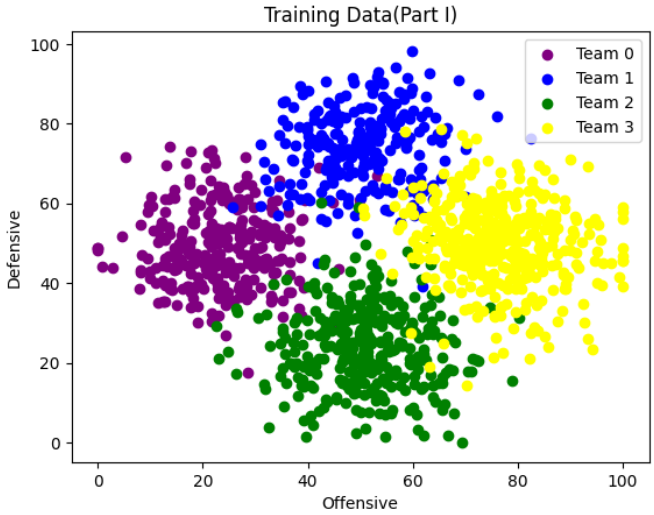
ML HW2 Report

110511277 蔡東宏

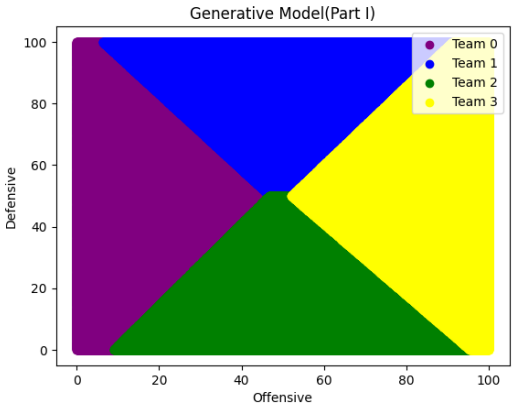
1. Part I

Training Data 分布圖



A. Generative Model:

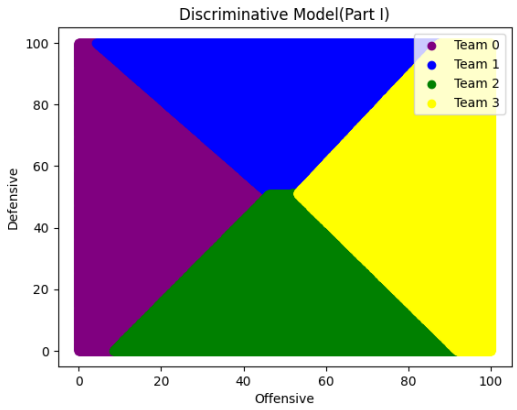
首先，透過 Generative Model 去進行 classification 的問題，先根據 Training Data 分別算出 4 個 class 的 mean vector 以及 covariance matrix，接下來根據權重得到一個共同的 covariance matrix，因為 4 個類別共用同一個 covariance matrix，因此最後得到的 boundary 為直線，接下來根據 4 個類別的 mean vector 以及共用的 1 個 covariance matrix 可以得到 4 個類別的 $P(C_K, x)$的 Model 並且可以進一步推出 $P(C_K \mid x)$，最後在將(x_1, x_2)代入四個 Model，並比較機率值決定屬於哪一個類別。觀察 Training data 以及 Testing data 的 accuracy，分別為 0.943 以及 0.912。




```
Training Data(Generative Model Part I) ACC: 0.943076923076923
Training Data(Generative Model Part I) Confusing Matrix:
 [[287.   6.   7.   0.]
 [  9. 229.   0.  12.]
 [ 10.   2. 329.   9.]
 [  0.  12.   7. 381.]]
Testing Data(Generative Model Part I) ACC: 0.912
Testing Data(Generative Model Part I) Confusing Matrix:
 [[187.  10.   3.   0.]
 [  6. 271.   1.  22.]
 [  8.   0. 135.   7.]
 [  0.   3.   6.  91.]]
```

