Introduction to ML Term Project

110550142 黃芷柔

1. Read csv file(有更改 helper.cpp)

   由於直接讀取時，遇到空的資料會出現錯誤訊息:

```
terminate called after throwing an instance of 'std::invalid_argument'
  what():  stof
```

   因此在讀資料時，如果偵測到空格就先輸入-1(原本數據裡沒有-1)。如果某一行的最後是逗號，代表那行缺最後一個資料，先在那行後面加上-1。

```
vector<float> row;
if(line[line.size()-1]==','){
    line+="-0.1";
}
stringstream ss(line);
string cell;


while (getline(ss, cell, ',')) {

    if (!cell.empty()) {
        row.push_back(stof(cell));
    } else {
        row.push_back(-1.0);
    }
}

data.push_back(row);
```

2. Data Preprocessing:

   觀察資料，可以看出 F1~F17 是 Numerical Data，F18~F77 是用 0/1 表的資料，可當作 Categorical Data 處理。

```
for (int col = 0; col < 17; ++col) {
    if(IsVal==0){
        float mean_val = 0.0;
        for (int row = 0; row < df.size(); ++row) {
            if(df[row][col]!=-1)mean_val += df[row][col];
        }
        mean_val /= df.size();
        preprocess_parameter[col][0]=mean_val;

        float std_dev = 0.0;
        for (int row = 0; row < df.size(); ++row) {
            if(df[row][col]!=-1)std_dev += pow(df[row][col] - mean_val, 2);
        }
        std_dev = sqrt(std_dev / df.size());
        preprocess_parameter[col][1]=std_dev;
    }

    // Apply Z-score normalization to each element in the column
    for (int row = 0; row < df.size(); ++row) {
        if(df[row][col]!=-1)df[row][col] = (df[row][col] - preprocess_parameter[col][0]) / preprocess_parameter[col][1];
        if(df[row][col]==-1)df[row][col]=0;
    }
}
```

   Numerical Data:用 Standardization 做 Normalization，先取不是-1 數據的平均跟標準差，再把每個數據代入以下算式，最後把空的數字用 0 取代(Standardization 後的平均是 0)。

$$a_i' = \frac{a_i - \bar{a}}{sd(a)}$$

```
for (size_t col = 17; col < 77; ++col) {

    if(IsVal==0){
        int cnt0=0,cnt1=0;
        for (int row = 0; row < df.size(); ++row) {
            if(df[row][col]==0)cnt0++;
            if(df[row][col]==1)cnt1++;
        }
        int mod;
        if(cnt0>cnt1){
            mod=0;
        }else{
            mod=1;
        }
        preprocess_parameter[col][0]=mod;
    }

    for(int row=0;row<df.size();row++){
        if(df[row][col]==-1)df[row][col]=preprocess_parameter[col][0];
    }

}
```

Categorical Data:計算數據的眾數，用眾數取代空的數據。

變數 IsVal 是紀錄數據是否為 validation，為了預防 potential data leakage issue，每個 fold 分出 train data 跟 validation data 後，先 preprocess train data，接著再用 train 算出來的 parameters(numeric data 的平均和標準差，categorical data 的眾數)用 global vector 處存，處理 validation data。

3. Classifiers

(1) Naïve Bayes:

```
private:
    // Implement private function or variable if you needed
    unordered_map<int, vector<float>> class_probabilities;
    unordered_map<int, vector<vector<float>>> class_parameters;
```

Class probabilities 負責記錄各種結果的機率，class parameters 是紀錄每個 feature 計算條件機率會用到的數據。

fit:

```
unordered_map<int, vector<vector<float>>> separated_data;
for (size_t i = 0; i < X.size(); ++i) {
    int label = y[i][0];
    separated_data[label].push_back(X[i]);
}

size_t total_samples = X.size();
for (auto &entry : separated_data) {
    int label = entry.first;
    size_t class_samples = entry.second.size();
    float class_probability = static_cast<float>(class_samples) / total_samples;
    class_probabilities[label] = {class_probability};
}
```

