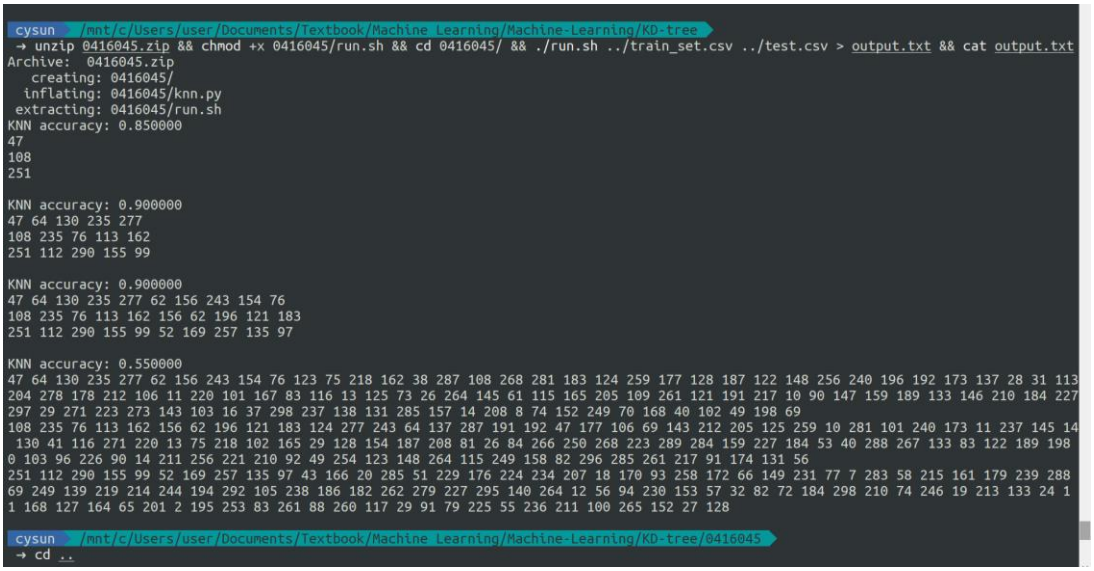
Machine Learning - assignment #2

0416045 孫嘉妤

程式運行結果

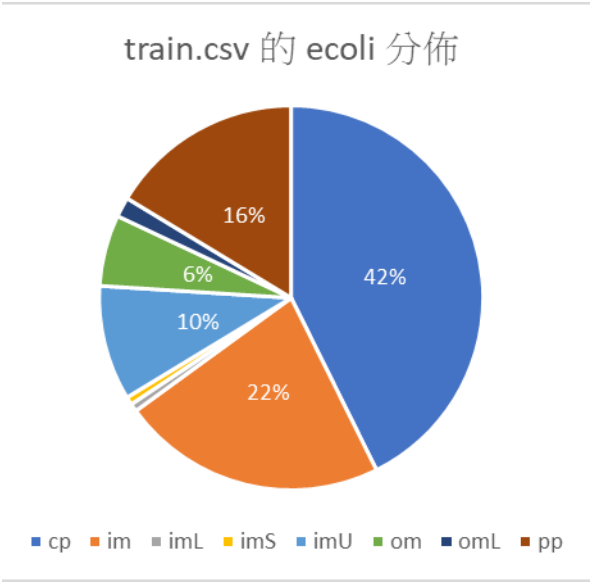
這次作業用 300 筆 ecoli 資料為 training data 建立 K-NN classifier 預測 testing data，並以 Kd-tree 提高效率。下圖為在我自己電腦執行時的截圖。



經過數次隨機從 train.csv 產生 test.csv 測試，我觀察到的結果概述如下：

- K = 1
  - 有一定程度的 accuracy，大概都有 0.7~0.8 甚至 0.9 以上，但視測資偶爾也會落到 0.6 左右。
- K = 5 和 K = 10
  - 相較 K = 1，accuracy 略有高低，不過基本上 K = 5 和 K = 10 的表現大部份準確於 K = 1。
  - 此外，整體來看 K = 5 的 accuracy 似乎有略高於 K = 10。
- K = 100
  - accuracy 為四種 K 中最低，明顯低於前三者

本次的 training set 和 testing set 數量都不多，因此選取到的 testing set 會大幅影響 accuracy。並且 training data 中各類別的 ecoli 數量差距略大。下面討論這樣的資料如何影響 K 的結果：



由 training data 的 ecoli 分佈可看出 cp、im 和 pp 佔了大部分，而 imS 與 imL 極為稀少。因此，若取 K = 100，在 K 已達 training data 三分之一的情況下，預測結果想必為數量最大者；而 imS, imL 過於稀缺，即使 K 僅取 5 預測結果也永遠不會為這兩者。一般而言 KNN K>1 可能還是會較 K=1 稍準一些，而由於有數個個數遠大於或少於他者的 class，我認為這是造成上述 accuracy K(5) >= K(10) >> K(100) 的原因。