B. Discriminative Model:

接下來，透過 Discriminative Model 去進行 classification 的問題，因為多類別分類的問題是 discriminative 假設 logistic sigmoid function，因此最佳的 w 沒有一個 closed-form 的解，需要用迭代的方式去找出最佳解，所以我使用了 Newton-Raphson($w^{(new)} = w^{(old)} - H^{-1}\nabla E(w)$, where $H = \nabla\nabla E(w)$) 的方法取找出最佳的 w，接下來根據最後得到的 w 去建立 4 個類別的 $P(C_K \mid x)$的 Model，最後在將(x_1, x_2)代入四個 Model，並比較機率值決定屬於哪一個類別。觀察 Training data 以及 Testing data 的 accuracy，分別為 0.942 以及 0.909。



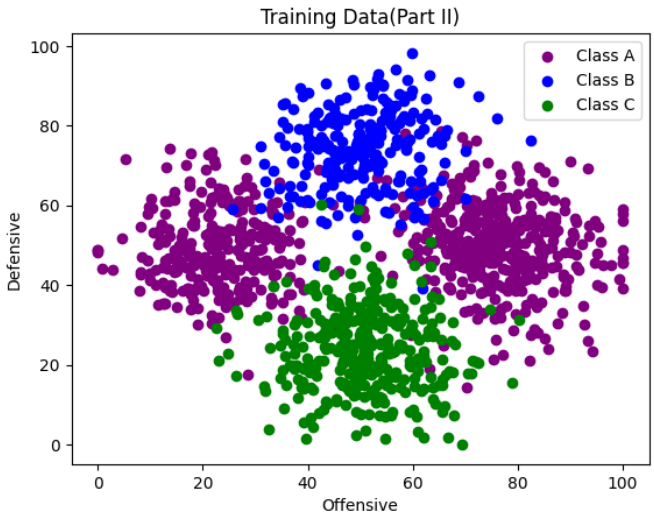

```
Training Data(Discriminative Part I) ACC: 0.9423076923076923
Training Data(Discriminative Part I) Confusing Matrix:
 [[285.   6.   9.   0.]
 [  9. 226.   1.  14.]
 [  9.   2. 331.   8.]
 [  0.  10.   7. 383.]]
Testing Data(Discriminative Part I) ACC: 0.9093333333333333
Testing Data(Discriminative Part I) Confusing Matrix:
 [[185.  10.   5.   0.]
 [  6. 267.   1.  26.]
 [  7.   0. 138.   5.]
 [  0.   2.   6.  92.]]
```

2. Part II

Training Data 分布圖

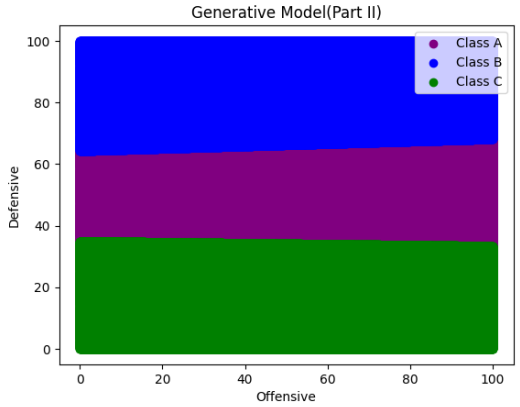
(Team 0 and Team 3 to be class A,

Team 1 to be class B, Team 2 to be class C)



A. Generative Model:

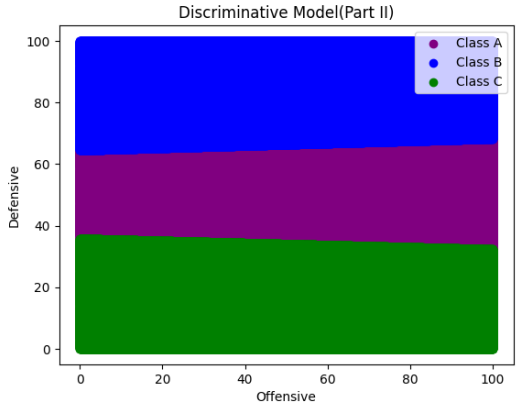
透過與 Part I 相同的 Generative Model 實做出來，因為 covariance 使用同一個，因此 boundary 為直線，觀察 Training data 以及 Testing data 的 accuracy，分別為 0.851 以及 0.841，accuracy 相較於 part I 變低。




```
Training Data(Generative Model Part II) ACC: 0.8507692307692307
Training Data(Generative Model Part II) Confusing Matrix:
 [[618.  37.  45.]
 [ 54. 196.   0.]
 [ 58.   0. 292.]]
Testing Data(Generative Model Part II) ACC: 0.8413333333333334
Testing Data(Generative Model Part II) Confusing Matrix:
 [[269.  15.  16.]
 [ 58. 242.   0.]
 [ 30.   0. 120.]]
```

