VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

———————— * ————————

PROJECT REPORT

Image Classification

COUNCIL: Computer Science

—o0o—

Tran Tuan Kiet - 2252410
Vo Nguyen Phat - 2252607
Nguyen Hoang Quan - 2252681
Phan Hong Quan - 2252685

HO CHI MINH CITY, 02/2025

Contents

List of Figures

Chapter 1

Distribution and Contribution

Assignment 1

Student_ID	Student	Contribution	Role
2252681	Nguyen Hoang Quan	100%	Report & Preprocessing & Decision Tree model
2252685	Phan Hong Quan	100%	Naives Bayes and Genetic algorithm & Report
2252410	Tran Tuan Kiet	100%	Bayesian Network model & Hidden Makov model
2252607	Vo Nguyen Phat	100%	Neural Network & Report

Table 1.1: Table of distribution and contribution in assignment 1

Assignment 2

Student_ID	Student	Contribution	Role
2252681	Nguyen Hoang Quan	100%	Support Vector Machine, PCA, Ensemble method & Write report
2252607	Vo Nguyen Phat	100%	Discriminative models, Graphical network, Organize github repository & Write report

Table 1.2: Table of distribution and contribution in assignment 2

Chapter 2

Introduction

2.1 Background

The PathMNIST dataset was introduced to advance the application of deep learning in the field of medical image analysis, specifically in predicting patient survival based on colorectal cancer histology slides. It was developed as part of a research study focused on improving survival prediction from digital pathology images, using machine learning models that could provide faster and more accurate analyses than traditional methods. PathMNIST serves as a benchmark for evaluating machine learning models on a large-scale set of hematoxylin and eosin stained (H&E) histological images, typically used in pathology for cancer diagnosis.

In its construction, PathMNIST uses 100,000 non-overlapping image patches (NCT-CRC-HE-100K) derived from colorectal cancer tissue and has an additional test set (CRC-VAL-HE-7K), which provides 7,180 image patches from a different clinical center. This dataset has been designed for multi-class classification, with 9 distinct tissue types. The images are resized from their original size of 3 x 224 x 224 pixels into a smaller 3 x 28 x 28 pixels to make it manageable for classification tasks. The study revealed that accurately predicting the survival of patients from colorectal cancer histology images could be achieved using deep learning methods, particularly convolutional neural networks (CNNs).

PathMNIST is part of the MedMNIST v2 dataset, which includes a large number of biomedical image datasets for classification tasks across different data modalities, ranging from binarymulti-class classification to multi-label and ordinal regression. It is designed with a primary focus on biomedical image classification and aims to provide a standardized, lightweight, and easy-to-use benchmark for evaluating various machine learning algorithms, particularly deep learning models.

The dataset has significant educational value and facilitates research in the field of artificial intelligence (AI) applied to healthcare, particularly in medical imaging. As an MNIST-like dataset for medical images, it offers an ideal starting point for researchers and educators looking to develop, benchmark, and compare new machine learning models for biomedical image analysis.

PathMNIST also contributes to overcoming the challenge of dataset standardization in biomedical image analysis, ensuring that a large, diverse, and easily accessible dataset is available to train and evaluate machine learning models, ultimately advancing the generalization capabilities of AI systems in healthcare applications.

2.2 Problem Statement

The primary goal of this study is to develop a robust and optimized machine learning pipeline for **colorectal cancer survival prediction** using **histology slide images** from the PathMNIST dataset. This dataset contains a diverse set of tissue images, and the challenge lies in extracting meaningful features from these images and leveraging them to predict patient survival outcomes.

The specific objectives of this study are as follows:

- **Efficient Data Preprocessing**:
  - Develop strategies for handling large-scale image data, ensuring efficient loading, transformation, and normalization of images.
  - Address any missing or noisy data and standardize it for use in machine learning models.
  - Implement feature engineering to extract both image-based and statistical features.
- **Effective Feature Extraction**:
  - Extract relevant features from histology slide images, considering both image content (e.g., texture, intensity) and statistical measures (e.g., mean, variance).
  - Utilize techniques such as sliding windows, mean, and variance to generate a comprehensive feature set for each image.
- **Model Implementation and Hyperparameter Tuning**:
  - Implement and train all models that already give in the course, leveraging advanced techniques to ensure generalization across the dataset.
  - Perform hyperparameter tuning using grid search or other optimization strategies to improve model accuracy and performance.
  - Evaluate different classification models for the best predictive performance, with an emphasis on accuracy, precision, recall, and F1 score.
- **Cross-validation and Performance Analysis**:
  - Use cross-validation techniques to evaluate model performance, ensuring robust estimates of accuracy across different data splits.
  - Implement key performance metrics, such as confusion matrices, to assess the model's effectiveness in predicting survival outcomes.
- **Model Persistence and Scalability**:
  - Ensure model persistence functionality to save and reload trained models for future use, thereby facilitating reuse and deployment.
  - The solution should be scalable, handling the large size of the dataset effectively and ensuring that performance is maintained even as the dataset or model complexity increases.

By using the **PathMNIST dataset**, this study aims to identify an effective machine learning model capable of accurately classifying tissue images and predicting patient survival outcomes.

Chapter 3

Data Preprocessing

3.1 Overview

An essential part of any machine learning pipeline is data preprocessing. The goal of preprocessing is to transform the raw PathMNIST dataset into a clean, organized, and training-optimized form. This process helps ensure that the dataset is ready for model training and that the features extracted from the data are suitable for achieving the best possible model performance. In this project, medical images are processed through several key preprocessing steps:

- **Data Loading:** The first step involves loading the images and their corresponding labels. The dataset is structured, and it is split into training and test sets to ensure proper model evaluation.
- **Normalization:** To ensure that pixel values are on a consistent scale and do not vary widely between images, we normalize the images by scaling the pixel values. This step is critical for improving model performance and ensuring that all pixel intensities contribute equally to the model's learning process.
- **Feature engineering:** This involves the extraction of additional statistical features from the images, such as mean pixel intensity (which represents overall brightness) and variance (which captures contrast and texture). These engineered features provide the model with a richer set of inputs, helping it to learn more complex patterns in the data and improve classification performance.
- **Feature selection:**After feature engineering, dimensionality reduction and feature selection techniques such as PCA (Principal Component Analysis) and SelectKBest are applied to reduce the feature space and keep only the most informative features. This reduces the complexity of the model while retaining critical information that contributes to accurate predictions.
- **Data Saving:** To avoid the redundancy of repeating preprocessing steps, the preprocessed data is saved in a compressed format. This allows for faster future experiments, as the preprocessed data can be directly loaded without requiring reprocessing. This step also optimizes memory usage, making it easier to store and retrieve the data efficiently during subsequent phases of model training and evaluation.

3.2 Data Handling & Transformation

3.2.1 Loading Dataset

The PathMNIST dataset is loaded using the MedMNIST library, which provides structured medical image data. The dataset is split into training and test sets to ensure proper evaluation of the model's performance. This split allows us to train the model on one subset of data and evaluate its performance on unseen data, minimizing the risk of overfitting.

3.2.2 Normalization & Flattening

To ensure consistent pixel intensity across images, the images are normalized by scaling pixel values to a range between 0 and 1 (by dividing each pixel by 255). This normalization ensures that pixel intensity differences do not disproportionately affect model performance, helping to stabilize and speed up the training process. Additionally, each image is flattened into a 1D array to transform the image data into a tabular form suitable for traditional machine learning models.

Why Normalize:

- Prevents pixel intensity differences from affecting model performance.
- Helps achieve stable and faster training.

Why Flatten?

- Flattening is required in most machine learning models because they expect 1D vectorized (tabular) data rather than 2D image matrices. Flattening transforms the 2D image into a 1D vector, making the data compatible with these models. It is essential for ensuring that pixel values are treated as individual features, allowing the model to process the data effectively..

3.3 Feature Engineering

To enhance classification performance, additional statistical features are extracted:

- Mean pixel intensity – Represents overall brightness.
- Variance of pixel values – Captures contrast and texture.

These features provide additional information beyond raw pixel values, helping the model learn patterns more effectively

3.4 Feature Selection

Unlike tabular datasets where irrelevant features can be removed, this dataset is enhanced by adding useful statistical features, such as the mean and variance. These added features provide additional information that aids the model in making more accurate predictions.

Further, dimensionality reduction is applied to reduce the size of the feature space while retaining most of the important information:

- PCA (Principal Component Analysis) is applied to reduce the number of features, while still preserving the majority of the variance in the dataset. This step is crucial for reducing the computational complexity of training while ensuring that the model can generalize effectively.

Additionally, SelectKBest with ANOVA F-test (f_classif) is used to select the top 50 features from the dataset, further optimizing the feature space and potentially improving the model's performance.

3.5 Handling Missing Data

- No missing values exist in the image dataset, as each sample is a complete image.
- If corrupted or missing images were present, they would be removed or interpolated, but this was not necessary.

3.6 Saving Preprocessed Data

To avoid redundant preprocessing in future experiments, the transformed dataset is stored in a compressed format. This approach allows for fast access to the preprocessed data, ensuring that future experiments can directly load the data without having to repeat the preprocessing steps.

- Reduces computation time by storing the preprocessed data in a compact form.
- Optimizes memory usage with efficient file storage, making it easier to manage and access the dataset during the model training and evaluation phases.

Chapter 4

Model Implementation

In this phase, we implement all of the models, ensuring correctness in algorithm selection, parameter handling, training, prediction, and model persistence.

4.1 Decision tree

4.1.1 Model Hypothesis

The hypothesis for the Decision Tree model: The Decision Tree model will efficiently classify images based on their features, such as texture, intensity, and statistical measures (e.g., mean and variance). By recursively splitting the dataset at each node, the model will maximize Information Gain to improve homogeneity within each subset of the data, resulting in better prediction accuracy. The tree will grow by selecting the best features and splitting criteria, balancing between model complexity and performance. Hyperparameters like maximum depth, minimum samples per split, and criterion will be tuned to avoid overfitting and to ensure generalization across the dataset.

