國立陽明交通大學

NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Introduction to Machine Learning

Final Project

Chan Helda 陳若鋆 112550812

Table of Contents

Introduction

This report delves into the intricate domain of healthcare analytics, specifically focusing on candidemia patients. The dataset comprises 77 features (F1-F77) and a binary class label "Outcome" (1 or 0), posing a binary class concept learning challenge.

My objective is to predict patient outcomes, employing a structured approach encompassing Exploratory Data Analysis, Data Preprocessing, and the evaluation of three classifiers—K-Nearest Neighbours (KNN), Multilayer Perceptron (MLP), and Naive Bayes. Each classifier's architecture, prediction results, and encountered challenges are discussed, culminating in a Comparative Assessment to unveil their relative performance.

Beyond the immediate task of predicting candidemia outcomes, this report signifies the commencement of my personal journey in machine learning, laying the groundwork for future exploration and contributions to the dynamic landscape of predictive modelling and healthcare analytics.

Exploratory Data Analysis (EDA)

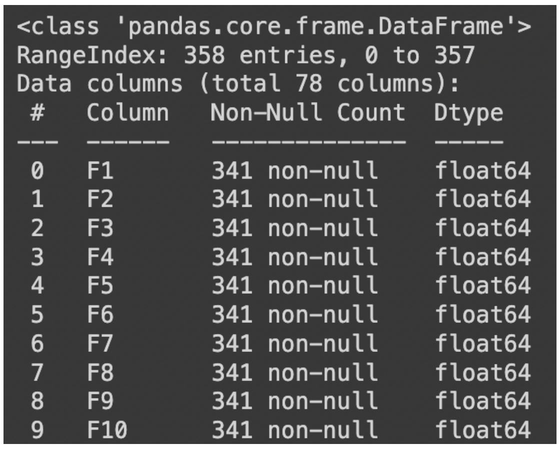
EDA serves as the compass guiding our understanding of the candidemia dataset. Through insightful visualisations and statistical summaries, we uncover patterns, identify outliers, and gain a comprehensive grasp of the dataset's nuances.

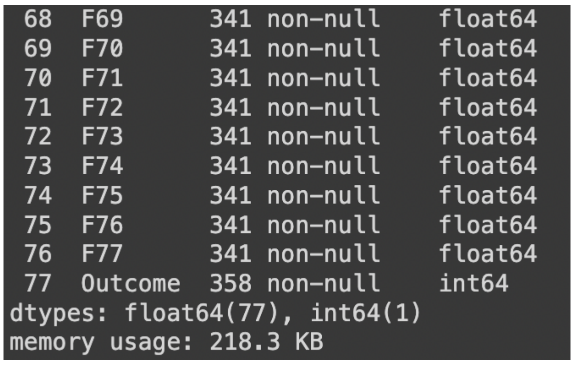
EDA not only sets the stage for subsequent modelling but also unveils the intrinsic characteristics of the data, enabling informed decisions in the pursuit of predicting patient outcomes.

Firstly, we view the summary of the dataset and we can see that there are no missing values.

```
df.info()
```

Output:

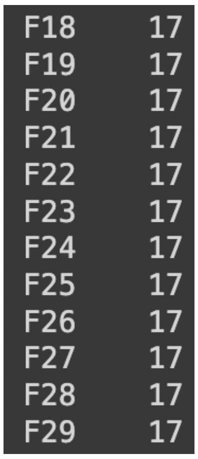

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 358 entries, 0 to 357
Data columns (total 78 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   F1       341 non-null    float64
 1   F2       341 non-null    float64
 2   F3       341 non-null    float64
 3   F4       341 non-null    float64
 4   F5       341 non-null    float64
 5   F6       341 non-null    float64
 6   F7       341 non-null    float64
 7   F8       341 non-null    float64
 8   F9       341 non-null    float64
 9   F10      341 non-null    float64
```
……


```
 68  F69      341 non-null    float64
 69  F70      341 non-null    float64
 70  F71      341 non-null    float64
 71  F72      341 non-null    float64
 72  F73      341 non-null    float64
 73  F74      341 non-null    float64
 74  F75      341 non-null    float64
 75  F76      341 non-null    float64
 76  F77      341 non-null    float64
 77  Outcome  358 non-null    int64
dtypes: float64(77), int64(1)
memory usage: 218.3 KB
```

Next, I explore the boolean_features (F18-F77). There seems to be 17 null values in each column.

```
df[boolean_features].isnull().sum()
```

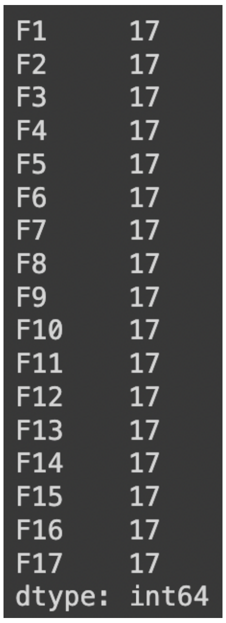
Output:


```
F18     17
F19     17
F20     17
F21     17
F22     17
F23     17
F24     17
F25     17
F26     17
F27     17
F28     17
F29     17
```
…….

Then, I explored the numerical features (F1-F17). There seems to also be 17 null values in each column.