B. Discriminative Model:

透過與 Part I 相同的 Discriminative Model 方法實做出來，因為選定的 basis function 為一階，因此 boundary 為直線，觀察 Training data 以及 Testing data 的 accuracy，分別為 0.853 以及 0.842，accuracy 相較於 part I 變低，而與 part II 的 Generative Model 大致相同。




```
Training Data(Discriminative Model Part II) ACC: 0.8530769230769231
Training Data(Discriminative Model Part II) Confusing Matrix:
 [[622.  35.  43.]
 [ 54. 196.   0.]
 [ 59.   0. 291.]]
Testing Data(Discriminative Model Part II) ACC: 0.8426666666666667
Testing Data(Discriminative Model Part II) Confusing Matrix:
 [[270.  14.  16.]
 [ 58. 242.   0.]
 [ 30.   0. 120.]]
```

Discussion:

1. **What is the difference between the generative model and the discriminative model?**

   Generative Model 的作法是假設各類別的資料是某種分布(此題假設是 Normal Distribution)，因此透過 training data 分別找出不同的 mean vector 以及 covariance matrix，如此一來就能夠找出 $P(x, C_k)$，進一步推導出 $P(C_k|x)$，Generative Model 的缺點是需先知道資料的分布情形，否則無法建立出有效的 model；相反地，Discriminative Model 的作法不需要假設資料是哪種分布，透過 training data 便能有效的找出最佳的 weight，並推導出 $P(C_k|x)$。

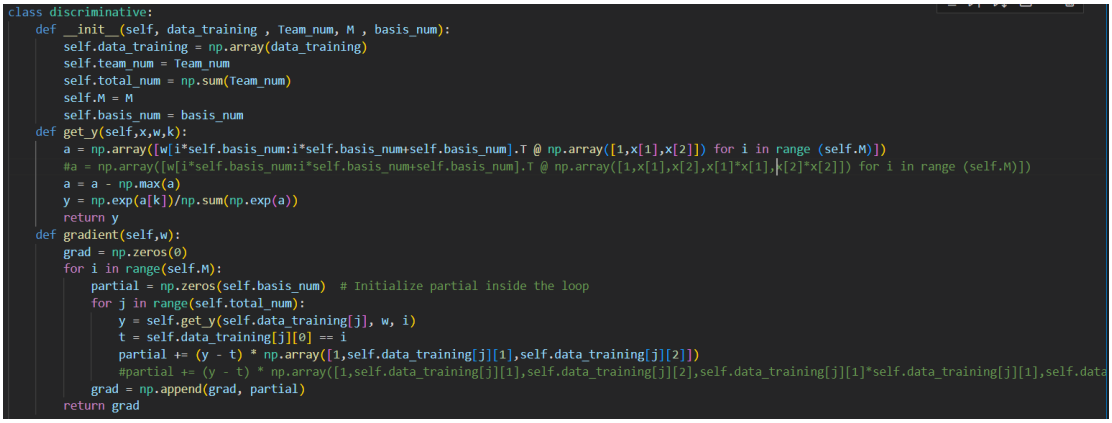
   在實作上，Discriminative Model 的方法相對複雜，因為需要透過一次次迭代去尋找最佳的 w，在我採用的 Newton-Raphson 中，每一次迭代都需要找到▽E(w)以及▽▽E(w)，計算較為複雜且執行時間較久，除此之外，basis 的選定也會影響訓練出來的結果。 最後觀察實作出來的結果會發現兩種模型建出來的 decision boundary 以及 accuracy 都差不多，兩種模型都有很高的 accuracy，然而在程式運行的速度上，Discriminative Model 相對於 Generative Model 慢非常多。

2. **How do you implement the code of the generative model and the discriminative model?**

