Machine Learning Term Project

林家甫
資訊工程學系
學號：110550124

摘要：

此次報告我將其分成四個階段，第一個階段說明我如何做資料前處理，第二個階段我會說明我如何實現我的分類器，第三個部分會是模型的評測與討論，最後一部份則說我遇到什麼困難，模型的總結和心得感想。

1. Data Preprocessing

```python
def _preprocess_imputation(self,df):
    # Using mode to imputate
    cnt_col = 0
    for col in df.columns:
        cnt_col+=1
        sum, denominator = 0, 0
        nan_index, t_index, f_index = [], [], []
        for index, row in df.iterrows():
            if cnt_col <= 17:
                if pd.isna(row[col]):
                    nan_index.append(index)
                else:
                    sum+=row[col]
                    denominator+=1
            else:
                if pd.isna(row[col]):
                    nan_index.append(index)
                else:
                    if row[col] == 1:
                        t_index.append(index)
                    else:
                        f_index.append(index)
        if cnt_col <= 17:
            avg = round(sum/denominator, 2)
            for index in nan_index:
                df.at[index,col] = avg
        elif cnt_col <= 77:
            if len(t_index) > len(f_index):
                for index in nan_index:
                    df.at[index,col] = 1
            else:
                for index in nan_index:
                    df.at[index,col] = 0
```

1.1 Data imputation:

首先，因為在training data 中，只有少少的300多筆資料而已，故每一筆資料所提供的資料都很重要，我並不想把有空缺的資料捨棄掉。因此，我利用imputation 將不齊全的資料補足。

- 對於連續資料，我利用平均數將資料補齊。
- 對於類別資料，我利用衆數將資料補齊。

```python
def _preprocess_numerical(self, df):
    # Custom logic for preprocessing numerical features goes here
    cnt_col = 0
    for col in df.columns:
        cnt_col+=1
        if cnt_col > 17: break
        maxx,minn = max(df[col]), min(df[col])
        # print(f"feature:{cnt_col} max: {maxx} min: {minn}")
        for index, row in df.iterrows():
            df.at[index, col] = round((row[col] - minn)/(maxx - minn), 4)
    # print("preprocess numeric:")
    # print(df.head())
    return df
```

1.2 Normalizaiton:

接下來，是我在做knn時發現的問題。當我沒有normalize資料時，距離很容易被連續性的特徵給影響。因為任意兩個連續性的特徵相減常常都遠大於1，但每一個類別的特徵相減最大值就是1。因此，會造成每個資料特徵"貢獻度" 嚴重不同。我利用normalization 的方法，將所有的連續性資料都變成介於0~1，來提升knn的結果。

公式:

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

我會在第四部分補充助教在最後幾天發布資料處理公告的程式碼改動。

2. Models' Details

本次作業的重點與精華，也是我花最多時間的地方。我這次把3種classifier 都實做一遍，下面我會一一做介紹。最後，我多補充了一個single neuron模型。

2.1 Naive Bayes Classifier

前置作業: 先將probability table 給建立好。我利用多個陣列的方式來記錄我之後要用到的機率資訊。這邊，我假設所有的特徵都是相互獨立的。做這樣的假設原因很簡單，可以大幅降低我模型的複雜度，有效的減小計算成本，優化執行時間。原因就是我可以不用考慮到機率中的連鎖律，直接相乘機率就能得到兩事件同時發收的機率。(第一張圖變成第二張圖)

公式:

- Chain rule

$$P(f_1 = v_1, f_2 = v_2, \dots, f_d = v_d)$$
$$= P(f_1)P(f_2|f_1)P(f_3|f_1f_2)P(f_4|f_1f_2f_3)\cdots P(f_d|f_1f_2f_3\cdots f_{d-1})$$