```
df[numerical_features].isnull().sum()
```

Output:



```
F1     17
F2     17
F3     17
F4     17
F5     17
F6     17
F7     17
F8     17
F9     17
F10    17
F11    17
F12    17
F13    17
F14    17
F15    17
F16    17
F17    17
dtype: int64
```

Data Preprocessing

In the Data Preprocessing phase, meticulous steps are taken to ensure the dataset's integrity and enhance its suitability for subsequent analysis. For numerical features, any missing values are addressed by replacing them with the median, a robust measure resistant to outliers. Additionally, extreme values are clipped to the 1st and 99th percentiles, promoting data resilience against outliers.

The process extends to boolean features, where missing values are substituted with the mode. This approach ensures a balanced representation of boolean attributes, contributing to the overall robustness and reliability of the dataset. The combined efforts in numerical and boolean data preprocessing lay a solid foundation for the subsequent application of classifiers in predicting candidemia outcomes.

Architecture

1. **Initialization**

```
    def __init__(self, df):
        # Initialize the preprocessor with a DataFrame
        self.df = df
        self.numerical_features = ['F{}'.format(i) for i in range(1, 18)]
        self.boolean_features = ['F{}'.format(i) for i in range(18, 78)]
```

- The class is initialized with a DataFrame df.
- Lists of **numerical_features** and **boolean_features** are created based on the column names in the DataFrame.

2. Preprocessing

```
    def preprocess(self):
        # Apply various preprocessing methods on the DataFrame
        self.df = self._preprocess_numerical(self.df)
        self.df = self._preprocess_categorical(self.df)
        return self.df
```

- The preprocess method is the main entry point for preprocessing. It invokes two private methods, _preprocess_numerical and _preprocess_categorical.
- The result is a preprocessed DataFrame.

3. Numerical Preprocessing

```
def _preprocess_numerical(self, df):
       # Custom logic for preprocessing numerical features goes here

       # Impute missing values with median of each col
       median = df[self.numerical_features].median()
       df[self.numerical_features] =
df[self.numerical_features].fillna(median)

       for feature in self.numerical_features:
         # Clip extreme values to 1st and 99th percentiles
         percentiles = df[feature].quantile([0.01, 0.99]).values
         df[feature] = df[feature].clip(percentiles[0], percentiles[1])

         # Z-score scaling
         mean = df[feature].mean()
         std_dev = df[feature].std()
         if std_dev != 0:
             df[feature] = (df[feature] - mean) / std_dev
         else:
             # Set all values to a default value
```

```
              df[feature] = 0

        return df
```

- The **_preprocess_numerical** method handles preprocessing for numerical features.
- Missing values in numerical features are imputed with the **median** of each column.
    - I experimented with imputing with mean and median, median seems to give a better overall result (refer to appendix Table. 2)
- Extreme values are clipped to the **1st and 99th percentiles.**
- **Z-score scaling** is applied to standardise the numerical features.
    - I experimented with Z-score and Min-max scaling, Z-score scaling gave a general better result (refer to appendix Table. 1)

4. Categorical Preprocessing

```
    def _preprocess_categorical(self, df):
        # Add custom logic here for categorical features

        #impute Nan with mode of each col
        mode = df[self.boolean_features].mode()
        df[self.boolean_features] =
df[self.boolean_features].fillna(mode).iloc[0]
        return df
```

- The _preprocess_categorical method handles preprocessing for boolean features.
- Missing values in boolean features are imputed with the **mode** of each column.

Analysis

Classifiers

1. K-Nearest Neighbours (KNN)

K-Nearest Neighbors (KNN) is a simple and effective machine learning algorithm for classification and regression tasks. It predicts outcomes by considering the majority class or average of the K-nearest neighbors in the feature space.

KNN is versatile, requiring no model training, and is especially useful for problems with complex decision boundaries. Key considerations include choosing an optimal value for K and the appropriate distance metric.

a. Architecture

1. Initialization

```
class KNearestNeighbors(Classifier):
    def __init__(self, k=3):
        self.k = k
        self.X_train = None
        self.y_train = None
```

- The KNearestNeighbors class is created as a subclass of a generic Classifier.
- The constructor (__init__) initialises the KNN classifier with the number of neighbours k, which defaults to 3. It also sets placeholders for training data and labels.

2. Training(fitting) model

```
   def fit(self, X, y):
        self.X_train = X
        self.y_train = y
```

- The fit method stores the training data (X_train) and labels (y_train) for later use in predictions.

3. Prediction

```
    def predict(self, X):
        predictions = []
        for pt in X.values:
            #calculates euclidean distance
            distances = [np.sqrt(np.sum((x - pt)**2)) for x in
self.X_train.values]
            sorted_indices = np.argsort(distances)[:self.k]
            predicted_outcome = self.y_train.loc[sorted_indices]
            prediction = max(set(predicted_outcome),
key=predicted_outcome.tolist().count)
            predictions.append(prediction)
        return np.array(predictions)
```

- The predict method performs predictions for a given set of input data (X) based on the K-nearest neighbors.
- For each test point, it calculates Euclidean distances to all training points, identifies the K-nearest neighbors, and predicts the outcome based on majority voting.

4. **Probability Estimation**

```
 def predict_proba(self, X):
        probabilities = []
        for test_point in X.values:
            distances = [np.sqrt(np.sum((train_point - test_point)**2))
for train_point in self.X_train.values]
            sorted_indices = np.argsort(distances)[:self.k]
            predicted_outcome = self.y_train[sorted_indices]

            # Calculate probabilities
            class_counts = {}
            total_neighbors = len(predicted_outcome)

            for cls in predicted_outcome:
                class_counts[cls] = class_counts.get(cls, 0) + 1

            target_probabilities = {cls: count / total_neighbors for
cls, count in class_counts.items()}

