INTRODUCTION TO MACHINE LEARNING

Term Project

Ng Wan Xing

112511808

Table of Contents

Introduction

In this project, we leverage machine learning techniques to develop classifiers capable of accurately identifying patients with candidemia based on a comprehensive set of features. Our analysis includes three distinct classifiers which are k-Nearest Neighbour (kNN), Naïve Bayes and a Multilayer Perceptron (MLP). Each classifier brings unique strengths and approaches to the task, from the simplicity and interpretability of kNN to the probabilistic nature of Naïve Bayes and the complex non-linear relationships captured by MLP.

Different pre-processing techniques are employed to ensure the robustness of the models such as handling outliers, null values and standardization of features. Additionally, the implementation details of each classifier, including hyperparameters, network architecture, and loss functions, are carefully outlined.

Moreover, we will discuss the specifics of the pre-processing steps, implementation details of the classifiers, challenges encountered during model construction, and a comprehensive discussion of model comparisons and results in the subsequence section. The aim is to provide a detailed understanding of the methodologies employed, insights gained, and the overall effectiveness of each classifier in predicting candidemia in real-world patients.

Dataset Description

The dataset we used in the project are real-world data of patients diagnosed with candidemia. Each patient is described by 77 features (F1 – F77) and a binary target variable (“Outcome”), indicating whether the patient has candidemia (1) or not (0). The first 17 features are numerical features and the rest is Boolean features. Different pre-processing techniques would be conducted to each type of the features.

Data Exploration

Data exploration is for exploring the data with general details such as non-null count and the type for each column. In this case, we could have better understanding on the dataset to implement specific data pre-processing techniques.

The code for data exploration is as below:

```
# Load trainWithLable data
df = pd.read_csv('trainWithLabel.csv')

di = df.info()
print(di)
```

Figure 1: Code for Showing Data Information

```
<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
 10  F11      341 non-null    float64
 11  F12      341 non-null    float64
 12  F13      341 non-null    float64
 13  F14      341 non-null    float64
 62  F63      341 non-null    float64
 63  F64      341 non-null    float64
 64  F65      341 non-null    float64
 65  F66      341 non-null    float64
 66  F67      341 non-null    float64
 67  F68      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
None
```

Figure 2: Result of Data Information

The result is showing that the dataset has 358 rows and 78 columns. Each column has 341 non-null rows, the type of each column is float except for the last column. The last column has no null values and is integer.

```
num_feature = df.iloc[:,0:17].isnull().sum()
boo_feature = df.iloc[:,18:77].isnull().sum()

print(num_feature)
print(boo_feature)
```

Figure 3: Code for Checking Null Values

```
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
```

Figure 4: Result of the Count of Null Values in Numerical Features

```
F19    17    F39    17    F60    17
F20    17    F40    17    F61    17
F21    17    F41    17    F62    17
F22    17    F42    17    F63    17
F23    17    F43    17    F64    17
F24    17    F44    17    F65    17
F25    17    F45    17    F66    17
F26    17    F46    17    F67    17
F27    17    F47    17    F68    17
F28    17    F48    17    F69    17
F29    17    F49    17    F70    17
F30    17    F50    17    F71    17
F31    17    F51    17    F72    17
F32    17    F52    17    F73    17
F33    17    F53    17    F74    17
F34    17    F54    17    F75    17
F35    17    F55    17    F76    17
F36    17    F56    17    F77    17
F37    17    F57    17    dtype: int64
F38    17    F58    17
```

Figure 5: Result of the Count of Null Values in Boolean Features

For checking the amount of null values in each column, we separated the columns into numerical features and Boolean features. From the result, we could know that the each column has 17 null values. Hence, we would need to handle the null values in later section of data pre-processing.

Data Pre-processing

```
def manual_train_val_split(data, test_size=0.2, random_state=None):
    # function to split the data
    np.random.seed(random_state)
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_size)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]

# Main function to execute the pipeline
def main():
    # Load trainWithLable data
    df = pd.read_csv('trainWithLabel.csv')

    # split the data into training and validation sets
    X_train, X_val = manual_train_val_split(df.drop('Outcome', axis=1), test_size=0.2, random_state=42)
    y_train, y_val = df['Outcome'].iloc[X_train.index], df['Outcome'].iloc[X_val.index]
```

Figure 6: Code for Splitting Data

Before data pre-processing, we split the data into training dataset and validation dataset. We only applied data pre-processing to the training dataset and then used the parameters obtained from the training dataset to transform the validation dataset.

Outliers are detected and capped at the 1st and 99th percentiles for the whole dataset.