   Generative Model: 首先先透過 get_w_w0 得到四個類別的 w 以及 w0，如此一來便能將 generative model 的 $P(C_K, x)$建立出來，接下來的 get_test 能夠將前面得到的 w、w0 以及想要測試的 x vector 算出 4 個類別的 $P(C_K|x)$，最後再根據哪個 $P(C_K|x)$比較大去決定屬於哪個類別，最後的 ACC_confuse 是用來得到 training data 以及 testing data 的 accuracy 以及 confusing matrix。

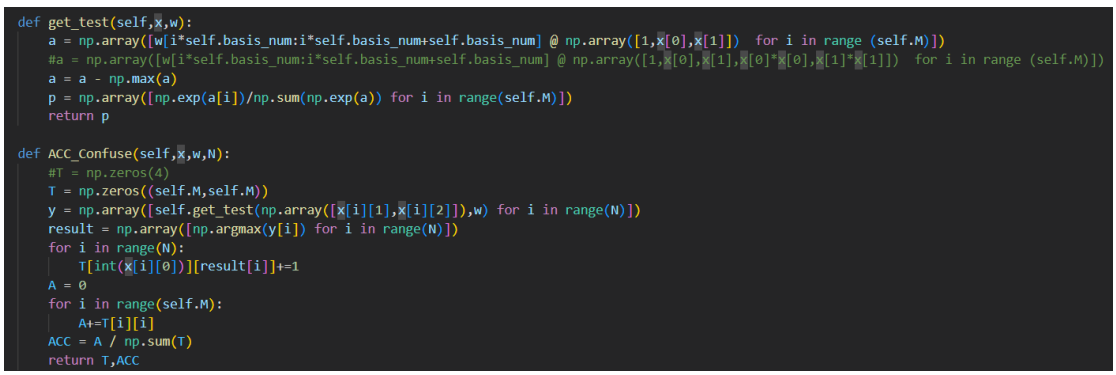
```python
class Generative:
    def __init__(self, data_training , Team_num , M):
        self.data_training = data_training
        self.team_num = Team_num
        self.total_num = np.sum(Team_num)
        self.M = M
    def get_w_w0(self):
        cov_matrix = np.zeros((self.M,2,2))
        mean_vec = np.zeros((self.M,2,))
        cov = np.zeros((2,2))
        for i in range(self.M):
            data = np.array(self.data_training[self.data_training["Team"] == i][["Offensive", "Defensive"]].T)
            cov_matrix[i] = np.cov(data ,rowvar=True)
            mean_vec[i] = np.mean(data ,axis = 1)
            cov += self.team_num[i] * cov_matrix[i] / self.total_num
        w = np.array([np.linalg.inv(cov) @ mean_vec[i] for i in range(self.M)])
        w0 = np.array([-0.5*mean_vec[i].T @ np.linalg.inv(cov) @ mean_vec[i] + np.log(self.team_num[i] / self.total_num) for i in range(self.M)])
        return w,w0
    def get_test(self,x,w,w0):
        a = np.array([w[i].T @ x + w0[i] for i in range (self.M)])
        p = np.array([np.exp(a[i])/np.sum(np.exp(a)) for i in range(self.M)])
        return p
    def ACC_Confuse(self,x,w,w0,N):
        #T = np.zeros(4)
        T = np.zeros((self.M,self.M))
        y = np.array([self.get_test(np.array([x[i][1],x[i][2]]),w,w0) for i in range(N)])
        result = np.array([np.argmax(y[i]) for i in range(N)])
        for i in range(N):
            T[int(x[i][0])][result[i]]+=1
        A = 0
        for i in range(self.M):
            A+=T[i][i]
        ACC = A / np.sum(T)
        return T,ACC
```