            # Ensure all classes have a probability entry, even if it's
zero
            all_classes = set(self.y_train)
            for cls in all_classes:
                if cls not in target_probabilities:
                    target_probabilities[cls] = 0.0
            probabilities.append(list(target_probabilities.values()))

        return np.array(probabilities)
```

- The predict_proba method estimates class probabilities for each test point. It calculates the proportion of each class among the K-nearest neighbors.

b. Prediction results

In this section, I analysed the performance of my KNN with the different metrics.

model	accuracy	f1	precision	recall	mcc	auc
KNN Average	0.72087302	0.41655929	0.5943651	0.3701765	0.2830742	0.60731479

1. Accuracy (0.7209):

- The accuracy of 72.09% indicates the proportion of correctly classified instances out of the total.

2. F1 Score (0.4166):

- The F1 Score, which considers both precision and recall, is relatively lower. This suggests that the model may face challenges in achieving a balance between minimising false positives and false negatives.

3. Precision (0.5944):

- Precision, measuring the accuracy of positive predictions, is around 59.44%. This implies that when the model predicts a positive outcome, it is correct approximately 59.44% of the time.

4. Recall (0.3702):

- The recall, or sensitivity, is approximately 37.02%.
- This indicates that the model captures only about 37.02% of the actual positive instances, suggesting room for improvement in identifying all positive cases.

5. Matthews Correlation Coefficient (MCC) (0.2831):

- The Matthews Correlation Coefficient is a balanced metric considering true positives, true negatives, false positives, and false negatives.
- The MCC of 0.2831 suggests a moderate level of agreement between predictions and actual outcomes.

6. Area Under the Curve (AUC) (0.6073):

- The AUC value of 0.6073 indicates the model's ability to distinguish between positive and negative instances.

- A value above 0.5 suggests better-than-random discrimination, but there is still room for improvement.

c. Result analysis and improvement

The KNN model exhibits decent performance, achieving notable accuracy but showing areas for improvement in terms of precision, recall, and overall F1 Score. The MCC suggests a moderate level of agreement between predictions and true outcomes, with areas of improvement.

F1 score
The F1 score is sensitive to the trade-off between precision and recall. KNN might face challenges in achieving a balanced F1 score due to the choice of k.

Maybe exploring different k values and adjusting the decision boundary. As well as optimising k to achieve a balance between precision and recall, possibly through grid search and cross-validation.

Precision
Precision in KNN can be influenced by the density of data points in the feature space.

I might need to evaluate the impact of different distance metrics and feature scaling on precision. As well as experiment with distance-weighted voting to give more influence to closer neighbours.

Recall
Recall is affected by the sensitivity to capturing positive instances. The choice of k and the density of points can impact recall in KNN.

I need to adjust the value of k and experiment with other techniques like oversampling the minority class to address imbalances.

MCC
MCC considers the balance between true positives, true negatives, false positives, and false negatives. In KNN, achieving a balance is crucial.

I will have to optimise k for better precision and recall balance. Investigate the impact of different distance metrics on MCC. Fine-tune the model based on MCC optimization.

d. Challenges

KNN was surprisingly easier than the other classifier. However i faced many shape mismatch issues with ***evaluate_mode***l function in ***roc_auc_score*** as shown:

```
ValueError: y should be a 1d array, got an array of shape (36, 2) instead.
```

Hence, i modified the code as such:

1. **Binary Classification Handling:**
    - Checks if there are only two unique classes in ***y_test*** (binary classification).
    - If the probability array (proba) has multiple columns (indicating it's handling multiple classes), it converts ***y_test*** to binary format using the positive class label.
2. **Multiclass Classification Handling:**
    - Checks if there are more than two unique classes in ***y_test*** (multiclass classification).
    - Converts ***y_test*** to binary format using the positive class label if necessary.

```
if len(np.unique(y_test)) == 2:  # Binary classification
    # Ensure y_test is binary
    if proba.ndim > 1 and proba.shape[1] > 1:
        # Convert y_test to binary format, specifying the positive class
label
        if len(np.unique(y_test)) > 2:
            y_test = (y_test == positive_class_label).astype(int)  #
Specify the positive class label
        auc = roc_auc_score(y_test, proba[:, 1])
    else:
        auc = roc_auc_score(y_test, proba)
else:  # Multiclass classification
    # Ensure y_test is not multilabel
    if len(np.unique(y_test)) > 2:
        y_test = (y_test == positive_class_label).astype(int)  # Specify
the positive class label
    auc = roc_auc_score(y_test, proba, multi_class='ovo')
```

By handling the binary classification scenario separately and ensuring that ***y_test*** is appropriately formatted, the improved code aims to prevent the ***ValueError*** related to the shape mismatch in ***roc_auc_score.***

2. Multilayer Perceptron (MLP)

a. Architecture

1. Initialization

- The ***__init__*** method initialises the MLP with default parameters such as the structure of hidden layers (hidden_layers_sizes), learning rate (learning_rate), and the number of epochs (epochs).
    - **Hidden_layers_sizes = [77, 10, 1]**
        - Input layer = 77 neuron (represents 77 features)
        - Hidden layer = 10 neuron
            - I've experimented with 8, 10, 12, 14 neurons and 10 neurons gave the best overall results (refer to Appendix. Table 3)
        - Output layer = 1 neuron (binary classification)
    - **Learning rate = 0.001**
        - I've experimented with Lr = 0.01, 0.001, 0.0001 and 0.001 gave the best overall results (refer to Appendix. Table 4)
    - **epochs = 100** is default values
- The ***variables*** dictionary is used to store weights, biases, and intermediate values during training.
- ***X*** and ***y*** are set to **None** initially.

```
    def __init__(self, hidden_layers_sizes = [77, 10, 1], learning_rate = 
0.001, epochs = 100):
        # Initialize MLP with given network structure
        self.hidden_layers_sizes = hidden_layers_sizes
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.variables = {'w0': None, 'b0': None, 'w1': None, 'b1': None, 
'o1': None, 'o2': None, 'a1': None}
        self.X = None
        self.y = None
        # self.input_size = None
        pass
```

2. Weight initialization

- The ***initialise_weights*** method initializes weights using Xavier/Glorot initialization for each layer
- Xavier/Glorot initialization is a technique used to initialize the weights of the neurons in a neural network. The goal of this initialization method is to address issues related to vanishing or exploding gradients during training.
- The initialization is performed using the following formula:

$$\text{variance} = \frac{2}{\text{number of input neurons} + \text{number of output neurons}}$$

- After calculating the variance, the weights are then sampled from a Gaussian distribution with mean 0 and the computed variance.