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

# Create a copy of the DataFrame to avoid SettingWithCopy issues
    df = df.copy()

# Detect and replace the outlier
    def cap_data(df):
        for col in df.columns:
            if (((df[col].dtype) == 'float64') | ((df[col].dtype) == 'int64')):
                percentiles = df[col].quantile([0.01, 0.99]).values
                df[col][df[col] <= percentiles[0]] = round(percentiles[0], 6)
                df[col][df[col] >= percentiles[1]] = round(percentiles[1], 6)
            else:
                df[col] = pd.to_numeric(df[col], errors='coerce')

    cap_data(df)
```

Figure 7: Code for Handling the Outliers

The cap_data function is aiming at handling outliers in numeric columns by capping extreme values within a specific range defined by percentiles.
For the numeric columns, it calculates the 1st and 99th percentiles of the column using df[col].quantile([0.01, 0.99]). Outliers below the 1st percentile are replaced with the rounded 1st percentile value, while outliers above the 99th percentile value are replaced with the rounded 99th percentile value.

For the non-numeric columns, it uses pd.to_numeric with errors='coerce' to convert the column to numeric values. Any non-numeric values are coerced to NaN (null values).

```
# Only for the numerical features
# Standardization
# Replace the null values with mean
    for feature in df.columns[0:18]:
        df[feature].fillna(df[feature].mean(), inplace=True)
        df[feature] = (df[feature] - df[feature].mean()) / df[feature].std()

# Only for the Binary features
# Replace the null values with mode of column [18:77] for binary features
    for x in df.columns[18:77]:
        df[x].fillna(df[x].mode().iloc[0], inplace=True)
```

Figure 8: Code for Handling the Null Values and Standardization

Numerical Features:

We first replaced the null values in numerical features with the mean and then standardized.

Boolean Features:

Null values is replaced with the mode from column “F18”.

Classifiers Architectures

Naïve Bayes

Naive Bayes is a simple yet powerful classification algorithm based on Bayes' Theorem, which is a probabilistic approach for making decisions.

Bayes’ Theorem:

$$P(A \mid B) = \frac{P(B \mid A).P(A)}{P(B)}$$

In the context of Naive Bayes classification:

P(A | B) is the probability of class A given the features B.
P(B | A) is the likelihood of observing features B given class A.

P(A) is the prior probability of class A.
P(B) is the prior probability of observing features B.

In this Naïve Bayes classifiers, we utilises Gaussian Naïve Bayes for numerical features. Priors, means, and variances for numerical features and probabilities for binary features are calculated during the fitting process. Laplace smoothing is used in fitting process for handling the zero probabilities. Lastly, Log likelihoods are computed during prediction, and the class with the highest likelihood is chosen.

Gaussian Naïve Bayes:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Gaussian Naive Bayes is a classification method in machine learning that employs a probabilistic approach, assuming a normal distribution for each class. This technique operates on the assumption of independence among the parameters, allowing it to predict the probability of the output variable belonging to each class.

```
class NaiveBayesClassifier():
    def __init__(self):
        # Initialize the classifier
        self.classes = None
        self.prior = None
        self.num_means = None
        self.num_variances = None
        self.bin_probabilities = None

    def fit(self, X, y, alpha=1):
        # Separate numerical and binary features
        X_num = X.iloc[:, :17]
        X_bin = X.iloc[:, 17:]

        # get each unique classes and calculate the priors
        self.classes = np.unique(y)
        self.prior = {classes: (np.sum(y == classes) + alpha) / (len(y) +
                                len(self.classes) * alpha) for classes in self.classes}

        # Calculate means and variances for numerical features with Laplace smoothing
        self.num_means = (X_num.groupby(y).sum() + alpha) / (X_num.groupby(y).count() + alpha * 17)
        self.num_variances = (X_num.groupby(y).apply(lambda x: ((x - x.mean()) ** 2).sum()) + alpha) /
        (X_num.groupby(y).count() + alpha * 17)