Discriminative Model: Discriminative 中，我採用的收斂方法是 Newton-Raphson，首先，我先利用 newton 得到 weight，過程中會用 error 去決定是否已經收斂，若收斂就跳出迴圈，gradient 以及 gradi_gradi 分別用來得到▽E(w)以及▽▽E(w)，接下來的 get_test 能夠將前面得到的 w 以及想要測試的 x vector 算出 4 個類別的 $P(C_K \mid x)$，最後再根據哪個 $P(C_K \mid x)$比較大去決定屬於哪個類別，最後的 ACC_confuse 是用來得到 training data 以及 testing data 的 accuracy 以及 confusing matrix。(code 中的 iteration 是 gradient descent 的方法)



```
class discriminative:
    def __init__(self, data_training , Team_num, M , basis_num):
        self.data_training = np.array(data_training)
        self.team_num = Team_num
        self.total_num = np.sum(Team_num)
        self.M = M
        self.basis_num = basis_num
    def get_y(self,x,w,k):
        a = np.array([w[i*self.basis_num:i*self.basis_num+self.basis_num].T @ np.array([1,x[1],x[2]]) for i in range (self.M)])
        #a = np.array([w[i*self.basis_num:i*self.basis_num+self.basis_num].T @ np.array([1,x[1],x[2],x[1]*x[1],x[2]*x[2]]) for i in range (self.M)])
        a = a - np.max(a)
        y = np.exp(a[k])/np.sum(np.exp(a))
        return y
    def gradient(self,w):
        grad = np.zeros(0)
        for i in range(self.M):
            partial = np.zeros(self.basis_num)  # Initialize partial inside the loop
            for j in range(self.total_num):
                y = self.get_y(self.data_training[j], w, i)
                t = self.data_training[j][0] == i
                partial += (y - t) * np.array([1,self.data_training[j][1],self.data_training[j][2]])
                #partial += (y - t) * np.array([1,self.data_training[j][1],self.data_training[j][2],self.data_training[j][1]*self.data_training[j][1],self.data
            grad = np.append(grad, partial)
        return grad
```

```
def gradi_gradi(self,w):
    grad = np.zeros([])
    for k in range(self.M):
        stacked_array = np.zeros([])
        for j in range(self.M):
            partial = np.zeros((self.basis_num,self.basis_num))
            for i in range(self.total_num):
                yk= self.get_y(self.data_training[i], w, k)
                yj= self.get_y(self.data_training[i], w, j)
                phi = np.array([1,self.data_training[i][1],self.data_training[i][2]])
                #phi = np.array([1,self.data_training[i][1],self.data_training[i][2],self.data_training[i][1]*self.data_training[i][1],self.data_training[i
                phi = phi.reshape(self.basis_num,1)
                partial += yk * ((k==j) - yj) * phi @ phi.T
            if j == 0 :
                stacked_array = partial
            else :
                stacked_array = np.hstack((stacked_array, partial))
        if k == 0 :
            grad = stacked_array
        else :
            grad = np.vstack((grad, stacked_array))
    #print(grad)
    return grad
```



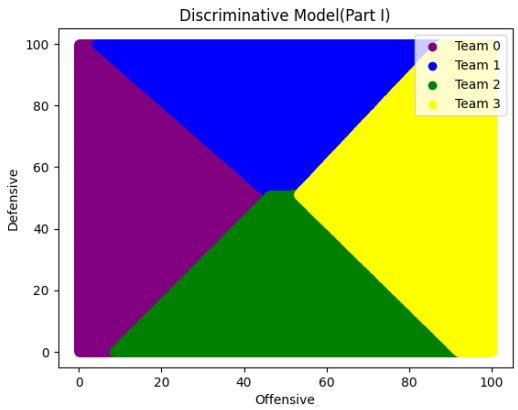
```
def get_test(self,x,w):
    a = np.array([w[i*self.basis_num:i*self.basis_num+self.basis_num] @ np.array([1,x[0],x[1]])  for i in range (self.M)])
    #a = np.array([w[i*self.basis_num:i*self.basis_num+self.basis_num] @ np.array([1,x[0],x[1],x[0]*x[0],x[1]*x[1]])  for i in range (self.M)])
    a = a - np.max(a)
    p = np.array([np.exp(a[i])/np.sum(np.exp(a)) for i in range(self.M)])
    return p

def ACC_Confuse(self,x,w,N):
    #T = np.zeros(4)
    T = np.zeros((self.M,self.M))
    y = np.array([self.get_test(np.array([x[i][1],x[i][2]]),w) for i in range(N)])
    result = np.array([np.argmax(y[i]) for i in range(N)])
    for i in range(N):
        T[int(x[i][0])][result[i]]+=1
    A = 0
    for i in range(self.M):
        A+=T[i][i]
    ACC = A / np.sum(T)
    return T,ACC
```

```python
def iteration(self):
    w = np.zeros(self.basis_num*self.M)
    ita = 1e-4
    error = self.error(w)
    for i in range(20000):
        gradi = self.gradient(w)
        w -= ita * gradi
        print(f"error: {error} iter: {i}")
        error_new = self.error(w)
        if abs(error - error_new) < abs(error)*0.000001:
            print("finish gradien descent")
            break
        error = error_new
    return w
def error(self,w):
    error = 0.0
    for i in range(self.total_num):
        error -= self.get_y(self.data_training[i],w,int(self.data_training[i][0]))
    return error
def newton(self):
    w = np.zeros(self.basis_num*self.M)
    error = self.error(w)
    while 1:
        gradi = self.gradient(w)
        gradi_gradi = self.gradi_gradi(w)
        w -= np.linalg.pinv(gradi_gradi) @ gradi
        error_new = self.error(w)
        if abs(error - error_new) < abs(error)*0.0001:
            print("finish NR")
            break
        error = error_new
    return w
```