先根據結果是 0 或 1 把數據分成兩疊，再除以全部數據數，得到 class probabilities。

```cpp
for (auto &entry : separated_data) {
    int label = entry.first;
    vector<vector<float>> class_data = entry.second;
    size_t num_features = class_data[0].size();

    vector<float> mean_values(17, 0.0);
    vector<float> std_dev_values(17, 0.0);

    for (size_t i = 0; i < 17; ++i) {
        for (size_t j = 0; j < class_data.size(); ++j) {
            mean_values[i] += class_data[j][i];
        }
        mean_values[i] /= class_data.size();
    }

    for (size_t i = 0; i < 17; ++i) {
        for (size_t j = 0; j < class_data.size(); ++j) {
            std_dev_values[i] += pow(class_data[j][i] - mean_values[i], 2);
        }
        std_dev_values[i] = sqrt(std_dev_values[i] / class_data.size());
    }

    vector<vector<float>> parameters;
    parameters.push_back(mean_values);
    parameters.push_back(std_dev_values);
    class_parameters[label] = parameters;
```

接著，用之前分類好的 data 處理每個 feature 的條件機率，F1~F17 是 numeric data，用 normal distribution 計算機率，算出每項的平均和標準差，存入 class parameter 以便之後預測時計算。

```cpp
for(size_t i=17;i<77;i++){
    float class_feature_count[2]={0,0};
    for (size_t j = 0; j < class_data.size(); ++j) {
        if(class_data[j][i]==1){
            class_feature_count[1]++;
        }else{
            class_feature_count[0]++;
        }
    }

    class_parameters[label][0].push_back(class_feature_count[0]/(float)class_data.size());
    class_parameters[label][1].push_back(class_feature_count[1]/(float)class_data.size());

}
```

F18~F77 是 categorical data，計算 0 和 1 在那個結果下出現的次數，再用計算出的次數除以該結果的總數，把機率存到 class parameter。

Predict:

```cpp
vector<vector<float>> predict(vector<vector<float>> &X) override {
    // Implement the prediction logic for Naive Bayes classifier
    vector<vector<float>> predictions;

    float zero=0.00001;
    for (auto &sample : X) {
        unordered_map<int,float> result;

        for (auto &entry : class_probabilities) {
            int label = entry.first;
            vector<float> class_prob = entry.second;

            float log_probability = log(class_prob[0]);

            for (size_t i = 0; i < 17; ++i) {
                float mean = class_parameters[label][0][i];
                float std_dev = class_parameters[label][1][i];

                float exponent = exp(-pow(sample[i] - mean, 2) / (2 * pow(std_dev, 2)));
                float feature_probability = (1 / (sqrt(2 * 3.14) * std_dev)) * exponent;
                if(feature_probability==0){
                    feature_probability=zero/(float)class_prob[1];
                }

                log_probability += log(feature_probability);
            }
            for(size_t i=17;i<77;i++){
                float feature_probability=class_parameters[label][sample[i]][i];
                if(feature_probability==0){
                    feature_probability=zero;
                }
                log_probability+=log(feature_probability);
            }
            result[label]=log_probability;
        }
```

分別計算每個 test data 不同結果的機率，把每個條件機率的 log 值相加，numeric data 代入這個算式:

$$N(x,\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Categorical data 依照 feature 的值找出 class parameter 存的條件機率。兩個種類的數據如果有遇到條件機率=0 的話，就設成 0.00001

```
    int predicted_label;
    if(result[1]>result[0]){
        predicted_label=1;
    }else{
        predicted_label=0;
    }
    predictions.push_back({static_cast<float>(predicted_label)});
}

return predictions;
```

算出機率後，最大的作為預測結果。

Predict probability:

```
        class_probability[label] = final_probability;
    }
    class_probability[0]=class_probability[0]/total_prob;
    class_probability[1]=class_probability[1]/total_prob;

    probabilities.push_back(class_probability);
}

return probabilities;
```

算機率的流程跟 *predict* 相同，只是最後算出來的 log 值要 exponential。predict 時只需要比大小，只有算出 P(D|h)*P(h)，而 predict probability 需要算出 P(h|D)，h 是 hypothesis，D 是 data，因此算完兩種結果的 P(D|h)*P(h)後，相加得到 P(D)，再拿原本算的 P(D|h)*P(h)除以 P(D)，得到 P(h|D)，回傳最後算出的機率。

(2) KNN

```
KNearestNeighbors(int k = 3): k(k) {

}
```

```
private:
    // Implement private function or variable if you needed
    int k;
    vector<vector<float>>train_X;
    vector<vector<float>>train_y;
```

把 KNN 的 k 設成 3，用最接近的三個點預測結果。train_X, train_y 用來儲存 fit 時收到的 X, y，在 predict 時用來預測。

Fit

```
void fit(vector<vector<float>> &X, vector<vector<float>> &y) override {
    // Implement the fitting logic for KNN
    train_X=X;
    train_y=y;
}
```

把輸入的 X, y 放在 train_X, train_y

predict

```
vector<vector<float>> predictions;
for(const auto &sample:X){
    vector<float>distances;
    for(int j=0;j<train_X.size();j++){
        float dis=coutDistance(sample,train_X[j]);
        distances.push_back(dis);

    }
    vector<int> indices(distances.size());
    iota(indices.begin(), indices.end(), 0);
    partial_sort(indices.begin(), indices.begin() + k, indices.end(),
        [&distances](int i, int j) { return distances[i] < distances[j]; });
```

建一個 vector<vector<float>>prediction，作為回傳的值。計算輸入的 X 和每個 train_X 的距離，存在 vector<float>distance 裡。再建立 vector<float> indices，大小跟 distance 一樣，是 train_X 的數量，內容設為從 0 開始的連續整數，接著把 indices 依據距離大小排序，取前 k 個最小的。

```
float coutDistance(const vector<float> &a, const vector<float> &b){
    float distance = 0.0;
    for (size_t i = 0; i < a.size(); ++i) {
        distance += pow(a[i] - b[i], 2);
    }
    return sqrt(distance);
}
```

算距離是用 Euclidean Distance，取每個 feature 差的平方，全部加在一起後開根號。

```
    unordered_map<int, int> class_counts;
    for (int i = 0; i < k; ++i) {
        int neighbor_index = indices[i];
        int label = static_cast<int>(train_y[neighbor_index][0]);
        class_counts[label]++;
    }
    int predict_label=-1;
    int max_cnt=-1;
    for (const auto &pair : class_counts) {
        if (pair.second > max_cnt) {
            max_cnt = pair.second;
            predict_label = pair.first;
        }

    }
    predictions.push_back({static_cast<float>(predict_label)});

}
//return vector<vector<float>>();
return predictions;
```

設 undered_map<int,int>class_count，用 indices 找出對應 y 值，紀錄 k 個數據裡結果是 0 跟 1 幾個。接著找出占 k 個中數目最多的 class，作為預測的 label，轉成 float 後存入 predictions。每個 X 的預測數據跑完後回傳 predictions。

predit probability

```
vector<unordered_map<int, float>> probabilities;
```

一開始先設 vector<unordered_map<int,float>>probabilities 紀錄要回傳的結果，Predict probability 跟 predict 算 distances，依據 distances 對 indices

排序並出前 k 個的方式一樣。

```
        unordered_map<int, float> class_probabilities;
        for (int i = 0; i < k; ++i) {
            int neighbor_index = indices[i];
            int label = static_cast<int>(train_y[neighbor_index][0]);
            class_probabilities[label]++;
        }

        for (auto &pair : class_probabilities) {
            pair.second /= k;
        }

        probabilities.push_back(class_probabilities);
    }

    return probabilities;
```

設 unordered_map<int, float>class_probabilities 用來存 sample X 的 label 和相對應的機率，用 indices 找出對應 y 值作為 label，計算 k 個資料裡每個 class 出現的次數，算完後把各個 class 的數量除以 k，得到機率。把算出來的機率存進 probabilities，全部算後完回傳 probabilities。

(3) MLP

```
private:
    // Implement private function or variable if you needed
    int input_size;
    int hidden_size;
    int output_size;
    int epochs;
    float learning_rate;

    vector<vector<float>> inputLayer;
    vector<vector<float>> hiddenLayer;
    vector<vector<float>> outputLayer;

    vector<vector<float>> inputWeights;
    vector<vector<float>> hiddenWeights;
```