        # Calculate probabilities for binary features with Laplace smoothing
        self.bin_probabilities = (X_bin.groupby(y).sum() + alpha) / (X_bin.groupby(y).count() +
                                  alpha * (X_bin.shape[1]))
```

Figure 9: Code for Naïve Bayes Classification

From the figure above, we first initializes various attributes that will be used to store information learned during training.

Parameters	**Explanation**
Classes	Classes is for counting each unique classes in target in target variable.
prior	Prior refers to the prior probability for each class.
Num_means	Num_means is the mean for numerical features for Gaussian Naïve Bayes
Num_variances	Num_variances is the variance for numerical features for Gaussian Naïve Bayes
Bin_probabilities	Bin_probabilities is the probability for binary features.

For the fitting process, we separated the dataset in to numerical features and binary features for implementing different data techniques on them respectively. We calculated means and variances for numerical features using Laplace smoothing to handle zero probabilities. Also, we calculated probabilities for binary features with Laplace smoothing.

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

    for na, instance in X.iterrows():
        class_likelihoods = []

        for classes in self.classes:
            #get the logarithm of the prior probability for classes
            prior = np.log(self.prior[classes])
            #Gaussian probability density function for the numerical features [1:17]
            num_likelihood = np.sum(
                -0.5 * np.log(2 * np.pi * self.num_variances.loc[classes])
                - 0.5 * ((instance[:17] - self.num_means.loc[classes]) ** 2 / self.num_variances.loc[classes])
            )
            #Log probability for the binary features [17:77]
            bin_likelihood = np.sum(
                instance[17:] * np.log(self.bin_probabilities.loc[classes]) +
                (1 - instance[17:]) * np.log(1 - self.bin_probabilities.loc[classes])
            )

            total_likelihood = prior + num_likelihood + bin_likelihood
            class_likelihoods.append(total_likelihood)

        #take the max class_likelihoods
        predicted_class = self.classes[np.argmax(class_likelihoods)]
        predictions.append(predicted_class)

    return predictions
```

Figure 10: Code for Naïve Bayes Classification

This predict function is used to make predictions based on the trained model to provide the most likely class labels. It iterates through instances in the input data (X), calculating likelihoods for each class using Gaussian assumptions for numerical features and log probabilities for binary features. The total_likelihoods represents the overall likelihood of the observed features for a given instance belonging to a particular class. This total likelihood is then appended to the class_likelihoods list for the current class. After iterating through all classes and calculating their likelihoods for the current instance, the predicted class is determined by selecting the class with the highest total likelihood. The predicted class is then added to the predictions list. Finally, the method returns the list of predicted class labels for all instances in the input data (X).

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

    for na, instance in X.iterrows():
        class_likelihoods = []

        for classes in self.classes:
            prior = np.log(self.prior[classes])
            num_likelihood = np.sum(
                -0.5 * np.log(2 * np.pi * self.num_variances.loc[classes])
                - 0.5 * ((instance[:17] - self.num_means.loc[classes]) ** 2 / self.num_variances.loc[classes])
            )
            bin_likelihood = np.sum(
                instance[17:] * np.log(self.bin_probabilities.loc[classes]) +
                (1 - instance[17:]) * np.log(1 - self.bin_probabilities.loc[classes])
            )

            total_likelihood = prior + num_likelihood + bin_likelihood
            class_likelihoods.append(total_likelihood)

        #softmax function: ensuring sum up to 1 for each instance
        exp_likelihoods = np.exp(class_likelihoods - np.max(class_likelihoods))
        class_probabilities = exp_likelihoods / np.sum(exp_likelihoods)
        probabilities.append(class_probabilities)

    return np.array(probabilities)
```

Figure 11: Code for Naïve Bayes Classification

The predict_proba function has similar code with predit function but the outcome is different. The predict_proba function returns the class probabilities for the given input instances. Another different is that we had applied softmax function to the class_likelihoods. This is to ensure that the class probabilities sum up to 1 for each instance. It exponentiates the class likelihoods and normalizes them.

K Nearest Neighbour (KNN)

The K-Nearest Neighbors (KNN) algorithm is widely employed in machine learning for tasks involving classification and regression. It classifies the label based on the assumption that data points with close distance share similar labels.

```python
# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return (distance)**(1/2)

# K-Nearest Neighbors Classifier
class KNearestNeighbors:
    def __init__(self, k=7):
        # Initialize KNN with k neighbors
        self.k = k

    def fit(self, X, y):
        # Store training data and labels for KNN
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        # Implement the prediction logic for KNN
        predictions = [self._predict(x) for x in X.values]
        return np.array(predictions)
```

Figure 12: Code for K Nearest Neighbour (KNN)

As KNN utilize the distance between the input data point and all the training data points to make prediction. Thus, we first defined a distance metric to be used in this KNN model. The chosen distance metric is Euclidean distance. It measures the distance between two points.

Later, we initialise the k neighbors for determining how many nearest neighbors should take into account for making the prediction. In this project, 7 is determined as the k neighbors as 7 gave better performance.

Parameters	**Explanation**
K	Stands for the k neighbors, it used to determine how many neighbors should be included to make the prediction.

During the fitting process, we simply stored the training data and labels for KNN. For the predict function, it stores the prediction result made from the _predict function into list of predictions.

