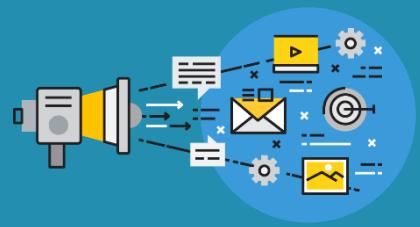


Algorithms and Data Analysis

-演算法與資料分析-

Machine Learning I

授課教師：張珀銀 老師

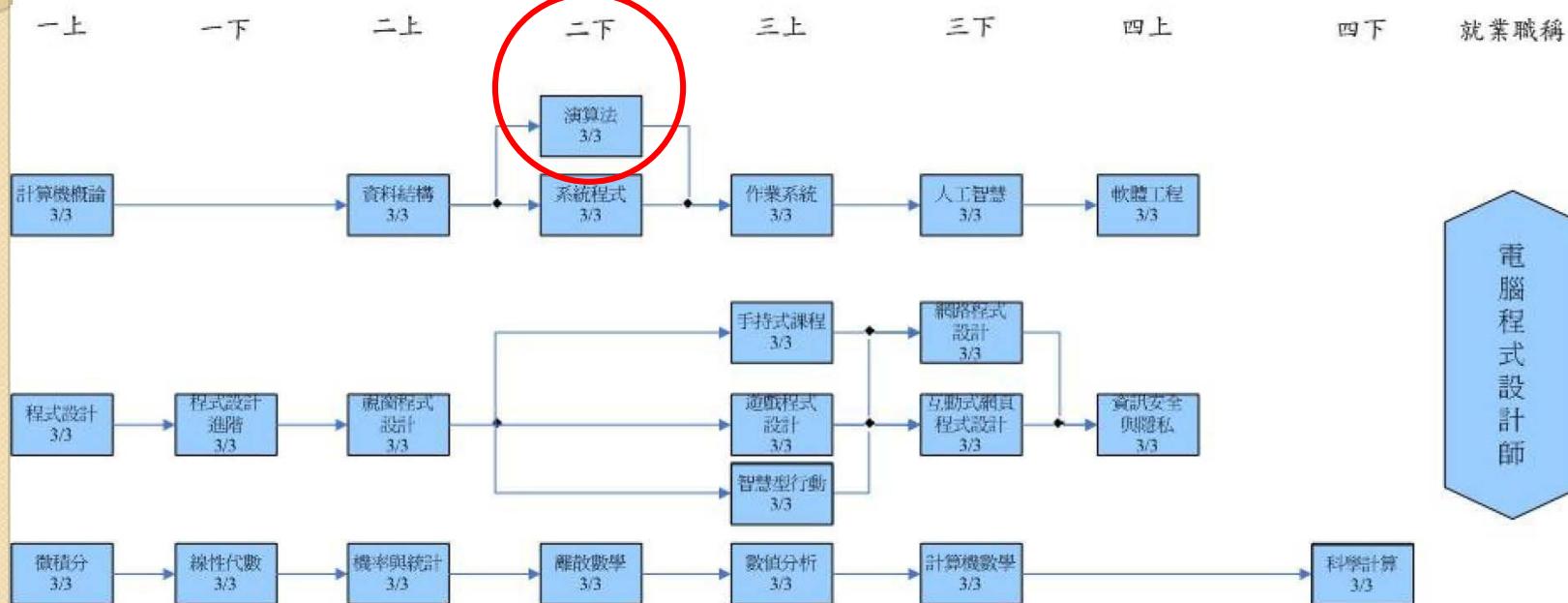


Outline

-
- ① AI 人工智能簡介
 - ② Machine Learning
 - ③ Decision Tree
 - ④ 資料結構回顧：程式與資料結構



課程學習路徑

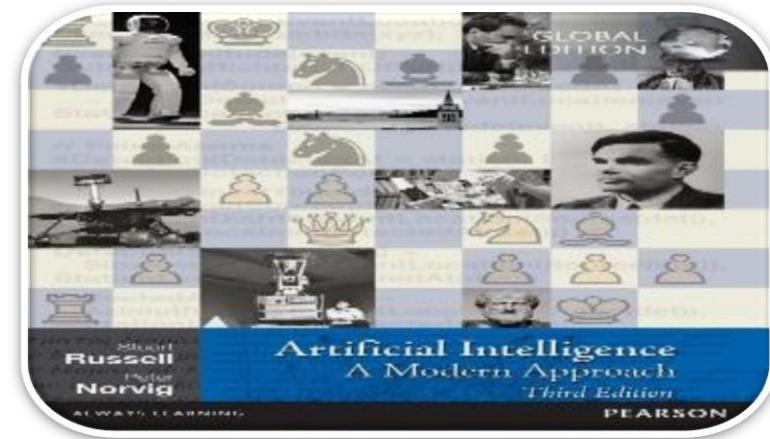


1

AI 人工智能簡介

人工智慧的定義(1/1)

- 令人覺得不可思議的電腦程式
- 與人類思考方式相似的電腦程式
- 與人類行為相似的電腦程式
- 會學習的電腦程式
- 根據對環境的感知做出合理行動，
獲致最大效益的電腦程式

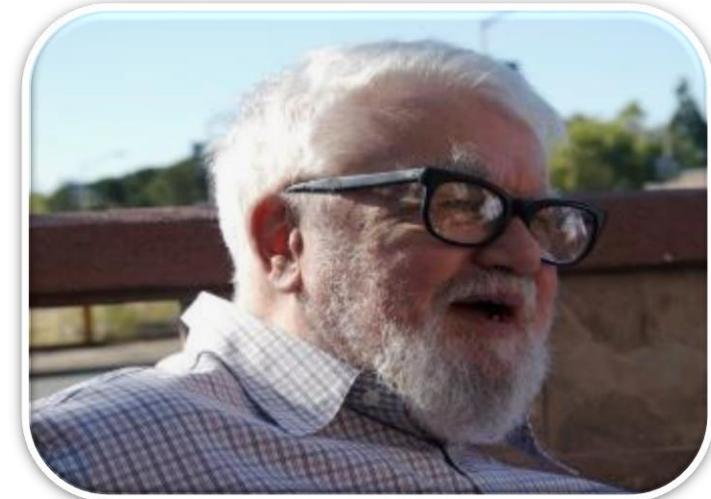


人工智能是有關「智慧主體研究與設計」的學問，而「智慧主體是指一個可以觀察周遭環境，並且採取行動以達成目標的系統」
(Artificial Intelligence: A Modern Approach, Peter Norvig)

資料來源：人工智慧來了

人工智慧的定義(1/2)

- 1955年，麻省理工學院的約翰·麥卡錫在達特茅斯會議上提出
 - 人工智能就是要讓機器的行為看起來就像是人所表現出的智慧行為一樣
- 由人工創造的系統所表現出的智慧
- 電腦系統具有學習、推理判斷及記憶的能力
- 能以人類的思維思考和推理 (強人工智能)



人工智能之父

約翰·麥卡錫 John McCarthy

- 有人曾經問他，創造一個機器，能達到人的智慧要多久？
- 麥卡錫給了一個很妙的答案，5年到500年！

達特茅斯會議

Dartmouth Summer Research Project on Artificial Intelligence



50年後的重聚：Trenchard More、John McCarthy、Marvin Minsky、Oliver Selfridge、Ray Solomonoff

主要議題

- 自動計算機
- 如何為計算機編程使其能夠使用語言
- 神經網絡
- 計算規模理論
- 自我改造
- 抽象
- 隨機性與創造性

資料來源：Wikipedia

人類智慧與人工智慧之差異

人類智慧	人工智慧
<ul style="list-style-type: none">• 具創造性• 具適應性• 包容較廣泛的事物• 容易遺忘• 不易轉移• 無法預測• 昂貴	<ul style="list-style-type: none">• 不會自動產生靈感• 被告知才會改變• 只能著重在特定幾點• 可長期擁有• 一致性高• 便宜

人類智慧與人工智慧之差異

分類	屬性	人 類	機 器
硬體	處理速度	最大 200 週期/秒	超過 20億 週期/秒
	互相連接速度	~ 120 公尺/秒	光速
	大小 / 儲存容量	一個頭顱 / 事件夠大思考會更慢	可不斷地擴展短期、長期及工作記憶體 / 並能「錯誤偵測」和「錯誤更正」
	可靠度 / 耐用度	<ul style="list-style-type: none"> • 容易疲勞 • 會隨著時間惡化 	<ul style="list-style-type: none"> • 更精準的類神經元 • 可修複或更換 • 可24小時不間斷工作
軟體	程式化	人腦「無法更新」	設計適合規則，可最佳化、可改善的、可解決的
	集體	因為廣泛的集體智慧，使人類成為最優秀的物種	所有電腦能在單一問題上一起工作，同質性高
整體	自我改善	???	Yes

強人工智慧與弱人工智慧

美國哲學家約翰. 瑟爾 (John Searle) 便提出

- 強人工智慧 (Strong A.I.)
- 弱人工智慧 (Weak A.I.)

弱人工智能

- 主張機器只能模擬人類具有思維的行為表現，而不是真正懂得思考。他們認為機器僅能模擬人類，並不具意識、也不理解動作本身意義。
- 若有一隻鸚鵡被訓練到能回答人類所有的問題，並不代表鸚鵡本身瞭解問題本身與答案的意義。

強人工智能

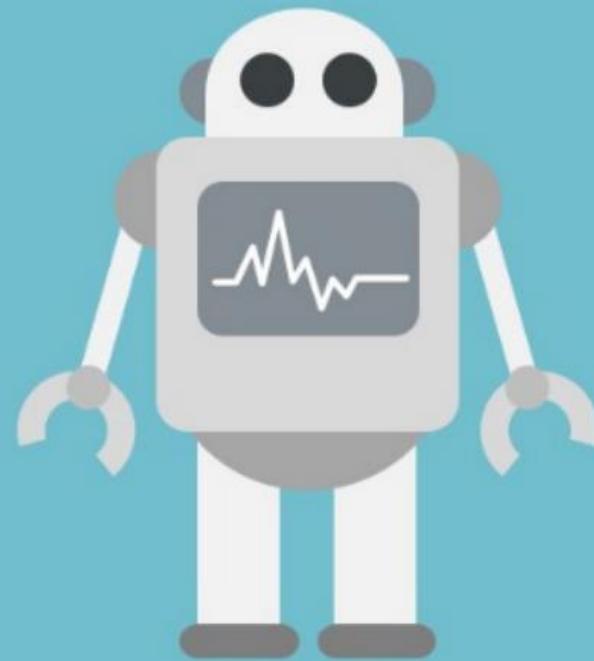
- 有可能製造出真正能推理和解決問題的智慧機器，且這樣的機器能將被認為是有知覺的，有自我意識的。強人工智能可以有兩類：
 - ◆ 類人的人工智慧，即機器的思考和推理就像人的思維一樣。
 - ◆ 非類人的人工智慧，即機器產生了和人完全不一樣的知覺和意識，使用和人完全不一樣的推理方式。

人工智慧的分類

	描述	思考速度	思考質量
弱人工智慧 Artificial Narrow Intelligence (ANI)	僅有單一方面的AI，如AI導航，僅有單一功能	超越人類	不及人類
強人工智慧 Artificial General Intelligence (AGI)	人類等級的AI，各方面與人類接近，包括思考質量	超越人類	接近人類
超級人工智慧 Artificial Super Intelligence (ASI)	在幾乎所有領域都比最聰明的人類大腦都聰明很多的AI	超越人類	超越人類

Artificial Intelligence

人工智能



1950's

Machine Learning

機器學習



1980's

Deep Learning

深度學習



2010's

機器學習 (Machine learning)

- 機器學習 (ML) 是對演算法和統計模型的科學研究，電腦系統使用這些演算法和統計模型來執行特定任務，而無需使用明確的指令，而是依靠模式和推理。它被視為人工智慧的子集。
- 機器學習演算法基於樣本資料（稱為“訓練資料”）建立數學模型，以便進行預測或決策而無需明確地程式設計以執行任務。
- 因為在開發各種應用程式的情境中，例如電子郵件過濾和電腦視覺，難以開發出有效執行任務的規則式演算法。

李開復：AI將改變世界，你準備好了嗎？(2:01)



資料來源：<https://www.youtube.com/watch?v=-xi2oYjZog>

2

決策樹(Decision Tree)

決策樹定義

決策樹的主要功能，是藉由分類已知的樣本來建立一個樹狀結構，並從中歸納出樣本裡的某些規則

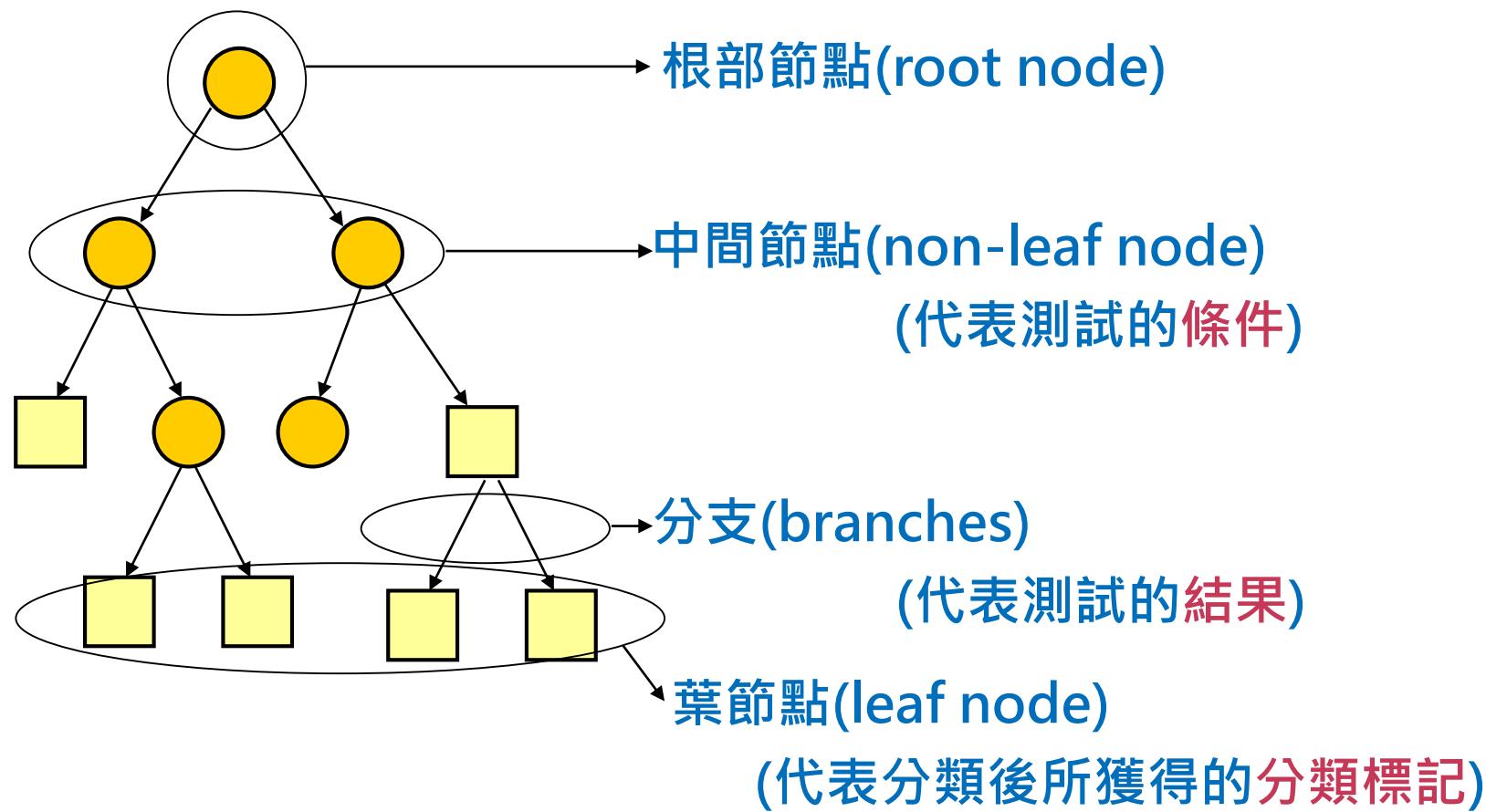
將範例 (Instance) 從根節點排列到某個葉子節點來分類範例

葉子節點即為範例所屬的分類

樹上每個內部節點說明了對範例的某個屬性的測試

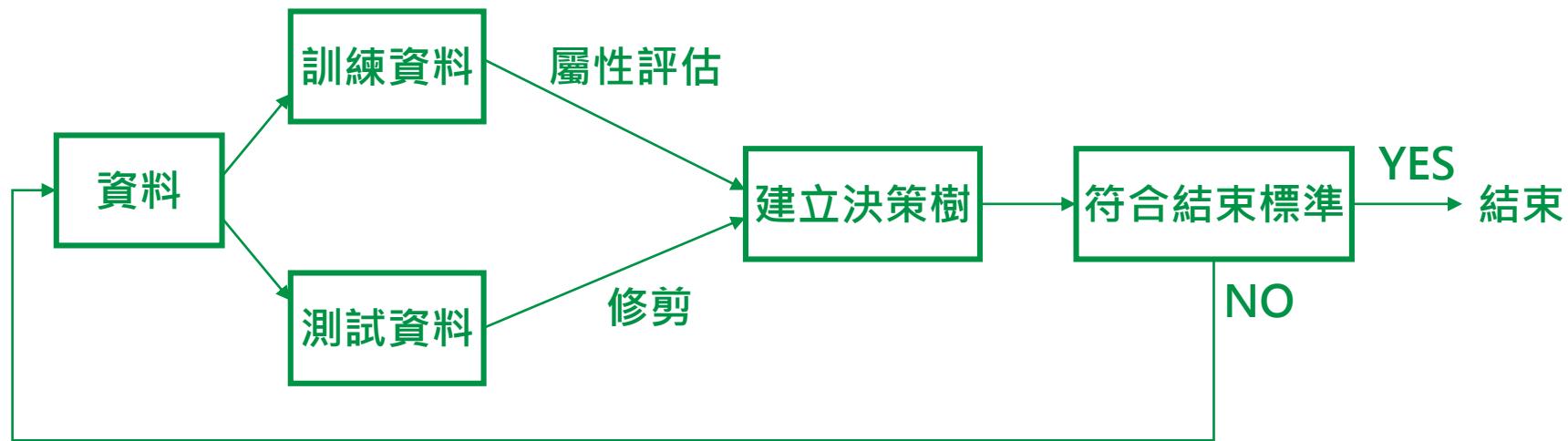
節點的每個後繼分支對應於該屬性的一個可能值

決策樹之樹狀結構

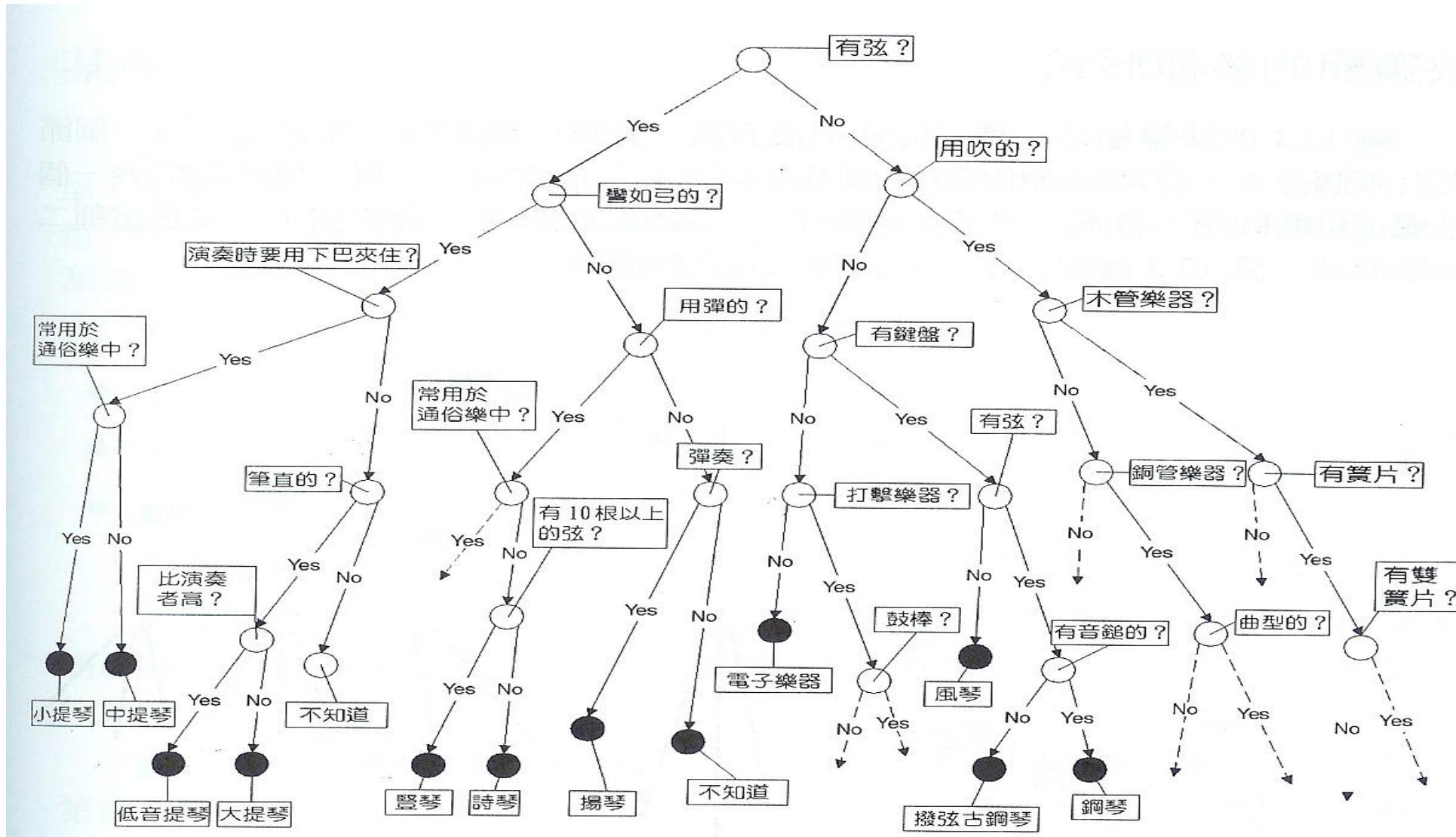


基本的演算法概念

- 直到所有的新內部節點都是樹葉節點為止，且：
 - 該群資料中，每一筆資料都已經歸類在同一類別下
 - 該群資料中，已經沒有辦法再找到新的屬性來進行節點分割
 - 該群資料中，已經沒有任何尚未處理的資料

