

Table of Contents

第 21 章 數據分析演算法-KNN 最近鄰居法.....	2
21.1 KNN 數學介紹.....	2
21.2 使用 sklearn 的 KNN 判斷水果種類.....	4
21.3 實戰案例-鳶尾花的種類判斷.....	5
21.3.1 鳶尾花資料下載和存到 Excel 檔案.....	7
21.3.2 使用 KNN 判別鳶尾花的種類.....	8
21.4 實戰案例新聞.....	10
21.5 實際例子 新聞分類.....	15
21.6 專案 交友網站	18
21.7 專案 手寫 OCR 辨識	21
第 22 章 數據分析演算法-決策樹 Decision Tree.....	25
22.1 決策樹.....	25
22.2 如何在 Python 中實現決策樹算法.....	29
22.2.1 1. 基尼指數.....	32
22.2.2 2. 創建分割.....	34
22.2.3 建樹.....	39
22.2.4 做一個預測.....	46
22.2.5 鈔票案例研究.....	48
第 23 章 數據分析演算法-naive-bayes.....	52
23.1 貝氏二元分類 Naïve-Bayes.....	53
23.1.1 演算法思路.....	53
23.1.2 訓練階段.....	55
23.1.3 獨立事件.....	56
23.1.4 字詞分佈模式.....	57
23.1.5 向下溢位.....	58
23.1.6 提高朴素貝葉斯模型 Naive Bayes Model 的力量的提示.....	63
23.2 如何在 Python 中實現朴素的貝葉斯.....	64
23.2.1 5. 準確度.....	74
第 24 章 隨機森林.....	75
1) 樹棲投票.....	76
第 25 章 常見算法優缺點.....	82
25.1 用 Python 做科學計算.....	87
25.2 機器學習和人工智慧.....	87
25.3 分析結果改進工作,結論分析,改善工作 #	90
25.4 SVM.....	93
25.5 演算法 SGD (Stochastic Gradient Descent).....	93
25.6 演算法教學.....	94
25.7 Stochastic Gradient Descent 的 Python 的程式.....	100

第21章 數據分析演算法-KNN 最近鄰居法

21.1 KNN 數學介紹

KNN (K Nearest Neighbor)最近鄰居法又譯 K-近鄰算法，此演算法在非常重要的分析演算法之一，以目前來說是廣泛使用的演算法，用於分類和回歸的非參數統計方法，輸出是分類 KNN 是所有的機器學習算法中最簡單也是使用最廣的演算法之一。

其數學原理透過 k 個最近的鄰居，並依照這些鄰居的分類，決定了賦予該對象的類別。也就是由其鄰居的“多數表決”確定的， k 個最近鄰居 (k 為正整數，通常較小)，KNN 最近鄰居法採用向量空間模型來分類，概念為相同類別的案例，彼此的相似度高，而可以藉由計算與已知類別案例之相似度，來評估未知類別案例可能的分類。

KNN(K-近鄰算法)的缺點是對數據的局部鄰居非常敏感，並且留意 KNN 與 k-means(K-平均算法)沒有任何關係，是二種不同的演算法，請勿混淆。

數學公式和原理：

假設我們有 $(X_1, Y_1), (X_2, Y_2), \dots, (X_n, Y_n)$ 取值 $R^d \times \{1, 2\}$ ，其中 Y 是 X 的類標籤，因此 $X | Y=r \sim P_r$ 而 $r=1, 2$ (和概率分佈 P_r 。給定一些規範

$\|\cdot\|$ 在 R^d 和 $x \in R^d$ ，讓

Python

$(X_{(1)}, Y_{(1)}), \dots, (X_{(n)}, Y_{(n)})$ 訓練數據的重新排序為 $\|X_{(1)} - x\| \leq \dots \|X_{(n)} - x\|$ 。

而計算 KNN 彼此之間的距離，可以用以下的公式：

$$\text{Similarity}(A,B) = \frac{A \cdot B}{\|A\| * \|B\|} = \frac{\sum_{i=1}^n A_i * B_i}{\sqrt{\sum_{i=1}^n A_i^2} * \sqrt{\sum_{i=1}^n B_i^2}}$$

再用剛剛的柳丁和檸檬的範例來看一下 KNN 演算法，透過以下的程式，同樣的也把柳丁和檸檬的大小，把它放在圖表上面，這時候有一個未知的紅色的新的物體，一樣的把該物體的寬度和高度量出來之後，並且在圖表上面用紅色的三角形來表示，這個時候 KNN 就可以拿出來使用了，首先需要先設定好 K 的數量，這裡我們用 K=3，然後以這一個紅色的位置的位置，來尋找一下附近最靠近的三個水果，透過畫出一個灰色的圓形，可以看得出來在這個範圍之中的三個水果都是檸檬，所以大膽的說這一個未知的物體就是檸檬。

範例程式： 01-kNN-Mat.py

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([9,9.2,9.6,9.2,6.7,7,7.6], [9.0,9.2,9.2,9.2,7.1,7.4,7.5 ], 'yx' )
plt.plot([7.2,7.3,7.2,7.3,7.2,7.3,7.3 ], [10.3,10.5,9.2,10.2,9.7,10.1,10.1 ], 'g.' )
plt.plot([7], [9], 'r^' )
circle1=plt.Circle((7,9),1.2,color='#eeeeee')
plt.gcf().gca().add_artist(circle1)
plt.axis([6, 11, 6, 11])
plt.ylabel('H cm')
plt.xlabel('W cm')
plt.legend(('Orange','Lemons'),
          loc='upper right')
plt.show()
```

執行結果

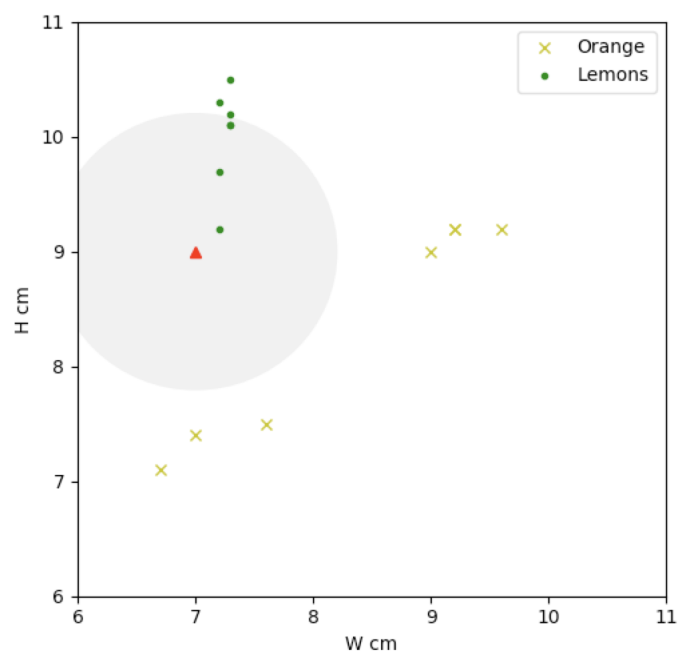


圖 1 執行結果

演算法邏輯

KNN 分類演算法簡單來說就是要找和新數據最近的 **K** 個鄰居，這些鄰居是什麼分類，那麼新數據就是什麼樣的分類。

現在給定一個特徵樣本 (我們稱為訓練集合)，我們輸入一個新樣本要把一個該樣本分藍色、紅色，我們可以從訓練集合找跟這新樣本距離最近的 **K** 個特徵樣本，看這些 **K** 個點是什麼顏色，來決定該點的最終顏色。

21.2 使用 sklearn 的 KNN 判斷水果種類

在本章節的範例之中將要使用 **KNN**，透過收集到的檸檬和柳丁的體積大小寬度和高度之間的訓練資料，並加以判別當新未知的水果量測相關的寬度和高度之後，並使用 **KNN** 的計算法，來判別這個位置水果到底是檸檬還是柳丁。

範例程式

```
1. from sklearn.neighbors import KNeighborsClassifier    #匯入 KNN 函示庫
2. X=[[9,9],[9.2,9.2],[9.6,9.2],[9.2,9.2],[6.7,7.1],[7,7.4],[7.6,7.5],
3.    [7.2,10.3], [7.3,10.5], [7.2,9.2], [7.3,10.2], [7.2,9.7], [7.3,10.1], [7.3,10.1]]
4. y=[1,1,1,1,1,1,1,
5.    2,2,2,2,2,2,2]
6. neigh = KNeighborsClassifier(n_neighbors=3)          #使用 KNN, K=3
7. neigh.fit(X, y)                                     #訓練
8. print("預測答案=",neigh.predict([[7,9]]))           #預測
9. print("預測樣本距離=",neigh.predict_proba([[7,9]])) # 測試數據 X 的返回概率估計。
```

執行結果

預測答案 = [2]

預測樣本距離 = [[0. 1.]]

21.3 實戰案例-鳶尾花的種類判斷

在這一個章節，將用植物數據範例，來探討 KNN 的在農業上的研究，這個植物數據資料來源是由，scikit-learn 函示庫中還附帶一些開發練習時的數據集，load_iris 鳶尾花數據集。

在本章節，將介紹植物學家透過尋找數據分析，對每個鳶尾花進行分類，而本章將會根據萼片和花瓣的長度和寬度測量來分類鳶尾花。

花萼是一朵花中所有萼片的總稱，位於花的最外層，一般是綠色，樣子類似小葉，但也有少數花的花萼樣子類似花瓣，有顏色。花萼在花還是芽時包圍著花，有保護作用。

本章節將會使用 `load_iris` 鳶尾花數據集，這是一個判別花的種類的數據集，主要包括 150 筆數據，4 個屬性值，分別是：

- Sepal Length 花萼長度
- Sepal Width 花萼寬度
- Petal Length 花瓣長度
- Petal Width 花瓣寬度

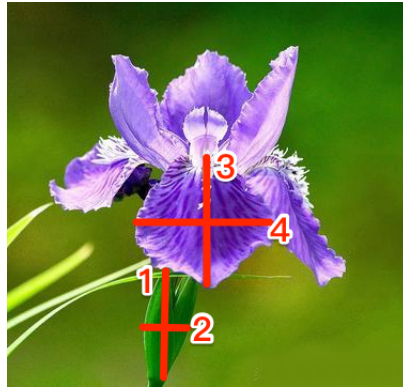


圖 2 鳶尾花的花萼和花瓣

而結果的部分 **Target**，鳶尾花目前有 300 多種，但範例的數據庫將只有以下三種：

- 柔滑鳶尾花 *Iris setosa*
- 弗吉尼亞鳶尾花 *Iris virginica*
- 雜色鳶尾花 *Iris versicolor*



圖 3 左到右，分別是 *setosa*, *virginica*, *versicolor*

透過以下的程式，確認相關函示庫是否有安裝成功，並取得版本編號

21.3.1 鳶尾花資料下載和存到 Excel 檔案

首先將透過以下的程式將資料下載取得，並瞭解這個鳶尾花的數據的樣貌。這個鳶尾花數據的特徵值只有 4 種，而判別的種類 Target 有三種，在本章節將透過 pandas 函式庫，將所取得的數值儲存在 Excel 表之中，這樣的話方便觀看這一個鳶尾花數據的內容。

範例程式 03-Iris.py

```

1. import numpy as np                                # 矩陣函示庫
2. from sklearn import datasets                       # 範例數據函示庫
3. from sklearn.neighbors import KNeighborsClassifier # KNN 函示庫
4.
5. # 取得鳶尾花的數據
6. iris = datasets.load_diabetes()
7. print("iris.data.shape=",iris.data.shape)         # 輸出(150, 4)
8. print("dir(iris)",dir(iris))
   # 輸出['DESCR', 'data', 'feature_names', 'target', 'target_names']
9. print("iris.target.shape=",iris.target.shape)      # 輸出 (150,)
10. try:
11.     print("iris .feature_names=",iris .feature_names) # 顯示特徵值名稱
12. except:
13.     print("No iris.feature_names=")
14. import xlswriter                                # Excel 函示庫
15. import pandas as pd                             # pandas 函示庫
16. # 轉換資料型態
17. try:
18.     df = pd.DataFrame(iris .data, columns=iris .feature_names) # 處理特徵值
19. except:
20.     df = pd.DataFrame(iris .data, columns= ['sepal length (cm)', 'sepal width (cm)', 'petal
        length (cm)', 'petal width (cm)'])
21. df['target'] = iris.target                         # 處理結果 Target
22.
23. #print(df.head())                                # 顯示前五筆資料
24. df.to_csv("iris .csv", sep='\t')                  # 儲存到 CSV
25.

```

```

26. writer = pd.ExcelWriter('iris.xlsx', engine='xlsxwriter')      # 儲存到 Excel
27. df.to_excel(writer, sheet_name='Sheet1')
28. writer.save()

```

執行結果

```
iris.data.shape= (150, 4)
```

```
dir(iris) ['DESCR', 'data', 'feature_names', 'target', 'target_names']
```

Backend TkAgg is interactive backend. Turning interactive mode on.

```
iris.target.shape= (150,)
```

```
iris.feature_names= ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

	A	B	C	D	E	F
1		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
2	0	5.1	3.5	1.4	0.2	0
3	1	4.9	3	1.4	0.2	0
4	2	4.7	3.2	1.3	0.2	0
5	3	4.6	3.1	1.5	0.2	0
6	4	5	3.6	1.4	0.2	0
7	5	5.4	3.9	1.7	0.4	0
8	6	4.6	3.4	1.4	0.3	0
9	7	5	3.4	1.5	0.2	0
10	8	4.4	2.9	1.4	0.2	0
11	9	4.9	3.1	1.5	0.1	0
12	10	5.4	3.7	1.5	0.2	0
13	11	4.8	3.4	1.6	0.2	0
14	12	4.8	3	1.4	0.1	0
15	13	4.3	3	1.1	0.1	0
16	14	5.8	4	1.2	0.2	0
17	15	5.7	4.4	1.5	0.4	0
18	16	5.4	3.9	1.3	0.4	0
19	17	5.1	3.5	1.4	0.3	0
20	18	5.7	3.8	1.7	0.3	0

圖 4 執行結果

21.3.2 使用 KNN 判別鳶尾花的種類

在章節中將透過 KNN 的方法，訓練已知的鳶尾花的種類，找出其關聯性，並且預測出未知的鳶尾花，並預測該鳶尾花的種類。

範例程式 09-LinearRegression-diabetes.py

```

1. import matplotlib.pyplot as plt          # 繪圖函示庫
2. import numpy as np                      # 矩陣函示庫
3. from sklearn import datasets           # 範例數據函示庫
4. from sklearn.neighbors import KNeighborsClassifier # KNN 函示庫
5. from sklearn.model_selection import train_test_split # 切割資料函示庫
6.
7. iris = datasets.load_iris() # 取得鳶尾花的數據
8.
9. # 切割 80%訓練和 20%的測試資料
10. iris_X_train , iris_X_test , iris_y_train , iris_y_test =
    train_test_split(iris.data,iris.target,test_size=0.2)
11.
12. # 研究和計算
13. knn = KNeighborsClassifier()            # 建立 KNN
14. knn.fit(iris_X_train, iris_y_train)     # 訓練
15.
16. print("預測",knn.predict(iris_X_test))
17. print("實際",iris_y_test)
18. print('準確率: %.2f' % knn.score(iris_X_test, iris_y_test))

```

執行結果:

預測 [1 0 2 1 0 2 2 1 1 2 1 0 0 1 0 0 1 2 2 2 2 2 0 1 2 0 0 2 1 1]

實際 [1 0 2 1 0 2 2 1 1 2 1 0 0 1 0 0 1 2 2 1 2 2 0 2 2 0 0 2 1 1]

準確率: 0.93

為什麼準確率只有 93% ? 這個程式所判別出來的預測結果，還是會有一筆的答案和實際是不一樣的，在實際的分類的資料很難會出現 100%，這就是數理統計實際的情況，所以改善的方法需要再補充大量的數據讓準確率再更精準一點，準確能夠再高一些。

21.4 實戰案例新聞

以下我們使用新聞分類來作為演算法範例，主要目標為將新聞依照內容說明來自動分類為汽車(C)、運動(S)、科技(T)。

實作步驟

前置作業

首先，我們要找特徵跟準備特徵樣本！！讓我們先理解這兩個名詞是什麼：

1.

Q: 什麼叫**特徵 (Feature)**?

A: 特徵就是那些能讓我們判斷分類的屬性，例如我們要分類男生、女生，一般是用是否有喉結、生殖器的不同來判斷，那麼有喉結、有生殖器就是我們所稱的特徵。

2.

Q: 什麼叫**特徵樣本 (訓練集合 Train Set)**?

A: 由特徵值和**正確分類**組成的集合，例如: (有喉結 = 男), (有小鳥 = 男)，這個集合用來告訴機器哪些特徵數據它的分類是什麼，讓機器可以去做學習。

對於新聞分類這問題，我們就可以找一些關鍵字作為各分類的特徵，我們稱做**特徵關鍵字 (Feature Keywords)**，這邊我們找的特徵關鍵字如下：

分類	關鍵字
汽車 C	賓士 寶馬
運動 S	籃球 路跑
科技 T	手機 App

接著，我們先找分類正確的新聞，而每則新聞都有內文，我們把內文去做切字，比對特徵關鍵字、找出該字詞出現的頻率，最後形成一個**特徵關鍵字字詞頻率向量** (以下用 **TF 向量** 簡稱)，每個 TF 向量再給它加上正確的分類，形成我們的特徵樣本如下：

新聞	分類	賓士	寶馬	籃球	路跑	手機	App
C63 發表會	P	15	25	0	5	8	3
BMW i8	P	35	40	1	3	3	2
林書豪	S	5	0	35	50	0	0
湖人隊	S	1	5	32	15	0	0
Android 5.0	T	10	5	7	0	2	30
iPhone6	T	5	5	5	15	8	32

C63 發表會= $\langle 15, 25, 0, 5, 8, 3 \rangle$

Python

*BMW*8 = <35,40,1,3,3,2>
林書豪 = <5,0,35,50,0,0>
湖人隊 = <1,5,32,15,0,0>
*Android*5.0 = <10,5,7,0,2,30>
*iPhone*6 = <5,5,5,15,8,32>

訓練集合準備完成。

這些向量就是上圖的藍色、紅色的點。

新的先文輸入後一樣找出 TF 向量，但是我們不曉得正確的分類為何，等等跑完 KNN 分類後才會知道結果。

地點	分類	賓士	寶馬	籃球	路跑	手機	App
騎士隊	?	10	2	50	56	8	5
騎士隊	=<10,2,50,56,8,5>						

這個向量就是上圖的問號點。

計算階段

分類演算法就是要先找和新樣本 騎士隊 距離最近 K 個特徵樣本，這邊距離就是 TF 向量的距離，我們用

Cosine Similarity 作為距離計算公式，公式如下圖，即是向量的內積除以向量的長度。

$$\text{similarity}(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n A_i^2} \times \sqrt{\sum_{i=1}^n B_i^2}}$$

對於我們來說就是要計算 騎士隊 和 C63 發表會 到 iPhone6 所有 TF 向量距離。

C63 發表會 = <15,25,0,5,8,3>
騎士隊 = <10,2,50,56,8,5>

$$\overrightarrow{C63發表會} \cdot \overrightarrow{騎士隊} = \frac{(15 \cdot 10 + 25 \cdot 2 + 0 \cdot 50 + 5 \cdot 56 + 8 \cdot 8 + 3 \cdot 5)}{(\sqrt{15^2 + 25^2 + 0^2 + 5^2 + 8^2 + 3^2} + \sqrt{10^2 + 2^2 + 50^2 + 56^2 + 8^2 + 5^2})}$$

這邊我們計算完所有向量距離並排序後分別為：

$$\begin{aligned} \overrightarrow{C63發表會} \cdot \overrightarrow{騎士隊} &= 0.237799321731 \\ \overrightarrow{BMW i8} \cdot \overrightarrow{騎士隊} &= 0.167385315225 \\ \overrightarrow{林書豪} \cdot \overrightarrow{騎士隊} &= 0.98388688183 \\ \overrightarrow{湖人隊} \cdot \overrightarrow{騎士隊} &= 0.902366548062 \\ \overrightarrow{Android 5.0} \cdot \overrightarrow{騎士隊} &= 0.249728394512 \\ \overrightarrow{iPhone 6} \cdot \overrightarrow{騎士隊} &= 0.483053209513 \end{aligned}$$

