[0][0]]	Y
conv2d_4 (Conv2D)	(None, 56, 56, 128)	73856	[max_pooling2d_1[0][0]]	Y
conv2d_5 (Conv2D)	(None, 56, 56, 128)	147584	[conv2d_4[0][0]]	Y
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 128)	0	[conv2d_5[0][0]]	Y
conv2d_6 (Conv2D)	(None, 28, 28, 256)	295168	[max_pooling2d_2[0][0]]	Y
conv2d_7 (Conv2D)	(None, 28, 28, 256)	590080	[conv2d_6[0][0]]	Y
conv2d_transpose (Conv2DTranspose)	(None, 56, 56, 128)	295040	[conv2d_7[0][0]]	Y
concatenate (Concatenate)	(None, 56, 56, 256)	0	[conv2d_transpose[0][0], conv2d_5[0][0]]	Y
conv2d_8 (Conv2D)	(None, 56, 56, 128)	295040	[concatenate[0][0]]	Y
densenet201 (Functional)	(None, 7, 7, 1920)	1832198	[input_2[0][0]]	Y
conv2d_9 (Conv2D)	(None, 56, 56, 128)	147584	[conv2d_8[0][0]]	Y
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1920)	0	[densenet201[0][0]]	Y
flatten (Flatten)	(None, 401408)	0	[conv2d_9[0][0]]	Y
concatenate_3 (Concatenate)	(None, 403328)	0	[global_average_pooling2d[0][0], flatten[0][0]]	Y
dense (Dense)	(None, 512)	2065044	[concatenate_3[0][0]]	Y
batch_normalization (BatchNormalization)	(None, 512)	2048	[dense[0][0]]	Y
dropout (Dropout)	(None, 512)	0	[batch_normalization[0][0]]	Y
dense_1 (Dense)	(None, 256)	131328	[dropout[0][0]]	Y
batch_normalization_1 (BatchNormalization)	(None, 256)	1024	[dense_1[0][0]]	Y
dropout_1 (Dropout)	(None, 256)	0	[batch_normalization_1[0][0]]	Y
dense_2 (Dense)	(None, 1)	257	[dropout_1[0][0]]	Y

Table 3: Model architecture using DenseNet-201

6.5.3 Test and train results

In the loss plot below (Figure 37), both training and validation losses decrease sharply at the beginning and eventually plateau at low values, indicating that the model is learning effectively and generalizing well without significant overfitting. In the accuracy plot, training and validation accuracy both increase rapidly and converge near 1.0, demonstrating strong model performance on both the training and validation datasets. The close alignment of the two curves suggests that the model is not overfitting and generalizes well to unseen data. This is an ideal outcome, reflecting a well-trained model.

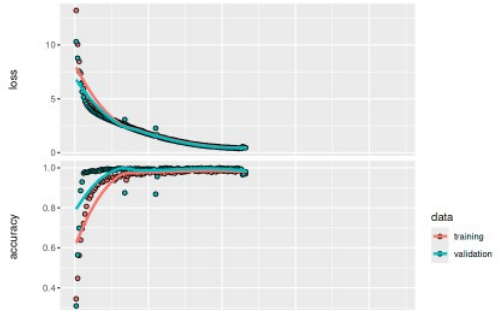


Figure 37: DenseNet201 loss and accuracy plot

```
Confusion Matrix and Statistics

          Reference
Prediction Benign Early Pre Pro
    Benign     79     0   0   0
    Early       1   157   0   0
    Pre         0     0 152   0
    Pro         0     0   1 128

Overall Statistics

               Accuracy : 0.9961
                 95% CI : (0.9861, 0.9995)
    No Information Rate : 0.3031
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9948

 Mcnemar's Test P-Value : NA

  Final Training Loss: 0.4692059

  Final Validation Loss: 0.4692059

  Final Train Accuracy: 0.969815

  Final Validation Accuracy: 0.98125


         Test Accuracy: 0.9960938
```

Figure 38: Confusion Matrix of DenseNet-201

The results in Figure 38 show exceptional performance in terms of classification accuracy and precision. The test accuracy of 99.61% suggests that the model is effectively distinguishing between classes, with minimal misclassification, as evidenced by the confusion matrix, which shows only a few false positives and false negatives. The overall test accuracy (0.9961) is significantly higher than the "No Information Rate" of 0.3031, indicating the model's strong predictive power. The kappa score of 0.9948 indicates excellent agreement between predicted and actual values.

```
Class-Specific Statistics:

             Sensitivity Specificity Pos Pred Value Neg Pred Value Precision
Class: Benign   0.9875000   1.0000000      1.0000000      0.9977221 1.0000000
Class: Early    1.0000000   0.9972299      0.9936709      1.0000000 0.9936709
Class: Pre      0.9934641   1.0000000      1.0000000      0.9972678 1.0000000
Class: Pro      1.0000000   0.9974359      0.9922481      1.0000000 0.9922481
                 Recall        F1 Prevalence Detection Rate
Class: Benign 0.9875000 0.9937107  0.1544402      0.1525097
Class: Early  1.0000000 0.9968254  0.3030888      0.3030888
Class: Pre    0.9934641 0.9967213  0.2953668      0.2934363
Class: Pro    1.0000000 0.9961089  0.2471042      0.2471042
             Detection Prevalence Balanced Accuracy
Class: Benign            0.1525097         0.9937500
Class: Early             0.3050193         0.9986150
Class: Pre               0.2934363         0.9967320
Class: Pro               0.2490347         0.9987179
```

Figure 39: DenseNet201 class-specific statistics

Class-specific metrics further highlight the model's efficiency, with high sensitivity, specificity, and precision across all classes, particularly in detecting "Early" and "Pro" stages, where it achieves 100% sensitivity. The test loss of 0.3849 is relatively low, indicating good model generalization. However, while the model performs exceptionally well in validation, the training and validation losses and accuracies suggest that further tuning may be beneficial for improved generalization.

6.5.4 Misclassification Cost

```
Cost-Weighted Misclassification Matrix:

       Benign Early Pre Pro
Benign      0     0   0   0
Early      10     0   0   0
Pre         0     0   0   0
Pro         0     0  10   0

Total Misclassification Cost: 20

Average Cost Per Sample: 0.03861004

Cost-Sensitive Accuracy: 0.999298

Weighted F1-score 0.9958416
```

Figure 40: DenseNet201 misclassification costs

The misclassification costs of DenseNet-201, as indicated by the cost-weighted matrix, show a total misclassification cost of 20, which is relatively low considering the dataset's size. The misclassifications primarily involve "Early" and "Pro" classes, where "Early" samples were misclassified as "Benign" (10 times), and "Pro" samples were misclassified as "Pre" (10 times). These misclassifications contribute to a small cost in the overall performance. Despite these errors, the model maintains a high cost-sensitive accuracy of 99.93%, indicating that the misclassifications are rare and have minimal impact on the model's overall ability to make correct predictions. The weighted F1-score of 0.9958 reflects the model's ability to balance precision and recall well, even in the presence of some misclassifications. The average cost per sample of 0.0386 further emphasizes that the financial or operational impact of these misclassifications is minimal, though this is much lesser compared to VGG16.

6.6 ConvNeXt

6.6.1 Model description

ConvNeXT (Convolutional Neural Network Transformer) is a recent CNN architecture inspired by vision transformer-based models. It was designed to enhance the performance of CNNs by incorporating design principles by transformers, such as a larger receptive field and more global context. ConvNeXT optimizes CNNs by using techniques like normalized convolutional layers and transformer-inspired block structures while maintaining computational efficiency. This hybrid approach allows ConvNeXT to achieve state-of-the-art performance on image classification tasks, leveraging both the local feature extraction power of CNNs and the long-range dependency modeling of transformers [24].

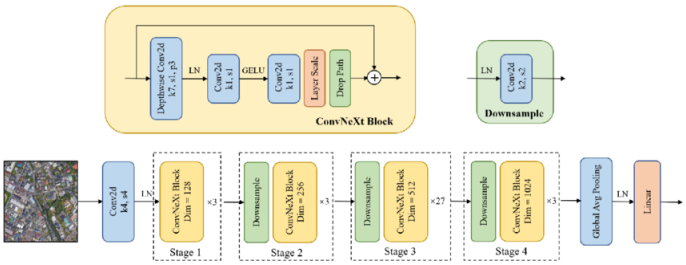


Figure 41: ConvNeXt architecture diagram

6.6.2 Training process and tuning

Layer (type)	Output Shape	Param #	Connected to	Trainable
input_2 (InputLayer)	(None, 224, 224, 3)	0	[]	Yes
tf.image.rgb_to_hsv (TFOpLambda)	(None, 224, 224, 3)	0	input_2[0][0]	Yes
tf.__operators__.getitem	(None, 224, 224)	0	tf.image.rgb_to_hsv[0][0]	Yes
tf.math.greater_equal (TFOpLambda)	(None, 224, 224)	0	tf.__operators__.getitem[0][0]	Yes
tf.math.logical_and (TFOpLambda)	(None, 224, 224)	0	tf.math.greater_equal[0][0]	Yes
tf.math.multiply (TFOpLambda)	(None, 224, 224, 3)	0	input_2[0][0], tf.tile[0][0]	Yes
conv2d (Conv2D)	(None, 224, 224, 32)	896	tf.math.multiply[0][0]	Yes
conv2d_1 (Conv2D)	(None, 224, 224, 32)	9,248	conv2d[0][0]	Yes
max_pooling2d (MaxPooling2D)	(None, 112, 112, 32)	0	conv2d_1[0][0]	Yes
conv2d_2 (Conv2D)	(None, 112, 112, 64)	18,496	max_pooling2d[0][0]	Yes
conv2d_3 (Conv2D)	(None, 112, 112, 64)	36,928	conv2d_2[0][0]	Yes
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 64)	0	conv2d_3[0][0]	Yes
conv2d_4 (Conv2D)	(None, 56, 56, 128)	73,856	max_pooling2d_1[0][0]	Yes
conv2d_5 (Conv2D)	(None, 56, 56, 128)	147,584	conv2d_4[0][0]	Yes
convnext_base (Functional)	(None, 7, 7, 1024)	87,566,464	input_2[0][0]	Yes
global_average_pooling2d	(None, 1024)	0	convnext_base[0][0]	Yes
flatten (Flatten)	(None, 401,408)	0	conv2d_9[0][0]	Yes
concatenate	(None, 402,432)	0	global_average_pooling2d, flatten	Yes
dense (Dense)	(None, 512)	206,045,696	concatenate[0][0]	Yes
batch_normalization	(None, 512)	2,048	dense[0][0]	Yes
dropout	(None, 512)	0	batch_normalization[0][0]	Yes
dense_3 (Dense)	(None, 4)	516	dropout[0][0]	Yes

Table 4: Model architecture using ConvNeXt

The splitting and shuffling of dataset remains the same as all the other models in this project. For hyperparameter tuning, initial learning rate is 1×10^{-4} for a quicker convergence. Layers are similarly regularized with L_2 regularization (0.01) and dropout (0.7) to prevent overfitting. For ConvNeXt, an early stopping patience of 8 and and using a learning rate scheduler of patience 5 is set. The values set for each augmentation property, including rotations, shifts, and zooms, remains the same. The model is also trained for 220 epochs and evaluated using categorical cross-entropy loss and accuracy.

6.6.3 Test and train results

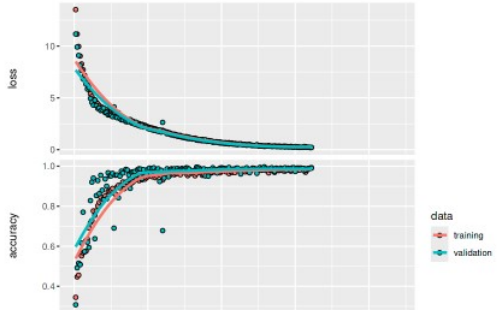


Figure 42: ConvNeXt loss and accuracy plot

The training and validation loss/accuracy plots for the ConvNeXt model reveal an effective and well-optimized training process. The training loss steadily decreases as epochs progress, while training accuracy rises smoothly, converging close to 1 (100%), indicating that the model learns the patterns in the training data effectively. Similarly, the validation loss shows a significant decline during the initial stages of training and stabilizes at a low value, while the validation accuracy increases and converges near 1. The close alignment between the training and validation curves for both loss and accuracy indicates that the model generalizes well to unseen data, with minimal signs of overfitting or underfitting.

```
Test Loss: 0.2232956

Test Accuracy: 0.9941406

Confusion Matrix and Statistics

          Reference
Prediction Benign Early Pre Pro
    Benign     80     0   0   0
    Early       0   156   1   0
    Pre         0     1 151   0
    Pro         0     0   1 128

Overall Statistics

               Accuracy : 0.9942
                 95% CI : (0.9832, 0.9988)
    No Information Rate : 0.3031
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9921

 Mcnemar's Test P-Value : NA

Final Training Loss: 0.2005041

Final Validation Loss: 0.2005041

Final Train Accuracy: 0.9926972

Final Validation Accuracy: 0.990625
```

Figure 43: Confusion Matrix of ConvNeXt

The ConvNeXt model demonstrates outstanding performance, achieving a test accuracy of 99.41% with a test loss of 0.2233, indicating minimal errors on unseen data. The confusion matrix highlights that the model classifies most instances correctly across all four classes, with only a few misclassifications: one "Early" instance predicted as "Pre," one "Pre" instance predicted as "Early," and another as "Pro".

```
Class-Specific Statistics:

             Sensitivity Specificity Pos Pred Value Neg Pred Value Precision
Class: Benign   1.0000000   1.0000000      1.0000000      1.0000000 1.0000000
Class: Early    0.9936306   0.9972299      0.9936306      0.9972299 0.9936306
Class: Pre      0.9869281   0.9972603      0.9934211      0.9945355 0.9934211
Class: Pro      1.0000000   0.9974359      0.9922481      1.0000000 0.9922481
                  Recall        F1 Prevalence Detection Rate
Class: Benign 1.0000000 1.0000000  0.1544402      0.1544402
Class: Early  0.9936306 0.9936306  0.3030888      0.3011583
Class: Pre    0.9869281 0.9901639  0.2953668      0.2915058
Class: Pro    1.0000000 0.9961089  0.2471042      0.2471042
             Detection Prevalence Balanced Accuracy
Class: Benign            0.1544402         1.0000000
Class: Early             0.3030888         0.9954302
Class: Pre               0.2934363         0.9920942
Class: Pro               0.2490347         0.9987179
```

Figure 44: ConvNeXt class-specific statistics

Class-specific metrics reveal excellent balance across all categories, indicating that the class weighting formula is able to work well with ConvNeXt model in rebalancing the dataset. Sensitivity and specificity values are close to 1 for each class, affirming the model's ability to identify true positives and avoid false positives with minimal bias. Precision values for all classes are similarly high, leading to F1-scores nearing 1, which confirms the models strong and consistent performance in handling the multi-class classification task. Despite the exceptional results, minor misclassifications between "Early" and "Pre" may indicate feature overlap, warranting further exploration.

6.6.4 Misclassification Costs

```
Cost-Weighted Misclassification Matrix:

       Benign Early Pre Pro
Benign      0     0   0   0
Early       0     0  15   0
Pre         0    15   0   0
Pro         0     0  10   0

Total Misclassification Cost: 40

Average Cost Per Sample: 0.07722008

Cost-Sensitive Accuracy: 0.998596

Weighted F1-score 0.9949759
```

Figure 45: ConvNeXt misclassification costs

The total misclassification cost, calculated by summing these values, is 40, which is slightly higher than that of the DenseNet-201 model (20) and much lower than VGG16 model (385). The average cost per sample is derived by dividing the total cost by the number of samples, yielding 0.077, indicating that, on average, each sample contributes a relatively low penalty for misclassification.

The cost-sensitive accuracy, which adjusts the traditional accuracy by factoring in the misclassification costs, is 0.9986. This indicates that the model performs exceptionally well in minimizing the cost impact of its errors, compared to VGG16 model, and slightly worse than DenseNet-201.