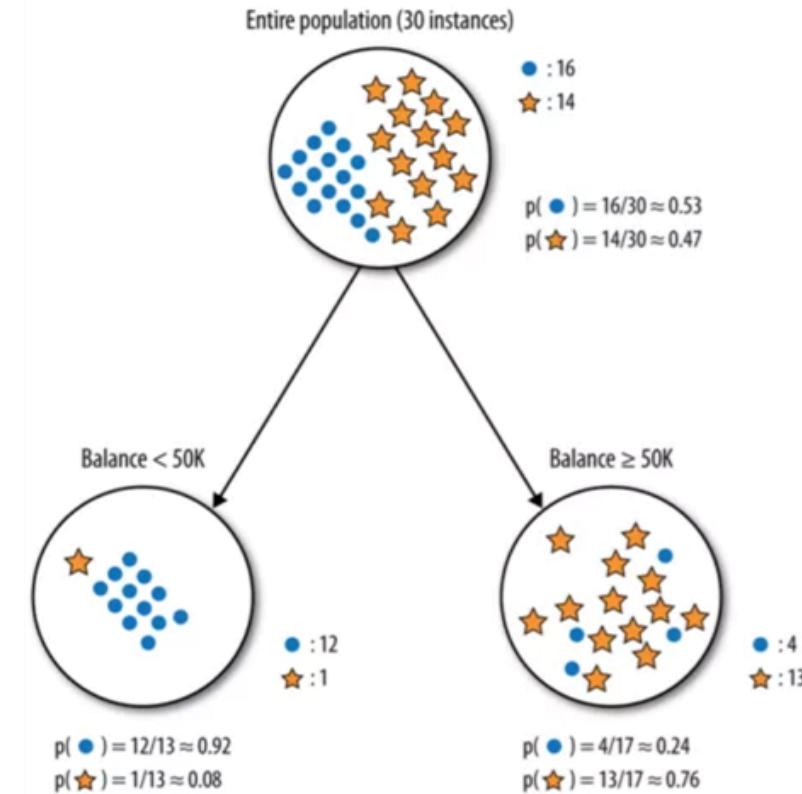
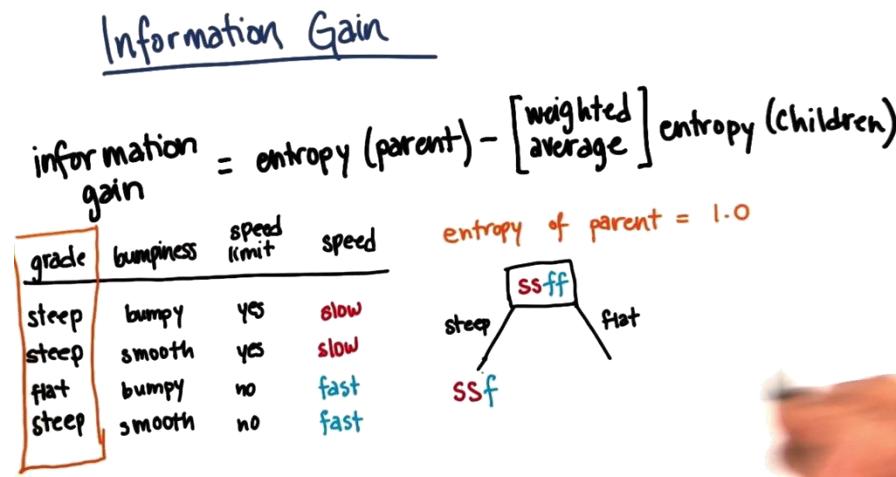


樂器分類的二分法決策樹



常用屬性選擇指標

- 資訊獲利 (Information Gain) – ID3、C4.5、C5.0
- 吉尼係數 (Gini Index) – CART
- χ^2 獨立性檢定 – CHAID



決策樹方法之比較

演算法	作者	資料屬性	分割規則	修剪樹規則
ID3	Quinlan (1979)	離散型資料	Entropy、Gain Ratio	Predicted Error Rate
C4.5	Quinlan (1993)	離散型資料	Gain Ratio	Predicted Error Rate
CHAID	Kass (1980)	離散型資料	Chi-Square Test	No Pruning
CART	Briemen (1984)	離散與連續型資料	Gini Index	Entire Error Rate

ID3演算法 (1/3)

- 決策樹最主要的原理是利用不同類別中資料分佈的差異性來做為分類的標準，因此，決策樹所擷取出的分類規則才能利用這些特性來分類新的資料
- 此外，決策樹的演算除了找出分類規則外，同時也將這些規則建立成樹狀結構
 - 利用此樹狀結構可以更快的利用已知的規則分類新的資料，而決策樹也就是這個分類探勘方法所產生的分類器

ID3演算法 (2/3)

- ID3是由頂端向下構建決策樹來進行學習的
- ID3計算過程
 - 具分類能力最好的屬性被選作樹的根節點
 - 根節點的每個可能值產生一個分支
 - 訓練範例排列到適當的分支
 - 重複上面的過程 (Recursive)

ID3演算法 (3/3)

- 核心問題是選取樹中的每個節點所要測試的屬性
- 建構決策樹過程中，以資訊增益(Information Gain)為準則，並選擇最大的資訊增益值作為分類屬性
- 熵(Entropy)，可當作資訊量的凌亂程度指標，當熵值愈大，則代表資訊的凌亂程度愈高，也代表期望資訊量越大
 - 計算公式：
 - 例：若丟了14次銅板，出現了9個正面與5個反面(記為 $[9+, 5-]$)，其熵為 $\text{Entropy}([9_+, 5_-]) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.94$

$$\text{Entropy}(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

ID3演算法 - 範例 (1/6)

特徵 (屬性)					目標 (類別)
已婚	信用卡 保險	性別	年齡	壽險促銷	
是	否	男	45	否	
是	否	女	40	是	
否	否	男	42	否	
是	是	男	43	是	
否	否	女	38	否	
是	否	女	55	是	
否	否	男	35	否	
否	是	男	27	否	
否	否	男	43	是	
是	否	女	41	否	
否	否	女	43	是	
是	否	女	29	否	
否	否	女	39	是	
否	否	男	55	否	
否	否	女	19	是	

ID3演算法 - 範例 (2/6)

特徵 (屬性)				目標 (類別)	
已婚	信用卡保險	性別	年齡	壽險	促銷
是	否	男	45	否	否
是	否	女	40	否	是
是	否	男	42	是	否
否	是	男	43	是	是
否	否	女	38	否	否
是	否	女	55	是	否
否	否	男	35	否	是
是	否	男	27	是	是
否	否	男	43	否	否
是	否	女	41	是	是
否	否	女	43	否	是
是	否	女	29	是	是
否	否	男	39	否	否
是	否	男	55	是	是
否	否	女	19	否	否

$$\text{Entropy}(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

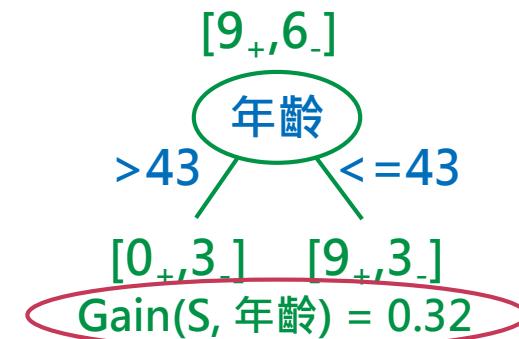
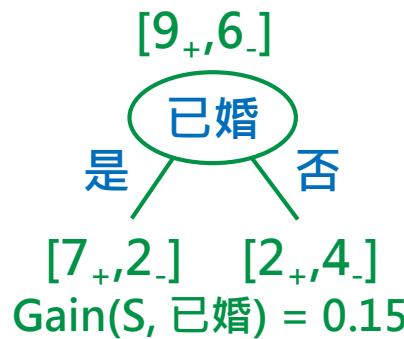
$$\text{Entropy}([9_+, 6_-]) = -(9/15) \log_2 (9/15) - (6/15) \log_2 (6/15) = 0.97$$



$$\text{Entropy}([3_+, 5_-]) = 0.95$$

$$\text{Entropy}([6_+, 1_-]) = 0.59$$

$$\text{Gain}(S, \text{性別}) = 0.97 - 0.95 * (8/15) - 0.59 * (7/15) = 0.19$$

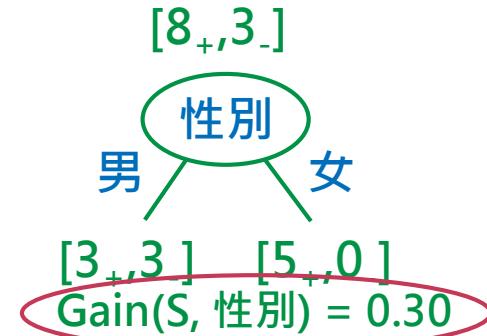
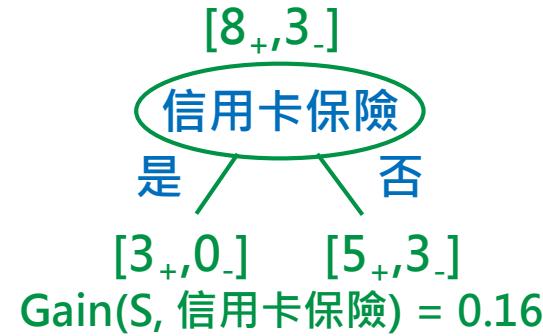
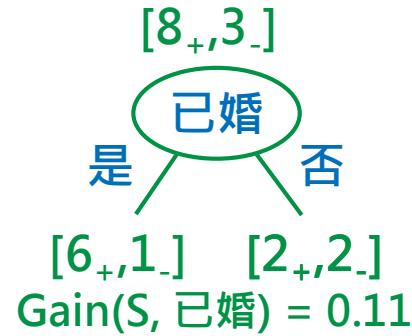


ID3演算法 - 範例 (3/6)

特徵 (屬性)				目標 (類別)
已婚	信用卡保險	性別	年齡	壽險促銷
是	否	男	45	否
否	否	女	40	是
是	是	男	42	否
否	否	女	43	是
是	是	男	38	否
否	否	女	55	是
是	否	男	35	否
否	否	女	27	是
否	是	男	43	否
是	否	女	41	是
否	否	男	43	否
是	否	女	29	是
否	否	男	39	否
是	是	女	55	是
否	否	男	19	否

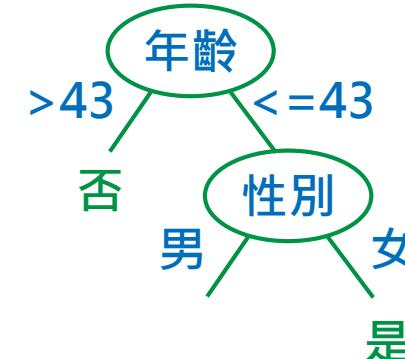


是

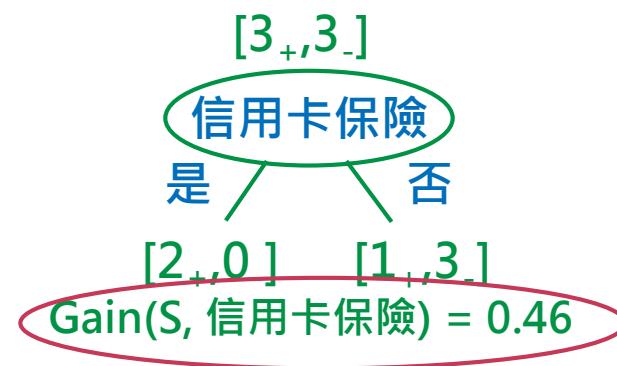
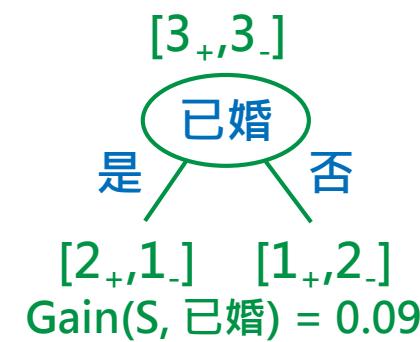


ID3演算法 - 範例 (4/6)

特徵 (屬性)				目標 (類別)
已婚	信用卡保險	性別	年齡	壽險促銷
是	否	男女	45	否
是	否	男	40	是
否	否	女	42	否
是	是	男	43	是
否	否	女	38	否
否	是	男	55	是
是	否	女	35	否
否	否	男	27	否
否	否	女	43	是
是	否	男	41	否
否	否	女	43	是
是	否	男	29	否
否	否	女	39	是
否	是	男	55	否
是	否	女	19	是



是



ID3演算法 - 範例 (5/6)

特徵 (屬性)				目標 (類別)
已婚	信用卡保險	性別	年齡	壽險促銷
是	否	男女	45	否
是	否	男	40	是
否	否	女	42	否
是	是	男	43	是
否	否	女	38	否
是	否	男	55	是
否	是	男	27	否
是	否	女	43	是
否	否	男	41	否
是	是	女	43	是
否	否	男	29	是
是	否	女	39	否
否	是	男	55	是
是	否	女	19	否

有可能造成分類錯誤!

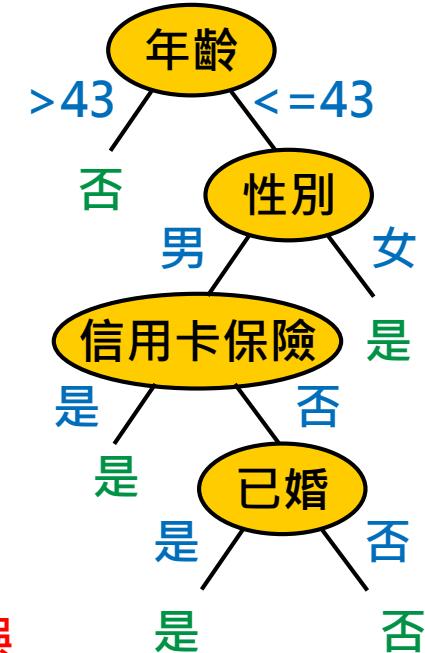


是 否

ID3演算法 - 範例 (6/6)

特徵 (屬性)				目標 (類別)
已婚	信用卡保險	性別	年齡	壽險促銷
是	否	男	45	否
否	否	女	40	是
是	是	男	42	否
否	否	男	43	是
是	否	女	38	否
否	是	女	55	是
是	否	男	35	否
否	否	男	27	是
是	是	男	43	否
否	否	男	41	是
是	否	女	43	是
否	是	男	29	是
是	否	女	39	否
否	否	男	55	是
是	否	女	19	是

分類錯誤



C4.5演算法 (1/2)

- C4.5演算法繼承了ID3 (Iterative Dichotomiser 3) 演算法的優點，並在以下幾方面對ID3演算法進行了改進：
 - 用資訊增益率 (Gain Ratio) 來評估分類屬性的強度，克服了用資訊增益選擇屬性時偏向選擇取值多的屬性的不足，例如學生ID編號，導致每個分支是單一的結果，變成沒有意義的決策樹
 - 能夠完成對連續屬性的離散化處理
 - 能夠對不完整資料進行處理
- 優點：產生的分類規則易於理解，準確率較高
- 缺點：在構造樹的過程中，需要對資料集進行多次的順序掃描和排序，因而導致演算法的低效

C4.5演算法 (2/2)

- 取Gain值仍然會有誤差，導致測試後會有多種結果
- C4.5演算法是ID3演算法的修訂版，採用**GainRatio**來加以改進方法，選取有最大GainRatio的分割變數作為準則，避免ID3演算法過度配適的問題

$$\text{Split-info}(X) = - \sum_{i=1}^n \left(\left(|T_i| / |T| \right) \log_2 \left(|T_i| / |T| \right) \right)$$

$$\text{Gain-ratio}(X) = \text{Gain}(X) / \text{Split-info}(X)$$

C4.5演算法 - 範例 (1/3)

特徵 (屬性)				目標 (類別)	
已婚	信用卡保險	性別	年齡	壽險	促銷
是	否	男	45	否	是
是否	否	女	40	是	否
是否	否	男	42	否	否
是否	是	女	43	是	是
是否	否	男	38	否	否
是否	否	女	55	是	否
是否	否	男	35	否	否
是否	否	男	27	是	是
是否	否	女	43	否	否
是否	否	男	41	是	是
是否	否	女	43	否	否
是否	否	男	29	是	是
是否	否	女	39	否	否
是否	否	男	55	是	是
是否	否	女	19	否	否

$$\text{Entropy}([9_+, 6_-]) = -(9/15)\log_2 (9/15) - (6/15) \log_2 (6/15) = 0.97$$

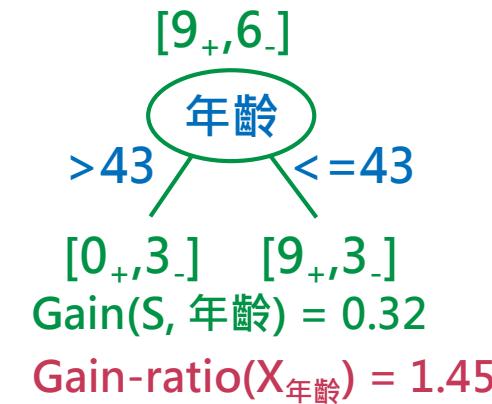
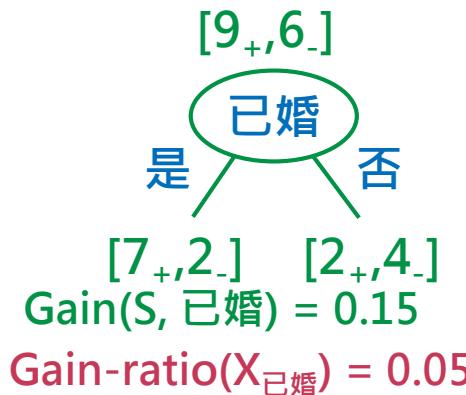


$$\text{Entropy}([3_+, 5_-]) = 0.95 \quad \text{Entropy}([6_+, 1_-]) = 0.59$$

$$\text{Gain}(S, \text{性別}) = 0.97 - 0.95*(8/15) - 0.59*(7/15) = 0.19$$

$$\text{Split-info}(X_{\text{性別}}) = -8/15\log_2(8/15) - 7/15\log_2(7/15) = 0.3$$

$$\text{Gain-ratio}(X_{\text{性別}}) = 0.19/0.3 = 0.63$$

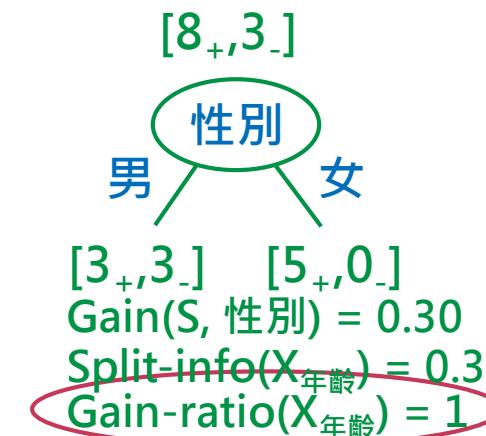
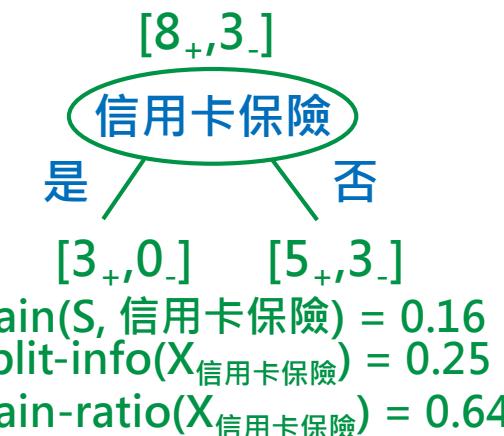
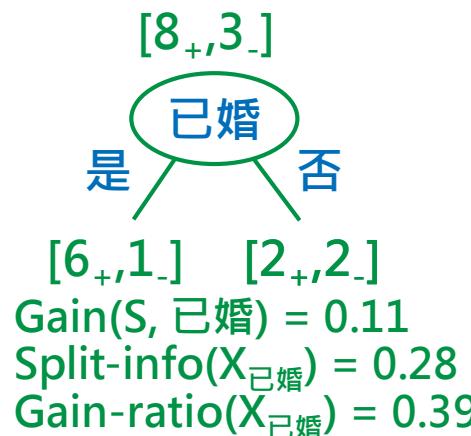