計算完所有向量距離後，我們找出 $K=3$ ，3 個距離最短的向量距離，然後分別看他們的分類為何，來決定最後新地點的分類。

範例程式 02-knn_blog_basic.py

```
1. #!/usr/bin/python
2. # -*- coding: utf-8 -*-
3. import math
4. def create_trainset():
5.     """
6.     產生訓練集合。
7.     :return:
8.     """
```

```

9.     trainset_tf = dict()
10.    trainset_tf[u'0'] = (0,1,1)
11.    trainset_tf[u'1'] = (1,1,1)
12.    trainset_tf[u'2'] = (2,1,1)
13.    trainset_tf[u'3'] = (3,1,1)
14.    trainset_class = dict()
15.    trainset_class[u'0'] = '0'
16.    trainset_class[u'1'] = '0'
17.    trainset_class[u'2'] = '1'
18.    trainset_class[u'3'] = '1'
19.    return trainset_tf, trainset_class
20. def cosine_similarity(v1, v2):
21.     """
22.     計算兩個向量的正弦相似度。距離越近，相似度數值會越高。
23.     :param v1:
24.     :param v2:
25.     :return:
26.     """
27.     sum_xx, sum_xy, sum_yy = 0.0, 0.0, 0.0
28.     for i in range(0, len(v1)):
29.         sum_xx += math.pow(v1[i], 2)
30.         sum_xy += v1[i] * v2[i]
31.         sum_yy += math.pow(v2[i], 2)
32.     return sum_xy / math.sqrt(sum_xx * sum_yy)
33. def knn_classify(input_tf, trainset_tf, trainset_class, k):
34.     """
35.     執行 kNN 分類演算法
36.     :param input_tf: 輸入向量
37.     :param trainset_tf: 訓練集合向量
38.     :param trainset_class: 訓練集合分類
39.     :param k: 取 k 個最近鄰居
40.     :return:
41.     """
42.     tf_distance = dict()
43.     # 計算每個訓練集合特徵關鍵字詞頻率向量和輸入向量的距離
44.     print('(1) 計算向量距離')
45.     for place in trainset_tf.keys():
46.         tf_distance[place] = cosine_similarity(trainset_tf.get(place), input_tf)
47.         print('\tTF(%s) = %f' % (place, tf_distance.get(place)))
48.     # 把距離排序，取出 k 個最近距離的分類
49.     class_count = dict()
50.     print('(2) 取 K 個最近鄰居的分類, k = %d' % k)
51.     for i, place in enumerate(sorted(tf_distance, key=tf_distance.get, reverse=True)):
52.         current_class = trainset_class.get(place)

```

```
53.     print('\tTF(%s) = %f, class = %s' % (place, tf_distance.get(place), current_class))
54.     class_count[current_class] = class_count.get(current_class, 0) + 1
55.     if (i + 1) >= k:
56.         break
57.     print('(3) K 個最近鄰居分類出現頻率最高的分類當作最後分類')
58.     input_class = ''
59.     for i, c in enumerate(sorted(class_count, key=class_count.get, reverse=True)):
60.         if i == 0:
61.             input_class = c
62.         print('\t%s, %d' % (c, class_count.get(c)))
63.     print('(4) 分類結果 = %s' % input_class)
64. if __name__ == '__main__':
65.     input_tf = (1.2,1,1)
66.     trainset_tf, trainset_class = create_trainset()
67.     knn_classify(input_tf, trainset_tf, trainset_class, k=3)
68.
```

執行結果

```
(1) 計算向量距離
    TF(0) = 0.762493
    TF(1) = 0.996116
    TF(2) = 0.968496
    TF(3) = 0.910359
(2) 取 K 個最近鄰居的分類, k = 3
    TF(1) = 0.996116, class = 0
    TF(2) = 0.968496, class = 1
    TF(3) = 0.910359, class = 1
(3) K 個最近鄰居分類出現頻率最高的分類當作最後分類
    1, 2
    0, 1
(4) 分類結果 = 1
```

21.5 實際例子 新聞分類

程式

1. 首先第一步我們先準備好特徵樣本(訓練集合)。

首先第一步我們先準備好特徵樣本(訓練集合)。

範例程式 03-knn_blog.py

```

1.  #!/usr/bin/python
2.  # -*- coding: utf-8 -*-
3.  import math
4.  def create_trainset():
5.      """
6.      產生訓練集合。
7.      :return:
8.      """
9.      trainset_tf = dict()
10.     trainset_tf[u'C63 發表會'] = (15, 25, 0, 5, 8, 3)
11.     trainset_tf[u'BMW i8'] = (35, 40, 1, 3, 3, 2)
12.     trainset_tf[u'林書豪'] = (5, 0, 35, 50, 0, 0)
13.     trainset_tf[u'湖人隊'] = (1, 5, 32, 15, 0, 0)
14.     trainset_tf[u'Android 5.0'] = (10, 5, 7, 0, 2, 30)
15.     trainset_tf[u'iPhone6'] = (5, 5, 5, 15, 8, 32)
16.     trainset_class = dict()
17.     trainset_class[u'C63 發表會'] = 'P'
18.     trainset_class[u'BMW i8'] = 'P'
19.     trainset_class[u'林書豪'] = 'S'
20.     trainset_class[u'湖人隊'] = 'S'
21.     trainset_class[u'Android 5.0'] = 'T'
22.     trainset_class[u'iPhone6'] = 'T'
23.     return trainset_tf, trainset_class
24. def cosine_similarity(v1, v2):
25.     """
26.     計算兩個向量的正弦相似度。距離越近，相似度數值會越高。
27.     :param v1:

```

```

28. :param v2:
29. :return:
30. """
31. sum_xx, sum_xy, sum_yy = 0.0, 0.0, 0.0
32. for i in range(0, len(v1)):
33.     sum_xx += math.pow(v1[i], 2)
34.     sum_xy += v1[i] * v2[i]
35.     sum_yy += math.pow(v2[i], 2)
36. return sum_xy / math.sqrt(sum_xx * sum_yy)
37. def knn_classify(input_tf, trainset_tf, trainset_class, k):
38.     """
39.     執行 kNN 分類演算法
40.     :param input_tf: 輸入向量
41.     :param trainset_tf: 訓練集合向量
42.     :param trainset_class: 訓練集合分類
43.     :param k: 取 k 個最近鄰居
44.     :return:
45.     """
46.     tf_distance = dict()
47.     # 計算每個訓練集合特徵關鍵字詞頻率向量和輸入向量的距離
48.     print('(1) 計算向量距離')
49.     for place in trainset_tf.keys():
50.         tf_distance[place] = cosine_similarity(trainset_tf.get(place), input_tf)
51.         print('\tTF(%s) = %f' % (place, tf_distance.get(place)))
52.     # 把距離排序，取出 k 個最近距離的分類
53.     class_count = dict()
54.     print('(2) 取 K 個最近鄰居的分類, k = %d' % k)
55.     for i, place in enumerate(sorted(tf_distance, key=tf_distance.get, reverse=True)):
56.         current_class = trainset_class.get(place)
57.         print('\tTF(%s) = %f, class = %s' % (place, tf_distance.get(place), current_class))
58.         class_count[current_class] = class_count.get(current_class, 0) + 1
59.         if (i + 1) >= k:
60.             break
61.     print('(3) K 個最近鄰居分類出現頻率最高的分類當作最後分類')
62.     input_class = ""
63.     for i, c in enumerate(sorted(class_count, key=class_count.get, reverse=True)):
64.         if i == 0:
65.             input_class = c
66.         print('\t%s, %d' % (c, class_count.get(c)))
67.     print('(4) 分類結果 = %s' % input_class)
68. if __name__ == '__main__':
69.     input_tf = (10, 2, 50, 56, 8, 5)
70.     trainset_tf, trainset_class = create_trainset()
71.     knn_classify(input_tf, trainset_tf, trainset_class, k=3)

```


72.

執行結果

```

(1) 計算向量距離
    TF(C63 發表會) = 0.237799
    TF(BMW i8) = 0.167385
    TF(林書豪) = 0.983887
    TF(湖人隊) = 0.902367
    TF(Android 5.0) = 0.249728
    TF(iPhone6) = 0.483053
(2) 取 K 個最近鄰居的分類, k = 3
    TF(林書豪) = 0.983887, class = S
    TF(湖人隊) = 0.902367, class = S
    TF(iPhone6) = 0.483053, class = T
(3) K 個最近鄰居分類出現頻率最高的分類當作最後分類
    S, 2
    T, 1
(4) 分類結果 = S

```

```

[powens-MacBook-Air:ch14_jieba powenko$ python /Users/powenko/Desktop/Python/share/SampleV2/SampleV2/ch25-kNN/
01-knn-湖人隊/knn-2.py
(1) 計算向量距離
    TF(Android 5.0) = 0.249728
    TF(湖人隊) = 0.902367
    TF(iPhone6) = 0.483053
    TF(BMW i8) = 0.167385
    TF(C63發表會) = 0.237799
    TF(林書豪) = 0.983887
(2) 取K個最近鄰居的分類, k = 3
    TF(林書豪) = 0.983887, class = S
    TF(湖人隊) = 0.902367, class = S
    TF(iPhone6) = 0.483053, class = T
(3) K個最近鄰居分類出現頻率最高的分類當作最後分類
    S, 2
    T, 1
(4) 分類結果 = S

```

經過計算分類後我們可以知道「騎士隊」的分類是屬於 S，也就是運動。(這邊我們不討論真實狀況，因為計算都非真實數據)

結論

kNN 分類演算法是一個基於實例(instance-based)的演算法，所以特徵樣本的好壞和樣本當中個分類的數量深深影響分類結果的準度，如果特徵樣本當中的 A 分類數量遠大於 B 分類，那麼我們可以預期 K 個最近距離的鄰居也會有很高的機率是 A 分類，這樣就會分類失去準度。

整體來說，優點在於簡單、不需要輸入資料的假設、對於異常值不敏感，而缺點在於計算量大、非常耗時，而且因為要載入所有特徵集合加入距離計算，所以記憶體空間用量也非常大。

可探討的議題有「參數 K 值如何取」、「如何提升計算效能」...等等，這些都有一些相關的書籍或論文都有探討，有興趣研究的可以自行搜尋。

21.6 專案 交友網站

範例程式 04-kNN_LikeitOrNot.py

```
1. #!/usr/bin/python
2. # -*- coding: utf-8 -*-
3. __author__ = 'enginebai'
4. from numpy import *
5. import operator
6. def create_data_set():
7.     """
8.     產生訓練集合。
9.     :return:
10.    """
11.    group = array([[1.0, 1.1], [1.0, 1.0], [0, 0], [0, 0.1]])
12.    labels = ['A', 'A', 'B', 'B']
13.    return group, labels
14. def knn_classify(input, data_set, labels, k):
```

```

15. """
16. 執行 kNN 分類演算法
17. :param input: 輸入新樣本
18. :param data_set: 訓練集合樣本
19. :param labels: 訓練集合分類
20. :param k: k 值
21. :return: 新樣本分類
22. """
23. # calculate the distance (cosine similarity)
24. data_set_size = data_set.shape[0]
25. diff_mat = tile(input, (data_set_size, 1)) - data_set
26. sqrt_diff_mat = diff_mat ** 2
27. sqrt_distance = sqrt_diff_mat.sum(axis=1)
28. distances = sqrt_distance ** 0.5
29. sorted_distance_indicies = distances.argsort()
30. # find k nearest neighbor
31. class_count = {}
32. for i in range(k):
33.     vote_label = labels[sorted_distance_indicies[i]]
34.     class_count[vote_label] = class_count.get(vote_label, 0) + 1
35.     sorted_class_count = sorted(class_count.iteritems(), key=operator.itemgetter(1),
reverse=True)
36. # return the knn label
37. return sorted_class_count[0][0]
38. def file2matrix(file_name):
39. """
40. 將輸入檔案的數值轉成 NumPy 可以解析的物件格式。
41. :param file_name:
42. :return:
43. """
44. with open(file_name, 'r') as f:
45.     array_lines = f.readlines()
46.     number_lines = len(array_lines)
47.     num_matrix = zeros((number_lines, 3))
48.     class_label_vector = []
49.     index = 0
50.     for line in array_lines:
51.         line = line.strip()
52.         list_from_line = line.split('\t')
53.         num_matrix[index, :] = list_from_line[0:3]
54.         label_text = list_from_line[-1]
55.         label = 0
56.         if label_text == 'didntLike':
57.             label = 1

```

```

58.         elif label_text == 'smallDoses':
59.             label = 2
60.         elif label_text == 'largeDoses':
61.             label = 3
62.         class_label_vector.append(label)
63.         index += 1
64.     return num_matrix, class_label_vector
65. def normalize(data_set):
66.     """
67.     Normalize the data set value to range(0, 1)
68.     :param data_set:
69.     :return:
70.     """
71.     min_val = data_set.min(0)
72.     max_val = data_set.max(0)
73.     range_val = max_val - min_val
74.     normalize_data_set = zeros(shape(data_set))
75.     m = data_set.shape[0]
76.     normalize_data_set = data_set - tile(min_val, (m, 1))
77.     normalize_data_set = normalize_data_set / tile(range_val, (m, 1))
78.     return normalize_data_set, range_val, min_val
79. def knn_classifier_test():
80.     # 訓練集資料用來當作測試的比例
81.     test_ratio = 0.05
82.     dating_data_matrix, dating_labels = file2matrix('data/datingTestSet.txt')
83.     normalize_dating_data_matrix, ranges, min_val = normalize(dating_data_matrix)
84.     print(normalize_dating_data_matrix)
85.     import matplotlib.pyplot as plot
86.     fig = plot.figure()
87.     ax = fig.add_subplot(111)
88.     ax.scatter(normalize_dating_data_matrix[:, 0], normalize_dating_data_matrix[:, 2],
89.               15.0 * array(dating_labels), 15.0 * array(dating_labels))
90.     plot.show()
91.     m = normalize_dating_data_matrix.shape[0]
92.     num_test = int(m * test_ratio)
93.     err_count = 0.0
94.     for i in range(num_test):
95.         classify_result =
96.         knn_classify(normalize_dating_data_matrix[i, :], normalize_dating_data_matrix[num_test:
97.         t: m, :], dating_labels[num_test:m], 7)
98.         print('Classifier = %d, real answer = %d' % (classify_result, dating_labels[i]))
99.         if classify_result != dating_labels[i]:
100.             err_count += 1
101.         print('[X]')

```

```

100.     else:
101.         print('[O]')
102.     print("Total error rate = %.2f%%" % (err_count / float(num_test) * 100.0))
103. def classify_person():
104.     result_list = ['not at all', 'a little', 'very like']
105.     play_ratio = float(input('Enter play ratio >> '))
106.     flier_mile = float(input('Enter flier miles >> '))
107.     ice_cream = float(input('Enter liters of ice cream >> '))
108.     dating_data_matrix, dating_labels = file2matrix('data/datingTestSet2.txt')
109.     normalize_dating_data_matrix, ranges, min_val = normalize(dating_data_matrix)
110.     input = array([flier_mile, play_ratio, ice_cream])
111.         classify_result = knn_classify((input - min_val) / ranges,
        normalize_dating_data_matrix,
112.         dating_labels, 3)
113.     print('You will like this person = %s' % result_list[classify_result - 1])
114. if __name__ == '__main__':
115.     knn_classifier_test()
116.     classify_person()
117.

```

執行結果

```

Classifier = 1, real answer = 1
[0]
Classifier = 2, real answer = 2
[0]
Total error rate = 2.00%
You will like this person = very like

```

圖 執行結果

21.7 專案 手寫 OCR 辨識

範例程式 05-kNN-sample-handWrite\KNN.py

```

1.  """
2.  Created on Sep 16, 2010
3.  kNN: k Nearest Neighbors

```

```

4. Input:  inX: vector to compare to existing dataset (1xN)
5.         dataSet: size m data set of known vectors (NxM)
6.         labels: data set labels (1xM vector)
7.         k: number of neighbors to use for comparison (should be an odd number)
8.
9. Output:  the most popular class label
10. @author: pbharrin
11. """
12. from numpy import *
13. import operator
14. from os import listdir
15. def classify0(inX, dataSet, labels, k):
16.     dataSetSize = dataSet.shape[0]
17.     diffMat = tile(inX, (dataSetSize,1)) - dataSet
18.     sqDiffMat = diffMat**2
19.     sqDistances = sqDiffMat.sum(axis=1)
20.     distances = sqDistances**0.5
21.     sortedDistIndicies = distances.argsort()
22.     classCount={}
23.     for i in range(k):
24.         votelabel = labels[sortedDistIndicies[i]]
25.         classCount[votelabel] = classCount.get(votelabel,0) + 1
26.     try:
27.         sortedClassCount = sorted(classCount.iteritems(), key=operator.itemgetter(1),
reverse=True)
28.     except: #python 3.x
29.         sortedClassCount = sorted(classCount.items(), key=operator.itemgetter(1),
reverse=True)
30.     return sortedClassCount[0][0]
31. def createDataSet():
32.     group = array([[1.0,1.1],[1.0,1.0],[0,0],[0,0.1]])
33.     labels = ['A','A','B','B']
34.     return group, labels
35. def file2matrix(filename):
36.     fr = open(filename)
37.     numberOfLines = len(fr.readlines())    #get the number of lines in the file
38.     returnMat = zeros((numberOfLines,3))   #prepare matrix to return
39.     classLabelVector = []                  #prepare labels return
40.     fr = open(filename)
41.     index = 0
42.     for line in fr.readlines():
43.         line = line.strip()
44.         listFromLine = line.split('\t')
45.         returnMat[index,:] = listFromLine[0:3]

```

```

46.     classLabelVector.append(int(listFromLine[-1]))
47.     index += 1
48.     return returnMat,classLabelVector
49.
50. def autoNorm(dataSet):
51.     minVals = dataSet.min(0)
52.     maxVals = dataSet.max(0)
53.     ranges = maxVals - minVals
54.     normDataSet = zeros(shape(dataSet))
55.     m = dataSet.shape[0]
56.     normDataSet = dataSet - tile(minVals, (m,1))
57.     normDataSet = normDataSet/tile(ranges, (m,1)) #element wise divide
58.     return normDataSet, ranges, minVals
59.
60. def datingClassTest():
61.     hoRatio = 0.50    #hold out 10%
62.     datingDataMat,datingLabels = file2matrix('datingTestSet2.txt')    #load data setfrom
        file
63.     normMat, ranges, minVals = autoNorm(datingDataMat)
64.     m = normMat.shape[0]
65.     numTestVecs = int(m*hoRatio)
66.     errorCount = 0.0
67.     for i in range(numTestVecs):
68.                                     classifierResult =
        classify0(normMat[i:],normMat[numTestVecs:m,:],datingLabels[numTestVecs:m],3)
69.     print("the classifier came back with: %d, the real answer is: %d" % (classifierResult,
        datingLabels[i]))
70.     if (classifierResult != datingLabels[i]): errorCount += 1.0
71.     print ("the total error rate is: %f" % (errorCount/float(numTestVecs)))
72.     print(errorCount)
73.
74. def img2vector(filename):
75.     returnVect = zeros((1,1024))
76.     fr = open(filename)
77.     for i in range(32):
78.         lineStr = fr.readline()
79.         for j in range(32):
80.             returnVect[0,32*i+j] = int(lineStr[j])
81.     return returnVect
82. def handwritingClassTest():
83.     hwLabels = []
84.     trainingFileList = listdir('trainingDigits')    #load the training set
85.     m = len(trainingFileList)
86.     trainingMat = zeros((m,1024))

```

```

87.     for i in range(m):
88.         fileNameStr = trainingFileList[i]
89.         fileStr = fileNameStr.split('.')[0]    #take off .txt
90.         classNumStr = int(fileStr.split('_')[0])
91.         hwLabels.append(classNumStr)
92.         trainingMat[i,:] = img2vector('trainingDigits/%s' % fileNameStr)
93.     testFileList = listdir('testDigits')    #iterate through the test set
94.     errorCount = 0.0
95.     mTest = len(testFileList)
96.     for i in range(mTest):
97.         fileNameStr = testFileList[i]
98.         if i == 0:
99.             print(fileNameStr)
100.        fileStr = fileNameStr.split('.')[0]    #take off .txt
101.        if i == 0:
102.            print(fileStr)
103.        classNumStr = int(fileStr.split('_')[0])
104.        if i == 0:
105.            print(classNumStr)
106.        vectorUnderTest = img2vector('testDigits/%s' % fileNameStr)
107.        classifierResult = classify0(vectorUnderTest, trainingMat, hwLabels, 3)
108.        print("the classifier came back with: %d, the real answer is: %d" % (classifierResult,
            classNumStr))
109.        if (classifierResult != classNumStr): errorCount += 1.0
110.        print("\nthe total number of errors is: %d" % errorCount)
111.        print("\nthe total error rate is: %f" % (errorCount/float(mTest)))
112. handwritingClassTest()
113. """
114. testVector = img2vector('testDigits/1_74.txt')
115. print(testVector[0,0:32])
116. print(testVector[0,32:64])
117. """
118.