7 Results and Findings

Below is a summary of the train accuracy, test accuracy, and their respective misclassification cost metrics:

Model	Train Accuracy	Test Accuracy	Total Misclassification Cost	Average Cost Per Sample	Cost-Sensitive Accuracy	Weighted F1-Score
SVM	0.9971	0.9517	345	0.6660	0.9879	0.9448
XGBoost	0.9947	0.9479	380	0.7336	0.9867	0.9419
VGG16	0.8637	0.9531	385	0.7432	0.9865	0.9503
DenseNet-201	0.9698	0.9961	20	0.0386	0.9993	0.9958
ConvNeXt	0.9927	0.9941	40	0.0772	0.9986	0.9950

Table 5: Performance Metrics of Different Models

From the table, SVM model, with its training accuracy of 99.71% and test accuracy of 95.17%, provides a strong fit for the training data but suffers from a relatively high total misclassification cost (345). This suggests that while SVM can generalize well to some extent, it struggles with certain subtypes, particularly the "Early" and "Pre" classes, as reflected by its cost-sensitive accuracy of 98.79% and weighted F1-score of 94.48%. The higher misclassification cost in SVM indicates that, in a clinical setting, the cost of incorrect diagnosis could be significant, especially in cases where early-stage detection is critical.

XGBoost performs similarly well with a test accuracy of 94.79%, but its total misclassification cost (380) is higher than SVM, indicating that while it also provides a decent overall performance, it may be prone to misclassifying the "Pre" and "Early" classes. The cost-sensitive accuracy (98.67%) and weighted F1-score (94.19%) are lower than those of SVM, further suggesting that XGBoost struggles more with the delicate balance required in accurately diagnosing early and pre-leukemia stages. VGG16, despite its high test accuracy of 95.31%, shows lower training accuracy (86.37%) and a significantly higher misclassification cost (385). This higher cost, coupled with an average cost per sample of 0.74, suggests that VGG16s performance could be problematic in real-world scenarios where precise and cost-efficient classification is paramount.

DenseNet-201, on the other hand, outperforms the other models with a test accuracy of 99.61%, the lowest total misclassification cost (20), and the lowest average cost per sample (0.04). With a cost-sensitive accuracy of 99.93% and a weighted F1-score of 99.58%, DenseNet-201 stands out as the most efficient and reliable model for leukemia classification, particularly when minimizing the risks of misclassification is crucial in clinical decision-making. ConvNeXt, though not as efficient as DenseNet-201, offers a similarly high test accuracy of 99.41%, with a moderate misclassification cost (40) and a strong cost-sensitive accuracy of 99.86%. While ConvNeXt provides excellent results, its slightly higher misclassification cost indicates that DenseNet-201 is more optimal for real-world applications where accurate and cost-effective classification of leukemia subtypes is critical to patient care.

8 Conclusion

In conclusion, DenseNet-201 stands out as the most reliable model for leukemia subtype classification, offering the highest test accuracy, the lowest misclassification cost, and best cost-sensitive metrics. This makes it the most suitable for clinical applications where precision in identifying leukemia stages is critical. However, while DenseNet-201 outperforms other models, it is important to note that even the best models exhibit trade-offs in terms of misclassification, particularly for early-stage subtypes.

For the limitations of the methods used, deep learning models, such as DenseNet-201 and ConvNeXt, are often considered "black-box" models. Their complex architectures make it difficult for clinicians to understand the reasoning behind the classification decisions, which could hinder trust and adoption in medical practice. In healthcare, interpretability is crucial for ensuring clinical decisions are transparent and justifiable.

Furthermore, the misclassification cost metrics used in our project were based on our subjective judgment, which means they were designed according to our perceived severity of misclassifications rather than a rigorous, scientifically grounded framework. While this approach aligns with our understanding of the potential clinical impact of misclassifying leukemia subtypes, it lacks objective validation or a consensus from medical experts regarding the relative costs of these errors. Therefore, consultation with medical professionals to define the misclassification costs based on clinical significance, or a data-driven method would help give more accurate and corroborated misclasssification cost metrics.

Also, there is a limitation of using such dataset for data mining and deep learning purposes. The dataset consists of 3,256 smaples which is small relative to the number of skin cancer cases worldwide. This might mean that the results shown in this report might not be indicative of its actual performance in real life. Further research in this area of study is needed to ensure that the risks of misdiagnosis are mitigated.

References

[1] Nan Zhang, Jinxian Wu, Qian Wang, Yuxing Liang, Xinqi Li, Guopeng Chen, Linlu Ma, Xiaoyan Liu, and Fuling Zhou. Global burden of hematologic malignancies and evolution patterns over the past 30 years. *Blood Cancer Journal*, 13(1):82, 2023.

[2] World Health Organization et al. Early cancer diagnosis saves lives, cuts treatment costs. *World Health Organization*, 2017.

[3] Tibor Mezei, Melinda Kolcsár, András Joó, and Simona Gurzu. Image analysis in histopathology and cytopathology: From early days to current perspectives. *Journal of Imaging*, 10(10):252, 2024.

[4] Ahmed J Abougarair, M Alshaibi, Amany K Alarbish, Mohammed Ali Qasem, Doaa Abdo, Othman Qasem, Fursan Thabit, and Ozgu Can. Blood cells cancer detection based on deep learning. *Journal of Advances in Artificial Intelligence*, 2(1):108–121, 2024.

[5] Amjad Rehman, Naveed Abbas, Tanzila Saba, Syed Ijaz ur Rahman, Zahid Mehmood, and Hoshang Kolivand. Classification of acute lymphoblastic leukemia using deep learning. *Microscopy Research and Technique*, 81(11):1310–1317, 2018.

[6] Mustafa Ghaderzadeh, Mehrad Aria, Azamossadat Hosseini, Farkhondeh Asadi, Davood Bashash, and Hassan Abolghasemi. A fast and efficient cnn model for b-all diagnosis and its subtypes classification using peripheral blood smear images. *International Journal of Intelligent Systems*, 37(8):5113–5133, 2022.

[7] Mehmet Erten, Prabal Datta Barua, Sengul Dogan, Turker Tuncer, Ru-San Tan, and UR Acharya. Concatnext: An automated blood cell classification with a new deep convolutional neural network. *Multimedia Tools and Applications*, pages 1–19, 2024.

[8] Murray H Loew. Feature extraction. *Handbook of medical imaging*, 2:273–342, 2000.

[9] Scott E Umbaugh. *Digital image processing and analysis: computer vision and image analysis*. CRC Press, 2023.

[10] David A Forsyth and Jean Ponce. *Computer vision: a modern approach*. prentice hall professional technical reference, 2002.

[11] Dengsheng Zhang and Guojun Lu. Review of shape representation and description techniques. *Pattern recognition*, 37(1):1–19, 2004.

[12] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

[13] Mingle Xu, Sook Yoon, Alvaro Fuentes, and Dong Sun Park. A comprehensive survey of image augmentation techniques for deep learning. *Pattern Recognition*, 137:109347, 2023.

[14] Ravindra Singh, Naurang Singh Mangat, Ravindra Singh, and Naurang Singh Mangat. Stratified sampling. *Elements of survey sampling*, pages 102–144, 1996.

[15] Corinna Cortes. Support-vector networks. *Machine Learning*, 1995.

[16] GeeksforGeeks. Difference between random forest vs xgboost, 2024. Accessed: 2024-11-18.

[17] Saiwa.ai. Xgboost in machine learning: Everything you need to know, 2024. Accessed: 2024-11-18.

[18] Guestrin Chen. Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[19] Random Realizations. Xgboost explained, 2024. Accessed: 2024-11-18.

[20] The AI Edge. Understanding xgboost: From a to z, 2024. Accessed: 2024-11-19.

[21] Ragav Venkatesan and Baoxin Li. *Convolutional neural networks in visual computing: a concise guide*. CRC Press, 2017.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[23] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[24] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11976–11986, 2022.

9 Appendices

9.1 SVM

```
# Load necessary libraries
library(keras)
library(tensorflow)
library(dplyr) # For data manipulation
library(imager)
library(caret)
library(e1071)
library(ggplot2)
library(EBImage)  # For bwlabel and shape feature extraction

# Set the path to your dataset
data_dir <- "/kaggle/input/leukemia-images/Original"

# Define image size and parameters
img_height <- 150
img_width <- 150
batch_size <- 32

# Create a list to hold all file paths and corresponding labels
file_paths <- c()
class_labels <- c()