C4.5演算法 - 範例 (2/3)

特徵 (屬性)				目標 (類別)
已婚	信用卡保險	性別	年齡	壽險促銷
是	否	男	45	否
否	否	女	40	是
是	是	男	42	否
否	否	女	43	是
是	否	男	38	否
否	是	女	55	是
是	否	男	35	否
否	否	女	27	是
是	是	男	43	否
否	否	女	41	是
是	否	男	43	否
否	否	女	29	是
是	是	男	39	否
否	否	女	55	是
是	否	男	19	否
否	是	女		是

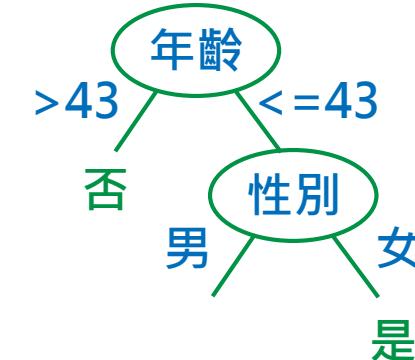
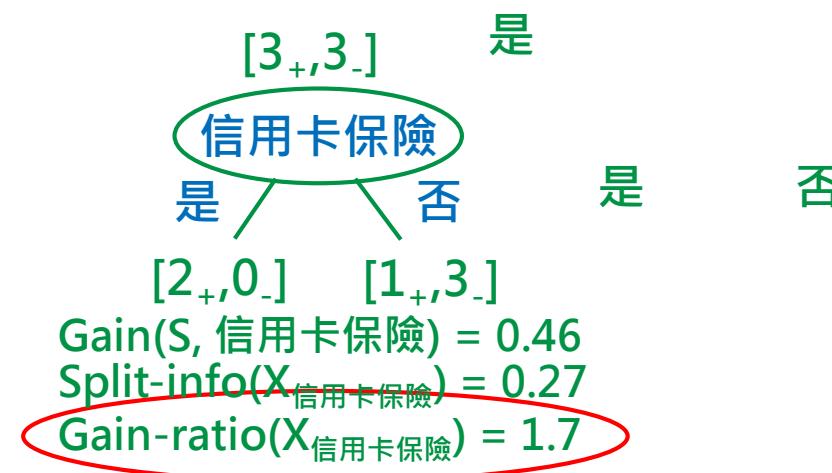
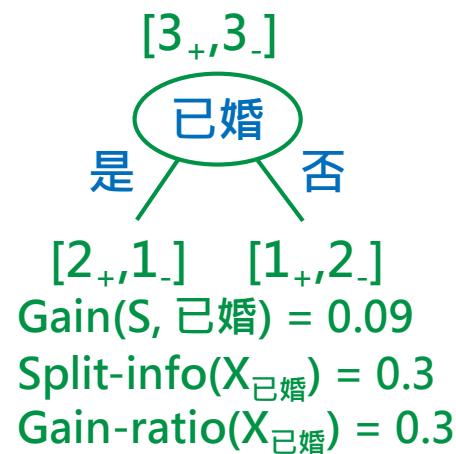


是



C4.5演算法 - 範例 (3/3)

特徵 (屬性)				目標 (類別)
已婚	信用卡保險	性別	年齡	壽險促銷
是	否	男	45	否
否	否	女	40	是
是	是	男	42	否
否	否	女	43	是
是	否	男	38	否
否	是	女	55	是
是	否	男	35	否
否	否	女	27	是
是	是	男	43	否
否	否	女	41	是
是	否	男	43	否
否	否	女	29	是
是	是	男	39	否
否	否	女	55	是
是	否	男	19	否
否	是	女		是



C5.0 演算法

- C5.0是C4.5的商業改進版，可應用於**海量資料**集合上之分類。主要在執行準確度和記憶體耗用方面做了改進。因其採用**Boosting**方式來提高模型準確率，且佔用系統資源與記憶體較少，所以計算速度較快
- 使用的演算法沒有被公開
- C5.0 的優點：
 - C5.0模型在面對遺漏值時非常穩定
 - C5.0模型不需要很長的訓練次數
 - C5.0模型比較其他類型的模型易於理解
 - C5.0的增強技術提高分類的精度

CART演算法

- CART : Classification and Regression Tree 分類與迴歸樹演算法
- 產生二元樹的技術，以吉尼係數(Gini index)做為選擇屬性的依據
- CART與ID3、C4.5、C5.0演算法的最大相異之處是，其在每一個節點上都是採用二分法，也就是一次只能夠有兩個子節點
- 計算上和ID3非常相似，只是評估函數替換，熵換成吉尼係數、資訊獲利換成吉尼獲利，並挑選獲利最大做分割

CART演算法

- 吉尼係數 Gini index
 - 假設資料集合 S 包含 n 個類別， 則吉尼係數 $Gini(S)$ 定義為：

$$Gini(S) = 1 - \sum_{n \in S} p_i^2$$

- 吉尼獲利 Gini gain
 - S 用屬性 A 來分割成數個 S_i ， 則吉尼獲利 $GiniGain(A, S)$ 可表示為：

$$GiniGain(A, S) = Gini(S) - Gini(A, S) = Gini(S) - \sum_{n \in S} \frac{|S_n|}{|S|} Gini(S_n)$$

CART演算法範例 (1/2)

假設我們想預測出喜歡打球類型的學生，從已知的訓練資料中，
系統有下列兩種分類方式如下，何者是較佳的分類呢？

以性別分類

學生人數 = 30
打球人數 = 15(50%)



女性



學生人數 = 10
打球人數 = 2 (20%)

男性



學生人數 = 20
打球人數 = 13 (65%)

以班級分類



Class A



學生人數 = 14
打球人數 = 6 (43%)

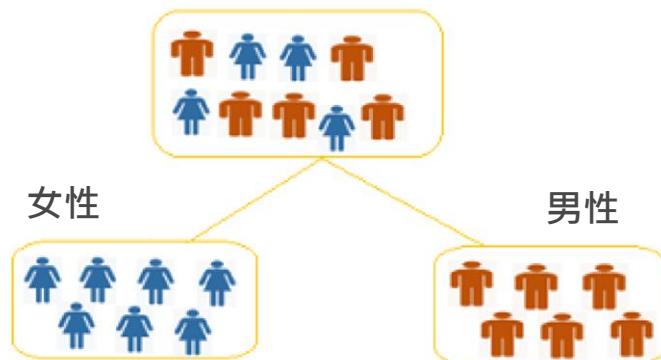
Class B



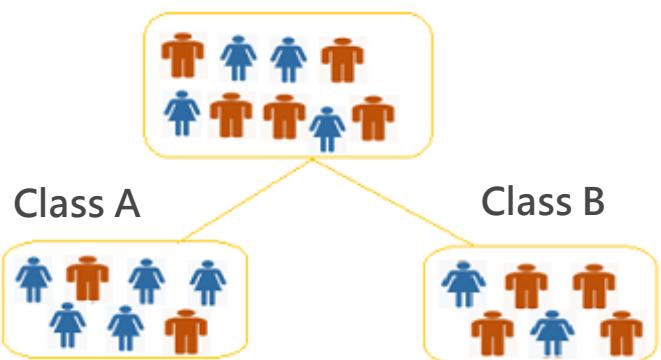
學生人數 = 16
打球人數 = 9 (56%)

CART演算法範例 (2/2)

以性別分類



以班級分類



Female節點：十位女性，其中有2位打球10位不打，
Gini係數為 $(0.2)^2 + (0.8)^2 = 0.68$

Male節點：20位男性，其中有13位打球7位不打，
Gini係數為 $(0.65)^2 + (0.35)^2 = 0.55$

Gini係數加權後為： $(10/30)*0.68 + (20/30)*0.55 = 0.59$

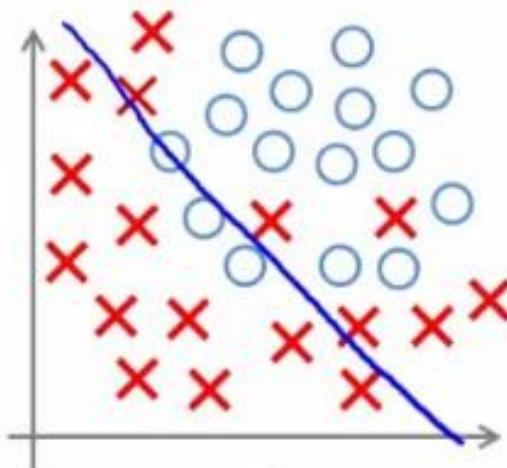
Class A節點：此班14位同學，其中6位打球8位不打，
因此Gini係數為 $(0.43)^2 + (0.57)^2 = 0.51$

Class B節點：此班16位同學，其中9位打球7位不打，
因此Gini係數為 $(0.56)^2 + (0.44)^2 = 0.51$

Gini係數加權結果： $(14/30)*0.51 + (16/30)*0.51 = 0.51$

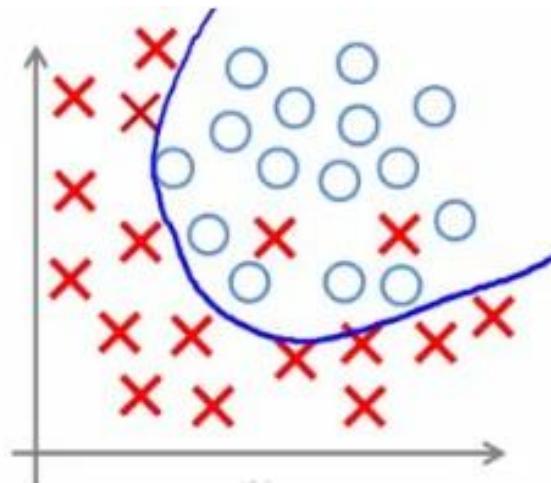
決策樹學習的常見問題

避免過度適配資料

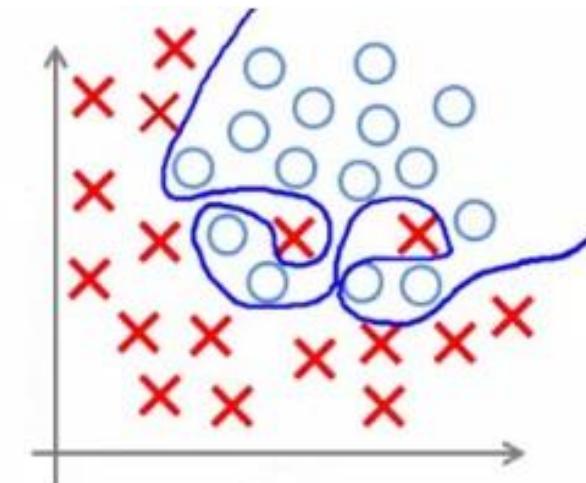


Under-fitting

Too simple to
explain the
variance



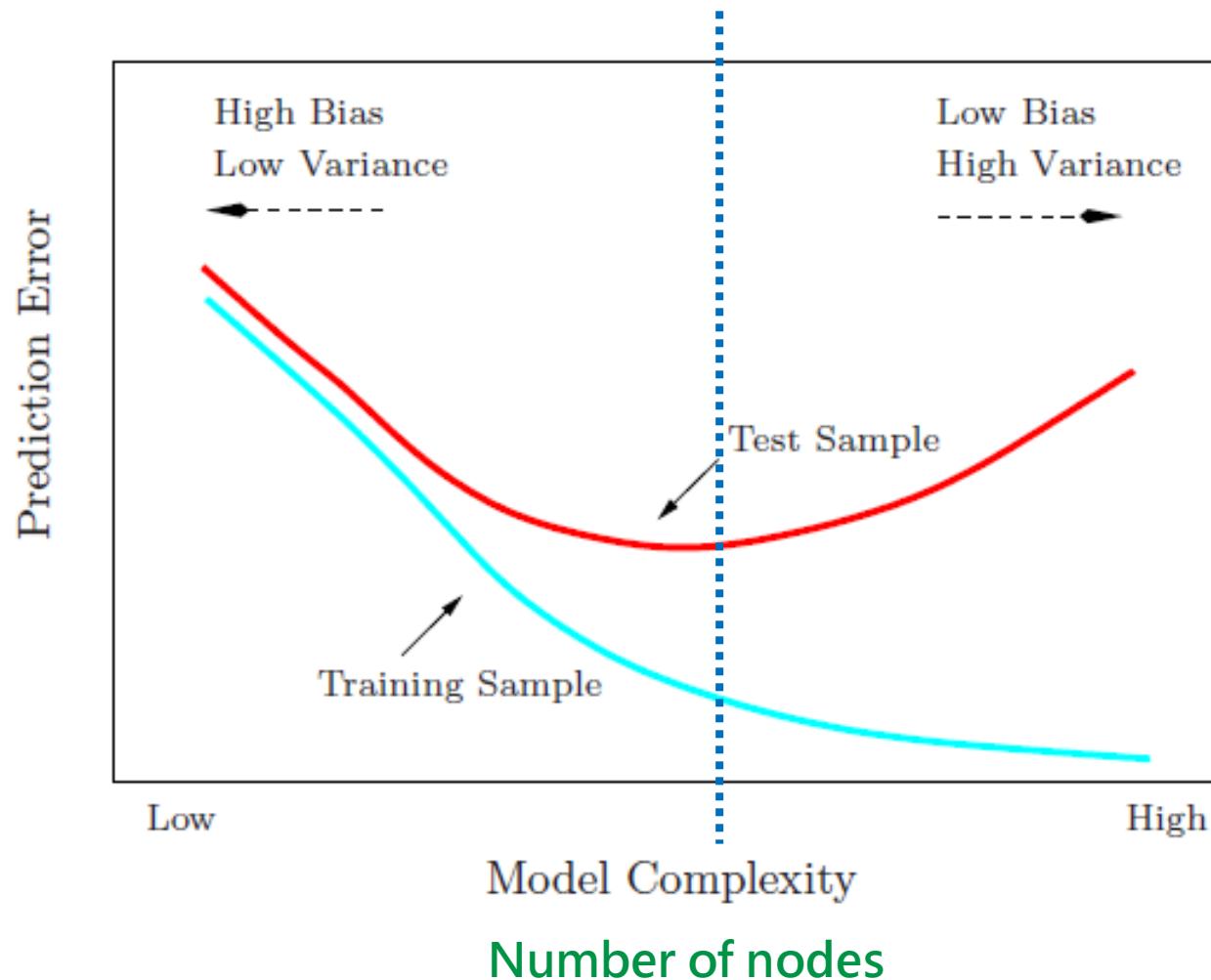
Appropriate-fitting



Over-fitting

forcefitting – too
good to be true

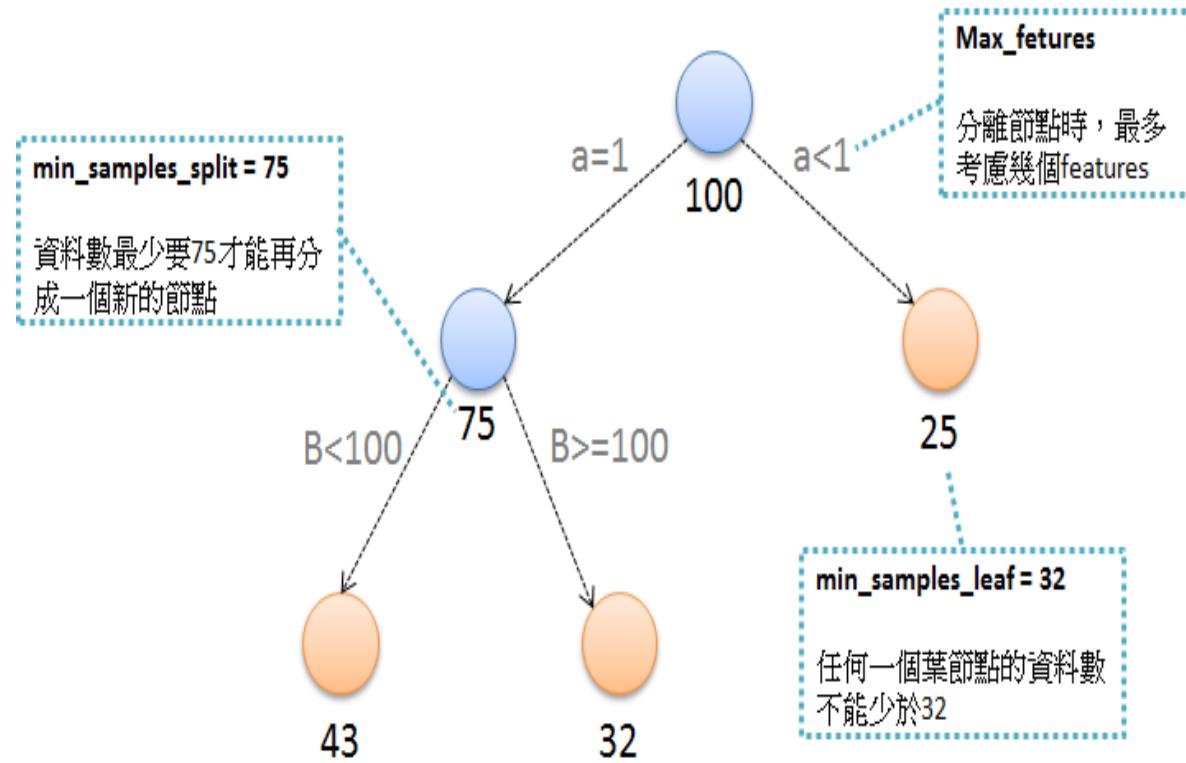
如何調整一般化錯誤(Generalization Error)



過度調適的情況之下，會有的幾個現象

1. 過度學習的結果容易造成複雜化
2. 決策樹不再提供好的預測水準
3. 需要一個新方法來估計預測誤差

設定生成限制



Minimum samples for a node split

資料數目不得小於多少才能再產生新節點

Minimum samples for a terminal node (leaf)

要成為葉節點，最少需要多少資料

Maximum depth of tree (vertical depth)

限制樹的高度最多幾層

Maximum number of terminal nodes

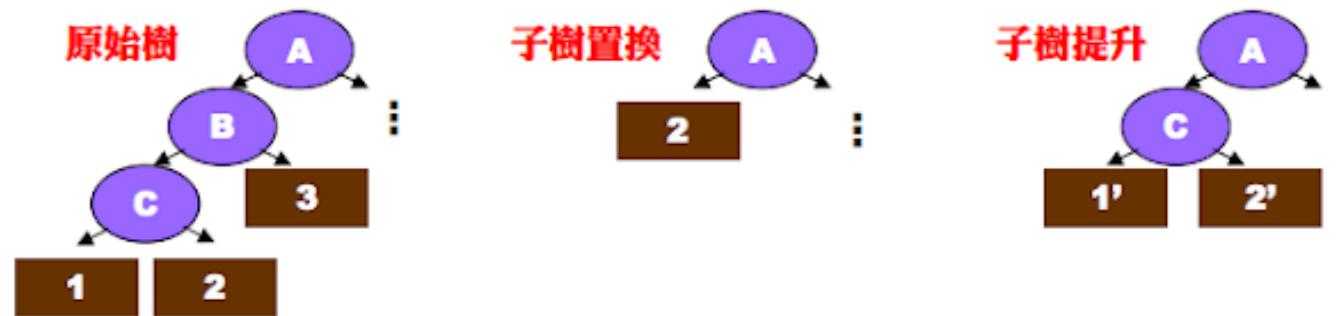
限制最終葉節點的數目

Maximum features to consider for split

在分離節點時，最多考慮幾種特徵值

剪枝 (Tree pruning)

- 事前剪枝 Pre-pruning
 - 透過決策樹不再增長的方式來達到修剪的目的
 - 選擇一個合適的臨界值往往很困難
- 後剪枝 Post-pruning
 - 子樹置換 (Subtree Replacement)
 - 子樹提升 (Subtree Raising)