```
    def initialise_weights(self):
        np.random.seed(1)
        input_size = self.X.shape[1]

        # Xavier/Glorot initialization for the first hidden layer
        self.variables["w0"] = np.random.randn(self.hidden_layers_sizes[0],
self.hidden_layers_sizes[1]) * np.sqrt(2 / (input_size +
self.hidden_layers_sizes[0]))
        self.variables["b0"] = np.random.randn(self.hidden_layers_sizes[1],)

        # Xavier/Glorot initialization for the second hidden layer
        self.variables["w1"] = np.random.randn(self.hidden_layers_sizes[1],
self.hidden_layers_sizes[2]) * np.sqrt(2 / (self.hidden_layers_sizes[0] +
self.hidden_layers_sizes[1]))
        self.variables["b1"] = np.random.randn(self.hidden_layers_sizes[2],)
```

3. Training (fitting)

- The ***fit*** method trains the MLP using forward and backward propagation for a specified number of epochs.
- It calls initialise_weights to set initial weights.
- It iteratively performs forward and backward propagation to update weights and biases.

```
    def fit(self, X, y):
        # Implement training logic for MLP including forward and backward
propagation
        self.X = X
        self.y = y

        #initialise weights
        self.initialise_weights()

        #perform forward and backward propagation
        for i in range(self.epochs):
            self._backward_propagation(self._forward_propagation(X))

        pass
```

4. Prediction

- The ***predict*** method predicts binary outcomes for the given input X using the trained MLP.
- The ***predict_proba*** method provides the probability estimates for binary outcomes.

```
    def predict(self, X):
        # Implement prediction logic for MLP

        a1 = self.relu_activation(X.dot(self.variables["w0"]) +
self.variables["b0"])
        prob_pred = self._sigmoid_func(a1.dot(self.variables["w1"]) +
self.variables["b1"])
        binary_output = [int(np.round(item[0])) for item in prob_pred.values]
        return binary_output


    def predict_proba(self, X):
        # Implement probability estimation for MLP

        a1 = self.relu_activation(X.dot(self.variables["w0"]) +
self.variables["b0"])
        prob_output = self._sigmoid_func(a1.dot(self.variables["w1"]) +
self.variables["b1"])
        return prob_output
```

5. Forward Propagation

- The ***_forward_propagation*** method computes the forward pass of the neural network, applying ReLU activation for hidden layers and sigmoid activation for the output layer.
- Stores weighted sum and activation in ***variables*** dictionary for use in backward propagation.

```
   def _forward_propagation(self,X):
       # Implement forward propagation for MLP

        o1 = self.variables["b0"] + self.X @ (self.variables["w0"]) #calc
weighted sum for first hidden layer
        a1 = self.relu_activation(o1) #apply ReLU activation function
        o2 = self.variables["b1"] + a1 @ (self.variables["w1"]) #weighted sum
for output layer
        output = self._sigmoid_func(o2) #apply sigmoid activation function

        #store for use in backpropagation
        self.variables["o1"] = o1
        self.variables["a1"] = a1
        self.variables["o2"] = o2

        return output
```

6. Backward Propagation

- The ***_backward_propagation*** method computes the backward pass, updating weights and biases based on the calculated gradients using gradient descent.

  Compute gradient with respect to outer layer:
  - ***sigmoid_derivative:*** Derivative of the sigmoid activation function applied to the output layer.
  - ***loss_gradient_output***: Gradient of the loss function with respect to the predicted output. It combines the effects of the loss function and the derivative of the sigmoid activation.
  - ***output_layer_gradient:*** Gradient of the loss with respect to the output layer.

  Compute gradient with respect to first hidden layer:
  - ***hidden_layer1_gradient***: Gradient of the loss with respect to the activations of the first hidden layer.
  - ***weight2_gradient:*** Gradient of the loss with respect to the weights of the output layer.
  - ***bias2_gradient:*** Gradient of the loss with respect to the biases of the output layer.
  - Update the weights and biases for the output layer using gradient descent as shown:

$$\theta = \theta - \alpha \cdot \frac{\partial J}{\partial \theta}$$

    - ***J*** : loss function.
    - ***α*** : learning rate, a hyperparameter that determines the step size in the update.

  Compute gradient with respect to input layer:
  - Similar to first hidden layer

```
    def _backward_propagation(self, predicted_output):
        # Implement backward propagation for MLP

        #transformation for calculation of gradient
        true_labels = self.y.to_numpy().reshape(predicted_output.shape)

        # Compute gradients with respect to the output layer
        sigmoid_derivative = predicted_output * (1 - predicted_output)
         # np.maximum :  prevent division by zero and ensures that
logarithmic functions have valid input.
```

```
        loss_gradient_output = np.divide((1 - true_labels),np.maximum(1 -
predicted_output, 1e-10)) - np.divide(true_labels,
np.maximum(predicted_output,1e-10))
        output_layer_gradient = sigmoid_derivative * loss_gradient_output

        # Compute gradients with respect to the first hidden layer
        hidden_layer1_gradient = output_layer_gradient @
self.variables["w1"].T
        weight2_gradient = self.variables["a1"].T @ output_layer_gradient
        bias2_gradient = np.sum(output_layer_gradient, axis=0)

        #update weights and biases
        self.variables['w1'] -= weight2_gradient * self.learning_rate
        self.variables['b1'] -= bias2_gradient * self.learning_rate

        # Compute gradients with respect to the input layer
        hidden_layer1_derivative = self.relu_derivative(self.variables['o1'])
* hidden_layer1_gradient
        weight1_gradient = self.X.T @ (hidden_layer1_derivative)
        bias1_gradient = np.sum(hidden_layer1_derivative, axis=0)