```

```
# Loop through each class to gather file paths and labels
for (class in c("Benign", "Early", "Pre", "Pro")) {
  class_path <- file.path(data_dir, class)
  files <- list.files(class_path, full.names = TRUE)

  # Ensure files are unique
  unique_files <- unique(files)

  file_paths <- c(file_paths, unique_files)
  class_labels <- c(class_labels, rep(class, length(unique_files)))  # Store the
    class name
}

# Create a DataFrame with only the required structure
full_df <- data.frame(
  image_filename = file_paths,
  class = class_labels,
  stringsAsFactors = FALSE
)

head(full_df)

# Define class-to-index mapping
classes <- c("Benign", "Early", "Pre", "Pro")
class_to_index <- setNames(0:(length(classes) - 1), classes)

# Add numeric columns to the full and training DataFrames
full_df$class_numeric <- as.numeric(factor(full_df$class, levels = classes)) - 1

# Stratified split into training (64%), validation (20%), and test (16%) sets
set.seed(123)
trainIndex <- createDataPartition(full_df$class, p = 0.64, list = FALSE)
train_df <- full_df[trainIndex, ]
remaining_df <- full_df[-trainIndex, ]

# Shuffle only train_df
train_df <- train_df[sample(nrow(train_df)), ]

# Split the remaining 36% into validation (20%) and test (16%)
validationIndex <- createDataPartition(remaining_df$class, p = 20 / (20 + 16),
    list = FALSE)
val_df <- remaining_df[validationIndex, ]
test_df <- remaining_df[-validationIndex, ]

head(train_df)
head(val_df)
head(test_df)

# Calculate class weights using only the training dataset
n_samples <- nrow(train_df)
n_classes <- length(unique(train_df$class))

# Get class frequencies from training dataset
class_counts <- table(train_df$class_numeric)

# Calculate balanced weights
class_weights <- n_samples / (n_classes * class_counts)

# Convert to a named list with numeric indices as names
```

```
class_weights <- as.list(class_weights)
names(class_weights) <- as.character(0:(n_classes-1))

# Print diagnostics
cat("\nClass distribution in training dataset:\n")
print(table(train_df$class))

cat("\nFinal Class Weights (based on training dataset):\n")
print(class_weights)

# Validation steps
cat("\nValidation:\n")
cat("Total samples in training dataset:", n_samples, "\n")
cat("Sum of class counts:", sum(class_counts), "\n")
cat("Number of classes:", n_classes, "\n")
cat("Maximum weight ratio:", max(unlist(class_weights)) /
    min(unlist(class_weights)), "\n\n")

# Additional validation checks
cat("Verification of weight calculation:\n")
for(i in 0:(n_classes-1)) {
  class_name <- classes[i+1]
  weight <- class_weights[[as.character(i)]]
  count <- class_counts[as.character(i)]
  cat(sprintf("%s: count=%d, weight=%.4f, count*weight=%.4f\n",
              class_name, count, weight, count*weight))
}

# Features extraction function
extract_features <- function(image_path, bins = 16, feature_params = NULL) {
  # Load and resize the image using imager
  image <- load.image(image_path)
  image <- resize(image, img_width, img_height)

  # Convert image to array and split into color channels
  red_channel <- as.vector(image[,,1,1])
  green_channel <- as.vector(image[,,1,2])
  blue_channel <- as.vector(image[,,1,3])

  # Color histogram features for each channel
  red_hist <- hist(red_channel, breaks = bins, plot = FALSE)$counts /
    length(red_channel)
  green_hist <- hist(green_channel, breaks = bins, plot = FALSE)$counts /
    length(green_channel)
  blue_hist <- hist(blue_channel, breaks = bins, plot = FALSE)$counts /
    length(blue_channel)

  # Convert to grayscale for texture and edge features
  gray_image <- 0.299 * red_channel + 0.587 * green_channel + 0.114 * blue_channel

  # Texture features: contrast, skewness, and kurtosis
  contrast <- var(gray_image)
  skewness <- e1071::skewness(gray_image)
  kurtosis <- e1071::kurtosis(gray_image)

  # Statistical descriptors
  mean_intensity <- mean(gray_image)
  sd_intensity <- sd(gray_image)
  max_intensity <- max(gray_image)
```

```
  min_intensity <- min(gray_image)

  # Edge detection using gradient magnitude approximation (Sobel-like)
  gray_img_cimg <- as.cimg(gray_image, x = img_width, y = img_height)
  grad_x <- imgradient(gray_img_cimg, "x")
  grad_y <- imgradient(gray_img_cimg, "y")
  edge_magnitude <- sqrt(grad_x^2 + grad_y^2)  # Approximate edge strength
  edge_intensity_mean <- mean(edge_magnitude)
  edge_intensity_sd <- sd(edge_magnitude)

  # Shape features using segmentation (using EBImage)
  gray_img_matrix <- matrix(gray_image, nrow = img_height, ncol = img_width)
  binary_image <- gray_img_matrix > 0.5  # Apply threshold to get binary mask
  binary_image <- as.Image(binary_image)  # Convert to EBImage object

  # Label the binary image
  labeled_image <- bwlabel(binary_image)

  # Compute shape features
  shape_features <- computeFeatures.shape(labeled_image)

  # Shape descriptors - Mean values of area, perimeter, and circularity
  area_mean <- mean(shape_features[,"s.area"], na.rm = TRUE)
  perimeter_mean <- mean(shape_features[,"s.perimeter"], na.rm = TRUE)
  circularity_mean <- mean((4 * pi * shape_features[,"s.area"]) /
    (shape_features[,"s.perimeter"]^2), na.rm = TRUE)

  # Combine all features into a single vector
  feature_vector <- c(red_hist, green_hist, blue_hist, contrast, skewness,
    kurtosis,
                      mean_intensity, sd_intensity, max_intensity, min_intensity,
                      edge_intensity_mean, edge_intensity_sd,
                      area_mean, perimeter_mean, circularity_mean)

  return(feature_vector)
}

# Function to analyze feature lengths across training data
get_feature_params <- function(train_image_paths, sample_size = NULL) {
  # If sample_size is provided, randomly sample images for efficiency
  if (!is.null(sample_size) && sample_size < length(train_image_paths)) {
    set.seed(123)  # for reproducibility
    image_paths <- sample(train_image_paths, sample_size)
  } else {
    image_paths <- train_image_paths
  }

  cat("Analyzing features from", length(image_paths), "training images...\n")

  # Extract features from all training images and analyze
  feature_lengths <- sapply(image_paths, function(path) {
    tryCatch({
      features <- extract_features(path)
      length(features)
    }, error = function(e) {
      warning(sprintf("Error processing image %s: %s", path, e$message))
      return(NA)
    })
  })
```

```
192
193   # Analyze feature length distribution
194   length_summary <- summary(feature_lengths)
195   length_table <- table(feature_lengths)
196
197   # Check for inconsistencies
198   if (length(unique(feature_lengths[!is.na(feature_lengths)])) > 1) {
199     warning("Inconsistent feature lengths detected in training data!")
200     print(length_table)
201     print(length_summary)
202   }
203
204   # Get most common feature length (mode)
205   modal_length <- as.numeric(names(length_table)[which.max(length_table)])
206
207   # Get feature names from a successful extraction
208   sample_features <- NULL
209   for (path in image_paths) {
210     tryCatch({
211       sample_features <- extract_features(path)
212       if (length(sample_features) == modal_length) break
213     }, error = function(e) {
214       next
215     })
216   }
217
218   params <- list(
219     feature_length = modal_length,
220     feature_names = paste0("feature_", seq_len(modal_length)),
221     length_distribution = length_table,
222     length_summary = length_summary,
223     sample_size = length(image_paths),
224     success_rate = sum(!is.na(feature_lengths)) / length(feature_lengths),
225     consistent_length = length(unique(feature_lengths[!is.na(feature_lengths)]))
      == 1
226   )
227
228   # Print summary
229   cat("\nFeature extraction analysis summary:\n")
230   cat("Modal feature length:", params$feature_length, "\n")
231   cat("Success rate:", sprintf("%.2f%%", params$success_rate * 100), "\n")
232   cat("Length consistency:", ifelse(params$consistent_length, "Consistent",
       "Inconsistent"), "\n")
233
234   if (!params$consistent_length) {
235     cat("\nFeature length distribution:\n")
236     print(params$length_distribution)
237   }
238
239   return(params)
240 }
241
242 # Modified feature extraction pipeline
243 set.seed(123)  # for reproducibility
244
245 # Step 1: Get feature parameters from training data
246 # Option to sample if training set is very large
247 sample_size <- if(nrow(train_df) > 1000) 1000 else NULL
248 feature_params <- get_feature_params(train_df$image_filename, sample_size)
```

```
249
250 # Enhanced feature extraction function with validation
251 extract_and_validate_features <- function(image_paths, feature_params,
        dataset_name) {
252   features_list <- lapply(seq_along(image_paths), function(i) {
253     path <- image_paths[i]
254
255     if (i %% 100 == 0) {
256       cat(sprintf("Processing %s image %d of %d\n",
257                   dataset_name, i, length(image_paths)))
258     }
259
260     tryCatch({
261       features <- extract_features(path)
262
263       # Validate feature length
264       if (length(features) != feature_params$feature_length) {
265
266         # Handle inconsistent length
267         if (length(features) < feature_params$feature_length) {
268           # Pad with NA if shorter
269           features <- c(features, rep(NA, feature_params$feature_length -
        length(features)))
270         } else {
271           # Truncate if longer
272           features <- features[1:feature_params$feature_length]
273         }
274       }
275
276       return(features)
277
278     }, error = function(e) {
279       warning(sprintf("Error processing %s image %s: %s",
280                       dataset_name, path, e$message))
281       return(rep(NA, feature_params$feature_length))
282     })
283   })
284
285   # Convert to matrix/data frame
286   features_matrix <- do.call(rbind, features_list)
287   colnames(features_matrix) <- feature_params$feature_names
288
289   # Report statistics about NA values
290   na_counts <- colSums(is.na(features_matrix))
291   if (any(na_counts > 0)) {
292     cat("\nNA counts in", dataset_name, "dataset:\n")
293     print(na_counts[na_counts > 0])
294   }
295
296   return(features_matrix)
297 }
298
299 # Extract features for each dataset
300 cat("\nExtracting training features...\n")
301 train_features <- extract_and_validate_features(
302   train_df$image_filename,
303   feature_params,
304   "training"
305 )
```

```
306
307 cat("\nExtracting validation features...\n")
308 val_features <- extract_and_validate_features(
309   val_df$image_filename,
310   feature_params,
311   "validation"
312 )
313
314 cat("\nExtracting test features...\n")
315 test_features <- extract_and_validate_features(
316   test_df$image_filename,
317   feature_params,
318   "test"
319 )
320
321 # Ensure that class labels are factors with consistent levels
322 train_df$class <- factor(train_df$class, levels = c("Benign", "Early", "Pre",
      "Pro"))
323 test_df$class <- factor(test_df$class, levels = c("Benign", "Early", "Pre",
      "Pro"))
324 val_df$class <- factor(val_df$class, levels = c("Benign", "Early", "Pre", "Pro"))
325
326 train_data <- cbind(as.data.frame(train_features), class_numeric =
      train_df$class_numeric)
327 validation_data <- cbind(as.data.frame(val_features), class_numeric =
      val_df$class_numeric)
328 test_data <- cbind(as.data.frame(test_features), class_numeric =
      test_df$class_numeric)
329
330 train_data[is.na(train_data)] <- 0
331 validation_data[is.na(validation_data)] <- 0
332 test_data[is.na(test_data)] <- 0
333
334 # Standardize the features
335 preprocess_params <- preProcess(train_data[, -ncol(train_data)], method =
      c("center", "scale"))
336 train_data_scaled <- predict(preprocess_params, train_data)
337 validation_data_scaled <- predict(preprocess_params, validation_data)
338 test_data_scaled <- predict(preprocess_params, test_data)
339
340 # Convert class_numeric to factor with proper labels for training and testing
341 train_data_scaled$class <- factor(train_data_scaled$class_numeric,
342                                   levels = 0:3,
343                                   labels = classes)
344 validation_data_scaled$class <- factor(validation_data_scaled$class_numeric,
345                                        levels = 0:3,
346                                        labels = classes)
347 test_data_scaled$class <- factor(test_data_scaled$class_numeric,
348                                  levels = 0:3,
349                                  labels = classes)
350
351 library(caret)
352 library(e1071)
353
354
355 # Function to identify NZV features from training data
356 identify_nzv_features <- function(train_data) {
357   nzv <- nearZeroVar(train_data, saveMetrics = TRUE)
358   nzv_features <- rownames(nzv)[nzv$zeroVar | nzv$nzv]
```

```
359
360   # Log the features being removed and their properties
361   cat("Features removed due to near-zero variance:\n")
362   for(feature in nzv_features) {
363     cat(sprintf("Feature: %s, Variance: %.6f\n",
364                 feature,
365                 var(train_data[[feature]], na.rm = TRUE)))
366   }
367
368   return(nzv_features)
369 }
370
371 # Function to check for differences in feature variance between datasets
372 check_feature_variance <- function(train_data, val_data, test_data, nzv_features)
      {
373   warning_features <- list()
374
375   for(feature in nzv_features) {
376     if(feature %in% names(val_data) && feature %in% names(test_data)) {
377       train_var <- var(train_data[[feature]], na.rm = TRUE)
378       val_var <- var(val_data[[feature]], na.rm = TRUE)
379       test_var <- var(test_data[[feature]], na.rm = TRUE)
380
381       # Check if feature has significant variance in validation or test
382       if(val_var > 0.1 || test_var > 0.1) {  # Threshold can be adjusted
383         warning_features[[feature]] <- list(
384           train_variance = train_var,
385           val_variance = val_var,
386           test_variance = test_var
387         )
388       }
389     }
390   }
391
392   return(warning_features)
393 }
394
395 # Apply the fixed NZV filtering process
396 # First, identify NZV features from training data only
397 nzv_features <- identify_nzv_features(train_data_scaled[,
      -ncol(train_data_scaled)])
398
399 # Check for potential issues in validation and test sets
400 variance_warnings <- check_feature_variance(
401   train_data_scaled[, -ncol(train_data_scaled)],
402   validation_data_scaled[, -ncol(validation_data_scaled)],
403   test_data_scaled[, -ncol(test_data_scaled)],
404   nzv_features
405 )
406
407 # Print warnings if any features show different behavior in val/test
408 if(length(variance_warnings) > 0) {
409   cat("\nWARNING: Some features show different variance patterns in
        validation/test sets:\n")
410   print(variance_warnings)
411   cat("\nConsider keeping these features in the model.\n")
412 }
413
414 # Remove NZV features from all datasets
```

```
train_data_filtered <- train_data_scaled[, !names(train_data_scaled) %in%
    nzv_features]
validation_data_filtered <- validation_data_scaled[,
    !names(validation_data_scaled) %in% nzv_features]
test_data_filtered <- test_data_scaled[, !names(test_data_scaled) %in%
    nzv_features]

head(train_data_filtered)
head(validation_data_filtered)
head(test_data_filtered)

# Set the number of folds for cross-validation
num_folds <- 5

# Create a 5-fold cross-validation partition for the training set
set.seed(123)  # For reproducibility
folds <- createFolds(train_data_filtered$class, k = num_folds, list = TRUE)

# Define the tune grid for SVM hyperparameters (sigma and C)
tune_grid <- expand.grid(
  sigma = 10^(-3:3),
  C = 10^(-3:3)
)

# Store performance results for each fold
cv_results <- data.frame(
  fold = integer(),
  train_accuracy = numeric(),
  valid_accuracy = numeric(),
  sigma = numeric(),
  C = numeric(),
  stringsAsFactors = FALSE
)

# Loop through each fold to perform training and validation
for (fold_idx in 1:num_folds) {
  cat("Training fold", fold_idx, "of", num_folds, "\n")

  # Get the training data for the current fold
  train_indices <- unlist(folds[[fold_idx]])
  train_fold <- train_data_filtered[train_indices, ]

  # Create class weights for the training fold
  class_weights_fold <- sapply(train_fold$class_numeric,
                               function(x) class_weights[[as.character(x)]])

  # Initialize tracking for best model
  best_valid_accuracy <- -Inf
  best_train_accuracy <- -Inf
  best_svm_model <- NULL
  best_sigma <- NULL
  best_C <- NULL

  # Grid search over hyperparameters
  for (sigma_val in tune_grid$sigma) {
    for (C_val in tune_grid$C) {
      # Train the SVM model
      tryCatch({
        model <- svm(
```

```
          class ~ .,
          data = train_fold[, !names(train_fold) %in% c("class_numeric")],
          kernel = "radial",
          cost = C_val,
          gamma = sigma_val,
          weights = class_weights_fold,
          probability = TRUE,
          scale = FALSE  # Disable internal scaling since data is already scaled
        )

        # Make predictions on the external validation set
        valid_predictions <- predict(model, validation_data_filtered)
        valid_accuracy <- confusionMatrix(valid_predictions,
    validation_data_filtered$class)$overall['Accuracy']

        # Make predictions on training fold
        train_predictions <- predict(model, train_fold)
        train_accuracy <- confusionMatrix(train_predictions,
                                          train_fold$class)$overall['Accuracy']

        # Track the best model based on validation accuracy
        if (valid_accuracy > best_valid_accuracy) {
          best_valid_accuracy <- valid_accuracy
          best_train_accuracy <- train_accuracy
          best_svm_model <- model
          best_sigma <- sigma_val
          best_C <- C_val
        }
      }, error = function(e) {
        cat("Error with sigma =", sigma_val, "and C =", C_val, ":", e$message,
    "\n")
      })
    }
  }

  # Store the results for the current fold
  cv_results <- rbind(cv_results, data.frame(
    fold = fold_idx,
    train_accuracy = best_train_accuracy,
    valid_accuracy = best_valid_accuracy,
    sigma = best_sigma,
    C = best_C,
    stringsAsFactors = FALSE
  ))

  cat("Best model for fold", fold_idx, ":\n",
      "sigma =", best_sigma, "\n",
      "C =", best_C, "\n",
      "Training accuracy:", best_train_accuracy, "\n",
      "Validation accuracy:", best_valid_accuracy, "\n\n")
}

# Print cross-validation results summary
cat("Cross-validation results:\n")
print(cv_results)

# Calculate summary statistics
mean_train_accuracy <- mean(cv_results$train_accuracy)
```

```
sd_train_accuracy <- sd(cv_results$train_accuracy)
mean_valid_accuracy <- mean(cv_results$valid_accuracy)
sd_valid_accuracy <- sd(cv_results$valid_accuracy)

cat("\nSummary Statistics:\n")
cat("Mean training accuracy:", mean_train_accuracy, "ś", sd_train_accuracy, "\n")
cat("Mean validation accuracy:", mean_valid_accuracy, "ś", sd_valid_accuracy,
    "\n")

# Select best hyperparameters based on validation performance
best_params <- cv_results[which.max(cv_results$valid_accuracy), ]

# Train final model on full training data
final_svm_model <- svm(
  class ~ .,
  data = train_data_filtered[, !names(train_data_filtered) %in%
    c("class_numeric")],
  kernel = "radial",
  cost = best_params$C,
  gamma = best_params$sigma,
  weights = sapply(train_data_filtered$class_numeric,
                   function(x) class_weights[[as.character(x)]]),
  probability = TRUE,
  scale = FALSE
)

# Evaluate final model on all datasets
evaluate_model <- function(model, data, set_name) {
  predictions <- predict(model, data)
  cm <- confusionMatrix(predictions, data$class)
  cat("\n", set_name, "set evaluation:\n")
  print(cm)
  return(cm)
}

# Final evaluations
train_cm <- evaluate_model(final_svm_model, train_data_filtered, "Training")
validation_cm <- evaluate_model(final_svm_model, validation_data_filtered,
    "Validation")
test_cm <- evaluate_model(final_svm_model, test_data_filtered, "Test")

cat("\nClass-Specific Statistics:\n")
print(test_cm$byClass)

cat("\nConfusion Matrix Mode:\n")
print(test_cm$mode)

cat("\nAdditional Arguments Passed to confusionMatrix():\n")
print(test_cm$dots)

# Print overall accuracy results
accuracy_test <- test_cm$overall['Accuracy']
accuracy_train <- train_cm$overall['Accuracy']
cat("Final Test Accuracy:", accuracy_test, "\n")
cat("Final Train Accuracy:", accuracy_train, "\n")

# Evaluating misclassification cost
# Define the cost matrix (replace with your actual costs)
cost_matrix <- matrix(
```

```
  c(0, 10, 20, 30,   # Benign -> Benign, Early, Pre, Pro
    10, 0, 15, 25,    # Early -> Benign, Early, Pre, Pro
    20, 15, 0, 10,    # Pre -> Benign, Early, Pre, Pro
    30, 25, 10, 0),   # Pro -> Benign, Early, Pre, Pro
  nrow = 4, byrow = TRUE
)
rownames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")
colnames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")

# Extract confusion matrix data
cm_table <- test_cm$table
classes <- rownames(cm_table)

visualize_cost_confusion_matrix <- function(test_cm, cost_matrix) {
  # Extract the confusion matrix from caret's result
  cm_table <- test_cm$table

  # Create cost-weighted matrix
  cost_weighted_cm <- matrix(0,
                             nrow = nrow(cost_matrix),
                             ncol = ncol(cost_matrix))

  for (i in 1:nrow(cm_table)) {
    for (j in 1:ncol(cm_table)) {
      if (i != j) {
        cost_weighted_cm[i, j] <- cm_table[i, j] * cost_matrix[i, j]
      }
    }
  }

  # Set row and column names
  rownames(cost_weighted_cm) <- rownames(cost_matrix)
  colnames(cost_weighted_cm) <- colnames(cost_matrix)

  return(cost_weighted_cm)
}

# Usage
cost_weighted_matrix <- visualize_cost_confusion_matrix(test_cm, cost_matrix)
cat("\nCost-Weighted Misclassification Matrix:\n")
print(cost_weighted_matrix)

# Initialize total misclassification cost
total_cost <- 0

# Calculate total misclassification cost by iterating over confusion matrix
    entries
for (i in 1:nrow(cm_table)) {
  for (j in 1:ncol(cm_table)) {
    if (i != j) {  # Misclassifications (i != j)
      total_cost <- total_cost + cm_table[i, j] * cost_matrix[i, j]
    }
  }
}

# Output the total misclassification cost
cat("Total Misclassification Cost:", total_cost, "\n")

# Calculate average cost per sample
```

```
total_samples <- sum(cm_table)
avg_cost_per_sample <- total_cost / total_samples
cat("Average Cost Per Sample:", avg_cost_per_sample, "\n")

# Calculate cost-sensitive accuracy
# Compute the maximum possible cost (if all instances were misclassified to the most costly class)
max_possible_cost <- sum(cost_matrix) * (total_samples / nrow(cost_matrix))
cost_sensitive_accuracy <- 1 - (total_cost / max_possible_cost)
cat("Cost-Sensitive Accuracy:", cost_sensitive_accuracy, "\n")

# Using cost matrix for cost-weighted F1 calculation
f1_scores <- test_cm$byClass[, "F1"]
weights <- 1 / (diag(cost_matrix) + 1)  # Add 1 to avoid divide-by-zero
weights <- weights / sum(weights)
weighted_f1 <- sum(f1_scores * weights, na.rm = TRUE)
cat("Weighted F1-score", weighted_f1, "\n")

# Save final model and results
save(final_svm_model, cv_results, train_cm, validation_cm, test_cm,
     file = "final_svm_model.RData")
```