Formular:

$$IG(D,A) = Entropy(D) - \sum_{v \in \text{values}(A)} \frac{|D_v|}{|D|} \cdot Entropy(D_v)$$

Where:

- D is the dataset being split.
- A is the feature being evaluated for the split.
- $\text{values}(A)$ are the distinct values of feature A.
- D_v is the subset of D where feature A has value v.
- $Entropy(D) = -\sum_{i=1}^{C} p_i \log_2(p_i)$ is the entropy of dataset D, where p_i is the probability of class i in D.

The Decision Tree algorithm will choose the feature that maximizes the **Information Gain**, improving the homogeneity of the data within each node and leading to better predictive accuracy.

4.1.2 Training Implementation

The model is trained using a 5-fold cross-validation approach (cv=5) to enhance generalization.

Training Steps:

- Load preprocessed training data from .npz format.
- Fit GridSearchCV to explore various hyperparameter combinations.
- Select the best estimator based on cross-validation accuracy.

4.1.3 Parameter Handling:

Hyperparameter tuning is conducted using GridSearchCV to find the optimal configuration.The key parameters explored:

```
param_grid = {
    "criterion": ["gini", "entropy"],
    "max_depth": [5, 10, 15],
    "min_samples_split": [2, 5],
    "min_samples_leaf": [1, 2]
}

dt_classifier = DecisionTreeClassifier()

grid_search = GridSearchCV(estimator=dt_classifier, param_grid=param_grid,
    cv=5, scoring='accuracy')

# Fit the model (this finds the best hyperparameters)
grid_search.fit(x_train_with_stats, y_train)

# Get best model
best_model = grid_search.best_estimator_
```

- Criterion: [”gini”, ”entropy”] – Defines the strategy for node splitting.
- Max Depth: [5, 10, 15] – Controls tree complexity and prevents overfitting.
- Min Samples Split: [2, 5] – Minimum number of samples required to split an internal node.
- Min Samples Leaf: [1, 2] – Minimum number of samples required to be a leaf node.

4.1.4 Model Persistence

To ensure reusability and efficiency, the trained Decision Tree model is saved using joblib, allowing future predictions and evaluations without the need for retraining.

Purpose:

- Facilitates model deployment and reproducibility.
- Enables quick loading for inference without re-running the training pipeline.

- Ensures consistency in performance evaluation across different experiments.

By storing the model, we streamline workflow efficiency and maintain version control for future improvements.

4.1.5 Prediction Implementation

The trained model is used to predict labels on the test dataset.

```
# Make predictions on test data
y_pred = best_model.predict(x_test_with_stats)

# Calculate Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy:.4f}")

# Generate Classification Report
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)

# Generate Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
```

- Accuracy Score – Overall model correctness.
- Precision, Recall, and F1-score – Measured using a classification report.
- Confusion Matrix – Visualizes misclassified instances.

4.1.6 Evaluation Results

Comprehensive model evaluation is conducted to assess performance and identify areas for improvement. The results are stored for further analysis and reference.

Classification Report:

- Contains accuracy, precision, recall, and F1-score for each class.

Confusion Matrix:

- Visual representation of correct vs. misclassified samples.
- Helps identify class imbalances and areas where the model struggles.

These evaluation metrics provide valuable insights into the model's effectiveness and guide future optimizations.

4.2 Naive Bayes with Genetic Algorithms

4.2.1 Model Hypothesis

A Decision Tree for classification partitions the input space into disjoint regions and assigns a class label to each region. The hypothesis function h can be expressed as:

$$h(\mathbf{x}) = \sum_{j=1}^{J} c_j \mathbb{1}(\mathbf{x} \in R_j)$$

, where:

- J is the number of leaf nodes (terminal nodes) in the tree.
- R_j represents the regions corresponding to the leaf nodes.
- c_j is the predicted class label for inputs belonging to region R_j.
- $\mathbb{1}(\mathbf{x} \in R_j)$ is an indicator function that equals 1 if $\mathbf{x}$ belongs to region R_j, otherwise 0.

The class label assigned to each region R_j is determined by:

$$c_j = \arg\max_k P(y = k \mid \mathbf{x} \in R_j)$$

where:

- k represents the possible class labels.
- $P(y = k \mid \mathbf{x} \in R_j)$ is the estimated probability of class k given that $\mathbf{x}$ belongs to region R_j.

4.2.2 Training Implementation

We preprocess the images by flattening them and applying PCA for dimensionality reduction. The dataset is split into 80% training and 20% validation to evaluate the GA.

The dataset loading and splitting process is implemented as follows:

Code 4.1: Dataset Loading and Splitting

```
from sklearn.model_selection import train_test_split
import numpy as np

data = np.load("train_data.npz")
x_train_full, y_train_full = data["x_train"], data["y_train"].ravel()
x_train, x_val, y_train, y_val = train_test_split(x_train_full, y_train_full,
    test_size=0.2, random_state=42)

data = np.load("test_data.npz")
x_test, y_test = data["x_test"], data["y_test"].ravel()
```

4.2.3 Parameter Handling

The Genetic Algorithm optimizes the *var_smoothing* parameter within the range 10^{-10} to 10^{0}. The fitness function is defined based on validation accuracy.

Code 4.2: Genetic Algorithm Implementation

```
import random
from deap import base, creator, tools, algorithms
from sklearn.naive_bayes import GaussianNB

def fitness_function(params):
    var_smoothing = 10 ** params[0]
    model = GaussianNB(var_smoothing=var_smoothing)
    model.fit(x_train, y_train)
    accuracy = model.score(x_val, y_val)
    return (accuracy,)

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.uniform, -10, 0)
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_float, n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", fitness_function)
toolbox.register("mate", tools.cxBlend, alpha=0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.1, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

population = toolbox.population(n=200)
algorithms.eaSimple(population, toolbox, cxpb=0.5, mutpb=0.2, ngen=100,
    stats=None, halloffame=None, verbose=True)

best_ind = tools.selBest(population, k=1)[0]
best_var_smoothing = 10 ** best_ind[0]
print(f"Best var_smoothing: {best_var_smoothing}")
```

4.2.4 Model Persistence

The best-found Naive Bayes model is saved using the joblib library for future use:

Code 4.3: Saving the Best Model

```
import joblib

final_model = GaussianNB(var_smoothing=best_var_smoothing)
final_model.fit(x_train, y_train)
model_save_path = "naive_bayes_with_ga_best.pkl"
joblib.dump(final_model, model_save_path)
print(f"Model saved successfully at: {model_save_path}")
```

4.2.5 Prediction Implementation

After training, the final model predicts labels for the test set. The performance is evaluated using accuracy and classification metrics:

Code 4.4: Model Evaluation

```
from sklearn.metrics import classification_report

y_pred = final_model.predict(x_test)
print(classification_report(y_test, y_pred))
```

4.2.6 Evaluation Results

We compare the test accuracy of the optimized Naive Bayes model against the default implementation. The best hyperparameter found using GA is reported, and we analyze how well it improves over the baseline.

4.3 Neural Networks

4.3.1 Overview

This section focuses on implementing a neural network to classify images using a Multi-Layer Perceptron (MLP). The MLP is a type of feedforward neural network that is widely used for tasks such as classification and regression. It processes the input data through multiple layers of neurons, applying transformations and non-linear activations to learn patterns and relationships within the data. In this implementation, the MLP is designed to take processed image data as input and classify it into nine categories, leveraging its ability to model complex decision boundaries and features.

4.3.2 Code libraries

Code 4.5: Loading and inspecting training data

```
import numpy as np
import pandas as pd
import random
import torch
import pickle
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
from sklearn.metrics import accuracy_score, classification_report
```

This implementation of neural network, makes use of libraries as shown in code snippet 4.5

4.3.3 Splitting the dataset

Before constructing the architecture of a neural network, it is crucial to thoroughly analyze and reconfirm the input size, as well as the number of output labels, because these factors play a fundamental role in determining the overall structure and design of the model. The input size directly influences the number of neurons in the input layer, while the output size dictates complex the classification problem might be. Additionally, these dimensions make some impact on the choice of hidden layers, the number of neurons within them, and the types of operations or transformations required to effectively map the input data to the desired output.

Code 4.6: Loading and inspecting training data

```
train = np.load("./data/preprocessed/train_data.npz")
print(train['x_train'].shape)
print(train['y_train'].shape)
print(np.unique(train['y_train']))
```

Output:

```
(89996, 50)
(89996,)
[0 1 2 3 4 5 6 7 8]
```

The code snippet referenced in 4.6 demonstrates the process of loading and inspecting the dataset to verify its structure and dimensions. The dataset comprises 50 distinct features along with 9 distinct output labels.

Code 4.7: Loading and inspecting training data

```
X_train = torch.tensor(train["x_train"], dtype=torch.float32)
y_train = torch.tensor(train['y_train'], dtype=torch.long)
# 80% for training and 20% for validation
train_size = int(0.8 * len(X_train))
val_size = len(X_train) - train_size
train_data, val_data = random_split(TensorDataset(X_train, y_train),
    [train_size, val_size])
```

In this implementation, we split the training dataset into 80% for training and 20% for validation while training.