- Naïve Bayes' Classifier – assuming conditional independence

$$Class_{pred} = argmax_{c_i \in C} P(C = c_i | f_1 = v_1, f_2 = v_2, \dots, f_d = v_d) =$$
$$argmax_{c_i \in C} P(f_1 = v_1, f_2 = v_2, \dots, f_d = v_d | C = c_i) \cdot P(C = c_i) =$$
$$argmax_{c_i \in C} P(C = c_i) \cdot \prod_{i=1}^{d} P(f_i = v_i | C = c_i)$$

```
def __init__(self):
    # Initialize the probability table for categorical features. feature_result -> true_true
    self.class_prior_true_true = []
    self.class_prior_true_false = []
    self.class_prior_false_true = []
    self.class_prior_false_false = []

    # Initialize the probability table for continuous features. feature_result -> true_true
    self.conti_mean_true = []
    self.conti_SD_true = []
    self.conti_mean_false = []
    self.conti_SD_false = []

    # Different between data set
    self.conti_feature_start = 0
    self.conti_feature_end = 16
    self.class_feature_start = 17
    self.class_feature_end = 76
    # Calculate how many true false cases in training dataset
    self.true_cases = 0
    self.false_cases = 0
```

```python
def fit(self, X, y):
    # Implement the fitting logic for Naive Bayes classifier

    # Continuous features
    for m, column_name in enumerate(X.columns):
        tmp1, tmp2 = [], []
        for n, row in X.iterrows():
            if y[n] == 1: tmp1.append(X[column_name][n])
            else: tmp2.append(X[column_name][n])

        self.conti_mean_true.append(np.mean(tmp1))
        self.conti_SD_true.append(np.std(tmp1))
        self.conti_mean_false.append(np.mean(tmp2))
        self.conti_SD_false.append(np.std(tmp2))
        if m >= self.conti_feature_end : break
```

```python
    # Categorical features
    for m, column_name in enumerate(X.columns):
        tmp1, tmp2, tmp3, tmp4, cnt1, cnt2 = 0, 0, 0, 0, 0, 0
        if m >= self.class_feature_start:
            for n, row in X.iterrows():
                if y[n] == 1:
                    cnt1 += 1
                    if X[column_name][n] == 1: tmp1 += 1
                    else: tmp2 += 1
                else:
                    cnt2+=1
                    if X[column_name][n] == 1: tmp3 += 1
                    else: tmp4 += 1
            self.class_prior_true_true.append(tmp1 / cnt1)
            self.class_prior_false_true.append(tmp2 / cnt1)
            self.class_prior_true_false.append(tmp3 / cnt2)
            self.class_prior_false_false.append(tmp4 /cnt2)
            self.true_cases = cnt1
            self.false_cases = cnt2
```

在fit的部分，將table 建立好，以便在之後predict 時較容易計算。

```python
def predict(self, X):
    # Implement the prediction logic for Naive Bayes classifier
    y = []
    for m, row in X.iterrows():
        truee, falsee = 1, 1
        # Calculate the prob of true
        for n, column_name in enumerate(X.columns):
            if n <= self.conti_feature_end:
                truee *= self.Normal_distribution(n, X[column_name][m], T = True)
            else:
                if row[column_name] == 1:
                    truee *= self.class_prior_true_true[n - self.class_feature_start]
                else:
                    truee *= self.class_prior_false_true[n - self.class_feature_start]

        # Calculate the prob of false
        for n, column_name in enumerate(X.columns):
            if n <= self.conti_feature_end:
                falsee *= self.Normal_distribution(n, X[column_name][m], T = False)
            else:
                if row[column_name] == 1:
                    falsee *= self.class_prior_true_false[n - self.class_feature_start]
                else:
                    falsee *= self.class_prior_false_false[n - self.class_feature_start]

        truee *= self.true_cases
        falsee *= self.false_cases

        if truee >= falsee:
            y.append(np.int64(1))
        else:
            y.append(np.int64(0))

    y = np.array(y)
    # print(y)
    return y
```

在 predict 的部分，則是帶公式，將model predict true 和false的"值"算出來，比較大小。1為true，0為false。

predict_prob基本上程式碼沒有差異太多，唯一的差異就是在輸出時是輸出機率值，而非預測0、1。

(補充:上述引號""中的值，僅是正比於機率的值，並非真正的機率值。predict_prob則是真正的機率值)

```python
def Normal_distribution(self, index, value, T):
    # I eliminate "np.sqrt(2 * np.pi)" to lower the truncation error
    if T:
        part1 = 1 / self.conti_SD_true[index]
        part2 = np.exp(-((value - self.conti_mean_true[index])**2) / (2 * self.conti_SD_true[index]**2))
    else:
        part1 = 1 / self.conti_SD_false[index]
        part2 = np.exp(-((value - self.conti_mean_false[index])**2) / (2 * self.conti_SD_false[index]**2))

    # print(part1*part2)
    return part1 * part2
```

在連續資料特徵時，我用的是normal distribution來預測。

公式:

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

2.2 K Nearest Neighbors (KNN)

```python
class KNearestNeighbors(Classifier):
    def __init__(self, k=5):
        self.k = k
        self.model = None
        self.p = 2

    def fit(self, X, y):
        data = pd.concat([X, y], axis=1)
        self.model = data
```

前置作業: KNN 就比Bayes簡單了，因為它基本上就是在資料集中創造一個feature space 用來計算特徵向量之間的距離就好，所以我就用一個陣列去將空間創造出來。其中P為我要計算距離的次方變數。(下頁會詳細說明)

```python
def predict(self, X):
    y = []
    for index, row in X.iterrows():
        minkowski_dis = np.power(np.sum(np.power(np.abs(self.model.iloc[:, :77] - row), self.p), axis=1), 1/self.p)
        sorted_indices = np.argsort(minkowski_dis).tolist() # You need to turn it into a original array

        true_num, false_num = 0, 0
        for candidate in range(self.k):
            if self.model.iloc[sorted_indices[candidate], -1] == 1:  true_num += 1
            else:  false_num += 1

        if true_num > false_num:
            y.append(np.int64(1))
        else:
            y.append(np.int64(0))

    return y
```