9.1.1 XGBoost

```
# Load necessary libraries
library(xgboost)
library(caret)
library(dplyr)
library(imager)
library(EBImage)  # For bwlabel and shape feature extraction
library(ggplot2)
library(e1071)  # For skewness and kurtosis

# Set the path to your dataset
data_dir <- "/kaggle/input/leukemia-images/Original"

# Define image size and parameters
img_height <- 150
img_width <- 150
batch_size <- 32  # Note: batch_size is defined but not used in the code

# Create a list to hold all file paths and corresponding labels
file_paths <- character()  # More explicit initialization
class_labels <- character()

# Loop through each class to gather file paths and labels
for (class in c("Benign", "Early", "Pre", "Pro")) {
  class_path <- file.path(data_dir, class)

  # Add error handling for directory access
  if (!dir.exists(class_path)) {
    stop(paste("Directory not found:", class_path))
  }

  files <- list.files(class_path, full.names = TRUE, pattern =
    "\\.(?i)(jpg|jpeg|png|tiff)$")

  # Ensure files are unique
```

```
  unique_files <- unique(files)

  file_paths <- c(file_paths, unique_files)
  class_labels <- c(class_labels, rep(class, length(unique_files)))
}

# Create a DataFrame with only the required structure
full_df <- data.frame(
  image_filename = file_paths,
  class = factor(class_labels, levels = c("Benign", "Early", "Pre", "Pro")),
  stringsAsFactors = FALSE
)

# Define class-to-index mapping
classes <- c("Benign", "Early", "Pre", "Pro")
class_to_index <- setNames(0:(length(classes) - 1), classes)

# Add numeric class column (0-based indexing)
full_df$class_numeric <- as.numeric(full_df$class) - 1

# Stratified split into training (64%), validation (20%), and test (16%) sets
set.seed(123)
trainIndex <- createDataPartition(full_df$class, p = 0.64, list = FALSE)
train_df <- full_df[trainIndex, ]
remaining_df <- full_df[-trainIndex, ]

# Shuffle only train_df
train_df <- train_df[sample(nrow(train_df)), ]

validationIndex <- createDataPartition(remaining_df$class, p = 20/36, list =
    FALSE)
val_df <- remaining_df[validationIndex, ]
test_df <- remaining_df[-validationIndex, ]

# Calculate class weights using only the training dataset
n_samples <- nrow(train_df)
n_classes <- length(unique(train_df$class))

# Get class frequencies from training dataset
class_counts <- table(train_df$class_numeric)

# Calculate balanced weights
class_weights <- n_samples / (n_classes * class_counts)

# Convert to a named list with numeric indices as names
class_weights <- as.list(class_weights)
names(class_weights) <- as.character(0:(n_classes-1))

# Print diagnostics
cat("\nClass distribution in training dataset:\n")
print(table(train_df$class))

cat("\nFinal Class Weights (based on training dataset):\n")
print(class_weights)

# Validation steps
cat("\nValidation:\n")
cat("Total samples in training dataset:", n_samples, "\n")
cat("Sum of class counts:", sum(class_counts), "\n")
```

```
cat("Number of classes:", n_classes, "\n")
cat("Maximum weight ratio:", max(unlist(class_weights)) /
    min(unlist(class_weights)), "\n\n")

# Additional validation checks
cat("Verification of weight calculation:\n")
for(i in 0:(n_classes-1)) {
  class_name <- classes[i+1]
  weight <- class_weights[[as.character(i)]]
  count <- class_counts[as.character(i)]
  cat(sprintf("%s: count=%d, weight=%.4f, count*weight=%.4f\n",
              class_name, count, weight, count*weight))
}

# Features extraction function
extract_features <- function(image_path, bins = 16, feature_params = NULL) {
  # Load and resize the image using imager
  image <- load.image(image_path)
  image <- resize(image, img_width, img_height)

  # Convert image to array and split into color channels
  red_channel <- as.vector(image[,,1,1])
  green_channel <- as.vector(image[,,1,2])
  blue_channel <- as.vector(image[,,1,3])

  # Color histogram features for each channel
  red_hist <- hist(red_channel, breaks = bins, plot = FALSE)$counts /
    length(red_channel)
  green_hist <- hist(green_channel, breaks = bins, plot = FALSE)$counts /
    length(green_channel)
  blue_hist <- hist(blue_channel, breaks = bins, plot = FALSE)$counts /
    length(blue_channel)

  # Convert to grayscale for texture and edge features
  gray_image <- 0.299 * red_channel + 0.587 * green_channel + 0.114 * blue_channel

  # Texture features: contrast, skewness, and kurtosis
  contrast <- var(gray_image)
  skewness <- e1071::skewness(gray_image)
  kurtosis <- e1071::kurtosis(gray_image)

  # Statistical descriptors
  mean_intensity <- mean(gray_image)
  sd_intensity <- sd(gray_image)
  max_intensity <- max(gray_image)
  min_intensity <- min(gray_image)

  # Edge detection using gradient magnitude approximation (Sobel-like)
  gray_img_cimg <- as.cimg(gray_image, x = img_width, y = img_height)
  grad_x <- imgradient(gray_img_cimg, "x")
  grad_y <- imgradient(gray_img_cimg, "y")
  edge_magnitude <- sqrt(grad_x^2 + grad_y^2)  # Approximate edge strength
  edge_intensity_mean <- mean(edge_magnitude)
  edge_intensity_sd <- sd(edge_magnitude)

  # Shape features using segmentation (using EBImage)
  gray_img_matrix <- matrix(gray_image, nrow = img_height, ncol = img_width)
  binary_image <- gray_img_matrix > 0.5  # Apply threshold to get binary mask
  binary_image <- as.Image(binary_image)  # Convert to EBImage object
```

```
147
148   # Label the binary image
149   labeled_image <- bwlabel(binary_image)
150
151   # Compute shape features
152   shape_features <- computeFeatures.shape(labeled_image)
153
154   # Shape descriptors - Mean values of area, perimeter, and circularity
155   area_mean <- mean(shape_features[,"s.area"], na.rm = TRUE)
156   perimeter_mean <- mean(shape_features[,"s.perimeter"], na.rm = TRUE)
157   circularity_mean <- mean((4 * pi * shape_features[,"s.area"]) /
        (shape_features[,"s.perimeter"]^2), na.rm = TRUE)
158
159   # Combine all features into a single vector
160   feature_vector <- c(red_hist, green_hist, blue_hist, contrast, skewness,
        kurtosis,
161                       mean_intensity, sd_intensity, max_intensity, min_intensity,
162                       edge_intensity_mean, edge_intensity_sd,
163                       area_mean, perimeter_mean, circularity_mean)
164
165   return(feature_vector)
166 }
167
168 # Function to analyze feature lengths across training data
169 get_feature_params <- function(train_image_paths, sample_size = NULL) {
170   # If sample_size is provided, randomly sample images for efficiency
171   if (!is.null(sample_size) && sample_size < length(train_image_paths)) {
172     set.seed(123)  # for reproducibility
173     image_paths <- sample(train_image_paths, sample_size)
174   } else {
175     image_paths <- train_image_paths
176   }
177
178   cat("Analyzing features from", length(image_paths), "training images...\n")
179
180   # Extract features from all training images and analyze
181   feature_lengths <- sapply(image_paths, function(path) {
182     tryCatch({
183       features <- extract_features(path)
184       length(features)
185     }, error = function(e) {
186       warning(sprintf("Error processing image %s: %s", path, e$message))
187       return(NA)
188     })
189   })
190
191   # Analyze feature length distribution
192   length_summary <- summary(feature_lengths)
193   length_table <- table(feature_lengths)
194
195   # Check for inconsistencies
196   if (length(unique(feature_lengths[!is.na(feature_lengths)])) > 1) {
197     warning("Inconsistent feature lengths detected in training data!")
198     print(length_table)
199     print(length_summary)
200   }
201
202   # Get most common feature length (mode)
203   modal_length <- as.numeric(names(length_table)[which.max(length_table)])
```

```
204
205   # Get feature names from a successful extraction
206   sample_features <- NULL
207   for (path in image_paths) {
208     tryCatch({
209       sample_features <- extract_features(path)
210       if (length(sample_features) == modal_length) break
211     }, error = function(e) {
212       next
213     })
214   }
215
216   params <- list(
217     feature_length = modal_length,
218     feature_names = paste0("feature_", seq_len(modal_length)),
219     length_distribution = length_table,
220     length_summary = length_summary,
221     sample_size = length(image_paths),
222     success_rate = sum(!is.na(feature_lengths)) / length(feature_lengths),
223     consistent_length = length(unique(feature_lengths[!is.na(feature_lengths)]))
      == 1
224   )
225
226   # Print summary
227   cat("\nFeature extraction analysis summary:\n")
228   cat("Modal feature length:", params$feature_length, "\n")
229   cat("Success rate:", sprintf("%.2f%%", params$success_rate * 100), "\n")
230   cat("Length consistency:", ifelse(params$consistent_length, "Consistent",
       "Inconsistent"), "\n")
231
232   if (!params$consistent_length) {
233     cat("\nFeature length distribution:\n")
234     print(params$length_distribution)
235   }
236
237   return(params)
238 }
239
240 # Modified feature extraction pipeline
241 set.seed(123)  # for reproducibility
242
243 # Step 1: Get feature parameters from training data
244 # Option to sample if training set is very large
245 sample_size <- if(nrow(train_df) > 1000) 1000 else NULL
246 feature_params <- get_feature_params(train_df$image_filename, sample_size)
247
248 # Enhanced feature extraction function with validation
249 extract_and_validate_features <- function(image_paths, feature_params,
      dataset_name) {
250   features_list <- lapply(seq_along(image_paths), function(i) {
251     path <- image_paths[i]
252
253     if (i %% 100 == 0) {
254       cat(sprintf("Processing %s image %d of %d\n",
255                   dataset_name, i, length(image_paths)))
256     }
257
258     tryCatch({
259       features <- extract_features(path)
```

```
260
261       # Validate feature length
262       if (length(features) != feature_params$feature_length) {
263
264         # Handle inconsistent length
265         if (length(features) < feature_params$feature_length) {
266           # Pad with NA if shorter
267           features <- c(features, rep(NA, feature_params$feature_length -
      length(features)))
268         } else {
269           # Truncate if longer
270           features <- features[1:feature_params$feature_length]
271         }
272       }
273
274       return(features)
275
276     }, error = function(e) {
277       warning(sprintf("Error processing %s image %s: %s",
278                       dataset_name, path, e$message))
279       return(rep(NA, feature_params$feature_length))
280     })
281   })
282
283   # Convert to matrix/data frame
284   features_matrix <- do.call(rbind, features_list)
285   colnames(features_matrix) <- feature_params$feature_names
286
287   # Report statistics about NA values
288   na_counts <- colSums(is.na(features_matrix))
289   if (any(na_counts > 0)) {
290     cat("\nNA counts in", dataset_name, "dataset:\n")
291     print(na_counts[na_counts > 0])
292   }
293
294   return(features_matrix)
295 }
296
297 # Extract features for each dataset
298 cat("\nExtracting training features...\n")
299 train_features <- extract_and_validate_features(
300   train_df$image_filename,
301   feature_params,
302   "training"
303 )
304
305 cat("\nExtracting validation features...\n")
306 val_features <- extract_and_validate_features(
307   val_df$image_filename,
308   feature_params,
309   "validation"
310 )
311
312 cat("\nExtracting test features...\n")
313 test_features <- extract_and_validate_features(
314   test_df$image_filename,
315   feature_params,
316   "test"
317 )
```

```
318
319 # Ensure that class labels are factors with consistent levels
320 train_df$class <- factor(train_df$class, levels = c("Benign", "Early", "Pre",
        "Pro"))
321 test_df$class <- factor(test_df$class, levels = c("Benign", "Early", "Pre",
        "Pro"))
322 val_df$class <- factor(val_df$class, levels = c("Benign", "Early", "Pre", "Pro"))
323
324 train_data <- cbind(as.data.frame(train_features), class_numeric =
        train_df$class_numeric)
325 validation_data <- cbind(as.data.frame(val_features), class_numeric =
        val_df$class_numeric)
326 test_data <- cbind(as.data.frame(test_features), class_numeric =
        test_df$class_numeric)
327
328 train_data[is.na(train_data)] <- 0
329 validation_data[is.na(validation_data)] <- 0
330 test_data[is.na(test_data)] <- 0
331
332 # Convert class_numeric to factor with proper labels for training and testing
333 train_data$class <- factor(train_data$class_numeric,
334                                levels = 0:3,
335                                labels = classes)
336 validation_data$class <- factor(validation_data$class_numeric,
337                                     levels = 0:3,
338                                     labels = classes)
339 test_data$class <- factor(test_data$class_numeric,
340                                levels = 0:3,
341                                labels = classes)
342
343 library(caret)
344 library(e1071)
345
346
347 # Function to identify NZV features from training data
348 identify_nzv_features <- function(train_data) {
349   nzv <- nearZeroVar(train_data, saveMetrics = TRUE)
350   nzv_features <- rownames(nzv)[nzv$zeroVar | nzv$nzv]
351
352   # Log the features being removed and their properties
353   cat("Features removed due to near-zero variance:\n")
354   for(feature in nzv_features) {
355     cat(sprintf("Feature: %s, Variance: %.6f\n",
356                 feature,
357                 var(train_data[[feature]], na.rm = TRUE)))
358   }
359
360   return(nzv_features)
361 }
362
363 # Function to check for differences in feature variance between datasets
364 check_feature_variance <- function(train_data, val_data, test_data, nzv_features)
        {
365   warning_features <- list()
366
367   for(feature in nzv_features) {
368     if(feature %in% names(val_data) && feature %in% names(test_data)) {
369       train_var <- var(train_data[[feature]], na.rm = TRUE)
370       val_var <- var(val_data[[feature]], na.rm = TRUE)
```

```
      test_var <- var(test_data[[feature]], na.rm = TRUE)

      # Check if feature has significant variance in validation or test
      if(val_var > 0.1 || test_var > 0.1) {  # Threshold can be adjusted
        warning_features[[feature]] <- list(
          train_variance = train_var,
          val_variance = val_var,
          test_variance = test_var
        )
      }
    }
  }

  return(warning_features)
}

# Apply the fixed NZV filtering process
# First, identify NZV features from training data only
nzv_features <- identify_nzv_features(train_data[, -ncol(train_data)])

# Check for potential issues in validation and test sets
variance_warnings <- check_feature_variance(
  train_data[, -ncol(train_data)],
  validation_data[, -ncol(validation_data)],
  test_data[, -ncol(test_data)],
  nzv_features
)

# Print warnings if any features show different behavior in val/test
if(length(variance_warnings) > 0) {
  cat("\nWARNING: Some features show different variance patterns in
    validation/test sets:\n")
  print(variance_warnings)
  cat("\nConsider keeping these features in the model.\n")
}

# Remove NZV features and 'class_numeric' from all datasets
train_data_filtered <- train_data[, !names(train_data) %in% c(nzv_features,
    "class_numeric", "class")]
validation_data_filtered <- validation_data[, !names(validation_data) %in%
    c(nzv_features, "class_numeric", "class")]
test_data_filtered <- test_data[, !names(test_data) %in% c(nzv_features,
    "class_numeric", "class")]

# Ensure 'class_numeric' is preserved separately for labels
train_labels <- train_data$class_numeric
validation_labels <- validation_data$class_numeric
test_labels <- test_data$class_numeric

# Ensure the label vector is preserved separately
train_labels <- train_data$class_numeric
validation_labels <- validation_data$class_numeric
test_labels <- test_data$class_numeric

str(train_data_filtered)

head(train_data_filtered)
head(validation_data_filtered)
head(test_data_filtered)
```

```
426
427 class_weights <- as.numeric(class_weights)
428 str(class_weights)
429
430 train_weights <- class_weights[train_df$class_numeric + 1]
431
432 # Check the train_labels values
433 cat("Train Labels:\n")
434 print(head(train_labels))  # Display the first few train labels
435 cat("\n")
436
437 # Check the class_weights values
438 cat("Class Weights:\n")
439 print(class_weights)  # Display the class weights
440 cat("\n")
441
442 # Ensure that the class_weights vector is correctly aligned with the class labels
443 # If we have 4 classes, class_weights should correspond to the following:
444 # 0 -> 1.615 (Benign), 1 -> 0.826 (Early), 2 -> 0.845 (Pre), 3 -> 1.013 (Pro)
445
446 # Assign the weights based on train_labels
447 train_weights <- class_weights[train_labels + 1]  # Adjusting for 1-based
      indexing in R
448
449 # Check the first few train_weights to confirm they match the expected values
450 cat("Assigned Train Weights (first few):\n")
451 print(head(train_weights))  # Display the first few train weights
452 cat("\n")
453
454 # Check for NA values in the train_weights
455 cat("Checking for NA values in train_weights:\n")
456 na_count <- sum(is.na(train_weights))  # Should return 0 if no NA values
457 cat("Number of NA values in train_weights: ", na_count, "\n")
458 cat("\n")
459
460 # If there are NA values, investigate further
461 if (na_count > 0) {
462   cat("Investigate the indices that caused NAs:\n")
463   na_indices <- which(is.na(train_weights))
464   print(train_labels[na_indices])  # Investigate which train_labels caused the NA
      values
465   cat("\n")
466 }
467
468 # Finally, print the first few rows of train_labels and train_weights to confirm
      alignment
469 cat("Train Labels and Corresponding Train Weights:\n")
470 df_check <- data.frame(train_labels = train_labels[1:10], train_weights =
      train_weights[1:10])  # First 10 rows for quick inspection
471 print(df_check)
472 cat("\n")
473
474 # Prepare DMatrix objects
475 train_matrix <- xgb.DMatrix(data = as.matrix(train_data_filtered), label =
      train_labels, weight = train_weights)
476 validation_matrix <- xgb.DMatrix(data = as.matrix(validation_data_filtered),
      label = validation_labels)
477 test_matrix <- xgb.DMatrix(data = as.matrix(test_data_filtered), label =
      test_labels)
```

```
478
479 # Set XGBoost parameters
480 params <- list(
481   objective = "multi:softmax",
482   num_class = 4,
483   eval_metric = c("merror", "mlogloss"),
484   max_depth = 6,
485   eta = 0.1,
486   subsample = 0.8,
487   colsample_bytree = 0.8,
488   min_child_weight = 1,
489   nthread = parallel::detectCores() - 1
490 )
491
492 # Create watchlist for early stopping
493 watchlist <- list(train = train_matrix, val = validation_matrix)
494
495 # Train model with early stopping
496 xgb_model <- xgb.train(
497   params = params,
498   data = train_matrix,
499   nrounds = 1000,
500   watchlist = watchlist,
501   early_stopping_rounds = 20,
502   verbose = 1
503 )
504
505 # Make predictions on test matrix
506 test_predictions <- predict(xgb_model, test_matrix)
507
508 # Make predictions on train matrix
509 train_predictions <- predict(xgb_model, train_matrix)
510
511 # Evaluate results
512 cm_test <- confusionMatrix(
513   factor(test_predictions, levels = 0:3, labels = levels(full_df$class)),
514   factor(test_df$class_numeric, levels = 0:3, labels = levels(full_df$class))
515 )
516
517 # Evaluate results
518 cm_train <- confusionMatrix(
519   factor(train_predictions, levels = 0:3, labels = levels(full_df$class)),
520   factor(train_df$class_numeric, levels = 0:3, labels = levels(full_df$class))
521 )
522
523 # Print results
524 print(cm_test)
525
526 cat("\nClass-Specific Statistics:\n")
527 print(cm_test$byClass)
528
529 cat("\nConfusion Matrix Mode:\n")
530 print(cm_test$mode)
531
532 cat("\nAdditional Arguments Passed to confusionMatrix():\n")
533 print(cm_test$dots)
534
535 # Print overall accuracy results
536 accuracy_test <- cm_test$overall['Accuracy']
```

```r
accuracy_train <- cm_train$overall['Accuracy']
cat("Final Test Accuracy:", accuracy_test, "\n")
cat("Final Train Accuracy:", accuracy_train, "\n")

feature_names <- colnames(train_data_filtered)
xgb_importance <- xgb.importance(feature_names = feature_names, model = xgb_model)

# Select Top 20 Features
top_20_features <- head(xgb_importance, 20)

# Plot the top 20 features
xgb.plot.importance(top_20_features)

# Define the cost matrix (replace with your actual costs)
cost_matrix <- matrix(
  c(0, 10, 20, 30,   # Benign -> Benign, Early, Pre, Pro
    10, 0, 15, 25,    # Early -> Benign, Early, Pre, Pro
    20, 15, 0, 10,    # Pre -> Benign, Early, Pre, Pro
    30, 25, 10, 0),   # Pro -> Benign, Early, Pre, Pro
  nrow = 4, byrow = TRUE
)
rownames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")
colnames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")

# Extract confusion matrix data
cm_table <- cm_test$table
classes <- rownames(cm_table)

visualize_cost_confusion_matrix <- function(cm_test, cost_matrix) {
  # Extract the confusion matrix from caret's result
  cm_table <- cm_test$table

  # Create cost-weighted matrix
  cost_weighted_cm <- matrix(0,
                             nrow = nrow(cost_matrix),
                             ncol = ncol(cost_matrix))

  for (i in 1:nrow(cm_table)) {
    for (j in 1:ncol(cm_table)) {
      if (i != j) {
        cost_weighted_cm[i, j] <- cm_table[i, j] * cost_matrix[i, j]
      }
    }
  }

  # Set row and column names
  rownames(cost_weighted_cm) <- rownames(cost_matrix)
  colnames(cost_weighted_cm) <- colnames(cost_matrix)

  return(cost_weighted_cm)
}

# Usage
cost_weighted_matrix <- visualize_cost_confusion_matrix(cm_test, cost_matrix)
cat("\nCost-Weighted Misclassification Matrix:\n")
print(cost_weighted_matrix)

# Initialize total misclassification cost
total_cost <- 0
```

```

# Calculate total misclassification cost by iterating over confusion matrix entries
for (i in 1:nrow(cm_table)) {
  for (j in 1:ncol(cm_table)) {
    if (i != j) {  # Misclassifications (i != j)
      total_cost <- total_cost + cm_table[i, j] * cost_matrix[i, j]
    }
  }
}

# Output the total misclassification cost
cat("Total Misclassification Cost:", total_cost, "\n")

# Calculate average cost per sample
total_samples <- sum(cm_table)
avg_cost_per_sample <- total_cost / total_samples
cat("Average Cost Per Sample:", avg_cost_per_sample, "\n")

# Calculate cost-sensitive accuracy
# Compute the maximum possible cost (if all instances were misclassified to the most costly class)
max_possible_cost <- sum(cost_matrix) * (total_samples / nrow(cost_matrix))
cost_sensitive_accuracy <- 1 - (total_cost / max_possible_cost)
cat("Cost-Sensitive Accuracy:", cost_sensitive_accuracy, "\n")

# Using cost matrix for cost-weighted F1 calculation
f1_scores <- cm_test$byClass[, "F1"]
weights <- 1 / (diag(cost_matrix) + 1)  # Add 1 to avoid divide-by-zero
weights <- weights / sum(weights)
weighted_f1 <- sum(f1_scores * weights, na.rm = TRUE)
cat("Weighted F1-score", weighted_f1, "\n")

# Save the trained model
xgb.save(xgb_model, "xgb_model.model")
```