        # Update the weights and biases
        self.variables['w0'] -= weight1_gradient * self.learning_rate
        self.variables['b0'] -= bias1_gradient * self.learning_rate

        pass
```

7. Activation Functions

- ***relu_activation*** implements the Rectified Linear Unit (ReLU) activation function:

$$f(x) = \max(0, x)$$

(***x*** : input to activation function)

- ***relu_derivative*** computes the derivative of the ReLU function:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

(***x*** : input to activation function)

- ***_sigmoid_func*** computes the sigmoid activation function:

$$f(x) = \frac{1}{1+e^{-x}}$$

(x : input to activation function)

```
def relu_activation(self, weighted_sum):
    return np.maximum(0, weighted_sum)

def relu_derivative(self, x):
    return np.where(x > 0, 1, 0)

def _sigmoid_func(self, x):
    return 1/(np.exp(-x)+1)
```

.

b. Prediction results

model	accuracy	f1	precision	recall	mcc	auc
MLP Average	0.796428571	0.598519967	0.70462038	0.5454962	0.48436317	0.83424066

1. **Accuracy (0.7964)**
   - My MLP model achieves a high accuracy of 79.64%, indicating the proportion of correctly classified instances out of the total.
   - This suggests that the model performs well in terms of overall classification accuracy.
2. **Precision (0.7046)**
   - The precision, measuring the accuracy of positive predictions, is around 70.46%.
   - This suggests that when the model predicts a positive outcome, it is correct approximately 70.46% of the time.

3. **Recall (0.5455):**
   - The recall, or sensitivity, is approximately 54.55%.
   - This indicates that the model captures about 54.55% of the actual positive instances, suggesting room for improvement in identifying all positive cases.
4. **Matthews Correlation Coefficient (MCC) (0.4844):**
   - The Matthews Correlation Coefficient is a balanced metric considering true positives, true negatives, false positives, and false negatives.
   - The MCC of 0.4844 suggests a moderate level of agreement between predictions and actual outcomes.
5. **Area Under the Curve (AUC) (0.8342):**
   - The AUC value of 0.8342 indicates the model's ability to distinguish between positive and negative instances.
   - A value above 0.5 suggests better-than-random discrimination, and the achieved AUC is relatively high.

c. Result analysis and improvement

My MLP model exhibited commendable accuracy (79.64%) and strong precision (70.46%). However, challenges in recall (54.55%) and the Matthews Correlation Coefficient (MCC) at 0.4844 point to opportunities for improvement.

Despite these nuances, the model's exceptional discriminatory ability, reflected in an Area Under the Curve (AUC) of 0.8342, underscores its potential in real-world applications. Ongoing refinement is crucial to address specific challenges and ensure the model's reliability in the complex landscape of candidemia patient data.

F1 Score
Imbalances in precision and recall could occur, and achieving a higher F1 score may require a delicate balance between these two metrics.

To improve, i could fine-tune the model to achieve a more balanced precision-recall trade-off. Explore advanced techniques like ensemble models or algorithm-specific optimizations.

Recall
The challenge lies in identifying a higher proportion of positive instances, especially when the class distribution is imbalanced.

To improve, I can address class imbalance through techniques like oversampling or adjusting class weights. As well as, experiment with different algorithms that handle class imbalance effectively.

MCC
MCC can be sensitive to imbalanced datasets and may require additional measures to handle class distribution.

To improve, I might need to explore data preprocessing techniques to address imbalances. Additionally, experiment with ensemble methods or fine-tune model parameters to optimise MCC.

d. Challenges

Developing the Multilayer Perceptron (MLP) came with significant challenges, primarily revolving around the multitude of matrix multiplications inherent in the neural network's structure.

The intricacies of managing multiple hidden layers and interconnected weights necessitated careful consideration of dimensionality. Incorrect dimension multiplication emerged as a prominent issue, demanding thorough debugging and meticulous validation of matrix dimensions throughout the forward and backward propagation processes.

Successfully overcoming these challenges involved implementing robust error-checking mechanisms and conducting rigorous testing to identify and rectify any issues related to incorrect matrix multiplications. The precision and alignment of matrices and vectors at every stage of computation became pivotal in ensuring the accurate functioning of the MLP implementation.

3. Naive Bayes

In this section, my code defines a Naive Bayes classifier, specifically designed to handle datasets with a mix of binary and numeric features, with the use of bernoulli and gaussian naive bayes classifiers, respectively.

a. Architecture

1. Initialization

- The constructor (__init__) initialises the Naive Bayes classifier with Gaussian and Bernoulli feature lists, a Laplace smoothing parameter (alpha), and dictionaries to store parameters for Gaussian and Bernoulli features, as well as class probabilities.

```
def __init__(self):
    self.numerical_features = ['F{}'.format(i) for i in range(1, 18)]
    self.binary_features = ['F{}'.format(i) for i in range(18, 78)]
    self.alpha = 1  # Laplace smoothing parameter
    self.gaussian_params = {}  # Parameters for Gaussian features
    self.bernoulli_params = {}  # Parameters for Bernoulli features
    self.class_probs = {}  # Class probabilities
```

2. Training model

- The fit method calculates class probabilities using laplace smoothing and fits Gaussian and Bernoulli models on the respective features.
- Gaussian model fitting
    - For each numerical feature, calculate the mean and standard deviation for instances belonging to each class.
    - Store the mean and standard deviation parameters in the “gaussian_params” dictionary.

- Bernoulli model fitting
    - For each binary feature, calculate the probability of the feature being true for instances belonging to each class.

- Store the probabilities in the “bernoulli_params” dictionary.

```
  def fit(self, X, y):
      # Calculate class probabilities
      num_samples = len(y)
      num_classes = len(set(y))

      for i in set(y):
          filter = (y == i)
          self.class_probs[i] = (sum(filter) + self.alpha) / (num_samples +
num_classes * self.alpha) #laplace smoothing to avoid 0 probabilities

          # Fit Gaussian model on Gaussian features
          for feature in self.numerical_features:
              mean = X.loc[filter, feature].mean()
              std = X.loc[filter, feature].std()
              self.gaussian_params[(i, feature)] = (mean, std)

          # Fit Bernoulli model on Bernoulli features
          for feature in self.binary_features:
              prob_true = (X.loc[filter, feature].sum() + self.alpha) /
(sum(filter) + 2 * self.alpha) #laplace smoothing
              self.bernoulli_params[(i, feature)] = prob_true
```

3. Prediction

- The predict method calculates the log probabilities for each class based on Gaussian and Bernoulli features.
- The class with the highest overall log probability is predicted for each instance.