接著就是尋歷整個feature space，並計算要predict 資料與每一特徵向量點的距離。我採用的是距離公式為minkowski。最後，再由最近的K個data 去投票，預測答案為true 或 false。

```
def predict_proba(self, X):
    # Implement probability estimation for KNN
    prob = []
    for index, row in X.iterrows():
        minkowski_dis = np.power(np.sum(np.power(np.abs(self.model.iloc[:, :77] - row), self.p), axis=1), 1/self.p)
        sorted_indices = np.argsort(minkowski_dis).tolist() # You need to turn it into a original array

        true_num, false_num = 0, 0
        for candidate in range(self.k):
            if self.model.iloc[sorted_indices[candidate], -1] == 1:  true_num += 1
            else:  false_num += 1

        if true_num > false_num:
            prob.append([(true_num/self.k), (false_num/self.k)])
        else:
            prob.append([(false_num/self.k), (true_num/self.k)])

    prob = np.array(prob)
    return prob
```

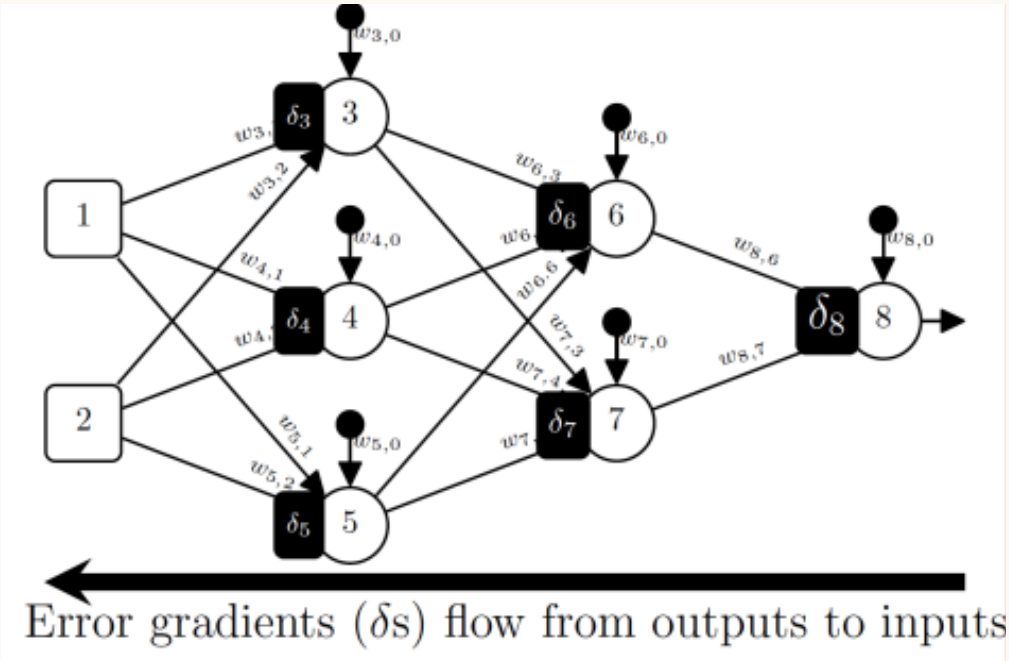
機率的部分就利用k票去計算。(true false 的得票百分比)

2.3 MultilayerPerceptron (MLP)

這邊我實作了一個可以有任意層的layer(希望助教可以幫我多加一些分數)。我預設的layer 數目則是我實作下來所得到最好的結果。(在下部份的評測可以得到)
不僅如此，我還多加了一個方法去初始化權重 ，來增加模型的準確度。我會在最後一個Disscussions 的部分做詳細補充。(註解起來的程式碼是一般的初始化權重方法)