9.1.2 VGG16

```

# Load necessary libraries
library(keras)
library(tensorflow)
library(dplyr)  # For data manipulation
library(tidyverse)  # For tidyverse functions
library(caret)  # For confusionMatrix and stratified sampling

# Set the path to your dataset
data_dir <- "/kaggle/input/leukemia-images/Original"

# Define image size and parameters
img_height <- 224
img_width <- 224
batch_size <- 32
epochs <- 220

# Gather file paths and corresponding labels
file_paths <- c()
class_labels <- c()

```

```
for (class in c("Benign", "Early", "Pre", "Pro")) {
  class_path <- file.path(data_dir, class)
  files <- list.files(class_path, full.names = TRUE)
  unique_files <- unique(files)
  file_paths <- c(file_paths, unique_files)
  class_labels <- c(class_labels, rep(class, length(unique_files)))
}

# Create DataFrame with the necessary structure
full_df <- data.frame(
  image_filename = file_paths,
  class = class_labels,
  stringsAsFactors = FALSE
)

# Stratified split into training (64%), validation (20%), and test (16%) sets
set.seed(123)
trainIndex <- createDataPartition(full_df$class, p = 0.64, list = FALSE)
train_df <- full_df[trainIndex, ]
remaining_df <- full_df[-trainIndex, ]

# Shuffle only train_df
train_df <- train_df[sample(nrow(train_df)), ]

# Split the remaining 36% into validation (20%) and test (16%)
validationIndex <- createDataPartition(remaining_df$class, p = 20 / (20 + 16),
    list = FALSE)
val_df <- remaining_df[validationIndex, ]
test_df <- remaining_df[-validationIndex, ]

# Load VGG16 model from Keras
base_model <- application_vgg16(
  weights = 'imagenet',
  include_top = FALSE,
  input_shape = c(img_height, img_width, 3)
)

# Freeze most of the layers of the base model, unfreezing the top few for
    fine-tuning
for (layer in base_model$layers[1:500]) {
  layer$trainable <- FALSE
}
for (layer in base_model$layers[501:length(base_model$layers)]) {
  layer$trainable <- TRUE
}

# Set the color threshold for purple in HSV space
lower_purple <- c(0.7, 0.5, 0.4)  # Adjust as needed
upper_purple <- c(0.8, 1.0, 1.0)  # Adjust as needed

# Define the model input layer
input_image <- layer_input(shape = c(img_height, img_width, 3))

# Convert the input image to HSV
input_image_hsv <- tf$image$rgb_to_hsv(input_image)

# Create a binary mask for purple regions (lymphoblasts) based on the HSV
    threshold
mask_purple <- tf$math$logical_and(
```

```
  input_image_hsv[,,,1] >= lower_purple[1], input_image_hsv[,,,1] <=
    upper_purple[1]
)
mask_purple <- tf$math$logical_and(mask_purple, input_image_hsv[,,,2] >=
    lower_purple[2])
mask_purple <- tf$math$logical_and(mask_purple, input_image_hsv[,,,3] >=
    lower_purple[3])

# Convert logical mask to float and expand dims to match RGB channels
mask_purple <- tf$cast(mask_purple, dtype = tf$float32)  # Ensure the mask is a
    float tensor
mask_purple <- tf$expand_dims(mask_purple, axis = as.integer(-1)) # Add a channel
    dimension
mask_purple <- tf$tile(mask_purple, multiples = as.integer(c(1, 1, 1, 3)))  #
    Repeat for each color channel

# Apply the mask to the original RGB image
segmented_image <- tf$math$multiply(input_image, mask_purple)

# Contracting Path (Encoder) with fewer filters
conv1 <- layer_conv_2d(segmented_image, filters = 32, kernel_size = c(3, 3),
    activation = "relu", padding = "same")
conv1 <- layer_conv_2d(conv1, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool1 <- layer_max_pooling_2d(conv1, pool_size = c(2, 2))

conv2 <- layer_conv_2d(pool1, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv2 <- layer_conv_2d(conv2, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool2 <- layer_max_pooling_2d(conv2, pool_size = c(2, 2))

conv3 <- layer_conv_2d(pool2, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv3 <- layer_conv_2d(conv3, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool3 <- layer_max_pooling_2d(conv3, pool_size = c(2, 2))

# Bottleneck layer with fewer filters
conv4 <- layer_conv_2d(pool3, filters = 256, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv4 <- layer_conv_2d(conv4, filters = 256, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Expansive Path (Decoder) with skip connections and fewer filters

# Upsampling + skip connection from conv3
upconv3 <- layer_conv_2d_transpose(conv4, filters = 128, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat3 <- layer_concatenate(list(upconv3, conv3))
conv5 <- layer_conv_2d(concat3, filters = 128, kernel_size = c(3, 3), activation
    = "relu", padding = "same")
conv5 <- layer_conv_2d(conv5, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Upsampling + skip connection from conv2
upconv2 <- layer_conv_2d_transpose(conv5, filters = 64, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat2 <- layer_concatenate(list(upconv2, conv2))
```

```
conv6 <- layer_conv_2d(concat2, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv6 <- layer_conv_2d(conv6, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Upsampling + skip connection from conv1
upconv1 <- layer_conv_2d_transpose(conv6, filters = 32, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat1 <- layer_concatenate(list(upconv1, conv1))
conv7 <- layer_conv_2d(concat1, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv7 <- layer_conv_2d(conv7, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Feature extraction using VGG16
features <- base_model(input_image)
features <- layer_global_average_pooling_2d(features)

# Flatten and concatenate segmentation output with VGG16 features
segmentation_features <- layer_flatten(conv5)
combined_features <- layer_concatenate(list(features, segmentation_features))

# Dense layers with L2 regularization, batch normalization, and increased dropout
x <- layer_dense(combined_features, units = 512, activation = "relu",
    kernel_regularizer = regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)  # Increased dropout rate to 0.7

x <- layer_dense(x, units = 256, activation = "relu", kernel_regularizer =
    regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)  # Increased dropout rate to 0.7

x <- layer_dense(x, units = 128, activation = "relu", kernel_regularizer =
    regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)  # Increased dropout rate to 0.7

output <- layer_dense(x, units = length(unique(full_df$class)), activation =
    "softmax")

# Create the model
model <- keras_model(inputs = input_image, outputs = output)

# Compile the model with a lower learning rate to prevent overfitting
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(learning_rate = 1e-5),
  metrics = c("accuracy")
)

print(model)

# Define the class-to-index mapping
classes <- c("Benign", "Early", "Pre", "Pro")
class_to_index <- setNames(0:(length(classes) - 1), classes)

# Add numeric columns to both full and training DataFrames
full_df$class_numeric <- as.numeric(factor(full_df$class, levels = classes)) - 1
```

```
train_df$class_numeric <- as.numeric(factor(train_df$class, levels = classes)) - 1

# Verify the class-to-index mapping
cat("Class-to-Index Mapping:\n")
print(class_to_index)

# Create improved mapping displays with counts
cat("\nMappings and counts in full dataset:\n")
unique_mapping_full <- data.frame(
  original_class = classes,
  numeric_class = 0:(length(classes)-1),
  count = as.vector(table(full_df$class))
)
print(unique_mapping_full)

cat("\nMappings and counts in training dataset:\n")
unique_mapping_train <- data.frame(
  original_class = classes,
  numeric_class = 0:(length(classes)-1),
  count = as.vector(table(train_df$class))
)
print(unique_mapping_train)

# Calculate class weights using the train dataset
n_samples <- nrow(train_df)
n_classes <- length(classes)

# Get class frequencies from train dataset
class_counts <- table(train_df$class_numeric)

# Calculate balanced weights
class_weights <- n_samples / (n_classes * class_counts)

# Convert to a named list with numeric indices as names
class_weights <- as.list(class_weights)
names(class_weights) <- as.character(0:(n_classes-1))

# Print diagnostics
cat("\nClass distribution in full dataset:\n")
print(table(full_df$class))
cat("\nClass distribution in training dataset:\n")
print(table(train_df$class))

cat("\nFinal Class Weights (based on train dataset):\n")
print(class_weights)

# Validation steps
cat("\nValidation:\n")
cat("Total samples in full dataset:", n_samples, "\n")
cat("Sum of class counts:", sum(class_counts), "\n")
cat("Number of classes:", n_classes, "\n")
cat("Maximum weight ratio:", max(unlist(class_weights)) /
    min(unlist(class_weights)), "\n\n")

# Additional validation checks
cat("Verification of weight calculation:\n")
for(i in 0:(n_classes-1)) {
  class_name <- classes[i+1]
  weight <- class_weights[[as.character(i)]]
```

```
  count <- class_counts[as.character(i)]
  cat(sprintf("%s: count=%d, weight=%.4f, count*weight=%.4f\n",
              class_name, count, weight, count*weight))
}

# Create a data generator with augmentation (no augmentation on validation set)
datagen_train <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,  # Increased rotation range for more variability
  width_shift_range = 0.4,  # Increased shift range for more variability
  height_shift_range = 0.4,  # Increased shift range for more variability
  shear_range = 0.4,  # Increased shear range for more variability
  zoom_range = 0.4,  # Increased zoom range for more variability
  horizontal_flip = TRUE
)

datagen_val <- image_data_generator(
  rescale = 1/255
)

datagen_test <- image_data_generator(
  rescale = 1/255
)

# Custom generator for training and validation (no augmentation on validation set)
train_generator <- flow_images_from_dataframe(
  train_df,
  directory = data_dir,
  x_col = "image_filename",
  y_col = "class",
  target_size = c(img_height, img_width),
  batch_size = batch_size,
  class_mode = "categorical",
  generator = datagen_train
)

val_generator <- flow_images_from_dataframe(
  val_df,
  directory = data_dir,
  x_col = "image_filename",
  y_col = "class",
  target_size = c(img_height, img_width),
  batch_size = batch_size,
  class_mode = "categorical",
  generator = datagen_val,
  shuffle = FALSE,
)

test_generator <- flow_images_from_dataframe(
  test_df,
  directory = data_dir,
  x_col = "image_filename",
  y_col = "class",
  target_size = c(img_height, img_width),
  batch_size = batch_size,
  class_mode = "categorical",
  generator = datagen_test,
  shuffle = FALSE,
)
```

```
285
286 # Initialize an empty vector to store the learning rate values
287 lr_values <- numeric(0)
288
289 # Custom callback to track the learning rate during training
290 callback_lr_tracker <- callback_lambda(
291   on_epoch_begin = function(epoch, logs) {
292     lr_values <<- c(lr_values, as.numeric(model$optimizer$lr))  # Store learning
      rate at the start of each epoch
293   }
294 )
295
296
297 # Train the model with class weights and validation
298 history <- model %>% fit(
299   train_generator,
300   epochs = epochs,
301   steps_per_epoch = nrow(train_df) / batch_size,
302   validation_data = val_generator,
303   validation_steps = nrow(val_df) / batch_size,
304   class_weight = class_weights,
305   callbacks = list(
306     callback_early_stopping(monitor = "val_loss", patience = 8,
      restore_best_weights = TRUE),  # Increased patience
307     callback_reduce_lr_on_plateau(monitor = "val_loss", factor = 0.1, patience =
      6),
308     callback_model_checkpoint("best_model.h5", monitor = "val_loss",
      save_best_only = TRUE),  # Save the best model
309     callback_lr_tracker
310   )
311 )
312
313 # Retrieve the class labels as defined in the test generator
314 class_indices <- test_generator$class_indices  # This is a named list
315 # Reverse the class_indices to map indices back to labels
316 index_to_label <- names(class_indices)[order(unlist(class_indices))]
317
318 # Evaluate the model on the test data
319 evaluation_metrics <- model %>% evaluate(
320   test_generator,
321   steps = nrow(test_df) / batch_size
322 )
323
324 # Extract the evaluation metrics (test loss and accuracy)
325 test_loss <- evaluation_metrics[1]
326 test_accuracy <- evaluation_metrics[2]
327
328 cat("Test Loss:", test_loss, "\n")
329 cat("Test Accuracy:", test_accuracy, "\n")
330
331 # Calculate the number of steps for test predictions
332 steps <- ceiling(nrow(test_df) / batch_size)
333
334 # Make predictions on the test set
335 predictions <- model %>% predict(test_generator, steps = steps)
336
337 # Map the predicted indices to class labels using index_to_label
338 predicted_classes <- factor(apply(predictions, 1, function(x)
    index_to_label[which.max(x)]), levels = index_to_label)
```

```
339
340 # Get true labels from the test data frame (already in factor form)
341 true_classes <- factor(test_df$class, levels = index_to_label)
342
343 # Calculate the confusion matrix
344 conf_matrix <- confusionMatrix(predicted_classes, true_classes)
345
346 # Print confusion matrix details
347 print(conf_matrix)
348
349 cat("\nClass-Specific Statistics:\n")
350 print(conf_matrix$byClass)
351
352 cat("\nConfusion Matrix Mode:\n")
353 print(conf_matrix$mode)
354
355 cat("\nAdditional Arguments Passed to confusionMatrix():\n")
356 print(conf_matrix$dots)
357
358 # Extract the confusion matrix table as a data frame
359 cm_table <- as.data.frame(as.table(conf_matrix))
360
361 # Rename the columns for easier reference
362 colnames(cm_table) <- c("Predicted", "Actual", "Count")
363
364 # Plot the confusion matrix as a heatmap using ggplot2
365 ggplot(cm_table, aes(x = Predicted, y = Actual, fill = Count)) +
366   geom_tile() +
367   scale_fill_gradient(low = "white", high = "blue") +
368   theme_minimal() +
369   labs(title = "Confusion Matrix Heatmap",
370        x = "Predicted Class",
371        y = "Actual Class") +
372   theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
373   geom_text(aes(label = Count), color = "black", size = 5)  # Add text labels
      with the count in each tile
374
375 plot(history)
376
377 library(ggplot2)
378
379 # Create a data frame for plotting
380 history_df <- data.frame(
381   epoch = 1:length(history$metrics$loss),
382   loss = history$metrics$loss,
383   val_loss = history$metrics$val_loss,
384   accuracy = history$metrics$accuracy,
385   val_accuracy = history$metrics$val_accuracy,
386   lr = lr_values   # Add the learning rate history
387 )
388
389 # Create history_df to include loss, accuracy, and learning rate values
390 history_df <- data.frame(
391   epoch = 1:length(history$metrics$loss),
392   loss = history$metrics$loss,
393   val_loss = history$metrics$val_loss,
394   accuracy = history$metrics$accuracy,
395   val_accuracy = history$metrics$val_accuracy,
396   lr = lr_values   # Add the learning rate history
```

```
)

library(ggplot2)
library(patchwork)
library(scales)  # for trans_format and log_breaks

# Linear scale plots
plot1 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  geom_line(aes(y = accuracy * 100, color = 'Accuracy (%)')) +
  scale_y_continuous(sec.axis = sec_axis(~./100, name = "Accuracy (%)")) +
  labs(title = "All Metrics (Linear Scale)", x = "Epoch", y = "Value") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red", "green"))

# Logarithmic scale plots with custom log base labels
plot2 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  geom_line(aes(y = accuracy * 100, color = 'Accuracy (%)')) +
  scale_y_log10(labels = trans_format("log10", math_format(10^.x))) +
  labs(title = "All Metrics (Logarithmic Scale)", x = "Epoch", y = "Value (log
    scale)") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red", "green"))

# Loss linear scale
plot3 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  labs(title = "Losses (Linear Scale)", x = "Epoch", y = "Loss") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red"))

# Loss logarithmic scale with custom log base labels
plot4 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  scale_y_log10(labels = trans_format("log10", math_format(10^.x))) +
  labs(title = "Losses (Logarithmic Scale)", x = "Epoch", y = "Loss (log scale)")
    +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red"))

# Combine all plots into a single grid
combined_plot <- (plot1 | plot2) / (plot3 | plot4) +
  plot_annotation(title = "Training Progress")

# Print combined plot
print(combined_plot)

# Get final training and validation accuracy from the history object
final_train_accuracy <- history$metrics$accuracy[length(history$metrics$accuracy)]
final_val_accuracy <-
    history$metrics$val_accuracy[length(history$metrics$val_accuracy)]

# Extract the final training loss from the history object
final_train_loss <- history$metrics$loss[length(history$metrics$loss)]
```

```
final_val_loss <- history$metrics$loss[length(history$metrics$val_loss)]

# Print the final training loss
cat("Final Training Loss:", final_train_loss, "\n")
cat("Final Validation Loss:", final_val_loss, "\n")
cat("Final Train Accuracy:", final_train_accuracy, "\n")
cat("Final Validation Accuracy:", final_val_accuracy, "\n")

# Evaluating classification cost
# Define the cost matrix (replace with your actual costs)
cost_matrix <- matrix(
  c(0, 10, 20, 30,   # Benign -> Benign, Early, Pre, Pro
    10, 0, 15, 25,    # Early -> Benign, Early, Pre, Pro
    20, 15, 0, 10,    # Pre -> Benign, Early, Pre, Pro
    30, 25, 10, 0),   # Pro -> Benign, Early, Pre, Pro
  nrow = 4, byrow = TRUE
)
rownames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")
colnames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")

# Extract confusion matrix data
cm_table <- conf_matrix$table
classes <- rownames(cm_table)

visualize_cost_confusion_matrix <- function(conf_matrix, cost_matrix) {
  # Extract the confusion matrix from caret's result
  cm_table <- conf_matrix$table

  # Create cost-weighted matrix
  cost_weighted_cm <- matrix(0,
                             nrow = nrow(cost_matrix),
                             ncol = ncol(cost_matrix))

  for (i in 1:nrow(cm_table)) {
    for (j in 1:ncol(cm_table)) {
      if (i != j) {
        cost_weighted_cm[i, j] <- cm_table[i, j] * cost_matrix[i, j]
      }
    }
  }

  # Set row and column names
  rownames(cost_weighted_cm) <- rownames(cost_matrix)
  colnames(cost_weighted_cm) <- colnames(cost_matrix)

  return(cost_weighted_cm)
}

# Usage
cost_weighted_matrix <- visualize_cost_confusion_matrix(conf_matrix, cost_matrix)
cat("\nCost-Weighted Misclassification Matrix:\n")
print(cost_weighted_matrix)

# Initialize total misclassification cost
total_cost <- 0

# Calculate total misclassification cost by iterating over confusion matrix
    entries
for (i in 1:nrow(cm_table)) {
```

```
  for (j in 1:ncol(cm_table)) {
    if (i != j) {  # Misclassifications (i != j)
      total_cost <- total_cost + cm_table[i, j] * cost_matrix[i, j]
    }
  }
}

# Output the total misclassification cost
cat("Total Misclassification Cost:", total_cost, "\n")

# Calculate average cost per sample
total_samples <- sum(cm_table)
avg_cost_per_sample <- total_cost / total_samples
cat("Average Cost Per Sample:", avg_cost_per_sample, "\n")

# Calculate cost-sensitive accuracy
# Compute the maximum possible cost (if all instances were misclassified to the most costly class)
max_possible_cost <- sum(cost_matrix) * (total_samples / nrow(cost_matrix))
cost_sensitive_accuracy <- 1 - (total_cost / max_possible_cost)
cat("Cost-Sensitive Accuracy:", cost_sensitive_accuracy, "\n")

# Using cost matrix for cost-weighted F1 calculation
f1_scores <- conf_matrix$byClass[, "F1"]
weights <- 1 / (diag(cost_matrix) + 1)  # Add 1 to avoid divide-by-zero
weights <- weights / sum(weights)
weighted_f1 <- sum(f1_scores * weights, na.rm = TRUE)
cat("Weighted F1-score", weighted_f1, "\n")

# Save the trained model as an HDF5 file
save_model_hdf5(model, "vgg16_model.h5")
```