4.3.4 Architecture

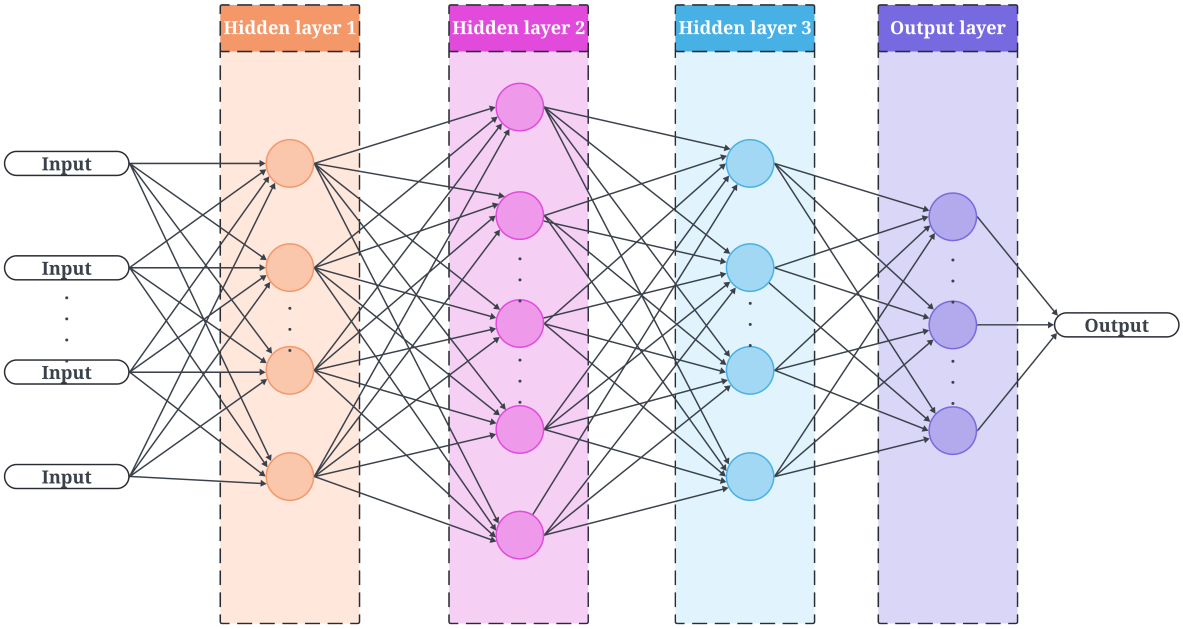

Figure 4.1: Neural Network Architecture

After analyzing the dimensions, we choose to have a neural networks with 1 input layer, 3 hidden layer and 1 output layer, and the architecture of the neural network is shown in Figure 4.1. However, we only know the size of input and output layer is 50 and 9 respectively. The size of hidden layers requires tests to get the most appropriate size for a neural network with 3 hidden layers.

Initially, we chose the number of neurons in the hidden layers to be 128, 64, and 32, respectively. This design follows a decreasing pattern in the number of neurons across layers. The rationale behind this is that the earlier layers capture broader, low-level features from the input data, while the later layers refine these features into more specific, high-level representations relevant to the task. By reducing the number of neurons in deeper layers, we encourage the network

to compress and focus on the most important features, reducing redundancy and improving generalization. Besides, reducing the number of neurons in deeper layers acts as a form of implicit dimensionality reduction. This helps the network focus on the most relevant features and discard redundant or less important information. It is also more efficient because it reduce the computational cost.

4.3.5 Model implementation

Code 4.8: Model implementation

```
class MLPModel(nn.Module):
def __init__(self, input_size=50, hidden_sizes=[256, 128, 64],
    output_size=9):
    super(MLPModel, self).__init__()
    self.model = nn.Sequential(
        nn.Linear(input_size, hidden_sizes[0]),
        nn.ReLU(),
        nn.Linear(hidden_sizes[0], hidden_sizes[1]),
        nn.Tanh(),
        nn.Linear(hidden_sizes[1], hidden_sizes[2]),
        nn.ReLU(),
        nn.Linear(hidden_sizes[2], output_size),
    )

def forward(self, x):
    return self.model(x)

model = MLPModel()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
```

Code snippet 4.8 show the implementation of the neural network, a sequential architecture. It consists of an input layer, three hidden layers, and an output layer. The input layer takes in a feature vector of size 50, which is then passed through the first hidden layer with 256 neurons and a ReLU activation function. The output of this layer is fed into the second hidden layer, also with 128 neurons, but this time using a Tanh activation function. The third hidden layer again has 64 neuron s and uses a ReLU activation function. Finally, the output layer produces a vector of size 9, which is likely used for a classification task with 9 possible classes. The network is trained using the CrossEntropyLoss function, which is suitable for classification tasks, and the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.1 and momentum of 0.9 to update the model's parameters during training.

4.3.6 Model training

Code 4.9: Model training

```
batch_size = 64
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)
```

```
def train_model(model, train_loader, val_loader, criterion, optimizer,
    epochs=200):
best_val_acc = 0.0
best_val_loss = float('inf')
model.train()
for epoch in range(epochs):
    total_loss = 0
    for batch_X, batch_y in train_loader:
        batch_X, batch_y = batch_X.to(device), batch_y.to(device)
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    # Validation phase
    model.eval()
    val_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_X, batch_y in val_loader:
            batch_X, batch_y = batch_X.to(device), batch_y.to(device)
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == batch_y).sum().item()
            total += batch_y.size(0)
    val_accuracy = correct / total
    avg_val_loss = val_loss / len(val_loader)
    model.train()

    # Save model if validation accuracy improves
    if val_accuracy > best_val_acc:
        best_val_acc = val_accuracy
        torch.save(model.state_dict(), "best_model_acc.pth")
        print(f"Model saved at epoch {epoch+1} with Val Acc:
            {val_accuracy:.4f}")

    # Save model if validation loss improves
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        torch.save(model.state_dict(), "best_model_loss.pth")
        print(f"Model saved at epoch {epoch+1} with Val Loss:
            {avg_val_loss:.4f}")

    print(f"Epoch {epoch+1}/{epochs}, Loss:
        {total_loss/len(train_loader):.4f}, Val Loss:
        {val_loss/len(val_loader):.4f}, Val Acc: {val_accuracy:.4f}")
```

```
    return best_val_acc, best_val_loss # Return validation accuracy and loss
```

Code snippet 4.9 shows the training process of the neural network. The training operates based on the dividing dataset into batches, with size of 64, and default total epochs of 200. In this process, we keep track of the models corresponding to the best validation loss and best validation accuracy as better validation loss indicate a better consistency while the other implies the performance of the model.

4.4 Bayesian Network Model

4.4.1 Model Hypothesis

The hypothesis for the Bayesian Network model: The Bayesian Network model will classify data based on feature relationships modeled as conditional dependencies between variables. By using an online learning approach, the model will update its parameters incrementally as new data is processed, leading to adaptive inference and improved predictions over time. The model will leverage GPU acceleration to efficiently handle large-scale datasets. Key hyperparameters like the smoothing factor (alpha) and batch size will be tuned for optimal model performance.

Formulation: The model computes the posterior probability $P(Y|X)$ using Bayes' theorem:

$$P(Y|X) = \frac{P(Y)\prod_{i=1}^{n} P(X_i|Y)}{P(X)}$$

Where:

- $P(Y)$ is the prior probability of the label.
- $P(X_i|Y)$ is the conditional probability of each feature X_i given the label Y.
- $P(X)$ is the marginal likelihood of the evidence X.

The Bayesian Network uses Conditional Probability Tables (CPTs) to model the dependencies between variables, and these are updated as new data is encountered.

4.4.2 Training Implementation

The model is trained using online learning, where data is processed in batches and the CPDs are updated incrementally. The key steps in the training process are as follows:

- Initialize the Bayesian Network structure with parent-child relationships between features and labels.
- Use the OnlineBayesianEstimator to update counts for each node based on batch data.
- Estimate CPDs after processing each batch.
- Train on the dataset in mini-batches, updating model parameters with each batch.

Training Procedure: The model is trained on the medical image dataset, which has been preprocessed using PCA for dimensionality reduction and discretization of features. The training data is shuffled and split into batches for incremental learning. The CPDs are updated iteratively for each batch of data.

Code Implementation: Here is the implementation of the Bayesian Network and Online Estimator:

```
import torch
import networkx as nx
import matplotlib.pyplot as plt
```

```
class BayesianNetwork:
    def __init__(self, edges, device="cpu"):
        """
        Bayesian Network with GPU support.
        """
        self.device = torch.device(device)
        self.graph = nx.DiGraph()
        self.graph.add_edges_from(edges)
        self.nodes = list(self.graph.nodes())
        self.parents = {node: list(self.graph.predecessors(node)) for node in
            self.nodes}
        self.cpts = {}

    def visualize(self):
        """Visualizes the Bayesian Network structure."""
        plt.figure(figsize=(8, 6))
        nx.draw(self.graph, with_labels=True, node_size=3000,
            node_color="lightblue", edge_color="gray", font_size=12)
        plt.title("Bayesian Network Structure")
        plt.show()
```

4.4.3 Parameter Handling

Hyperparameter tuning is crucial for ensuring the model performs well across different datasets. The key hyperparameters explored include:

- **Alpha (Laplace smoothing factor)**: Controls the smoothing applied to the CPDs.
- **Batch Size**: Defines the number of samples processed per batch during training.

Hyperparameter tuning is performed using cross-validation, and the best configuration is selected based on performance metrics.

Code Implementation: Here is the implementation of the Online Bayesian Estimator and how the counts are updated:

```
class OnlineBayesianEstimator:
    def __init__(self, model, alpha=1):
        self.model = model
        self.alpha = alpha # Laplace smoothing factor
        self.counts = {}

    def update_counts(self, batch_data):
        for node in self.model.nodes:
            parents = self.model.parents[node]
            if parents:
                grouped_data = batch_data.pivot_table(index=parents,
                    columns=node, aggfunc='size', fill_value=0)
            else:
                grouped_data = batch_data[node].value_counts().to_frame().T
```

```
            if node not in self.counts:
                self.counts[node] = grouped_data
            else:
                self.counts[node] += grouped_data # Accumulate counts across
                    batches

    def estimate_cpds(self):
        cpts = {}
        for node, count_matrix in self.counts.items():
            smoothed_counts = count_matrix + self.alpha
            cpt_tensor =
                torch.tensor(smoothed_counts.div(smoothed_counts.sum(axis=1),
                axis=0).values,
                                    dtype=torch.float32,
                                        device=self.model.device)
            cpts[node] = cpt_tensor

        self.model.cpts = cpts # Update model CPDs
```

4.4.4 Model Persistence

To ensure reproducibility and avoid retraining the model for future predictions, the trained Bayesian Network is saved using pickle. This allows for easy loading of the model in deployment environments.