input_size, hidden_size, output_size 分別是指各個 layer 的 node 數。epochs 跟 learning_rate 是訓練時會用到的參數。inputLayer, hiddenLayer, outputLayer 用來記錄每層訓練的狀況。inputWeight 是 inputLayer 到 hiddenLayer 的 weight，hiddenWeight 則是 hiddenLayer 到 outputLayer 的 weight。

```
MultilayerPerceptron(int input_size=77, int hidden_size=77, int output_size=1)
    : input_size(input_size), hidden_size(hidden_size), output_size(output_size) {
    initializeParameters();
}
```

```
void initializeParameters() {
    epochs=10;
    learning_rate = 0.05;

    inputWeights = vector<vector<float>>(input_size, vector<float>(hidden_size));
    hiddenWeights = vector<vector<float>>(hidden_size, vector<float>(output_size));

    default_random_engine generator;
    normal_distribution<float> distribution(0.0, 1.0);


    for (int i = 0; i < input_size; ++i) {
        for (int j = 0; j < hidden_size; ++j) {
            inputWeights[i][j]= distribution(generator);
        }
    }

    for (int i = 0; i < hidden_size; ++i) {

        for (int j = 0; j < output_size; ++j) {
            hiddenWeights[i][j]= distribution(generator);
        }
    }

}
```

初始化時，input_size 設為 77(feature 數)，hidden_size 設為 77，output_size 設為 1(用這個 node 算出來數值大小判斷結果)。把 epochs 設 10，learning_rate 設 0.05。而 inputWeight 跟 hiddenWeight 是用 default_random_engine 跟平均為 0 標準差為 1 的 normal_distributoin 生成的隨機變數。

fit

```
void fit(vector<vector<float>> &X, vector<vector<float>> &y) override {
    // Implement training logic for MLP including forward and backward propagation
    learning_rate = 0.05;
    for (int epoch = 0; epoch < epochs; ++epoch) {
        cout<<"epoch:"<<epoch<<"/"<<epochs<<endl;

        learning_rate=(20*learning_rate)/(20+epoch);

        for (size_t i = 0; i < X.size(); ++i) {
            vector<vector<float>> input = {X[i]};
            vector<vector<float>> target = {y[i]};

            _forward_propagation(input);
            _backward_propagation(target);

        }
    }
}
```

訓練時，為了讓每個 iteration 初始 learning rate 一樣，一開始把 learning rate 設成 0.05，learning rate 的是配合 epoch 做變化 epoch 越後面 learning rate 越小。每個 epoch 中，一次拿一個 train X 和 train Y 進行 forward propagation 跟 backward propagation，直到全部 train X 訓練完，這個 epoch 才結束。總共執行 epochs=10 次。

predict

```
vector<vector<float>> predict(vector<vector<float>> &X) override {
    // Implement prediction logic for MLP
    vector<vector<float>> predictions;
    for (auto &sample : X) {
        vector<vector<float>> input = {sample};

        _forward_propagation(input);
        float threshold = 0.5;
        vector<float>predicted_label;
        predicted_label.push_back((outputLayer[0][0]> threshold) ? 1 : 0);


        predictions.push_back(predicted_label);

    }
    return predictions;
    //return vector<vector<float>>();
}
```

首先設 predicions 用來儲存預測結果。把 test X 一個一個進行 forward propagation，接著比較 outputLayer[0][0]跟 threshold 大小，因為 forward propagation 是用 sigmoid，結果界在 0~1，如果超過 0.5 預測是 1，否則是 0。

Predict probability

```
unordered_map<int, float> class_probabilities;

class_probabilities[0] = 1-outputLayer[0][0];
class_probabilities[1] = outputLayer[0][0];


probabilities.push_back(class_probabilities);
```

做法跟 predict 相似，把 test X 一個一個 forward propagation 後，得到預測結果，因為 forward propagation 是用 sigmoid，outputLayar[0][0]界在 0~1，就作為結果是 1 的機率，1- outputLayar[0][0]則當作結果是 0 的機率，把機率存在 class_probabilities，算完放入 probabilities，全部的 test X 預測完後，回傳 probabilities。

Forward propagation

```
void _forward_propagation(vector<vector<float>> &X) {
    // Implement forward propagation for MLP
    inputLayer = X;
    hiddenLayer = activate(dotProduct(inputLayer, inputWeights));
    outputLayer = activate(dotProduct(hiddenLayer, hiddenWeights));

}
```

inputLayer 設成輸入的 X，inputLayer(size: 1*input_size)乘以 inputWeights(size: input_size*hidden_size)，再用 sigmoid activate，得到 hiddenLayer。

hiddenLayer(size: 1*hidden_size)乘以 hiddenWeight(size: hidden_size*output_size)，用 sigmoid activate 後，得到 outptLayer。

```
vector<vector<float>> dotProduct(const vector<vector<float>> &A, const vector<vector<float>> &B) {
    size_t m = A.size();
    size_t n = B[0].size();
    size_t p = B.size();

    vector<vector<float>> result(m, vector<float>(n, 0.0));

    for (size_t i = 0; i < m; ++i) {
        for (size_t j = 0; j < n; ++j) {
            for (size_t k = 0; k < p; ++k) {
                result[i][j] += A[i][k] * B[k][j];

            }
        }
    }

    return result;
}
```