```python
def _predict(self, x):
    # Helper method for prediction
    distances = [euclidean_distance(x, x_train) for x_train in self.X_train.values]
    #Find the indices of the k samples with the smallest distances.
    k_idx = np.argsort(distances)[:self.k]
    #locate the class labels with the indices with smallest distance
    k_neighbor_labels = self.y_train.iloc[k_idx]

    #count each unique class label in k neighbours
    unique_labels, counts = np.unique(k_neighbor_labels, return_counts=True)
    #return the highest amount of unqiue class label
    most_label = unique_labels[np.argmax(counts)]

    return most_label
```

Figure 13: Code for K Nearest Neighbour (KNN)

The _predict function is helping to find the nearest k neighbors for a given input instance, counts the occurrences of each unique class label, and returns the class label with the highest count. This label is considered the predicted label for the input instance.

```python
def predict_proba(self, X):
    # Implement probability estimation for KNN
    probabilities = []
    for x in X.values:
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train.values]
        k_idx = np.argsort(distances)[:self.k]
        k_neighbor_labels = self.y_train.iloc[k_idx]

        # count for each class label
        class_counts = {}
        for label in k_neighbor_labels:
            class_counts[label] = class_counts.get(label, 0) + 1

        # Convert counts to probabilities
        class_probs = [class_counts.get(c, 0) for c in sorted(np.unique(self.y_train))]
        probabilities.append(class_probs)

    return np.array(probabilities) / self.k
```

Figure 14: Code for K Nearest Neighbour (KNN)

The predict_proba method calculates the probabilities of each class for each input instance in X. It finds the k-nearest neighbors for each instance, counts the occurrences of each class label, and converts the counts to probabilities. The probabilities are then normalized by dividing by self.k.

Multilayer Perceptron Classifier (MLP)

The Multilayer Perceptron (MLP) is a neural network algorithm designed to grasp the connections within both linear and non-linear datasets. It has input layer, output layer and hidden layer. In this MLP model, we only implemented 1 hidden layer.

```
# Multilayer Perceptron Classifier
class MultilayerPerceptron():
    def __init__(self, hidden_layers_sizes = 10):
        # Initialize MLP with given network structure
        self.para = {}
        self.hidden_layers_sizes = hidden_layers_sizes
        self.loss = []
        self.X = None
        self.y = None
        self.input_size = None
        self.output_size = None

        pass

    def init_weights(self):

        np.random.seed(1)

        self.para["W1"] = np.random.randn(self.input_size, self.hidden_layers_sizes)
        self.para["b1"] = np.random.randn(self.hidden_layers_sizes,)
        self.para["W2"] = np.random.randn(self.hidden_layers_sizes, self.output_size)
        self.para["b2"] = np.random.randn(self.output_size,)

    # prevent division by zero
    def eps(self, x):
        epsilon = 0.00000000001
        return np.clip(x, epsilon, None)

    def sigmoid(self, Z):
        sig = 1/(1+np.exp(-Z))
        return sig

    def relu(self, Z):
        return np.maximum(0,Z)

    def dRelu(self, x):
        x[x <= 0] = 0
        x[x > 0] = 1
        return x
```

Figure 15: Code for Multilayer Perceptron Classifier (MLP)

From the figure above, we first initializes various attributes that will be used to store information learned during training.

Parameters	**Explanation**
para	Stands for an empty dictionary of parameters to store the weights and bias
Hidden_layer_sizes	The size of hidden layer (number of neuron in the hidden layer)
loss	Num_means is the mean for numerical features for Gaussian Naïve Bayes
Input_size	The size of the input layer (number of input feature)
Output_size	The size of the output layer (number of target feature)
X	The features of training data
Y	The outcome of training data

For the function of init_weights, this method initializes the weights and biases of the neural network with random values using NumPy's randn function. It sets up weights W1 and W2 for the hidden and output layers, and biases b1 and b2 for the hidden and output layers, respectively.

The eps function is for avoiding the division by zero. If the input is zero, it uses 0.00000000001 instead of zero for the input.

The sigmoid method computes the sigmoid activation function for a given input Z.

The relu method implements the rectified linear unit (ReLU) activation function.

The dRelu method computes the derivative of the ReLU activation function.

```
def entropy_loss(self, y, output):

    #inverse of output
    output_inv = 1.0 - output
    #reshape y to have the same shape as predicted output
    y = y.to_numpy().reshape(output.shape)
    y_inv = 1.0 - y
    output = self.eps(output)
    output_inv = self.eps(output_inv)

    loss = -1/len(y) * np.sum(y * np.log(self.eps(output)) + (1 - y) * np.log(self.eps(1 - output_inv)))

    return loss

def fit(self, X, y, learning_rate = 0.001, epochs = 150):
    # Implement training logic for MLP including forward and backward propagation

    self.X = X
    self.y = y
    self.input_size = X.shape[1]
    self.output_size = 1
    self.learning_rate = learning_rate
    self.epochs = epochs
    self.init_weights()

    for i in range(self.epochs):
        output, loss = self._forward_propagation(X)

        gra_w1, gra_w2, gra_b1, gra_b2 = self._backward_propagation(output)

        #update the weights and bias
        self.para['W1'] = self.para['W1'] - self.learning_rate * gra_w1
        self.para['W2'] = self.para['W2'] - self.learning_rate * gra_w2
        self.para['b1'] = self.para['b1'] - self.learning_rate * gra_b1
        self.para['b2'] = self.para['b2'] - self.learning_rate * gra_b2

        self.loss.append(loss)

    pass
```

Figure 16: Code for Multilayer Perceptron Classifier (MLP)

The entropy_loss function is used to calculate the loss between the predicted output and the true target values. In neural network training, the goal is to minimize this loss during the optimization process.

Entropy Loss:

If the number of class is equal to 2, entropy loss can be calculated as:

$$-(y \log(p) + (1 - y) \log(1 - p))$$

If the number of class is larger than 2, we calculated a separate loss for each class label per observation and sum the result. The formular is as below.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

We utilise the formula above for calculating the loss value for each class in the code.

The fit method trains the MLP using the specified training data X and labels y for a given number of epochs. In our model, the epochs is 150. It initializes weights, performs forward and backward propagation, and updates the weights using gradient descent with learning rate equals to 0.001.

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

    L1 = X.dot(self.para["W1"]) + self.para["b1"]
    A1 = self.relu(L1)
    L2 = A1.dot(self.para["W2"]) + self.para["b2"]

      Z1, A1, Z2, na, na = self._forward_propagation(X)
    pred = self.sigmoid(L2)
    return np.round(pred)
    pass

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

    L1 = X.dot(self.para["W1"]) + self.para["b1"]
    A1 = self.relu(L1)
    L2 = A1.dot(self.para["W2"]) + self.para["b2"]
    output = self.sigmoid(L2)
    return output

      Z1, A1, Z2, output, loss = self._forward_propagation(X)
      return output

    pass
```

Figure 17: Code for Multilayer Perceptron Classifier (MLP)

Parameters	**Explanation**
L1	The first layer
A1	The first hidden layer
L2	The second layer

The predict method makes predictions for the input data X using the trained MLP. It calculates the forward pass and applies a threshold to round the output to binary predictions (0 or 1).

L1 = X.dot(self.para["W1"]) + self.para["b1"]:
Compute the weighted sum of the input features (X) using the weights (self.para["W1"]) and add the bias term (self.para["b1"]). This represents the input to the first hidden layer.

A1 = self.relu(L1):
Apply the rectified linear unit (ReLU) activation function to the result of the first hidden layer. ReLU introduces non-linearity to the model.

L2 = A1.dot(self.para["W2"]) + self.para["b2"]:
Compute the weighted sum of the output from the first hidden layer (A1) using the weights (self.para["W2"]) and add the bias term (self.para["b2"]). This represents the input to the output layer.

Pred = self.sigmoid(L2):
Apply the sigmoid activation function to the result of the output layer. This step compresses the values to the range (0, 1), representing probabilities.

The predict_proba method computes the probabilities of the positive class for each instance in the input data. Similar approach with predict function used in this function.

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

        L1 = self.X.dot(self.para["W1"]) + self.para["b1"]
        A1 = self.relu(L1)
        L2 = A1.dot(self.para["W2"]) + self.para["b2"]
        output = self.sigmoid(L2)
        loss = self.entropy_loss(self.y, output)

        self.para["L1"] = L1
        self.para["L2"] = L2
        self.para["A1"] = A1


        return output, loss
#         return Z1, A1, Z2, output, loss
        pass
```

Figure 18: Code for Multilayer Perceptron Classifier (MLP)

In the _forward_propagation function, this function is to generate predictions that can be compared with the true labels to compute the loss, which is a measure of how well the network is performing on the given task.