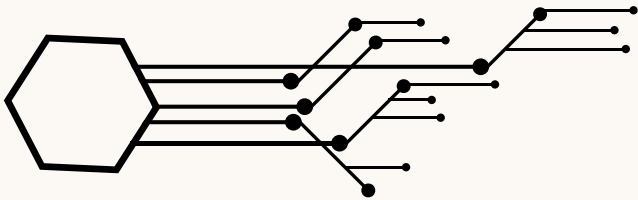
```
# Output size in this task is always 1
class MultilayerPerceptron(Classifier):
    def __init__(self, input_size, hidden_layers_sizes, output_size):
        # Initialize MLP with given network structure
        self.hidden_layers_sizes = hidden_layers_sizes
        # Normal way to initialize weights (Cause tremendous total error)
        '''
        # self.weights = [np.random.rand(input_size, hidden_layers_sizes[0])] # matrix dimension: inputsize * hidden_layer[0]
        # self.weights += [np.random.rand(hidden_layers_sizes[i], hidden_layers_sizes[i+1]) for i in range(len(hidden_layers_sizes)-1)]
        # self.weights += [np.random.rand(hidden_layers_sizes[-1], output_size)]
        '''
        # Using Xavier/Glorot initialization to initialize weights
        self.weights = [np.random.randn(input_size, hidden_layers_sizes[0]) / np.sqrt(input_size)]
        self.weights += [np.random.randn(hidden_layers_sizes[i], hidden_layers_sizes[i+1]) / np.sqrt(hidden_layers_sizes[i]) for i in range(len(hidden_layers_sizes)-1)]
        self.weights += [np.random.randn(hidden_layers_sizes[-1], output_size) / np.sqrt(hidden_layers_sizes[-1])]

        self.biases = [np.ones(hidden_size) for hidden_size in hidden_layers_sizes] # Every neuron has one bias
        self.biases += [np.ones(output_size)] # Output size in this task is always one
        self.delta = [np.random.rand(hidden_layers_sizes[i]) for i in range(len(hidden_layers_sizes))] #The val will be flush during backward_prop
        self.delta += [np.random.rand(output_size)] # Output size in this task is always one
        self.inner_output = [np.random.rand(hidden_layers_sizes[i]) for i in range(len(hidden_layers_sizes))] #The val will be flush during forward_prop
        self.inner_output += [np.random.rand(1)] # This val is equal to the final Ouput

        self.learning_rate = 0.03
        self.epochs = 2000
```

參考實作圖:



參數介紹:

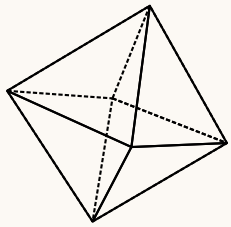
1. weights(權重): 一個array of 二維array。例如: self.weights[0]除存input layer 到第一層的所有參數。
2. delta: 一個array of 一維array。用來做back propagation。例如: self.delta[0] = 上圖的delta 3、4、5。
3. biases: 一個array of 一維array。用來加入dummy input。例如: self.biases[0] = 上圖的weighs (3,0)、(4,0)、(5,0)。
4. inner_output: 一個array of 一維array。用來每一次每一個nueron 的output，用來做back propagation。x 例如: inner_output[0] = neuron 3、4、5 做完 activation function 的輸出。
5. learning rate: back propagation 時調整每次 weights 的大小
6. epochs: 總共要訓練幾次。



```python
def _forward_propagation(self, X):
    # Implement forward propagation for MLP
    layer_output = X
    for i in range(len(self.weights)):
        weighted_sum = np.dot(layer_output, self.weights[i]) + self.biases[i] # Notice the size of the weights
        layer_output = self.sigmoid(weighted_sum)
        self.inner_output[i] = layer_output

    return layer_output
```

```python
def _backward_propagation(self, input, output, target):
    # Implement backward propagation for MLP
    # Ouput Unit Weights
    error = target - output
    self.delta[len(self.weights)-1] = self.sigmoid(output, derivative=True) * error # ouput == self.inner_output[len(self.weights)-1]
    self.weights[len(self.weights)-1] += self.learning_rate * (self.inner_output[len(self.weights)-2][:, np.newaxis] * self.delta[len(self.weights)-1][np.newaxis, :])

    # Hidden Unit Weights
    for layer in range(len(self.weights) - 2, 0, -1):
        self.delta[layer] = np.multiply(self.sigmoid(self.inner_output[layer], derivative=True), np.dot(self.weights[layer + 1], self.delta[layer + 1]))
        self.weights[layer] += self.learning_rate * (self.inner_output[layer - 1][:, np.newaxis] * self.delta[layer][np.newaxis, :])


    # First layer of the hidden layer
    self.delta[0] = np.multiply(self.sigmoid(self.inner_output[0], derivative=True), np.dot(self.weights[1], self.delta[1]))
    self.weights[0] += self.learning_rate * (np.array(input)[:, np.newaxis] * self.delta[0][np.newaxis, :])


    # Update biases
    for i in range(len(self.biases)):
        self.biases[i] += self.learning_rate * self.delta[i]
```

再來就是本作業最難的部分。在back propagation 中，你必須要清楚了解公式的計算方法，才有辦法做出來。
至於我如何實作的? 就是先假設一個簡單的神經網路模型，再將下面的公式寫成矩陣，實際手算一遍之後，在將他寫成程式碼。我假設的層數和神經元個數就是和參考實作圖是一樣的。
中間的過程非常繁瑣，花了我許多時間。