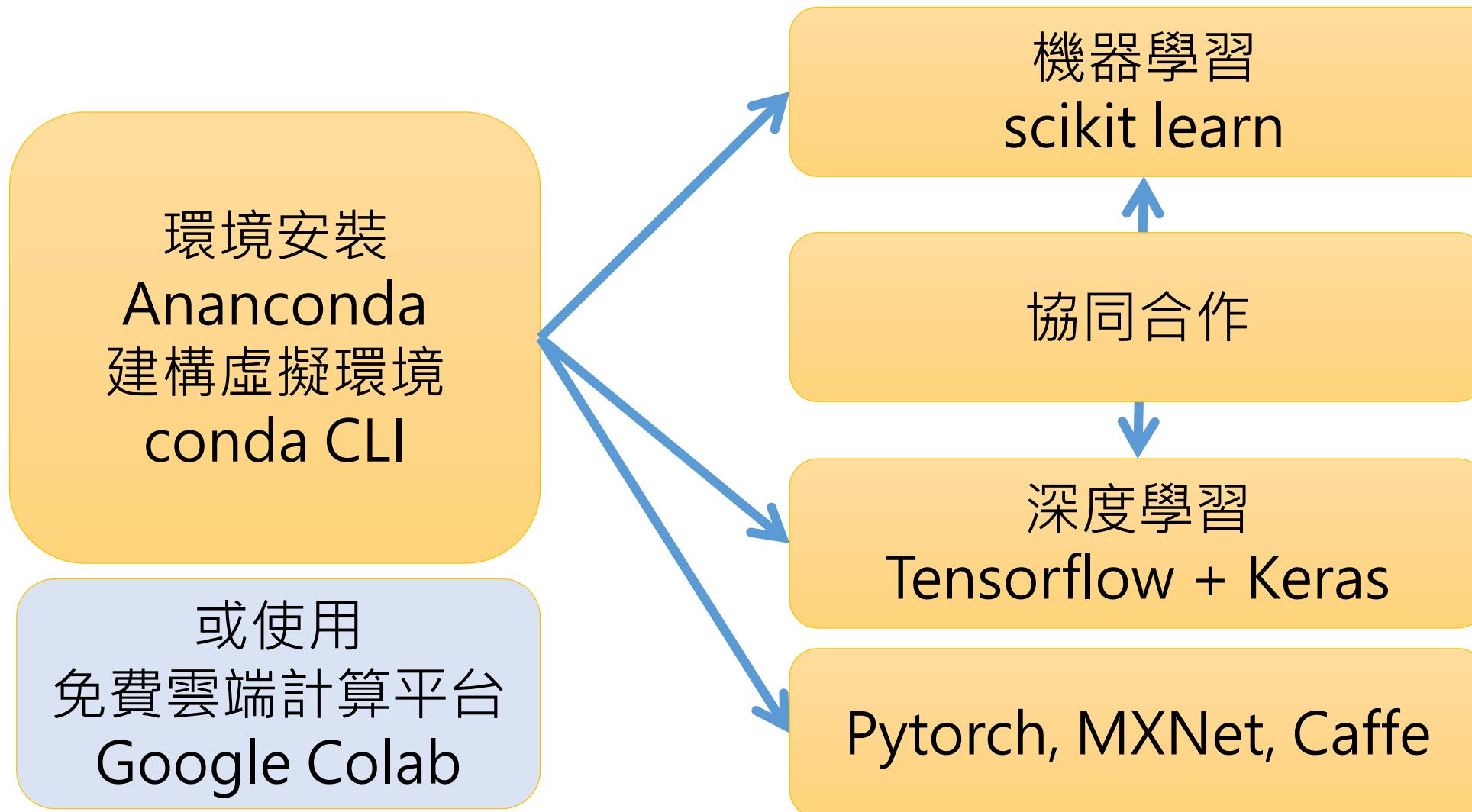




5

機器學習分析環境

建置分析環境



Python 安裝

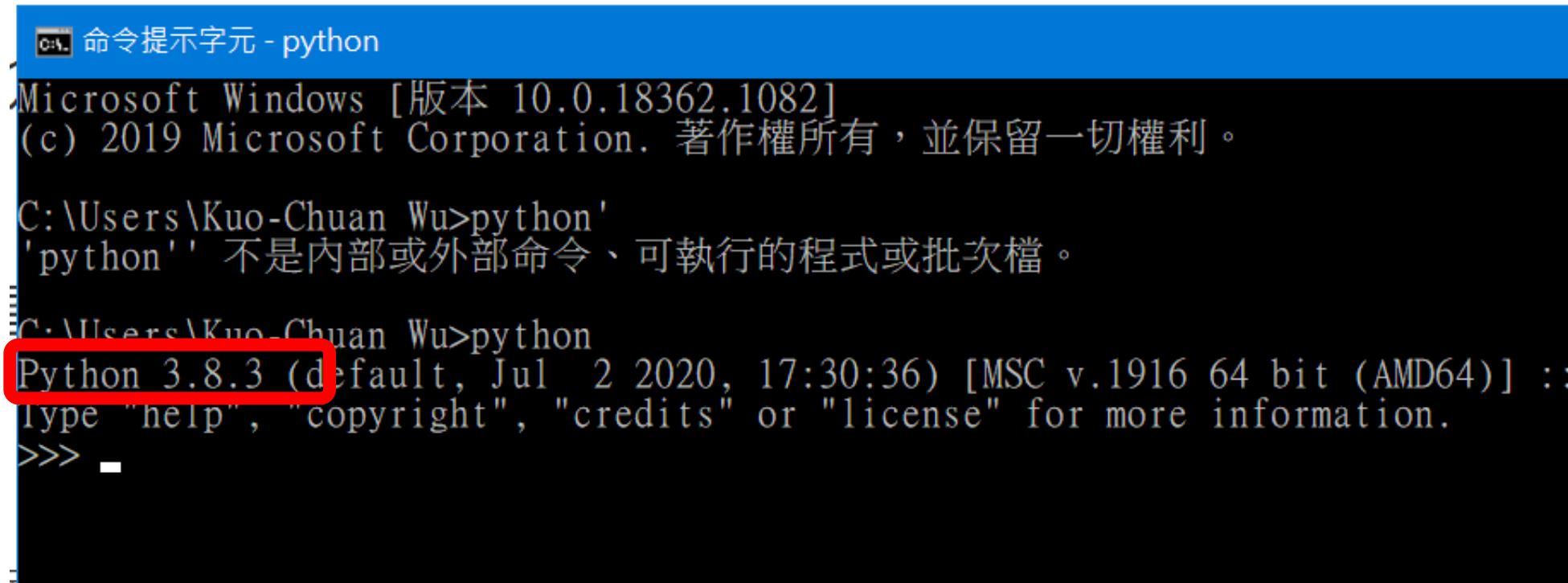
- 使用 Anaconda 套件安裝 Python

Anaconda 官網: <https://www.anaconda.com/download/>

Windows	MacOS	Linux
 Python 3.8 64-Bit Graphical Installer (466 MB) 32-Bit Graphical Installer (397 MB)	 Python 3.8 64-Bit Graphical Installer (462 MB) 64-Bit Command Line Installer (454 MB)	 Python 3.8 64-Bit (x86) Installer (550 MB) 64-Bit (Power8 and Power9) Installer (290 MB)

Python 安裝

- Anaconda 安裝好後，跟著下面的步驟來試運行 Python。

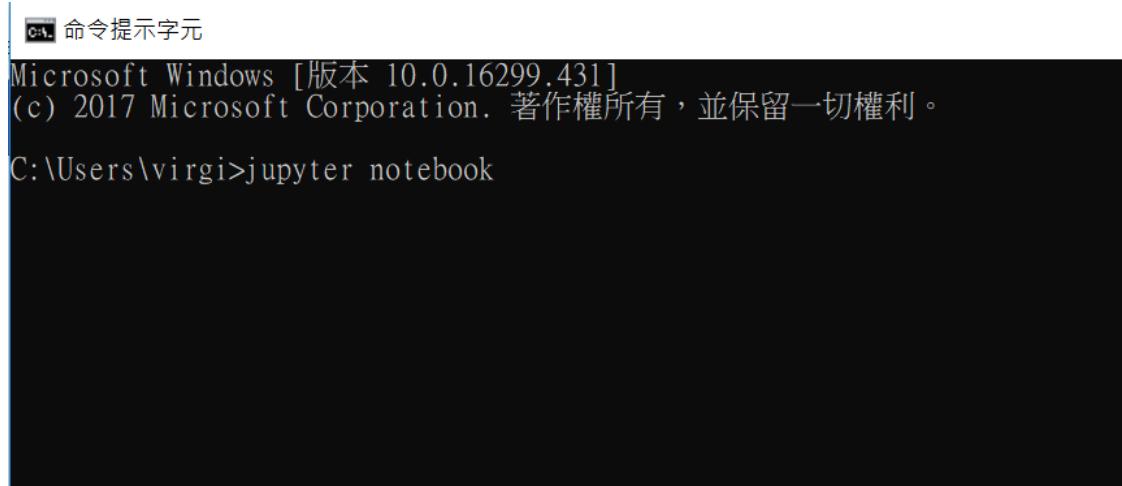


```
命令提示字元 - python
Microsoft Windows [版本 10.0.18362.1082]
(c) 2019 Microsoft Corporation. 著作權所有，並保留一切權利。
C:\Users\Kuo-Chuan Wu>python
'python' 不是內部或外部命令、可執行的程式或批次檔。
C:\Users\Kuo-Chuan Wu>python
Python 3.8.3 (default, Jul 2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)] ::::  
type "help", "copyright", "credits" or "license" for more information.
>>> -
```

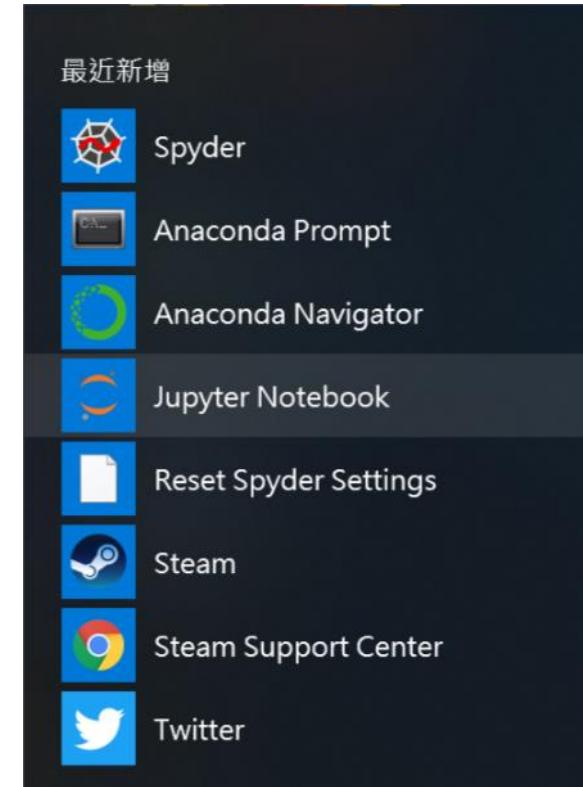
Note: 請務必注意是 3.6.x 以上的版本

啟動 Jupyter notebook

- Step 1. 在命令提示字元(Window)/終端機(Mac)中輸入 jupyter notebook , 或著也可以直接點選執行 jupyter notebook 的應用程式。



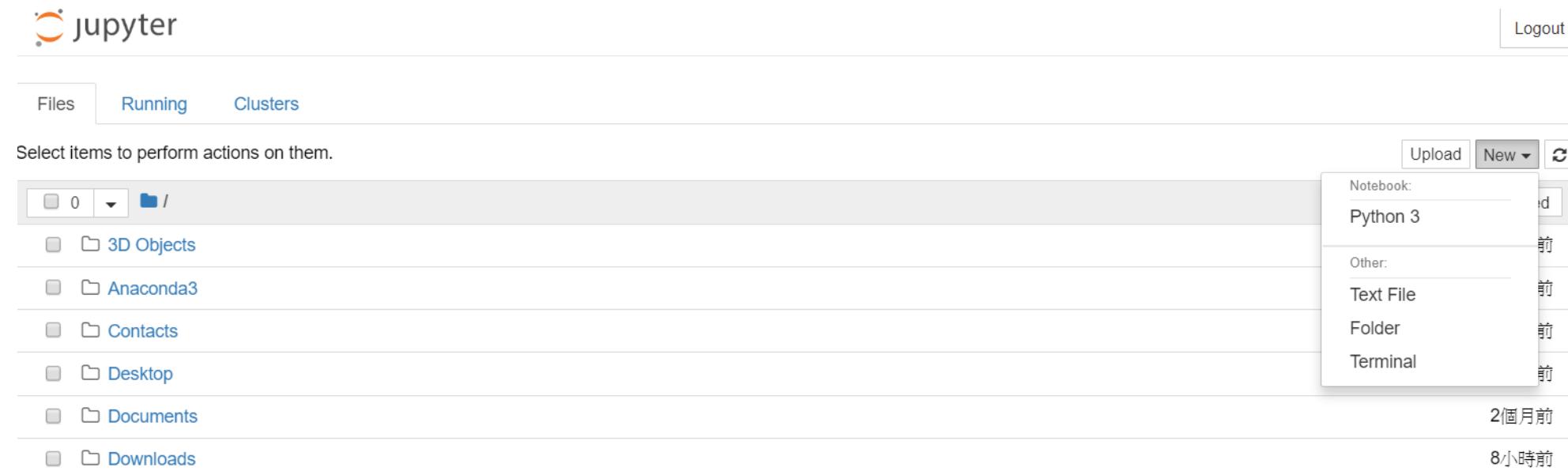
```
命令提示字元
Microsoft Windows [版本 10.0.16299.431]
(c) 2017 Microsoft Corporation. 著作權所有，並保留一切權利。
C:\Users\virgi>jupyter notebook
```



啟動 Jupyter notebook

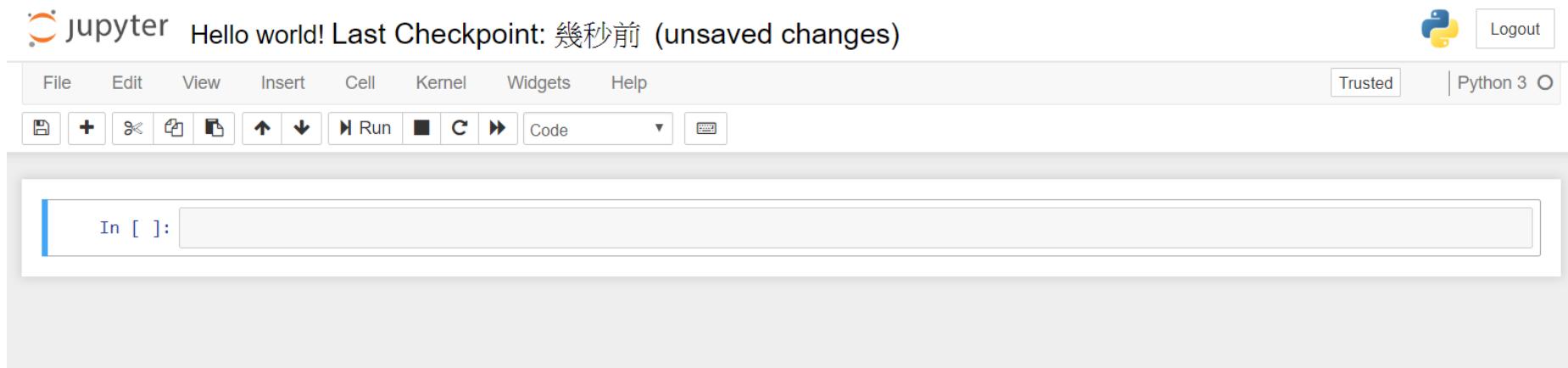
- Step 2. 啟動 Jupyter Notebook 之後，jupyter 會以瀏覽器的形式打開，並且出現下面的介面。點選界面右方的new，按下 Python 3，便能夠進到寫code的頁面中。

Note: 在執行jupyter notebook時不要關閉命令提示字元/終端機。



啟動 Jupyter notebook

- Step 3. 按下python 3 後， 會進入到以下的頁面， 此時就可以開始寫 code 了。



啟動 Jupyter notebook

- Step 4. 嘗試在第一行輸入以下程式碼，並且按下 shift + enter 鍵執行程式。如果成功印出所打的字串內容，代表安裝 python 的步驟已經全部都完成，直譯器也就緒。

```
print("hello world")
```

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with various icons for file operations, cell creation, and kernel selection. The main area displays a code cell labeled 'In [1]:' containing the Python command `print("hello world")`. Below the code, the output 'hello world' is shown. A new cell 'In []:' is currently active at the bottom, indicated by a cursor icon.

啟動 Jupyter notebook

執行程式碼的方式有兩種，分別是：

- Shift + Enter：執行後，游標會移至下一個 cell (程式儲存格)。
- Ctrl + Enter：執行後，游標仍在目前的 cell (程式儲存格)。

在 Anaconda 虛擬環境下安裝 Tensorflow 與 Keras

利用命令提示字元輸入命令視窗建立tensorflow虛擬環境。

- (1.) 按下搜尋圖示， 輸入cmd， 點選 [命令提示字元] 以啟動命令提示字元視窗。
- (2) 建立工作資料夾pythonwork， 並且切換進入這個工作資料夾裡：

```
md \pythonwork  
cd \pythonwork
```

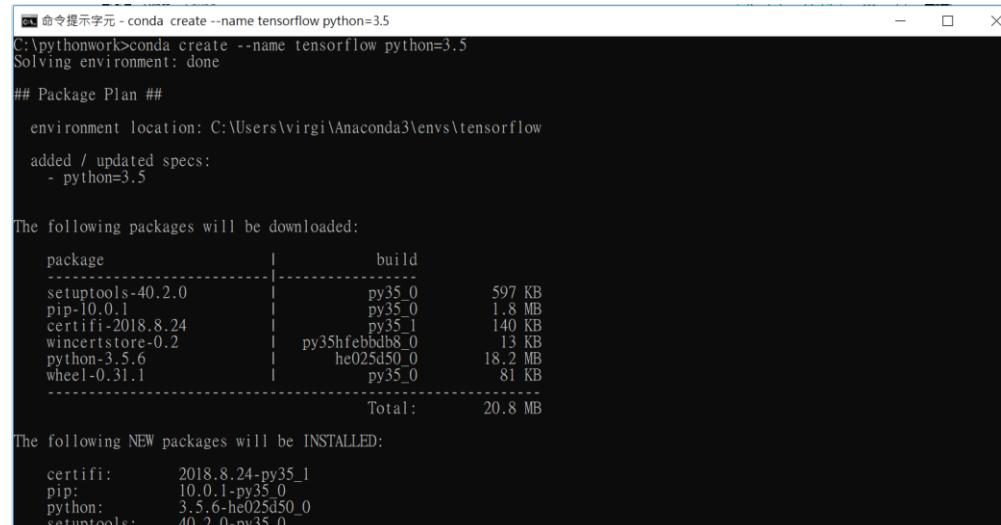
前面加斜線的意思是根目錄， 就不會在現在的資料夾往下建立， 而是直接在根目錄建立 pythonwork 這個資料夾。

在 Anaconda 虛擬環境下安裝 Tensorflow 與 Keras

(3.) 接下來，使用 conda 命令來建立一個命名為 tensorflow 的虛擬環境，並在裡面安裝 Python 3.5 版本。接下來再用這個虛擬環境來安裝 tensorflow 的 CPU 版本。

conda create -n tensorflow python=3.6

執行畫面



```
命令提示字元 - conda create --name tensorflow python=3.6
C:\pythonwork>conda create --name tensorflow python=3.5
Solving environment: done
## Package Plan ##

environment location: C:\Users\virgi\Anaconda3\envs\tensorflow

added / updated specs:
- python=3.5

The following packages will be downloaded:

  package          | build
  -----|-----
  setuptools-40.2.0 | py35_0      597 KB
  pip-10.0.1        | py35_0      1.8 MB
  certifi-2018.8.24 | py35_1      140 KB
  wincertstore-0.2  | py35hfbebdb8_0 13 KB
  python-3.5.6       | he025d50_0   18.2 MB
  wheel-0.31.1        | py35_0      81 KB
  -----
                           Total:   20.8 MB

The following NEW packages will be INSTALLED:

  certifi:          2018.8.24-py35_1
  pip:              10.0.1-py35_0
  python:           3.5.6-he025d50_0
  setuptools:        40.2.0-py35_0
```

在 Anaconda 虛擬環境下安裝 Tensorflow 與 Keras

畫面會出現 “Proceed([y]/n)?”, 按下 y 之後，會開始安裝 tensorflow虛擬環境，並且安裝套件。

```
命令提示字元 - conda create --name tensorflow python=3.5
python-3.5.6           | he025d50_0      18.2 MB
wheel-0.31.1            | py35_0          81 KB
Total:                 20.8 MB

The following NEW packages will be INSTALLED:

certifi:    2018.8.24-py35_1
pip:        10.0.1-py35_0
python:     3.5.6-he025d50_0
setuptools: 40.2.0-py35_0
vc:         14.1-h0510ff6_4
vs2015_runtime: 14.15.26706-h3a45250_0
wheel:      0.31.1-py35_0
wincertstore: 0.2-py35hfebdb8_0

Proceed ([y]/n)?
```

在 Anaconda 虛擬環境下安裝 Tensorflow 與 Keras

利用activate tensorflow 命令啟動tensorflow的anaconda虛擬環境(若要關閉虛擬環境則是使用deactivate)

activate tensorflow

安裝 Tensorflow :

conda install tensorflow

畫面一樣會出現 “Proceed([y]/n)?”, 按下 y 之後，開始安裝。

安裝 Keras :

conda install -c conda-forge keras

在 Anaconda 虛擬環境下安裝 Tensorflow 與 Keras

在 tensorflow 的虛擬環境底下安裝 jupyter notebook。

conda install jupyter notebook

(4.) 在 Jupyter Notebook 中輸入測試碼來測試 Tensorflow 是否安裝成功。

在已安裝好的 Anaconda 資料夾中開啟剛剛安裝的 Jupyter Notebook(tensorflow)。

Installing scikit-learn

切換到要安裝的虛擬環境

activate your_env_name

安裝 scikit-learn

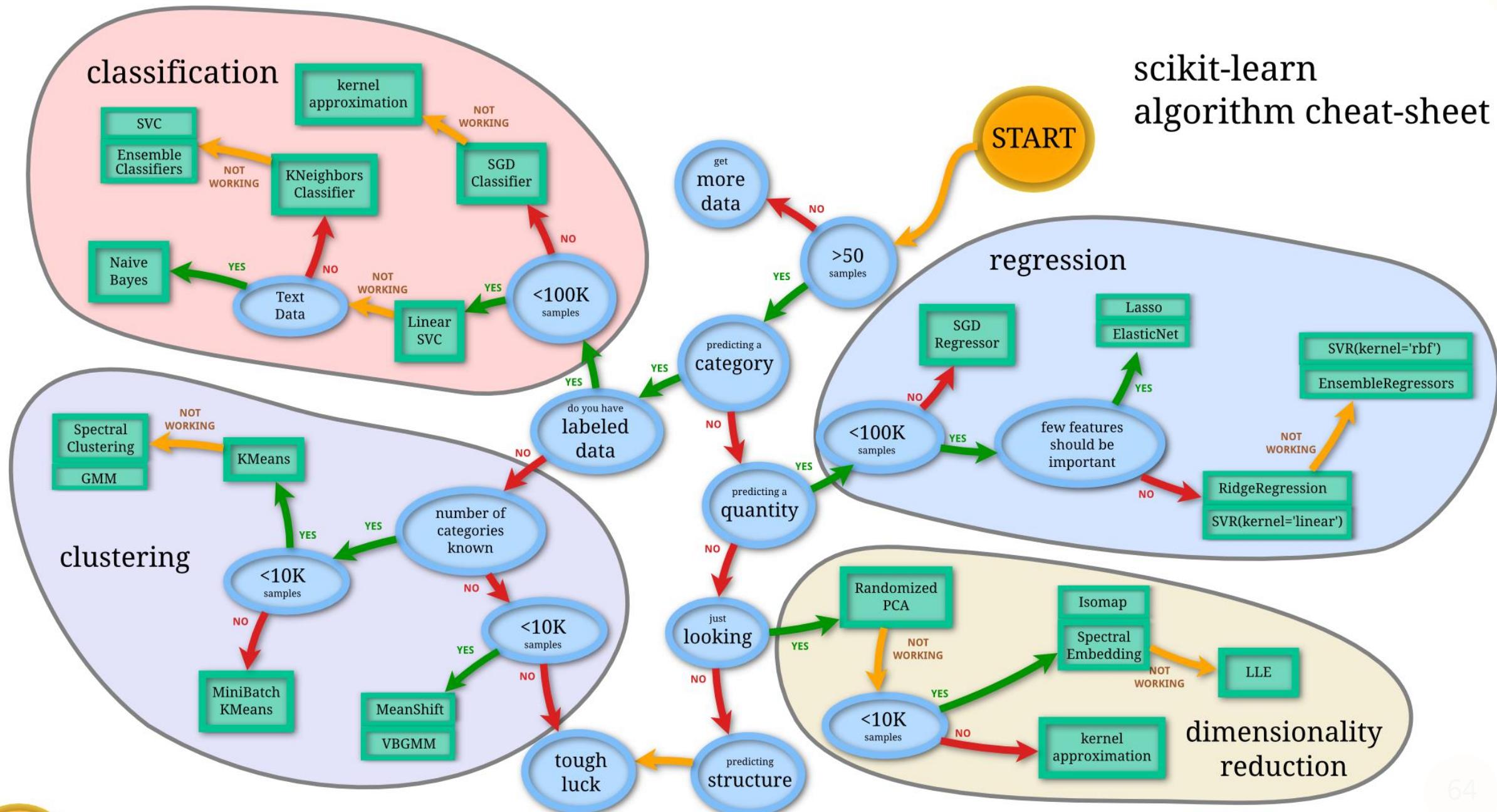
pip install -U scikit-learn

scikit-learn 簡介

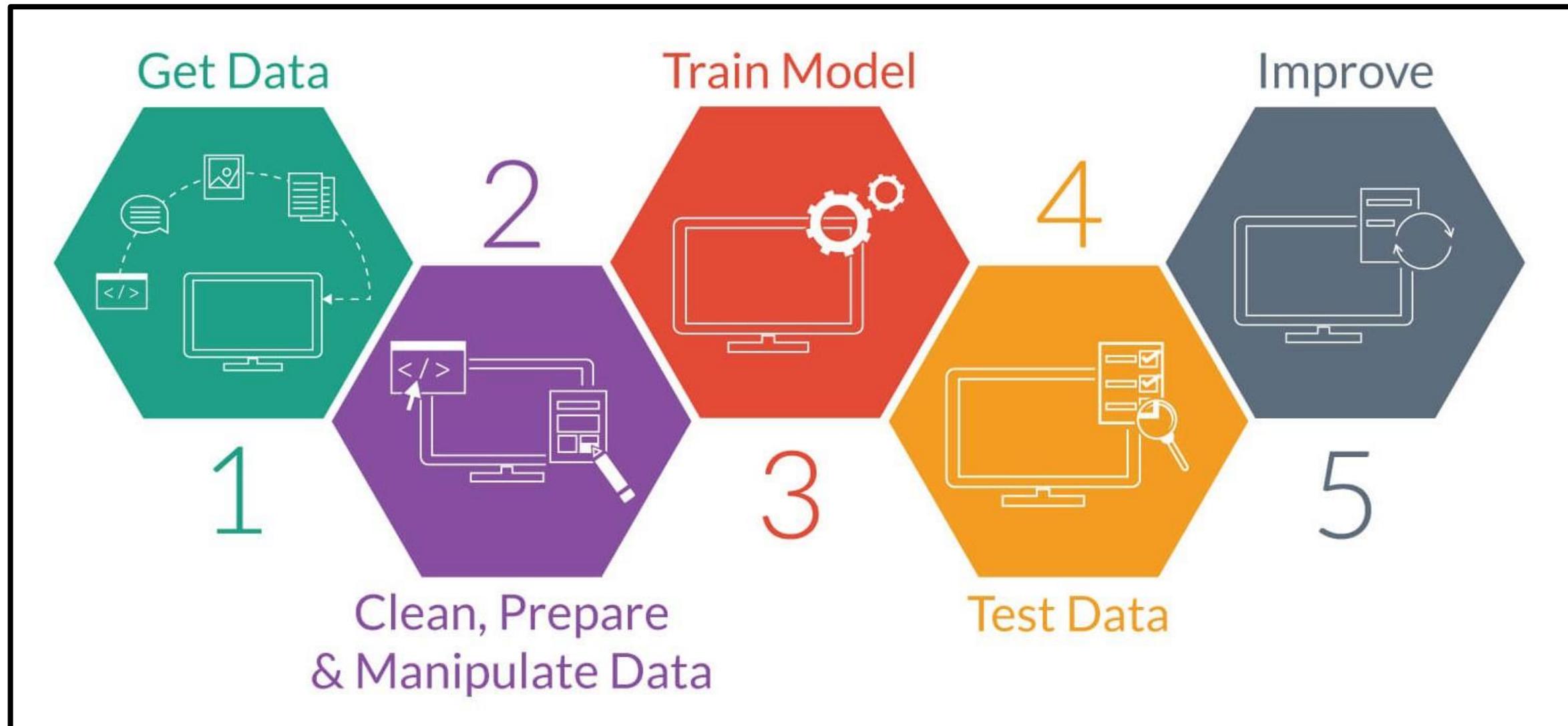
scikit-learn 簡介

- scikit-learn : 提供眾多機器學習演算法的Python 套件,
- 演算法採用一致的操作方式
- 瞭解 基本概念及語法, 可快速瞭解使用方式

scikit-learn algorithm cheat-sheet



基本機器學習工作流程



scikit-learn 簡介

- 資料集以 Bunch 物件格式儲存，類似 Python 字典，包含鍵與值，
- 存取方式1字典方式：`<dict>['<key>']`
- 存取方式2點號方式：`<dict>.<key>`

進行機器學習須有特徵X及目標Y，scikit-learn以2個陣列表示如下：

1. X：特徵陣列 (Feature matrix)

列 (Row)：資料集的樣本

欄 (Column)：資料集的屬性

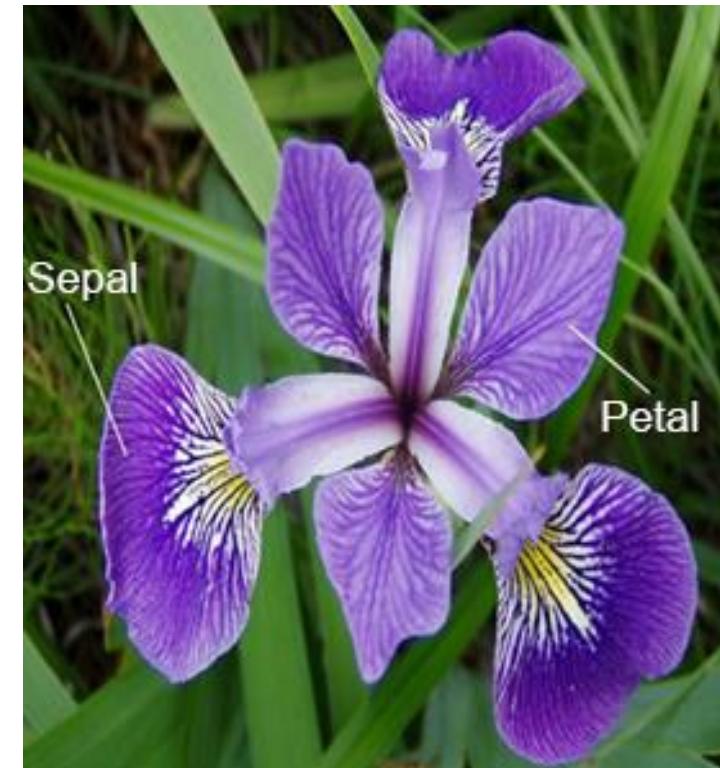
2. y：目標陣列 (Target matrix) 通常命名為

表示每個樣本的類別

資料表示法

例：

安德森鳶尾花資料集 (Anderson's Iris dataset)：鳶尾花的數據包括：
花萼 (Sepal，下垂) 花瓣 (Petal，直立)
的長度與寬度



包括3品種：山鳶尾 (Setosa)、變色鳶尾 (Versicolor)、維吉尼卡鳶尾 (Virginica)

鳶尾花資料

- 新增 python 檔案，輸入以下程式讀取鳶尾花資料 (類似 Python 字典結構)

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
iris.keys()
```

執行結果：

dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])

意義：有 5 筆資料：data, target, target_names, DESCR, feature_names

樣本陣列

```
X = iris.data  
print(X.shape)  
X[:5]
```

(150, 4)

```
array([[5.1, 3.5, 1.4, 0.2],  
       [4.9, 3. , 1.4, 0.2],  
       [4.7, 3.2, 1.3, 0.2],  
       [4.6, 3.1, 1.5, 0.2],  
       [5. , 3.6, 1.4, 0.2]])
```

意義：

* iris.data (或 iris['data']) 為樣本 (Sample) · 為多維陣列 (150, 4)

- 共 150 筆資料 (n_samples) : 每一列就是一個鳶尾花樣本 (Sample)

- 共 4 個欄位 (n_features) : 每一欄就是一個鳶尾花的特徵 (花萼與花瓣的長寬度)

目標陣列

```
y = iris.target  
y
```

```
array([0, 0, 0, ..., 1, 1, 1, ..., 2, 2, 2])
```

意義：

* `iris.target` (或 `iris['target']`) 為目標 (Target)，是一個一維 Numpy 陣列，記錄每筆樣本的類別，長度亦為 `n_samples`，通常以 `y` 表示：0 為山鳶尾，1 為變色鳶尾，2 為維吉尼卡鳶尾

特徵名稱

```
iris.feature_names
```

```
['sepal length (cm)',  
 'sepal width (cm)',  
 'petal length (cm)',  
 'petal width (cm)']
```

意義：

`iris.feature_names` (或 `iris['feature_names']`) 為特徵名稱 (花萼與花瓣的長度與寬度)，是一個 Python 串列

目標名稱

```
iris.target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='|<U10')
```

意義：

`iris.target_names` (或 `iris['feature_names']`) 為目標名稱 (山鳶尾，變色鳶尾，維吉尼卡鳶尾)，是一個一維 Numpy 陣列

資料集說明

```
print(iris.DESCR)
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='|<U10')
```

意義：

關於資料集的文字描述。

資料集說明

Number of Instances: 150 (50 in each of three classes)

Number of Attributes: 4 numeric, predictive attributes and the class

Attribute Information: (略...)

Summary Statistics:					
	Min	Max	Mean	SD	Class Correlation
sepal length:	4.3	7.9	5.84	0.83	0.7826
sepal width:	2.0	4.4	3.05	0.43	-0.4194
petal length:	1.0	6.9	3.76	1.76	0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76	0.9565 (high!)

自訂實驗資料集

- 首先安裝輔助套件 mglearn：

pip install mglearn

- forge dataset：人造二元分類資料集，利用 mglearn.datasets.make_forge() 程式產生

X：資料點陣列，每個資料點有兩個特徵值

y：資料點分類 (0 與 1)

自訂實驗資料集

輸入以下程式並執行：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import mglearn

# Generate dataset: X and y are of 'numpy.ndarray'
X, y = mglearn.datasets.make_forge()
print(X)
print(y)
```

```
...
Backend TkAgg is interactive backend. Turning interactive mode on.
H:\anaconda3\envs\web\lib\site-packages\sklearn\utils\deprecation.py:56: FutureWarning: Backend TkAgg is interactive backend. Turning interactive mode on.
  warnings.warn(msg, category=FutureWarning)
[[ 9.96346605  4.59676542]
 [11.0329545 -0.16816717]
 [11.54155807  5.21116083]
 [ 8.69289001  1.54322016]
 [ 8.1062269   4.28695977]
 [ 8.30988863  4.80623966]
 [11.93027136  4.64866327]
 [ 9.67284681 -0.20283165]
 [ 8.34810316  5.13415623]
 [ 8.67494727  4.47573059]
 [ 9.17748385  5.09283177]
 [10.24028948  2.45544401]
 [ 8.68937095  1.48709629]
 [ 8.92229526 -0.63993225]
 [ 9.49123469  4.33224792]
 [ 9.25694192  5.13284858]
 [ 7.99815287  4.8525051 ]
 [ 8.18378052  1.29564214]
 [ 8.7337095  2.49162431]
 [ 9.32298256  5.09840649]
 [10.06393839  0.99078055]
 [ 9.50048972 -0.26430318]
 [ 8.34468785  1.63824349]
 [ 9.50169345  1.93824624]
 [ 9.15072323  5.49832246]
 [11.563957   1.3389402 ]
[1 0 1 0 0 1 1 0 1 1 1 1 0 0 1 1 1 0 0 1 0 0 0 0 1 0]
```

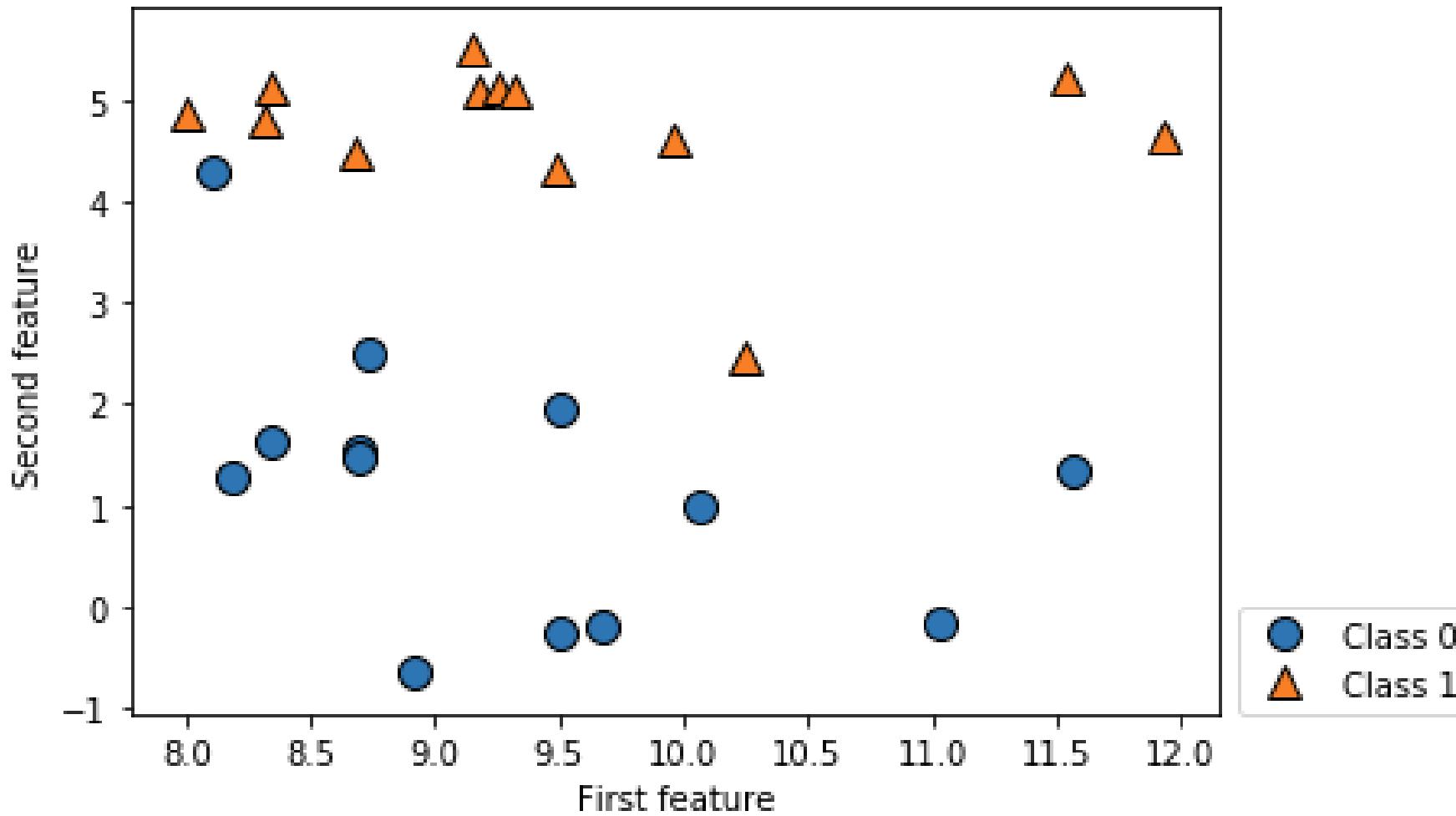
自訂二元分類資料集

利用 `mglearn.discrete_scatter()` 輔助程式繪製示意圖， 資料點分類在圖中以不同顏色之符號呈現：

`discrete_scatter()` 有三個參數：第一、第二個特徵，以及分類標籤(均為一維 Numpy 陣列)

```
# Plot dataset
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(['Class 0', 'Class 1'], loc=(1.02, 0))
plt.xlabel('First feature')
plt.ylabel('Second feature')
```

自訂二元分類資料集



自訂回歸資料集

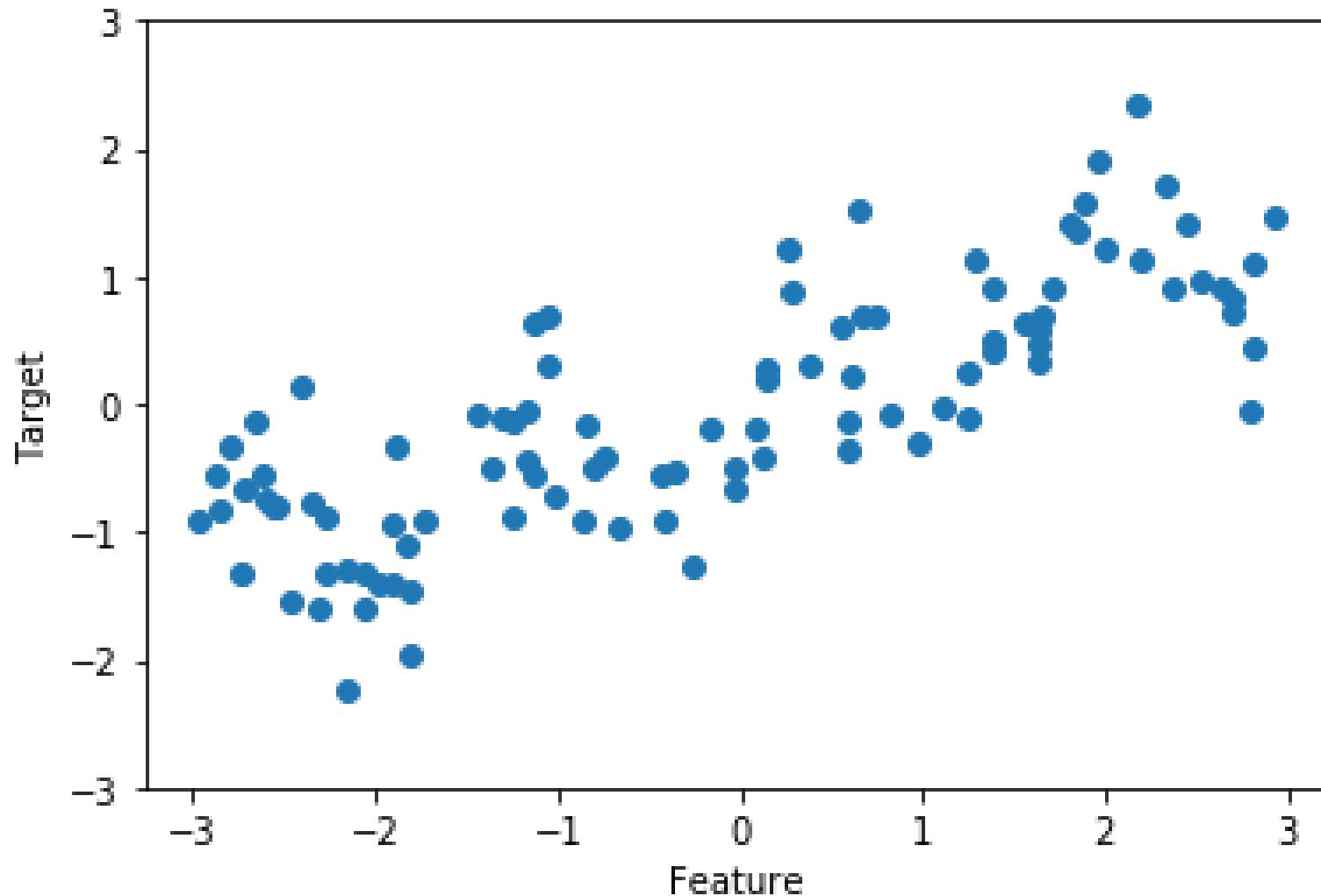
利用 `mglearn.datasets.make_wave()` 程式產生

X ：資料點陣列，每個資料點有一個特徵值

y ：目標值 (連續性數值)

```
# Generate dataset: X and y are of 'numpy.ndarray'  
X, y = mglearn.datasets.make_wave()  
  
# Plot dataset  
plt.plot(X, y, 'o')  
plt.ylim(-3, 3)  
plt.xlabel('Feature')  
plt.ylabel('Target')  
print(X)  
print(y)
```