```
def _backward_propagation(self, output):
    # Implement backward propagation for MLP
    y_inv = 1-self.y.to_numpy().reshape(output.shape)
    output_inv = 1 - output

    #reshape y to match the output shape
    y_np = self.y.to_numpy().reshape(output.shape)

    #gradient of loss

    gra_output = (y_inv / self.eps(output_inv)) - (y_np / self.eps(output))

    # sigmoid activation in the second layer
    gra_sig = output * (output_inv)
    #gradient to the weighted sum with second layer
    gra_l2 = gra_output * gra_sig

    # the activation of the first layer
    gra_A1 = gra_l2.dot(self.para["W2"].T)
    gra_w2 = self.para["A1"].T.dot(gra_l2)
    gra_b2 = np.sum(gra_l2, axis=0)

    gra_l1 = gra_A1 * self.dRelu(self.para['L1'])
    gra_w1 = self.X.T.dot(gra_l1)
    gra_b1 = np.sum(gra_l1, axis=0)

    return gra_w1, gra_w2, gra_b1, gra_b2
```

Figure 19: Code for Multilayer Perceptron Classifier (MLP)

This function is for calculating the gradients of the loss with respect to the weights and bias during the backward propagation. This is important to update the weight and bias for the optimization of the model.

```
# Implement backward propagation for MLP
y_inv = 1-self.y.to_numpy().reshape(output.shape)
output_inv = 1 - output
```

Figure 20: Code for Calculating the Difference

This is for calculating the difference for using in the gradient computation to represent the "inverse" information from the original values.

```
gra_output = (y_inv / self.eps(output_inv)) - (y_np / self.eps(output))

# sigmoid activation in the second layer
gra_sig = output * (output_inv)
#gradient to the weighted sum with second layer
gra_l2 = gra_output * gra_sig
```

Figure 21: Code for Gradient Decent to Second Layer

Sigmoid Activation (gra_sig)
The output of a neural network is often passed through an activation function to introduce non-linearity. Computing the gradient of the sigmoid activation function (gra_sig) is required during backpropagation to apply the chain rule and calculate the impact of changes in the output on the overall loss.

Gradient Calculation (gra-output)
The gradient (gra_output) represents how much the loss function changes concerning changes in the output of the network. The calculation involves the differences between the predicted output (output) and the true labels (self.y). The goal is to understand how the loss would change based on small changes in the predicted output.

Gradient in the Second Layer (gra_l2):
The product of gra_output and gra_sig gives the gradient in the second layer (gra_l2). This gradient is crucial for understanding how the loss changes concerning changes in the weighted sum of inputs to the second layer. It is the contribution of the second layer to the overall loss.

These computation are part of the chain rule in calculus which is used to calculate the gradient descent in the network. By iteratively applying the chain rule through the network layer, this can help to minimise the loss by adjusting the weights and biases during the training process.

```
# the activation of the first layer
gra_A1 = gra_l2.dot(self.para["W2"].T)
gra_w2 = self.para["A1"].T.dot(gra_l2)
gra_b2 = np.sum(gra_l2, axis=0)
```

Figure 22: Code for Gradient Decent to First Layer

Gradient of Activation in the First Layer (gra_A1):
It represents how much the output of the first layer contributes to the overall error in the network.

Gradient of Weights in the Second Layer (gra_w2):
Gradient of Biases in the Second Layer (gra_b2):
It represents how much the weights and biases in the second layer need to be adjusted to reduce the error in the output.