公式: (節錄自課本)

output layer 的 delta

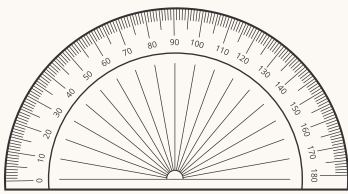
$$\delta_j = -\frac{\partial E_d}{\partial net_j} \qquad \frac{\partial E_d}{\partial net_j} = -(t_j - o_j) \cdot o_j \cdot (1 - o_j)$$

input layer 的 delta

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

透過delta來去更新weights

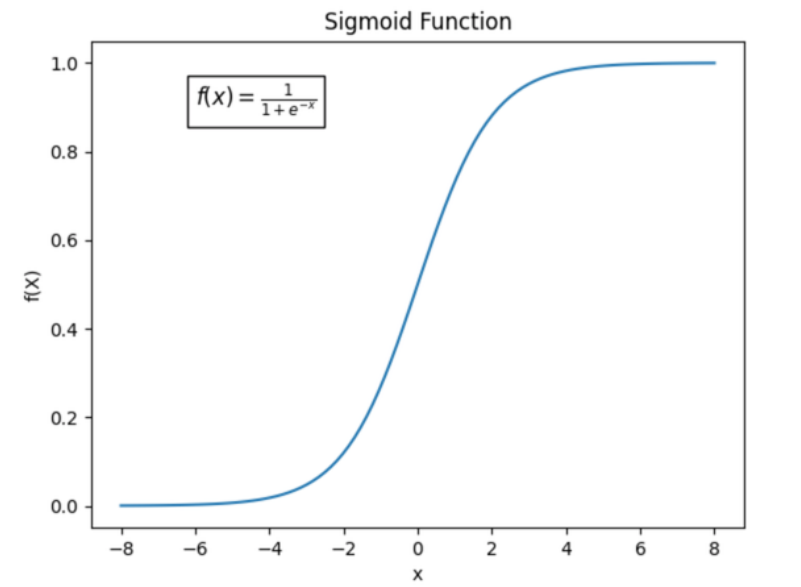
$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = -\eta \cdot \frac{\partial E_d}{\partial net_j} \cdot x_{ji} = \eta \cdot \delta_j \cdot x_{ji}$$



Activation function: Sigmoid

```python
# Activation Funciton
def sigmoid(self, x, derivative=False):
    if derivative:
        return x * (1 - x)
    return 1 / (1 + np.exp(-x))
```

最後的預測結果，就是最後一個layer的神經元輸出。因為sigmoid 函數輸出值會介於0和1，所以我們能說其值是機率(符合機率的公設)。若大於0.5則判斷會得到candidemia，反之。



程式在執行的過程中，我會不斷的印出他的loss，來看他是否有遞減成功 (從圖中可以發現error慢慢變小)

```
Epoch 1164/1500, Error: 8.256934677946255
Epoch 1165/1500, Error: 8.25681541707112
Epoch 1166/1500, Error: 8.25669628655212
Epoch 1167/1500, Error: 8.256577286141724
Epoch 1168/1500, Error: 8.256458415593105
Epoch 1169/1500, Error: 8.256339674660122
Epoch 1170/1500, Error: 8.256221063097339
Epoch 1171/1500, Error: 8.256102580660018
Epoch 1172/1500, Error: 8.255984227104092
Epoch 1173/1500, Error: 8.255866002186217
Epoch 1174/1500, Error: 8.2557479056637
Epoch 1175/1500, Error: 8.255629937294556
Epoch 1176/1500, Error: 8.255512096837464
Epoch 1177/1500, Error: 8.25539438405179
Epoch 1178/1500, Error: 8.255276798697585
```

```
Epoch 941/1500, Error: 0.0015231362922982327
Epoch 942/1500, Error: 0.0015228503925590728
Epoch 943/1500, Error: 0.0015225647122075218
Epoch 944/1500, Error: 0.001522279250785697
Epoch 945/1500, Error: 0.001521994007837297
Epoch 946/1500, Error: 0.0015217089829076657
Epoch 947/1500, Error: 0.0015214241755437006
Epoch 948/1500, Error: 0.0015211395852938942
Epoch 949/1500, Error: 0.0015208552117083325
Epoch 950/1500, Error: 0.001520571054338642
Epoch 951/1500, Error: 0.0015202871127380309
Epoch 952/1500, Error: 0.0015200033864612565
```

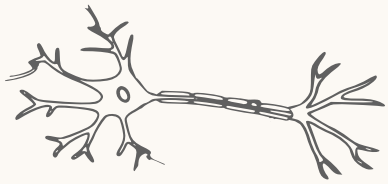
可以看到我的模型可以將error 降低到非常非常小的

2.4 Single Neuron Perceptron (SNP)

我會多加這個模型的原因非常簡單，因為我是先完成了單一個神經元(可以說該神經網路只有一層 layer)，在延伸其概念到multiple layer。所以若剛剛2.3的部分有理解我的程式碼的話，這邊會非常簡單。不僅如此，我也想比較一下，Single Neuron Perceptron 預測的結果會不會和Multiple layer 差異非常多。到底哪一個會有較好的performance。

```python
def _forward_propagation(self, X):
    weighted_sum = np.dot(X, self.weights) + self.bias
    output = self.sigmoid(weighted_sum)
    return output

def _backward_propagation(self, input, output, target):
    error = target - output
    gradient = error * self.sigmoid(output, derivative = True)
    self.weights += self.learning_rate * gradient * np.array(input)
    self.bias += self.learning_rate * gradient
```

