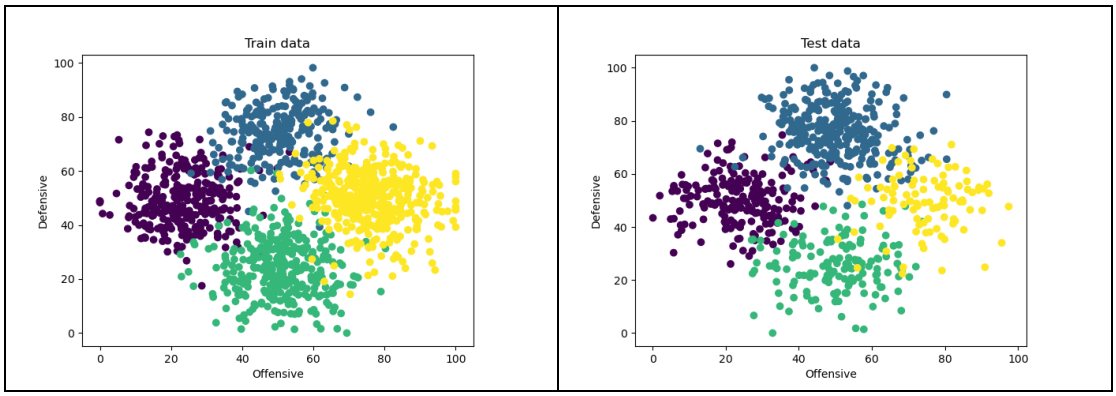
ML HW2 110511164 張語楹

Part 1

首先，我們先看看資料長甚麼樣:



訓練資料和測試資料都是被分為 4 類且從相同的分布所取樣出來。

Generative model:

在 generative model 部分，我們假設 data distribution 是 Gaussian distribution 因此可以推出下面的式子:

$$a_k(x) = w_k^T x + w_{ko}\ where \quad w_k = \Sigma^{-1}\mu_k \quad w_{k0} = -\frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \ln p(C_k)$$

根據上面的式子，我依照下面 4 個步驟來求出 $y(x)$:

1. compute $\mu_1, \mu_2, \ldots \ldots \mu_k$

```
# compute the average of each feature of each team
team_feature = np.zeros((num_teams, num_features))
for i in range(num_teams):
    team_feature[i,0] = train_data[train_data['Team'] == i]['Offensive'].mean()
    team_feature[i,1] = train_data[train_data['Team'] == i]['Defensive'].mean()
```

2. compute the common covariance matrix Σ

```
# compute the covariance matrix
cov = np.zeros((num_features, num_features))
for i in range(len(train_data)):
    team = int(train_data['Team'][i])
    x = np.array([train_data['Offensive'][i], train_data['Defensive'][i]])
    x = x - team_feature[team]
    x = x.reshape(-1,1)
cov += np.dot(x, x.T)
cov /= len(train_data)
```

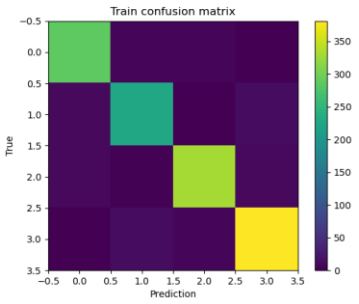
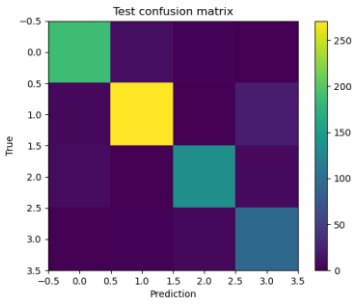
3. For each class, compute w_k and w_{k0}

```
# for each class, compute wk and wko (different classes is seperated by different
rows)
w = np.zeros((num_teams, num_features))
w0 = np.zeros(num_teams)
for i in range(num_teams):
     w[i] = np.dot(np.linalg.inv(cov), team_feature[i])
     w0[i] = -0.5 * np.dot(np.dot(team_feature[i], np.linalg.inv(cov)),
team_feature[i]) + np.log(c[i] / len(train_data))
```

4. Use w_k and w_{k0} to compute a_k and apply softmax

```
# use train data to compute the accuracy
train_predictions = np.zeros(len(train_data))
train_logits = np.zeros((len(train_data), num_teams))
correct = 0
for i in range(len(train_data)):
     target = int(train_data['Team'][i])
     x = np.array([train_data['Offensive'][i], train_data['Defensive'][i]])
     for j in range(num_teams):
          train_logits[i,j] = np.dot(w[j], x) + w0[j]
          # apply softmax
          predict = np.exp(train_logits[i]) / np.sum(np.exp(train_logits[i]))
          train_predictions[i] = np.argmax(predict)
     if train_predictions[i] == target:
            correct += 1
```

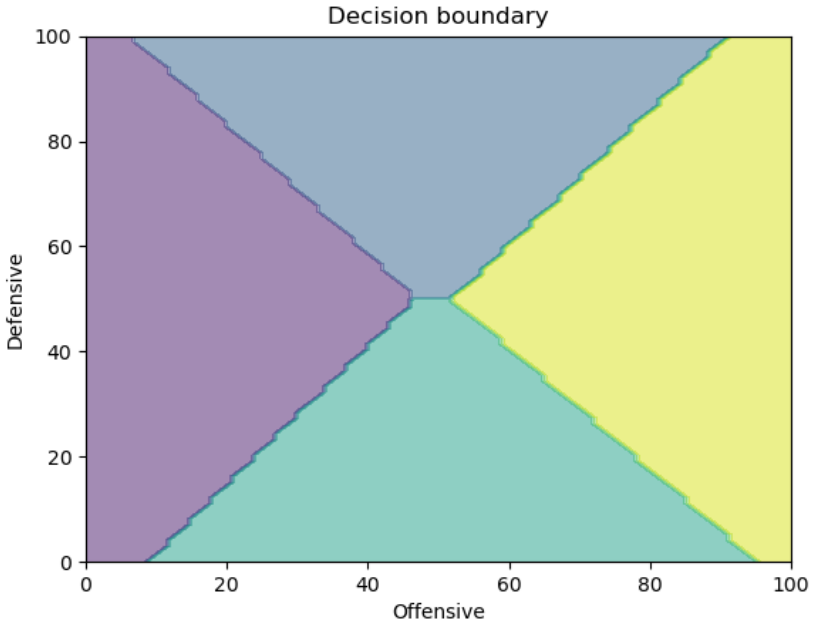
計算出每個 input 所對應的 class 後，就可以算出 accuracy 和 confusion matrix

Train accuracy: 94.3%	Test accuracy: 91.2%
[287. 6. 7. 0.] [9. 229. 0. 12.] [10. 2. 329. 9.] [0. 12. 7. 381.]	[187. 10. 3. 0.] [6. 271. 1. 22.] [8. 0. 135. 7.] [0. 3. 6. 91.]
	

接下來，我利用剛剛算出來的 weight 來畫出 decision boundary:

```
# plot the decision boundary
x1, x2 = np.meshgrid(np.linspace(0, 100, 100), np.linspace(0, 100, 100)) # create a
100x100 grid
x = np.array([x1.ravel(), x2.ravel()]).T # flatten and transpose to get a 2D array
predictions = np.zeros(x.shape[0])
logit = np.zeros((x.shape[0], num_teams))
for i in range(x.shape[0]):
     for j in range(num_teams):
          logit[i,j] = np.dot(w[j], x[i]) + w0[j]
         # get the class with the highest probability
         predictions[i] = np.argmax(logit[i])
plt.figure()
plt.contourf(x1, x2, predictions.reshape(x1.shape), alpha=0.5)
plt.xlabel('Offensive')
plt.ylabel('Defensive')
plt.title('Decision boundary')
plt.savefig('decision_boundary.png')
```

這個 decision boundary 是先製造出 100*100 的 grid，然後每個 grid 去做分類，並將分類的結果以顏色來表示。若顏色相同，則代表類別相同。



Discriminative model:

Discriminative model 和 generative model 不同的點是，discriminative model 直接求出 conditional PDF 而 generative model 是先求出 joint PDF 然後再根據 Bassian 來算出 conditional PDF. 此外，discriminative model 是以迭代的方法來求出 weight，這個和 generative model 不同。

和第一次作業一樣，我們要先將輸入轉成 design matrix:

```
def design_matrix(features):
    '''
    input: features (N,2)
    output: design matrix (N,3)
    '''
    return np.concatenate((np.ones((features.shape[0],1)), features), axis=1)
```

接下來，我們 weight 會用 Newton-Raphson 的方法做更新:

$$w^{new} = w^{old} - H^{-1}\,\nabla \mathrm{E(w)}$$

H 是 $\mathrm{E(w)}$ 的二階微分，也就是 curvature. 而 $\nabla \mathrm{E(w)}$ 是 $\mathrm{E(w)}$ 的一階微分，也就是斜率。我們的$\mathrm{E(w)}$會隨著更新而往下降。下面的 N 代表是總共資料的數量，K 指的是有幾個類別，K 是指 basis function 的數量。

此外，為了讓輸出矩陣的 dimension 為 (N,K)，本題在計算輸出的時候我採用比較不同的維度設計。首先 weight matrix 是 (M,K) 的矩陣，而 x 是 (N,M)的矩陣，因此兩者相乘 x@w 的矩陣的 dimension 為 (N,K)，符合我所要的。但這樣會遇到一個問題，就是我們 weight 的更新值的 dimension 為 (MK,1)，因此在更新時和做 forward pss 時都要先 reshape 我們的 weight matrix。

在 initialization 時，我們將 weight matrix 初始化為 0 的矩陣，並且計算出它原始的 accuracy 和 cross-entropy loss。

```
w_old = np.zeros((num_features, num_of_classes))
w_new = w_old
old_train_accuracy = accuracy(np.matmul(train_features,w_old),
train_targets_onehot)
print('Initial train accuracy:', old_train_accuracy)
old_test_accuracy = accuracy(np.matmul(test_features,w_old), test_targets_onehot)
print('Initial test Accuracy:', old_test_accuracy)
train_losses = []
test_losses = []
train_loss = cross_entropy(forward_pass(train_features, w_old),
train_targets_onehot)
```

```
test_loss = cross_entropy(forward_pass(test_features, w_old), test_targets_onehot)
print('Initial train loss:', train_loss)
print('Initial test loss:', test_loss)
train_losses.append(train_loss)
test_losses.append(test_loss)
```

接下來，就可以開始 iterative 更新 weight。

```
for i in range(args.epochs):
    print('Epoch:', i)
    if i % 1 == 0:
    # do the forward pass
    y = forward_pass(train_features, w_old) # (N,K)
    # compute the gradient of w
    partial_w = grd_w(train_features, y, train_targets_onehot) # (MK,1)
    Hassian_w = grd_grd(train_features, y, train_targets_onehot) # (MK,MK)
     # reshape w to (MK,1)
     w_old = w_old.T.reshape(-1,1)
     w_new = w_old - np.linalg.pinv(Hassian_w) @ partial_w
     # w = w - 0.1*partial_w
    # reshape w back to (M,K)
     w_old = reshape_weight(w_old)
     w_old = w_new
```

上面的程式碼省略一些紀錄 loss 和 early stopping 的部分，可以歸納為下面幾個 operation:

1. forward pass
2. compute the gradient
3. compute the Hessian matrix
4. reshape the weight from (M,K) to (MK,1)
5. update the weight
6. reshape the weight from (MK,1) to (M,K)

1. forward pass 非常單純，其實就是把輸出算一遍就好:

```
def forward_pass(x,w):
    '''
    input x: design matrix (N,M)
    input w: weight matrix (M,K)
    '''
```

```
    predictions = np.matmul(x,w) # (N,K)
    predictions = predictions - np.max(predictions, axis=1, keepdims=True)
    predictions = np.exp(predictions) / np.sum(np.exp(predictions),
axis=1).reshape(-1,1)
    return predictions
```

2. compute the gradient 也是根據公式算就可以:

先算出一個 vector 後沿著 row 的方向把他 concatenate:

```
def grd_w(x,y,t):
    '''
    compute the gradient of w
    input x: design matrix (N,M)
    input y: predicted target (N,K)
    input t: true one hot target (N,K)
    '''
    def partial_w_j(x,y,t,j):
        '''
        compute the element of the gradient of w
        j is the index of the classes, compute the error of the j-th class
        returns a (M,1) vector
        '''
        w_k = 0
        for i in range(x.shape[0]):
            w_k += (y[i,j] - t[i,j])*x[i,:] # (M,)

        return w_k.reshape(-1,1)

    return np.concatenate([partial_w_j(x,y,t,j) for j in range(num_of_classes)],
axis=0) # (MK,1)
```

3. compute the Hessian matrix 也是根據公式算就可以:

先算出一個 block 後沿著 row 和 column 的方向把他 concatenate:

```
def grd_grd(x,y,t):
    '''
    compute the Hassian matrix of w
    input x: design matrix (N,M)
    input y: predicted target (N,K)
    input t: true one hot target (N,K)
    returns a (MK,MK) matrix
```

```
    '''
    def partial_w_k_w_j(x,y,t,k,j):
        '''
        compute the block of the Hassian matrix of w
        k is the index of the classes, compute the error of the k-th class
        j is the index of the classes, compute the error of the j-th class
        returns a (M,M) matrix
        '''
        w_k_w_j = np.zeros((x.shape[1],x.shape[1]))
        for i in range(x.shape[0]):
            w_k_w_j += y[i,k]*(int(k==j) - y[i,j])*np.outer(x[i,:],x[i,:])

        return w_k_w_j

    # j is the column index, k is the row index
    return np.block([[partial_w_k_w_j(x,y,t,k,j) for j in range(num_of_classes)] for
k in range(num_of_classes)]) # (MK,MK)
```

4. reshape the weight from (M,K) to (MK,1)

```
w_old = w_old.T.reshape(-1,1)
```

5. update the weight

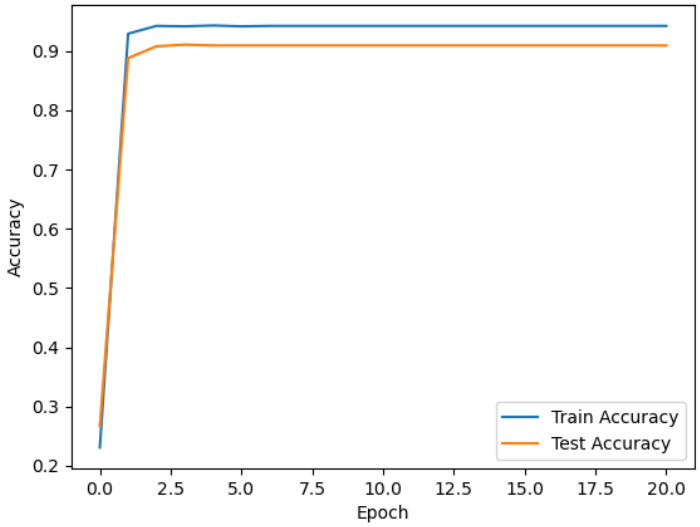
```
w_new = w_old - np.linalg.pinv(Hassian_w) @ partial_w
```

6. reshape the weight from (MK,1) to (M,K)

```
w_old = reshape_weight(w_old)
w_new = reshape_weight(w_new)
def reshape_weight(w):
    '''
    reshape the weight matrix to (M,K)
    '''
    w_new = np.zeros((num_features,num_of_classes))
    for i in range(num_of_classes* num_features):
        w_new[i % num_features, i // num_features] = w[i]
    return w_new
```

不斷的更新 weight 後就能得到和上面 generative model 的方法接近的效果。

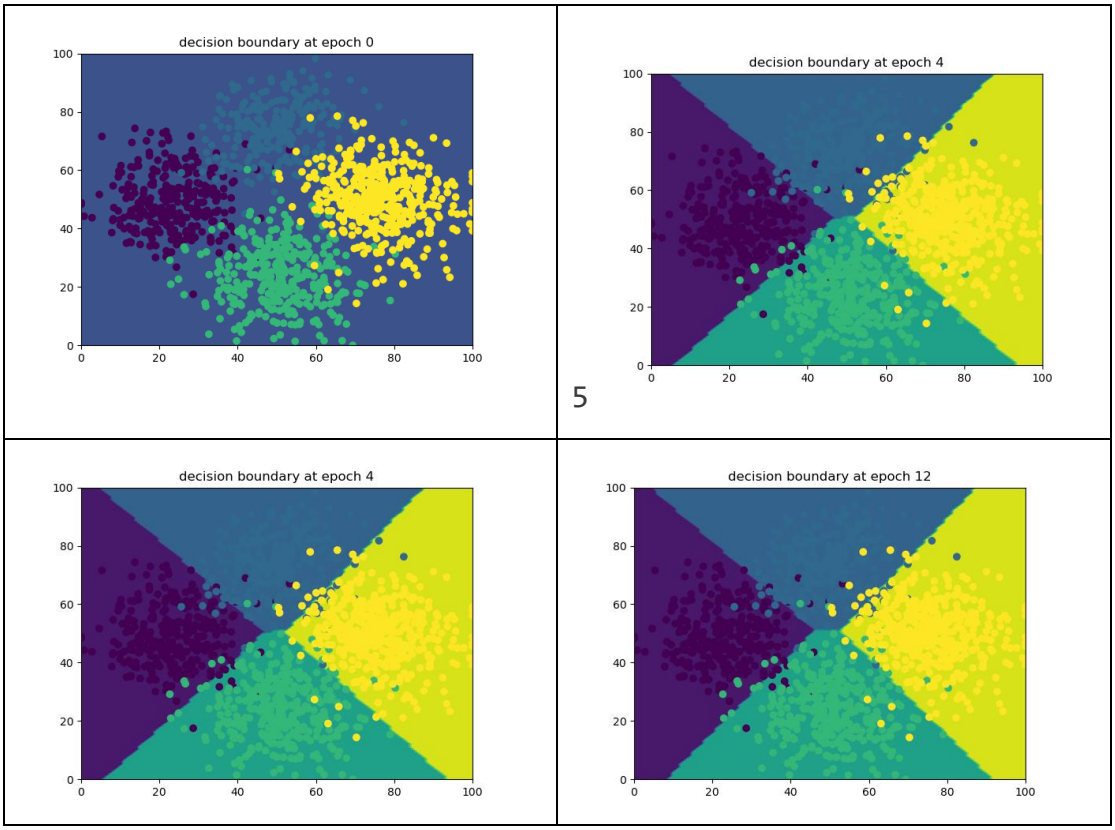
Accuracy:



Train Accuracy: 94.2%

Test Accuracy: 90.1%

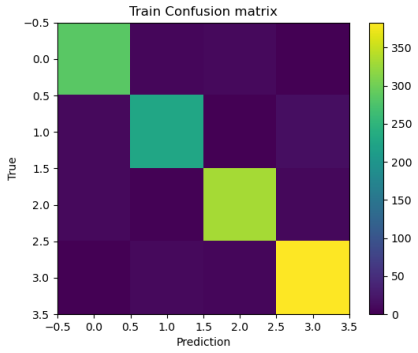
Decision boundary at different epoch:



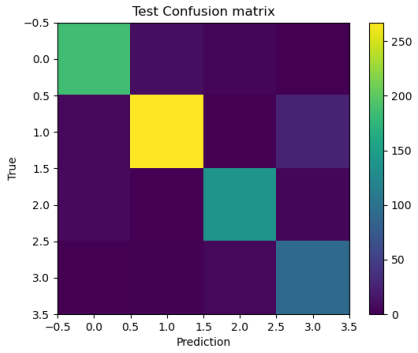
5

Confusion matrix:

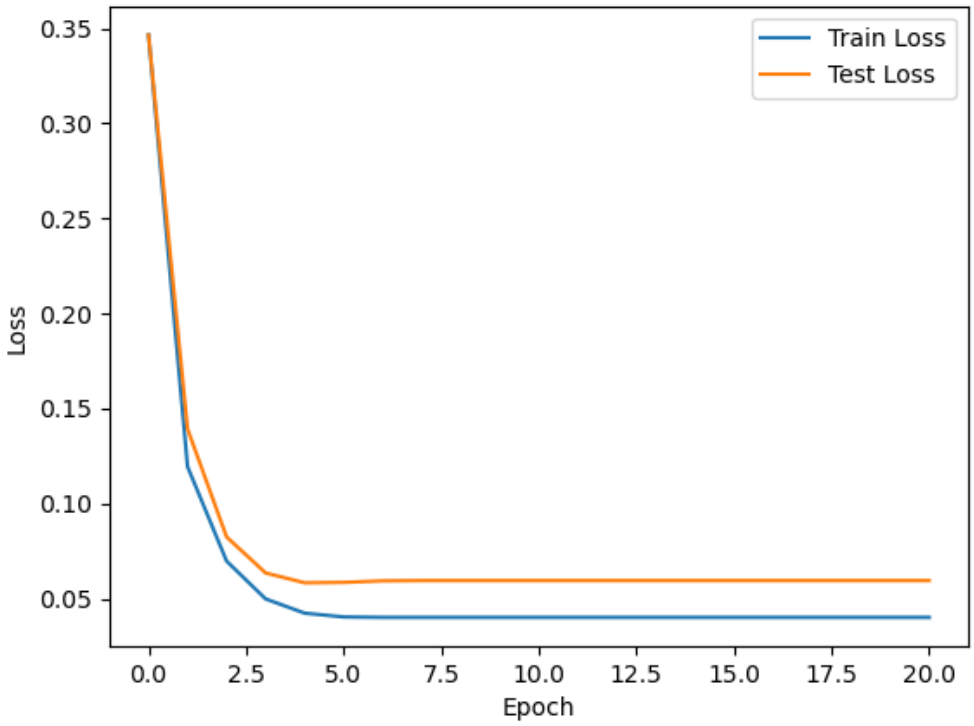
```
[[285.   6.   9.   0.]
 [  9. 226.   1.  14.]
 [  9.   2. 331.   8.]
 [  0.  10.   7. 383.]]
```




```
[[185.  10.   5.   0.]
 [  6. 267.   1.  26.]
 [  7.   0. 138.   5.]
 [  0.   2.   6.  92.]]
```

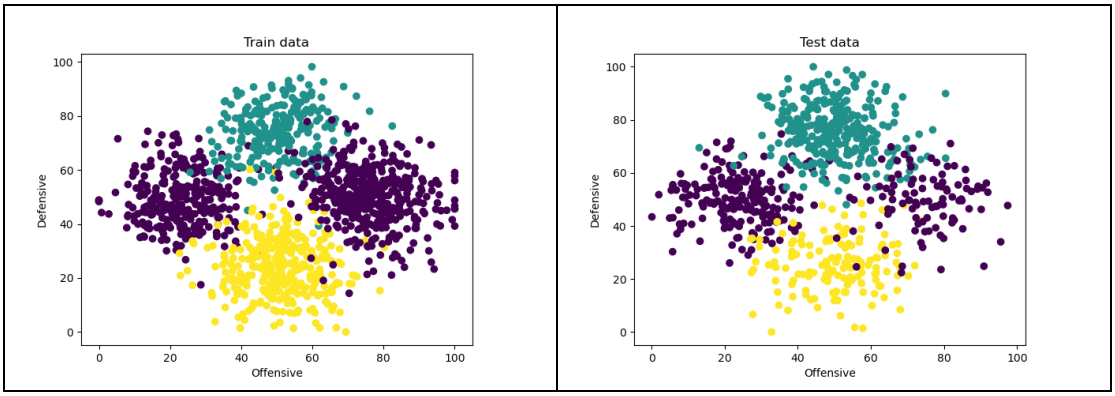


Loss:



Part 2

這部分和上個部分類似，只是將 Team 3 和 Team 0 歸成相同的類別。我的方法是直接是更改.csv 檔案，這樣就不用更改程式碼。
因為這個部分的類別比較少，因此這個部分的準確度以直覺來說應該會比上一個部分的準確率還要高。然而，結果卻和預測的不同。
首先，我們先看看資料長甚麼樣:



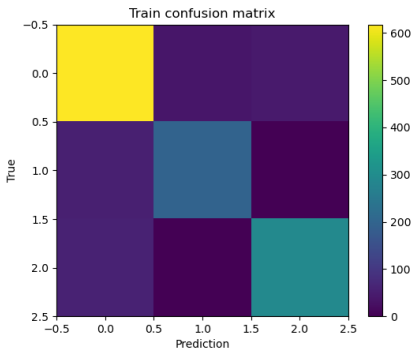
訓練資料和測試資料都是被分為 4 類且從相同的分布所取樣出來。

Generative model:

Accuracy and confusion matrix:

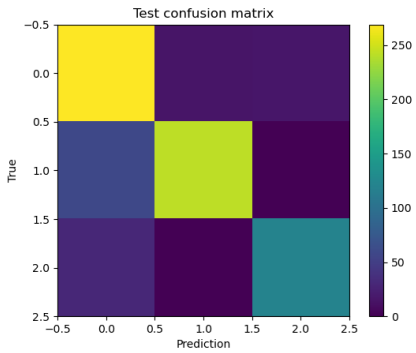
Train accuracy: 85.1%

[[618. 37. 45.]
[54. 196. 0.]
[58. 0. 292.]]

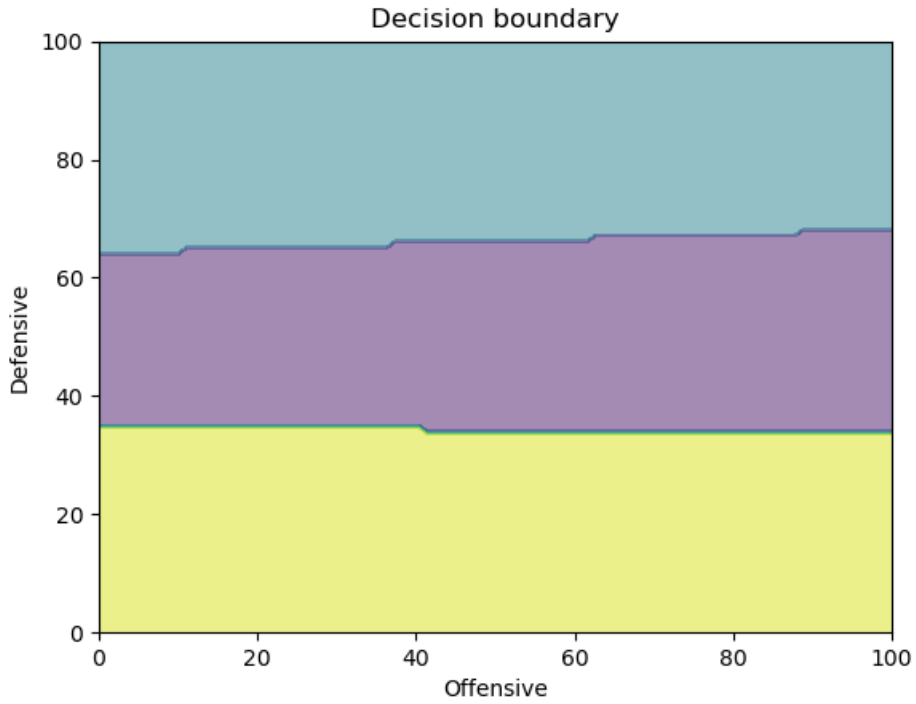


Test accuracy: 84.1%

[[269. 15. 16.]
[58. 242. 0.]
[30. 0. 120.]]

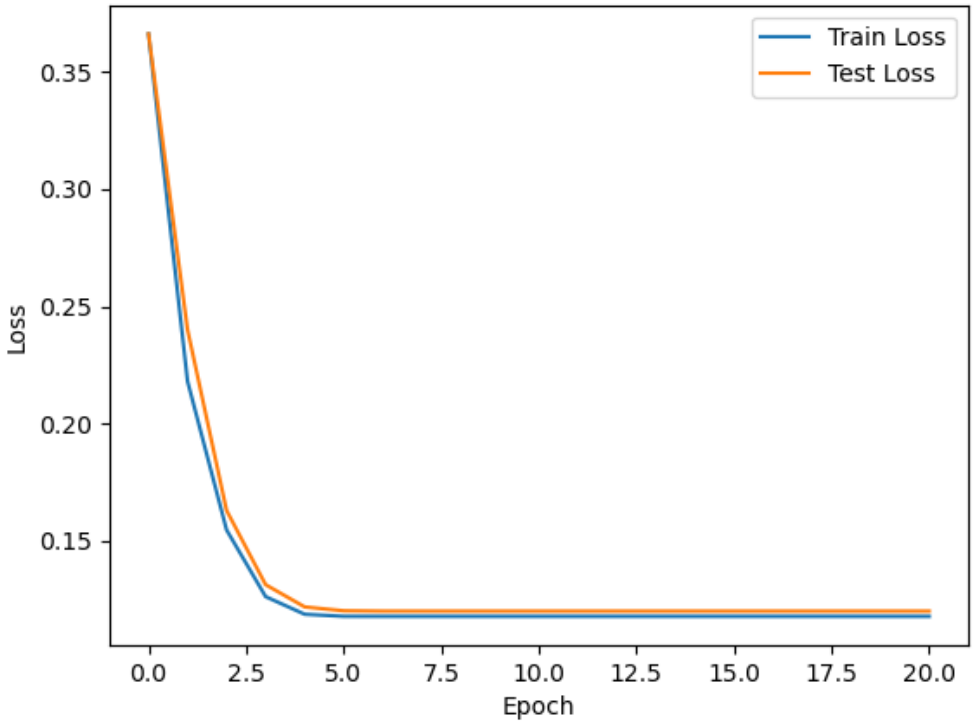


Decision boundary:



Discriminative model:

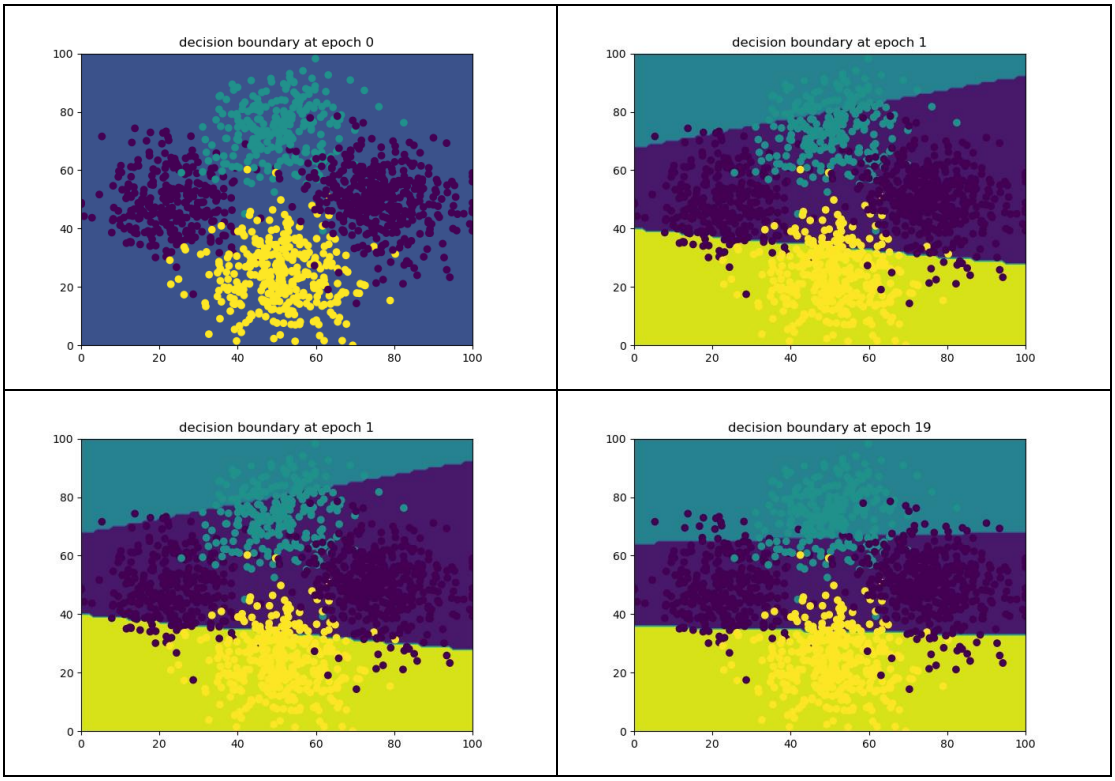
Accuracy:



Train Accuracy: 85.3%

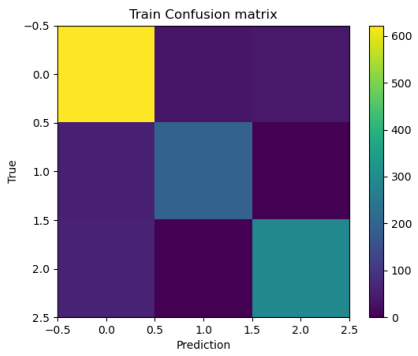
Test Accuracy: 84.3%

Decision boundary at different epoch:

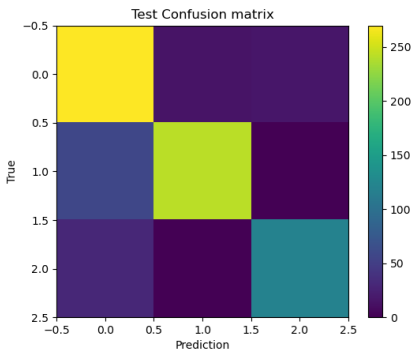


Confusion matrix:

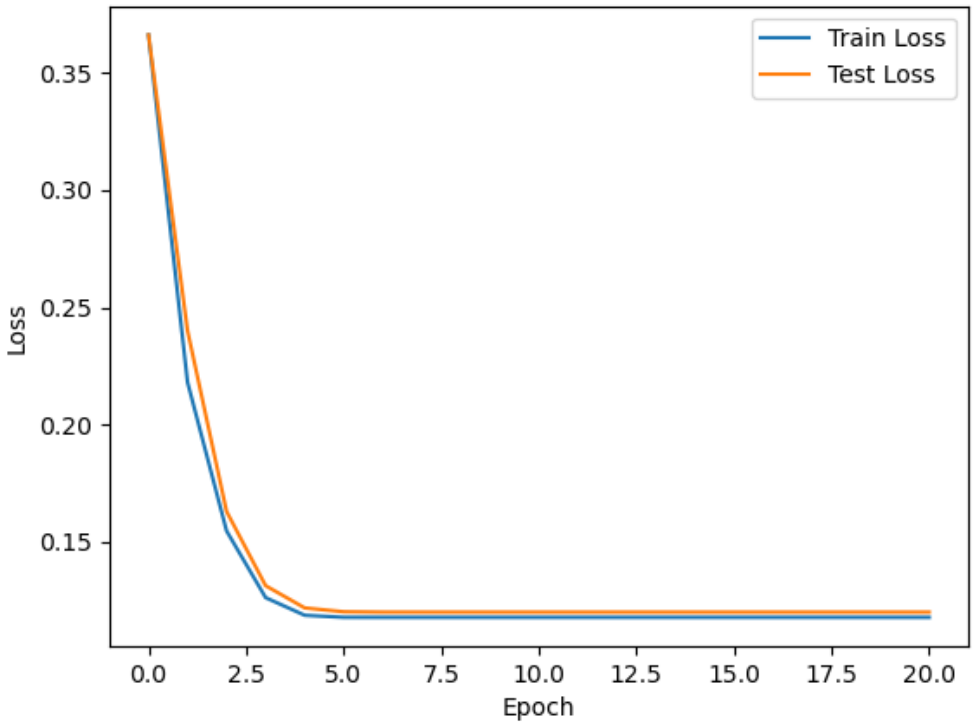
```
[[622.  35.  43.]
 [ 54. 196.   0.]
 [ 59.   0. 291.]]
```




```
[[270.  14.  16.]
 [ 58. 242.   0.]
 [ 30.   0. 120.]]
```



Loss:



Discussion:

What is the difference between the generative model and the discriminative model?

這題剛剛有回答過。Discriminative model 和 generative model 不同的點在於 discriminative model 直接求出 conditional PDF 而 generative model 是先求出 joint PDF 然後再根據 Bassian 來算出 conditional PDF. 此外，discriminative model 是以迭代的方法來求出 weight，這個和 generative model 不同。

How to prevent overflow problem when calculating SoftMax?

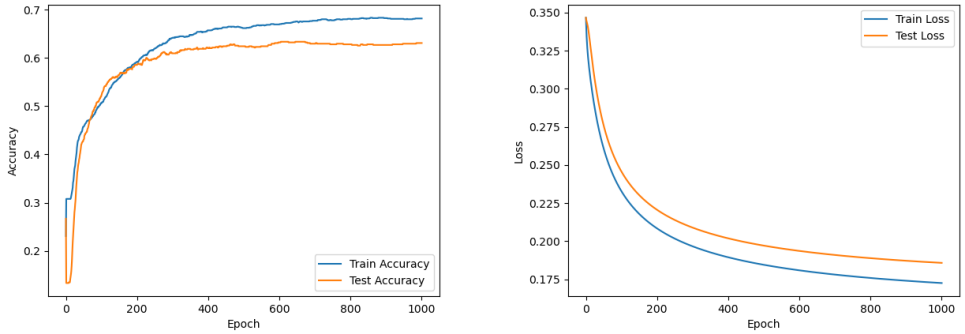
Softmax 的數學表示為:

$$softmax(x) = \frac{e^{a_k(x)}}{\sum_{i=1}^{N} e^{a_j(x)}}$$

可以看到分母有 exponential 又有 summation，很容易造成 overflow 的問題。因此，我們可以先取$a_k(x)$中的最大值，並將分子分母除掉$e^{\max(a_k(x))}$來避免 overflow 的問題。

```
predictions = predictions - np.max(predictions, axis=1, keepdims=True)
predictions = np.exp(predictions) / np.sum(np.exp(predictions), axis=1).reshape(-1,1)
```

What is the difference between gradient descent and Newton-Raphson method?
Newton-Raphson method 其實就是可以動態調整學習率 (learning rate)的 gradient descent。其中，調整的依據就是 Hessian matrix，也就是 error function 的 curvature。我們也可以用 gradient descent 來實作 discriminative model。
經過調整 learning rate 之後，我用 1e-7 的學習率訓練 1000 個 epoch，loss curve 和 accuracy 如下圖:



其實光看數據就知道，gradient descent 訓練更多次卻沒有帶來更好的效果。所以可以很明顯地看到 Newton-Raphson method 所帶來的好處。然而，Newton-Raphson method 的 tradeoff 就是要計算 Hessian matrix 帶來更多的運算量。

心得:
這次作業分為 generative 和 discriminative。我覺得 generative 比較簡單，運算的步驟比較少一點，而 discriminative 我寫得比較久。其中，我覺得最難的是我最一開始的 weight matrix 並不是(MK,1)所以在 reshape 的過程中出了問題，也依職沒有 debug 出來，直到把 weight matrix 印出來來後才發現問題。再次提醒我們打扣的時候要非常小心。
GitHub repo: https://github.com/Benchangatrul284/NYCU_ML