dotProduct 是輸入兩個 vector<vector<float>>進行矩陣乘法，回傳的矩陣一樣是 vector<vector<float>>。

```
vector<vector<float>> activate(const vector<vector<float>> &matrix) {
    vector<vector<float>> result = matrix;

    for (auto &row : result) {
        for (float &val : row) {
            val = 1.0 / (1.0 + exp(-val));
        }
    }

    return result;
}
```

Activate 是用 sigmoid，算法是把輸入的矩陣每一項都帶入：

$$\frac{1}{1+e^{-x}}$$

最後可以得到界在 0~1 的值。

Backward propagation

```
void _backward_propagation(vector<vector<float>> &target) {
    // Implement backwardpropagation for MLP

    vector<vector<float>> out_error;
    vector<float>out_error_temp;

    out_error_temp.push_back(outputLayer[0][0]-target[0][0]);
    out_error.push_back(out_error_temp);


    vector<vector<float>> outputGradient = multiply(out_error,(1-outputLayer[0][0])*outputLayer[0][0]);
    hiddenWeights = subtract(hiddenWeights, multiply(dotProduct(transpose(hiddenLayer),outputGradient), learning_rate));

    vector<vector<float>> hiddenGradient = dotProduct(outputGradient, transpose(hiddenWeights));
    for(int i=0;i<hidden_size;i++){
        hiddenGradient[0][i]*=hiddenLayer[0][i]*(1-hiddenLayer[0][i]);
    }
    inputWeights = subtract(inputWeights, multiply(dotProduct(transpose(inputLayer), hiddenGradient), learning_rate*10));

}
```

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = -\eta \cdot \frac{\partial E_d}{\partial net_j} \cdot x_{ji} = \eta \cdot \delta_j \cdot x_{ji}$$

參考講義的算式求得△hiddenWeight。

首先求 error=output-target，雖然算出來只有一項，但用矩陣 out_error 儲存方便之後運算，設 outGradient=out_error*(1-o)*o。最後算出 hiddenWeight= hiddenWeight - (trans(hiddenLayer)*outputGradient*learning rate)

接著求 hiddenGradient，參考以下算式

$$\delta_j = o_j \cdot (1 - o_j) \sum_{m \in Downstream(j)} \delta_m \cdot w_{mj}$$

先求出 outputGradient* trans(hiddenWeight)，得到 size=1*hidden size 的矩陣，接著這個矩陣的每項乘以對應的$o_j \cdot (1 - o_j)$，o_j是 hiddenLayer 裡對應的項，可以得到 hiddenGradient。最後用 hiddenGradient 更新 inputWeight，也就是每項 inputWeight 減 inputLayer* hiddenGradient* learning rate*10(hidden layer 的 learning rate 比 output layer 的 learning rate 大十倍的效果較好)。

```cpp
vector<vector<float>> transpose(const vector<vector<float>> &matrix) {
    // Transpose a matrix
    size_t m = matrix.size();
    size_t n = matrix[0].size();

    vector<vector<float>> result(n, vector<float>(m, 0.0));

    for (size_t i = 0; i < m; ++i) {
        for (size_t j = 0; j < n; ++j) {
            result[j][i] = matrix[i][j];
        }
    }

    return result;
}
```

Transpose function 是輸入 matrix，會把新的 matrix[j][i]的值設為 matrix[i][j]，最後回傳 transpose matrix。

```cpp
vector<vector<float>> subtract(const vector<vector<float>> &A, const vector<vector<float>> &B) {
    size_t m = A.size();
    size_t n = A[0].size();

    vector<vector<float>> result(m, vector<float>(n, 0.0));

    for (size_t i = 0; i < m; ++i) {
        for (size_t j = 0; j < n; ++j) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }

    return result;
}
```

substract 是輸入兩個大小相同的矩陣，相同位置的項相減，回傳減完的結果矩陣。

```cpp
vector<vector<float>> multiply(const vector<vector<float>> &A, float scalar) {
    size_t m = A.size();
    size_t n = A[0].size();

    vector<vector<float>> result(m, vector<float>(n, 0.0));

    for (size_t i = 0; i < m; ++i) {
        for (size_t j = 0; j < n; ++j) {
            result[i][j] = A[i][j] * scalar;
        }
    }

    return result;
}
```