9.1.3 DenseNet-201

```
# Load necessary libraries
library(keras)
library(tensorflow)
library(dplyr)
library(tidyverse)
library(caret)

# Set the path to your dataset
data_dir <- "/kaggle/input/leukemia-images/Original"

# Define image size and parameters for DenseNet-201
img_height <- 224
img_width <- 224
batch_size <- 32
epochs <- 220

# Gather file paths and corresponding labels
file_paths <- c()
class_labels <- c()

for (class in c("Benign", "Early", "Pre", "Pro")) {
  class_path <- file.path(data_dir, class)
  files <- list.files(class_path, full.names = TRUE)
  unique_files <- unique(files)
```

```
  file_paths <- c(file_paths, unique_files)
  class_labels <- c(class_labels, rep(class, length(unique_files)))
}

# Create DataFrame with the necessary structure
full_df <- data.frame(
  image_filename = file_paths,
  class = class_labels,
  stringsAsFactors = FALSE
)

# Stratified split into training (64%), validation (20%), and test (16%) sets
set.seed(123)
trainIndex <- createDataPartition(full_df$class, p = 0.64, list = FALSE)
train_df <- full_df[trainIndex, ]
remaining_df <- full_df[-trainIndex, ]

# Shuffle only train_df
train_df <- train_df[sample(nrow(train_df)), ]

# Split the remaining 36% into validation (20%) and test (16%)
validationIndex <- createDataPartition(remaining_df$class, p = 20 / (20 + 16),
    list = FALSE)
val_df <- remaining_df[validationIndex, ]
test_df <- remaining_df[-validationIndex, ]

# Load DenseNet-201 model from TensorFlow
base_model <- tf$keras$applications$DenseNet201(
  weights = 'imagenet',
  include_top = FALSE,
  input_shape = as.integer(c(img_height, img_width, 3))
)

# Freeze most of the layers of the base model, unfreezing the top few for
    fine-tuning
for (layer in base_model$layers[1:500]) {
  layer$trainable <- FALSE
}
for (layer in base_model$layers[501:length(base_model$layers)]) {
  layer$trainable <- TRUE
}

# Define the input and segmentation layer, as before
input_image <- layer_input(shape = c(img_height, img_width, 3))

# Define the color threshold for purple in HSV space
lower_purple <- c(0.7, 0.5, 0.4)
upper_purple <- c(0.8, 1.0, 1.0)

# Convert the input image to HSV and create a binary mask for purple regions
input_image_hsv <- tf$image$rgb_to_hsv(input_image)
mask_purple <- tf$math$logical_and(
  input_image_hsv[,,,1] >= lower_purple[1], input_image_hsv[,,,1] <=
    upper_purple[1]
)
mask_purple <- tf$math$logical_and(mask_purple, input_image_hsv[,,,2] >=
    lower_purple[2])
mask_purple <- tf$math$logical_and(mask_purple, input_image_hsv[,,,3] >=
    lower_purple[3])
```

```
mask_purple <- tf$cast(mask_purple, dtype = tf$float32)
mask_purple <- tf$expand_dims(mask_purple, axis = as.integer(-1))
mask_purple <- tf$tile(mask_purple, multiples = as.integer(c(1, 1, 1, 3)))
segmented_image <- tf$math$multiply(input_image, mask_purple)

# Contracting Path (Encoder) and Expansive Path (Decoder) as before
conv1 <- layer_conv_2d(segmented_image, filters = 32, kernel_size = c(3, 3),
    activation = "relu", padding = "same")
conv1 <- layer_conv_2d(conv1, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool1 <- layer_max_pooling_2d(conv1, pool_size = c(2, 2))

conv2 <- layer_conv_2d(pool1, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv2 <- layer_conv_2d(conv2, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool2 <- layer_max_pooling_2d(conv2, pool_size = c(2, 2))

conv3 <- layer_conv_2d(pool2, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv3 <- layer_conv_2d(conv3, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool3 <- layer_max_pooling_2d(conv3, pool_size = c(2, 2))

conv4 <- layer_conv_2d(pool3, filters = 256, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv4 <- layer_conv_2d(conv4, filters = 256, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Decoder with skip connections
upconv3 <- layer_conv_2d_transpose(conv4, filters = 128, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat3 <- layer_concatenate(list(upconv3, conv3))
conv5 <- layer_conv_2d(concat3, filters = 128, kernel_size = c(3, 3), activation
    = "relu", padding = "same")
conv5 <- layer_conv_2d(conv5, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

upconv2 <- layer_conv_2d_transpose(conv5, filters = 64, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat2 <- layer_concatenate(list(upconv2, conv2))
conv6 <- layer_conv_2d(concat2, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv6 <- layer_conv_2d(conv6, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

upconv1 <- layer_conv_2d_transpose(conv6, filters = 32, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat1 <- layer_concatenate(list(upconv1, conv1))
conv7 <- layer_conv_2d(concat1, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv7 <- layer_conv_2d(conv7, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Feature extraction using DenseNet-201
features <- base_model(input_image)
features <- layer_global_average_pooling_2d(features)

# Combine segmentation and DenseNet-201 features
```

```
segmentation_features <- layer_flatten(conv5)
combined_features <- layer_concatenate(list(features, segmentation_features))

# Dense layers with regularization, batch normalization, and dropout
x <- layer_dense(combined_features, units = 512, activation = "relu",
    kernel_regularizer = regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)

x <- layer_dense(x, units = 256, activation = "relu", kernel_regularizer =
    regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)

x <- layer_dense(x, units = 128, activation = "relu", kernel_regularizer =
    regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)

output <- layer_dense(x, units = length(unique(full_df$class)), activation =
    "softmax")

# Create the final model
model <- keras_model(inputs = input_image, outputs = output)

# Compile the model with adjusted learning rate
model %>% compile(
  loss = loss_categorical_crossentropy(),
  optimizer = optimizer_adam(learning_rate = 1e-4),
  metrics = c("accuracy")
)

print(model)

# Define the class-to-index mapping
classes <- c("Benign", "Early", "Pre", "Pro")
class_to_index <- setNames(0:(length(classes) - 1), classes)

# Add numeric columns to both full and training DataFrames
full_df$class_numeric <- as.numeric(factor(full_df$class, levels = classes)) - 1
train_df$class_numeric <- as.numeric(factor(train_df$class, levels = classes)) - 1

# Verify the class-to-index mapping
cat("Class-to-Index Mapping:\n")
print(class_to_index)

# Create improved mapping displays with counts
cat("\nMappings and counts in full dataset:\n")
unique_mapping_full <- data.frame(
  original_class = classes,
  numeric_class = 0:(length(classes)-1),
  count = as.vector(table(full_df$class))
)
print(unique_mapping_full)

cat("\nMappings and counts in training dataset:\n")
unique_mapping_train <- data.frame(
  original_class = classes,
  numeric_class = 0:(length(classes)-1),
```

```
  count = as.vector(table(train_df$class))
)
print(unique_mapping_train)

# Calculate class weights using the train dataset
n_samples <- nrow(train_df)
n_classes <- length(classes)

# Get class frequencies from train dataset
class_counts <- table(train_df$class_numeric)

# Calculate balanced weights
class_weights <- n_samples / (n_classes * class_counts)

# Convert to a named list with numeric indices as names
class_weights <- as.list(class_weights)
names(class_weights) <- as.character(0:(n_classes-1))

# Print diagnostics
cat("\nClass distribution in full dataset:\n")
print(table(full_df$class))
cat("\nClass distribution in training dataset:\n")
print(table(train_df$class))

cat("\nFinal Class Weights (based on train dataset):\n")
print(class_weights)

# Validation steps
cat("\nValidation:\n")
cat("Total samples in full dataset:", n_samples, "\n")
cat("Sum of class counts:", sum(class_counts), "\n")
cat("Number of classes:", n_classes, "\n")
cat("Maximum weight ratio:", max(unlist(class_weights)) /
    min(unlist(class_weights)), "\n\n")

# Additional validation checks
cat("Verification of weight calculation:\n")
for(i in 0:(n_classes-1)) {
  class_name <- classes[i+1]
  weight <- class_weights[[as.character(i)]]
  count <- class_counts[as.character(i)]
  cat(sprintf("%s: count=%d, weight=%.4f, count*weight=%.4f\n",
              class_name, count, weight, count*weight))
}

# Create a data generator with augmentation (no augmentation on validation set)
datagen_train <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,  # Increased rotation range for more variability
  width_shift_range = 0.4,  # Increased shift range for more variability
  height_shift_range = 0.4,  # Increased shift range for more variability
  shear_range = 0.4,  # Increased shear range for more variability
  zoom_range = 0.4,  # Increased zoom range for more variability
  horizontal_flip = TRUE
)

datagen_val <- image_data_generator(
  rescale = 1/255
)
```

```
234
235 datagen_test <- image_data_generator(
236   rescale = 1/255
237 )
238
239 # Custom generator for training and validation (no augmentation on validation set)
240 train_generator <- flow_images_from_dataframe(
241   train_df,
242   directory = data_dir,
243   x_col = "image_filename",
244   y_col = "class",
245   target_size = c(img_height, img_width),
246   batch_size = batch_size,
247   class_mode = "categorical",
248   generator = datagen_train
249 )
250
251 val_generator <- flow_images_from_dataframe(
252   val_df,
253   directory = data_dir,
254   x_col = "image_filename",
255   y_col = "class",
256   target_size = c(img_height, img_width),
257   batch_size = batch_size,
258   class_mode = "categorical",
259   generator = datagen_val,
260   shuffle = FALSE,
261 )
262
263 test_generator <- flow_images_from_dataframe(
264   test_df,
265   directory = data_dir,
266   x_col = "image_filename",
267   y_col = "class",
268   target_size = c(img_height, img_width),
269   batch_size = batch_size,
270   class_mode = "categorical",
271   generator = datagen_test,
272   shuffle = FALSE,
273 )
274
275 # Initialize an empty vector to store the learning rate values
276 lr_values <- numeric(0)
277
278 # Custom callback to track the learning rate during training
279 callback_lr_tracker <- callback_lambda(
280   on_epoch_begin = function(epoch, logs) {
281     lr_values <<- c(lr_values, as.numeric(model$optimizer$lr))  # Store learning
      rate at the start of each epoch
282   }
283 )
284
285
286 # Train the model with class weights and validation
287 history <- model %>% fit(
288   train_generator,
289   epochs = epochs,
290   steps_per_epoch = nrow(train_df) / batch_size,
291   validation_data = val_generator,
```

```
  validation_steps = nrow(val_df) / batch_size,
  class_weight = class_weights,
  callbacks = list(
    callback_early_stopping(monitor = "val_loss", patience = 5,
    restore_best_weights = TRUE),  # Increased patience
    callback_reduce_lr_on_plateau(monitor = "val_loss", factor = 0.1, patience =
    3),
    callback_model_checkpoint("best_model.h5", monitor = "val_loss",
    save_best_only = TRUE),  # Save the best model
    callback_lr_tracker
  )
)



# Retrieve the class labels as defined in the test generator
class_indices <- test_generator$class_indices  # This is a named list
# Reverse the class_indices to map indices back to labels
index_to_label <- names(class_indices)[order(unlist(class_indices))]

# Evaluate the model on the test data
evaluation_metrics <- model %>% evaluate(
  test_generator,
  steps = nrow(test_df) / batch_size
)

# Extract the evaluation metrics (test loss and accuracy)
test_loss <- evaluation_metrics[1]
test_accuracy <- evaluation_metrics[2]

cat("Test Loss:", test_loss, "\n")
cat("Test Accuracy:", test_accuracy, "\n")

# Calculate the number of steps for test predictions
steps <- ceiling(nrow(test_df) / batch_size)

# Make predictions on the test set
predictions <- model %>% predict(test_generator, steps = steps)

# Map the predicted indices to class labels using index_to_label
predicted_classes <- factor(apply(predictions, 1, function(x)
    index_to_label[which.max(x)]), levels = index_to_label)

# Get true labels from the test data frame (already in factor form)
true_classes <- factor(test_df$class, levels = index_to_label)

# Calculate the confusion matrix
conf_matrix <- confusionMatrix(predicted_classes, true_classes)

# Print confusion matrix details
print(conf_matrix)

cat("\nClass-Specific Statistics:\n")
print(conf_matrix$byClass)

cat("\nConfusion Matrix Mode:\n")
print(conf_matrix$mode)

cat("\nAdditional Arguments Passed to confusionMatrix():\n")
```

```
print(conf_matrix$dots)

# Extract the confusion matrix table as a data frame
cm_table <- as.data.frame(as.table(conf_matrix))

# Rename the columns for easier reference
colnames(cm_table) <- c("Predicted", "Actual", "Count")

# Plot the confusion matrix as a heatmap using ggplot2
ggplot(cm_table, aes(x = Predicted, y = Actual, fill = Count)) +
  geom_tile() +
  scale_fill_gradient(low = "white", high = "blue") +
  theme_minimal() +
  labs(title = "Confusion Matrix Heatmap",
       x = "Predicted Class",
       y = "Actual Class") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  geom_text(aes(label = Count), color = "black", size = 5)  # Add text labels
    with the count in each tile

plot(history)

library(ggplot2)

# Create a data frame for plotting
history_df <- data.frame(
  epoch = 1:length(history$metrics$loss),
  loss = history$metrics$loss,
  val_loss = history$metrics$val_loss,
  accuracy = history$metrics$accuracy,
  val_accuracy = history$metrics$val_accuracy,
  lr = lr_values   # Add the learning rate history
)

# Create history_df to include loss, accuracy, and learning rate values
history_df <- data.frame(
  epoch = 1:length(history$metrics$loss),
  loss = history$metrics$loss,
  val_loss = history$metrics$val_loss,
  accuracy = history$metrics$accuracy,
  val_accuracy = history$metrics$val_accuracy,
  lr = lr_values   # Add the learning rate history
)

library(ggplot2)
library(patchwork)
library(scales)  # for trans_format and log_breaks

# Linear scale plots
plot1 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  geom_line(aes(y = accuracy * 100, color = 'Accuracy (%)')) +
  scale_y_continuous(sec.axis = sec_axis(~./100, name = "Accuracy (%)")) +
  labs(title = "All Metrics (Linear Scale)", x = "Epoch", y = "Value") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red", "green"))

# Logarithmic scale plots with custom log base labels
```

```
plot2 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  geom_line(aes(y = accuracy * 100, color = 'Accuracy (%)')) +
  scale_y_log10(labels = trans_format("log10", math_format(10^.x))) +
  labs(title = "All Metrics (Logarithmic Scale)", x = "Epoch", y = "Value (log
    scale)") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red", "green"))

# Loss linear scale
plot3 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  labs(title = "Losses (Linear Scale)", x = "Epoch", y = "Loss") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red"))

# Loss logarithmic scale with custom log base labels
plot4 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  scale_y_log10(labels = trans_format("log10", math_format(10^.x))) +
  labs(title = "Losses (Logarithmic Scale)", x = "Epoch", y = "Loss (log scale)")
    +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red"))

# Combine all plots into a single grid
combined_plot <- (plot1 | plot2) / (plot3 | plot4) +
  plot_annotation(title = "Training Progress")

# Print combined plot
print(combined_plot)

# Get final training and validation accuracy from the history object
final_train_accuracy <- history$metrics$accuracy[length(history$metrics$accuracy)]
final_val_accuracy <-
    history$metrics$val_accuracy[length(history$metrics$val_accuracy)]

# Extract the final training loss from the history object
final_train_loss <- history$metrics$loss[length(history$metrics$loss)]
final_val_loss <- history$metrics$loss[length(history$metrics$val_loss)]

# Print the final training loss
cat("Final Training Loss:", final_train_loss, "\n")
cat("Final Validation Loss:", final_val_loss, "\n")
cat("Final Train Accuracy:", final_train_accuracy, "\n")
cat("Final Validation Accuracy:", final_val_accuracy, "\n")

# Evaluating classification cost
# Define the cost matrix (replace with your actual costs)
cost_matrix <- matrix(
  c(0, 10, 20, 30,   # Benign -> Benign, Early, Pre, Pro
    10, 0, 15, 25,    # Early -> Benign, Early, Pre, Pro
    20, 15, 0, 10,    # Pre -> Benign, Early, Pre, Pro
    30, 25, 10, 0),   # Pro -> Benign, Early, Pre, Pro
  nrow = 4, byrow = TRUE
)
```

```r
rownames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")
colnames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")

# Extract confusion matrix data
cm_table <- conf_matrix$table
classes <- rownames(cm_table)

visualize_cost_confusion_matrix <- function(conf_matrix, cost_matrix) {
  # Extract the confusion matrix from caret's result
  cm_table <- conf_matrix$table

  # Create cost-weighted matrix
  cost_weighted_cm <- matrix(0,
                             nrow = nrow(cost_matrix),
                             ncol = ncol(cost_matrix))

  for (i in 1:nrow(cm_table)) {
    for (j in 1:ncol(cm_table)) {
      if (i != j) {
        cost_weighted_cm[i, j] <- cm_table[i, j] * cost_matrix[i, j]
      }
    }
  }

  # Set row and column names
  rownames(cost_weighted_cm) <- rownames(cost_matrix)
  colnames(cost_weighted_cm) <- colnames(cost_matrix)

  return(cost_weighted_cm)
}

# Usage
cost_weighted_matrix <- visualize_cost_confusion_matrix(conf_matrix, cost_matrix)
cat("\nCost-Weighted Misclassification Matrix:\n")
print(cost_weighted_matrix)

# Initialize total misclassification cost
total_cost <- 0

# Calculate total misclassification cost by iterating over confusion matrix
    entries
for (i in 1:nrow(cm_table)) {
  for (j in 1:ncol(cm_table)) {
    if (i != j) {  # Misclassifications (i != j)
      total_cost <- total_cost + cm_table[i, j] * cost_matrix[i, j]
    }
  }
}

# Output the total misclassification cost
cat("Total Misclassification Cost:", total_cost, "\n")

# Calculate average cost per sample
total_samples <- sum(cm_table)
avg_cost_per_sample <- total_cost / total_samples
cat("Average Cost Per Sample:", avg_cost_per_sample, "\n")

# Calculate cost-sensitive accuracy
```

```r
# Compute the maximum possible cost (if all instances were misclassified to the most costly class)
max_possible_cost <- sum(cost_matrix) * (total_samples / nrow(cost_matrix))
cost_sensitive_accuracy <- 1 - (total_cost / max_possible_cost)
cat("Cost-Sensitive Accuracy:", cost_sensitive_accuracy, "\n")

# Using cost matrix for cost-weighted F1 calculation
f1_scores <- conf_matrix$byClass[, "F1"]
weights <- 1 / (diag(cost_matrix) + 1)  # Add 1 to avoid divide-by-zero
weights <- weights / sum(weights)
weighted_f1 <- sum(f1_scores * weights, na.rm = TRUE)
cat("Weighted F1-score", weighted_f1, "\n")

# Save the trained model as an HDF5 file
save_model_hdf5(model, "densenet201_model.h5")
```