作法的話簡單很多，且非常類似。forward propagation 就是做一次矩陣乘法，然後丟到activation function 即可。而 back propagation就直接省去hidden layer 的運算就好 。



3. Models Evaluation

給定的評測模型參數

```
Cross-validation results:
                model  accuracy        f1  precision    recall       mcc       auc
```

- 針對Bayes 去做評測與討論

```
Naive Bayes Average  0.717698  0.097479   0.350000  0.058298  0.085570  0.536538
```

在此次評測中，我只有列舉出一個結果，是因為Bayes並沒有參數可以去調控。所以輸出會是一個定值。(在training set不變的情況底下)
不過，因為極度容易受到training set 的情況波動，在Discussions 的部分，我們會探討到資料前處理的位置，會發現Bayes 會受到很大的影響。
準確率偏低的原因，我猜測是因為我假定各個特徵間都是相互獨立的，雖然可以降低我程式的負擔，不過也減低了他的準確率。

- 針對SNP和MLP去做評測與討論: (兩個訓練參數皆相同)
  - MLP layer:

```
'MLP': MultilayerPerceptron(77, (8, 4), 1)
```

  - Epoch = 1500 learning rate = 0.1

```
        KNN Average  0.748492  0.408020   0.631746  0.331489  0.315673  0.653853
        MLP Average  0.930397  0.854365   0.891667  0.828680  0.813526  0.777778
Naive Bayes Average  0.717698  0.097479   0.350000  0.058298  0.085570  0.536538
        SNP Average  0.857857  0.714515   0.783750  0.676390  0.632344  0.638213
```

  - Epoch = 1500 learning rate = 0.2

```
        KNN Average  0.748492  0.408020   0.631746  0.331489  0.315673  0.653853
        MLP Average  0.936111  0.858027   0.885552  0.838485  0.820275  0.820113
Naive Bayes Average  0.717698  0.097479   0.350000  0.058298  0.085570  0.536538
        SNP Average  0.860635  0.715635   0.772523  0.679226  0.630581  0.620097
```

  - Epoch = 1500 learning rate = 0.23

```
        KNN Average  0.748492  0.408020   0.631746  0.331489  0.315673  0.653853
        MLP Average  0.933175  0.858679   0.876905  0.847013  0.817115  0.620746
Naive Bayes Average  0.717698  0.097479   0.350000  0.058298  0.085570  0.536538
        SNP Average  0.863333  0.728795   0.809354  0.676239  0.649180  0.640378
```

- MLP layer:

```
'MLP': MultilayerPerceptron(77, (32, 16, 4), 1)
```

- Epoch = 1500 learning rate = 0.1

KNN Average	0.748651	0.292195	0.750000	0.191091	0.285291	0.787550
MLP Average	0.952778	0.891703	0.896364	0.896104	0.865151	0.537452
Naive Bayes Average	0.717698	0.097479	0.350000	0.058298	0.085570	0.536538
SNP Average	0.857857	0.714515	0.783750	0.676390	0.632344	0.639216

- Epoch = 1500 learning rate = 0.2

KNN Average	0.748651	0.292195	0.750000	0.191091	0.285291	0.787550
MLP Average	0.933175	0.849912	0.887500	0.828225	0.813326	0.344448
Naive Bayes Average	0.717698	0.097479	0.350000	0.058298	0.085570	0.536538
SNP Average	0.860635	0.713254	0.779190	0.670135	0.629744	0.622762

- Epoch = 1500 learning rate = 0.23

40	KNN Average	0.748492	0.408020	0.631746	0.331489	0.315673	0.653853
41	MLP Average	0.927778	0.853625	0.884227	0.834356	0.810743	0.755973
42	Naive Bayes Average	0.717698	0.097479	0.350000	0.058298	0.085570	0.536538
43	SNP Average	0.863333	0.728795	0.809354	0.676239	0.649180	0.642846

- 針對KNN去做評測與討論:
  - p = 1 (Manhattan Distance) k = 5

KNN Average	0.768095	0.476171	0.689603	0.384908	0.385178	0.653853

  - p = 2 (Euclidean Distance) k = 5

KNN Average	0.748492	0.408020	0.631746	0.331489	0.315673	0.653853

  - p = 3 k = 5

KNN Average	0.748492	0.405964	0.649603	0.331489	0.322093	0.658723

  - p = 1 (Manhattan Distance) k = 7

KNN Average	0.765476	0.432315	0.710000	0.332173	0.364200	0.693329

  - p = 2 (Euclidean Distance) k = 7

KNN Average	0.773889	0.448584	0.704762	0.344554	0.377169	0.695280

- p = 3 k = 7

```
KNN Average  0.776587  0.449995  0.742262  0.345463  0.391824  0.695720
```

- p = 1 (Manhattan Distance) k = 9

```
KNN Average  0.773889  0.419109  0.706667  0.315909  0.364386  0.748053
```

- p = 2 (Euclidean Distance) k = 9

```
KNN Average  0.773889  0.421119  0.636667  0.319437  0.350890  0.726887
```

- p = 3 k = 9

```
KNN Average  0.776667  0.440755  0.663333  0.337771  0.368054  0.735909
```

- p = 1 (Manhattan Distance) k = 11

```
KNN Average  0.776746  0.407850  0.758333  0.291104  0.385368  0.770505
```

- p = 2 (Euclidean Distance) k = 11

```
KNN Average  0.779444  0.396435  0.808333  0.278485  0.393264  0.776893
```

- p = 3 k = 11

```
KNN Average  0.779444  0.411582  0.800000  0.296667  0.388377  0.762404
```

- p = 1 (Manhattan Distance) k = 15

```
KNN Average  0.757063  0.329308  0.858333  0.216848  0.336926  0.781436
```

- p = 2 (Euclidean Distance) k = 15

```
KNN Average  0.757063  0.329308  0.858333  0.216848  0.336926  0.781436
```

為什麼K的值增大到一定常數後會導致正確率下降?
我的猜測是因為過大的K值會導致模型去讓更多無關的資料影響決策。
為什麼不要一直增加計算距離所用的P值?
我猜想他會大幅減小正確率，因為在計算的過程中，會有更多的捨棄(truncation)或四捨五入發生。

4. Disscussions

4.1 Implementation challenges

4.1.1 Initialization weights

在做MLP 的時候，我的error 起初都沒有降低的很明顯，而且還非常的顯著。模型的預測結果甚至比single nueron perceptron時還要糟糕許多。我一度以為我的程式碼哪裡出問題，甚至下課也請教老師我的問題。
在老師猜測可能問題之後，我上網找到了解決方法，Xavier initializaiton。將此方法用在我的程式後，就有了大幅的改善，可以非常準確的預測。輕而易舉的就比 single nueron perceptron還要精準(訓練batch數目相同為前提)。
相比之下，single nueron perceptron不需要做xavier initialization就可以非常準確的原因，我認為是因為weights數目沒有很多，影響不大。只要經過多次一點的訓練，就可以輕易找到極小值得error。

Xavier initializaiton

權重的生成會影響神經元的輸出太集中或是過於飽和(當 weights 的標準差太小，造成輸出太集中0，反之)。
Xavier Initialization 是希望神經元輸入和輸出的變異數保持一致。除此之外，不同的初始化權重方法也和激勵函數有很大的關係。因此，我選擇了 Xavier initialization可以有效搭配sigmoid 和 tanh 損失函數。

公式: (節錄自課本)

Simplified Xavier initialization, $W^k \sim N(0, \sqrt{\frac{1}{n_{in}^k}})$, for logistic activation.

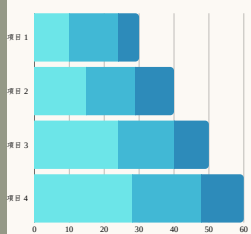
4.1.2 Where to preprocess your data?

在有kfold 的情況下，資料要在哪裡做前處理比較好? 要在kfold 前面嗎? 還是說要在kfold 將資料分成validation dataset 和 training dataset後? 這個問題起初我沒有仔細去思考，所以也沒有察覺出助教的程式碼有問題。
然而，當助教發公告後，經過百般折騰，才慢慢發現這中間有偌大的不同!
造成問題的主要地方就是imputation部分。

- 將資料前處理放置在kfold 前面: 則validation dataset 會被training dataset 給影響到。因為他在計算衆數和平均數時，是有將 training dataset 的資料一起下去做平均。如此一來，模型就會"偷看"到最終的validation set 進而影響到結果。
- 將資料前處理放置在kfold 後面: 則validation dataset 和 training dataset 裡不見的資料就是完全由各自補齊的。

雖然這個做法會降低模型的訓練正確度，卻可以增加模型的泛化程度，讓模型變得更能適應新的資料。

```
X_train = df.drop('Outcome', axis=1)
y_train = df['Outcome']

# Perform K-Fold cross-validation
kf = KFold(n_splits=10, random_state=42, shuffle=True)
cv_results = []

for name, model in models.items():
    for fold_idx, (train_index, val_index) in enumerate(kf.split(X_train), start=1):
        # Slightly modify here
        X_train_fold, X_val_fold = X_train.iloc[train_index], X_train.iloc[val_index]
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.iloc[val_index]

        # We should do preprocess here
        processed = Preprocessor(X_train_fold)
        X_train_fold = processed.preprocess()
        processed = Preprocessor(X_val_fold)
        X_val_fold = processed.preprocess()

        model.fit(X_train_fold, y_train_fold)
        fold_result = evaluate_model(model, X_val_fold, y_val_fold)
        fold_result['model'] = name
        fold_result['fold'] = fold_idx
        cv_results.append(fold_result)
```



資料前處理在Kfold 前面:

KNN Average	0.759603	0.444604	0.732143	0.356380	0.375783	0.633343
MLP Average	0.946984	0.893064	0.883766	0.906147	0.858553	0.690623
Naive Bayes Average	0.813175	0.702357	0.651513	0.797231	0.587489	0.499852
SNP Average	0.838175	0.709298	0.726424	0.712145	0.606430	0.600244

KNN Average	0.759603	0.444604	0.732143	0.356380	0.375783	0.633343
MLP Average	0.961111	0.911443	0.898647	0.928571	0.888157	0.564638
Naive Bayes Average	0.813175	0.702357	0.651513	0.797231	0.587489	0.499852
SNP Average	0.863571	0.750483	0.777933	0.742145	0.665515	0.629331

KNN Average	0.759603	0.444604	0.732143	0.356380	0.375783	0.633343
MLP Average	0.947143	0.888989	0.883409	0.898528	0.855961	0.747879
Naive Bayes Average	0.813175	0.702357	0.651513	0.797231	0.587489	0.499852
SNP Average	0.899762	0.806544	0.837521	0.795075	0.746437	0.588910

資料前處理在Kfold 後面和建模前面:

KNN Average	0.748492	0.408020	0.631746	0.331489	0.315673	0.653853
MLP Average	0.936111	0.858027	0.885552	0.838485	0.820275	0.820113
Naive Bayes Average	0.717698	0.097479	0.350000	0.058298	0.085570	0.536538
SNP Average	0.860635	0.715635	0.772523	0.679226	0.630581	0.620097

KNN Average	0.748492	0.408020	0.631746	0.331489	0.315673	0.653853
MLP Average	0.930397	0.854365	0.891667	0.828680	0.813526	0.777778
Naive Bayes Average	0.717698	0.097479	0.350000	0.058298	0.085570	0.536538
SNP Average	0.857857	0.714515	0.783750	0.676390	0.632344	0.638213

KNN Average	0.748492	0.408020	0.631746	0.331489	0.315673	0.653853
MLP Average	0.933175	0.858679	0.876905	0.847013	0.817115	0.620746
Naive Bayes Average	0.717698	0.097479	0.350000	0.058298	0.085570	0.536538
SNP Average	0.863333	0.728795	0.809354	0.676239	0.649180	0.640378

經過好幾次的測試，可以發現在若在kfold 前面做資料前處理，對於預測結果來說會有很大的影響。

- 對於MLP來說，正確率能達到96%。然而，若不讓模型"偷看"結果，只能達到94%。
- 對於bayes來說，影響最劇烈，正確率從81%直接掉到71%。我猜是因為對於計算機率的演算法，對於平均數的設置都會非常的敏感。若模型"偷看"到結果，就可以大幅提升準確度。

最後，希望可以提升模型的泛化能力，能在test dataset有好一點的表現。

4.1.3 Model with many Neurons

Multiple layers 中，我發現增加很多神經元並不會有效提升準確率，反而是和訓練的batch數目和learning rate較有關係。我想會有這樣的原因是因為訓練模型最主要就是要想辦法把error 降低，而多次的訓練次數能有助於幫助權重找到誤差最小值。而許多神經元反而會讓模型變得非常複雜，不僅計算時間拉長，層數過多的話，也有可能會有weights vanishing 發生。

4.2 Model Comparison

在不同的應用場景和數據特性可能會適合不同種的模型。以下是這次作業3個模型的優點。

- 神經網絡通常用於複雜的非線性問題。
- Bayes分類器適用於簡單的分類任務
- KNN在鄰近性方面則表現出色。

結論:
在這次的作業中，我們觀察到神經網路模型表現最佳。由於資料擁有77個不同的特徵，這對於適用於非線性複雜模型的情境而言是相當適合的。相對而言，Bayes分類器可能顯得力不從心。至於KNN模型，我猜測是因為資料規模不夠龐大，才難以精確預測。
這一觀察也在此次結果中得到了印證，神經網路模型呈現出最高的準確率。

4.3 Reflection

這次的作業學到許多知識，也更徹底了解各種不同的模型實作。當自己實際手刻出三個模型時，我覺得非常有成就感。資料前處理的部分也讓我思考許久該如何提高準確率。我覺得這次的作業難易度頗有挑戰性，非常推薦給想要把機器學習基礎打穩的同學修課。