Multiply 是輸入一個 matrix 跟 scalar，矩陣的每項都乘上 scalar，回傳乘完的結果。

Discussion:

1. implementation challenges:

   Naïve Bayes:

   主要是要注意 numeric data 跟 categorical data 要分開處理，以及要處理 categorical data 的 feature 條件機率=0，這次 term project 是分配一個很小的值。除此之外沒有遇到太大的問題。

   KNN:

   k 的大小難以決定，太小的話怕容易被特定值干擾，太大的話怕被無關的數據影響，嘗試了大一些的數字效果不太好，最後決定 k=3。另外 feature 數多，計算距離需要花比較多時間。

   MLP:

   MLP 比前兩個 model 複雜，很容易搞混變數的形式跟位置造成錯誤的結果。原本打算照 frame 給的 input size=64, hidden size=64, output size=64，但只需要判斷 0/1 的話，一個 output node 也可以達成，較簡單且花較少時間。Activate 的方式 sigmoid 效果看起來比 relu 好，可能是沒有那麼多層，還沒遇到 weight vanishing。另外 epochs 跟 learning rate 也難以決定，需要多次嘗試和比較。

2. prediction results:

   Naïve Bayes:

1	0.861111	0.8	0.769231	0.833333	0.695182	0.920139
1	0.75	0.608696	0.538462	0.7	0.43756	0.851923
1	0.694444	0.56	0.411765	0.875	0.431254	0.848214
1	0.833333	0.769231	0.714286	0.833333	0.644658	0.875
1	0.75	0.571429	0.461538	0.75	0.432771	0.727679
1	0.694444	0.421053	0.363636	0.5	0.22563	0.723214
1	0.805556	0.758621	0.733333	0.785714	0.597148	0.896104
1	0.75	0.64	0.615385	0.666667	0.449823	0.638889
1	0.771429	0.666667	0.615385	0.727273	0.498581	0.814394
1	0.742857	0.571429	0.5	0.666667	0.401363	0.777778

   KNN:

2	0.75	0.526316	0.714286	0.416667	0.397033	0.756944
2	0.833333	0.625	0.833333	0.5	0.5547	0.803846
2	0.861111	0.666667	0.714286	0.625	0.5815	0.866071
2	0.805556	0.631579	0.857143	0.5	0.545921	0.888889
2	0.805556	0.363636	0.666667	0.25	0.322329	0.790179
2	0.777778	0.333333	0.5	0.25	0.236228	0.544643
2	0.805556	0.666667	1	0.5	0.615882	0.849026
2	0.75	0.470588	0.8	0.333333	0.397573	0.628472
2	0.685714	0.47619	0.5	0.454545	0.253012	0.632576
2	0.771429	0.333333	0.666667	0.222222	0.286896	0.615385

MLP:

3	0.777778	0.636364	0.7	0.583333	0.482382	0.885417
3	0.916667	0.857143	0.818182	0.9	0.800315	0.923077
3	0.916667	0.842105	0.727273	1	0.805823	0.964286
3	0.916667	0.857143	1	0.75	0.816497	0.9375
3	0.972222	0.933333	1	0.875	0.919145	0.883929
3	1	1	1	1	1	1
3	1	1	1	1	1	1
3	0.972222	0.956522	1	0.916667	0.938083	1
3	0.971429	0.952381	1	0.909091	0.934199	0.916667

三個結果相比，MLP 的表現較 Naïve Bayes, KNN 好。Naïve Bayes, KNN 的表現差不多，整體上 Naïve bayes 的 accuracy 比 KNN 低，但 f1 score 和 mcc 表現得比 KNN 稍好。

3. key insights

根據前面結果可以發現 MLP 的結果較好。我認為可能這個數據的 features 之間不是完全 conditional independent，也不能單純就距離判斷結果，可能有更複雜的關係。MLP 有 back propagation，會對比 output 跟 target 的差異，進而修正 Model，使得 model 可以判斷哪個 feature 比較重要，而 Naïve Bayes 跟 KNN 沒有這個機制，運算時會以同樣比重考慮所有的 features。這代表或許有些 features 跟結果沒有關係，會干擾預測結果。