Purpose:

- Facilitates quick model deployment for future inference tasks.
- Ensures that the model can be reused without retraining, saving computation time.
- Enables consistency in performance evaluation across different experiments.

Code Implementation: Here is how the trained model is saved for later use:

```
import pickle

# Save the best model based on F1-score
with open("best_bayesian_network.pkl", "wb") as f:
    pickle.dump(best_model, f)
```

4.4.5 Prediction Implementation

Once the model is trained, it is used to make predictions on the test dataset. The inference procedure involves the following steps:

- Extract the evidence (input features) from the test data.
- Compute the posterior probability distribution $P(Y|X)$ for each test instance.
- Assign the label with the highest posterior probability as the predicted label.

Code Implementation: Here is the implementation of the prediction function:

```
def predict(bn_model, test_data):
    inference = BayesianInference(bn_model)
    predictions = []

    for _, row in test_data.iterrows():
        evidence = row.to_dict()
        del evidence['label'] # Remove true label (to predict it)
        posterior_probs = inference.compute_posterior(evidence)
        predicted_label = max(posterior_probs, key=posterior_probs.get) #
            Argmax P(L | F)
        predictions.append(predicted_label)

    return torch.tensor(predictions, dtype=torch.int64, device=bn_model.device)
```

4.4.6 Evaluation Results

The performance of the trained model is evaluated using accuracy, precision, recall, and F1-score. These metrics provide insights into the model's effectiveness and help identify areas for improvement.

Accuracy and F1-Score:

- **Accuracy**: Measures the proportion of correct predictions over all predictions.
- **F1-Score**: Provides a balance between precision and recall, especially useful when dealing with imbalanced classes.

Confusion Matrix:

- A confusion matrix is used to visualize the number of correct vs. misclassified samples.
- It helps identify where the model struggles, such as misclassifying certain classes more often than others.

Classification Report:

- The classification report provides detailed metrics for each class, including precision, recall, and F1-score.
- It offers a comprehensive view of the model's performance on different categories in the dataset.

Code Implementation: Here's how the classification report and confusion matrix are generated:

```
from sklearn.metrics import classification_report, confusion_matrix

# Generate classification report
class_report = classification_report(test_df['label'], y_pred)
```

```
# Generate confusion matrix
conf_matrix = confusion_matrix(test_df['label'], y_pred)
```

4.5 Augmented Naive Bayes

4.5.1 Model Hypothesis

The hypothesis for the Augmented Naïve Bayes model: The Augmented Naïve Bayes (ANB) model enhances the standard Naïve Bayes classifier by incorporating feature dependencies learned through Mutual Information (MI). Unlike traditional Naïve Bayes, which assumes conditional independence between features, ANB learns the relationships between features and models them through a Bayesian Network. This structure improves classification accuracy by acknowledging that features may have interdependencies that influence the label.

The model hypothesizes that learning and leveraging the feature dependencies will improve its ability to predict the label, especially when the dataset contains interdependent features that cannot be independently treated.

4.5.2 Training Implementation

The training process of the ANB model involves two main steps: learning the structure of the Bayesian Network and training a Naïve Bayes classifier on the augmented features.

Learning the ANB Structure: The ANB structure is learned by computing the Mutual Information (MI) scores between the features and the label, as well as between the features themselves. A graph is created to represent the feature dependencies, and the Maximum Spanning Tree (MST) is extracted to capture the strongest dependencies. This MST is then converted into a Directed Acyclic Graph (DAG) that encodes the feature dependencies.

Training the Naïve Bayes Model: After learning the ANB structure, the features are augmented with the learned dependencies, and a Multinomial Naïve Bayes classifier is trained using the augmented features. Hyperparameter tuning is performed to select the best configuration based on the validation F1-score.

Code Implementation: The ANB structure learning and training process is as follows:

```
import numpy as np
import networkx as nx
from sklearn.feature_selection import mutual_info_classif
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV
import pickle
import os

# Learning the ANB structure
def learn_anb_structure(train_df):
    features = train_df.columns[:-1]
    label = 'label'

    # Compute Mutual Information between each feature and label
    mi_scores = mutual_info_classif(train_df[features], train_df[label])

    # Create a graph and add edges
```

```
    G = nx.Graph()
    for i, feature in enumerate(features):
        G.add_edge(label, feature, weight=mi_scores[i])

    # Compute Mutual Information between features
    for i, f1 in enumerate(features):
        for j, f2 in enumerate(features):
            if i < j:
                mi = mutual_info_classif(train_df[[f1]], train_df[f2])[0]
                G.add_edge(f1, f2, weight=mi)

    # Use Maximum Spanning Tree (MST)
    mst = nx.maximum_spanning_tree(G)

    # Convert to Directed Acyclic Graph (DAG)
    DAG = nx.DiGraph()
    for edge in mst.edges(data=True):
        parent, child, _ = edge
        DAG.add_edge(parent, child)

    return list(DAG.edges)

# Training the ANB Model
def train_anb(train_df, test_df, val_df, model_path="best_anb.pkl"):
    print("Starting ANB model training...")

    # Learn feature dependencies
    edges = learn_anb_structure(train_df)
    print(f"Learned ANB Structure: {edges}")

    # Train Nave Bayes model
    X_train, y_train = train_df.iloc[:, :-1], train_df.iloc[:, -1]
    X_test, y_test = test_df.iloc[:, :-1], test_df.iloc[:, -1]
    X_val, y_val = val_df.iloc[:, :-1], val_df.iloc[:, -1]

    param_grid = {
        'alpha': [0.01, 0.1, 0.5, 1.0, 2.0],
        'fit_prior': [True, False]
    }

    anb_model = MultinomialNB()
    grid_search = GridSearchCV(anb_model, param_grid, cv=5,
        scoring='f1_weighted')
    grid_search.fit(X_train, y_train)

    best_model = grid_search.best_estimator_
    print(f"Best hyperparameters: {grid_search.best_params_}")

    # Save the best model
    model_data = {
        'model': best_model,
```

```
        'feature_dependencies': edges,
        'metrics': grid_search.best_params_
    }

    save_path = os.path.join("models", model_path)
    os.makedirs(os.path.dirname(save_path), exist_ok=True)
    with open(save_path, 'wb') as f:
        pickle.dump(model_data, f)

    print(f"Model saved at: {save_path}")
    return model_data
```

4.5.3 Parameter Handling

The key hyperparameters for the Augmented Naïve Bayes model are the smoothing parameter (α) and the choice of whether to fit prior probabilities. These parameters are tuned using GridSearchCV to identify the best combination based on the validation F1-score.

- **Alpha** (α): The Laplace smoothing factor, which helps handle the possibility of zero probability estimates in the Naïve Bayes model.
- **Fit Prior**: Whether to learn the prior probabilities from the data or assume uniform priors.

Hyperparameter tuning is performed through GridSearchCV, which evaluates multiple combinations of α and fit_prior settings and selects the configuration that maximizes the weighted F1-score.

4.5.4 Model Persistence

After training, the best model is saved to a file using Python's 'pickle' module. This allows for future use of the model without needing to retrain it. The model is stored along with the feature dependencies and the hyperparameters used during training.

Code for Saving the Model:

```
# Save the best model with feature dependencies and metrics
with open(save_path, 'wb') as f:
    pickle.dump(model_data, f)
```

4.5.5 Prediction Implementation

Once the model is trained and saved, it can be used to predict labels for new data. The following code demonstrates how to load the model and use it for predictions on test data.

Code for Making Predictions:

```
# Load the saved model
with open("models/best_anb.pkl", 'rb') as f:
    model_data = pickle.load(f)
best_model = model_data['model']

# Make predictions on new test data
```

```
y_pred = best_model.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
```

4.5.6 Evaluation Results

The evaluation of the ANB model includes accuracy, precision, recall, and F1-score metrics on the validation set. These metrics are used to assess the model's performance.

Code for Evaluation Metrics:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score, classification_report

# Evaluate the model on test and validation datasets
y_test_pred = best_model.predict(X_test)
test_accuracy = accuracy_score(y_test, y_test_pred)

y_val_pred = best_model.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_pred)
precision = precision_score(y_val, y_val_pred, average='macro',
    zero_division=0)
recall = recall_score(y_val, y_val_pred, average='macro', zero_division=0)
f1 = f1_score(y_val, y_val_pred, average='macro', zero_division=0)

print("Validation Metrics:")
print(f" - Accuracy: {val_accuracy * 100:.2f}%")
print(f" - Precision: {precision:.4f}")
print(f" - Recall:  {recall:.4f}")
print(f" - F1 Score: {f1:.4f}")
```

These evaluation metrics are used to assess the ANB model's performance on the validation dataset and guide future improvements.

4.6 Hidden Markov Model

4.6.1 Model Hypothesis

The hypothesis for the Hidden Markov Model (HMM): The Hidden Markov Model (HMM) assumes that the observed data is generated by a hidden process with a set of unobserved states. The model's hypothesis is that each class in the dataset can be represented by a sequence of hidden states, and the observations are dependent on these hidden states. By modeling the temporal or sequential dependencies in the data, HMMs can capture patterns that are not directly observable but influence the observed data.

We aim to train a separate HMM for each class in the dataset using the GaussianHMM implementation from the 'hmmlearn' package. The number of hidden states (components) is treated as a hyperparameter and optimized using cross-validation.

4.6.2 Training Implementation

The training process of the HMM involves the following steps: 1. **Data Preprocessing**: We scale the feature data using 'StandardScaler' to ensure that the features are on the same scale. 2. **Hyperparameter Tuning**: We tune the number of hidden states for the HMM using cross-validation. The number of components is varied from 2 to 4, and the model is trained separately for each class in the dataset. 3. **Model Training**: The model is trained for each class using 'GaussianHMM' with different numbers of hidden states. We use the K-fold cross-validation approach to evaluate the model during training. 4. **Model Selection**: The best model is selected based on the average F1-score across all folds, and the best HMM is saved for future use.

Code Implementation:

```
from hmmlearn import hmm
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
import pickle
import os

class HiddenMarkovModelTrainer:
    def __init__(self):
        self.hmm_models = None # Store trained HMM models

    def train_hmm(self, train_df, test_df, val_df, n_components_range=[2, 3,
        4], random_state=42, model_path="hmm_models.pkl"):
        """
        Trains a Hidden Markov Model (HMM) for each class in the dataset with
            cross-validation and hyperparameter tuning.
        """
        print("Starting Hidden Markov Model (HMM) training...")