9.1.4 ConvNeXt

```r
# Load necessary libraries
library(keras)
library(tensorflow)
library(dplyr)  # For data manipulation
library(tidyverse)  # For tidyverse functions
library(caret)  # For confusionMatrix and stratified sampling

# Set the path to your dataset
data_dir <- "/kaggle/input/leukemia-images/Original"

# Define image size and parameters for ConvNeXT
img_height <- 224
img_width <- 224
batch_size <- 32
epochs <- 220

# Gather file paths and corresponding labels
file_paths <- c()
class_labels <- c()

for (class in c("Benign", "Early", "Pre", "Pro")) {
  class_path <- file.path(data_dir, class)
  files <- list.files(class_path, full.names = TRUE)
  unique_files <- unique(files)
  file_paths <- c(file_paths, unique_files)
  class_labels <- c(class_labels, rep(class, length(unique_files)))
}

# Create DataFrame with the necessary structure
full_df <- data.frame(
  image_filename = file_paths,
  class = class_labels,
  stringsAsFactors = FALSE
)

# Stratified split into training (64%), validation (20%), and test (16%) sets
set.seed(123)
trainIndex <- createDataPartition(full_df$class, p = 0.64, list = FALSE)
train_df <- full_df[trainIndex, ]
remaining_df <- full_df[-trainIndex, ]
```

```
41
42 # Shuffle only train_df
43 train_df <- train_df[sample(nrow(train_df)), ]
44
45 # Split the remaining 36% into validation (20%) and test (16%)
46 validationIndex <- createDataPartition(remaining_df$class, p = 20 / (20 + 16),
       list = FALSE)
47 val_df <- remaining_df[validationIndex, ]
48 test_df <- remaining_df[-validationIndex, ]
49
50 # Load ConvNeXT model from TensorFlow using tf$keras$applications$ConvNeXtBase
51 base_model <- tf$keras$applications$ConvNeXtBase(
52   weights = 'imagenet',
53   include_top = FALSE,
54   input_shape = as.integer(c(img_height, img_width, 3))
55 )
56
57 # Freeze most of the layers of the base model, but unfreeze the top few layers
       for fine-tuning
58 for (layer in base_model$layers[1:100]) {
59   layer$trainable <- FALSE
60 }
61 for (layer in base_model$layers[101:length(base_model$layers)]) {
62   layer$trainable <- TRUE
63 }
64
65 # Set the color threshold for purple in HSV space
66 lower_purple <- c(0.7, 0.5, 0.4)  # Adjust as needed
67 upper_purple <- c(0.8, 1.0, 1.0)  # Adjust as needed
68
69 # Define the model input layer
70 input_image <- layer_input(shape = c(img_height, img_width, 3))
71
72 # Convert the input image to HSV
73 input_image_hsv <- tf$image$rgb_to_hsv(input_image)
74
75 # Create a binary mask for purple regions (lymphoblasts) based on the HSV
       threshold
76 mask_purple <- tf$math$logical_and(
77   input_image_hsv[,,,1] >= lower_purple[1], input_image_hsv[,,,1] <=
       upper_purple[1]
78 )
79 mask_purple <- tf$math$logical_and(mask_purple, input_image_hsv[,,,2] >=
       lower_purple[2])
80 mask_purple <- tf$math$logical_and(mask_purple, input_image_hsv[,,,3] >=
       lower_purple[3])
81
82 # Convert logical mask to float and expand dims to match RGB channels
83 mask_purple <- tf$cast(mask_purple, dtype = tf$float32)  # Ensure the mask is a
       float tensor
84 mask_purple <- tf$expand_dims(mask_purple, axis = as.integer(-1)) # Add a channel
       dimension
85 mask_purple <- tf$tile(mask_purple, multiples = as.integer(c(1, 1, 1, 3)))  #
       Repeat for each color channel
86
87 # Apply the mask to the original RGB image
88 segmented_image <- tf$math$multiply(input_image, mask_purple)
89
90 # Pass the segmented image into segmentation block
```

```
# Modified U-Net inspired segmentation block with color-based segmentation
# Contracting Path (Encoder) with fewer filters
conv1 <- layer_conv_2d(segmented_image, filters = 32, kernel_size = c(3, 3),
    activation = "relu", padding = "same")
conv1 <- layer_conv_2d(conv1, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool1 <- layer_max_pooling_2d(conv1, pool_size = c(2, 2))

conv2 <- layer_conv_2d(pool1, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv2 <- layer_conv_2d(conv2, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool2 <- layer_max_pooling_2d(conv2, pool_size = c(2, 2))

conv3 <- layer_conv_2d(pool2, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv3 <- layer_conv_2d(conv3, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
pool3 <- layer_max_pooling_2d(conv3, pool_size = c(2, 2))

# Bottleneck layer with fewer filters
conv4 <- layer_conv_2d(pool3, filters = 256, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv4 <- layer_conv_2d(conv4, filters = 256, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Expansive Path (Decoder) with skip connections and fewer filters

# Upsampling + skip connection from conv3
upconv3 <- layer_conv_2d_transpose(conv4, filters = 128, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat3 <- layer_concatenate(list(upconv3, conv3))
conv5 <- layer_conv_2d(concat3, filters = 128, kernel_size = c(3, 3), activation
    = "relu", padding = "same")
conv5 <- layer_conv_2d(conv5, filters = 128, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Upsampling + skip connection from conv2
upconv2 <- layer_conv_2d_transpose(conv5, filters = 64, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat2 <- layer_concatenate(list(upconv2, conv2))
conv6 <- layer_conv_2d(concat2, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv6 <- layer_conv_2d(conv6, filters = 64, kernel_size = c(3, 3), activation =
    "relu", padding = "same")

# Upsampling + skip connection from conv1
upconv1 <- layer_conv_2d_transpose(conv6, filters = 32, kernel_size = c(3, 3),
    strides = c(2, 2), padding = "same")
concat1 <- layer_concatenate(list(upconv1, conv1))
conv7 <- layer_conv_2d(concat1, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same")
conv7 <- layer_conv_2d(conv7, filters = 32, kernel_size = c(3, 3), activation =
    "relu", padding = "same" )

# Feature extraction using ConvNeXT
features <- base_model(input_image)
features <- layer_global_average_pooling_2d(features)