```

執行結果


```
the classifier came back with: 7, the real answer is: 7
the classifier came back with: 7, the real answer is: 7
the classifier came back with: 4, the real answer is: 4
the classifier came back with: 2, the real answer is: 2
the classifier came back with: 5, the real answer is: 5
the classifier came back with: 5, the real answer is: 5
the classifier came back with: 2, the real answer is: 2
the classifier came back with: 6, the real answer is: 6
the classifier came back with: 6, the real answer is: 6
the classifier came back with: 8, the real answer is: 8
the classifier came back with: 1, the real answer is: 1
the classifier came back with: 1, the real answer is: 1
the classifier came back with: 8, the real answer is: 8
the classifier came back with: 8, the real answer is: 8
the classifier came back with: 0, the real answer is: 0
```

圖 執行結果

第22章 數據分析演算法-決策樹 Decision

Tree

22.1 決策樹

決策論中（如風險管理），決策樹（Decision tree）由一個決策圖和可能的結果（包括資源成本和風險）組成，用來創建到達目標的規劃。決策樹建立並用來輔助決策，是一種特殊的樹結構。決策樹是一個利用像樹一樣的圖形或決策模型的決策支援工具，包括隨機事件結果，資源代價和實用性。它是一個算法顯示的方法。決策樹經常在運籌學中使用，特別是在決策解析中，它幫助確

定一個能最可能達到目標的策略。如果在實際中，決策不得不在沒有完備知識的情況下被在線採用，一個決策樹應該平行機率模型作為最佳的選擇模型或在線選擇模型算法。決策樹的另一個使用是作為計算條件機率的說明性手段。

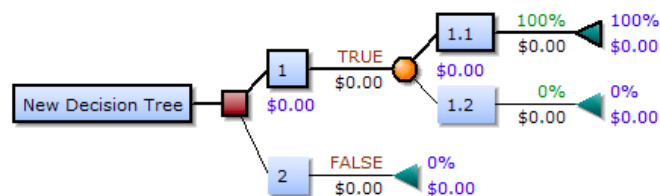
簡介

機器學習中，決策樹是一個預測模型；他代表的是對象屬性與對象值之間的一種對映關係。樹中每個節點表示某個對象，而每個分叉路徑則代表的某個可能的屬性值，而每個葉結點則對應從根節點到該葉節點所經歷的路徑所表示的對象的值。決策樹僅有單一輸出，若欲有複數輸出，可以建立獨立的決策樹以處理不同輸出。數據挖掘中決策樹是一種經常要用到的技術，可以用於解析數據，同樣也可以用來作預測。

從數據產生決策樹的機器學習技術叫做決策樹學習,通俗說就是決策樹。

一個決策樹包含三種類型的節點：

- 決策節點：通常用矩形框來表示
- 機會節點：通常用圓圈來表示
- 終結點：通常用三角形來表示



決策樹學習也是數據挖掘中一個普通的方法。在這裡，每個決策樹都表述了一種樹型結構，它由它的分支來對該類型的對象依靠屬性進行分類。每個決策樹可以依靠對源資料庫的分割進行數據測試。這個過程可以遞歸式的對樹進行修剪。當無法再進行分割或一個單獨的類可以被應用於某一支時，遞歸過程就完成了。另外，隨機森林分類器將許多決策樹結合起來以提升分類的正確率。

決策樹同時也可以依靠計算條件機率來構造。

決策樹如果依靠數學的計算方法可以取得更加理想的效果。資料庫已如下所示：

$$(x, y) = (x_1, x_2, x_3, \dots, x_k, y)$$

相關的變量 y 表示我們嘗試去理解，分類或者更一般化的結果。其他的變量 x_1, x_2, x_3 等則是幫助我們達到目的的變量。

類型

決策樹有幾種產生方法：

分類樹解析是當預計結果可能為離散類型（例如三個種類的花，輸贏等）使用的概念。

回歸樹解析是當局域結果可能為實數（例如房價，患者住院時間等）使用的概念。

CART 解析是結合了上述二者的一個概念。CART 是 Classification And Regression Trees 的縮寫。

en:CHAID（Chi-Square Automatic Interaction Detector）

建立方法

以資料母群體為根節點。

作單因子變異數解析等，找出變異量最大的變項作為分割準則。（決策樹每個葉節點即為一連串法則的分類結果。）

若判斷結果的正確率或涵蓋率未滿足條件，則再依最大變異量條件長出分岔。

在教學中的使用

決策樹，影響性圖表，應用函數以及其他的決策解析工具和方法主要的授課對象是學校裡商業、健康經濟學和公用衛生專業的本科生，屬於運籌學和管理科學的範疇。

舉例闡述

下面我們用例子來說明：

小王是一家著名高爾夫俱樂部的經理。但是他被雇員數量問題搞得心情十分不好。某些天好像所有人都來玩高爾夫，以至於所有員工都忙的團團轉還是應付不過來，而有些天不知道什麼原因卻一個人也不來，俱樂部為雇員數量浪費了不少資金。

小王的目的是通過下周天氣預報尋找什麼時候人們會打高爾夫，以適時調整雇員數量。因此首先他必須了解人們決定是否打球的原因。

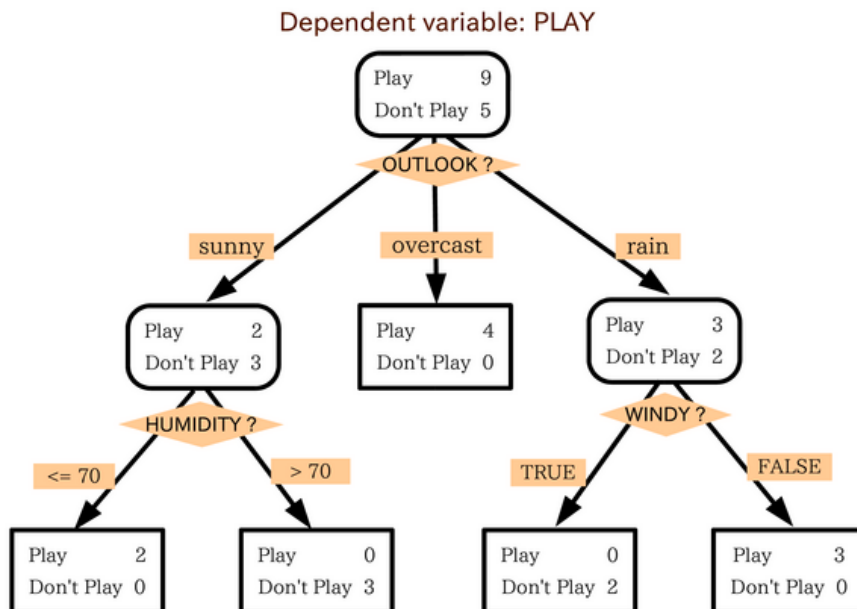
在 2 周時間內我們得到以下記錄：

天氣狀況有晴，雲和雨；氣溫用華氏溫度表示；相對濕度用百分比；還有有無風。當然還有顧客是不是在這些日子光顧俱樂部。最終他得到了 14 行 5 列的數據表格。

Play golf dataset

Independent variables				Dep. var
OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY
sunny	85	85	FALSE	Don't Play
sunny	80	90	TRUE	Don't Play
overcast	83	78	FALSE	Play
rain	70	96	FALSE	Play
rain	68	80	FALSE	Play
rain	65	70	TRUE	Don't Play
overcast	64	65	TRUE	Play
sunny	72	95	FALSE	Don't Play
sunny	69	70	FALSE	Play
rain	75	80	FALSE	Play
sunny	75	70	TRUE	Play
overcast	72	90	TRUE	Play
overcast	81	75	FALSE	Play
rain	71	80	TRUE	Don't Play

決策樹模型就被建起來用於解決問題。



決策樹是一個有向無環圖。根結點代表所有數據。分類樹算法可以通過變量 **outlook**，找出最好地解釋非獨立變量 **play**（打高爾夫的人）的方法。變量 **outlook** 的範疇被劃分為以下三個組：

晴天，多雲天和雨天。

我們得出第一個結論：如果天氣是多雲，人們總是選擇玩高爾夫，而只有

少數很著迷的甚至在雨天也會玩。

接下來我們把晴天組的分為兩部分，我們發現顧客不喜歡濕度高於 70% 的天氣。最終我們還發現，如果雨天還有風的話，就不會有人打了。

這就通過分類樹給出了一個解決方案。小王（老闆）在晴天，潮濕的天氣或者颶風的雨天解僱了大部分員工，因為這種天氣不會有人打高爾夫。而其他的天氣會有很多人打高爾夫，因此可以雇用一些臨時員工來工作。

公式 - 熵 (Entropy)

Entropy = 系統的凌亂程度，使用演算法 ID3, C4.5 和 C5.0 生成樹算法使用熵。這一度量是基於資訊學理論中熵的概念。

決策樹的優點

相對於其他數據挖掘算法，決策樹在以下幾個方面擁有優勢：

決策樹易於理解和實現。人們在通過解釋後都有能力去理解決策樹所表達的意義。

對於決策樹，數據的準備往往是簡單或者是不必要的。其他的技術往往要求先把數據一般化，比如去掉多餘的或者空白的屬性。

能夠同時處理數據型和常規型屬性。其他的技術往往要求數據屬性的單一。

是一個白盒模型如果給定一個觀察的模型，那麼根據所產生的決策樹很容易推出相應的邏輯表達式。

易於通過靜態測試來對模型進行評測。表示有可能測量該模型的可信度。

在相對短的時間內能夠對大型數據源做出可行且效果良好的結果。

22.2 如何在 Python 中實現決策樹算法

決策樹是一種強大的預測方法，非常受歡迎。

它們是受歡迎的，因為最終的模型很容易被實踐者和領域專家所理解。最終的決策樹可以準確地解釋為什麼要做出具體的預測，使其對於作業使用非常有吸引力。

決策樹也為更先進的集合方法提供了基礎，如裝袋，隨機森林和梯度提升。

在本教程中，您將發現如何使用 Python 從頭開始實現分類和回歸樹算法。

完成本教程後，您將知道：

如何計算和評估數據中的候選分割點。
如何將分裂排列成決策樹結構。
如何將分類和回歸樹算法應用到一個真正的問題。

說明

本節簡要介紹本教程中使用的分類和回歸樹算法和 Banknote 數據集。

分類和回歸樹

分類和回歸樹或簡稱 CART 是 Leo Breiman 參照的縮寫，可用於分類或回歸預測建模問題的決策樹算法。

本教程將重點介紹如何使用 CART 進行分類。

CART 模型的表示是二叉樹。這是與算法和數據結構相同的二叉樹，沒有什麼太奇特（每個節點可以有零個，一個或兩個子節點）。

節點表示單個輸入變量（ x ）和該變量上的分割點，假設變量為數字。樹的葉節點（也稱為終端節點）包含用於進行預測的輸出變量（ y ）。

一旦創建，可以使用分割後的每個分支之後的新行數據導航樹，直到進行最終預測。

創建二進制決策樹實際上是分割輸入空間的過程。使用貪心的方法來劃分稱為遞歸二進制分割的空間。這是一個數值程式，其中所有的值都排成一列，並且使用成本函數嘗試和測試不同的分割點。

選擇最佳成本（最低成本，因為我們最小化成本）的分割。所有輸入變量和所有可能的分割點根據成本函數以貪心的方式進行評估和選擇。

回歸：最小化以選擇分割點的成本函數是所有在矩形內的訓練樣本的平方誤差。

分類：使用 Gini 成本函數，提供節點純淨度的指示，節點純度指的是配置給每個節點的訓練數據的混合度。

分割繼續，直到節點包含最少數量的訓練示例或達到最大樹深度。

鈔票數據集

紙幣數據集包括根據從照片採取的一些措施來預測給定的紙幣是否是真實

的。

數據集包含 1,372 個 5 個數字變量。它是兩類（二進制分類）的分類問題。

下面提供了數據集中五個變量的清單。

1. 小波變換圖像（連續）的方差。
2. 小波變換圖像的偏度（連續）。
3. 小波變換圖像（連續）的峰度。
4. 圖像（連續）。
5. 類別（整數）。

以下是數據集前 5 行的示例

```
3.6216,8.6661,-2.8073,-0.44699,0
4.5459,8.1674,-2.4586,-1.4621,0
3.866,-2.6383,1.9242,0.10645,0
3.4566,9.5228,-4.0112,-3.5944,0
0.32924,-4.4552,4.5718,-0.9888,0
4.3684,9.6718,-3.9606,-3.1625,0
```

使用零規則算法來預測最常見的類值，問題的基準精度約為 50%。

您可以從 [UCI Machine Learning Repository](#) 了解更多信息並下載數據集。

下載數據集並將其放在您當前的工作目錄中，文件名為 `data_banknote_authentication.csv`。

教程

本教程分為 5 部分：

1. 基尼指數。
2. 創建分割。
3. 建樹。
4. 做一個預測
5. 鈔票案例研究。

這些步驟將為您提供從頭開始實施 CART 算法的基礎，並將其應用於您自己的預測建模問題。

22.2.1 1. 基尼指數

基尼係數是用於評估數據集中分割的成本函數的名稱。

數據集中的拆分涉及一個輸入屬性和該屬性的一個值。它可以用於將訓練模式分成兩組行。

一個 Gini 評分給出了一個分裂是多麼好的想法，通過分組創建的兩個組中的類是如何混合的。完美的分離導致基尼評分為 0，而最差的分組，導致每組 50/50 級別的基尼評分為 1.0（對於 2 類問題）。

最佳的例子就是計算基尼係數。

我們有兩組數據，每組有 2 行。第一組中的行都屬於類 0，第二組中的行屬於類 1，因此它是完美的拆分。

我們首先需要計算每組中的班級比例。

```
proportion = count(class_value) / count(rows)
```

這個例子的比例是：

```
group_1_class_0 = 2/2 = 1  
group_1_class_1 = 0/2 = 0  
group_2_class_0 = 0/2 = 0  
group_2_class_1 = 2/2 = 1
```

然後計算 Gini 如下：

```
gini_index = sum(proportion * (1.0 - proportion))
```

在每個組和每個類別值計算的所有比例。在我們的例子中，這個計算公式如下：


```
gini_index = (group_1_class_0 * (1.0 - group_1_class_0)) +
              (group_1_class_1 * (1.0 - group_1_class_1)) +
              (group_2_class_0 * (1.0 - group_2_class_0)) +
              (group_2_class_1 * (1.0 - group_2_class_1))
```

或

```
gini_index = 0 + 0 + 0 + 0 = 0
```

下面是一個名為 `gini_index()` 的函數，它計算組列表的 Gini 索引和已知類值的列表。

您可以看到有一些安全檢查，以避免為空組 `empty group` 除以零。

```
# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
    gini = 0.0
    for class_value in class_values:
        for group in groups:
            size = len(group)
            if size == 0:
                continue
            proportion = [row[-1] for row in group].count(class_value) / float(size)
            gini += (proportion * (1.0 - proportion))
    return gini
```

我們可以用我們上面的例子來測試這個函數。我們也可以測試每組 50/50 分的最壞情況。完整的示例如下所示。

範例程式： `DecisionTree-1.py`

```
1. # Calculate the Gini index for a split dataset
2. def gini_index(groups, class_values):
3.     gini = 0.0
4.     for class_value in class_values:
5.         for group in groups:
6.             size = len(group)
7.             if size == 0:
8.                 continue
9.             proportion = [row[-1] for row in group].count(class_value) / float(size)
10.            gini += (proportion * (1.0 - proportion))
```

```

11. return gini
12.
13. # test Gini values
14. print(gini_index([[[1, 1], [1, 0]], [[1, 1], [1, 0]]], [0, 1]))
15. print(gini_index([[[1, 0], [1, 0]], [[1, 1], [1, 1]]], [0, 1]))

```

運行示例打印兩個基尼 Gini 分數，首先是最壞情況下的得分為 1.0，最佳情況下的得分為 0.0。

1.0

0.0

現在我們知道如何評估分裂的結果，我們來看看創建分割。

```

powens-MacBook-Air:Desktop powenko$ python DecisionTree-1.py
1.0
0.0

```

圖 3

22.2.2 2.創建分割

拆分由數據集中的屬性和值組成。

我們可以將其總結為要拆分的屬性的索引以及在該屬性上拆分的值。這只是索引到數據行的有用的簡寫。

創建一個拆分涉及三個部分，第一個我們已經看過哪個是計算基尼分數。剩下的兩個部分是：

1. 拆分數據集。
2. 評估所有分割。

我們來看看每一個。

拆分數據集

拆分數據集意味著將屬性的索引和該屬性的拆分值分隔為兩列列。

一旦我們有兩個組，我們可以使用我們上面的基尼分數來評估分組的成本。

分割數據集涉及對每一行進行迭代，檢查屬性值是否低於或高於分割值，並將其分配給左組或右組。

以下是實現此過程的名為 `test_split ()` 的函數。

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right
```

沒有多少

請注意，正確的組包含所有行的索引高於或等於拆分值的值。

評估所有分割

使用上面的 Gini 功能和測試分割功能，我們現在有了我們需要評估分割的一切。

給定一個數據集，我們必須檢查每個屬性上的每個值作為候選分割，評估分割的成本，並找到我們可以做出最好的分割。

一旦找到最好的分裂，我們就可以將其用作決策樹中的一個節點。

這是一個窮舉和貪心的算法。

我們將使用字典來表示決策樹中的一個節點，因為我們可以按名稱存儲數據。當選擇最佳分割並將其用作樹的新節點時，我們將存儲所選屬性的索引，要分割的屬性的值以及所選分割點分割的兩組數據。

每組數據都是其分配過程分配給左組或右組的那些行的自己的小數據集。您可以想像，當我們構建我們的決策樹時，我們可以再次分解每個組。

以下是實現此過程的名為 `get_split()` 的函數。您可以看到它遍歷每個屬性（除了類值除外），然後遍歷該屬性的每個值，分割和評估拆分。

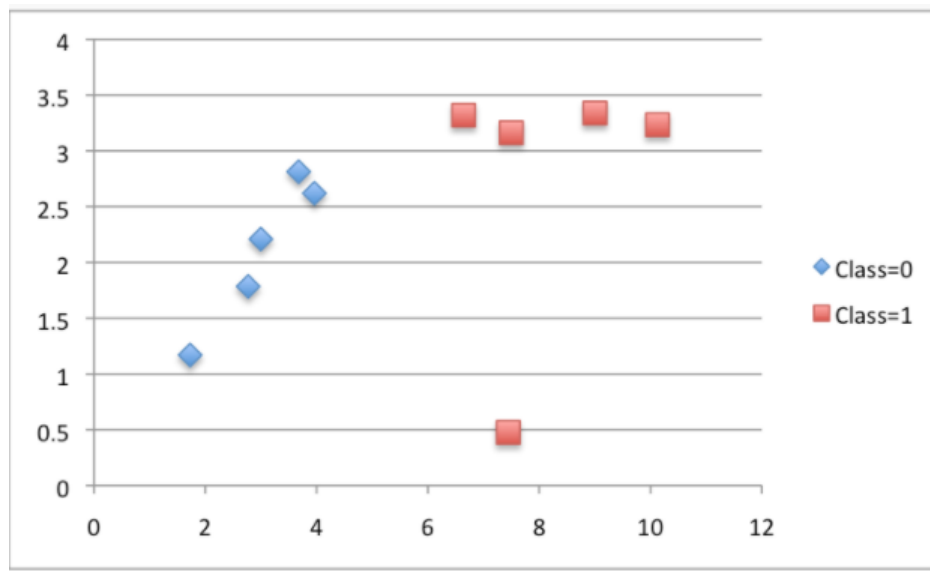
記錄最好的分割，然後在所有檢查完成後返回。

```
# Select the best split point for a dataset
def get_split(dataset):
    class_values = list(set(row[-1] for row in dataset))
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    for index in range(len(dataset[0])-1):
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini, groups
    return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

我們可以設計一個小數據集來測試這個功能和我們整個數據集的分割過程。

X1	X2	Y
2.771244718	1.784783929	0
1.728571309	1.169761413	0
3.678319846	2.81281357	0
3.961043357	2.61995032	0
2.999208922	2.209014212	0
7.497545867	3.162953546	1
9.00220326	3.339047188	1
7.444542326	0.476683375	1
10.12493903	3.234550982	1
6.642287351	3.319983761	1

我們可以為每個類使用單獨的顏色繪製該數據集。您可以看到，手動選擇 `x1`（圖中的 `x` 軸）來分割此數據集並不難。



他下面的例子將所有這一切放在一起。

範例程式： DecisionTree-2.py

```

1. # Split a dataset based on an attribute and an attribute value
2. def test_split(index, value, dataset):
3.     left, right = list(), list()
4.     for row in dataset:
5.         if row[index] < value:
6.             left.append(row)
7.         else:
8.             right.append(row)
9.     return left, right
10.
11. # Calculate the Gini index for a split dataset
12. def gini_index(groups, class_values):
13.     gini = 0.0
14.     for class_value in class_values:
15.         for group in groups:
16.             size = len(group)
17.             if size == 0:
18.                 continue
19.             proportion = [row[-1] for row in group].count(class_value) / float(size)
20.             gini += (proportion * (1.0 - proportion))
21.     return gini
22.
23. # Select the best split point for a dataset
24. def get_split(dataset):
25.     class_values = list(set(row[-1] for row in dataset))
26.     b_index, b_value, b_score, b_groups = 999, 999, 999, None

```

```

27. for index in range(len(dataset[0])-1):
28.     for row in dataset:
29.         groups = test_split(index, row[index], dataset)
30.         gini = gini_index(groups, class_values)
31.         print('X%d < %.3f Gini=%.3f' % ((index+1), row[index], gini))
32.         if gini < b_score:
33.             b_index, b_value, b_score, b_groups = index, row[index], gini, groups
34.     return {'index':b_index, 'value':b_value, 'groups':b_groups}
35.
36. dataset = [[2.771244718,1.784783929,0],
37. [1.728571309,1.169761413,0],
38. [3.678319846,2.81281357,0],
39. [3.961043357,2.61995032,0],
40. [2.999208922,2.209014212,0],
41. [7.497545867,3.162953546,1],
42. [9.00220326,3.339047188,1],
43. [7.444542326,0.476683375,1],
44. [10.12493903,3.234550982,1],
45. [6.642287351,3.319983761,1]]
46. split = get_split(dataset)
47. print('Split: [X%d < %.3f]' % ((split['index']+1), split['value']))

```

get_split () 函數被修改為打印出每個分割點，並且它是 Gini 索引，因為它被評估。

運行示例打印所有基尼分數，然後在 $X_1 < 6.642$ 的數據集中打印最佳分割的分數，基尼係數為 0.0 或完美分割。

```

X1 < 2.771 Gini=0.494
X1 < 1.729 Gini=0.500
X1 < 3.678 Gini=0.408
X1 < 3.961 Gini=0.278
X1 < 2.999 Gini=0.469
X1 < 7.498 Gini=0.408
X1 < 9.002 Gini=0.469
X1 < 7.445 Gini=0.278
X1 < 10.125 Gini=0.494
X1 < 6.642 Gini=0.000
X2 < 1.785 Gini=1.000
X2 < 1.170 Gini=0.494
X2 < 2.813 Gini=0.640
X2 < 2.620 Gini=0.819
X2 < 2.209 Gini=0.934

```

Python

```
X2 < 3.163 Gini=0.278
X2 < 3.339 Gini=0.494
X2 < 0.477 Gini=0.500
X2 < 3.235 Gini=0.408
X2 < 3.320 Gini=0.469
Split: [X1 < 6.642]
```

現在我們知道如何在數據集或行列表中找到最佳分割點，我們來看看如何使用它來構建一個決策樹。

```
[powens-MacBook-Air:Desktop powenko$ python DecisionTree-2.py
X1 < 2.771 Gini=0.494
X1 < 1.729 Gini=0.500
X1 < 3.678 Gini=0.408
X1 < 3.961 Gini=0.278
X1 < 2.999 Gini=0.469
X1 < 7.498 Gini=0.408
X1 < 9.002 Gini=0.469
X1 < 7.445 Gini=0.278
X1 < 10.125 Gini=0.494
X1 < 6.642 Gini=0.000
X2 < 1.785 Gini=1.000
X2 < 1.170 Gini=0.494
X2 < 2.813 Gini=0.640
X2 < 2.620 Gini=0.819
X2 < 2.209 Gini=0.934
X2 < 3.163 Gini=0.278
X2 < 3.339 Gini=0.494
X2 < 0.477 Gini=0.500
X2 < 3.235 Gini=0.408
X2 < 3.320 Gini=0.469
Split: [X1 < 6.642]
```

22.2.3 建樹

創建樹的根節點很容易。

我們使用整個數據集 `entire dataset` 使用上述 `get_split ()` 函數。

在我們的樹上添加更多的節點更有趣。

建樹可分為 3 個主要部分：

1. 終端節點。
2. 遞歸拆分
3. 建樹

終端節點

我們需要決定什麼時候停止種樹。

我們可以使用訓練數據集中節點負責的深度和行數來做到這一點。

- 最大樹深。

這是樹的根節點的最大節點數。一旦滿足樹的最大深度，我們必須停止分割添加新節點。更深的樹木更複雜，更有可能超過訓練數據。

- 最小節點記錄。

這是給定節點負責的訓練模式的最小數量。一旦達到或低於此最小值，我們必須停止拆分和添加新節點。考慮到訓練模式太少的節點預計太具體，可能會超載訓練數據。

這兩種方法將是我們樹構建過程的用戶指定的參數。

還有一個條件。可以選擇所有行屬於一個組的拆分。在這種情況下，我們將無法繼續拆分和添加子節點，因為我們將沒有記錄在一邊或另一側分割。

現在我們有一些想法，何時停止生長樹。當我們在給定的點停止增長時，該節點稱為終端節點，並用於進行最終預測。

這是通過將分配給該節點的行組選為組中最常見的類值來完成的。這將用於做出預測。

下面是一個名為 `to_terminal()` 的函數，它將為一組行選擇一個類值。它返回行列表中最常見的輸出值。

```
1. # Create a terminal node value
2. def to_terminal(group):
3.     outcomes = [row[-1] for row in group]
4.     return max(set(outcomes), key=outcomes.count)
```

遞歸拆分

我們知道如何和何時創建終端節點，現在我們可以構建我們的樹。

構建決策樹包括在為每個節點創建的組上重複調用上述開發的 `get_split()` 函數。

添加到現有節點的新節點稱為子節點。節點可以具有零子節點（終端節點），

一個子節點（一側直接進行預測）或兩個子節點。我們將在給定節點的字典表示中向左和向右引用子節點。

一旦創建了一個節點，我們可以通過再次調用相同的函數來從分裂的每組數據上遞歸地創建子節點。

以下是實現此遞歸過程的函數。它需要一個節點作為參數以及節點中的最大深度，最小模式數和節點的當前深度。

您可以想像這可能是如何首先稱為傳遞根節點和深度 1。此功能最好在步驟中解釋：

1. 首先，提取節點分割的兩組數據，以便從節點中刪除。當我們處理這些組時，節點不再需要訪問這些數據。
2. 接下來，我們檢查左或右組行是否為空，如果是，我們使用我們擁有的記錄創建一個終端節點。
3. 然後我們檢查一下我們是否達到了最大的深度，如果是，我們創建一個終端節點。
4. 然後，我們處理左邊的孩子，如果一組行太小，創建終端節點，否則以深度第一種方式創建和添加左側節點，直到在該分支上到達樹的底部。
5. 然後以相同的方式處理右側，因為我們將構建的樹恢復到根。

```

1. # Create child splits for a node or make terminal
2. def split(node, max_depth, min_size, depth):
3.     left, right = node['groups']
4.     del(node['groups'])
5.     # check for a no split
6.     if not left or not right:
7.         node['left'] = node['right'] = to_terminal(left + right)
8.         return
9.     # check for max depth
10.    if depth >= max_depth:
11.        node['left'], node['right'] = to_terminal(left), to_terminal(right)
12.        return
13.    # process left child
14.    if len(left) <= min_size:
15.        node['left'] = to_terminal(left)
16.    else:
17.        node['left'] = get_split(left)
18.        split(node['left'], max_depth, min_size, depth+1)
19.    # process right child
20.    if len(right) <= min_size:

```

```

21.     node['right'] = to_terminal(right)
22. else:
23.     node['right'] = get_split(right)
24.     split(node['right'], max_depth, min_size, depth+1)

```

建樹

我們現在可以將所有的部分放在一起。

構建樹包括創建根節點並調用 `split()` 函數，然後調用自身遞歸地構建整個樹。

以下是實現此過程的小型 `build_tree()` 函數。

```

1. # Build a decision tree
2. def build_tree(train, max_depth, min_size):
3.     root = get_split(train)
4.     split(root, max_depth, min_size, 1)
5.     return root

```

我們可以使用我們上面提到的小數據集來測試這個整個過程。

下面是完整的例子。

還包括一個小的 `print_tree()` 函數，它遞歸地打印出每個節點一行決策樹的節點。雖然不像真正的決策樹圖那麼醒目，但它給出了整個樹結構和決策的想法。

範例程式 DecisionTree-3.py

```

1. # Split a dataset based on an attribute and an attribute value
2. def test_split(index, value, dataset):
3.     left, right = list(), list()
4.     for row in dataset:
5.         if row[index] < value:
6.             left.append(row)
7.         else:
8.             right.append(row)
9.     return left, right
10.
11. # Calculate the Gini index for a split dataset
12. def gini_index(groups, class_values):
13.     gini = 0.0
14.     for class_value in class_values:

```

```

15.     for group in groups:
16.         size = len(group)
17.         if size == 0:
18.             continue
19.         proportion = [row[-1] for row in group].count(class_value) / float(size)
20.         gini += (proportion * (1.0 - proportion))
21.     return gini
22.
23. # Select the best split point for a dataset
24. def get_split(dataset):
25.     class_values = list(set(row[-1] for row in dataset))
26.     b_index, b_value, b_score, b_groups = 999, 999, 999, None
27.     for index in range(len(dataset[0])-1):
28.         for row in dataset:
29.             groups = test_split(index, row[index], dataset)
30.             gini = gini_index(groups, class_values)
31.             if gini < b_score:
32.                 b_index, b_value, b_score, b_groups = index, row[index], gini, groups
33.     return {'index':b_index, 'value':b_value, 'groups':b_groups}
34.
35. # Create a terminal node value
36. def to_terminal(group):
37.     outcomes = [row[-1] for row in group]
38.     return max(set(outcomes), key=outcomes.count)
39.
40. # Create child splits for a node or make terminal
41. def split(node, max_depth, min_size, depth):
42.     left, right = node['groups']
43.     del(node['groups'])
44.     # check for a no split
45.     if not left or not right:
46.         node['left'] = node['right'] = to_terminal(left + right)
47.         return
48.     # check for max depth
49.     if depth >= max_depth:
50.         node['left'], node['right'] = to_terminal(left), to_terminal(right)
51.         return
52.     # process left child
53.     if len(left) <= min_size:
54.         node['left'] = to_terminal(left)
55.     else:
56.         node['left'] = get_split(left)
57.         split(node['left'], max_depth, min_size, depth+1)
58.     # process right child

```

```

59. if len(right) <= min_size:
60.     node['right'] = to_terminal(right)
61. else:
62.     node['right'] = get_split(right)
63.     split(node['right'], max_depth, min_size, depth+1)
64.
65. # Build a decision tree
66. def build_tree(train, max_depth, min_size):
67.     root = get_split(train)
68.     split(root, max_depth, min_size, 1)
69.     return root
70.
71. # Print a decision tree
72. def print_tree(node, depth=0):
73.     if isinstance(node, dict):
74.         print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
75.         print_tree(node['left'], depth+1)
76.         print_tree(node['right'], depth+1)
77.     else:
78.         print('%s[%s]' % ((depth*' ', node)))
79.
80. dataset = [[2.771244718,1.784783929,0],
81. [1.728571309,1.169761413,0],
82. [3.678319846,2.81281357,0],
83. [3.961043357,2.61995032,0],
84. [2.999208922,2.209014212,0],
85. [7.497545867,3.162953546,1],
86. [9.00220326,3.339047188,1],
87. [7.444542326,0.476683375,1],
88. [10.12493903,3.234550982,1],
89. [6.642287351,3.319983761,1]]
90. tree = build_tree(dataset, 1, 1)
91. print_tree(tree)

```

我們可以在運行此示例時更改最大深度參數，並查看打印樹上的效果。

最大深度為 1（調用 `build_tree()` 函數時的第二個參數），我們可以看到，該樹使用了我們在上一節中發現的完美拆分。這是一個有一個節點的樹，也稱為一個樹樁。

```

[powens-MacBook-Air:Desktop powenko$ python DecisionTree-3.py
[X1 < 6.642]
[0]
[1]

```

圖 4

將最大深度增加到 2，即使不需要，我們強制樹分裂。然後， x_1 屬性再次被根節點的左和右子節點使用，以分解已經完美的類組合。

```
[X1 < 6.642]
[X1 < 2.771]
[0]
[0]
[X1 < 7.498]
[1]
[1]
```

最後，反過來，我們可以強制更多的分級，最大深度為 3。

```
[X1 < 6.642]
[X1 < 2.771]
[0]
[X1 < 2.771]
[0]
[0]
[X1 < 7.498]
[X1 < 7.445]
[1]
[1]
[X1 < 7.498]
[1]
[1]
```

這些測試表明，有很好的機會來完善實施，以避免不必要的分裂。這作為一個擴展。

現在我們可以創建一個決策樹，讓我們看看我們如何使用它來對新數據做出預測。

22.2.4 做一個預測

使用決策樹進行預測涉及使用專門提供的數據行導航樹。

再次，我們可以使用遞歸函數實現這一點，其中根據分割如何影響提供的數據，再次使用左子節點或右子節點調用相同的預測例程。

我們必須檢查子節點是要作為預測返回的終端值，還是檢查是否包含要考慮的樹的另一個級別的字典節點。

以下是實現此過程的 `predict ()` 函數。您可以看到給定節點中的索引和值如何

您可以看到給定節點中的索引和值如何用於評估提供的數據行是否落在拆分的左側或右側。

```

1. # Make a prediction with a decision tree
2. def predict(node, row):
3.     if row[node['index']] < node['value']:
4.         if isinstance(node['left'], dict):
5.             return predict(node['left'], row)
6.         else:
7.             return node['left']
8.     else:
9.         if isinstance(node['right'], dict):
10.            return predict(node['right'], row)
11.        else:
12.            return node['right']

```

我們可以使用我們設計的數據集來測試此功能。下面是一個使用硬編碼決策樹與一個最好分割數據（一個決定樹樁）的單個節點的例子。

該示例對數據集中的每一行進行預測。

範例程式：DecisionTree-4.py

```

1. # Make a prediction with a decision tree
2. def predict(node, row):
3.     if row[node['index']] < node['value']:
4.         if isinstance(node['left'], dict):
5.             return predict(node['left'], row)

```

```

6.         else:
7.             return node['left']
8.     else:
9.         if isinstance(node['right'], dict):
10.            return predict(node['right'], row)
11.        else:
12.            return node['right']
13.
14. dataset = [[2.771244718,1.784783929,0],
15. [1.728571309,1.169761413,0],
16. [3.678319846,2.81281357,0],
17. [3.961043357,2.61995032,0],
18. [2.999208922,2.209014212,0],
19. [7.497545867,3.162953546,1],
20. [9.00220326,3.339047188,1],
21. [7.444542326,0.476683375,1],
22. [10.12493903,3.234550982,1],
23. [6.642287351,3.319983761,1]]
24.
25. # predict with a stump
26. stump = {'index': 0, 'right': 1, 'value': 6.642287351, 'left': 0}
27. for row in dataset:
28.     prediction = predict(stump, row)
29.     print('Expected=%d, Got=%d' % (row[-1], prediction))

```

運行示例按預期列印每行的正確預測。

```

[powens-MacBook-Air:Desktop powenko$ python DecisionTree-4.py
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1

```

圖 5

我們現在知道如何創建決策樹並使用它來進行預測。現在，我們將它應用到一個真正的數據集。

22.2.5 鈔票案例研究

本節將 CART 算法應用於 Bank Note 數據集。

第一步是加載數據集並將加載的數據轉換為可用於計算分割點的數字。為此，我們將使用幫助函數 `load_csv ()` 加載文件和 `str_column_to_float ()` 將字符串數字轉換為浮點數。

我們將使用 5 倍的 k 倍交叉驗證來評估算法。這意味著每個折疊中將使用 $1372/5 = 274.4$ 或僅 270 個記錄。我們將使用輔助函數 `evaluate_algorithm ()` 來評估具有交叉驗證和 `precision_metric ()` 的算法來計算預測的準確性。

開發了一個名為 `decision_tree ()` 的新函數來管理 CART 算法的應用，首先從訓練數據集創建樹，然後使用樹對測試數據集進行預測。

完整的示例如下所示。

範例程式：`DecisionTree-5.py`

```
1. # CART on the Bank Note dataset
2. from random import seed
3. from random import randrange
4. from csv import reader
5.
6. # Load a CSV file
7. def load_csv(filename):
8.     file = open(filename, "rb")
9.     lines = reader(file)
10.    dataset = list(lines)
11.    return dataset
12.
13. # Convert string column to float
14. def str_column_to_float(dataset, column):
15.     for row in dataset:
16.         row[column] = float(row[column].strip())
17.
18. # Split a dataset into k folds
```



```

19. def cross_validation_split(dataset, n_folds):
20.     dataset_split = list()
21.     dataset_copy = list(dataset)
22.     fold_size = int(len(dataset) / n_folds)
23.     for i in range(n_folds):
24.         fold = list()
25.         while len(fold) < fold_size:
26.             index = randrange(len(dataset_copy))
27.             fold.append(dataset_copy.pop(index))
28.         dataset_split.append(fold)
29.     return dataset_split
30.
31. # Calculate accuracy percentage
32. def accuracy_metric(actual, predicted):
33.     correct = 0
34.     for i in range(len(actual)):
35.         if actual[i] == predicted[i]:
36.             correct += 1
37.     return correct / float(len(actual)) * 100.0
38.
39. # Evaluate an algorithm using a cross validation split
40. def evaluate_algorithm(dataset, algorithm, n_folds, *args):
41.     folds = cross_validation_split(dataset, n_folds)
42.     scores = list()
43.     for fold in folds:
44.         train_set = list(folds)
45.         train_set.remove(fold)
46.         train_set = sum(train_set, [])
47.         test_set = list()
48.         for row in fold:
49.             row_copy = list(row)
50.             test_set.append(row_copy)
51.             row_copy[-1] = None
52.         predicted = algorithm(train_set, test_set, *args)
53.         actual = [row[-1] for row in fold]
54.         accuracy = accuracy_metric(actual, predicted)
55.         scores.append(accuracy)
56.     return scores
57.
58. # Split a dataset based on an attribute and an attribute value
59. def test_split(index, value, dataset):
60.     left, right = list(), list()
61.     for row in dataset:
62.         if row[index] < value:

```

```

63.         left.append(row)
64.     else:
65.         right.append(row)
66.     return left, right
67.
68. # Calculate the Gini index for a split dataset
69. def gini_index(groups, class_values):
70.     gini = 0.0
71.     for class_value in class_values:
72.         for group in groups:
73.             size = len(group)
74.             if size == 0:
75.                 continue
76.             proportion = [row[-1] for row in group].count(class_value) / float(size)
77.             gini += (proportion * (1.0 - proportion))
78.     return gini
79.
80. # Select the best split point for a dataset
81. def get_split(dataset):
82.     class_values = list(set(row[-1] for row in dataset))
83.     b_index, b_value, b_score, b_groups = 999, 999, 999, None
84.     for index in range(len(dataset[0])-1):
85.         for row in dataset:
86.             groups = test_split(index, row[index], dataset)
87.             gini = gini_index(groups, class_values)
88.             if gini < b_score:
89.                 b_index, b_value, b_score, b_groups = index, row[index], gini, groups
90.     return {'index':b_index, 'value':b_value, 'groups':b_groups}
91.
92. # Create a terminal node value
93. def to_terminal(group):
94.     outcomes = [row[-1] for row in group]
95.     return max(set(outcomes), key=outcomes.count)
96.
97. # Create child splits for a node or make terminal
98. def split(node, max_depth, min_size, depth):
99.     left, right = node['groups']
100.    del(node['groups'])
101.    # check for a no split
102.    if not left or not right:
103.        node['left'] = node['right'] = to_terminal(left + right)
104.        return
105.    # check for max depth
106.    if depth >= max_depth:

```

```

107.         node['left'], node['right'] = to_terminal(left), to_terminal(right)
108.         return
109.     # process left child
110.     if len(left) <= min_size:
111.         node['left'] = to_terminal(left)
112.     else:
113.         node['left'] = get_split(left)
114.         split(node['left'], max_depth, min_size, depth+1)
115.     # process right child
116.     if len(right) <= min_size:
117.         node['right'] = to_terminal(right)
118.     else:
119.         node['right'] = get_split(right)
120.         split(node['right'], max_depth, min_size, depth+1)
121.
122. # Build a decision tree
123. def build_tree(train, max_depth, min_size):
124.     root = get_split(train)
125.     split(root, max_depth, min_size, 1)
126.     return root
127.
128. # Make a prediction with a decision tree
129. def predict(node, row):
130.     if row[node['index']] < node['value']:
131.         if isinstance(node['left'], dict):
132.             return predict(node['left'], row)
133.         else:
134.             return node['left']
135.     else:
136.         if isinstance(node['right'], dict):
137.             return predict(node['right'], row)
138.         else:
139.             return node['right']
140.
141. # Classification and Regression Tree Algorithm
142. def decision_tree(train, test, max_depth, min_size):
143.     tree = build_tree(train, max_depth, min_size)
144.     predictions = list()
145.     for row in test:
146.         prediction = predict(tree, row)
147.         predictions.append(prediction)
148.     return(predictions)
149.
150. # Test CART on Bank Note dataset

```

```
151. seed(1)
152. # load and prepare data
153. filename = 'data_banknote_authentication.csv'
154. dataset = load_csv(filename)
155. # convert string attributes to integers
156. for i in range(len(dataset[0])):
157.     str_column_to_float(dataset, i)
158. # evaluate algorithm
159. n_folds = 5
160. max_depth = 5
161. min_size = 10
162. scores = evaluate_algorithm(dataset, decision_tree, n_folds, max_depth, min_size)
163. print('Scores: %s' % scores)
164. print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

該示例使用 5 層的最大樹深度和每個節點的最小行數為 10.通過一些實驗選擇了 CART 的這些參數，但絕不是最佳選擇。

運行示例打印每個折疊的平均分類精度以及所有折疊的平均分類精度。

您可以看到 CART 和所選配置實現了大約 83% 的均值分類精度，這顯著優於實現 50% 精度的零規則算法。

Scores: [83.57664233576642, 82.84671532846716, 86.86131386861314, 79.92700729927007, 82.11678832116789]

Mean Accuracy: 83.066%

第23章 數據分析演算法-naive-bayes

23.1 貝氏二元分類 Naïve-Bayes

資料來源：<http://enginebai.logdown.com/posts/241677/bayes-classification>

單純貝氏 (Naïve-Bayes) 分析是簡單而且實用的分類方法。單純貝氏分類法是以貝氏定理 (Bayes' theorem) 為基礎。貝氏定理來自於 18 世紀數學家湯瑪斯·貝耶斯。單純貝氏分類法希望能透過機率的計算，用以判斷未知類別的資料應該屬於那一個類別。

特徵欄位:天氣、溫度、濕度、風速

標籤欄位:是否可以打網球

透過機率的計算，決定依照天氣條件，分類是否可以打網球。

貝式分類器，擁有幾項特點：

- 基於機率型分類 (probabilistic classification)
- 使用貝氏定理來做計算。
- 假設特徵之間事件獨立 (independence)。

最常用於文件自動分類的應用上。

所建立出來簡單且有效的分類演算法，以下我們一樣使用新聞分類來作為演算法範例，主要目標為將新聞依照內容描述來自動分類為汽車(C)、運動(S)、科技(T)。

23.1.1 演算法思路

貝氏定理

一切都從條件機率開始說起，我們說 B 事件發生的條件下發生 A 事件的機率

公式如下：

$$P(A|B)=P(A \cap B)P(B) \dots (1)$$

現在把「B 事件發生的條件下發生 A 事件的機率」和「A 事件發生的條件下發生 B 事件的機率」公式寫在一起，然後做個公式位移：

$$P(B|A)=P(B \cap A)P(A) \dots (2)$$

把 (1) 和 (2) 公式調整一下

$$P(B|A) \cdot P(A)=P(B \cap A)=P(A \cap B)=P(A|B) \cdot P(B)$$

$$\rightarrow P(B|A) \cdot P(A)=P(A|B) \cdot P(B)$$

最後把 $P(B \cap A)$ 帶入置換我們一開始說的 $P(A|B)$ 公式，就可以導出貝氏定理：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

這個公式也可解讀成下列意思：

$$P(\text{posterior}) = \frac{P(\text{conditinal}) \cdot P(\text{prior})}{P(\text{evidence})}$$

$$P(\text{posterior})=P(\text{conditinal}) \cdot P(\text{prior})P(\text{evidence})$$

prior probability: 先驗機率， $P(A)$ 是在觀察特徵或證據「前」，不考慮任何事件 B 的因素，能表達事件 A 的機率分佈。

posterior probability: 後驗機率， $P(A|B)$ 是在觀察特徵或證據「後」，所得到的條件機率。

貝氏分類

回歸到貝氏分類，我們該如何應用貝氏定理還幫我們執行分類的工作呢？
我們剛剛提的「後驗機率」可以解釋成

給定一些觀察特徵值，我們能計算事物是屬於某個分類的機率。

上述的觀察特徵值我們可以把表示成一個向量，對於新聞分類這問題，我們可以把特徵關鍵字字詞頻率表示成向量：

$$\vec{tf} = (tf_1, tf_2, tf_3, \dots, tf_n)$$

如果你對於「特徵關鍵字字詞頻率」陌生，可以參考我寫的 KNN 分類演算法裡面的「前置作業」篇幅。

我們現在的問題是要把新聞自動分類，那麼「後驗機率」就可以說成

給定一些特徵關鍵字字詞頻率向量，我們能計算這篇新聞是屬於某個分類的機率。

寫成貝氏定理的公式就是

$$P(\text{分類}|\text{特徵關鍵字字詞頻率向量}) = \frac{P(\text{特徵關鍵字字詞頻率向量}|\text{分類}) \cdot P(\text{分類})}{P(\text{特徵關鍵字字詞頻率向量})}$$

換成實際的例子就是

$$P(\text{汽車}|\text{賓士, 寶馬, 籃球, 路跑, 手機, App}) = \frac{P(\text{賓士, 寶馬, 籃球, 路跑, 手機, App}|\text{汽車}) \cdot P(\text{汽車})}{P(\text{賓士, 寶馬, 籃球, 路跑, 手機, App})}$$

中文解釋就是

在出現「賓士, 寶馬, 籃球, 路跑, 手機, App」這些特徵關鍵字的情況下，該篇新聞是屬於「汽車」的機率是多少？

會等於

在「汽車」新聞當中出現「賓士, 寶馬, 籃球, 路跑, 手機, App」字詞的機率 x 「汽車」新聞的機率 / 「賓士, 寶馬, 籃球, 路跑, 手機, App」字詞的機率。
上面字過長，滑鼠往右拉。

23.1.2 訓練階段

貝氏分類器的訓練階段是計算

$P(\text{特徵關鍵字字詞頻率向量} | \text{分類}) \cdot P(\text{分類})$

這個算式的數值就要從訓練集合而來，我們要準備各個分類（汽車、運動、科技）的數篇新聞集合，然後做切字並且比對計算特徵關鍵字字詞頻率向量。

新聞	分類	賓士	寶馬	籃球	路跑	手機	App
C63發表會	P	15	25	0	5	8	3
BMW i8	P	35	40	1	3	3	2
林書豪	S	5	0	35	50	0	0
湖人隊	S	1	5	32	15	0	0
Android 5.0	T	10	5	7	0	2	30
iPhone6	T	5	5	5	15	8	32

→ $P(\text{賓士,寶馬,籃球,路跑,手機,App} | \text{汽車}) \cdot P(\text{汽車})$

→ $P(\text{賓士,寶馬,籃球,路跑,手機,App} | \text{運動}) \cdot P(\text{運動})$

→ $P(\text{賓士,寶馬,籃球,路跑,手機,App} | \text{科技}) \cdot P(\text{科技})$

這邊我已經沒有寫 $P(\text{特徵關鍵字字詞頻率向量})$ ，因為比較不同分類之間的后驗機率時，分母 $P(\text{特徵關鍵字字詞頻率向量})$ 總是常數，因此可以忽略。

23.1.3 獨立事件

實際上，即便有了各分類的新聞集合，我們也很難計算 $P(\text{特徵關鍵字字詞頻率向量} | \text{分類})$ ，也就是很難計算

$P(\text{賓士,寶馬,籃球,路跑,手機,App} | \text{汽車}) \cdot P(\text{汽車})$

所以要引進貝氏分類最重要的「獨立事件」假設，所謂獨立事件就是一個事件 A 的結果不會影響另一個事件 B 發生的機率，舉個例子，給予兩個公正的硬幣，投擲硬幣兩次，那麼第一次投擲的結果不影響第二次投擲的機率。兩個獨立事件發生的機率就會變成兩個事件機率的乘積。

$P(A \cap B) = P(A) \cdot P(B)$

回到我們的 $P(\text{特徵關鍵字字詞頻率向量} | \text{分類})$ ，我們假設每個分類下的各個特徵關鍵字出現的機率彼此獨立，所以公式可以寫成：

$P(\text{賓士,寶馬,籃球,路跑,手機,App} | \text{汽車}) \cdot P(\text{汽車}) =$

$P(\text{賓士} | \text{汽車}) \cdot P(\text{寶馬} | \text{汽車}) \cdot P(\text{籃球} | \text{汽車}) \cdot P(\text{路跑} | \text{汽車}) \cdot P(\text{手機} | \text{汽車}) \cdot P(\text{App} | \text{汽$

車) · P(汽車)

23.1.4 字詞分佈模式

這邊我們有兩個字詞分佈模式，分別為：

Bernouli: 只判斷字詞是否有出現，就出現就是 1，沒有出現就是 0。

$P(\text{分類}) = \text{該分類新聞篇數} / \text{所有訓練集合新聞篇數}$

$P(\text{特徵關鍵字} | \text{分類}) = (\text{該分類下包含特徵關鍵字的新聞篇數} + 1) / (\text{該分類下包含特徵關鍵字的新聞篇數} + 2)$

Multinomial: 直接採用字詞出現頻率。

$P(\text{分類}) = \text{該分類下字詞頻率總和} / \text{所有訓練集合字詞頻率總和}$

$P(\text{特徵關鍵字} | \text{分類}) = (\text{該分類下、該關鍵字字詞頻率總和} + 1) / (\text{該分類下所有關鍵字字詞頻率總和} + \text{訓練集合關鍵字個數})$

以下我們都採用 Multinomial 來計算。

這邊有一個議題可以探討：不同的字詞分佈模式是否會影響我們最後分類的準度呢？

計算步驟

我們開始先訓練分類器，這邊只用「汽車」分類當作例子，其他分類計算方式類似，各個特徵關鍵字的分類機率如下

$P(\text{賓士} | \text{汽車}) = ((15+35)+1) / (140+6) = 0.3493150684931507$

$P(\text{寶馬} | \text{汽車}) = ((25+40)+1) / (140+6) = 0.4520547945205479$

$P(\text{籃球} | \text{汽車}) = ((0+1)+1) / (140+6) = 0.0136986301369863$

$P(\text{路跑} | \text{汽車}) = ((5+3)+1) / (140+6) = 0.06164383561643835$

$P(\text{手機} | \text{汽車}) = ((25+40)+1) / (140+6) = 0.0821917808219178$

$P(\text{App} | \text{汽車}) = ((25+40)+1) / (140+6) = 0.0410958904109589$

$P(\text{汽車}) = 0.343980343980344$

訓練階段完成，這些數值等等會使用到。

現在有一篇新的新聞，其特徵關鍵字字詞頻率

地點	分類	賓士	寶馬	籃球	路跑	手機	App
騎士隊	?	10	2	50	56	8	5

我們要計算該篇新聞屬於「汽車」的機率

$$P(\text{汽車}|\text{特徵關鍵字}) = (0.3493150684931507^{10} \cdot 0.4520547945205479^2 \cdot 0.0136986301369863^{50} \\ \cdot 0.06164383561643835^{56} \cdot 0.0821917808219178^8 \cdot 0.0410958904109589^5) \\ \cdot 0.343980343980344$$

這些乘積出來的結果就是這篇新的新聞屬於「汽車」的機率。

23.1.5 向下溢位

如果把這個公式的數值給電腦算，應該有 99.9999...% 的機率算不出來，為何？因為機率小於 1，越小的數字會越乘越小，這樣乘下去電腦就產生「向下溢位」的問題，這邊我們要修改一下機率的計算公式，我們把公式兩邊都取 log，指數就變成相乘，原本相乘就變成相加，算出來的就是機率的 log 值。

注意，這邊我們重點在於比較各分類的機率大小關係，而非數值本身，所以所有分類機率數值都取 log 一樣可以比較所屬分類。

結論

貝氏分類對於少量的訓練集合同樣會有不錯的分類準度，它的威力恰好在於小樣本下，專家意見或歷史經驗，也就是所謂的先驗分配，能夠補足小樣本的不足，使得推論能夠進行。

適合用在資料會不斷成長的應用。

介紹

這是一個你遇到的情況：

您正在分類問題，您已經生成了一套假設，創建的功能，並討論了變量的重要性。在一個小時內，利益相關者希望看到模型的第一個削減。

你會怎麼做？您的訓練數據集中有數千個數據點和相當多的變量。在這種情況下，如果我在你的地方，我會使用“樸素貝葉斯”，相對於其他分類算法來說，它們可以非常快。它可以用貝葉斯概率定理來預測未知數據集的類別。

在這篇文章中，我將解釋這個算法的基礎知識，所以下次當你遇到大數據集時，可以使這個算法動作。另外，如果您是 Python 中的新手，則應該在本文存在可用代碼的情況下被淹沒。

目錄

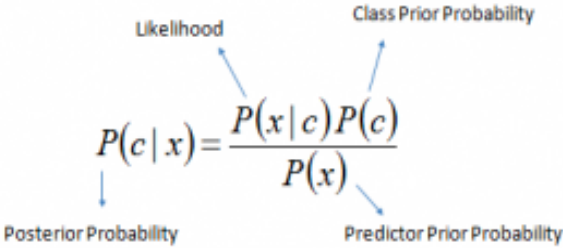
什麼是樸素貝葉斯算法？
 樸素貝葉斯算法如何運作？
 使用樸素貝葉斯的優點和缺點是什麼？
 樸素貝葉斯算法的應用
 在 Python 中構建基本的樸素貝葉斯模型的步驟
 提高樸素貝葉斯模型的力量提示

什麼是樸素貝葉斯算法？

它是基於貝葉斯定理的分類技術，假設預測因子之間是獨立的。簡單來說，樸素貝葉斯分類器假設類中特定特徵的存在與任何其他特徵的存在無關。例如，如果果實紅色，圓形，直徑約 3 英寸，則果實可能被認為是蘋果。即使這些特徵依賴於彼此或其他特徵的存在，所有這些屬性都獨立地促成了這種果實是蘋果的概率，這就是為什麼它被稱為“天真的”。

樸素貝葉斯模型易於構建，對於非常大的數據集尤其有用。除了簡單之外，樸素貝葉斯也被稱為超高級分類方法。

貝葉斯定理提供了一種從 $P(c)$ ， $P(x)$ 和 $P(x|c)$ 計算後驗概率 $P(c|x)$ 的方法。看下面的等式：



The diagram shows the formula $P(c|x) = \frac{P(x|c)P(c)}{P(x)}$ with arrows pointing to each term: $P(c|x)$ is labeled 'Posterior Probability', $P(x|c)$ is labeled 'Likelihood', $P(c)$ is labeled 'Class Prior Probability', and $P(x)$ is labeled 'Predictor Prior Probability'.

$$P(c|X) = P(x_1|c) \times P(x_2|c) \times \dots \times P(x_n|c) \times P(c)$$

以上,

$P(c|x)$ 是給定預測器 (x , 屬性) 的類 (c , 目標) 的後驗概率。

$P(c)$ 是先前的上課概率。

$P(x|c)$ 是預測器給定類的概率的可能性。

$P(x)$ 是預測器的先驗概率。

朴素貝葉斯算法如何運作？

讓我們用一個例子來理解它。下面我有一個天氣和相應的目標變量“播放”的訓練數據集（建議播放的可能性）。現在，我們需要根據天氣條件對玩家是否玩遊戲進行分類。我們按照以下步驟執行。

步驟 1：將數據集轉換為頻率表

步驟 2：通過發現像“Overcast”概率= 0.29 和播放概率為 0.64 的概率來創建似然表。

Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

步驟 3：現在，使用樸素貝葉斯方程來計算每個類的後驗概率。具有最高後驗概率的課程是預測的結果。

問題：如果天氣晴朗，玩家將會玩。這個說法是正確的嗎？

我們可以使用上述討論的後驗概率的方法來解決它。

$$P(\text{Yes} | \text{Sunny}) = P(\text{Sunny} | \text{Yes}) * P(\text{Yes}) / P(\text{Sunny})$$

這裡， $P(\text{Sunny} | \text{Yes}) = 3/9 = 0.33$ ， $P(\text{Sunny}) = 5/14 = 0.36$ ， $P(\text{Yes}) = 9/14 = 0.64$

現在， $P(\text{Yes} | \text{Sunny}) = 0.33 * 0.64 / 0.36 = 0.60$ ，其概率較高。

樸素貝葉斯使用類似的方法根據各種屬性預測不同類的概率。該算法主要用於文本分類和具有多個類的問題。

樸素貝葉斯的優點和缺點是什麼？

優點：

預測類別的測試數據集是容易和快速的。它也在多類預測中表現良好。當獨立性假設成立時，樸素貝葉斯分類器與其他模型（如邏輯回歸）相比表現更好，您需要較少的培訓數據。

與數字變量相比，在分類輸入變量的情況下表現良好。對於數值變量，假設正態分佈（鐘形曲線，這是一個很強的假設）。

缺點：

如果分類變量具有在訓練數據集中沒有觀察到的類別（在測試數據集中），則模型將分配 0（零）概率，並且將不能進行預測。這通常被稱為“零頻率”。為了解決這個問題，我們可以使用平滑技術。最簡單的平滑技術之一稱為拉普拉斯估計。

另一方面，幼稚貝葉斯也被稱為不良估計量，所以來自 `predict_proba` 的概率輸出不會被太過於重視。

樸素貝葉斯的另一個限制是獨立預測因子的假設。在現實生活中，我們幾乎不可能得到一套完全獨立的預測因子。

樸素貝葉斯算法的應用

實時預測：樸素貝葉斯是一種熱切的學習分類器，它確實很快。因此，它可

以用於實時預測。

多類預測：該算法對於多類預測特徵也是眾所周知的。這裡我們可以預測目標變量多類的概率。

文本分類/垃圾郵件過濾/情緒分析：與其他算法相比，主要用於文本分類的樸素貝葉斯分類器（由於更好地導致多類問題和獨立性規則）具有較高的成功率。因此，它廣泛應用於垃圾郵件過濾（識別垃圾郵件）和情緒分析（在社交媒體分析中，以確定客戶的積極和消極情緒）

推薦系統：樸素貝葉斯分類器和協同過濾一起構建了一個推薦系統，該系統使用機器學習和數據挖掘技術來過濾未看到的信息，並預測用戶是否喜歡給定的資源

如何使用樸素貝葉斯在 Python 中構建基本模型？

再次，scikit 學習（python 庫）將幫助這裡在 Python 中構建一個樸素的貝葉斯模型。在 scikit 學習庫中有三種類型的樸素貝葉斯模型：

高斯：它在分類中被使用，它假設特徵遵循正態分佈。

多項式：用於離散計數。例如，假設我們有文本分類問題。在這裡，我們可以考慮 bernoulli 試驗，這是進一步的，而不是“在文檔中出現文字”，我們已經“計算文檔中出現的字數”，您可以將其視為“觀察結果數 x_i 的次數在 n 個試驗”。

Bernoulli：如果您的特徵向量是二進制（即零和一），則二項模型很有用。一個應用程序將是文本分類，其中“1”和“0”是文本中出現的“單詞”，“單詞不會出現在文檔中”。

根據您的數據集，您可以選擇上述任何一種模型。以下是高斯模型的例子。

範例程式 Bayes-1.py

```
1. #Import Library of Gaussian Naive Bayes model
2. from sklearn.naive_bayes import GaussianNB
3. import numpy as np
4.
5. #assigning predictor and target variables
6. x=np.array([[[-3,7],[1,5], [1,2], [-2,0], [2,3], [-4,0], [-1,1], [1,1], [-2,2], [2,7], [-4,1], [-2,7]]])
7. y=np.array([3, 3, 3, 3, 4, 3, 3, 4, 3, 4, 4, 4])
8. #Create a Gaussian Classifier
9. model = GaussianNB()
10. # Train the model using the training sets
11. model.fit(x,y)
```

```
12. #Predict Output
13. predicted= model.predict([[1,2],[3,4]])
14. print predicted
```

```
[powens-MacBook-Air:Desktop powenko$ python Bayes-1.py
[3 4]
```

圖 0517-1

以上，我們來看基本的樸素貝葉斯模型，您可以通過調整參數和智能處理假設來提高這個基本模型的力量。我們來看看提高樸素貝葉斯模型性能的方法。我建議您通過本文檔了解有關使用樸素貝葉斯的文本分類的更多詳細信息。

23.1.6 提高樸素貝葉斯模型 Naive Bayes Model 的力

量的提示

以下是提高樸素貝斯模型的一些技巧：

如果連續特徵沒有正態分佈，我們應該使用轉換或不同的方法在正態分佈中進行轉換。

如果測試數據集具有零頻率問題，應用平滑技術“拉普拉斯校正”來預測測試數據集的類別。

刪除相關特徵，因為高度相關的特徵在模型中被投票了兩次，並且可能導致過度膨脹的重要性。

樸素貝葉斯分類器具有參數調整的有限選項，如 $\alpha = 1$ ，用於平滑，`fit_prior = [True | False]` 以學習類先驗概率和一些其他選項（請看這裡的細節）。我建議您專注於對數據的預處理和功能選擇。

您可能會考慮應用一些分類器組合技術，如合奏，包裝和提升，但這些方法將無濟於事。其實，“合奏，提升，裝袋”是不會有幫助的，因為他們的目的只是減少差異。樸素貝葉斯沒有差異最小化。

總結：

我們研究了一種主要用於分類的監督機器學習算法“朴素貝葉斯”。恭喜，如果您徹底了解本文，您已經採取了第一步來掌握此算法。從這裡，你所需要的就是練習。

此外，我建議您在應用朴素貝葉斯算法之前，更多地關注數據預處理和特徵選擇。在將來的帖子中，我將更詳細地討論使用貝葉斯的文本和文檔分類。

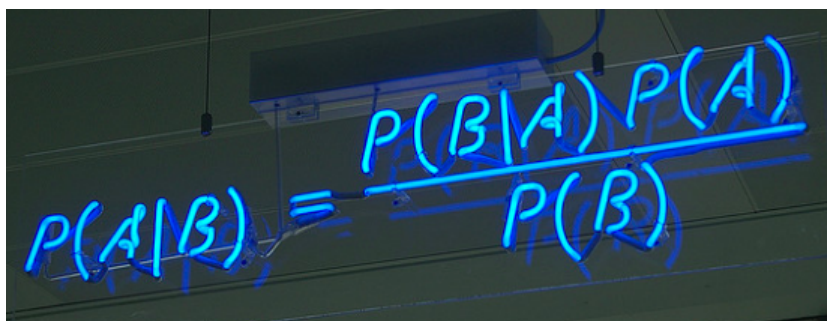
Python Lib **NaiveBayes 1.0.0**

<https://pypi.python.org/pypi/NaiveBayes>

23.2 如何在 Python 中實現朴素的貝葉斯

The Naive Bayes algorithm 貝葉斯算法簡單有效，應該是您嘗試分類問題的第一種方法之一。

在本教程中，您將了解 Naive Bayes 算法，包括它的工作原理以及如何在 Python 中從頭開始實現。


$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

關於 Naive 貝葉斯

Naive 貝葉斯算法是一種直觀的方法，它使用屬於每個類的每個屬性的概率進行預測。如果您想要概率地建模一個預測模型問題，那麼你會想出來的監督學習方法。

朴素貝葉斯通過假設屬於給定類值的每個屬性的概率與所有其他屬性無關，簡化了概率的計算。這是一個強有力的假設，但結果是快速有效的方法。

賦予屬性值的類值的概率稱為條件概率。通過將給定類值的每個屬性的條件概率相乘，我們具有屬於該類的數據實例的概率。

為了進行預測，我們可以計算屬於每個類的實例的概率，並以最高概率選擇類值。

通常使用分類數據描述天真的基礎，因為它易於描述和計算使用比率。為了我們的目的，算法的更有用的版本支持數字屬性，並且假設每個數值屬性的值是正態分佈的（落在鐘形曲線上的某個地方）。再次，這是一個強有力的假設，但仍然給出了可觀的結果。

預測糖尿病發病

我們將在本教程中使用的測試問題是 Pima Indians 糖尿病問題。

這個問題包括對皮馬印第安人專利的醫療細節的 768 條意見。記錄描述了從患者獲取的即時測量，例如其年齡，懷孕和血液處理次數。所有患者均為 21 歲以上的女性。所有屬性都是數字的，它們的單位因屬性而異。

每個記錄都有一個類別值，表明患者是否在進行測量後 5 年內發生糖尿病（1）（否）（0）。

這是一個在機器學習文獻中已經被很多研究的標準數據集。良好的預測精度為 70%-

76%。

以下是 `pima-indians.data.csv` 文件中的一個示例，以了解我們將要使用的數據。

樸素貝葉斯算法教程

本教程分為以下步驟：

處理數據：從 `CSV` 文件中加載數據並將其分解為訓練和測試數據集。

總結數據：總結訓練數據集中的屬性，以便我們可以計算概率並進行預測。

進行預測：使用數據集的摘要來生成單個預測。

進行預測：生成給定測試數據集和匯總的訓練數據集的預測。

評估準確性：評估測試數據集所做預測的準確性，因為所做預測的百分比正確。

連接在一起：使用所有代碼元素來呈現樸素貝葉斯算法的完整和獨立的實現。

處理數據

我們需要做的第一件事是加載我們的數據文件。數據是 `CSV` 格式，沒有標題行或任何引號。我們可以使用打開的功能打開文件，並使用 `csv` 模塊中的閱讀器功能讀取數據行。

我們還需要將作為字符串加載的屬性轉換為可以與它們一起使用的數字。以下是用於加載 Pima indians 數據集的 `loadCsv()` 函數。

```
import csv

def loadCsv(filename):
    lines = csv.reader(open(filename, "rb"))
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset
```

我們可以通過加載 `pima indians` 數據集並打印加載的數據實例的數量來測試此功能。

```
filename = 'pima-indians-diabetes.data.csv'
```

Python

```
dataset = loadCsv(filename)

print('Loaded data file {0} with {1} rows').format(filename, len(dataset))
```

運行這個測試，你應該看到如下：

Loaded data file pima-indians-diabetes.data.csv rows

接下來，我們需要將數據分割成一個訓練數據集，讓 **Naive Bayes** 可以用來做出預測，並且可以使用一個測試數據集來評估模型的準確性。我們需要將數據集隨機分為訓練和數據集，比例為 67% 和 33% 的測試（這是測試數據集上算法的常用比例）。

以下是將給定的數據集拆分成給定的分割比率的 `splitDataset`（）函數。

```
import random

def splitDataset(dataset, splitRatio):

    trainSize = int(len(dataset) * splitRatio)

    trainSet = []

    copy = list(dataset)

    while len(trainSet) < trainSize:

        index = random.randrange(len(copy))

        trainSet.append(copy.pop(index))

    return [trainSet, copy]
```

我們可以通過定義一個具有 5 個實例的模擬數據集來進行測試，將其分為訓練和測試數據集，然後打印出來，查看哪些數據實例結束於哪裡。

```
dataset = [[1], [2], [3], [4], [5]]

splitRatio = 0.67

train, test = splitDataset(dataset, splitRatio)

print('Split {0} rows into train with {1} and test with {2}').format(len(dataset), train, test)
```

運行這個測試，你應該看到如下：

Split 5 rows into train with [[4], [3], [5]] and test with [[1], [2]]

總結數據

幼稚貝葉斯模型由訓練數據集中數據的總結組成。然後在進行預測時使用此摘要。

收集的培訓數據的總結涉及每個屬性的平均值和標準差，按類別值。例如，如果有兩個類值和 7 個數值屬性，則我們需要每個屬性（7）和類值（2）組合的平均值和標準偏差，即 14 個屬性摘要。

當進行預測以計算屬於每個類值的特定屬性值的概率時，這些是必需的。

我們可以將此摘要數據的準備工作分解為以下子任務：

按類分隔數據

計算平均值

計算標準偏差

匯總數據集

按類匯總屬性

按類分隔數據

第一個任務是通過類值分隔訓練數據集實例，以便我們可以計算每個類的統計信息。我們可以通過將每個類值的映射創建為屬於該類的實例列表，並將整個實例數據集排序到適當的列表中。

下面的 `separateByClass()` 函數只是這樣。

```
def separateByClass(dataset):  
    separated = {}  
    for i in range(len(dataset)):  
        vector = dataset[i]  
        if (vector[-1] not in separated):  
            separated[vector[-1]] = []  
        separated[vector[-1]].append(vector)  
    return separated
```

您可以看到該函數假定最後一個屬性（-1）是類值。該函數將類值的映射返回到數據實例的列表。

我們可以用一些樣本數據來測試這個函數，如下所示：

```
dataset = [[1,20,1], [2,21,0], [3,22,1]]
separated = separateByClass(dataset)
print('Separated instances: {0}').format(separated)
```

運行這個測試，你應該看到如下：

```
Separated instances: {0: [[2, 21, 0]], 1: [[1, 20, 1], [3, 22, 1]]}
```

計算平均值

我們需要計算一個類值的每個屬性的平均值。平均值是數據的中心或中心趨勢，在計算概率時，我們將其用作高斯分佈的中間。

我們還需要計算一個類值的每個屬性的標準偏差。標準偏差描述了數據擴散的變化，我們將使用它來表徵計算概率時每個屬性在我們的高斯分佈中的預期擴展。

標準差計算為方差的平方根。方差計算為每個屬性值與平均值的平方差的平均值。注意，我們使用 $N-1$ 方法，它在計算方差時從屬性值的數量中減去 1。

```
import math
def mean(numbers):
    return sum(numbers)/float(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)
```

我們可以通過將數字的平均值從 1 到 5 進行測試。

Python

```
numbers = [1,2,3,4,5]
print('Summary of {0}: mean={1}, stdev={2}'.format(numbers, mean(numbers),
stdev(numbers)))
```

運行這個測試，你應該看到如下：

```
Summary of [1, 2, 3, 4, 5]: mean=3.0, stdev=1.58113883008
```

匯總數據集

現在我們有了總結數據集的工具。對於給定的實例列表（對於類值），我們可以計算每個屬性的平均值和標準偏差。

zip 功能將我們的數據實例中的每個屬性的值分組到自己的列表中，以便我們可以計算屬性的平均值和標準偏差值。

```
def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
    del summaries[-1]
    return summaries
```

我們可以用一些測試數據來測試這個 summarize () 函數，這些測試數據顯示了第一和第二個數據屬性的顯著不同的平均值和標準偏差值。

```
dataset = [[1,20,0], [2,21,1], [3,22,0]]
summary = summarize(dataset)
print('Attribute summaries: {0}').format(summary)
```

運行這個測試，你應該看到如下：

```
Attribute summaries: [(2.0, 1.0), (21.0, 1.0)]
```

按類匯總屬性

我們可以通過首先將我們的訓練數據集分為按課程分組的實例，將它們整合在一起。然後計算每個屬性的摘要。

```
def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.iteritems():
```

Python

```
summaries[classValue] = summarize(instances)
return summaries
```

我們可以用一個小的測試數據集來測試這個 `summaryByClass ()` 函數。

```
dataset = [[1,20,1], [2,21,0], [3,22,1], [4,22,0]]
summary = summarizeByClass(dataset)
print('Summary by class value: {0}'.format(summary))
```

運行這個測試，你應該看到如下：

```
Summary by class value:
{0: [(3.0, 1.4142135623730951), (21.5, 0.7071067811865476)],
1: [(2.0, 1.4142135623730951), (21.0, 1.4142135623730951)]}
```

3. 預測

我們現在可以使用從我們的培訓數據準備的摘要進行預測。進行預測涉及計算給定數據實例屬於每個類的概率，然後選擇具有最大概率的類作為預測。

我們可以將這部分劃分為以下任務：

- 計算高斯概率密度函數
- 計算類概率
- 做一個預測
- 估計精度
- 計算高斯概率密度函數

給定從訓練數據估計的屬性的已知平均值和標準偏差，我們可以使用高斯函數來估計給定屬性值的概率。

假設屬性摘要為每個屬性和類值準備，結果是給定類值的給定屬性值的條件概率。

關於高斯概率密度函數的這個方程的細節參見參考文獻。總而言之，我們將我們已知的細節插入高斯（屬性值，平均值和標準偏差），並讀出我們的屬性值屬於類的可能性。

在 `calculateProbability ()` 函數中，我們首先計算指數，然後計算主分區。這使我們能夠很好地適應兩個方程式的方程式。

```
import math
```

Python

```
def calculateProbability(x, mean, stdev):  
    exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))  
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent
```

我們可以用一些樣本數據進行測試，如下所示。

```
x = 71.5  
mean = 73  
stdev = 6.2  
probability = calculateProbability(x, mean, stdev)  
print('Probability of belonging to this class: {0}').format(probability)
```

運行這個測試，你應該看到如下：

Probability of belonging to this class: 0.0624896575937

計算類概率

現在我們可以計算一個屬性屬性的概率，我們可以組合一個數據實例的所有屬性值的概率，並得出屬於該類的整個數據實例的概率。

我們將概率相乘在一起。在以下的 `calculateClassProbabilities()` 中，通過將每個類的屬性概率相乘來計算給定數據實例的概率。結果是類值到概率的映射。

```
def calculateClassProbabilities(summaries, inputVector):  
    probabilities = {}  
    for classValue, classSummaries in summaries.iteritems():  
        probabilities[classValue] = 1  
        for i in range(len(classSummaries)):  
            mean, stdev = classSummaries[i]  
            x = inputVector[i]  
            probabilities[classValue] *= calculateProbability(x, mean, stdev)  
    return probabilities
```

我們可以測試 `calculateClassProbabilities()` 函數。

```
summaries = {0:[(1, 0.5)], 1:[(20, 5.0)]}  
inputVector = [1.1, '?']  
probabilities = calculateClassProbabilities(summaries, inputVector)  
print('Probabilities for each class: {0}').format(probabilities)
```

運行這個測試，你應該看到如下：

Probabilities for each class: {0: 0.7820853879509118, 1: 6.298736258150442e-05}

做一個預測

現在可以計算屬於每個類值的數據實例的概率，我們可以查找最大的概率並返回相關類。

`belongs ()` 函數屬於這個。

```
def predict(summaries, inputVector):
    probabilities = calculateClassProbabilities(summaries, inputVector)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.iteritems():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel
```

我們可以測試 `predict ()` 函數如下：

```
summaries = {'A':[(1, 0.5)], 'B':[(20, 5.0)]}
inputVector = [1.1, '?']
result = predict(summaries, inputVector)
print('Prediction: {0}').format(result)
```

運行這個測試，你應該看到如下：

Prediction: A

4. 預測

最後，我們可以通過對我們的測試數據集中的每個數據實例進行預測來估計模型的準確性。`getPredictions ()` 將執行此操作並返回每個測試實例的預測列表。

```
def getPredictions(summaries, testSet):
    predictions = []
```

Python

```
for i in range(len(testSet)):
    result = predict(summaries, testSet[i])
    predictions.append(result)
return predictions
```

我們可以測試 `getPredictions ()` 函數。

```
summaries = {'A':[(1, 0.5)], 'B':[(20, 5.0)]}
testSet = [[1.1, '?'], [19.1, '?']]
predictions = getPredictions(summaries, testSet)
print('Predictions: {0}').format(predictions)
```

運行這個測試，你應該看到如下：

```
Predictions: ['A', 'B']
```

23.2.1 5. 準確度

可以將預測與測試數據集中的類值進行比較，並且分類精度可以計算為 0 和 100% 之間的精度比。 `getAccuracy ()` 將計算此精度比。

```
def getAccuracy(testSet, predictions):
    correct = 0
    for x in range(len(testSet)):
        if testSet[x][-1] == predictions[x]:
            correct += 1
    return (correct/float(len(testSet))) * 100.0
```

我們可以使用下面的示例代碼測試 `getAccuracy ()` 函數。

```
testSet = [[1,1,1,'a'], [2,2,2,'a'], [3,3,3,'b']]
predictions = ['a', 'a', 'a']
accuracy = getAccuracy(testSet, predictions)
print('Accuracy: {0}').format(accuracy)
```

運行這個測試，你應該看到如下：

```
Accuracy: 66.6666666667
```

第24章 隨機森林

隨機森林

隨機森林是一種多功能的機器學習方法，具有從營銷到醫療保健和保險的眾多應用。它可用於模擬營銷對客戶獲取，保留和流失的影響，或預測患者的疾病風險和易感性。

隨機森林能夠回歸和分類。它可以處理大量功能，它有助於估計哪些變量在建模的基礎數據中很重要。

這是一篇關於使用 Python 的隨機森林的帖子。

什麼是隨機森林？

隨機森林幾乎可以用於任何預測問題（甚至是非線性問題）。這是一種相對較新的機器學習策略（它出自 90 年代的貝爾實驗室），幾乎可以用於任何事情。它屬於一類稱為集合方法的機器學習算法。

集成學習涉及幾種模型的組合以解決單個預測問題。它的工作原理是生成多個獨立學習和預測的分類器/模型。然後將這些預測組合成單個（兆）預測，該預測應該與任何一個分類器的預測一樣好或更好。

隨機森林是一個集合學習的品牌，因為它依賴於決策樹的集合。

隨機決策樹

所以我們知道隨機森林是其他模型的聚合，但它聚合的模型類型是什麼？正如您可能從其名稱中猜到的那樣，隨機森林聚合了分類（或回歸）樹。決策樹由一系列決策組成，可用於對數據集中的觀察進行分類。

隨機森林

誘導隨機森林的算法將自動創建一堆隨機決策樹。由於樹是隨機生成的，因此大多數樹對於學習分類/回歸問題（可能是 99.9% 的樹）都沒有意義。

1) 樹棲投票

10000（可能）壞模型有什麼用呢？事實證明，他們真的沒那麼有用。但是有用的是少數非常好的決策樹，你也會和壞的決策樹一起生成。

進行預測時，新觀察將被推送到每個決策樹並分配預測值/標籤。一旦森林中的每棵樹都報告了其預測值/標籤，就會對預測進行統計，並將所有樹的模式投票作為最終預測返回。

簡單地說，99.9%的無關樹木會在地圖上進行預測並相互抵消。少數樹木的預測是最好的噪音，並產生良好的預測。

本教程的數據很有名。稱為 **Iris** 據集，它包含四個變量，用於測量三個相關物種的虹膜花的各個部分，然後是第四個變量的物種名稱。它在機器學習和統計社區中如此著名的原因是因為數據需要很少的預處理（即沒有缺失值，所有特徵都是浮點數等）。

匯入資料

```
# Load the library with the iris dataset
from sklearn.datasets import load_iris
# Load scikit's random forest classifier library
from sklearn.ensemble import RandomForestClassifier
# Load pandas
import pandas as pd
# Load numpy
import numpy as np
# Set random seed
np.random.seed(0)
```

將 Iris 資料讀取

```
# Create an object called iris with the iris data
iris = load_iris()
# Create a dataframe with the four feature variables
df = pd.DataFrame(iris.data, columns=iris.feature_names)
# View the top 5 rows
```

```
print(df.head())
```

	sepal length (cm)	...	petal width (cm)
0	5.1	...	0.2
1	4.9	...	0.2
2	4.7	...	0.2
3	4.6	...	0.2
4	5.0	...	0.2

添加種類資料

```
# Add a new column with the species names, this is what we are going to try to predict
df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)
# View the top 5 rows
print(df.head())
```

	sepal length (cm)	sepal width (cm)	...	petal width (cm)	species
0	5.1	3.5	...	0.2	setosa
1	4.9	3.0	...	0.2	setosa
2	4.7	3.2	...	0.2	setosa
3	4.6	3.1	...	0.2	setosa
4	5.0	3.6	...	0.2	setosa

創建培訓和測試數據

```
# Create a new column that for each row, generates a random number between 0 and 1, and
# if that value is less than or equal to .75, then sets the value of that cell as True
# and false otherwise. This is a quick and dirty way of randomly assigning some rows to
# be used as the training data and some as the test data.
df['is_train'] = np.random.uniform(0, 1, len(df)) <= .75
# View the top 5 rows
print(df.head())
```

	sepal length (cm)	sepal width (cm)	...	species	is_train
0	5.1	3.5	...	setosa	True
1	4.9	3.0	...	setosa	True
2	4.7	3.2	...	setosa	True
3	4.6	3.1	...	setosa	True
4	5.0	3.6	...	setosa	True

```
# Create two new dataframes, one with the training rows, one with the test rows
train, test = df[df['is_train']==True], df[df['is_train']==False]
# Show the number of observations for the test and training dataframes
print('Number of observations in the training data:', len(train))
print('Number of observations in the test data:', len(test))
```

```
Number of observations in the training data: 118
Number of observations in the test data: 32
```

處理資料

```
# Create a list of the feature column's names
features = df.columns[:4]
# View features
print(features)
```

```
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
      'petal width (cm)'],
      dtype='object')
```

```
# train['species'] contains the actual species names. Before we can use it,
# we need to convert each species name into a digit. So, in this case there
# are three species, which have been coded as 0, 1, or 2.
y = pd.factorize(train['species'])[0]
# View target
print(y)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2]
```

訓練隨機森林

```
# Create a random forest Classifier. By convention, clf means 'Classifier'
clf = RandomForestClassifier(n_jobs=2, random_state=0)
# Train the Classifier to take the training features and learn how they relate
# to the training y (the species)
clf.fit(train[features], y)
```

```
RandomForestClassifier(bootstrap=True,
class_weight=None, criterion='gini',
max_depth=None, max_features='auto',
max_leaf_nodes=None,
min_impurity_split=1e-07, min_samples_leaf=1,
min_samples_split=2, min_weight_fraction_leaf=0.0,
n_estimators=10, n_jobs=2, oob_score=False,
random_state=0,
verbose=0, warm_start=False)
```

好哇！我們做到了！我們正式訓練了我們的隨機森林分類器！現在讓我們玩吧。分類器模型本身存儲在 `clf` 變量中。

測試數據

你會知道我們只在部分數據上訓練了我們的分類器，剩下的就完成了。在我看來，這是機器學習中最重要的部分。為什麼？因為省略了部分數據，我們有一組數據來測試我們模型的準確性！

我們現在就這樣做。

```
# Apply the Classifier we trained to the test data (which, remember, it has never seen before)
print(clf.predict(test[features]))
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2]
```

你在上面看什麼？請記住，我們將三种植物中的每一種都編碼為 0,1 或 2. 上面的數字列表顯示了我們的模型預測每种植物的種類是基於萼片長度，萼片寬度，花瓣長度，和花瓣寬度。每种植物的分類器有多自信？我們也可以看到。

```
print(clf.predict_proba(test[features]))[:])
```

```
[[1.  0.  0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [0.9 0.1 0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [1.  0.  0. ]
 [0.9 0.1 0. ]
 [1.  0.  0. ]
 [0.  0.5 0.5]
 [0.  1.  0. ]
 [0.  0.9 0.1]
 [0.  0.2 0.8]
 [0.  0.3 0.7]
```

有三種植物，因此[1., 0., 0.]告訴我們，分類器確定植物是第一類。再舉一個例子，[0.9,0.1,0.]告訴我們，分類器給出了植物屬於第一類的概率為90%，植物屬於第二類的概率為10%。因為90大於10，分類器預測植物是第一類。

評估分類器

現在我們已經預測了測試數據中所有植物的種類，我們可以將我們預測的物種與該植物的實際物種進行比較。

```
# Evaluate Classifier
# Create actual english names for the plants for each predicted plant class
preds = iris.target_names[clf.predict(test[features])]
```



```
# View the PREDICTED species for the first five observations
print(preds[0:5])
```

```
['setosa' 'setosa' 'setosa' 'setosa' 'setosa']
```

```
print(test['species'].head())
```

```
7      setosa
8      setosa
10     setosa
13     setosa
17     setosa
```

看起來很不錯！ 至少對於前五個觀察結果。 現在讓我們看看所有數據。

創建一個混淆矩陣

混淆矩陣可以是，沒有雙關語意思，起初有點令人困惑，但它實際上非常簡單。 列是我們為測試數據預測的物種，行是測試數據的實際物種。 因此，如果我們採取最上面的行，我們可以完美地預測測試數據中的所有 13 個 *setosa* 植物。然而，在下一行中，我們正確地預測了 5 種雜色植物，但錯誤地預測了兩種雜色植物為維吉尼亞。

如何解釋混淆矩陣的簡短解釋是：對角線上的任何東西都被正確分類，對角線上的任何東西都被錯誤地分類。

```
# Create confusion matrix
print(pd.crosstab(test['species'], preds, rownames=['Actual Species'], colnames=['Predicted Species']))
```

Predicted Species	setosa	versicolor	virginica
Actual Species			
setosa	13	0	0
versicolor	0	5	2
virginica	0	0	12

Python

```
#View Feature Importance
```

```
print(list(zip(train[features], clf.feature_importances_)))
```

查看功能重要性

雖然我們沒有像 OLS 那樣得到回歸係數，但我們得到的分數告訴我們每個特徵在分類中的重要性。這是隨機森林中最強大的部分之一，因為我們可以清楚地看到花瓣寬度在分類中比萼片寬度更重要。

```
#View Feature Importance
```

```
print(list(zip(train[features], clf.feature_importances_)))
```

輸出：

```
[('sepal length (cm)', 0.11185992930506346), ('sepal width (cm)', 0.016341813006098178),  
('petal length (cm)', 0.36439533040889194), ('petal width (cm)', 0.5074029272799464)]
```

第25章 常見算法優缺點

1. 朴素貝葉斯

朴素貝葉斯屬於生成式模型（關於生成模型和判別式模型，主要還是在於是否是要求聯合分布），非常簡單，你只是做了一堆計數。如果注有條件獨立性假設（一個比較嚴格的條件），朴素貝葉斯分類器的收斂速度將快於判別模型，如邏輯回歸，所以你只需要較少的訓練數據即可。即使NB條件獨立假設不成立，NB分類器在實踐中仍然表現的很出色。它的主要缺點是它不能學習特徵間的相互作用，用mRMR中R來講，就是特徵冗餘。引用一個比較經典的例子，比如，雖然你喜歡Brad Pitt和Tom Cruise的電影，但是它不能學習出你不喜歡他們在一起演的電影。

優點：

朴素貝葉斯模型發源於古典數學理論，有著堅實的數學基礎，以及穩定的分類效率。

對小規模的數據表現很好，能處理多分類任務，適合增量式訓練；

對缺失數據不太敏感，算法也比較簡單，常用於文本分類。

缺點：

Python

需要計算先驗機率；
分類決策存在錯誤率；
對輸入數據的表達形式很敏感。

2. Logistic Regression (邏輯回歸)

屬於判別式模型，有很多正則化模型的方法 (L0 , L1 , L2 , etc)，而且你不必像在用樸素貝葉斯那樣擔心你的特徵是否相關。與決策樹與 SVM 機相比，你還會得到一個不錯的機率解釋，你甚至可以輕鬆地利用新數據來更新模型 (使用在線梯度下降算法，online gradient descent)。如果你需要一個機率架構 (比如，簡單地調節分類閾值，指明不確定性，或者是要獲得置信區間)，或者你希望以後將更多的訓練數據快速整合到模型中去，那麼使用它吧。

Sigmoid 函數：

$$f(x) = \frac{1}{1 + e^{-x}}$$

優點：

實現簡單，廣泛的應用於工業問題上；
分類時計算量非常小，速度很快，存儲資源低；
便利的觀測樣本機率分數；
對邏輯回歸而言，多重共線性並不是問題，它可以結合 L2 正則化來解決該問題；

缺點：

當特徵空間很大時，邏輯回歸的性能不是很好；
容易欠擬合，一般準確度不太高
不能很好地處理大量多類特徵或變量；
只能處理兩分類問題 (在此基礎上衍生出來的 softmax 可以用於多分類)，且必須線性可分；
對於非線性特徵，需要進行轉換；

3. 線性回歸

線性回歸是用於回歸的，而不像 Logistic 回歸是用於分類，其基本思想是用梯度下降法對最小二乘法形式的誤差函數進行優化，當然也可以用 normal equation 直接求得參數的解，結果為：

$$\hat{w} = (X^T X)^{-1} X^T y$$

而在 LWLR (局部加權線性回歸) 中，參數的計算表達式為：

$$\hat{w} = (X^T X)^{-1} X^T y$$

由此可見 LWLR 與 LR 不同，LWLR 是一個非參數模型，因為每次進行回歸計算都要遍歷訓練樣本至少一次。

優點：實現簡單，計算簡單；

缺點：不能擬合非線性數據。

4.最近鄰算法——KNN

KNN 即最近鄰算法，其主要過程為：

計算訓練樣本和測試樣本中每個樣本點的距離（常見的距離度量有歐式距離，馬氏距離等）；

對上面所有的距離值進行排序；

選前 k 個最小距離的樣本；

根據這 k 個樣本的標籤進行投票，得到最後的分類類別；

如何選擇一個最佳的 K 值，這取決於數據。一般情況下，在分類時較大的 K 值能夠減小噪聲的影響。但會使類別之間的界限變得模糊。一個較好的 K 值可通過各種啟發式技術來獲取，比如，交叉驗證。另外噪聲和非相關性特徵向量的存在會使 K 近鄰算法的準確性減小。

近鄰算法具有較強的一致性結果。隨著數據趨於無限，算法保證錯誤率不會超過貝葉斯算法錯誤率的兩倍。對於一些好的 K 值，K 近鄰保證錯誤率不會超過貝葉斯理論誤差率。

KNN 算法的優點

理論成熟，思想簡單，既可以用來做分類也可以用來做回歸；

可用於非線性分類；

訓練時間複雜度為 $O(n)$ ；

對數據沒有假設，準確度高，對 outlier 不敏感；

缺點

計算量大；

樣本不平衡問題（即有些類別的樣本數量很多，而其它樣本的數量很少）；

需要大量的內存；

5.決策樹

易於解釋。它可以毫無壓力地處理特徵間的交互關係並且是非參數化的，因此你不必擔心異常值或者數據是否線性可分（舉個例子，決策樹能輕鬆處理好類別 A 在某個特徵維度 x 的末端，類別 B 在中間，然後類別 A 又出現在特徵維度 x 前端的情況）。它的缺點之一就是不支持在線學習，於是在新樣本到來後，決策樹需要全部重建。另一個缺點就是容易出現過擬合，但這也就是諸如隨機森林 RF（或提升樹 boosted tree）之類的集成方法的切入點。另外，隨機森林經常是很多分類問題的贏家（通常比支持向量機好上那麼一丁點），它訓練快速並且可調，同時你無須擔心要像支持向量機那樣調一大堆參數，所以在以前都一直很受歡迎。

決策樹中很重要的一點就是選擇一個屬性進行分枝，因此要注意一下信息增益的計算公式，並深入理解它。

信息熵的計算公式如下：

$$H = - \sum_{i=1}^n p(x_i) \log_2 p(x_i)$$

其中的 n 代表有 n 個分類類別（比如假設是 2 類問題，那麼 $n=2$ ）。分別計算這 2 類樣本在總樣本中出現的機率 p_1 和 p_2 ，這樣就可以計算出未選中屬性分枝前的信息熵。

現在選中一個屬性 x_i 用來進行分枝，此時分枝規則：如果 $x_i = v$ 的話，將樣本分到樹的一個分支；如果不相等則進入另一個分支。很顯然，分支中的樣本很有可能包括 2 個類別，分別計算這 2 個分支的熵 H_1 和 H_2 ，計算出分枝後的總信息熵 $H' = p_1 H_1 + p_2 H_2$ ，則此時的信息增益 $\Delta H = H - H'$ 。以信息增益為原則，把所有的屬性都測試一邊，選擇一個使增益最大的屬性作為本次分枝屬性。

決策樹自身的優點

計算簡單，易於理解，可解釋性強；

比較適合處理有缺失屬性的樣本；

能夠處理不相關的特徵；

在相對短的時間內能夠對大型數據源做出可行且效果良好的結果。

缺點

容易發生過擬合（隨機森林可以很大程度上減少過擬合）；

忽略了數據之間的相關性；

對於那些各類別樣本數量不一致的數據，在決策樹當中，信息增益的結果偏向於那些具有更多數值的特徵（要是使用了信息增益，都有這個缺點，如 RF）。

5 Adaboosting

Adaboost 是一種加和模型，每個模型都是基於上一次模型的錯誤率來建立的，過分關注分錯的樣本，而對正確分類的樣本減少關注度，逐次疊代之後，可以得到一個相對較好的模型。是一種典型的 boosting 算法。下面是總結下它的優缺點。

優點

adaboost 是一種有很高精度的分類器。

可以使用各種方法構建子分類器，Adaboost 算法提供的是框架。

當使用簡單分類器時，計算出的結果是可以理解的，並且弱分類器的構造極其簡單。

簡單，不用做特徵篩選。

不容易發生 overfitting。

關於隨機森林和 GBDT 等組合算法，參考這篇文章：機器學習-組合算法總結

缺點：對 outlier 比較敏感

6.SVM 支持向量機

高準確率，為避免過擬合提供了很好的理論保證，而且就算數據在原特徵空間線性不可分，只要給個合適的核函數，它就能運行得很好。在動輒超高維的文本分類問題中特別受歡迎。可惜內存消耗大，難以解釋，運行和調參也有些煩人，而隨機森林卻剛好避開了這些缺點，比較實用。

優點：

可以解決高維問題，即大型特徵空間；

能夠處理非線性特徵的相互作用；

無需依賴整個數據；

可以提高泛化能力；

缺點：

當觀測樣本很多時，效率並不是很高；

對非線性問題沒有通用解決方案，有時候很難找到一個合適的核函數；

對缺失數據敏感；

對於核的選擇也是有技巧的（libsvm 中自帶了四種核函數：線性核、多項式核、RBF 以及 sigmoid 核）：

第一，如果樣本數量小於特徵數，那麼就沒必要選擇非線性核，簡單的使用線性核就可以了；

第二，如果樣本數量大於特徵數目，這時可以使用非線性核，將樣本映射到更高維度，一般可以得到更好的結果；

第三，如果樣本數目和特徵數目相等，該情況可以使用非線性核，原理和第二種一樣。

對於第一種情況，也可以先對數據進行降維，然後使用非線性核，這也是一種方法。

7. 人工神經網絡的優缺點

人工神經網絡的優點：

分類的準確度高；

並行分布處理能力強，分布存儲及學習能力強，

對噪聲神經有較強的魯棒性和容錯能力，能充分逼近複雜的非線性關係；

具備聯想記憶的功能。

人工神經網絡的缺點：

神經網絡需要大量的參數，如網絡拓撲結構、權值和閾值的初始值；

不能觀察之間的學習過程，輸出結果難以解釋，會影響到結果的可信度和可接受程度；

學習時間過長，甚至可能達不到學習的目的。

8、K-Means 聚類

K-Means 聚類的文章，博文連結：[機器學習算法-K-means 聚類](#)。關於K-Means 的推導，裡面有著很強大的EM 思想。

優點

算法簡單，容易實現；

對處理大數據集，該算法是相對可伸縮的和高效率的，因為它的複雜度大約是 $O(nkt)$ ，其中 n 是所有對象的數目， k 是簇的數目， t 是疊代的次數。通常 $k \ll n$ 。這個算法通常局部收斂。

算法嘗試找出使平方誤差函數值最小的 k 個劃分。當簇是密集的、球狀或團狀的，且簇與簇之間區別明顯時，聚類效果較好。

缺點

對數據類型要求較高，適合數值型數據；

可能收斂到局部最小值，在大規模數據上收斂較慢

K 值比較難以選取；

對初值的簇心值敏感，對於不同的初始值，可能會導致不同的聚類結果；

不適合於發現非凸面形狀的簇，或者大小差別很大的簇。

對於「噪聲」和孤立點數據敏感，少量的該類數據能夠對平均值產生極大影響。

算法選擇參考

之前翻譯過一些國外的文章，有一篇文章中給出了一個簡單的算法選擇技巧：

首當其衝應該選擇的就是邏輯回歸，如果它的效果不怎麼樣，那麼可以將它的結果作為基準來參考，在基礎上與其他算法進行比較；

然後試試決策樹（隨機森林）看看是否可以大幅度提升你的模型性能。即便最後你並沒有把它當做為最終模型，你也可以使用隨機森林來移除噪聲變量，做特徵選擇；

如果特徵的數量和觀測樣本特別多，那麼當資源和時間充足時（這個前提很重要），使用 SVM 不失為一種選擇。

通常情況下：【GBDT>SVM>RF>Adaboost>Other...】，現在深度學習很熱門，很多領域都用到，它是以神經網絡為基礎的，目前我自己也在學習，只是理論知識不是很厚實，理解的不夠深，這裡就不做介紹了。

算法固然重要，但好的數據卻要優於好的算法，設計優良特徵是大有裨益的。假如你有一個超大數據集，那麼無論你使用哪種算法可能對分類性能都沒太大影響（此時就可以根據速度和易用性來進行抉擇）。

25.1 用 Python 做科学计算

<http://bigsec.net/b52/scipydoc/index.html#>

25.2 機器學習和人工智慧

原文網址：<https://kknews.cc/tech/o8jo4q.html>

常見機器學習 5 大算法優缺點

Logistic Regression（邏輯回歸）

屬於判別式模型，有很多正則化模型的方法（L0，L1，L2，etc），而且你不必像在用朴素貝葉斯那樣擔心你的特徵是否相關。與決策樹與 SVM 機相比，你還會得到一個不錯的機率解釋，你甚至可以輕鬆地利用新數據來更新模型（使用在線梯度下降算法，onlinegradientdescent）。如果你需要一個機率架構（比如，簡單地調節分類閾值，指明不確定性，或者是要獲得置信區間），或者你希望以後將更多的訓練數據快速整合到模型中去，那麼使用它吧。

Sigmoid 函數

Sigmoid函數：

$$f(x) = \frac{1}{1 + e^{-x}}$$

優點：

實現簡單，廣泛的應用於工業問題上；

分類時計算量非常小，速度很快，存儲資源低；

便利的觀測樣本機率分數；

對邏輯回歸而言，多重共線性並不是問題，它可以結合 L2 正則化來解決該問題；

缺點：

當特徵空間很大時，邏輯回歸的性能不是很好；

容易欠擬合，一般準確度不太高

不能很好地處理大量多類特徵或變量；

只能處理兩分類問題（在此基礎上衍生出來的 softmax 可以用於多分類），且必須線性可分；

對於非線性特徵，需要進行轉換；

2. 樸素貝葉斯

樸素貝葉斯屬於生成式模型（關於生成模型和判別式模型，主要還是在於是否是要求聯合分布），非常簡單，你只是做了一堆計數。如果注有條件獨立性假設（一個比較嚴格的條件），樸素貝葉斯分類器的收斂速度將快於判別模型，如邏輯回歸，所以你只需要較少的訓練數據即可。即使 NB 條件獨立假設不成立，NB 分類器在實踐中仍然表現的很出色。它的主要缺點是它不能學習特徵間的相互作用，用 mRMR 中 R 來講，就是特徵冗餘。引用一個比較經典的例子，比如，雖然你喜歡 BradPitt 和 TomCruise 的電影，但是它不能學習出你不喜歡他們在一起演的電影。

優點：

樸素貝葉斯模型發源於古典數學理論，有著堅實的數學基礎，以及穩定的分類效率。

對小規模的數據表現很好，能個處理多分類任務，適合增量式訓練；

對缺失數據不太敏感，算法也比較簡單，常用於文本分類。

缺點：

需要計算先驗機率；

分類決策存在錯誤率；

對輸入數據的表達形式很敏感。

3. 線性回歸

線性回歸是用於回歸的，而不像 Logistic 回歸是用於分類，其基本思想是用梯度下降法對最小二乘法形式的誤差函數進行優化，當然也可以用 normal equation 直接求得參數的解，結果為：

$$\hat{w} = (X^T X)^{-1} X^T y$$

而在 LWLR (局部加權線性回歸) 中，參數的計算表達式為：

$$\hat{w} = (X^T W X)^{-1} X^T W y$$

由此可見 LWLR 與 LR 不同，LWLR 是一個非參數模型，因為每次進行回歸計算都要遍歷訓練樣本至少一次。

優點：實現簡單，計算簡單；

缺點：不能擬合非線性數據。

最近領算法——KNN

KNN 即最近鄰算法，其主要過程為：

1. 計算訓練樣本和測試樣本中每個樣本點的距離（常見的距離度量有歐式距離，馬氏距離等）；
2. 對上面所有的距離值進行排序；
3. 選前 k 個最小距離的樣本；
4. 根據這 k 個樣本的標籤進行投票，得到最後的分類類別；

如何選擇一個最佳的 k 值，這取決於數據。一般情況下，在分類時較大的 k 值能夠減小噪聲的影響。但會使類別之間的界限變得模糊。一個較好的 k 值可通過各種啟發式技術來獲取，比如，交叉驗證。另外噪聲和非相關性特徵向量的存在會使 k 近鄰算法的準確性減小。

近鄰算法具有較強的一致性結果。隨著數據趨於無限，算法保證錯誤率不會超過貝葉斯算法錯誤率的兩倍。對於一些好的 k 值， k 近鄰保證錯誤率不會超過貝葉斯理論誤差率。

KNN 算法的優點

理論成熟，思想簡單，既可以用來做分類也可以用來做回歸；

可用於非線性分類；

訓練時間複雜度為 $O(n)$ ；

對數據沒有假設，準確度高，對 outlier 不敏感；

缺點

計算量大；

樣本不平衡問題（即有些類別的樣本數量很多，而其它樣本的數量很少）；

需要大量的內存；

5. 決策樹

易於解釋。它可以毫無壓力地處理特徵間的交互關係並且是非參數化的，因此你不必擔心異常值或者數據是否線性可分（舉個例子，決策樹能輕鬆處理好類別 A 在某個特徵維度 x 的末端，類別 B 在中間，然後類別 A 又出現在特徵維度 x 前端的情況）。它的缺點之一就是不支持在線學習，於是在新樣本到來後，決策樹需要全部重建。另一個缺點就是容易出現過擬合，但這也就是諸如隨機森林 RF（或提升樹 boostedtree）之類的集成方法的切入點。另外，隨機森林經常是很多分類問題的贏家（通常比支持向量機好上那麼一丁點），它訓練快速並且可調，同時你無須擔心要像支持向量機那樣調一大堆參數，所以在以前都一直很受歡迎。

決策樹中很重要的一點就是選擇一個屬性進行分枝，因此要注意一下信息

增益的計算公式，並深入理解它。

信息熵的計算公式如下：

其中的 n 代表有 n 個分類類別（比如假設是 2 類問題，那麼 $n=2$ ）。分別計算這 2 類樣本在總樣本中出現的機率 p_1 和 p_2 ，這樣就可以計算出未選中屬性分枝前的信息熵。

現在選中一個屬性 x_i 用來進行分枝，此時分枝規則是：如果 $x_i=v_{xi}=v$ 的話，將樣本分到樹的一個分支；如果不相等則進入另一個分支。很顯然，分支中的樣本很有可能包括 2 個類別，分別計算這 2 個分支的熵 H_1 和 H_2 ，計算出分枝後的總信息熵 $H_{\downarrow} = p_1 H_1 + p_2 H_2$ ，則此時的信息增益 $\Delta H = H - H_{\downarrow}$ 。以信息增益為原則，把所有的屬性都測試一邊，選擇一個使增益最大的屬性作為本次分枝屬性。

決策樹自身的優點

計算簡單，易於理解，可解釋性強；

比較適合處理有缺失屬性的樣本；

能夠處理不相關的特徵；

在相對短的時間內能夠對大型數據源做出可行且效果良好的結果。

缺點：

容易發生過擬合（隨機森林可以很大程度上減少過擬合）；

忽略了數據之間的相關性；

對於那些各類別樣本數量不一致的數據，在決策樹當中，信息增益的結果偏向於那些具有更多數值的特徵（只要是使用了信息增益，都有這個缺點，如 RF）。

25.3 分析結果改進工作,結論分析,改善工作

機器學習算法太多了，分類、回歸、聚類、推薦、圖像識別領域等等，要想找到一個合適算法真的不容易，所以在實際應用中，我們一般都是採用啟發式學習方式來實驗。通常最開始我們都會選擇大家普遍認同的算法，諸如 SVM，G BDT，Adaboost，現在深度學習很火熱，神經網絡也是一個不錯的選擇。假如你在乎精度（accuracy）的話，最好的方法就是通過交叉驗證（cross-validation）對各個算法一個個地進行測試，進行比較，然後調整參數確保每個算法達到最優解，最後選擇最好的一個。但是如果你只是在尋找一個「足夠好」的算法來解決你的問題，或者這裡有些技巧可以參考，下面來分析下各個算法的優缺點，基於算法的優缺點，更易於我們去選擇它。

偏差&方差

在統計學中，一個模型好壞，是根據偏差和方差來衡量的，所以我們先來普及一下偏差和方差：

偏差：描述的是預測值（估計值）的期望 E 與真實值 Y 之間的差距。偏差越大，越偏離真實數據。

$$\text{Bias}[\hat{f}(x)] = E[\hat{f}(x)] - f(x)$$

方差：描述的是預測值 P 的變化範圍，離散程度，是預測值的方差，也就是離其期望值 E 的距離。方差越大，數據的分布越分散。

$$\text{Var}[\hat{f}(x)] = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$$

模型的真實誤差是兩者之和，如下圖：

$$E[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2$$

如果是小訓練集，高偏差/低方差的分類器（例如，樸素貝葉斯 NB）要比低偏差/高方差大分類的優勢大（例如，KNN），因為後者會過擬合。但是，隨著你訓練集的增長，模型對於原數據的預測能力就越好，偏差就會降低，此時低偏差/高方差分類器就會漸漸的表現其優勢（因為它們有較低的漸近誤差），此時高偏差分類器此時已經不足以提供準確的模型了。

當然，你也可以認為這是生成模型（NB）與判別模型（KNN）的一個區別。

為什麼說樸素貝葉斯是高偏差低方差？

以下內容引自知乎：

首先，假設你知道訓練集和測試集的關係。簡單來講是我們要在訓練集上學習一個模型，然後拿到測試集去用，效果好不好要根據測試集的錯誤率來衡量。但很多時候，我們只能假設測試集和訓練集的是符合同一個數據分布的，但卻拿不到真正的測試數據。這時候怎麼在只看到訓練錯誤率的情況下，去衡量測試錯誤率呢？

由於訓練樣本很少（至少不足夠多），所以通過訓練集得到的模型，總不是真正正確的。（就算在訓練集上正確率 100%，也不能說明它刻畫了真實的數據分布，要知道刻畫真實的數據分布才是我們的目的，而不是只刻畫訓練集的有限的數據點）。而且，實際中，訓練樣本往往還有一定的噪音誤差，所以如果太追求在訓練集上的完美而採用一個很複雜的模型，會使得模型把訓練

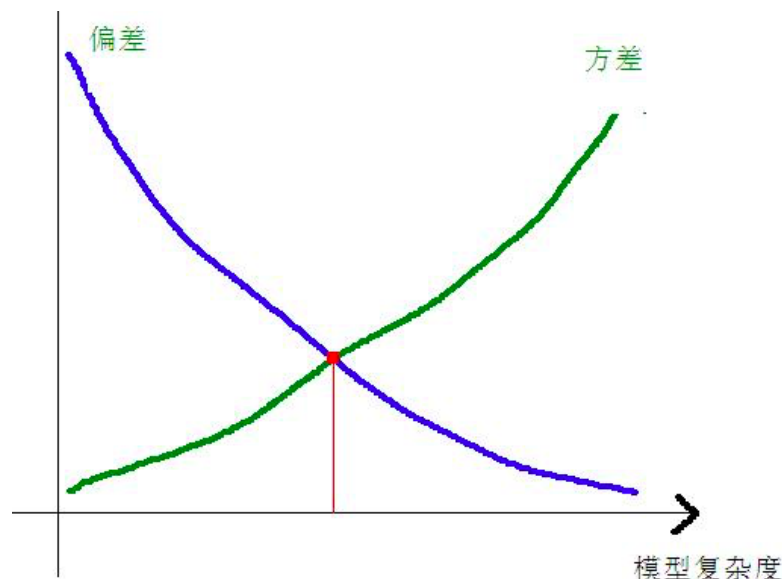
集裡面的誤差都當成了真實的數據分布特徵，從而得到錯誤的數據分布估計。這樣的話，到了真正的測試集上就錯的一塌糊塗了（這種現象叫過擬合）。但是也不能用太簡單的模型，否則在數據分布比較複雜的時候，模型就不足以刻畫數據分布了（體現為連在訓練集上的錯誤率都很高，這種現象較欠擬合）。過擬合表明採用的模型比真實的數據分布更複雜，而欠擬合表示採用的模型比真實的數據分布要簡單。

在統計學習框架下，大家刻畫模型複雜度的時候，有這麼個觀點，認為 $\text{Error} = \text{Bias} + \text{Variance}$ 。這裡的 Error 大概可以理解為模型的預測錯誤率，是有兩部分組成的，一部分是由於模型太簡單而帶來的估計不準確的部分（Bias），另一部分是由於模型太複雜而帶來的更大的變化空間和不確定性（Variance）。

所以，這樣就容易分析樸素貝葉斯了。它簡單的假設了各個數據之間是無關的，是一個被嚴重簡化了的模型。所以，對於這樣一個簡單模型，大部分場合都會 Bias 部分大於 Variance 部分，也就是說高偏差而低方差。

在實際中，為了讓 Error 儘量小，我們在選擇模型的時候需要平衡 Bias 和 Variance 所占的比例，也就是平衡 over-fitting 和 under-fitting。

偏差和方差與模型複雜度的關係使用下圖更加明了：



當模型複雜度上升的時候，偏差會逐漸變小，而方差會逐漸變大。

25.4 SVM

<https://www.learnopencv.com/svm-using-scikit-learn-in-python/>

25.5 演算法 SGD (Stochastic Gradient Descent)

隨着應用資料的增長，在大規模資料集上進行統計解析和機器學習越來越成為一個巨大的挑戰。目前，適用於統計解析/機器學習的語言/程式庫有很多，如專為資料解析用途而設計的 R 語言，Python 語言的機器學習程式庫 Scikits，支援分散式環境延伸的有基於 Map-Reduce 實現的 Mahout，以及分散式記憶體計算框架 Spark 上的機器學習程式庫 MLlib 等等。目前 Spark 框架也推出了 R 語言的介面 SprakR。但是，本文要討論的，則是另外一種設計思路，在 database 中實現統計解析和機器學習算法，即 In-Database Analysis，Madlib 程式庫就是這種設計思路的代表。

把機器學習程式庫內建到 database 中（通過 database 的 UDF）有許多優點，執行機器學習算法時只需要編寫相應的 SQL 敘述就可以了，同時 database 本身作為解析的資料源，使用非常方便，大大降低了機器學習的應用門檻。當然缺點也是明顯的，由於受限於 database 提供的 UDF 程式介面，實現算法時會受到很多限制，很多優化難以實現，而大規模資料集上的機器學習，尤其是需要迭代計算的，通常對算法效能和結果收斂速度要求較高，否則很難做到實用。本文的重點就是討論如何在 In-database Analysis 的框架下，高效的實現機器學習中的 SGD（隨機梯度下降）算法。由於很多機器學習算法如 linear SVM 分類器、K-mean、[Logistic Regression](#) 都可以採用 SGD 算法來實現，只需要針對不同算法設計不同的目標函數即可。因此在 database 上實現高效能的 SGD 算法框架，便可用來執行一大類機器學習算法。

以 Madlib 為例，如果要 Madlib 用 SVM 算法對資料集做訓練，可以執行如下 SQL 敘述：

```
SELECT madlib.lsvm_classification( 'my_schema.my_train_data',
                                   'myexpc',
                                   false);
```

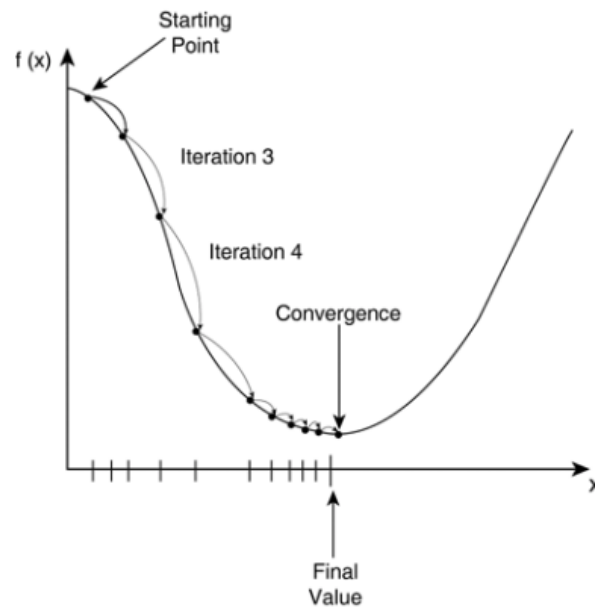
`madlib.lsvm_classification` 是 Madlib 中實現的 SVM 計算函數，上述方式的呼叫則是採用 SGD 算法進行 linear SVM 分類，其中 `my_schema.my_train_data` 是訓練資料表，必須滿足如下架構定義：

```
TABLE/VIEW my_schema.my_train_table
(
  id      INT,          -- point ID
  ind     FLOAT8[],     -- data point
  label   FLOAT8        -- label of data point, 即分類結果
);
```

25.6 演算法教學

超棒的

http://scikit-learn.org/stable/tutorial/statistical_inference/supervised_learning.html



但是隨著要分析的對象越來越複雜，可用變項指數成長，建模的時間也呈倍數縮短的情況下，還是要回到分析的根本--也就是數學模型來尋找突破的方法，所以開始認真的研讀演算法，這也是給予自己未來的目標，希望更深入的去了解常用的演算法。

不過因為數學仍然不是我的專長，所以我只能以解釋性的方式來說明對於每個算式的理解，無法太深入去推倒公式。

今天 **Stochastic gradient descent**（簡稱 **SGD**），中文可以稱為梯度下降法、線性回歸方程式、數學推倒法，主要是用來尋找迴歸模型的係數。當初會開始注意這個演算法，在於相較於統計上的迴歸需要一次性的將所有資料一起放入分析（又稱之為 **Batch Mode**），**SGD** 演算法可以單筆單筆資料放入模型中推算（又稱之為 **Online Mode**），雖然精準度不若一次性的算法，但是卻可以即時地根據 **input** 的資料動態的調整推估 **model**。

Batch Model 和 **Online Model** 的用途以及差異其實也是和學科本身性質串在一起的。統計資料往往就是一次性的搜集然後一起分析，推算母體。但是一些物理學的資料中（**SGD** 演算法最早提出來是在物理領域，**S. Amari Natural Gradient Works Efficiently in Learning, 1988**），資料（例如聲波）是不斷的發送出來的，而且還會因為環境的改變而產生變化，所以才需要這種可以動態調整的分析 **model**。

透過回歸方程式，來看一下這種驗算法的想法以及原理（公式參考史丹佛的上課講義）：

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

上面一條一般的回歸方程式，下面是簡化（將常數項 * $x_0=1$ ）（這一步其實不太重要）

$$h(x) = \sum_{i=0}^n \theta_i x_i = \theta^T x,$$

重點在於以下的 **Cost Function**:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2.$$

先不要被公式嚇到，概念很簡單。雖然長得不一樣，但是核心原理就是最小平方方法 -- 讓預測值及觀察值的差異平方最小。 θ 代表了迴歸係數（單一個係數）， $h(x(i))$ 為在這個迴歸係數下的預測值， $y(i)$ 為觀察值。不同的迴歸係數會得到不同的 $h(x(i))$ ，我們的目的就是要尋找最合適的迴歸係數，讓這整個方程式得到最小值。

Cost Function，再複習一下他的樣子：

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2.$$

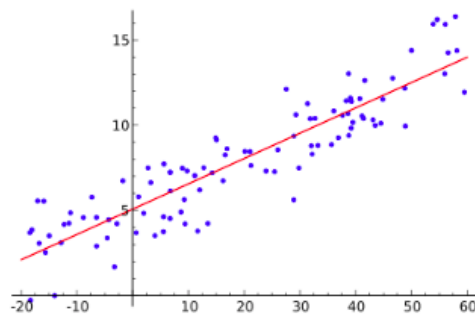
θ 為迴歸係數，要求得 θ 的最小值，也就是求上述 **Cost Function** 的最小值，統計上的做法就是直接將上述式子對每個係數做偏微分=0 就可以計算出 θ 。這個方法完全沒錯，但是今天我要介紹的 **SGD** 是另外一種概念。

如果說直接讓 **Cost Function** 微分=0 來求得最小值是一種站在宏觀的做法，**SGD**(其實嚴格來說目前還只是 **Gradient Descent**)，是站在微觀的觀點，讓 θ 值根據當下的狀況（也就是對迴歸方程式最小平方和的影響），動態的調整 θ 的大小，來求得最小值。

讓我們以圖形化的方式來了解這個過程。

線性回歸方程式

$$h(x) = \sum_{i=0}^n \theta_i x_i = \theta^T x$$



Cost Function

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2.$$



迴歸方程式是一次方程式，所以是一條線（如果是多元迴歸請自行想像成某個變項的切面謝謝 orz）

Cost Function 是計算最小平方和，是自變項的二次方程式，所以是個曲線（如果是多元迴歸就是一個高維度的山谷，我無法想像，不過就每個切面來說還是長得像山谷。）

那我們今天的做法就是將問題從線性方程式轉換成求最小平方和的最佳解。要求最佳解我們用的做法就是，先隨便站在山谷上的任何一點，發現哪邊比較低，就往哪邊走（如右圖）。透過多次的迭代來找到最低點，化成公式就長得像下面這樣：

直到收斂 `while{`

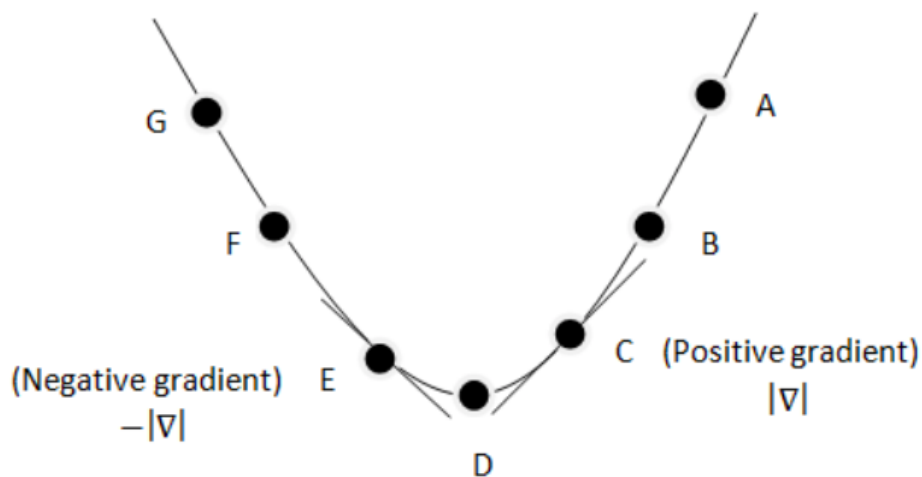
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta).$$

`}`

這個公式中的： $=$ 不是一般數學上的 $=$ （**equal**），而是一般程式語言中“ $=$ ”的用法或是 **R** 中的 `<-` (**assign**)，將右邊的值指派給左邊的意思。整個公式比起數學，更像程式語言，使用一個 **while loop** 來控制程式流程，反覆迭代運算 θ 值來得到最佳解：

公式中左邊的 θ 代表了 $n+1$ 次時的值，右邊的 θ 是第 n 次的值。那每次 θ 要加減多少就是以一個 α （又稱 **learning rate**）以及 $J(\theta)$ 在 θ 點的微分來控制。先不管 α ，先來看微分這件事情。一個二次方程式在不同自變項的偏微分在圖形上就代表了這一點的斜率。

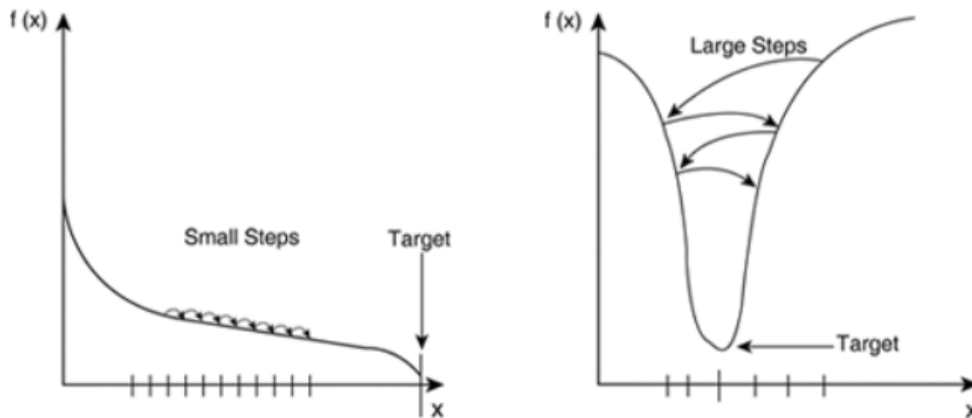
（見下圖，圖片來源：<http://stackoverflow.com/questions/21064030/gradient-descent-in-linear-regression>）



當 θ 落在最小值的右邊，斜率是正的，帶到公式中，就會減少 θ 值；反之，如果 θ 比最小值還小，那斜率是負的，帶到迭代公式中就會增加 θ 值。由於曲線越接近谷底越為平緩，所以當 θ 靠近最低點時，每次移動的距離也會越來越小，最終將收斂在最低點。

講完微分的部分，接下來是 α 值（**learning rate**）。 α 值的用途主要在控制每次 θ 移動的距離，下圖顯示移動距離太大或太小的狀況：

圖片來源：http://www.yaldex.com/game-development/1592730043_ch18lev1sec4.html



如果一次移動太短的距離（左圖），那收斂速度會相當的慢。但是如果移動的距離太大（右圖），反而可能造成無法收斂的情形。所以會需要根據不同的狀況來調整這個 α 的大小，這部分也有相當多的討論和證明，就請另外參考其他相關討論。概念是這樣，接著我們將微分後的結果展開：

$$\begin{aligned}
 \frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_\theta(x) - y)^2 \\
 &= 2 \cdot \frac{1}{2} (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x) - y) \\
 &= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^n \theta_i x_i - y \right) \\
 &= (h_\theta(x) - y) x_j
 \end{aligned}$$

如果有是多元迴歸，就是根據個別的係數做偏微分，最後結果如下（如果是常數項就把最右邊那個 x 丟掉就好）：

$$\theta_j := \theta_j + \alpha (y^{(i)} - h_\theta(x^{(i)})) x_j^{(i)}.$$

所以整理起來，為了要得到迴歸方程式的最佳解，我們會透過下述方式：

Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^m (y^{(i)} - h_\theta(x^{(i)})) x_j^{(i)} \quad (\text{for every } j).$$

}

這邊的 x 和 y 就是一次帶入所有資料進去做運算，這邊到目前為止，我們已經將整個 **Gradient Descent** 的推倒介紹完，但是這樣的方法在運算上會有個缺點：就是每次的迭代都必須將所有變項丟進去（又稱之為 **Batch Mode** 的演算法），會大幅增加運算的時

間，所以下一篇我們將介紹一次丟一筆資料進去計算的方法，並且都過這樣的方法，讓我們的演算法從 **Batch Mode** 轉換成 **Online Mode**，也才有可能對 **Streaming** 資料做機器學習。

25.7 Stochastic Gradient Descent 的 Python 的程式

資料來源：<http://scikit-learn.org/stable/modules/sgd.html>

範例程式 SGD-1.py

```
1. print(__doc__)
2.
3. import numpy as np
4. import matplotlib.pyplot as plt
5. from sklearn.linear_model import SGDClassifier
6. from sklearn.datasets.samples_generator import make_blobs
7.
8. # we create 50 separable points
9. X, Y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.60)
10.
11. # fit the model
12. clf = SGDClassifier(loss="hinge", alpha=0.01, n_iter=200, fit_intercept=True)
13. clf.fit(X, Y)
14.
15. # plot the line, the points, and the nearest vectors to the plane
16. xx = np.linspace(-1, 5, 10)
17. yy = np.linspace(-1, 5, 10)
18.
19. X1, X2 = np.meshgrid(xx, yy)
20. Z = np.empty(X1.shape)
21. for (i, j), val in np.ndenumerate(X1):
22.     x1 = val
23.     x2 = X2[i, j]
24.     p = clf.decision_function([[x1, x2]])
25.     Z[i, j] = p[0]
26. levels = [-1.0, 0.0, 1.0]
27. linestyles = ['dashed', 'solid', 'dashed']
28. colors = 'k'
```

```
29. plt.contour(X1, X2, Z, levels, colors=colors, linestyle=linestyle)
30. plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.Paired)
31.
32. plt.axis('tight')
33. plt.show()
```