```
   def predict(self, X):
      predictions = []

      for nil, instance in X.iterrows():
          class_probs = {}

          #initialise log prob
          for p in self.class_probs:
              gaussian_log_prob = 0
              bernoulli_log_prob = 0

              # Calculate Gaussian log probabilities
              for feature in self.numerical_features:
                  mean, std = self.gaussian_params[(p, feature)]
                  gaussian_log_prob +=
self.gaussian_log_probability(instance[feature], mean, std)

               # Calculate Bernoulli log probabilities
               for feature in self.binary_features:
                   prob_true = self.bernoulli_params[(p, feature)]
                   bernoulli_log_prob +=
self.bernoulli_log_probability(instance[feature], prob_true)

               # Calculate overall log probability for the class
               class_probs[p] = np.log(self.class_probs[p]) +
gaussian_log_prob + bernoulli_log_prob

           # Predict the class with the highest probability
           prediction = max(class_probs, key=class_probs.get)
           predictions.append(prediction)

       return predictions
```

4. Probability Estimation

- The predict_proba method calculates class probabilities similar to the predict method but normalizes them to obtain a probability distribution across classes.

```
def predict_proba(self, X):
       probabilities = []

       for nil, instance in X.iterrows():
           class_probs = {}

           #initialise log prob
           for i in self.class_probs:
               gaussian_log_prob = 0
               bernoulli_log_prob = 0

               # Calculate Gaussian log probabilities
               for feature in self.numerical_features:
                   mean, std = self.gaussian_params[(i, feature)]
                   gaussian_log_prob +=
self.gaussian_log_probability(instance[feature], mean, std)

               # Calculate Bernoulli log probabilities
               for feature in self.binary_features:
                   prob_true = self.bernoulli_params[(i, feature)]
                   bernoulli_log_prob +=
self.bernoulli_log_probability(instance[feature], prob_true)

               # Calculate overall log probability for the class
               class_probs[i] = np.log(self.class_probs[i]) +
gaussian_log_prob + bernoulli_log_prob

           # Normalize the probabilities
           exp_probs = np.exp(list(class_probs.values()))
           probabilities.append(exp_probs / np.sum(exp_probs))

       return np.array(probabilities)
```

5. Gaussian and Bernoulli Log Probability Calculations

```
    def gaussian_log_probability(self, x, mean, std):
        exponent = -((x - mean) ** 2) / (2 * (std ** 2))
        return exponent + -0.5 * np.log(2 * np.pi * (std ** 2))