        # Extract features and labels
        X_train, y_train = train_df.iloc[:, :-1], train_df.iloc[:, -1]
        X_test, y_test = test_df.iloc[:, :-1], test_df.iloc[:, -1]
        X_val, y_val = val_df.iloc[:, :-1], val_df.iloc[:, -1]
```

```
# Normalize the feature data using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
X_val_scaled = scaler.transform(X_val)

# Perform K-fold cross-validation for hyperparameter tuning
kf = KFold(n_splits=5, shuffle=True, random_state=random_state)
best_hmm_models = {}
best_f1 = -1 # Initialize best F1 score as a very low value

# Iterate over possible n_components for hyperparameter tuning
for n_components in n_components_range:
    hmm_models = {}
    avg_f1_score = 0 # Average F1 score for this n_components across
        folds

    for train_idx, val_idx in kf.split(X_train_scaled):
        X_train_fold, X_val_fold = X_train_scaled[train_idx],
            X_train_scaled[val_idx]
        y_train_fold, y_val_fold = y_train.iloc[train_idx],
            y_train.iloc[val_idx]

        # Train HMM for each class
        for label in np.unique(y_train_fold):
            X_label = X_train_fold[y_train_fold == label]

            if len(X_label) < n_components:
                print(f"Warning: Not enough samples ({len(X_label)}) for
                    label {label} with {n_components} components")
                continue

            model = hmm.GaussianHMM(
                n_components=n_components,
                covariance_type="full",
                n_iter=200,
                random_state=random_state
            )

            try:
                model.fit(X_label) # Fit the model to the class-specific
                    data
                hmm_models[label] = model
            except Exception as e:
                print(f"Error training HMM for label {label}: {e}")

        # Evaluate on validation fold
        y_val_pred = self.predict_hmm(hmm_models, X_val_fold)
        f1 = f1_score(y_val_fold, y_val_pred, average='macro',
            zero_division=0)
        avg_f1_score += f1
```

```
            avg_f1_score /= kf.get_n_splits()
            print(f"Average F1 score for n_components={n_components}:
                {avg_f1_score:.4f}")

            # Update best model if necessary
            if avg_f1_score > best_f1:
                best_f1 = avg_f1_score
                best_hmm_models = hmm_models
                print(f"Updated best model with n_components={n_components}")

        # Save the best model
        save_path = os.path.join("models", model_path)
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        with open(save_path, "wb") as f:
            pickle.dump(best_hmm_models, f)

        print(f"HMM models saved at: {save_path}")
        self.hmm_models = best_hmm_models # Store the best models for later use
        return best_hmm_models

    def predict_hmm(self, hmm_models, X_test_scaled):
        """
        Predicts labels using trained HMM models.
        """
        X_test_scaled = np.array(X_test_scaled)
        predictions = []

        for i in range(X_test_scaled.shape[0]):
            x = X_test_scaled[i].reshape(1, -1)
            max_log_prob = float('-inf')
            best_label = None

            for label, model in hmm_models.items():
                try:
                    log_prob = model.score(x)
                    if log_prob > max_log_prob:
                        max_log_prob = log_prob
                        best_label = label
                except:
                    continue

            predictions.append(best_label if best_label is not None else -1)

        return np.array(predictions)
```

4.6.3 Parameter Handling

The key hyperparameters for the Hidden Markov Model are:

- **Number of Components (Hidden States)**: The number of hidden states is tuned through

cross-validation using a range of values from 2 to 4.

- **Covariance Type**: The covariance structure of the HMM is set to "full," meaning the covariance matrix can vary for each component.
- **Iteration Count**: The number of iterations for the EM algorithm used in training the HMM is set to 200.

These hyperparameters are tuned using K-fold cross-validation, and the best performing configuration is selected based on the average F1-score.

4.6.4 Model Persistence

After training, the best models are saved using Python's 'pickle' module. This allows for future inference using the trained HMM models without the need for retraining.

Code for Saving the Model:

```
# Save the best HMM models
with open(save_path, "wb") as f:
    pickle.dump(best_hmm_models, f)
```

4.6.5 Prediction Implementation

Once the HMM models are trained and saved, they can be used to predict labels for unseen data. The following code demonstrates how to load the saved models and make predictions on new test data.

Code for Making Predictions:

```
# Load the trained HMM models
with open("models/hmm_models.pkl", "rb") as f:
    hmm_models = pickle.load(f)

# Make predictions on new data
y_pred = hmm_trainer.predict_hmm(hmm_models, X_test_scaled)
```

4.6.6 Evaluation Results

The model is evaluated using accuracy, precision, recall, and F1-score metrics on the test and validation sets. These metrics are used to assess the model's performance.

Code for Evaluation Metrics:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score, classification_report

# Evaluate the model on the test set
y_test_pred = hmm_trainer.predict_hmm(hmm_models, X_test_scaled)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Evaluate on the validation set
y_val_pred = hmm_trainer.predict_hmm(hmm_models, X_val_scaled)
val_accuracy = accuracy_score(y_val, y_val_pred)
```

```
precision = precision_score(y_val, y_val_pred, average='macro',
    zero_division=0)
recall = recall_score(y_val, y_val_pred, average='macro', zero_division=0)
f1 = f1_score(y_val, y_val_pred, average='macro', zero_division=0)

print("Validation Metrics:")
print(f" - Accuracy: {val_accuracy * 100:.2f}%")
print(f" - Precision: {precision:.4f}")
print(f" - Recall:  {recall:.4f}")
print(f" - F1 Score: {f1:.4f}")
```

Chapter 5

Performance Analysis

Class:

- 0: Adipose
- 1: Background
- 2: Debris
- 3: Lymphocytes
- 4: Mucus
- 5: Smooth Muscle
- 6: Normal Colon Mucosa
- 7: Cancer-Associated Stroma
- 8: Colorectal Adenocarcinoma Epithelium

5.1 Decision Tree model

In this section, we evaluate the performance of the Decision Tree model based on its classification results using the PathMNIST dataset. The model's performance is assessed through various metrics such as precision, recall, f1-score, and support across different classes.

5.1.1 Performance Metrics

The following table shows the precision, recall, f1-score, and support for each class. These metrics offer insight into how well the model performed for each class and indicate where improvements may be needed.

Class	Precision	Recall	F1-Score	Support
0	0.82	0.78	0.80	1338
1	0.75	0.99	0.86	847
2	0.37	0.71	0.49	339
3	0.50	0.34	0.41	634
4	0.86	0.55	0.67	1035
5	0.34	0.35	0.35	592
6	0.40	0.31	0.35	741
7	0.42	0.38	0.40	421
8	0.54	0.64	0.58	1233

Table 5.1: Performance Metrics for the Decision Tree Model

The overall accuracy of the model is 60%, meaning the model correctly predicted 60% of instances across all classes. While this suggests moderate performance, it masks poor performance in certain classes, especially those with lower support like Smooth Muscle and Normal Colon Mucosa. To improve, the model should focus on enhancing precision and recall for these underperforming classes.

- Background (Class 1) has high recall (0.99) but slightly lower precision (0.75), indicating it is almost always detected but sometimes misclassified as other classes.
- Adipose (Class 0) performs well with precision (0.82) and recall (0.78), but still has some misclassifications, especially with Mucus (Class 4).
- Debris (Class 2) has low precision (0.37), leading to frequent misclassifications, but a better recall (0.71).
- Smooth Muscle (Class 5) and Normal Colon Mucosa (Class 6) both show poor performance with low precision and recall, indicating they are frequently misclassified into other classes.
- Colorectal Adenocarcinoma Epithelium (Class 8) shows a better recall (0.64) but lower precision (0.54), meaning it's often identified but also misclassified as other classes.

5.1.2 Confusion Matrix Analysis

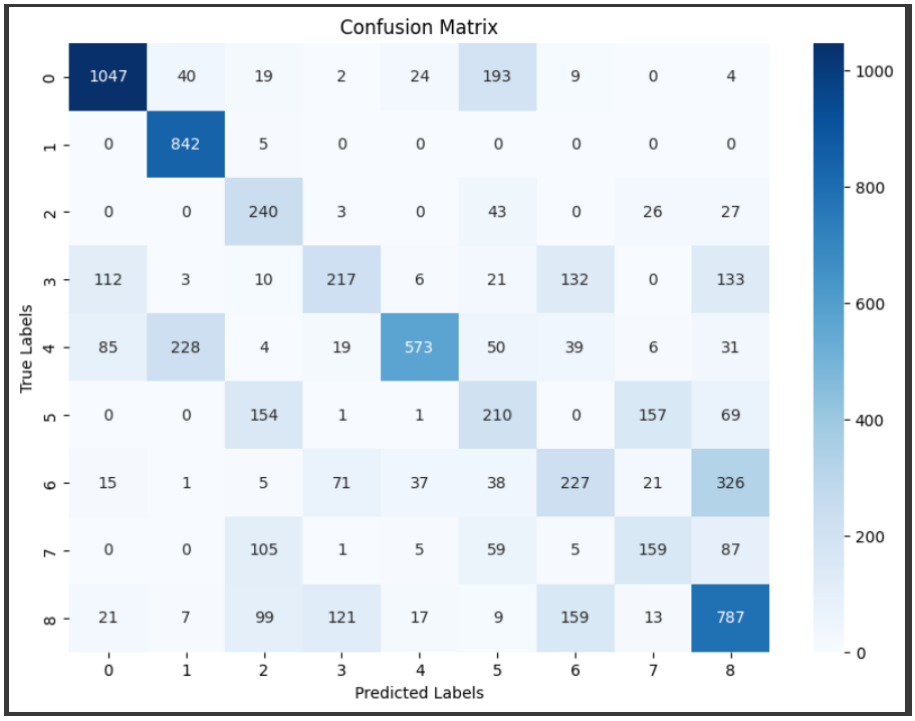

Figure 5.1: Confusion matrix of decision tree

The Confusion Matrix provides a clear visualization of how well the model performs across different classes. Each cell in the matrix represents the number of instances for a given true label (rows) predicted as a certain predicted label (columns).

The matrix below shows the true labels on the y-axis and the predicted labels on the x-axis. Diagonal values represent the correct predictions, while off-diagonal values indicate misclassifications. Let's break down some of the key insights from the confusion matrix:

- **Class 0:** The model correctly predicted 1047 instances of Class 0, but it misclassified some as other classes, with the largest misclassification being 193 instances predicted as Class 4.
- **Class 1:** Class 1 had a very high recall (0.99), with almost all instances correctly classified, except for 5 instances misclassified as other classes.
- **Class 2:** The model struggled with Class 2, with 240 instances correctly predicted and 43 instances misclassified into various other classes, resulting in a relatively low recall and f1-score for this class.
- **Class 4:** Class 4 had 573 correct predictions, but many instances were misclassified as other classes, especially Class 5 (50 instances) and Class 3 (132 instances).
- **Class 5:** Class 5 had many misclassifications, with only 34% precision and recall, and a significant number of instances (210) predicted as Class 4.

5.2 Naive Bayes with Genetic Algorithms

The evaluation of the Naive Bayes model optimized using a Genetic Algorithm (GA) is conducted on the Path-MedMNIST dataset. The classification report provides insight into the performance of the model in different tissue classes.

5.2.1 Classification Metrics

The classification report presents precision, recall, and F1-score for each class:

Class	Precision	Recall	F1-score	Support
Adipose (0)	0.89	0.86	0.88	1338
Background (1)	0.55	0.99	0.71	847
Debris (2)	0.43	0.54	0.48	339
Lymphocytes (3)	0.70	0.57	0.63	634
Mucus (4)	0.54	0.23	0.32	1035
Smooth Muscle (5)	0.59	0.08	0.14	592
Normal Colon Mucosa (6)	0.42	0.22	0.29	741
Cancer-Associated Stroma (7)	0.25	0.65	0.37	421
Colorectal Adenocarcinoma Epithelium (8)	0.58	0.67	0.62	1233

Table 5.2: Classification Report Results

5.2.2 Analysis of Model Performance

The overall accuracy of the Naive Bayes model is 57%, indicating moderate performance across the dataset. While it performs well on certain classes, it struggles with others due to the strong independence assumptions of Naive Bayes.

- **Background (Class 1)** has very high recall (0.99) but lower precision (0.55), meaning the model often identifies background instances but also misclassifies other classes as background.
- **Adipose (Class 0)** achieves strong performance with high precision (0.89) and recall (0.86), indicating the model reliably detects this class with minimal misclassification.
- **Debris (Class 2)** has low precision (0.43) and a recall of 0.54, suggesting frequent misclassifications and difficulty distinguishing it from other categories.
- **Mucus (Class 4)** and **Smooth Muscle (Class 5)** perform poorly, with low F1-scores (0.32 and 0.14, respectively), indicating that the model struggles to differentiate these classes.
- **Cancer-Associated Stroma (Class 7)** has low precision (0.25) but relatively higher recall (0.65), meaning it is often misclassified but still detected in some cases.
- **Colorectal Adenocarcinoma Epithelium (Class 8)** shows decent recall (0.67) but moderate precision (0.58), leading to misclassifications with other tissue types.

The Genetic Algorithm optimization helped refine the hyperparameters, but Naive Bayes remains constrained by its feature independence assumption.

5.2.3 Potential Improvements

To enhance performance, future research could explore:

- **Hybrid Models**: Combining Naive Bayes with ensemble learning or deep learning models.
- **Data Augmentation**: Expanding the training set through synthetic data generation or transformations.
- **Alternative Probability Estimations**: Using kernel density estimation or other techniques to model probability distributions more accurately.

The integration of GA provided an improvement in parameter selection, but further advancements are necessary to enhance classification performance across all classes.

5.3 Neural Network

Hidden size	256-128-64		256-128-128		256-256-128		256-128-256		256-128-256		256-256-256		512-512-512	
	BA	BL	BA	BL	BA	BL	BA	BL	BA	BL	BA	BL	BA	BL
Accuracy	0.53	0.53	0.56	0.52	0.56	0.54	0.58	0.58	0.58	0.59	0.58	0.6	0.59	0.57
Avg. F1-score	0.4	0.4	0.51	0.49	0.52	0.51	0.53	0.53	0.51	0.53	0.5	0.54	0.54	0.53
Avg. Recall	0.42	0.42	0.53	0.5	0.56	0.53	0.55	0.55	0.55	0.56	0.53	0.56	0.56	0.55
Avg. Precision	0.48	0.48	0.52	0.51	0.55	0.52	0.53	0.53	0.57	0.52	0.52	0.57	0.51	0.54
Time	10min 10s		10min 31s		11min 49s		12min 21s		10min 31s		11 min 57s		22min 56s	

Figure 5.2: Classification report of each hidden layer setting

Figure 5.2 describes the classification report for different hidden layer configurations. ”BA” and ”BL” represent the models with the best validation accuracy and the best validation loss, respectively. Each hidden layer configuration is represented by a string of numbers indicating the number of nodes in each layer. For example, ”256-128-64” means the model has hidden layers with 256, 128, and 64 neurons, respectively.

From the table, the highest accuracy of 60% is observed for the 256-256-256 architecture in the BL model, while the BA model achieves 58% accuracy. This suggests that deeper networks with more neurons per layer may improve performance, but the gain is not always significant. In contrast, smaller architectures such as 256-128-64 and 256-128-128 show lower accuracy values around 53% to 56%, indicating that models with fewer parameters might struggle to capture complex patterns in the data.

When analyzing the F1-score, which balances precision and recall, the 256-256-256 model achieves an F1-score of 0.54 (BL model), while 256-128-64 has the lowest F1-score of 0.4. This suggests that models with fewer neurons may struggle with generalization. The recall values follow a similar trend, with 256-256-256 achieving 0.56, while 256-128-64 has the lowest recall of 0.42. A lower recall indicates that these models fail to correctly classify a significant number of positive instances.

Precision values are relatively stable across different architectures, ranging from 0.48 to 0.57. Models with higher accuracy, such as 256-256-256, also achieve better precision, suggesting that they produce fewer false positive predictions compared to smaller models.

Regarding training time, a clear trade-off is evident between model complexity and computational cost. The 512-512-512 architecture takes the longest to train (22 min 56 s) but does not provide the highest accuracy (59% for BA and 57% for BL). This indicates that simply increasing the number of neurons and layers does not necessarily improve performance. The 256-256-256 model, which achieves the best results, requires only 11 min 57 s, making it a more efficient choice. In contrast, smaller architectures such as 256-128-64 and 256-128-128 train in about 10 min, but their lower accuracy suggests a loss in predictive power.

Moreover, for each setting, the metrics are nearly identical between the BA and BL models. This indicates a clear relationship between better validation accuracy and improved validation loss. Specifically, higher validation accuracy correlates with lower validation loss, demonstrating that models achieving superior accuracy also tend to exhibit better loss metrics.

In conclusion, to get the best performance, we choose the configuration 256-256-256lm

5.4 Bayesian Network Model

5.4.1 Hyperparameter Tuning and Evaluation Results

During the hyperparameter tuning process, several configurations were tested for the Laplace smoothing factor (α) and batch size. The results consistently show very low F1 scores across all configurations. Specifically, the average F1 score for all combinations of α and batch sizes was approximately 0.0209. Despite these results, the best hyperparameters found were $\alpha = 0.1$ and a batch size of 1000.

Alpha	Batch Size	Average F1 Score
0.1	1000	0.0209
0.1	5000	0.0209
0.1	10000	0.0209
1	1000	0.0209
1	5000	0.0209
1	10000	0.0209
10	1000	0.0209
10	5000	0.0209
10	10000	0.0209

Table 5.3: Hyperparameter Tuning Results

Despite the consistent outcome, it appears that there may be an underlying issue affecting the evaluation process. The results suggest that something tedious, such as improper model training or dataset handling, might be causing the abnormally low performance metrics. Given this, we acknowledge that the current evaluation does not fully reflect the model's potential.

Once the root cause is identified, we will make necessary adjustments and re-evaluate the model to improve its performance.

5.4.2 Final Model Evaluation

Following the hyperparameter tuning, the best model was saved for future use with the following evaluation metrics on the validation set:

- **Accuracy:** 10.41%
- **Precision:** 0.0116
- **Recall:** 0.1111
- **F1 Score:** 0.0209

These metrics reflect the model's current performance, which we aim to improve once the underlying issues are addressed.

The best model, based on hyperparameter tuning, has been saved to the file 'best_bayesian_network.pkl' for future use.