```
gra_l1 = gra_A1 * self.dRelu(self.para['L1'])
gra_w1 = self.X.T.dot(gra_l1)
gra_b1 = np.sum(gra_l1, axis=0)
```

Figure 23: Code for Gradient Decent to Weights and Bias

Gradient of Activation in the First Layer with Respect to Weighted Sum (gra_l1):
It accounts for the impact of the first layer's activation on the overall error, taking into consideration the derivative of the ReLU activation function (self.dRelu).

Gradient of Weights in the First Layer (gra_w1):
Gradient of Biases in the First Layer (gra_b1):
It represents how much the weights and biases in the second layer need to be adjusted to reduce the error in the output.

Discussion

Predicted Result

Naïve bayes

model	accuracy	f1	precision	recall	mcc	auc
Naive Bayes	0.82758621	0.73684211	0.63636364	0.875	0.63053704	0.94047619
Naive Bayes	0.75862069	0.53333333	0.8	0.4	0.43709568	0.78421053
Naive Bayes	0.72413793	0.6	0.75	0.5	0.42133242	0.85784314
Naive Bayes	0.75862069	0.63157895	0.75	0.54545455	0.47153205	0.67171717
Naive Bayes	0.86206897	0.71428571	0.83333333	0.625	0.63705591	0.92857143
Naive Bayes	0.72413793	0.42857143	0.5	0.375	0.25613588	0.70833333
Naive Bayes	0.75862069	0.53333333	0.66666667	0.44444444	0.39338328	0.76666667
Naive Bayes	0.75	0.58823529	0.625	0.55555556	0.41110321	0.9005848
Naive Bayes	0.78571429	0.57142857	0.57142857	0.57142857	0.42857143	0.82993197
Naive Bayes	0.75	0.36363636	1	0.22222222	0.40298035	0.73099415
Naive Bayes Average	0.76995074	0.57012451	0.71327922	0.51141053	0.44897272	0.81193294

Figure 24: Predicted Result of Naïve Bayes

K Nearest Neighbour (KNN)

model	accuracy	f1	precision	recall	mcc	auc
KNN	0.827586207	0.615384615	0.8	0.5	0.535264361	0.913690476
KNN	0.793103448	0.625	0.833333333	0.5	0.524931436	0.723684211
KNN	0.724137931	0.5	1	0.333333333	0.476095229	0.904411765
KNN	0.75862069	0.631578947	0.75	0.545454545	0.471532048	0.744949495
KNN	0.862068966	0.666666667	1	0.5	0.64807407	0.946428571
KNN	0.75862069	0.363636364	0.666666667	0.25	0.297014037	0.663690476
KNN	0.793103448	0.5	1	0.333333333	0.506369684	0.877777778
KNN	0.714285714	0.333333333	0.666666667	0.222222222	0.256076256	0.739766082
KNN	0.714285714	0.428571429	0.428571429	0.428571429	0.238095238	0.724489796
KNN	0.75	0.461538462	0.75	0.333333333	0.374634325	0.815789474
KNN Average	0.769581281	0.512570982	0.78952381	0.39462482	0.432808668	0.805467812

Figure 25: Predicted Result of KNN

Multilayer Perceptron Classifier (MLP)

model	accuracy	f1	precision	recall	mcc	auc
MLP	0.75862069	0.631578947	0.545454545	0.75	0.471532048	0.851190476
MLP	0.75862069	0.588235294	0.714285714	0.5	0.438454067	0.705263158
MLP	0.689655172	0.571428571	0.666666667	0.5	0.34442336	0.799019608
MLP	0.724137931	0.555555556	0.714285714	0.454545455	0.389417942	0.797979798
MLP	0.793103448	0.666666667	0.6	0.75	0.526134054	0.857142857
MLP	0.724137931	0.428571429	0.5	0.375	0.256135882	0.607142857
MLP	0.75862069	0.363636364	1	0.222222222	0.405720413	0.772222222
MLP	0.892857143	0.823529412	0.875	0.777777778	0.749658792	0.964912281
MLP	0.678571429	0.571428571	0.428571429	0.857142857	0.412393049	0.741496599
MLP	0.714285714	0.428571429	0.6	0.333333333	0.278110657	0.842105263
MLP Average	0.749261084	0.562920224	0.664426407	0.552002165	0.427198026	0.793847512

Figure 26: Predicted Result of MLP

Table 1: Comparative Assessment of Performance Metric

	Naïve Bayes	**KNN**	**MLP**
Accuracy	Highest average accuracy (76.99%) among the three models, indicating the overall correctness of predictions.	Comparable accuracy (76.96%), demonstrating competitive performance	Moderately accurate (74.93%), slightly behind Naive Bayes and KNN.
	Naive Bayes and KNN show similar accuracies, while MLP has a slightly lower accuracy.		
F1 Score	Show a balanced F1 score (57.01%), indicating good precision and recall.	Demonstrates competitive F1 score (51.26%), balancing precision and recall.	Moderately balanced F1 score (56.29%), capturing precision and recall effectively.
	Naive Bayes and MLP exhibit better F1 scores compared to KNN.		
Precision and Recall	Shows higher precision (71.33%) but may sacrifice some recall.	Demonstrates strong recall performance (39.46%) but may have lower precision	Achieves a balanced trade-off between precision and recall.
	KNN demonstrates the highest precision, indicating a good ability to avoid false positives, but Naive Bayes and MLP show better recall compared to KNN.		

MCC	Demonstrates a moderate MCC (44.90%), indicating good agreement between predicted and observed classifications.	Competitive MCC (43.28%), suggesting good agreement.	Moderate MCC (42.72%), indicating reasonable agreement.
	The three models show similar MCC values, suggesting comparable overall performance.		
AUC	High average AUC (81.19%), reflecting good discriminatory power.	Competitive AUC (80.54%), indicating good discriminative ability.	Moderate AUC (79.38%), showing reasonable discriminative power.
	The three models show similar AUC values, suggesting comparable overall performance.		

Key Insights

Naïve Bayes

- The Naive Bayes model performs reasonably well, achieving a balanced trade-off between precision and recall, as indicated by the F1 score.
- The model has a moderate ability to correctly identify positive instances (recall), but there is room for improvement, especially in scenarios where recall is crucial.
- The MCC score indicates a decent overall performance, considering both true and false positives and negatives.
- The AUC value suggests that the model has a good ability to discriminate between positive and negative instances.

K Nearest Neighbour (KNN)

- The model shows reasonable accuracy, precision, and AUC, suggesting its ability to make correct positive predictions.
- The F1 score, MCC, and recall indicate a balance between precision and sensitivity, but there is room for improvement.
- The model tends to have higher precision, suggesting a good ability to correctly identify positive cases, but it might miss some instances due to lower recall.

Multilayer Perceptron Classifier (MLP)

- The MLP model has competitive performance compared to Naive Bayes and KNN.
- Precision and recall are balanced, suggesting an overall fair classification performance.
- There is room for improvement, especially in boosting recall and overall model robustness.

Implementation Challenges

Naïve Bayes

The challenge I met in Naïve Bayes is the result were all 0. This is confusing as there is no error in my code for Naïve Bayes but the result showed all 0. I was trying to check through and adjusting my algorithm of Naïve Bayes multiple times, the result was still all 0. In the end, I did my research and then realised that the Laplace smoothing I did to prevent zero probabilities is the one who causing the problem. This is due to I set the smoothing parameter too high, causing the probabilities to be too evenly distributed among classes. Hence, I

kept adjusting the smoothing parameter to achieve a suitable number to make the Naïve Bayes predict the result successfully.

K Nearest Neighbour (KNN)

KNN is relatively easier than the other two models. The biggest challenge I met is to successfully save the result into csv file. As KNN is the first model I did among the three model, it took me some time to figure it out what kind of format should I used to save the result. As the format of the result of KNN keep having trouble to save into csv file. Thus, I had to make some changes to evaluation_model to save my result to ensure that the evaluation_model can handle different data type consistent.

```
# Check if the model supports predict_proba method for AUC calculation
if hasattr(model, 'predict_proba'):
    proba = model.predict_proba(X_test)

    if isinstance(proba, pd.DataFrame):
        proba = proba.to_numpy()

    if len(np.unique(y_test)) == 2:  # Binary classification
        if proba.ndim > 1 and proba.shape[1] > 1:
            if len(np.unique(y_test)) > 2:
                y_test = (y_test == positive_class_label).astype(int)
        else:
            auc = roc_auc_score(y_test, proba)

        #print(f"AUC: {auc}")

    else:  # Multiclass classification
        if len(np.unique(y_test)) > 2:
            y_test = (y_test == positive_class_label).astype(int)
        #print(f"AUC: {auc}")
```

Figure 27: Code of Edited Evaluate_Model Function

The changes make sure that the predicted probabilities convert to numPy array for the consistency. Furthermore, for binary classification, it checks if the positive class label is specified. If it is, it converts y_test to a binary format with 1 for the positive class and 0 for others. For multiclass classification, it ensures that y_test is not a multilabel indicator matrix. If more than two unique classes are present in y_test, it checks if the positive class label is specified and converts y_test to binary format accordingly.

Multilayer Perceptron Classifier (MLP)

When building the MLP model, the problem I kept meeting is matric misshape. Due to there are multiple layers include in the model, it would need to perform multiple matric multiplications for adjusting the weights and bias. Thus, it is the point where the matric misshape easily happened. It required me to check through every shape of the input data, weights and bias to understand where is the source of the problem. Moreover, I would need to go through a few times of trial and error to reshape and transform the matric into correct shape for working. It was time consuming for solving the problem.

The other problem I met is the number of input features was wrong at the beginning. It should have 77 features for the MLP model to run through but there were only 36 features had been processed. At first, I thought of the problem existed in the cross validation section, so I tried to split the data again with different way but nothing is working. The problem was solved by I accidentally found out that my data pre-processing was wrong. I extracted the wrong number of Boolean features which causing the number of input data is incorrect.

I should be more aware about the shape of the input data, weights and bias for the ease of building the MLP model.

Conclusion

In conclusion, the three models demonstrate moderate performance and has their own strength and weakness to deal with. Each model is suitable for different condition. For precision emphasis, Naïve Bayes could be chosen where high precision is crucial. For recall emphasis, KNN could be considered for applications

prioritizing high recall. MLP is suitable to provide a good balance between precision and recall for moderately complex relationships. A lot challenges have been met during the processing of implementing the three models but those are managed to solve it at the end. However, there is a room for improvement. For future optimization, ensemble methods, or feature engineering may be considered to enhance model performance based on specific dataset characteristics.