```

```
# Flatten and concatenate segmentation output with ConvNeXT features
segmentation_features <- layer_flatten(conv5)
combined_features <- layer_concatenate(list(features, segmentation_features))

# Dense layers with L2 regularization, batch normalization, and increased dropout
x <- layer_dense(combined_features, units = 512, activation = "relu",
    kernel_regularizer = regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)  # Increased dropout rate to 0.7

x <- layer_dense(x, units = 256, activation = "relu", kernel_regularizer =
    regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)  # Increased dropout rate to 0.7

x <- layer_dense(x, units = 128, activation = "relu", kernel_regularizer =
    regularizer_l2(0.01))
x <- layer_batch_normalization(x)
x <- layer_dropout(x, rate = 0.7)  # Increased dropout rate to 0.7

output <- layer_dense(x, units = length(unique(full_df$class)), activation =
    "softmax")

# Create the model
model <- keras_model(inputs = input_image, outputs = output)

# Compile the model with a lower learning rate to prevent overfitting
model %>% compile(
  loss = loss_categorical_crossentropy(),
  optimizer = optimizer_adam(learning_rate = 1e-4),  # Lower learning rate to
    prevent overfitting
  metrics = c("accuracy")
)

print(model)

# Define the class-to-index mapping
classes <- c("Benign", "Early", "Pre", "Pro")
class_to_index <- setNames(0:(length(classes) - 1), classes)

# Add numeric columns to both full and training DataFrames
full_df$class_numeric <- as.numeric(factor(full_df$class, levels = classes)) - 1
train_df$class_numeric <- as.numeric(factor(train_df$class, levels = classes)) - 1

# Verify the class-to-index mapping
cat("Class-to-Index Mapping:\n")
print(class_to_index)

# Create improved mapping displays with counts
cat("\nMappings and counts in full dataset:\n")
unique_mapping_full <- data.frame(
  original_class = classes,
  numeric_class = 0:(length(classes)-1),
  count = as.vector(table(full_df$class))
)
print(unique_mapping_full)

cat("\nMappings and counts in training dataset:\n")
unique_mapping_train <- data.frame(
```

```
  original_class = classes,
  numeric_class = 0:(length(classes)-1),
  count = as.vector(table(train_df$class))
)
print(unique_mapping_train)

# Calculate class weights using the train dataset
n_samples <- nrow(train_df)
n_classes <- length(classes)

# Get class frequencies from train dataset
class_counts <- table(train_df$class_numeric)

# Calculate balanced weights
class_weights <- n_samples / (n_classes * class_counts)

# Convert to a named list with numeric indices as names
class_weights <- as.list(class_weights)
names(class_weights) <- as.character(0:(n_classes-1))

# Print diagnostics
cat("\nClass distribution in full dataset:\n")
print(table(full_df$class))
cat("\nClass distribution in training dataset:\n")
print(table(train_df$class))

cat("\nFinal Class Weights (based on train dataset):\n")
print(class_weights)

# Validation steps
cat("\nValidation:\n")
cat("Total samples in full dataset:", n_samples, "\n")
cat("Sum of class counts:", sum(class_counts), "\n")
cat("Number of classes:", n_classes, "\n")
cat("Maximum weight ratio:", max(unlist(class_weights)) /
    min(unlist(class_weights)), "\n\n")

# Additional validation checks
cat("Verification of weight calculation:\n")
for(i in 0:(n_classes-1)) {
  class_name <- classes[i+1]
  weight <- class_weights[[as.character(i)]]
  count <- class_counts[as.character(i)]
  cat(sprintf("%s: count=%d, weight=%.4f, count*weight=%.4f\n",
              class_name, count, weight, count*weight))
}

# Create a data generator with augmentation (no augmentation on validation set)
datagen_train <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,  # Increased rotation range for more variability
  width_shift_range = 0.4,  # Increased shift range for more variability
  height_shift_range = 0.4,  # Increased shift range for more variability
  shear_range = 0.4,  # Increased shear range for more variability
  zoom_range = 0.4,  # Increased zoom range for more variability
  horizontal_flip = TRUE
)

datagen_val <- image_data_generator(
```

```
  rescale = 1/255
)

datagen_test <- image_data_generator(
  rescale = 1/255
)

# Custom generator for training and validation (no augmentation on validation set)
train_generator <- flow_images_from_dataframe(
  train_df,
  directory = data_dir,
  x_col = "image_filename",
  y_col = "class",
  target_size = c(img_height, img_width),
  batch_size = batch_size,
  class_mode = "categorical",
  generator = datagen_train
)

val_generator <- flow_images_from_dataframe(
  val_df,
  directory = data_dir,
  x_col = "image_filename",
  y_col = "class",
  target_size = c(img_height, img_width),
  batch_size = batch_size,
  class_mode = "categorical",
  generator = datagen_val,
  shuffle = FALSE,
)

test_generator <- flow_images_from_dataframe(
  test_df,
  directory = data_dir,
  x_col = "image_filename",
  y_col = "class",
  target_size = c(img_height, img_width),
  batch_size = batch_size,
  class_mode = "categorical",
  generator = datagen_test,
  shuffle = FALSE,
)

# Initialize an empty vector to store the learning rate values
lr_values <- numeric(0)

# Custom callback to track the learning rate during training
callback_lr_tracker <- callback_lambda(
  on_epoch_begin = function(epoch, logs) {
    lr_values <<- c(lr_values, as.numeric(model$optimizer$lr))  # Store learning
    rate at the start of each epoch
  }
)


# Train the model with class weights and validation
history <- model %>% fit(
  train_generator,
  epochs = epochs,
```

```
  steps_per_epoch = nrow(train_df) / batch_size,
  validation_data = val_generator,
  validation_steps = nrow(val_df) / batch_size,
  class_weight = class_weights,
  callbacks = list(
    callback_early_stopping(monitor = "val_loss", patience = 8,
    restore_best_weights = TRUE),  # Increased patience
    callback_reduce_lr_on_plateau(monitor = "val_loss", factor = 0.1, patience =
    5),
    callback_model_checkpoint("best_model.h5", monitor = "val_loss",
    save_best_only = TRUE),  # Save the best model
    callback_lr_tracker
  )
)



# Retrieve the class labels as defined in the test generator
class_indices <- test_generator$class_indices  # This is a named list
# Reverse the class_indices to map indices back to labels
index_to_label <- names(class_indices)[order(unlist(class_indices))]

# Evaluate the model on the test data
evaluation_metrics <- model %>% evaluate(
  test_generator,
  steps = nrow(test_df) / batch_size
)

# Extract the evaluation metrics (test loss and accuracy)
test_loss <- evaluation_metrics[1]
test_accuracy <- evaluation_metrics[2]

cat("Test Loss:", test_loss, "\n")
cat("Test Accuracy:", test_accuracy, "\n")

# Calculate the number of steps for test predictions
steps <- ceiling(nrow(test_df) / batch_size)

# Make predictions on the test set
predictions <- model %>% predict(test_generator, steps = steps)

# Map the predicted indices to class labels using index_to_label
predicted_classes <- factor(apply(predictions, 1, function(x)
    index_to_label[which.max(x)]), levels = index_to_label)

# Get true labels from the test data frame (already in factor form)
true_classes <- factor(test_df$class, levels = index_to_label)

# Calculate the confusion matrix
conf_matrix <- confusionMatrix(predicted_classes, true_classes)

# Print confusion matrix details
print(conf_matrix)

cat("\nClass-Specific Statistics:\n")
print(conf_matrix$byClass)

cat("\nConfusion Matrix Mode:\n")
print(conf_matrix$mode)
```

```
358
359 cat("\nAdditional Arguments Passed to confusionMatrix():\n")
360 print(conf_matrix$dots)
361
362 # Extract the confusion matrix table as a data frame
363 cm_table <- as.data.frame(as.table(conf_matrix))
364
365 # Rename the columns for easier reference
366 colnames(cm_table) <- c("Predicted", "Actual", "Count")
367
368 # Plot the confusion matrix as a heatmap using ggplot2
369 ggplot(cm_table, aes(x = Predicted, y = Actual, fill = Count)) +
370   geom_tile() +
371   scale_fill_gradient(low = "white", high = "blue") +
372   theme_minimal() +
373   labs(title = "Confusion Matrix Heatmap",
374        x = "Predicted Class",
375        y = "Actual Class") +
376   theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
377   geom_text(aes(label = Count), color = "black", size = 5)  # Add text labels
      with the count in each tile
378
379 plot(history)
380
381                                    # Assuming you have already trained the model
    and have the 'history' object
382 library(ggplot2)
383
384 # Create a data frame for plotting
385 history_df <- data.frame(
386   epoch = 1:length(history$metrics$loss),
387   loss = history$metrics$loss,
388   val_loss = history$metrics$val_loss,
389   accuracy = history$metrics$accuracy,
390   val_accuracy = history$metrics$val_accuracy,
391   lr = lr_values   # Add the learning rate history
392 )
393
394 # Create history_df to include loss, accuracy, and learning rate values
395 history_df <- data.frame(
396   epoch = 1:length(history$metrics$loss),
397   loss = history$metrics$loss,
398   val_loss = history$metrics$val_loss,
399   accuracy = history$metrics$accuracy,
400   val_accuracy = history$metrics$val_accuracy,
401   lr = lr_values   # Add the learning rate history
402 )
403
404 library(ggplot2)
405 library(patchwork)
406
407 # Linear scale plots
408 plot1 <- ggplot(history_df, aes(x = epoch)) +
409   geom_line(aes(y = loss, color = 'Training Loss')) +
410   geom_line(aes(y = val_loss, color = 'Validation Loss')) +
411   geom_line(aes(y = accuracy * 100, color = 'Accuracy (%)')) +
412   scale_y_continuous(sec.axis = sec_axis(~./100, name = "Accuracy (%)")) +
413   labs(title = "All Metrics (Linear Scale)", x = "Epoch", y = "Value") +
414   theme_minimal() +
```

```
  scale_color_manual(values = c("blue", "red", "green"))

# Logarithmic scale plots
plot2 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  geom_line(aes(y = accuracy * 100, color = 'Accuracy (%)')) +
  scale_y_log10() +
  labs(title = "All Metrics (Logarithmic Scale)", x = "Epoch", y = "Value (log
    scale)") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red", "green"))

# Loss linear scale
plot3 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  labs(title = "Losses (Linear Scale)", x = "Epoch", y = "Loss") +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red"))

# Loss logarithmic scale
plot4 <- ggplot(history_df, aes(x = epoch)) +
  geom_line(aes(y = loss, color = 'Training Loss')) +
  geom_line(aes(y = val_loss, color = 'Validation Loss')) +
  scale_y_log10() +
  labs(title = "Losses (Logarithmic Scale)", x = "Epoch", y = "Loss (log scale)")
    +
  theme_minimal() +
  scale_color_manual(values = c("blue", "red"))

# Combine all plots into a single grid
combined_plot <- (plot1 | plot2) / (plot3 | plot4) +
  plot_annotation(title = "Training Progress")

# Print combined plot
print(combined_plot)

# Get final training and validation accuracy from the history object
final_train_accuracy <- history$metrics$accuracy[length(history$metrics$accuracy)]
final_val_accuracy <-
    history$metrics$val_accuracy[length(history$metrics$val_accuracy)]

# Extract the final training loss from the history object
final_train_loss <- history$metrics$loss[length(history$metrics$loss)]
final_val_loss <- history$metrics$loss[length(history$metrics$val_loss)]

# Print the final training loss
cat("Final Training Loss:", final_train_loss, "\n")
cat("Final Validation Loss:", final_val_loss, "\n")
cat("Final Train Accuracy:", final_train_accuracy, "\n")
cat("Final Validation Accuracy:", final_val_accuracy, "\n")

# Evaluating classification cost
# Define the cost matrix (replace with your actual costs)
cost_matrix <- matrix(
  c(0, 10, 20, 30,   # Benign -> Benign, Early, Pre, Pro
    10, 0, 15, 25,    # Early -> Benign, Early, Pre, Pro
    20, 15, 0, 10,    # Pre -> Benign, Early, Pre, Pro
```

```
    30, 25, 10, 0),   # Pro -> Benign, Early, Pre, Pro
  nrow = 4, byrow = TRUE
)
rownames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")
colnames(cost_matrix) <- c("Benign", "Early", "Pre", "Pro")

# Extract confusion matrix data
cm_table <- conf_matrix$table
classes <- rownames(cm_table)

visualize_cost_confusion_matrix <- function(conf_matrix, cost_matrix) {
  # Extract the confusion matrix from caret's result
  cm_table <- conf_matrix$table

  # Create cost-weighted matrix
  cost_weighted_cm <- matrix(0,
                             nrow = nrow(cost_matrix),
                             ncol = ncol(cost_matrix))

  for (i in 1:nrow(cm_table)) {
    for (j in 1:ncol(cm_table)) {
      if (i != j) {
        cost_weighted_cm[i, j] <- cm_table[i, j] * cost_matrix[i, j]
      }
    }
  }

  # Set row and column names
  rownames(cost_weighted_cm) <- rownames(cost_matrix)
  colnames(cost_weighted_cm) <- colnames(cost_matrix)

  return(cost_weighted_cm)
}

# Usage
cost_weighted_matrix <- visualize_cost_confusion_matrix(conf_matrix, cost_matrix)
cat("\nCost-Weighted Misclassification Matrix:\n")
print(cost_weighted_matrix)

# Initialize total misclassification cost
total_cost <- 0

# Calculate total misclassification cost by iterating over confusion matrix
    entries
for (i in 1:nrow(cm_table)) {
  for (j in 1:ncol(cm_table)) {
    if (i != j) {  # Misclassifications (i != j)
      total_cost <- total_cost + cm_table[i, j] * cost_matrix[i, j]
    }
  }
}

# Output the total misclassification cost
cat("Total Misclassification Cost:", total_cost, "\n")

# Calculate average cost per sample
total_samples <- sum(cm_table)
avg_cost_per_sample <- total_cost / total_samples
cat("Average Cost Per Sample:", avg_cost_per_sample, "\n")
```

```
529
530 # Calculate cost-sensitive accuracy
531 # Compute the maximum possible cost (if all instances were misclassified to the
      most costly class)
532 max_possible_cost <- sum(cost_matrix) * (total_samples / nrow(cost_matrix))
533 cost_sensitive_accuracy <- 1 - (total_cost / max_possible_cost)
534 cat("Cost-Sensitive Accuracy:", cost_sensitive_accuracy, "\n")
535
536 # Using cost matrix for cost-weighted F1 calculation
537 f1_scores <- conf_matrix$byClass[, "F1"]
538 weights <- 1 / (diag(cost_matrix) + 1)  # Add 1 to avoid divide-by-zero
539 weights <- weights / sum(weights)
540 weighted_f1 <- sum(f1_scores * weights, na.rm = TRUE)
541 cat("Weighted F1-score", weighted_f1, "\n")
542
543 # Save the trained model as an HDF5 file
544 save_model_hdf5(model, "convnext_model.h5")
```