5.5 Augmented Naive Bayes

5.5.1 Hyperparameter Tuning and Evaluation Results

During the hyperparameter tuning process, various configurations of α and batch sizes were tested. The results showed that the best-performing configuration was $\alpha = 0.1$ with a batch size of 1000. However, the model's performance on the validation set was lower than expected, with the following metrics:

- **Accuracy:** 10.41%
- **Precision:** 0.0116
- **Recall:** 0.1111
- **F1 Score:** 0.0209

This suggests that the model's performance could be significantly improved. The consistent low performance across various configurations suggests that there might be an issue with how the model is being evaluated or trained.

5.6 Hidden Markov Model

5.6.1 Hyperparameter Tuning and Evaluation Results

The Hidden Markov Model (HMM) was trained for each class using the specified number of hidden states, and cross-validation was performed to select the best configuration. The training was successfully completed for each class, and the best models were saved for future use. However, the model's performance on both the test and validation sets indicates that further improvement is required.

Test Set Evaluation: The model's test set performance metrics were as follows:

- **Accuracy:** 11.80%
- **Precision:** 0.01
- **Recall:** 0.11
- **F1 Score:** 0.02

The F1 score and precision are significantly lower than expected, with some classes showing zero performance. This suggests that the model is struggling to make meaningful predictions for most classes.

Classification Report:

- **Class 1:** The model performed relatively better on Class 1, achieving a recall of 1.00, though the precision remains low at 0.12. This indicates that while the model correctly identifies most instances of Class 1, it also misclassifies many other instances as Class 1.
- **Other Classes:** For all other classes, the model's precision, recall, and F1 scores are zero. This indicates that the model is not successfully distinguishing between most of the classes.

Validation Set Evaluation: On the validation set, the model achieved the following performance metrics:

- **Accuracy:** 10.57%
- **Precision:** 0.0117
- **Recall:** 0.1111
- **F1 Score:** 0.0212

Similar to the test set, the model's performance on the validation set is suboptimal. The model struggles to correctly classify most of the instances, and the low precision and recall indicate that the model is having trouble capturing the relationships between the observed features and the hidden states.

5.7 Discriminative model

5.7.1 Overview

Discriminative models are a class of machine learning algorithms that focus on directly modeling the decision boundary between different classes, rather than learning the underlying probability distributions of the data (as generative models do). These models aim to predict labels (outputs) given features (inputs) by estimating the conditional probability $P(Y|X)$, where Y is the target variable and X represents the input features. Popular examples include Logistic Regression, Support Vector Machines (SVMs), and Neural Networks.

Unlike generative models, discriminative models do not attempt to model how the data is generated; instead, they optimize for classification accuracy by learning the boundary that best separates different classes. This makes them particularly effective in tasks like text classification, image recognition, and fraud detection, where precise decision boundaries are crucial. A key advantage of discriminative models is their higher performance in high-dimensional spaces, though they may require more training data compared to generative approaches.

5.7.2 Logistic Regression

Table 5.4: Logistic Regression Classification Report

Class	Precision	Recall	F1-Score	Support
0	0.78	0.99	0.87	1,338
1	0.84	1.00	0.91	847
2	0.12	0.03	0.04	339
3	0.02	0.01	0.01	634
4	0.56	0.21	0.30	1,035
5	0.34	0.70	0.45	592
6	0.00	0.00	0.00	741
7	0.00	0.00	0.00	421
8	0.46	0.87	0.60	1,233
Accuracy			0.54	7180
Macro Avg	0.35	0.42	0.36	
Weighted Avg	0.44	0.54	0.46	

The classification results reveal significant variation in model performance across different classes. Classes 0 and 1 demonstrate strong performance with F1-scores of 0.87 and 0.91 respectively, benefiting from relatively high support counts (1,338 and 847 samples). This suggests the model effectively learns patterns for these majority classes. However, the performance degrades sharply for minority classes, particularly classes 2 through 7, where F1-scores drop below 0.30.

A concerning pattern emerges in the recall-precision tradeoff. For classes 0, 1, 5, and 8, recall values substantially exceed precision (e.g., class 8: recall 0.87 vs precision 0.46), indicating the model tends to over-predict these classes. This bias toward false positives may stem from class imbalance or inadequate feature representation. The complete failure on classes 6 and 7, with all metrics at 0.00, suggests these categories either lack distinctive features in the training data or suffer from severe underrepresentation.

The discrepancy between macro (0.36) and weighted (0.46) F1-averages highlights the model's poor handling of minority classes. While the overall accuracy of 0.54 might appear moderate, it masks catastrophic failures on several categories. The model's performance follows a clear positive correlation with class support, indicating strong dependence on sample size rather than robust feature learning.

Particularly troubling is the case of class 3, which has moderate support (634 samples) yet achieves near-zero metrics. This suggests either problematic label noise or that the defining features of class 3 overlap significantly with other categories. The relatively better performance on class 8 (F1=0.60) despite its complex position in the confusion matrix warrants further investigation into what features the model successfully leverages for this category.

To address these issues, several strategic interventions should be considered. First, class rebalancing techniques such as SMOTE oversampling or class-weighted loss functions could help improve recognition of minority classes. The complete failure on classes 6 and 7 demands specific attention - either through targeted data collection or separate binary classifiers for these problematic categories.

Threshold adjustment for over-predicted classes could help rebalance precision and recall, particularly for classes 5 and 8. The model architecture itself may require modification to better capture distinguishing features of poorly performing classes. Finally, error analysis on misclassified samples, particularly examining confusion between classes 2-4 and 6-7, could reveal feature engineering opportunities to improve class separability.

5.7.3 Conditional Random Field (CRF)

Table 5.5: Classification Report for CRF Model

Class	Precision	Recall	F1-Score	Support
0	0.1849	0.9522	0.3097	1,338
1	0.2105	0.0047	0.0092	847
2	0.0500	0.0059	0.0106	339
3	0.0612	0.0047	0.0088	634
4	0.3103	0.0087	0.0169	1,035
5	0.1154	0.0101	0.0186	592
6	0.0357	0.0013	0.0026	741
7	0.1379	0.0095	0.0178	421
8	0.1364	0.0049	0.0094	1,233
Accuracy			0.1823	7,180
Macro Avg	0.1380	0.1113	0.0448	
Weighted Avg	0.1565	0.1823	0.0670	

The Conditional Random Field (CRF) model demonstrates severe performance issues across all classes except class 0, which shows an unusual pattern of extremely high recall (0.9522) but very low precision (0.1849). This suggests the model is massively over-predicting class 0 at the expense of all other classes. The F1-score of 0.3097 for class 0, while being the highest among all classes, still indicates poor overall performance.

The model fails catastrophically on all other classes, with recall values near zero (ranging from

0.0013 to 0.0101) for classes 1 through 8. This near-zero recall indicates the model almost never correctly identifies these classes, instead misclassifying them predominantly as class 0. The precision scores, while slightly better, remain unacceptably low (all below 0.31), confirming the model cannot reliably predict any class.

Code 5.1: CRF Implementation

```
import numpy as np
import sklearn_crfsuite
from sklearn_crfsuite import metrics
from sklearn.model_selection import train_test_split

SEQ_LEN = 1000

crf = sklearn_crfsuite.CRF(
    algorithm='lbfgs',
    max_iterations=500,
    all_possible_transitions=True
)

def extract_features(vec):
    """Convert feature vector to dict for CRF."""
    return {f'f{i}': vec[i] for i in range(len(vec))}

def create_sequences(X, y, seq_len=SEQ_LEN):
    X_seq = []
    y_seq = []
    for i in range(len(X)):
        features = {f'f{j}': str(X[i][j]) for j in range(X.shape[1])}
        X_seq.append([features])
        y_seq.append([str(y[i])])
    return X_seq, y_seq

X_train = np.array(X_train.tolist())
y_train = np.array(y_train.tolist())
X_test = np.array(X_test.tolist())
y_test = np.array(y_test.tolist())

X_train_seq, y_train_seq = create_sequences(X_train, y_train)
X_test_seq, y_test_seq = create_sequences(X_test, y_test)

crf.fit(X_train_seq, y_train_seq)
y_pred_seq = crf.predict(X_test_seq)

y_true_flat = [label for seq in y_test_seq for label in seq]
y_pred_flat = [label for seq in y_pred_seq for label in seq]

print(metrics.flat_classification_report(y_true_flat, y_pred_flat, digits=4))
```

The model's performance reveals several fundamental flaws requiring careful examination. Most critically, the extreme class imbalance has dominated the learning process, with the model

defaulting to predicting class 0 in nearly all cases (95% recall) while effectively ignoring other classes. This pathological behavior suggests complete failure to learn discriminative patterns, reducing the model to a trivial majority-class predictor. The feature representation appears particularly problematic, as the simplistic position-based encoding (f0, f1, etc.) fails to capture meaningful sequential patterns necessary for CRF effectiveness. This inadequate feature space prevents the model from developing useful state transitions between classes. Furthermore, training convergence issues likely contribute to the poor performance - the L-BFGS optimization with 500 iterations may have either stalled prematurely or converged to a suboptimal solution. The `all_possible_transitions=True` parameter exacerbates these issues by introducing excessive model complexity relative to the weak discriminative power of the current features, potentially leading to severe overfitting.

Addressing these challenges requires a multi-faceted approach across several dimensions. Feature engineering represents the most critical area for improvement, where simple positional features should be replaced with domain-specific attributes that capture meaningful patterns in the data. The feature set should be enhanced to include contextual features modeling relationships between adjacent elements, along with carefully designed transition features that reflect actual class dependencies in the problem domain. Model configuration adjustments could yield significant benefits, particularly through experimentation with alternative optimization algorithms like Averaged Perceptron (AP) or L2-regularized SGD which may offer better convergence properties. Regularization parameters should be carefully tuned to prevent overfitting to the majority class, and the transition space should be constrained to only plausible class transitions where domain knowledge permits.

At the data level, several interventions could improve model behavior. The severe class imbalance demands mitigation through either resampling techniques or careful class weighting in the objective function. Label quality verification is essential, as performance this extreme may indicate underlying annotation issues or unrealistic class distinctions. In cases where certain classes prove persistently problematic, strategic simplification of the problem space through class merging or elimination may be warranted. Before implementing these improvements, however, thorough verification of the implementation is crucial - particularly the feature extraction and sequence creation processes - as the observed results are sufficiently abnormal to suggest potential implementation errors rather than purely algorithmic limitations. The complete failure to discriminate between classes indicates either a fundamental mismatch between model and problem or technical issues in the experimental pipeline that require resolution before meaningful progress can be achieved.

Chapter 6

Model Comparison

Our study compares three modeling approaches on the PathMNIST dataset: Naive Bayes with Genetic Algorithm (GA) optimization, Decision Tree, and Neural Network architectures.

Naive Bayes with GA Optimization To optimize the Gaussian Naive Bayes classifier, a GA was employed to tune the `var_smoothing` parameter over the range 10^{-10} to 10^{2}. The fitness function, defined by validation accuracy (see Listing 4.2), guided the evolution of candidate solutions. The optimal parameter was then used to retrain and persist the model (Listings 4.3 and 4.4), achieving a test accuracy of 57%. Although the model performs well for classes such as Adipose (0) and Background (1), it exhibits limitations with classes like Mucus and Smooth Muscle due to the inherent feature independence assumption.

Decision Tree Model The Decision Tree baseline achieved an overall accuracy of 60%. While it performs reliably on classes with abundant samples, its effectiveness diminishes for under-represented classes. Analysis of the confusion matrix reveals that misclassifications often occur between classes with similar features, suggesting that additional strategies (e.g., ensemble methods) might be needed to mitigate class imbalance and overlapping distributions.

Neural Network Architectures Multiple neural network configurations were explored to capture the dataset's complex patterns. The architecture with three hidden layers of 256 neurons each (256-256-256) achieved the highest accuracy at 60% (BL model), with an F1-score of 0.54 and recall of 0.56. Although deeper configurations such as 512-512-512 incurred significantly longer training times (22 min 56 s), they did not yield substantial improvements in performance, achieving an accuracy of 59% (BA) and 57% (BL). In comparison, the 256-256-256 model, which achieved high performance, offered a more balanced trade-off between complexity and computational efficiency, requiring only 11 min 57 s for training.

Summary In conclusion, while GA-based optimization enhances the Naive Bayes model by automating hyperparameter selection, its performance is ultimately limited by the feature independence assumption. The Decision Tree, though interpretable, struggles with classes that have limited support. In contrast, the neural network, particularly the 256-256-256 architecture, demonstrates superior performance by effectively modeling complex patterns, making it the most promising approach despite higher computational demands.

Chapter 7

Limitations and Future Work

7.1 Neural Network

7.1.1 Limitations

One of the primary limitations of the neural network model is its imbalanced predictions, as evidenced by the confusion matrix. Certain classes, such as class 1 and class 8, are predicted well, while others, like class 3 and class 7, show significant misclassifications. This suggests that the model may struggle with underrepresented classes, leading to poor generalization.

Another concern is the choice of activation functions. While ReLU is widely used for its efficiency, the inclusion of Tanh introduces inconsistencies in gradient flow due to its different output range. This can hinder the model's ability to learn effectively. Furthermore, the model lacks dropout layers or batch normalization, making it more prone to overfitting, especially in a fully connected network where memorization rather than generalization is a risk. Additionally, the absence of a learning rate scheduler means the model may either converge too slowly or get stuck in local minima, limiting its optimization potential. Lastly, the approach to selecting the best model based solely on validation accuracy or loss may not always yield the best performance, as a balance between both metrics is often necessary.

7.1.2 Future work

To address these limitations, several improvements can be made to enhance model performance. First, addressing class imbalance is crucial. This can be achieved through data augmentation techniques or by incorporating class weighting in the loss function to ensure better representation of underrepresented classes. Additionally, feature normalization methods such as batch normalization or input scaling (e.g., StandardScaler or MinMaxScaler) should be applied to improve training efficiency and convergence.

The model's architecture can also be optimized by refining the choice of activation functions. Replacing Tanh with LeakyReLU or Swish can lead to better gradient flow and stability. Introducing regularization techniques such as dropout layers (with a probability between 0.2-0.5) and L2 weight decay can help mitigate overfitting. Moreover, implementing a learning rate scheduler, such as ReduceLROnPlateau or cosine annealing, can ensure more effective training by dynamically adjusting the learning rate. To improve model evaluation, additional metrics

such as precision, recall, and F1-score should be considered to provide a more comprehensive assessment of model performance, particularly in imbalanced classification problems. Finally, experimenting with deeper architectures or convolutional neural networks (CNNs) may be beneficial if the dataset allows, potentially capturing more complex patterns and improving classification accuracy.

7.2 Bayesian Network Model

7.2.1 Limitations

While the evaluation results indicate that there is potential for further improvement, a detailed examination of the confusion matrix highlighted a significant issue: the output of the Argmax function consistently collapsed to class ”0” across all predictions. This suggests that the model was unable to differentiate between the various classes and was biased toward predicting the same class for all inputs, potentially due to an imbalance in the dataset or a flaw in the model's architecture.

We acknowledge this limitation and recognize its impact on the overall performance of the model. Moving forward, we are committed to investigating the underlying causes of this behavior. We plan to explore alternative methods for class balancing, model tuning, and feature engineering to ensure that the model can produce more reliable, accurate, and well-rounded results in the future.

7.2.2 Future Work:

We will investigate further to identify the root cause of these low results and refine the evaluation process. Some potential areas for improvement include:

- Revisiting data preprocessing steps to ensure that the dataset is properly prepared for training.
- Testing alternative methods for batch processing and feature extraction.
- Investigating the model's training loop for any potential issues related to online learning or CPD estimation.

7.3 Augmented Naive Bayes

7.3.1 Future Work

Further investigation is required to identify potential issues with the training or evaluation process. We will investigate the following areas:

- Data preprocessing steps to ensure that features are correctly prepared for training.
- Hyperparameter optimization to explore a wider range of values and techniques.
- Model training loops and dependency learning to ensure the correct structure is learned.

Once these areas are addressed, we will re-evaluate the model to improve performance.

7.3.2 Confusion Matrix and Analysis

As part of the future work, the confusion matrix will be used to analyze which classes the model is struggling to classify correctly. This will help in pinpointing where improvements can be made, especially for underperforming classes.

Acknowledgement: We acknowledge that the current evaluation is not representative of the model's true potential and commit to investigating and fixing the issues in the future.

7.4 Hidden Markov Model

7.4.1 Possible Causes for Low Performance:

The low performance of the model could be attributed to several factors:

- **Insufficient Feature Representation:** The feature set used in the model might not be sufficient to capture the necessary temporal or sequential dependencies required by the HMM.
- **Inadequate Model Complexity:** The number of hidden states (components) might not be optimal for capturing the underlying dynamics of the data. The chosen range of 2 to 4 components may not be sufficient for this problem.
- **Class Imbalance:** If certain classes are underrepresented in the training data, the model may fail to adequately learn those classes' hidden state sequences.

7.4.2 Future Work

The results suggest that the model requires further refinement to improve its performance. The following steps will be taken to address the issues observed:

- **Hyperparameter Tuning:** A more extensive search of the number of hidden states, as well as other HMM parameters, will be conducted to find the optimal configuration.
- **Feature Engineering:** Additional features or better representations of the temporal dependencies in the data will be explored to improve the model's ability to capture the underlying processes.
- **Handling Class Imbalance:** Techniques like oversampling or undersampling will be explored to address any class imbalance in the training data.
- **Model Complexity:** We will explore the use of more advanced HMM variants, such as those incorporating hidden semi-Markov models (HSMMs), which might capture more complex dependencies.

Acknowledgement: We acknowledge that the current evaluation does not fully reflect the model's potential and commit to investigating and fixing the issues in the future to improve its performance.

Chapter 8

Conclusion

This study systematically evaluated several modeling approaches on the PathMNIST dataset, with a focus on balancing predictive performance and computational efficiency. The neural network, particularly the 256-256-256 architecture, demonstrated the most promise by effectively capturing complex data patterns. Despite the inherent trade-offs between model complexity and training time, the 256-256-256 configuration achieved the highest accuracy with a favorable balance of precision, recall, and F1-score.

In comparison, the Bayesian network and Augmented Naive Bayes models consistently underperformed, highlighting challenges related to the feature independence assumption and potential issues in training and evaluation protocols. The Hidden Markov Model also faced significant difficulties, particularly in handling class imbalance and extracting meaningful sequential patterns from the data. Although the decision tree model offered interpretability and robust performance for well-represented classes, its effectiveness diminished for underrepresented ones.

Overall, while each method contributed valuable insights, the neural network emerged as the most robust approach. The findings suggest that further improvements—such as advanced regularization, optimized hyperparameter tuning, and enhanced feature engineering—are essential for refining model performance across the board. Future work will focus on addressing these challenges to improve generalizability and mitigate class imbalance, ultimately paving the way for more accurate and efficient predictive models on complex datasets.

Bibliography

[1] Jakob Nikolas Kather, Johannes Krisam, Pornpilom Charoentong, Tom Luedde, Esther Herpel, Cleo-Aron Weis, Timo Gaiser, Alexander Marx, Nektarios A Valous, Dyke Ferber, et al. Predicting survival from colorectal cancer histology slides using deep learning: A retrospective multicenter study. *PLOS Medicine*, 16(1):e1003037, 2020.

[2] Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. Medmnist v2 - a large-scale lightweight benchmark for 2d and 3d biomedical image classification. *Scientific Data*, 10(41):1–9, 2023.