自訂回歸資料集



載入 sci-kit learn 內建資料集

- cancer dataset : 美國威斯康辛州乳癌資料集 (真實)

```
from sklearn.datasets import load_breast_cancer  
  
cancer = load_breast_cancer()
```

- boston dataset : 美國波士頓房價資料集 (真實)

```
from sklearn.datasets import load_breast_cancer  
  
boston = load_boston()
```

機器學習步驟

機器學習步驟

- 1. 將資料分為「訓練資料」與「測試資料」
- 2. 建構機器學習模型並訓練
- 3. 評估模型 (Evaluating the model)
- 4. 預測 (Prediction)

將資料分為「訓練資料」與「測試資料」

► 常用訓練模型策略

資料分為兩部份：「訓練資料」、「測試集」

scikit-learn 有 `train_test_split()` 函式將樣本資料隨機排序 (Shuffle) 後
分成兩部份：75% 樣本為訓練集，25% 為測試

將資料分為「訓練資料」與「測試資料」

```
from sklearn.datasets import load_breast_cancer  
from sklearn.model_selection import train_test_split  
  
cancer = load_breast_cancer()  
X = cancer.data  
y = cancer.target  
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,  
random_state=0)  
print(X_train)  
print(y_train)
```

將資料分為「訓練資料」與「測試資料」

- * stratify=y : 依據原目標的分類比例來分層

例如：569 個資料點中，212 個點屬於惡性（約 37%），357 個點屬於良性（約 63%），則 X_train 與 X_test 中的分類比例也相同

- * random_state=0 : 設定整數固定種子值（其他整數亦可），讓每次實驗跑的資料都相同

- * train_test_split() 函式回覆 4 個 NumPy 陣列

X_train, X_test 訓練與測試資料，都是二維陣列，每一列是一筆樣本資料

y_train, y_test 訓練與測試答案，都是一維陣列，每一個元素是一個類別資料

K -NN

演算法實作

- 將資料分為訓練與測試資料，以便評估一般化的效能

```
from sklearn.model_selection import train_test_split  
  
X, y = mglearn.datasets.make_forge()  
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=0)
```

- 初始化模型：產生分類器 (3 鄰居)

```
from sklearn.neighbors import KNeighborsClassifier  
  
clf = KNeighborsClassifier(n_neighbors=3) # clf: classifier
```

演算法實作

- 分類器擬合訓練資料

```
clf.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,  
metric='minkowski',  
metric_params=None, n_jobs=1, n_neighbors=3, p=2,  
weights='uniform')  
  
# 此步驟其實就是將訓練資料儲存在模型中，以便預測時使用
```

演算法實作

- 評估一般化效能：利用測試資料來評估

```
clf.score(X_test, y_test).round(2)
```

1.0

100% 正確率：在測試資料集中，正確預測的比率是 100%

亦可檢查訓練的正確率

```
clf.score(X_train, y_train).round(2)
```

0.89

* 訓練資料為 89% 正確率

演算法實作

- 實際預測

```
clf.predict(X_test)
```

```
array([1, 0, 1, 0, 1, 0, 0])
```

最近鄰居回歸 (k-Neighbors regression)

- ▶ 使用 wave 資料集，並加上三個測試資料點（下圖綠星號）
- ▶ 僅考慮一個最近鄰居：目標值就是水平距離最近的資料點的值（藍星號）
 - * 利用 `mglearn.plots.plot_knn_regression()` 繪製示意圖

```
mglearn.plots.plot_knn_regression(n_neighbors=1)
```

最近鄰居回歸 (k-Neighbors regression) 演算法實作

- 載入資料、分為訓練與測試資料、初始化模型、然後擬合（因為並非「分類」問題，因此不需要 stratify 參數）

```
from sklearn.neighbors import KNeighborsRegressor
```

```
X, y = mglearn.datasets.make_wave(n_samples=40)
```

```
# Split the wave dataset into a training and a test set
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
# Instantiate the model and set the number of neighbors to consider to 3
```

```
reg = KNeighborsRegressor(n_neighbors=3)
```

```
# Fit the model using the training data and training targets
```

```
reg.fit(X_train, y_train)
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=3, p=2,
                     weights='uniform')

array([-0.054,  0.357,  1.137, -1.894, -1.139, -1.631,  0.357,  0.912,
       -0.447, -1.139])
```

最近鄰居回歸 (k-Neighbors regression) 演算法實作

* 評估效能：對於回歸模型而言，效能的計算是利用決定係數 R^2 值 (Coefficient of determination)，數值範圍：0~1

```
>>> reg.score(X_test, y_test)
```

0.834

* 預測

```
>>> reg.predict(X_test)
```

```
array([-0.05396539, 0.35686046, 1.13671923, -1.89415682, -1.13881398,
       -1.63113382, 0.35686046, 0.91241374, -0.44680446, -1.13881398])
```

線性模型

* 線性回歸 (Linear regression)

- ▶ 線性回歸也稱為普通最小平方 (Ordinary least squares, OLS), 是最簡單、最典型的回歸線性模型

線性回歸將預測 y 與真實回歸目標 \hat{y} 之間的誤差最小化來學習參數 w 與 b :

預測值 : $\hat{y}_i = w x_i + b$

演算法實作

- 載入 wave 資料、分為訓練與測試資料、初始化模型、然後擬合

```
from sklearn.linear_model import LinearRegression  
  
X, y = mglearn.datasets.make_wave(n_samples=60)  
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)  
  
lr = LinearRegression().fit(X_train, y_train)  
print(lr.coef_)  
print(lr.intercept_)
```

[0.44153666]

-0.01711124414733381

- # scikit-learn 的命名慣例：由訓練資料所導出的參數名稱後面會加一個底線，以便和使用者輸入的參數區隔

演算法實作

評估效能：

```
>>> print(lr.score(X_train, y_train))
```

```
>>> print(lr.score(X_test, y_test))
```

0.66

0.69

測試資料的 R^2 值約為 0.69 並不好，但訓練資料的值也很相近，這有可能是低度擬合的問題

演算法實作

- 改以 boston 真實資料測試 (506 個樣本, 104 個特徵)：載入資料、分為訓練與測試資料、初始化模型、然後擬合

```
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_train, y_train))
print(lr.score(X_test, y_test))
```

脊回歸 (Ridge regression)

上頁結果

0.95

0.61

- * 結果：訓練資料的 R^2 值很高，但測試資料值更差了
- * 很清楚的，訓練集與測試集之間的效能差來自於過度擬合，因此必須找一個可以控制複雜度的模型，最常用的替代方案是**脊回歸 (Ridge regression)**

演算法實作

```
from sklearn.linear_model import Ridge
```

```
ridge = Ridge().fit(X_train, y_train)
print(ridge.score(X_train, y_train))
print(ridge.score(X_test, y_test))
```

```
>>> 0.87
```

```
>>> 0.75
```

- * 訓練資料的效能降低，但測試資料的效能提昇，這符合我們的預期：複雜度較低的模型，一般化的效能較高
- * 由於我們的目標在於提昇測試效能，因此應該選擇脊回歸

- scikit-Learn 簡介
- 實驗資料集
- 機器學習步驟
- 最近鄰居法
- 線性模型
- TensorFlow+Keras



4

演算法分析



2.1 程式的效能分析

- 評估一個程式，除了可讀性，容易維護之外，程式的效能也是必須被評估的項目。
- 就效能而言，一般我們會關注於**程式佔用記憶體的空間**以及**程式執行的時間**
 - 此稱之為**空間複雜度**與**時間複雜度**。

2.1.1 空間複雜度(Space Complexity)

- 空間的複雜度代表**程式執行時所需要的記憶體空間**
 - 又可以分為「**固定的**」及「**變動的**」記憶體空間需求
- 1. **固定的記憶體空間需求**
 - 與輸出和輸入的資料沒有關係，也與程式運作時實際執行的狀況沒有關係
 - 儲存程式碼的空間、儲存靜態變數、全域變數及常數的空間。
 - 在程式編譯完成時，就可以確定。



2. 變動的記憶體空間需求

- 通常與使用者的輸入有關，也與程式執行的過程有關
 - 例如開出大小樂透程式（49 取 6 與 38 取 6），接受使用者的輸入，決定總球數，並透過**動態記憶體配置**（例如 C 語言的 malloc 函式即具備此功能）取得記憶體。
- 函式呼叫的深度
 - 系統堆疊
 - 函式呼叫會使用一些堆疊記憶體，返回也會釋放一些堆疊記憶體空間。
 - 遞迴呼叫可能在遇到邊界條件前耗費大量的堆疊空間
 - 在程式執行前就先預測函式呼叫的次數是很困難的，因為有時候它也與使用者的輸入有關，例如計算 $N!$ 的遞迴函式。



老師的叮嚀

系統堆疊是為了函式呼叫而設計的專用記憶體。當函式被呼叫後，會先把下一個指令位址（稱之為返回位址）存入，然後將被呼叫函式的區域變數也存入，當函式返回後，就會從堆疊中刪除區域變數，然後拿出返回位址，讓 CPU 可以繼續執行原本呼叫前的下一個指令。



- 程式所需要的空間=固定的「記憶體空間需求」+「變動的記憶體空間」需求：

【公式 2-1】程式的記憶體空間需求

$$S(P) = S_f + S_v(P,I) = c + S_v(P,I)$$

$S(P)$ ：程式所需要的記憶體空間總和。

S_f ：固定的記憶體空間需求，它應該是一個常數 c 。

$S_v(P,I)$ ：變動的記憶體空間需求，它應該是一個函數，與輸入 (I) 有關，也與程式內容 (P) 有關，由於在程式執行中，是一個不斷變動的數值，因此應以最大值 Max 方式求得同一時間點內的可變動記憶體空間需求。

舉例來說，一個開大小樂透的函式， $S_v(P,I)=\text{Max}(38,49)=49$ 。



【2-1】試求出下列 C 語言函式的變動記憶體空間需求，並分析其固定記憶體空間需求包含哪些項目。

```
int func1(int x,int y)
{
    return (x+y) * (x-y);
}
```

程式 2.1



解答

- 本範例的變動記憶體空間需求 $S_v(P.I)=0$ ，因為並不包含對其他函式的呼叫
- 而本函式也看不出是否一定會被呼叫，因此函式的 x,y 變數並不會進入系統堆疊。
- 固定記憶體空間需求 S_f 則為函式被編譯完成後的機器碼大小。



【2-2】試分析下列 C 語言函式 func2 的固定記憶體空間需求及變動記憶體空間需求。

```
void main()
{
    int array[10];
    func2(5, array, 10);
}
void func2(int x, int ary[], int n)
{
    const int k=10;
    int i;
    for(i=0; i<n; i++)
        *(ary+i)=i*x*k;
}
```

程式 2.2



解答

■ 固定記憶體空間需求 S_f 包含下列幾個部分：

1. func2 函式被編譯完成後的機器碼大小。
2. 常數 k 的儲存空間。



- 變動記憶體空間需求 $S_v(P.I)$ 是函式被呼叫時，被疊入系統堆疊的參數資料，包含：
 1. func2 函式執行後的返回位址（也就是 main 函式的『』之處）。
 2. 區域整數變數 x, n, i 。
 3. 指標變數 ary。
 - 因為 C 語言傳遞陣列採用傳指標呼叫方式(Pass by Pointer)，傳遞陣列時，只需要傳遞陣列第一個元素的位址，它就能夠透過指標運作方式對陣列進行存取，而陣列大小則由 n 決定。
 - 以上，若對 32 位元的作業系統與編譯器而言，整數變數與常數都將佔用 4 個位元組。而指標變數 ary 則佔用 32 個位元（因為要記錄一個記憶體位址）。



老師的叮嚀

「指標變數」也是一個變數，不過它存放的資料是一個記憶體位址，所以長度與平台有關，在 32 位元的平台中，記憶體位址長度為 32 位元，所以每一個「指標變數」需要占用 32 個位元的記憶體空間。



2.1.2 時間複雜度(Time Complexity)

- 時間的複雜度代表執行程式所需要的時間，分為翻譯所需時間及執行所需時間。
 - 翻譯所需時間在使用編譯器及組譯器的狀況下，都會被忽略。
 - 本書主要使用 C 作為程式語言，只討論程式的執行時間 $\text{RunTime}(P)$ 。
- 計算 $\text{RunTime}(P)$ 有兩種方式：
 - 一種是實際量測
 - 一種是將程式執行的時間分為『敘述被執行的次數』 \times 『敘述所需要的時間』的總和。



一. 實際量測

- 進行實際量測，可透過 `clock()` 或 `time()`、`difftime()` 函式：

1. 利用 `clock` 計算函式執行時間：

函式語法：`long clock();`

標頭檔：`<time.h>`

語法說明：`clock` 函式會回傳目前的 `clock` 數。

使用範例：計算 `func1()` 的執行時間。

```
#include <time.h>
.....
{
    double Run_Time;
    long S,E;
    S=clock();
    func1();
    E=clock();
    Run_Time=((double)(S-E))/CLK_TCK;
}
```

程式 2.3

宣呼叫 `func1()` 之前與之後
各呼叫一次 `clock()`

兩者之差除以 1000，即為
`func1()` 執行的時間
CLK_TCK 是 1000，即系統
的每秒 `clock` 數值



2. 利用 time 計算函式執行時間：

函式語法：long time (time_t *tp);

標頭檔： <time.h>

語法說明：

回傳 1970/1/1 至今經過的秒數。如不需要保留，tp 設定為 NULL 即可。

函式語法：double difftime(time_t t2, time_t t1);

標頭檔： <time.h>

語法說明：difftime 會回傳 t2 與 t1 的時間差。

使用範例：計算 func1()的執行時間。

```
#include <time.h>
.....
{
    double Run_Time;
    long S,E;
    S=time(NULL);
    func1();
    E=time(NULL);
    Run_Time=difftime(E,S);
}
```

程式 2.4

呼叫 func1() 之前與之後
各呼叫一次 time(NULL)

利用 difftime() 可以計算時間差



二.乘積總和

- 程式執行的時間可分為兩個部分的乘積總和。公式如下：

程式執行時間 = Σ (敘述被執行的次數 \times 執行敘述所需要的時間)

- 『執行敘述所需要的時間』與編譯器或作業系統、硬體設備有非常大的關聯。
- 因此，在採取本方法時，我們關注的是『敘述被執行的次數』
 - 『敘述被執行的次數』事實上就是**程式的步驟**，並可透過兩種方式取得。
 - (1)**手動計算**：模擬程式的執行，以求得該步驟會被執行幾次。在非常複雜的程式中，可能必須運用**數學配合觀察**導出結果。
 - (2)**加入計數器**：計數器應該**在開始前被歸零**，以及每一次步驟被執行時，計數器都必須**累加一**
 - 在區段符號 {} 被省略的狀況時要特別注意
 - 如下範例中的 {} 就不應該被省略，否則無法正確計算 $Sum=Sum+i;$ 步驟被執行的次數。

程式 2.5

```
:          執行之前應先歸零
:
for(i=0, counter=0; i<100; i++)
{
    Sum=Sum+i;      /* 這是準備被量測的步驟 */
    counter++;      放在{}內可以量測{}內敘述被執行的次數
}
```



2.2 演算法的效能分析

- 估算演算法的效能，不需要考量軟硬體工具，因此大多著重在**執行次數**
- 大多時候估算演算法的效能只會以**漸進式估算**來表示。

2.2.1 估算演算法的步驟執行次數

- 估計演算法的步驟執行次數只能透過**觀察與數學手段**來達成，因為演算法尚未能由電腦直接執行。
- 以下，我們透過兩個實際的案例來做示範。



【 2-3 】

試列出下列階乘演算法中各行號步驟的執行次數，以及整個演算法共執行幾個步驟。

```
1  Procedure FACTORIAL
2      read(n)
3~4      If n<0 then [ print('error'); stop]
5~6      If n=0 then [ print('0'); stop]
7      FACT ← 1
8      For i←1 to n do
9          FACT ← i * FACT
10     endFor
11     print(FACT)
12 endProcedure
```



解答

- 本範例可以依「n 是否小於 0」來分項討論。



1. 當 $n \leq 0$ 時，各步驟的執行次數如下：

行號	執行次數	
	$n < 0$	$n = 0$
第 1 行	1	1
第 2 行	1	1
第 3 行	1	1
第 4 行	1	0
第 5 行	0	1
第 6 行	0	1
第 7~11 行	0	0



```

8      For i←1 to n do
9          FACT ← i * FACT
10     endFor

```

2. 當 $n > 0$ 時，各步驟的執行次數如下：

行號	執行次數	行號	執行次數
第 1 行	1	第 7 行	1
第 2 行	1	第 8 行	$n+1$
第 3 行	1	第 9 行	n
第 4 行	0	第 10 行	n
第 5 行	1	第 11 行	1
第 6 行	0	第 12 行	1

- 其中，第 8 行會被執行 $n+1$ 次，是因為 i 值會遞增到 $n+1$ 時才跳出迴圈。
- 令本演算法共會執行 $S(n)$ 個步驟，則 $S(n)$ 可整理如下公式：

$$S(n) = \begin{cases} 4 & ,n<0 \\ 5 & ,n=0 \\ 3n+8 & ,n>0 \end{cases}$$



【2-4】 請問下列片段程式執行後，counter 值為多少？（相當於計算 counter++;被執行幾次）。

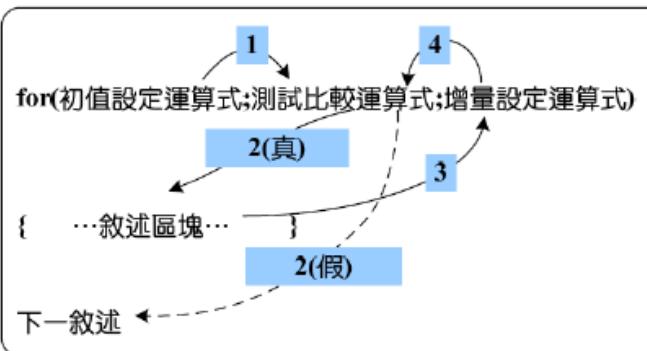
```
for(i=1,counter=0;i<=n;i++)
    counter++;
```

這個敘述位於迴圈內



解答

- 觀察範例的執行次數，必須瞭解 C 語言 for 迴圈的執行流程，如下圖：



- 在上圖中，敘述區塊內的執行次數將會與測試比較運算式的比較次數差 1 次。
- 以這個範例來說，我們可以將 n 假設為 0,5,10 等值列表來觀察 i=1、counter=0、i<=n?、i++、counter++各會執行多少次。

```
for(i=1,counter=0;i<=n;i++)
    counter++;
```

運算式	執行次數		
	$n \leq 0$	$n=5$	$n=10$
$i=1$	1	1	1
$counter=0$	1	1	1
$i \leq n?$	1	$6=(5+1)$	$11=(10+1)$
$counter++$	0	5	10
$i++$	0	5	10

- 不論 n 為多少， $i=1$ 與 $counter$ 都只會執行一次。
- 如果 $n \leq 0$ ，則 $counter++$ 與 $i++$ 都不會執行，不過 $i \leq n?$ 最少會執行一次，因為它要判斷是否執行迴圈內的敘述。
- 當 $n > 0$ 時， $counter++$ 與 $i++$ 都會執行 n 次，而 $i \leq n?$ 會執行 $n+1$ 次
 - 換句話說 $i \leq n?$ 一定會比 $counter++$ 多執行一次，因為最後一次進行判斷時，會因為不符合條件而離開迴圈，此時 $counter++$ 不會被執行。

```
for (i=1,counter=0;i<=n;i++)
    counter++;
```

- 另外值得注意的是，當離開這個迴圈後，counter 的值為 n，但 i 的值為 n+1，因為第一次執行 `i++` 時，是將 `i` 由 1 遞增為 2。
- 故 counter 值如下公式：

$$\text{counter} = \begin{cases} 0 & , n \leq 0 \\ n & , n > 0 \end{cases}$$



範例

【2-5】請問下列片段程式執行後，counter 值為多少？（相當於計算 counter++;被執行幾次）。

```
for (i=1, counter=0; i<=n; i++)  
    for (j=2; j<=n; j++)  
        counter++;
```

這個敘述位於兩層迴圈內



解答

- $j=2$ 會執行 n 次（若 $n > 0$ ），執行次數必定與 $i++$ 相同，因為是外層迴圈內的敘述。
- $n \leq 0$ 的情況，迴圈內的敘述不會被執行。
- 可以把 i 的值攤平來看看內層迴圈的各個運算式被執行幾次
 - $counter++$ 被執行的次數必定與 $j++$ 被執行的次數相同。
- 下面的表格先假設 n 為 5，所以在討論內層迴圈時， i 值最大只可能為 5。



```
for(i=1,counter=0;i<=n;i++)
    for(j=2;j<=n;j++)
        counter++;
```

運算式	執行次數				
	i=1 n=5	i=2 n=5	i=3 n=5	i=4 n=5	i=5=n n=5
j=2	1	1	1	1	1
j<=n?	5	5	5	5	5
j++	4	4	4	4	4

(brace spanning the last two rows of the table)

- j++執行的次數一共為 4×5 一共有 n 欄
 - 前面的 4 是跟隨著 n 而變化，所以 4 可以表示為 $(n-1)$
 - 後面的 5 則是因為 i 從 1 到 5（所以有 5 欄），而 i 在內層迴圈中真正可能出現的狀況是從 1 到 n，所以 5 可以表示為 n。
- 所以 j++的執行次數一共有 $n(n-1)$ 次。counter++的執行次數同樣是 $n(n-1)$ 次。

👉 小試身手 2-1 範例 2-5 的 $j \leq n$ 一共有會被執行幾次？



【2-6】請問下列片段程式執行後，counter 值為多少？（相當於計算 counter++; 被執行幾次）。

```
for(i=1,counter=0;i<=n;i++)
    for(j=1;j<=i;j++)
        counter++;
```

- 需要注意內層迴圈的條件值為 $j \leq i$ ，我們還是把 i 值攤開，看看每次內部迴圈執行時，敘述會執行幾次？

運算式	執行次數				
	i=1 n=5	i=2 n=5	i=3 n=5	i=4 n=5	i=5=n n=5
j=1	1	1	1	1	1
j<=i?	2=i+1	3=i+1	4=i+1	5=i+1	6=i+1
j++	1=i	2=i	3=i	4=i	5=i

一共有 n 欄

- 所以 $j++$ 執行的次數一共為 $1+2+3+\dots+n = \frac{n(n+1)}{2}$ 次。 $counter++$ 的執行次數也是 $\frac{n(n+1)}{2}$ 次，條件則是 $n \geq 0$ 。



2.2.2 估算演算法複雜度的 Big-O 漸近式表示法

- 在範例 2-4 中，`count++`會執行 n 次，而 `i<=n?` 會執行 $n+1$ 次，差這一次很重要嗎？當 n 的值越來越大時，那多一次又何妨？
 - 現在電腦的速度越來越快，多出的一次越顯得微不足道。
 - 事實上，即便多增加 c 次，只要 c 是常數，當 n 越來越大時， c 的影響將越小。
- 範例 2-5 的 `counter++` 敘述執行次數為 n^2-n ，當 n 越來越大時，幾乎 n^2 就可以決定了效能，是否 $-n$ 或 $+n$ 並不會影響效能太多。
 - 較低次的項目與常數是相同的道理，因常數項目 c 相當於 $c \times n^0$ 。
 - 除了較低次的項目以外，是否還有影響性比較小的因素呢？答案是有的，以範例 2-5 與 2-6 的 `counter++` 敘述為例，前者會執行 $n(n-1)$ 次，後者會執行 $\frac{n(n+1)}{2}$ 次，兩者幾乎為倍數關係，但 2 倍很重要嗎？在評估演算法時，是不重要的。
 - 因為演算法並不考慮執行的軟硬體平台，根據硬體的摩爾定律，倍數因素可以因為硬體效能改善而解決。
 - 摩爾定律－「硬體的效能每隔 18 個月便會將提升一倍」



- 以上的項目是微不足道的，因此，大多採用一種**漸進式表示法**來描述演算法的複雜度。
 - 漸進式表示法一般以數學函數來表現，常見的有 O 、 Ω 、 Θ 等三種數學漸近表示函數，其中又以 O （唸作 Big-Oh 或 Big-O）函數最常被使用。

- 以上影響較小的項目會被漸近表示函數的定義或定理所吸收，Big-Oh 的定義如下：

【定義 2-1】Big-Oh Notation（在數學領域稱為 after Landau）

若且唯若有兩個大於 0 的常數 c 與 n_0 ，當所有 $n \geq n_0$ 都滿足 $|f(n)| \leq c|g(n)|$ ，則 $f(n)$ is $O(g(n))$ 。

其中 $f(n)$ is $O(g(n))$ 唸作 $f(n)$ is Big-Oh of $g(n)$ 。

數學式如下：

$$f(n) \text{ is } O(g(n)) \Leftrightarrow \exists c \exists n_0 \forall n (n \geq n_0 \rightarrow |f(n)| \leq c|g(n)|)$$

【註】 有時候 is 也會寫成 = 。



- 簡單來說，Big-Oh 函數代表的含意是「當 n 夠大時（ $n \geq n_0$ 時）， $f(n)$ 的上限是 $g(n)$ 」，也就是 $f(n)$ 的增長絕對不會超過 $c|g(n)|$ 的增長。

- 例如 $n^3 + 1000 = O(n^3)$

- 因為當 $n \geq 1$ 時， $n^3 + 1000$ 的增長不會超過 $1001n^3$ 的增長。
- 此時可假設 $f(n)$ 為 $n^3 + 1000$ ， n_0 為 1， c 為 1001， $g(n)$ 為 n^3 。

- 例如 $3n^2 + 2n + 1 = O(n^2)$

- 因為當 $n \geq 1$ 時， $3n^2 + 2n + 1$ 的增長不會超過 $6n^2$ 的增長
- 此時可假設 $f(n)$ 為 $3n^2 + 2n + 1$ ， n_0 為 1， c 為 6， $g(n)$ 為 n^2 。

【註】一般我們在以 Big-Oh 描述演算法的複雜度時，都會加上一個條件，也就是「 $g(n)$ 必須是最小的函數」，如此才能確保其唯一性。

- 由上述幾個例子，讀者可以發現多項式的 Big-O 會由最高次方項來決定。

【定理 2-1】多項式的 Big-Oh notation

若 $f(n) = a_p n^p + a_{p-1} n^{p-1} + \dots + a_1 n + a_0$
則 $f(n)$ is $O(n^p)$



【2-7】試求出下列多項式的 Big-Oh 表示法。

① $10n^3 + 0.2n^2 - 3$

② $9n^2 + \sqrt{3n}$



解答

①因為 3 次方是最高次方，所以 $10n^3 + 0.2n^2 - 3 = O(n^3)$ 。

②因為 2 次方是最高次方，所以 $9n^2 + \sqrt{3n} = O(n^2)$ 。



- 除了 $O(n)$ 、 $O(n^2)$ 等 Big-Oh 之外，常見的 Big-Oh 還有很多（如下表所列）。

Big-Oh 函數	名稱
$O(1)$	常數時間 (constant)
$O(\log_2 n)$	次線性時間 (sub-linear) 或對數時間 (logarithm)
$O(n)$	線性時間 (linear)
$O(n \log_2 n)$	次平方時間 (sub-quadratic)
$O(n^2)$	平方時間 (quadratic)
$O(n^3)$	立方時間 (cubic)
$O(2^n)$	指數時間 (exponential)
$O(n!)$	階乘時間 (factorial)

表 2-1 常見的 Big-Oh 函數

- 當 $n \geq 16$ ，所代表的時間複雜度由快到慢，正如上表之順序（由上而下）。



■ 將各類 Big-Oh 函數製作如表 2-2 及圖 2-1

- 更能發現當 n 值越大時對該 Big-Oh 函數的影響也越大。

$O(g(n))$	$n=16$	$n=64$	$n=1,024$	$n=1,048,576$
$O(1)$	1	1	1	1
$O(\log_2 n)$	4	6	10	20
$O(n)$	16	64	1024	1048576
$O(n \log_2 n)$	64	384	10240	20971520
$O(n^2)$	256	4096	1048576	1099511627776
$O(n^3)$	4096	262114	1073741824	1152921504606846976
$O(2^n)$	65536	2^{64}	2^{1024}	$2^{1048576}$
$O(n!)$	$16!$	$64!$	$1024!$	$1048576!$

表 2-2 常見的 Big-Oh 函數的數值變化

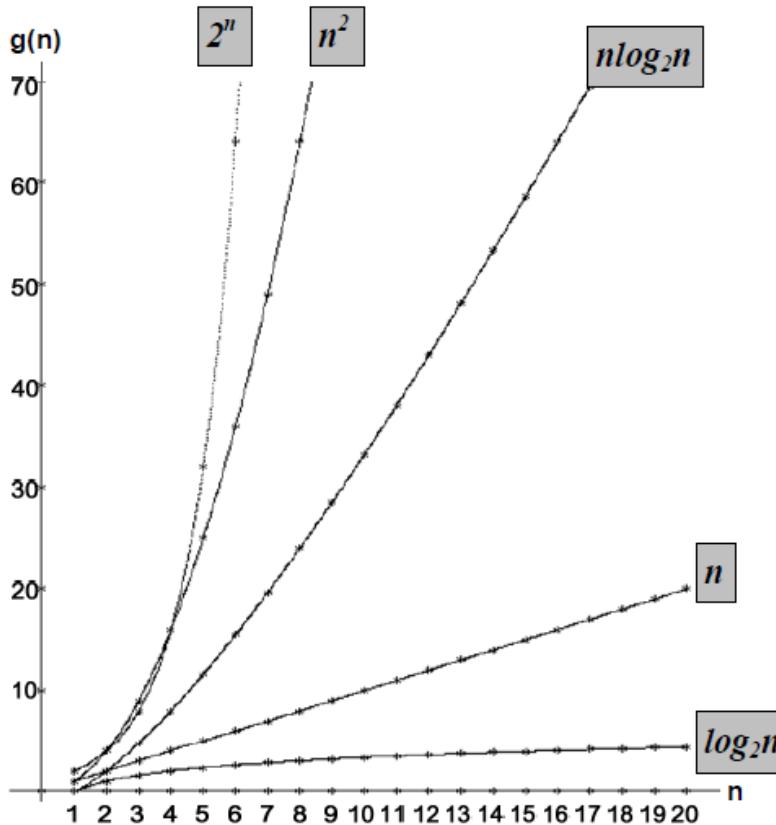


圖 2-1 常見 Big-Oh 函數圖形

■ 由表 2-1 可以很容易看出，常用的 Big-Oh 函數大小如下：

- $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$



【 2-8 】試求出下列多項式的 Big-Oh 表示法。

① $7 \log_2 n + 3$

② $20n \log_2 n + 7n^2 + 20n$

③ $10n^3 + 20n^2 \log_2 n + 30(2^n)$

④ $2^n + 3^n + 5n!$



解答

①因為 $O(\log_2 n) > O(1)$ ，所以 $7 \log_2 n + 3 = O(\log_2 n)$ 。

②因為 $O(n^2) > O(n \log_2 n) > O(n)$ ，所以 $20n \log_2 n + 7n^2 + 20n = O(n^2)$ 。

③因為 $O(2^n) > O(n^3) > O(n^2 \log_2 n)$ ，所以 $10n^3 + 20n^2 \log_2 n + 30(2^n) = O(2^n)$ 。

④因為 $O(n!) > O(3^n) > O(2^n)$ ，所以 $2^n + 3^n + 5n! = O(n!)$ 。



【2-9】試以 Big-Oh 表示範例 2-4 片段程式的時間複雜度。

```
for (i=1, counter=0; i<=n; i++)  
    counter++;
```

這個敘述位於迴圈內



■ 想要求出實際的執行步驟總和，有兩種方式

- 第一種是將 for 迴圈看成一個敘述。
 - for 敘述（專指包含 for 關鍵字的那一行）執行次數為 1 次，而迴圈內的 counter++ 敘述執行次數為 n 次，所以總共是 $n+1$ 次。
- 另一種方式則是將 for 迴圈拆開來，初值設定、條件測試、遞增運算等都當成獨立的敘述。
 - $i=1$ 與 counter 敘述執行次數各為 1 次
 - $i \leq n?$ 的執行次數為 $n+1$ 次
 - $i++$ 執行次數為 n 次，counter++ 執行次數為 n 次
 - 所以總共為 $3n+3$ 次。



■ 當我們以 Big-Oh 來表示時，不必分成兩種方式來討論

- 因為迴圈內的敘述與 $i++$ 的執行次數相同，也就是 n 次，而 $n+n$ 仍然為 $O(n)$
- 常數項也會被吸收，也就是 $3n+3$ 也是 $O(n)$ 。
- 因此，這一題的答案為 $O(n)$ 。



老師的叮嚀

簡單來說，使用 Big-Oh 來評估一個演算法時，只要看哪一個步驟會被執行最多次即可，它的 Big-Oh 就是整個演算法的 Big-Oh，因為其他較少或相同等級的因素會被吸收。

👉 小試身手 2-2

試以 Big-Oh 表示範例 2-5 片段程式的時間複雜度。

延伸學習

更多的漸進式表示法 Ω 、 θ

見課本的完整版（資料結構初學指引 - 使用 C 語言）



2.2.3 遞迴演算法的複雜度估計

- 相信各位讀者在撰寫程式時，曾經練習過遞迴呼叫，遞迴呼叫又分為**直接遞迴**(direct recursion)與**間接遞迴**(indirect recursion)兩種，其含意如下：
- **遞迴呼叫**：
 - 一個函式經由直接或間接呼叫函式本身，稱之為函式的『遞迴呼叫』。
 - func1() 呼叫 func1() 為直接遞迴呼叫
 - func1() 呼叫 func2() 且 func2() 呼叫 func1() 為間接遞迴呼叫。
- 遞迴最早是使用在數學定義式，它代表的是**自我參考定義**的一種機制
 - 例如費氏數列、數學的階層與河內塔都是常見的遞迴問題。
- 估算遞迴演算法複雜度有時很簡單，有時較複雜，需利用較多的數學手段或觀察。



費本納西數列（簡稱費氏數列）

【定義 2-2】費氏數列

『費氏數列』：一個無限數列，該數列上的任一元素值皆為前兩項元素值的和，數列如下所示：

0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , 89 , 144.....

- 使用數學函數來表示費氏數列時，費氏數列的定義本身就是一個遞迴定義如下：

費氏數列的遞迴定義式	
$F(0) = 0$	$n = 0$
$F(1) = 1$	$n = 1$
$F(n) = F(n-1)+F(n-2)$	$n \geq 2$

- 在實際計算時（例如計算 $F(10)$ ），數學的遞迴函數必須不斷地重複呼叫自己，直到遇上非遞迴定義式為止，才能夠求出答案。
- 通常遞迴演算法可以輕鬆解決一些資訊領域常見的問題，而且相當簡潔又非常容易讓人理解。
 - 例如：樹狀圖的相關演算法



- 在設計遞迴演算法時，必須符合下列兩個條件：
 1. 遞迴演算法必須有邊界條件，當符合邊界條件時，就應該返回
 - 在費氏數列中， $F(0)=0$ 與 $F(1)=1$ 就是邊界條件。
 2. 遞迴演算法在邏輯上，必須漸漸往邊界條件移動，否則將無法停止
 - 這將造成不符合演算法的有限性要求。
- 類似地，在設計程式的遞迴函式時，也必須符合上述兩個條件，但原因則有些不同，簡述如下：
 1. 遞迴函式必須有邊界條件，當符合邊界條件時，就應該返回函式呼叫處
 - 在費氏數列中， $F(0)=0$ 與 $F(1)=1$ 就是邊界條件。
 2. 遞迴函式在邏輯上，必須漸漸往邊界條件移動，否則函式呼叫無法停止
 - 雖然程式允許無限地執行下去（例如無窮迴圈），但由於每次進行函式呼叫時，都必須耗費系統堆疊來記錄某些資料，這將使得堆疊的記憶體終被耗盡，進而不正常的中斷程式，甚至造成當機。



【2-10】試以遞迴方式設計一個求費氏數列元素的演算法，並使用非遞迴方式設計具有相同功能的演算法。



■ 非遞迴演算法

Input：輸入費氏數列元素的索引值。

Output：輸出對應的費氏數列元素值。

```
1  Function Fib(n)
2~3    If (n=1) or (n=0)  then  [return n]
4        sum←1
5        For i←2 to n do
6            n1←sum
7            sum←sum+n2
8            n2←n1
9        endFor
10       return sum
11    endFunction
```



■ 遞迴演算法

Input：輸入費氏數列元素的索引值。

Output：輸出對應的費氏數列元素值。

```
1  Function Fib(n)
2~3    If (n=1) or (n=0) then [return n]
4        return Fib(n-1)+Fib(n-2)
5  endFunction
```



範例說明

- 遞迴方式比較容易讓人理解，幾乎只是把數學的遞迴定義轉換為演算法而已。
- 分析遞迴演算法的複雜度較複雜一些，請見下一個範例。

延伸學習 iteration

在程式或演算法設計中，for 或 while 迴圈每執行一次稱之為一個 iteration，iteration 不易尋找到適當的翻譯，有些書籍將之翻為「疊代法」或「重複」，因為它代表迴圈的每一次重複執行。



階層計算

- 階層計算也可以使用遞迴函式來完成，因為階層本身就是一個遞迴定義式
 - $n! = n \times (n-1) \times (n-2) \times \dots \times 1$ ，另一方面， $n!$ 也可以寫成 $n! = n \times (n-1)!$ ，所以令 $\text{Fac}(n)$ 代表 $n!$ 的函數，則其定義如下：

階層的遞迴定義式	
$\text{Fac}(0) = 1$	$n = 0$
$\text{Fac}(n) = n \times \text{Fac}(n-1)$	$n \geq 1$



【 2-11 】

試分析下列 C 語言的遞迴函式之空間複雜度與時間複雜度。

```

1 int FAC(int n)
2 {
3     if(n<0)
4         return -1;
5     if(n==0)
6         return 1;
7     return n * FAC(n-1);
8 }
```

程式 2.6



解答

- 空間複雜度可分為固定的記憶體需求與變動的記憶體需求，我們暫不討論固定的記憶體需求（因為這和編譯後的程式碼大小有關）。
 - 變動的記憶體需求，主要出現在呼叫函式時對系統堆疊的需求
 - 假設我們在 32 位元的平台上，則一個整數變數 n 需要使用 4 個位元組，函式呼叫的返回位址(return address)也需要 4 個位元組（以 32 位元定址）。
 - 每一次呼叫 FAC 函式需 8 個位元組的堆疊記憶體需求來存放這些必要資訊
 - 假設在 main 函式中，呼叫 $\text{FAC}(n)$ ，則在執行過程中，最多需要 $n \times 8 = 8n$ 個位元組的堆疊記憶體需求
 - 所以空間複雜度為 $8n = O(n)$ 。



- 關於函式的時間複雜度，我們可以先加入計數器，也就是將程式改寫如下：

```
1 int counter;
2
3 int FAC(int n)
4 {
5     counter++;    代表 if(n<0) 敘述
6     if(n<0)
7     {
8         counter++; /* 代表 return 敘述 */
9         return -1;
10    }
11    counter++;    代表 if(n==0) 敘述
12    if(n==0)
13    {
14        counter++; /* 代表 return 敘述 */
15        return 1;
16    }
17    counter++;    /* 代表 return 敘述 */
18    return n * FAC(n-1);
19 }
```

程式 2.7

- 改寫程式後，我們可以很容易地透過觀察 counter 值的變化，推導出總執行步驟次數 $S(n)$ 的數學公式如下：



$$S(n) = \begin{cases} 2 & ,n<0 \quad (\text{最佳狀況}) \\ 3 & ,n=0 \quad (\text{正常輸入的最佳狀況}) \\ 3n+3 & ,n>0 \quad (\text{最差狀況}) \end{cases}$$

- 若以 Big-Oh 來表示，則如下結果，且平均值亦為 O(n)。

$$S(n) = \begin{cases} O(1) & ,n<0 \quad (\text{最佳狀況}) \\ O(1) & ,n=0 \quad (\text{正常輸入的最佳狀況}) \\ O(n) & ,n>0 \quad (\text{最差狀況}) \end{cases}$$

延伸學習

尾歸遞迴(tail recursion)

在某些使用遞迴撰寫的演算法實例中，常常出現一種特殊的遞迴稱之為尾歸遞迴（tail recursion），其定義如下：

尾歸遞迴：遞迴呼叫僅出現在程序的最後一個步驟稱之尾歸遞迴。換句話說，遞迴呼叫返回後，必須只剩下程序結尾或返回(return)敘述。

例如範例 2-11 就是一個尾歸遞迴的實例，尾歸遞迴具有一個特性，就是特別容易將之轉換為疊代法(iteration)，也就是迴圈撰寫方式。



河內塔(Towers of Hanoi)

- 數學家 Lucas 在 1883 年提出的數學遊戲問題，它能夠充分表達遞迴的特別之處。
- 當時在印度的某個寺院中擺放了 64 個大小不同的金盤，這些盤子被疊放在一起，越大的放在越下面，每個盤子中間都有一個孔，並被一個由鑲有寶石的木樁貫穿。
 - Lucas 提出的假設是，假設有另外兩個木樁可供使用，照下列規則將 64 個大小不同的金盤搬移到另外的木樁後，世界末日將會來臨。

【定義 2-3】河內塔遊戲的搬移規則

- (1) 每次只能移動一個盤子（即各木樁最上方的那個盤子）。
- (2) 任何盤子可以由任一個木樁搬移到另一個木樁。
- (3) 較小的盤子永遠必須放在較大的盤子之上。

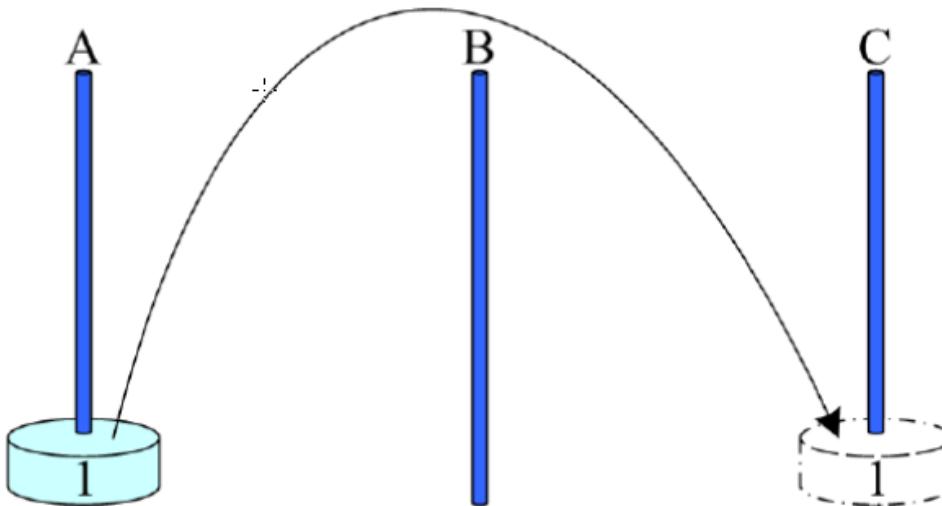


- 為了方便說明解法，所以我們先說明 1 個、2 個及 3 個盤子時的解答。
 - 然後再觀察 n 個盤子時的解答需要 $2^n - 1$ 個步驟。



Case 只有一個盤子：

Step 1：直接將盤子 1 由 A 搬到 C。（完成）



◆ 後續接動畫頁

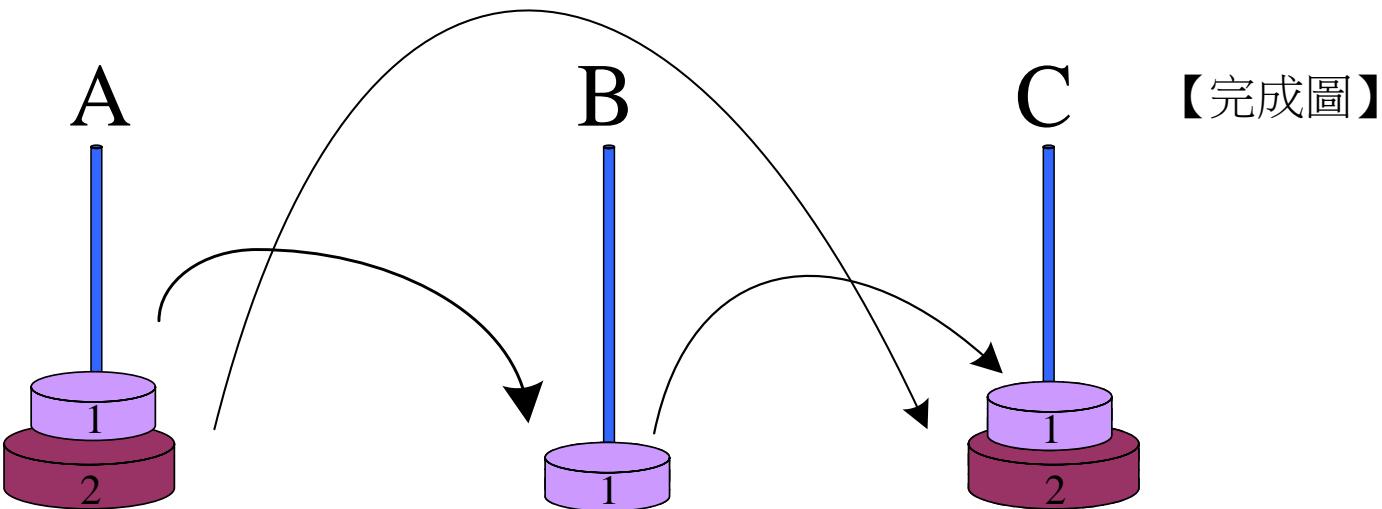


Case 有兩個盤子：

Step1：將盤子1由A搬到B。

Step2：將盤子2由A搬到C。

Step3：將盤子1由B搬到C。





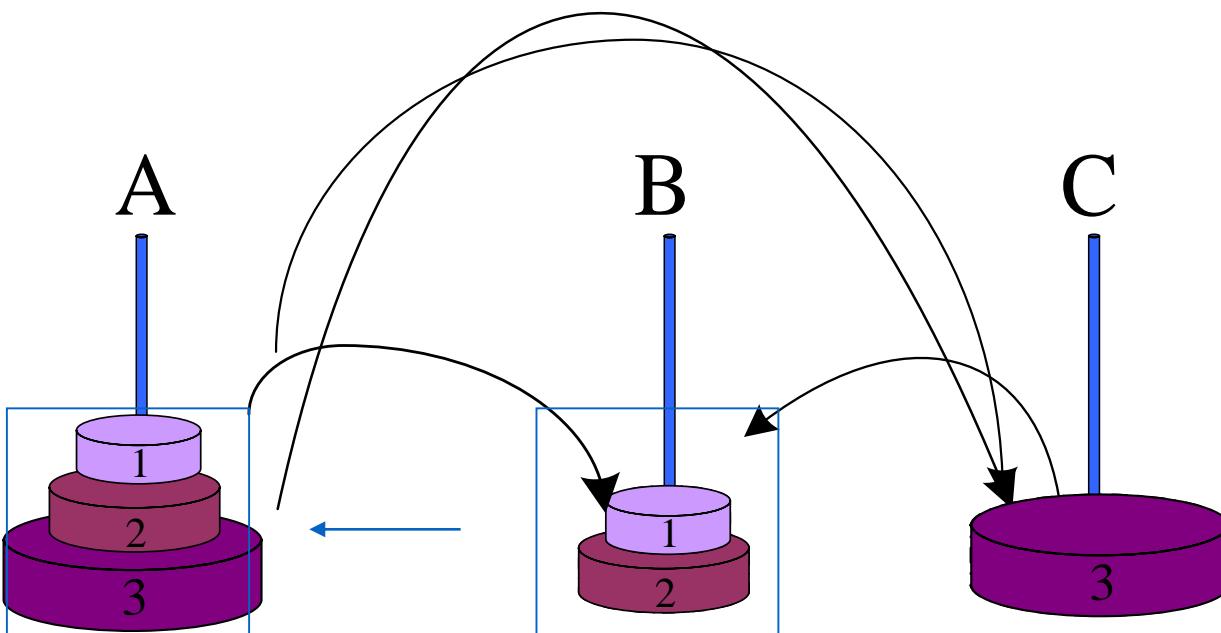
Case 有三個盤子：

Step1：將盤子1由A搬到C。

Step2：將盤子2由A搬到B。

Step3：將盤子1由C搬到B。

Step4：將盤子3由A搬到C。



此時請注意，當我們將B視為來源木樁，A視為暫時木樁（即A,B角色對調），則剩餘的狀況恰好如同『Case有兩個盤子』，因此剩餘步驟應該還有3步如下：

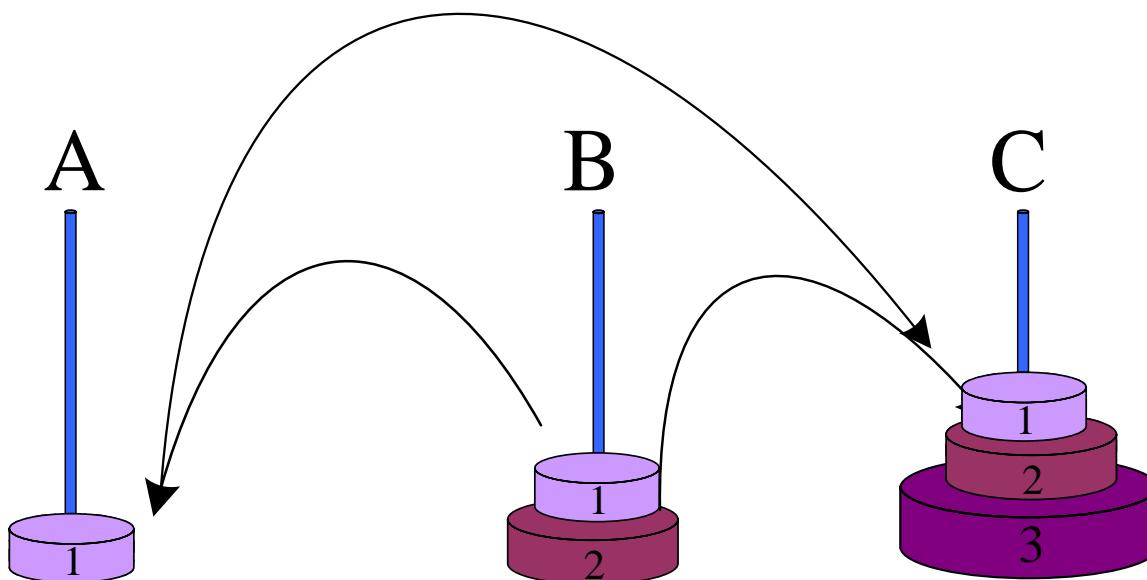


Case 有三個盤子（續）：

Step5：將盤子1由B搬到A。

Step6：將盤子2由B搬到C。

Step7：將盤子1由A搬到C。



【完成圖】



- 在『Case 有三個盤子』中，可以發現搬移三個盤子時，包含只有兩個盤子的解法。換句話說，我們可以得到下列結論：

- 假設現在有 n 個盤子，當我們將最底層的盤子成功地由來源木樁 A 搬移到目標木樁 C 後
 - 代表我們必須先將上面的 $n-1$ 個盤子，由來源木樁 A 搬移到目標木樁 B。
- 剩餘的 $n-1$ 個盤子的解答，必定與剛好有 $n-1$ 個盤子時極度類似
 - 只不過來源木樁與暫時木樁的角色對調而已。

- 我們可以將 n 個盤子的解法策略明確敘述如下：

河內塔 n 個盤子的解法策略
Step 1: 將 $n-1$ 個盤子，從木樁 A 移到木樁 B。
Step 2: 將最大的盤子（第 n 個盤子），從木樁 A 移到木樁 C。
Step 3: 將 $n-1$ 個盤子，從木樁 B 移到木樁 C。

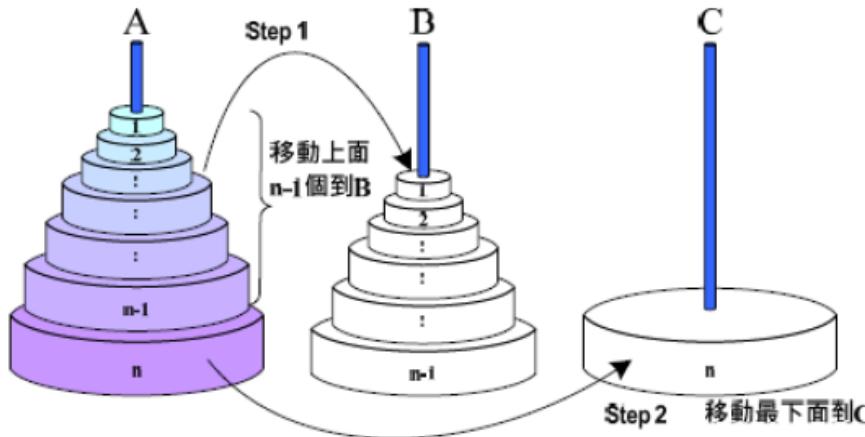


圖 2-2 河內塔遞迴解法之 Step 1 與 Step 2

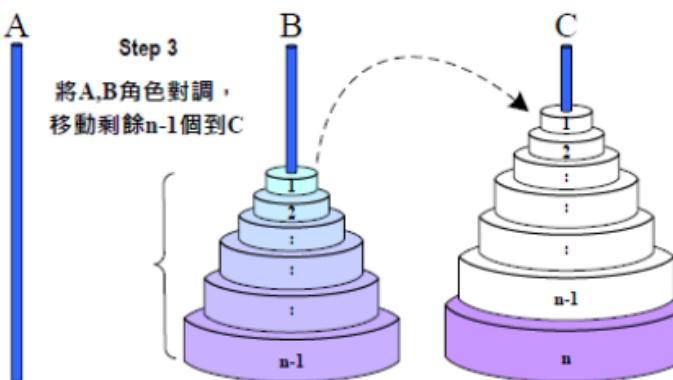


圖 2-3 河內塔遞迴解法之 Step 3



■ 演算法如下：

Input：3 根木樁 a,b,c，n 個盤子由大到小依序疊放在木樁 a 底部。

Output：n 個盤子由大到小依序疊放在木樁 c 底部。

```
1 Procedure hanoi(n,a,b,c)
2   If (n<1) then [print('error'); stop]
3   If (n=1) then
4     [ print(a); print('木樁最上層盤子移動到木樁'); print(c) ]
5   else
6   [
7     hanoi(n-1,a,c,b)
8     print(a); print('木樁最上層盤子移動到木樁'); print(c)
9     hanoi(n-1,b,a,c)
10    ]
11 endProcedure
```



說明

■ 此演算法第 7~9 行恰好對應 n 個盤子的解法策略 Step1, Step2, Step3。

- 其中，hanoi 程序的第 2 個參數 a 為來源木樁，第 3 個參數 b 為暫時木樁，第 4 個參數 c 為目標木樁。



範例

【2-12】試分析 n 個盤子河內塔問題的演算法之時間複雜度，
並撰寫相對應的 C 程式。



解答

1. 本演算法為遞迴演算法，我們可以回到解法策略中，觀察移動 n 個盤子
一共需要幾個步驟。

- 令搬移 n 個盤子，從木樁 A 移到木樁 C，需要 S_n 個步驟。
- 則 $S_1=1$, $S_2=3$, $S_3=7$ ，且 $S_n=S_{n-1}+1+1+S_{n-1}$ 。
- 所以它滿足下列遞迴定義：

$$S_n = \begin{cases} 2S_{n-1}+1 & , n>1 \\ 1 & , n=1 \end{cases} \quad \dots\dots \text{邊界條件}$$

- 如果您對於數字敏感一點，可以觀察到 $1, 3, 7$ 恰為 $2^1-1, 2^2-1, 2^3-1$ ，所以 $S_n=2^n-1$ ($n\geq 1$)，至於詳細的證明，則可利用數學歸納法證明，詳見本書完整版。
- 由於一共需要 2^n-1 個步驟，所以時間複雜度為 $O(2^n)$ 。



2. 下列是其 hanoi 函式，完整程式請見光碟。

```
int hanoi(int n,char T1,char T2,char T3)
{
    if(n<1)
        return 1;
    if(n==1)
        printf("%c 木樁最上層盤子移動到%c 木樁\n",T1,T3);
    else
    {
        hanoi(n-1,T1,T3,T2);
        printf("%c 木樁最上層盤子移動到%c 木樁\n",T1,T3);
        hanoi(n-1,T2,T1,T3);
    }
    return 0;
}
```

程式 2.8



2.2.4 資訊界的難題

- 之前範例的時間複雜度為 $O(n)$ 、 $O(n^2)$ 、 \dots ，這些時間複雜度稱之為**多項式時間複雜度**，其標準格式為 $O(n^k)$ ，其中 k 為常數。
 - 但河內塔演算法則不屬於多項式時間複雜度。
- 「P 問題」、「P 演算法」、「集合 P」
 - 問題最佳解法的演算法之最差狀況屬於多項式時間複雜度，此類問題稱之為「P 問題」
 - P 問題的集合稱為「集合 P」。
 - 能夠提供該問題的多項式時間複雜度演算法稱之為「P 演算法」



- 事實上，有更多的問題目前無法設計出 P 演算法，例如分割問題(Partition problem)。
 - 分割問題的定義是，給定一個整數集合 $S = \{S_1, S_2, \dots, S_n\}$ ，是否能將之分割成兩個子集合，使得其所含數字總和相等？
 - 例如， $S = \{2,3,5\}$ ，則分割為 $S_1 = \{2,3\}$, $S_2=\{5\}$ 就符合要求，而 $S = \{2,3,6\}$ ，則無法分割。而整個問題並不需要您找出分割結果，只要回答能不能分割成功即可，即答案為「Yes」或「No」。
 - 分割問題目前已知最佳解法的時間複雜度為 $O(2^n)$ ，並非一個 P 演算法。
- P 演算法有一個特性，也就是當電腦速度提升後，該演算法對應之程式所需使用的時間將會大幅下降。
- 對於非 P 演算法而言，改良演算法的時間複雜度成為重要課題（因為硬體的速度提升對於解決問題的幫助並不大）。
- 若將所有可找到 P 演算法問題集合起來，該集合稱為集合 P，集合 P 之外還有三個集合，分別是 NP, NP-complete, NP-hard。而這些問題都是資訊科學界的難題。
- 並非所有問題都存在多項式時間解答（或目前為無多項式時間解答），且有些問題是無解的。



2.3 演算法的種類

- 演算法為了解答問題，常使用各種技巧，這些技巧的特性各有不同，適用處也有所不同，我們將在本書的眾多章節中，介紹分屬各類技巧的演算法。
- 以下是演算法的常見技巧：
 - **各個擊破法(Divide-and-Conquer Strategy)：**
 - 各個擊破法將原始問題分割為彼此獨立的眾多子問題，每一個子問題都與原始問題接近但較小些。然後對子問題進行解答後，最後再將子問題的解答合併作為原始問題的最終解答。
 - 通常，演算法在結構上具有遞迴解決子問題者，都是採用本法來解決問題。
 - **動態規劃法(Dynamic Programming Strategy)：**
 - 動態規劃法技巧使用在求問題的最佳解上
 - 它與各個擊破法技巧相似，但它的子問題並非彼此獨立，而是互相分享解的，因此，動態規劃法對於每一個子問題只解決一次，並將解答存入表格中，透過查表以避免再遇到同類問題時重新求出解答。



- 動態規劃法採用的是**由下而上**(Bottom-Up)的設計技巧，每一個上層的結果都必須參考自下層的結果，因此，其輸出必定是整體最佳解。
- **貪婪演算法(Greedy Strategy)**：
 - 貪婪演算法也是求最佳解演算法常使用的技巧之一。
 - 採用的是**由上而下**(Top-Down)的解答技巧，若一個最佳解問題將產生一連串選擇以求得最佳解，則使用貪婪演算法將每一次都選擇區域最佳解，以便求出最終的解答。
 - 貪婪演算法無法保證解答為整體最佳解，它只能保證解答為區域最佳解，但它比動態規劃法較省時間。
- **樹狀搜尋法(Tree searching strategies)**：
 - 問題使用樹狀結構來表達，並使用樹狀結構的相關演算法來求得解答，演算法若使用 Tree searching 技巧，則將會拜訪樹中各節點，以求得解答。
- **其他**：
 - 例如近似解法、隨機解法等。您可以在演算法專書中，看到這些解法的特色，在本書中，我們並未使用這類解法。



2.4 本章重點

- 分析演算法的效能與實際程式的效能有一個很大的不同
 - 在分析程式的效能時，我們常常採用的是實測方式
 - 分析演算法的效能時，則關注在步驟的執行次數與所需記憶體之空間，也就是時間複雜度與空間複雜度。
- 我們常使用漸進表示法 Big-Oh, Ω 及 Θ 來表示演算法的複雜度
 - 因為當演算法的變數很大時，較低次項與係數的影響會相對不重要。
 - 在本書中，我們主要使用的是 Big-Oh 來討論演算法的時間複雜度。
- 常見的 Big-Oh 時間複雜度有 $O(1)$, $O(\log_2 n)$, $O(n)$, $O(n \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $O(n!)$ 等等。
- 若一個問題可以找到一個演算法解答為多項式時間複雜度 $O(n^k)$ 的話，則該問題屬於 P 集合內的問題，若無法找到，則屬於 NP、NP-complete 或 NP-hard 的問題，這些問題都是資訊界的難題。
- 演算法使用的技巧可大致區分為各個擊破法、動態規劃法、貪婪演算法、樹狀搜尋法或其他種類的技巧。

- 遞迴是設計演算法及程式的一種技巧，善用遞迴設計，有時可以使得問題變得簡單而直觀，但在設計遞迴演算法時，必須符合下列兩個條件：
 1. 遞迴演算法必須有邊界條件，當符合邊界條件時，就應該返回。
 2. 遞迴演算法在邏輯上，必須漸漸往邊界條件移動。
- 本章介紹了三種使用遞迴來解答的問題，包含費氏數列、數學階層與河內塔
 - 「河內塔」問題的演算法之時間複雜度為 $O(2^n)$ ，它並非一個 P 演算法。



Exercise 1: Install Weka

Exercise 1: Install Weka



Exercise 2: Weka Decision Tree

Implement Gradient Descent