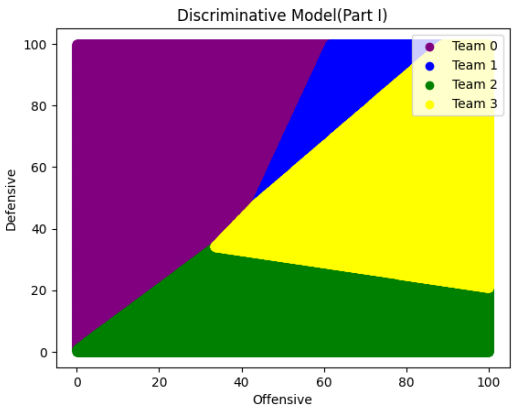
3. **Part I 中的 Discriminative Model 比較 Newton-Raphson method 以及 Gradient Descent method**

在 Discriminative Model 中，有兩種方式用來尋找最佳的 weight，Newton-Raphson method 的方式比起 Gradient Descent method 能夠更快速地收斂，因此程式運行的時間 Newton-Raphson 比 Gradient Descent 快非常多，然而，Newton-Raphson 需要建立▽▽E(w)，此算式相對不容易建立，通常會根據題目的需求去選取適合的方法。在 gradient descent 中，我迭代出來的結果(迭代大概 17000 次)如圖右下，此種方法非常花時間(大概跑了 1 個小時)，且可能因為迭代不夠多次而影響最後的準確率(training data 的 accuracy 只有 0.766)，因此可以得知此題使用 Newton-Raphson method 的方式比起 Gradient Descent method 好非常多。

Newton-Raphson method:



Gradient Descent method:

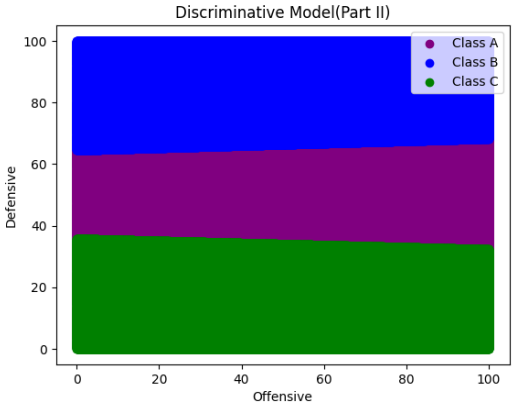


4. Part II 中的 Discriminative Model，使用二階的 basis function

$$\varphi(x) = [1, x_1, x_2, x_1^2, x_2^2]^T$$

在 Discriminative Model 中，訓練出的模型會根據 basis 的複雜程度而影響最後分類的 accuracy，因為 part I 使用一階的 basis function $\varphi(x) = [1, x_1, x_2]^T$ 訓練出的模型 accuracy 已經很高了，因此我拿 part II 來進行比較，若用一階的 basis 訓練出的 decision boundary 如下左圖，此 boundary 為直線，然而使用二階的 basis 訓練出的 decision boundary 如下右圖，此時 boundary 變成曲線，而 Training data 以及 Testing data 的 accuracy，分別上升到 0.942 以及 0.909(原本為 0.853 以及 0.842)。

一階 basis:



二階 basis: