04
chapter

分類問題

上一個章節的資料探勘中，我們沒有探討機器學習的各類方法中，屬於監督式學習的重要任務——分類（Classification）。分類是機器學習最常見的任務，相關的算法也遠比聚類或其他規則分析來得多。以下我們會先討論分類問題的主要架構和目標，再介紹一些現今常見的分類演算法，並進行簡單的實作。

4.1 分類問題的形式和目標

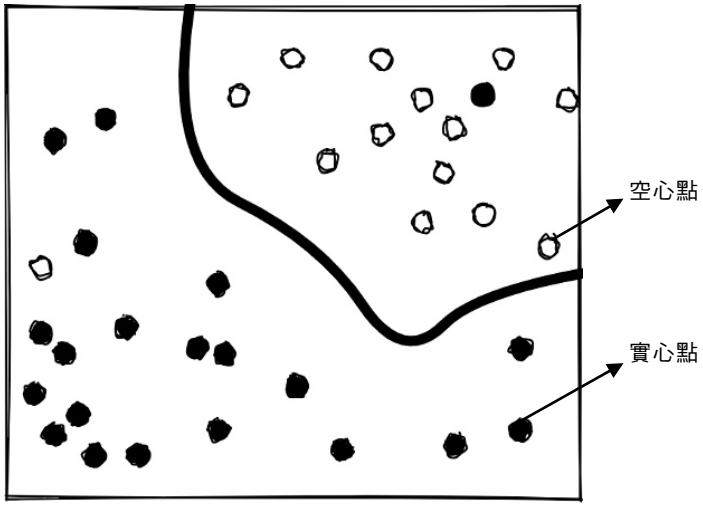
4.1.1 詐騙郵件

電子郵件問世至今，時常有大量廣告或詐騙郵件，收 email 時總要確認是不是垃圾郵件，相當耗費心力。垃圾郵件越來越多，但近年來這個問題大幅改善，因為這類郵件的初步篩選不再是用戶的問題，而是交由電子郵件平台上的程式來處理。這便是分類問題在現今生活的一個重要應用。

電子郵件平台發現垃圾郵件氾濫的問題後，建立了回報垃圾郵件的機制。透過使用者的回饋，他們可以輕鬆取得被認定有問題的郵件，並假設大部分沒有被回報的文章是正常的郵件。接下來程式就能比較被眾多使用者歸類的垃圾郵件與正常郵件有什麼不一樣（例如：詐騙郵件可能有 ATM 轉帳相關的敘述），最後運用垃圾郵件的特性，建立一些分類原則，就能得到一個郵件分類器。

4.1.2 分類器

在分類問題中，我們稱每筆資料的數據部分叫做資料（data），分類名稱叫做標籤（label）。分類問題的目標，就是從資料中的各項數據，找出這筆資料有其對應標籤的原因，並運用一些判別規則構成分類器（classifier），盡可能區隔不同標籤的資料，讓沒有標籤的資料點輸入分類器時，能有正確的分類歸屬。



我們用前面的例子跟上方的示意圖，描繪分類問題的運作目的。在一個平面上有數個點代表每一封 email，空心點與實心點分別代表正常郵件與垃圾郵件，可以發現它們各自聚成一群。圖上的曲線代表的就是分類器，我們稱為分類線，當一封 email 輸入分類器時，它所代表的資料點如果落在分類線右側就會被當作正常郵件，否則會被當作垃圾郵件。

分類線不一定會成功判別每一個數據點，畢竟有些非典型的垃圾郵件仍會騙過分類器，但機器學習的目的通常是獲得一個一般化的分類器，圖上的分類線落在兩個點群的正中間，便是相當理想的學習成果。

接著來介紹一些常見的分類演算法。每種算法各有長短，重要的是針對眼前的資料，選擇最適當的模型，才能有期望的結果。

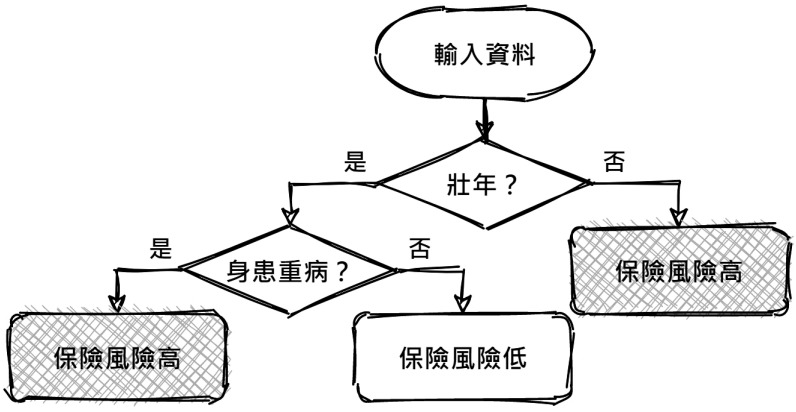
4.2 決策樹

4.2.1 二分法與決策樹

生物課曾提過依據動植物外貌或生存特性逐一分類的二分法，機器學習也有個幾乎相同的分類結構，稱為**決策樹（Decision Tree）**。決策樹利用特定性質的資訊，將資料不斷地分為數個群，直到每個子群大都屬於同一類別為止，通常以分類樹上該層變異數最大的單項數據作為分類依據，但有顯而易見的規律時，也可參雜人工的分類規則。決策樹適用在大原則的歸納，太細的分類規則與分類標籤的相關性通常不高，新資料的歸類結果也未必正確。

編號	年齡層	投保年份	身患重病	保險風險
1	壯年	2018	無	低
2	老年	2018	無	高
3	兒童	2019	有	高
4	老年	2019	有	高
5	壯年	2020	有	高
6	壯年	2020	無	低

假設某間保險公司要依照客戶性質，用決策樹評估保險的風險。從表格可以推論身患重病或非壯年者，保險風險比較高，且看不出投保年份與風險之相關性，於是歸納出下圖的決策樹。



程式實作 決策樹應用

■ 檔案：ch4/Decision_Tree.ipynb

了解決策樹的架構後，我們用 *scikit-learn* 套件和 Iris 資料集實際分類一次，再從決策樹的資訊了解決策樹的判別算法。

```
# 匯入套件
from sklearn import tree, datasets
import matplotlib.pyplot as plt
```

載入資料集

這次的實作和先前聚類的實作一樣都會使用 Iris，不一樣的是我們把 *return_X_y* 參數設成 *True*，直接把資料集分成資料部分和標籤部分。

```
X, y = datasets.load_iris(return_X_y=True)  # 載入資料
```

模型學習

我們使用 *DecisionTreeClassifier()*建立一個決策樹物件，並用 *fit()*函式擬合 *X* 和 *y*，讓對應的資料都能有對應的標籤。為了得到比較一般化的決策樹，我們使用 max_depth 參數控制決策樹的深度，讓模型不會為了處理特例繼續細分，而是在有限的決策樹深度下，找到最佳的二分方式。

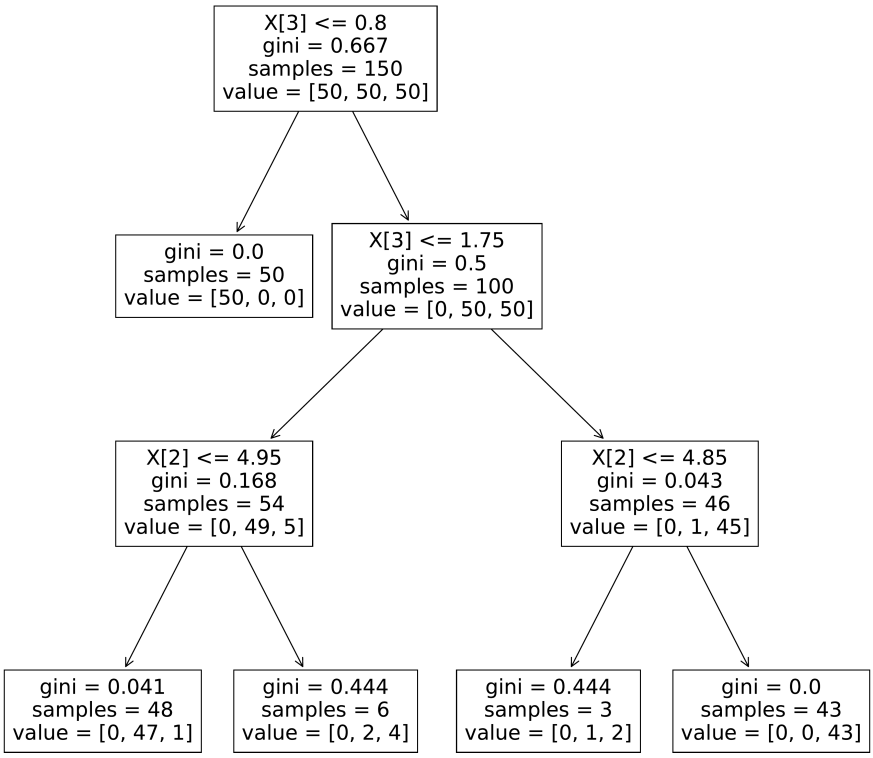
```
clf = tree.DecisionTreeClassifier(max_depth=3)  # 建立決策樹模型
clf = clf.fit(X, y)  # 模型訓練
```

結果視覺化

在 *scikit-learn* 的 *tree* 子套件中，我們可以用 *plot_tree* 函式直接把學習完的決策樹畫出來，但要先用 *plt* 的 *figure()*函式調整圖表框架變大，才會得到比較大張的樹狀圖。

```
plt.figure(figsize=(10,10))  # 圖框放大
tree.plot_tree(clf)  # 繪製決策樹
plt.show()  # 圖像輸出
```

輸出



4.2.2 決策樹架構和數學原理

我們可以從上方的決策樹圖形中，理解這個算法的運作方式。首先，我們要先能理解每個方框的內容。

有些方框的第一行寫著一個小於等於的判斷式，也就是這個點向下將資料分為兩群的方式。如果某筆資料符合上述的條件，就會被分到左下方的決策樹，反之會被分到右下方的決策樹。

Iris 資料集中，`X[0] ~ X[3]` 分別是花朵的花萼長度、花萼寬度、花瓣長度和花瓣寬度，所以從樹的頂端的判斷式來看，如果花瓣寬度小於0.8，就會被分到左下方只有一個方框的決策樹，否則會被分到右下方較為複雜的決策樹。

方框最下方的 `samples` 和 `value` 就是訓練資料中，`samples` 是符合這個方框上方所有分類條件的樹，而 `value` 就是每一個分類個別的資料筆數。如果我們能讓某個方框內的 `values` 中的某一項數字特別高，表示符合這個決策樹分支上所有條件的資料集，有很高的機率都是同一個類別，這個也會被視為是相當成功的分類結果。以這顆決策樹頂端的分類來說，左下方的子樹全都是第一類的山鳶尾，能在決策樹的上端直接排除其中一個分類，是非常成功的二分條件。

吉尼不純度

雖然剛剛說 Iris 決策樹頂端 `X[3] <= 0.8` 是個很好的分類條件，但怎麼量化這個分類條件有多好？細心的讀者可能留意到每個方框中，都會有一個名為 `gini` 的參數，這是決策樹選擇二分條件的重要指標。

假設有J個分類，且資料群中每個分類所占的比例為p_i，則我們可以求出吉尼不純度（Gini Impurity）。吉尼不純度的定義就是資料集當中一個隨機樣本在子集中被分錯的可能性高低，其計算方法為樣本被選中的機率乘以被分錯的機率，吉尼不純度的數值範圍為 0 到 1。

以第i個分類來說，他的分類占比為p_i，則這個分類的吉尼不純度就是p_i乘上其他分類的占比，也就是$p_i \sum_{j \neq i} p_j$。也就是說，一整個資料群的吉尼不純度，可以用下列的算式表示。

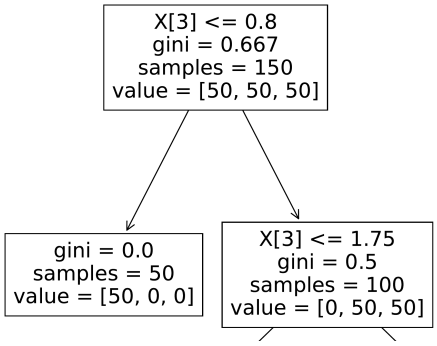
$$I_G(p) = \sum_{i=1}^{J} (p_i \sum_{j \neq i} p_j)$$

因為其他分類的占比就是$1 - p_i$，所以我們又能繼續簡化成以下的算式。

$$I_G(p) = \sum_{i=1}^{J} (p_i(1 - p_i)) = \sum_{i=1}^{J} (p_i - p_i^2) = \sum_{i=1}^{J} p_i - \sum_{i=1}^{J} p_i^2 = 1 - \sum_{i=1}^{J} p_i^2$$

也就是說，一個資料群中的吉尼不純度就是1減去各個分類比率的平方和。這個指標有什麼性質呢？

假設一個資料群集中，某一個分類比率特別突出，占了絕大部分的資料，那吉尼不純度就會接近$1-1^2$，也就會趨近於0，如圖中的左下節點，當一個節點中所有樣本都是同一類時，吉尼不純度數值為 0；反之如果 J 種分類都平均分布，每個類別的資料比率都是$\frac{1}{J}$，那麼吉尼不純度就會接近$1-J\times(\frac{1}{J})^2=1-\frac{1}{J}$，如圖中的頂點，3 種分類平均分布，每個類別的資料比率是$\frac{1}{3}$，其吉尼不純度數值為$1-3\times(\frac{1}{3})^2=1-\frac{1}{3}=0.667$；圖中右下節點的吉尼不純度為$1-(\frac{0}{100})^2-(\frac{50}{100})^2-(\frac{50}{100})^2=1-\frac{1}{4}-\frac{1}{4}=0.5$。



也就是說吉尼不純度越低的資料群，個別的資料屬於同一個類別的機率越大。

吉尼增量

決策樹的二分條件之所以參考吉尼不純度，就是希望能夠在分群的過程降低吉尼不純度。當今天有一個資料群C，且根據某個條件分成兩個資料群A和B，如何算出吉尼不純度的變化量？

首先我們可以算出三個資料群的吉尼不純度I_{GC}、I_{GA}和I_{GB}。再來，我們根據資料群的大小，算出資料群A和B的加權吉尼不純度I_{GAB}。

$$I_{GAB}=\frac{|A|}{|C|}\times I_{GA}+\frac{|B|}{|C|}\times I_{GB}$$

此處I_{GC}和I_{GAB}的差，就是分群後吉尼不純度的減少量，也被稱為吉尼增量（Gini Gain）。所以每次要二分一個資料群時，決策樹會選用吉尼增量最大的條件二分資料，讓下一層的決策樹逐步趨向同一個分類。

我們以 Iris 資料決策樹頂端的三個資料群當作C、A、B實際算一次吉尼增量。

$$I_{GC} = 0.667, I_{GA} = 0, I_{GB} = 0.5$$

$$\Rightarrow I_{GAB} = \frac{|A|}{|C|} \times I_{GA} + \frac{|B|}{|C|} \times I_{GB} = \frac{50}{150} \times 0 + \frac{100}{150} \times 0.5 \approx 0.333$$

$$\Rightarrow I_{GC} - I_{GAB} = 0.667 - 0.333 \approx 0.333$$

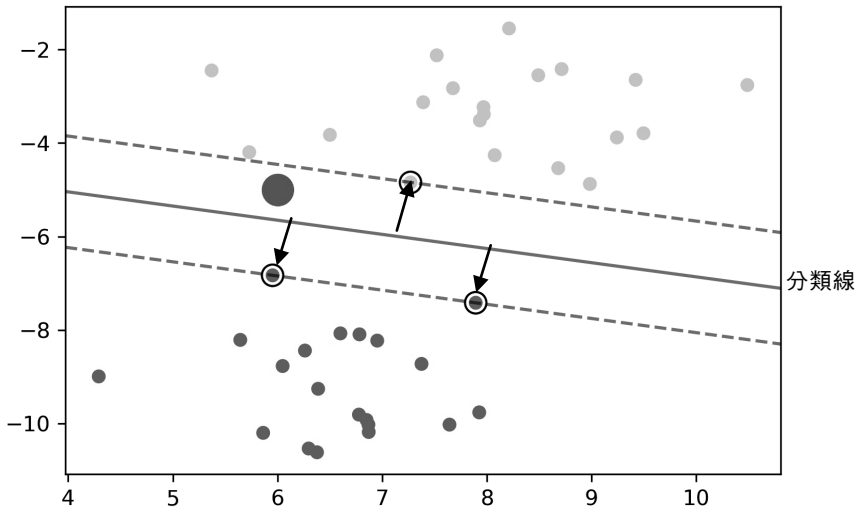
吉尼增量是原本吉尼不純度的一半，所以我們從數學上也說明了第一個二分條件有多好，畢竟它能直接消去一半的不純度，從 0.667 降為 0.333。

4.3 支持向量機

4.3.1 分類線與支持向量機

前面提過如果把分類問題視覺化，可以想像成用分類線，將資料點隔成同標籤的點群。不過相較於精確的分類曲線，找一條與不同點群距離相當的直線相對容易許多。用直線區分各類資料點的演算法，稱為**支持向量機**（Support Vector Machine，**簡稱** SVM）。

SVM 的目標是找一條直線分出兩類資料點的區域。當然這樣的直線有很多條，但比較理想的解是距離兩個點群相等的直線。在這裡分類線與點群的距離定義是點群到直線的最近點之最短法向量，而這個法向量就是支持向量（Support Vector）。以下面這張圖來說，支持向量就是通過圈起來的點的分類線法向量。既然希望分類線與兩點群距離相等，SVM 要找的就是長度相等、方向相反、且長度最大化的支持向量，如同下方的示意圖。學習完畢後，當分類器輸入一筆新的資料，它會視資料點落在哪個半平面，將該筆資料分給那個半平面的點群對應的分類。



以上面這張圖為例，如果小的資料點是訓練集，那麼 SVM 會用被圈起來的三個點決定出圖上的實線作為分類線。假設輸入的測試資料對應到圖上的大點，即使它落在兩個點群之間的模糊地帶，也會因為它在分類線的右上半平面，而被視為與右上方的點群同類。對 SVM 有初步的概念後，我們來用 Python 實作看看，觀察這個算法的運作。

程式實作 支持向量機視覺化

■ 檔案：ch4/SVM.ipynb

```
# 引入套件
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import make_blobs
```

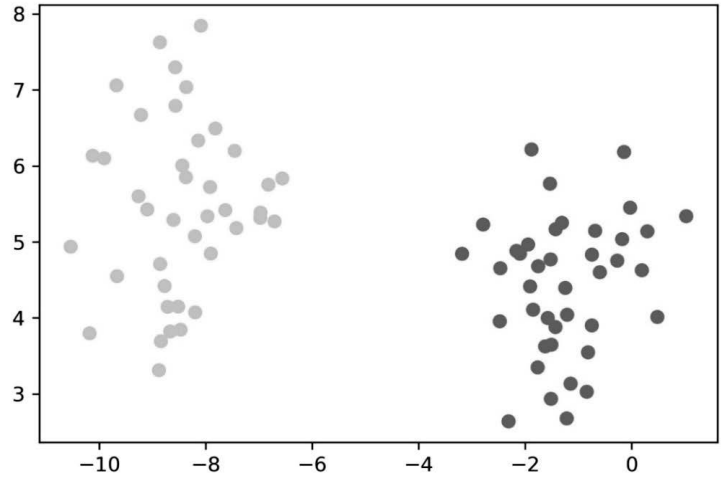
生成隨機資料

支持向量機大多會運用在資訊相對複雜且難以繪圖表示，本實作使用隨機生成的兩類資料點，數據相對單調，以便視覺化與理解觀念。首先，我們將 `centers` 和 `n_samples` 參數分別設為 2 和80，隨機生成具2種分類標籤的80個點。建議把 `random_state` 參數設為7，用固定的亂數種子確保散佈圖一致且線性可分。`make_blobs` 函式的回傳值X為80×2之座標點矩陣，y為80×1之分類矩陣。將點散佈在座標平面上之後，我們可以發現兩個分類的點分別聚於圖上兩側，參數 `c=y` 就是

將點的顏色依 y 的數值來區分；將 cmap 參數設為 plt.cm.Paired 代表要以 Paired 的顏色輸出點圖（colormaps 顏色請參考網址：https://matplotlib.org/stable/tutorials/colors/colormaps.html）。

```
# 構造資料
X, y = make_blobs(n_samples=80, centers=2, random_state=7)
# 資料點分布圖
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
plt.show()
```

輸出



SVM 學習

運用 *scikit-learn* 的 *SVC* 物件，創造支持向量分類器 *clf*，並將分類模式 *kernel* 設為線性*('linear')*，再用 *fit* 函式擬合先前生成散佈點的分類。

```
# 建構分類器
clf = SVC(kernel='linear')
clf.fit(X, y)
```

輸出

```
SVC(kernel='linear')
```

構造採樣格點

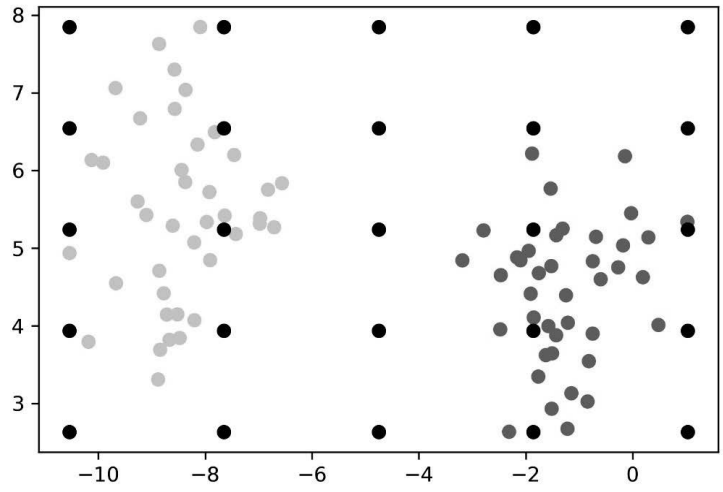
接下來我們要以散佈點的上下左右界，構造平均分布的格點。先分別使用 *np.min* 和 *np.max* 函式，找到兩軸的最大值與最小值後，用 *np.linspace* 分別以兩軸的極值為首項及末項，構造5項的等差數列，當作座標的採樣點。若兩數列分別為 *sample_x* 與 *sample_y*，則從兩個數列各取一點就是其中一個採樣格點的座標。

套用函式 *np.meshgrid* 後，我們可以把所有採樣格點的座標展開成兩個5 × 5的矩陣 *grid_x* 與 *grid_y*。把兩個矩陣的內容輸出，可以發現兩者的對應項恰好形成一個座標點；我們可以進一步地把資料點和採樣點散佈在圖上，確認採樣點的確是以資料點在兩軸極值的等分點。為了方便觀察，繪製採樣點時可以設定參數 *c='k'*，把這些格點標示為黑色。

```
# 構造格點
sample_x = np.linspace(np.min(X[:, 0]), np.max(X[:, 0]), 5)
sample_y = np.linspace(np.min(X[:, 1]), np.max(X[:, 1]), 5)
grid_x, grid_y = np.meshgrid(sample_x, sample_y)
print(grid_x)
print(grid_y)
# 散佈資料點
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
# 散佈採樣點
plt.scatter(grid_x, grid_y, c='k')
plt.show()
```

輸出

```
[[-10.53824902  -7.64665386  -4.7550587   -1.86346354   1.02813162]
 [-10.53824902  -7.64665386  -4.7550587   -1.86346354   1.02813162]
 [-10.53824902  -7.64665386  -4.7550587   -1.86346354   1.02813162]
 [-10.53824902  -7.64665386  -4.7550587   -1.86346354   1.02813162]
 [-10.53824902  -7.64665386  -4.7550587   -1.86346354   1.02813162]]
[[2.63687759 2.63687759 2.63687759 2.63687759 2.63687759]
 [3.93915092 3.93915092 3.93915092 3.93915092 3.93915092]
 [5.24142424 5.24142424 5.24142424 5.24142424 5.24142424]
 [6.54369757 6.54369757 6.54369757 6.54369757 6.54369757]
 [7.8459709  7.8459709  7.8459709  7.8459709  7.8459709 ]]
```



決策函數

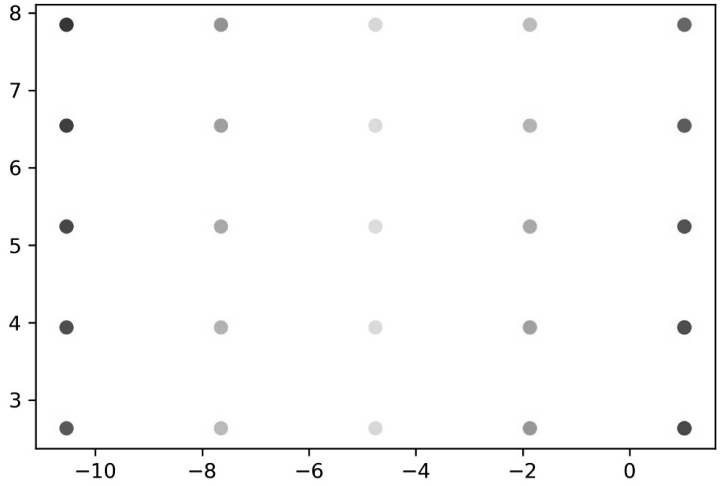
SVM 模型輸入的資料必須跟擬合時的資料點維度相同，所以我們要把格點座標從兩個5 × 5的矩陣變成一個25 × 2的矩陣才能輸入模型，完成25個格點的分類。我們可以先用 *ravel()*函式把 *grid_x* 與 *grid_y* 攤平為一維矩陣，再以 *vstack()*函式堆疊成2 × 25的矩陣，轉置後就是可以輸入模型的採樣資料 *sample_xy*。我們輸出 *sample_xy*，與上方的 *grid_x* 和 *grid_y* 對照，可以看出的確是由兩矩陣的對應項構成一組座標。

我們把上述矩陣傳入 *decision_function()*，並調整維度為格點的5 × 5後，得到各個格點的決策函數值矩陣 *z*。這些數值便是各個點與分類線，以支持向量長為單位的有向距離。我們把格點以 *z* 值進行色彩映射散佈在圖上，可以發現兩側的顏色較深，對應兩個資料點群的主要分布。注意到這裡用的是以數值的連續變化呈現暖色和冷色的 *coolwarm* 顏色映射，與離散顏色映射的 *Paired* 不同。

```
# 矩陣維度轉換
sample_xy = np.vstack([grid_x.ravel(), grid_y.ravel()]).T
print(sample_xy)
# 決策函數
z = clf.decision_function(sample_xy).reshape(grid_x.shape)
plt.scatter(grid_x, grid_y, c=z, cmap=plt.cm.coolwarm)
plt.show()
```

輸出

```
[[-10.53824902   2.63687759]
 [ -7.64665386   2.63687759]
 [ -4.7550587    2.63687759]
 [ -1.86346354   2.63687759]
 [  1.02813162   2.63687759]
 [-10.53824902   3.93915092]
 [ -7.64665386   3.93915092]
 [ -4.7550587    3.93915092]
 [ -1.86346354   3.93915092]
 [  1.02813162   3.93915092]
 [-10.53824902   5.24142424]
 [ -7.64665386   5.24142424]
 [ -4.7550587    5.24142424]
 [ -1.86346354   5.24142424]
 [  1.02813162   5.24142424]
 [-10.53824902   6.54369757]
 [ -7.64665386   6.54369757]
 [ -4.7550587    6.54369757]
 [ -1.86346354   6.54369757]
 [  1.02813162   6.54369757]
 [-10.53824902   7.8459709 ]
 [ -7.64665386   7.8459709 ]
 [ -4.7550587    7.8459709 ]
 [ -1.86346354   7.8459709 ]
 [  1.02813162   7.8459709 ]]
```



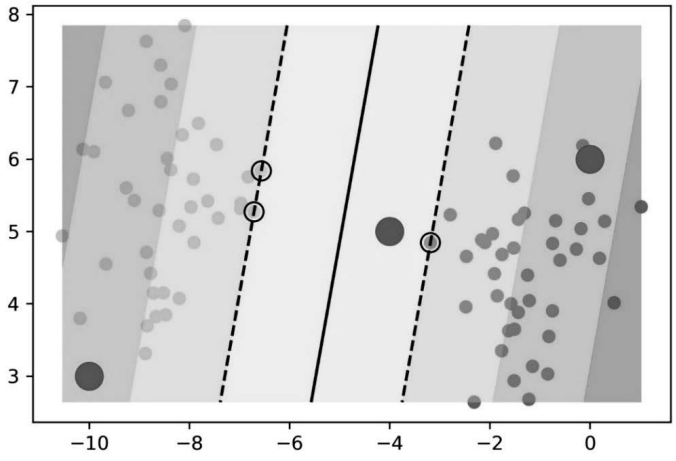
模型預測

用 scikit-learn 的支持向量機進行預測時，僅需用 ***predict*** 函式並傳入一個含有大量資料點的二維陣列即可。以下三筆資料的分類可得其中兩個點為1號分類（深色點），一個點為0號分類（淺色點）。將三個資料點（大點）放在座標上，可以發現三個點的確在分類線上的對應側。特別注意中間的點落在兩個點群之間，但是因為其在分類線右側，故和深色點群歸為同一類。

```
# 分類模型
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
plt.contour(grid_x, grid_y, z, colors='k',
            levels=[-1, 0, 1], linestyles=['--', '-', '--'])
plt.contourf(grid_x, grid_y, z, alpha=0.5, cmap=plt.cm.coolwarm)
plt.scatter(clf.support_vectors_[:, 0],
            clf.support_vectors_[:, 1],
            s=100, facecolors='none', edgecolors='k')
# 預測結果
test_points = np.array([
    [-4, 5],
    [0, 6],
    [-10, 3]])
print(clf.predict(test_points))
plt.scatter(test_points[:, 0], test_points[:, 1], s=200, color='green')
plt.show()
```

輸出

```
[1 1 0]
```

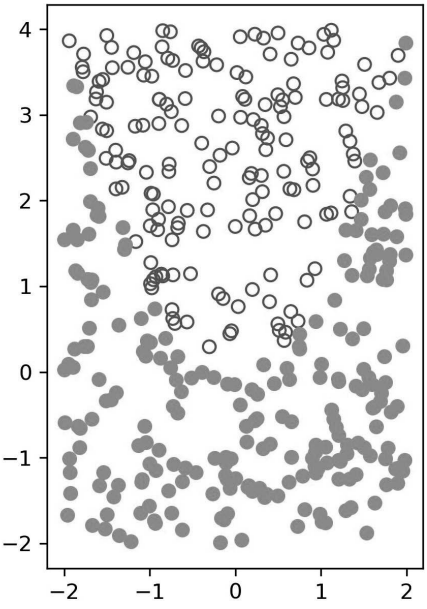


人的運算能力可能輸給電腦，但思考能力遠勝於電腦，模擬大腦結構進行機器學習會不會是個好方法？

機器學習剛出現時，有人提出了這樣的想法，建立模擬動物神經元傳遞並整合資訊的**人工神經網路(Artificial Neural Network，簡稱 ANN)**，取得輸入後並輸出一個最後的分類結果，概念如上方的圖片。不過如同人類身上有數億個神經細胞，運用大量參數的 ANN，因為所需運算資源太大而遇到一些發展的瓶頸。直到近年電腦硬體能力有顯著成長，模擬生物神經系統的機器學習再度被提出來，發展為近期相當熱門的**深度學習（Deep Learning）**，本書後半段也會針對這個部分多做討論。

4.6 習題

( ) 1. 下圖的二維資料分布若要使用 SVM 進行二元分類，應該用核方法把座標進行什麼樣的轉換比較恰當？



(a) $(x, y) \rightarrow (x, y, x^2 + y^2)$

(b) $(x, y) \rightarrow (x, y, x^2)$

(c) $(x, y) \rightarrow (x, y, x^3)$

(d) 不須使用核方法，直接輸入模型即可

( ) 2. KNN 演算法中，假設資料筆數是N，多數資料科學家所推薦的k值是多少？

(a) N (b) $\frac{N}{10}$

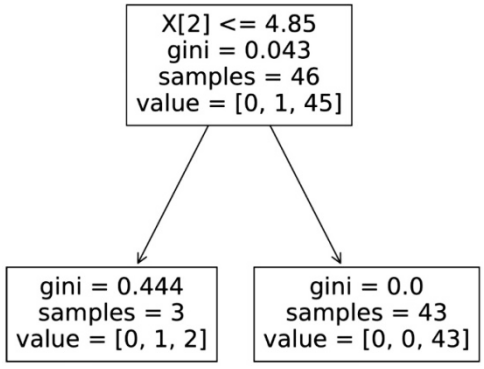
(c) $\sqrt{N}$ (d) $\frac{N}{\ln N}$

( ) 3. 決策樹的二分條件是怎麼決定的？

(a) 計算吉尼不純度 (b) 人為定義標準

(c) 以上皆是 (d) 隨機決定

( ) 4. 試計算這個決策樹分群的吉尼增量。



(a) 0.028 (b) 0.043

(c) 0.014 (d) 0.021

( ) 5. scikit-learn 套件的模型中以哪一個函式進行測試資料的預測？

(a) fit() (b) test()

(c) try() (d) predict()

( ) 6. 下列的分類演算法中，何者需要最多的運算資源？

(a) 神經網路 (b) 決策樹

(c) 支持向量機 (d) KNN

6.1 全連接神經網路

6.1.1 人工神經網路的發想

人類的神經系統

先把電腦擺一邊，想一下我們人類或其他動物，平常怎麼應對外界環境變化呢？

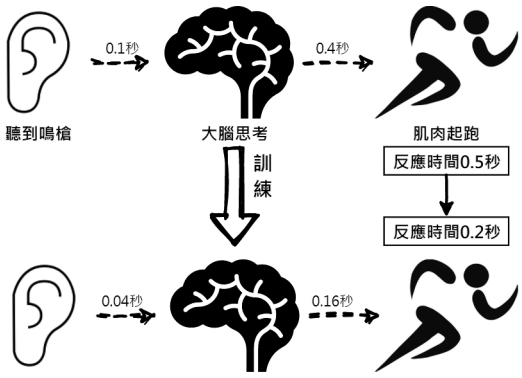
動物的神經系統，可以透過皮膚或五官感知外界環境，由大腦綜合外界資訊進行決策，再透過肌肉或腺體做出反應。主要進行這三種功能的組織或器官，分別就是受器、中樞與動器，其概念如圖所示。

從人工智慧草創的 1950 年代開始，有些計算機科學家試著模擬一個抽象的神經網路，他們嘗試模仿動物的神經系統，設計了輸入層、隱藏層和輸出層，分別對應受器、中樞與動器。這樣的架構奠定了至今人工神經網路（Artificial Neural Network，簡稱 ANN）的基礎。

赫布理論

神經系統或許可以仿造，但這代表 ANN 具有可行性嗎？他們當然有更多立論基礎。再往回推 10 年的 1940 年代末期，加拿大心理學家赫布（D. Hebb）提出了赫布理論，其中描述了突觸的可塑性。神經元間藉由突觸來接收和發出訊號，突觸的可塑性說明的是，當一個神經元在訊號傳遞時，不斷地刺激另一個神經元，兩個神經元突觸間的傳遞效能也會提高。

世界上每個人的才能各有不同，就是突觸可塑性的例子，例如短跑選手可能會因為大量訓練起跑，使得聽覺神經、腿部肌肉或大腦運動區的活躍度會比其他神經細胞來得強，以加速其反應時間。



人剛出生時可能各類細胞的活躍度都差不多，但經過後天訓練會有各種面向的發展。這不只代表神經網路遇到錯誤可以進行自我修正，也因為這樣讓我們可以讓一個神經網路以想要的功能運作。

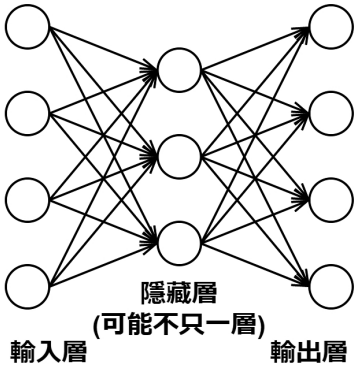
6.1.2 神經網路架構和運算

「層」的運算

在講自我修正等關鍵技術之前，我們先來認識 ANN 的架構和基本運算，首先要介紹神經網路「層」的概念。

人體全身上下大約有 860 億個神經元，來應對生活中大大小小的事，所以一個 ANN 的虛擬神經元個數自然也不會太少。動物的神經系統是用網狀架構相互連接，但套用在電腦運算上，我們需要對每個神經元個別處理，整體實作上會非常複雜。

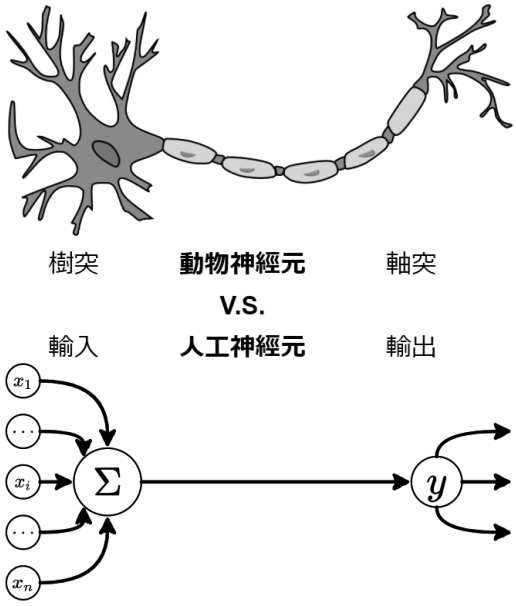
因此在基本的 ANN 中，是以一層一層的神經元作為建構單位，每一層的神經元，只會連接到它上一層的所有神經元，以及它下一層的所有神經元；其中 ANN 的第一層和最後一層，分別就是與外界交流資訊的**輸入層**（input layer）和**輸出層**（output layer），中間則是只跟上下層進行數學運算的**隱藏層**（hidden layer）。這樣基本的 ANN，因為每個神經元和它上下兩層的所有神經元，因此又被稱為**全連結神經網路**（Fully Connected Neural Network，**簡稱** FCNN），其概念如圖所示。



神經元的運算

實際的數學運算呢？

我們回顧一下小時候生物課學過的內容。一個神經系統的基本單位是神經元，它的基本架構如圖所示，除了中心的細胞核，還有許多突觸。這些突觸又分兩種－－接收訊號的樹突和發出訊號的軸突。

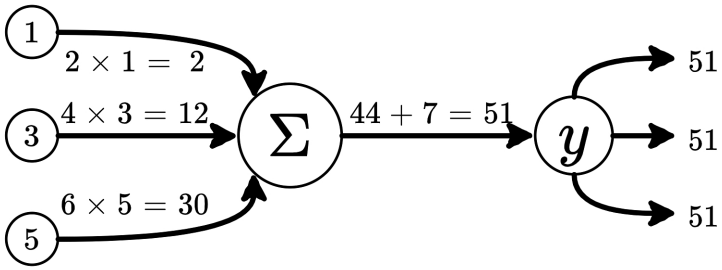


我們要怎麼模擬一個神經元呢？我們可以把神經元想像成一個函數，接收前一層n個神經元的輸出$x_1, x_2, \ldots, x_n$，計算後輸出y傳遞到下一層。一個神經網路中有大量的神經元，太複雜的運算會影響整體效率，所以應該盡量簡化單一神經元的運算。

神經元的運算有兩個大原則——每個輸入對輸出的影響力不同，以及神經元的基本輸出不一定是0。能做到這兩點的最簡單算法，就是加權運算。我們將n個輸入賦予權重$w_1, w_2, \ldots, w_n$，並假設基本輸出的偏差值（bias）為b，則一個神經元的運算就能寫成下列的式子。如果把權重和輸入表示成向量的話，又可以更簡單地寫成內積的形式。

$$y = \sum_{i=1}^{n}(w_i x_i) + b = \vec{w} \cdot \vec{x} + b$$

舉個例子，假設今天某個神經元有三個輸入分別是1,3,5，對應的權重是2,4,6，偏差值是7，則這個神經元的輸出$y = \vec{w} \cdot \vec{x} + b = (1,3,5) \cdot (2,4,6) + 7 = 51$。



運算的矩陣表示

神經元的運算或許容易，但神經網路層之間的運算呢？

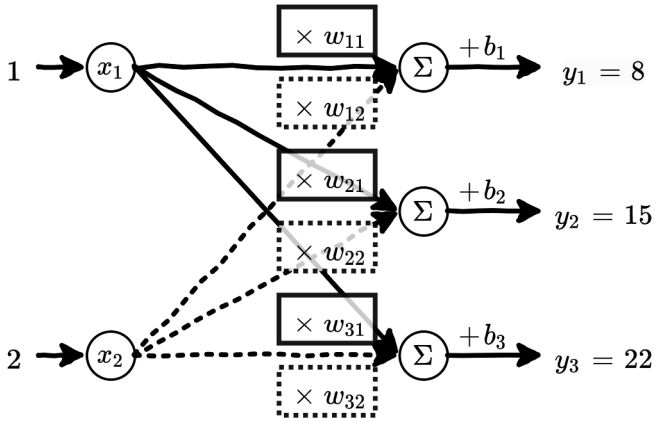
假定現在有兩個相互連接的神經網路層，前後兩層分別有n個和m個神經元。我們可以令x_i是前層第i個神經元的輸入，y_j是後層第j個神經元的輸出，W_{ji}是前層第i個神經元連接到後層第j個神經元的權重。後層第j個神經元的輸出，就能像單一神經元的運算，寫成下列的算式。

$$y_j = \sum_{i=1}^{n}(W_{ji} x_i) + b_j = \overrightarrow{W_j} \cdot \vec{x} + b_j$$

因為同一層的神經元不會相互連接，同時運算並不衝突，所以我們就能用矩陣表示兩個神經網路層的運算。下列的式子，也能更簡潔地用$\vec{y} = W\vec{x} + \vec{b}$。這時，我們就能看到以層為單位建構神經網路在運算上的方便性。

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_j \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1n} \\ W_{21} & W_{22} & \cdots & W_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & W_{ij} & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ W_{m1} & W_{m2} & \cdots & W_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \\ \vdots \\ b_m \end{bmatrix}$$

實際來算一次，某兩層神經網路的權重、輸入及偏差值如下所示，則$W = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \vec{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \vec{b} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$，所以輸出$\vec{y} = W\vec{x} + \vec{b} = \begin{bmatrix} 8 \\ 15 \\ 22 \end{bmatrix}$。



理解基本的運算後，我們可以用 Python 和矩陣乘法實作基本的神經網路運算。

程式實作 Pytorch 基礎張量運算 ■ 檔案：ch6/ Pytorch_Tensor_Calculations.ipynb

深度學習通常不是一維的向量或二維的矩陣計算就能解決的，所以 Pytorch 等相關套件中，以張量（tensor）作為運算單位，*torch* 套件中所對應的類別為 *torch.Tensor*。以下我們會用一些基本的張量運算，熟悉幾個 Pytorch 中常用的函式。首先，我們匯入 torch 套件。

```
01  # 匯入 torch 套件
02  import torch
```

單一神經元的運算在前面有提到，一個神經元的運算結果可以看做兩個向量相乘再加上常數向量，即$\vec{w}\vec{x}+b$。把得到的數值傳入非線性的激勵函數（Activation Function） ***torch.sigmoid()***後，就會是最後輸出的數字。激勵函數在之後的章節會再進一步地介紹。

運用 ***torch.view()***這個函式，可以把張量重新排列成適合矩陣乘法的大小，再用 ***torch.mm()***函式完成矩陣乘法。

```
torch.manual_seed(3)  # 固定亂數種子，方便對照結果
X = torch.randn((1, 6))  # 宣告亂數矩陣(常態分布，平均為 0、標準差為 1)
W = torch.randn_like(X)  # 宣告與 X 同大小的亂數矩陣
B = torch.randn((1, 1))
W = W.view(6, 1)  # 將 W 大小轉換為(6, 1)

# 輸入乘權重加偏差，傳入激勵函數 sigmoid 得到輸出
Y = torch.sigmoid(torch.mm(X, W) + B)
print(Y)
```

輸出

```
tensor([[0.0566]])
```

在深度學習中，***torch.view()***常被用在張量的降維與升維上，也可以把其中一個維度設成***-1***，讓它隨運算張量的大小變動。這個部分我們在之後討論卷積神經網路時會再有進一步的運用。以下列了幾個將具有 60 個元素的張量變形的範例。

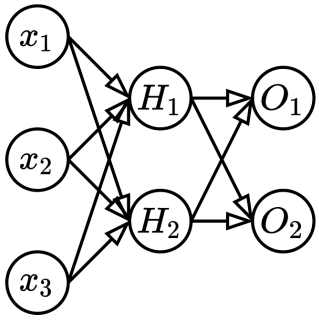
```
X = torch.randn((3, 4, 5))
print(X.shape)   # 原張量大小
print(X.view(3, 4, 5, 1).shape)
print(X.view(6, 10).shape)
print(X.view(3, 2, -1).shape)
```

輸出

```
torch.Size([3, 4, 5])
torch.Size([3, 4, 5, 1])
torch.Size([6, 10])
torch.Size([3, 2, 10])
```

多層全連結神經網路的運算

以全連結神經網路來看，只要決定了每層有幾個神經元，就會有對應的張量大小。相鄰的神經網路層間，參數張量W的長寬設成前層神經元數 × 後層神經元數，就能正常進行矩陣乘法。輸入張量$\vec{X}$乘上W，再加上與$W\vec{X}$相同大小的偏差張量$\vec{B}$，即完成一層神經元的運算。訓練時有時候會訓練不只一筆資料，因此可以傳入多一個維度的張量來計算一筆資料的輸出。



上圖是一個三層的神經網路，下面這段程式碼即是其所對應的正向傳播運算。

```
torch.manual_seed(3)
Data_Count = 2  # 資料量
N_Input = 3  # 單筆資料向量長(輸入層神經元數)
N_Hidden = 2  # 隱藏層神經元數
N_Output = 2  # 輸出層神經元數

X = torch.randn(Data_Count, N_Input)  # 輸入張量

W1 = torch.randn(N_Input, N_Hidden)  # 隱藏層權重張量
B1 = torch.randn((Data_Count, N_Hidden))  # 隱藏層偏差張量

W2 = torch.randn(N_Hidden, N_Output)  # 輸出層權重張量
B2 = torch.randn((Data_Count, N_Output))  # 輸出層偏差張量

Hidden = torch.sigmoid(torch.mm(X, W1) + B1)  # 輸入層傳至隱藏層
Output = torch.sigmoid(torch.mm(Hidden, W2) + B2)  # 隱藏層傳至輸出層

print(Output)
```

輸出

```
tensor([[0.5016, 0.2814],
        [0.7551, 0.6270]])
```