    def bernoulli_log_probability(self, x, prob_true):
        return (1 - x) * np.log(1 - prob_true) + x *
np.log(prob_true)
```

- Two separate methods, gaussian_log_probability and bernoulli_log_probability, compute the log probabilities for Gaussian and Bernoulli features, respectively.
- The formula for the **Gaussian log probability** is derived from the probability density function (PDF) of the normal distribution. It is calculated as follows:

$$\text{Gaussian Log Probability} = -\frac{1}{2}\ln(2\pi\sigma^2) - \frac{(x-\mu)^2}{2\sigma^2}$$

  - **σ** : standard deviation
  - **μ** : mean
  - ***x*** : observed value

- The formula for the **Bernoulli log probability** is derived from the probability mass function (PMF) of the Bernoulli distribution. It is calculated as follows:

$$\text{Bernoulli Log Probability} = x\ln(p) + (1-x)\ln(1-p)$$

  - ***p*** : probability of the feature being true (having a value of 1)
  - ***x*** : observed value for the feature.

b. Prediction results

model	accuracy	f1	precision	recall	mcc	auc
Naive Bayes Average	0.760079365	0.577350168	0.59659091	0.58096737	0.41986751	0.780321642

1. **Accuracy (0.760)**
    - Accuracy is the proportion of correctly classified instances among the total instances.
    - My Naive Bayes model achieves an accuracy of approximately 76.01%. This indicates that 76.01% of the instances in the dataset were correctly predicted by the model.
2. **F1 Score (0.577)**
    - A higher F1 score indicates a better balance between precision and recall.
    - In my case, the F1 score is approximately 0.577, indicating a reasonable balance between precision and recall.
3. **Precision (0.596)**
    - Precision is the proportion of true positive predictions among all positive predictions. A higher precision value indicates fewer false positives.
    - The Naive Bayes model achieves a precision of approximately 59.66%.
4. **Recall (0.581)**
    - Recall, also known as sensitivity or true positive rate, is the proportion of actual positives that were correctly identified by the model. A higher recall value indicates fewer false negatives.
    - The Naive Bayes model achieves a recall of approximately 58.10%.
5. **Matthews Correlation Coefficient (MCC) (0.419)**
    - MCC takes into account true and false positives and negatives, providing a measure of the overall agreement between predictions and observations.
    - The Naive Bayes model achieves an MCC of approximately 0.419, indicating a moderate level of agreement.
6. **Area Under the Curve (AUC) (0.780)**
    - A higher AUC value (closer to 1) indicates better discrimination between positive and negative instances.
    - My Naive Bayes model achieves an AUC of approximately 0.780, suggesting a good ability to distinguish between positive and negative instances.

c. Result analysis and Improvements

My Naive Bayes model demonstrates reasonable performance across multiple metrics, with notable accuracy, a balanced F1 score, and good discrimination capability as indicated by the AUC.

However, my precision, recall and MCC has room for improvement.

Precision

A low precision can be an issue if minimising false positives is critical for the task. Adjusting the model's threshold, incorporating additional features, or fine-tuning the model might enhance precision. Handling outliers or noise in the data could also improve precision.

Recall
Low recall could be problematic if it's essential to minimise false negatives (missing positive instances).Adjusting the model's threshold or exploring techniques to handle class imbalance might improve recall. Additionally, feature engineering or model tuning could address specific challenges affecting recall.

MCC
Achieving a higher MCC value is generally desirable, especially for imbalanced datasets. Model tuning, optimising for MCC, or exploring different evaluation metrics might improve the model's performance. Addressing imbalances in the dataset could also enhance MCC.

d. Challenges

Developing a Naive Bayes classifier for a mixed dataset, comprising both numerical and binary features, presented several challenges. Initially, the application of the Bernoulli Naive Bayes algorithm to the entire dataset was explored by discretizing numerical features into bins.

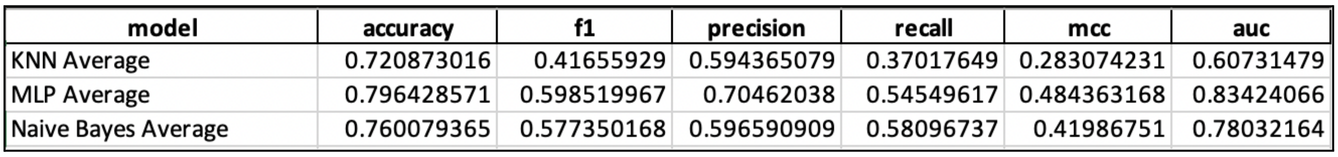
However, this approach encountered limitations, particularly regarding the loss of information during the discretization process. Challenges were further exacerbated by the sensitivity of the model's performance to the choice of bin width and the number of bins.

The attempt to utilize Bernoulli Naive Bayes on the mixed dataset resulted in suboptimal accuracy and predictive power. The algorithm struggled to effectively capture the intricate complexities and dependencies inherent in numerical features.

These challenges prompted a shift in strategy towards a dual-model approach, where the dataset was split into binary and numerical features, allowing for more tailored and effective modeling strategies for each feature type.

Comparative Assessment

Results Assessment



model	accuracy	f1	precision	recall	mcc	auc
KNN Average	0.720873016	0.41655929	0.594365079	0.37017649	0.283074231	0.60731479
MLP Average	0.796428571	0.598519967	0.70462038	0.54549617	0.484363168	0.83424066
Naive Bayes Average	0.760079365	0.577350168	0.596590909	0.58096737	0.41986751	0.78032164

1. **Accuracy**
   - MLP demonstrates the highest accuracy at 79.64%, outperforming KNN (72.09%) and Naive Bayes (76.01%).
2. **F1 Score**
   - MLP achieves the highest F1 score (59.85%), indicating a higher balance between precision and recall.
   - Naive Bayes follows closely with an F1 score of 57.74%, while KNN lags behind at 41.66%.
3. **Precision**
   - MLP leads in precision (70.46%), showcasing its ability to make accurate positive predictions.
   - Naive Bayes follows with precision at 59.66%, and KNN trails at 59.44%.
4. **Recall**
   - MLP exhibits the highest recall (54.55%), capturing a significant proportion of actual positive instances.
   - Naive Bayes follows with recall at 58.10%, while KNN has the lowest at 37.02%.
5. **MCC**
   - MLP achieves the highest MCC at 0.4844, indicating a moderate level of agreement.
   - Naive Bayes follows with MCC at 0.4199, and KNN has the lowest at 0.2831.
6. **AUC**
   - MLP excels in AUC (83.42%), showcasing superior discriminatory ability.
   - Naive Bayes follows with an AUC of 78.03%, while KNN lags behind at 60.73%.

Model Assessment

MLP

- The Multilayer Perceptron (MLP) stands out as the top-performing model due to its versatility and effectiveness.
- It excels in various metrics, including accuracy, precision, recall, and AUC. This suggests that MLP effectively captures complex patterns and relationships within the candidemia patient data.
- The model's neural network architecture allows it to learn intricate representations, contributing to its superior performance across a range of evaluation criteria.

Naive Bayes

- Naive Bayes demonstrates competitive performance, particularly in AUC (Area Under the Curve) and F1 score.
- These strengths highlight Naive Bayes' ability to provide reliable predictions and make it a viable alternative to more complex models like MLP.
- However, there is room for improvement, as discussed in the comprehensive assessment.

KNN

- K-Nearest Neighbors (KNN) showcases robustness in certain aspects.
- However, challenges arise in precision, recall, and overall accuracy.
- The challenges in these areas suggest that KNN may struggle with balancing false positives and false negatives, impacting its overall predictive accuracy.

Conclusion

In conclusion, this study provided valuable insights into predicting candidemia outcomes using machine learning models. MLP showcased its superiority, Naive Bayes demonstrated competitive performance, and KNN revealed potential areas for enhancement.

The diverse challenges encountered during this project, from handling mixed datasets to addressing the intricacies of different algorithms, have significantly contributed to my understanding of machine learning concepts and their practical implementation.

As I reflect on the outcomes and challenges faced, this project motivates me to delve deeper into advanced machine learning techniques, explore larger datasets, and contribute to the ongoing advancements in the field. It marks a personal milestone in my journey toward mastering machine learning for meaningful applications.

Appendix

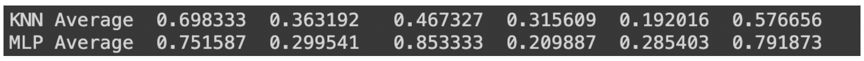
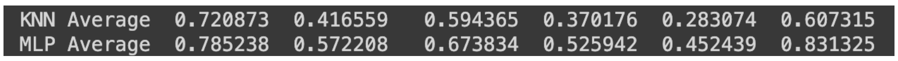
Type of Scaling	Metric (accuracy, f1, precision, recall, mcc, auc)
Z-score scaling	 `KNN Average  0.698333  0.363192  0.467327  0.315609  0.192016  0.576656` `MLP Average  0.751587  0.299541  0.853333  0.209887  0.285403  0.791873`
Min max scaling	 `KNN Average  0.720873  0.416559  0.594365  0.370176  0.283074  0.607315` `MLP Average  0.785238  0.572208  0.673834  0.525942  0.452439  0.831325`

Table 1: Preprocessing - Different types of scaling

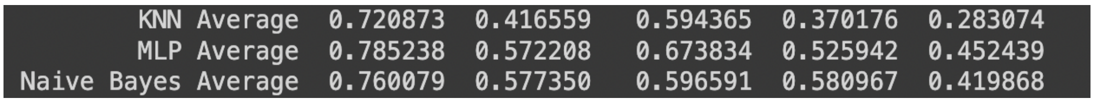
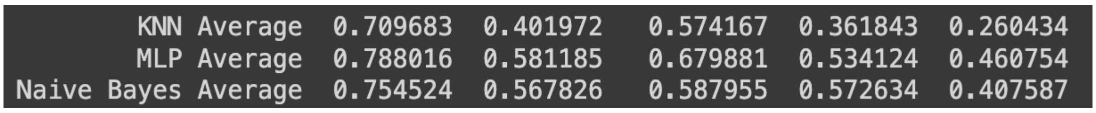
Replace Nan with:	**Metric (accuracy, f1, precision, recall, mcc, auc)**
Median	 KNN Average 0.720873 0.416559 0.594365 0.370176 0.283074 MLP Average 0.785238 0.572208 0.673834 0.525942 0.452439 Naive Bayes Average 0.760079 0.577350 0.596591 0.580967 0.419868
Mean	 KNN Average 0.709683 0.401972 0.574167 0.361843 0.260434 MLP Average 0.788016 0.581185 0.679881 0.534124 0.460754 Naive Bayes Average 0.754524 0.567826 0.587955 0.572634 0.407587

Table 2: Preprocessing - Replace NAN in numerical feature

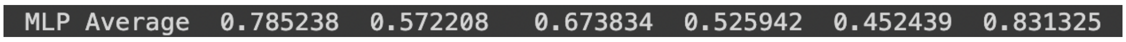
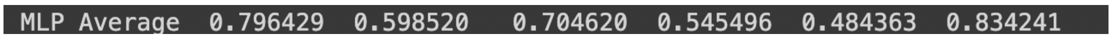
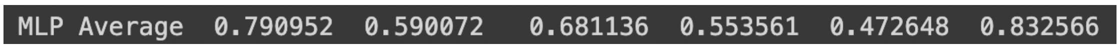
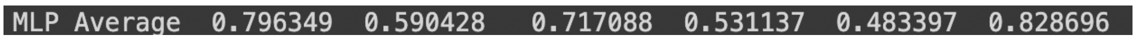
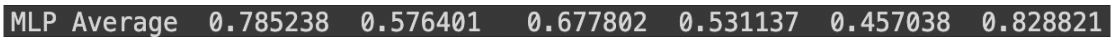
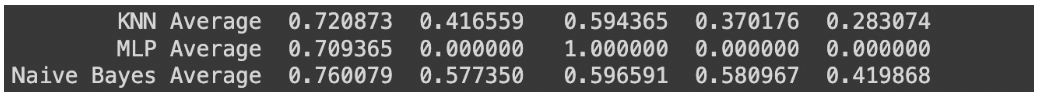
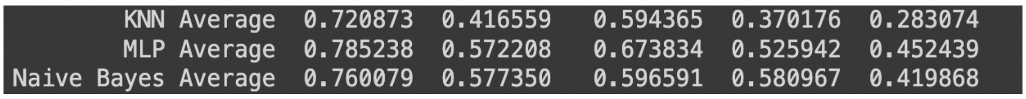
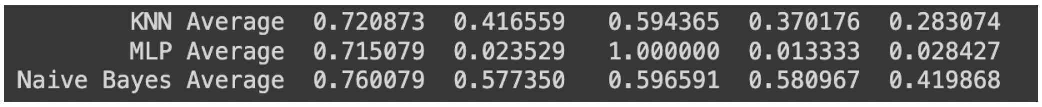
Num. inner layer neurons	Metric results (accuracy, f1, precision, recall, mcc, auc)
8	 MLP Average 0.785238 0.572208 0.673834 0.525942 0.452439 0.831325
10	 MLP Average 0.796429 0.598520 0.704620 0.545496 0.484363 0.834241
11	 MLP Average 0.790952 0.590072 0.681136 0.553561 0.472648 0.832566
12	 MLP Average 0.796349 0.590428 0.717088 0.531137 0.483397 0.828696
14	 MLP Average 0.785238 0.576401 0.677802 0.531137 0.457038 0.828821

Table 3: MLP - Different num. Of inner layer neurons

Learning Rate (LR)	**Metric results (accuracy, f1, precision, recall, mcc, auc)**
LR = 0.01	 KNN Average 0.720873 0.416559 0.594365 0.370176 0.283074 MLP Average 0.709365 0.000000 1.000000 0.000000 0.000000 Naive Bayes Average 0.760079 0.577350 0.596591 0.580967 0.419868
LR=0.001	 KNN Average 0.720873 0.416559 0.594365 0.370176 0.283074 MLP Average 0.785238 0.572208 0.673834 0.525942 0.452439 Naive Bayes Average 0.760079 0.577350 0.596591 0.580967 0.419868

LR=0.0001	

```
        KNN Average  0.720873  0.416559   0.594365  0.370176  0.283074
        MLP Average  0.715079  0.023529   1.000000  0.013333  0.028427
Naive Bayes Average  0.760079  0.577350   0.596591  0.580967  0.419868
```

Table 4: MLP - different learning rate