此外，我發現我寫的程式預測相同資料的 accuracy 有時會有變動，這是因為 k 個最近鄰中可能會有數量相等的類別：例如離 target 最近的 5 個點 class 分別是（pp, pp, cp, cp, imS），那預測結果可能是 pp 或 cp。為了讓預測結果一致，若數量最多的 class 有兩個，則回傳 class 名字典序較大者。

程式語言

python3

從讀取 csv、kd-tree 建立搜索到 K-NN，整份作業皆使用 python3。我自己的執行環境是 3.5.2 及系上工作站的 3.6.2。

使用程式庫

code 中使用的所有 header 如下：

```
import csv
import math
import operator // sorting 用
import sys
from heapq import * // k-NN 搜尋時用 heap 儲存目前找到的 neighbor
```

開發環境

程式撰寫主要使用文字編輯器 Atom，並在 windows 10 運行（Windows Subsystem for Linux）。程式部份完成後也有在系上工作站測試，以上兩者的執行結果並無明顯差異。

code explanation **(※為了易讀較實際 code 稍有刪略改動)**

建立 Kd-Tree

```
def build_KD_tree(train):

    # 分別計算各 attribute 的 variance，選出 variance 最大的 attribute
      (略，variance(), median() 等的 code 會附在後面)

    # set the mid of the dimension as pivot，建立一個新的 tree node
    pivot = median(select_attr)
    new_node = Node()
    new_node.pivot = pivot
    new_node.dim = dim

    # 用 pivot 把 data 分成兩份
    for i in range(len(train)):
        if train[i].attr[dim] <= pivot:
            left_train.append(train[i])
        else:
            right_train.append(train[i])

    # 如果左右其中一份個數為 0，代表 data 有多個 split attribute 值相等因此
      無法用 <= pivot 區分。此時將 < pivot 作為 left，>= pivot 為 right
      (略)

    # 當 data = 1 時，建立 leaf 並 return，否則繼續建立 node
    if len(left_train) == 1:
        new_leaf = Leaf()
        new_leaf.ecoli = left_train[0]
        new_node.leftChild = new_leaf
    else:
        new_node.leftChild = build_KD_tree(left_train)
    new_node.leftChild.parent = new_node
    new_node.leftChild.childType = 'leftChild'

    (right 略)

    return new_node
```

計算用 function (mean / variance / median / Euclidean distance)

```
def mean(seq):
    sum = 0;
    for x in range(len(seq)):
        sum += seq[x]
    return sum / len(seq)

def variance(seq):
    m = mean(seq)
    var_sum = 0
    for i in range(len(seq)):
        diff = seq[i] - m
        var_sum += diff*diff
    return var_sum / len(seq)

def median(seq):
    seq.sort()
    size = len(seq)
    if size % 2 == 0:
        m = (seq[size//2] + seq[size//2-1])/2
    if size % 2 == 1:
        m = seq[(size-1)//2]
    return m

def euclidean_distance(p1, p2):
    e_sum = 0
    for i in range(len(p1.attr)):
        diff = p1.attr[i] - p2.attr[i]
        e_sum += diff*diff
    return(math.sqrt(e_sum))
```

尋找 nearest neighbor

```
def nearest_neighbor(root, target):
    # 用 path 記錄走過的 node
    while cur_node.__class__.__name__ == 'Node':
        path.append(cur_node)
        if target.attr[cur_node.dim] <= cur_node.pivot:
            cur_node = cur_node.getLeftChild()
        else:
            cur_node = cur_node.getRightChild()
```

```
    min_dist = euclidean_distance(cur_node.ecoli, target)
    nearest = leaf.ecoli
    # 比較所有經過的 node，比較 target 相距 min_dist 是否可能與其
      子集相交，若是則進入其另一側 subtree 檢查是否有更近的 neighbor
    while len(path) != 0:
        back = path.pop()
        b_node = target
        b_node.attr[back.dim] = back.pivot # Hyperplane
        b_dist = euclidean_distance(b_node, target)
        if b_dist <= min_dist:
            if target.attr[back.dim] > back.pivot:
                cur_node = back.getLeftChild()
            else:
                cur_node = back.getRightChild()
            min_ecoli = traversal(cur_node, target)
            n_dist = euclidean_distance(min_ecoli, target)
            if n_dist < min_dist:
                min_dist = n_dist
                nearest = min_ecoli

    return nearest
```

k-NN heap（回傳從近到遠的 k 個 nearest neighbor）

這份作業中使用 heap 來實作 k-NN：建立一個儲存目前 neighbor 的 heap，從上述 query_tree() 到達的 leaf 開始，當 neighbor 數 < k 時將回溯節點的另一側子節點加入 heap。由於 heap 的特性，節點一加入便已完成排序，並能快速拿到最大最小值。當 neighbor 數 > k，則 pop 多餘節點（每次 pop 的便是 heap 中距離 target 最遠的 neighbor）並繼續回溯。當 target 相距 min_dist 可能與回溯節點的另一側子集相交，進入 path 的另一側並加入其子集內的 leaf，如此重複 push 新 k-NN 進 heap 並 pop 多餘節點。

```
def kNN_heap(root, target, k):
    # 先走到 leaf，並記錄在 kd-tree 上的 path
    (略，同上述找尋最近鄰的前段部份)
    # 將抵達 leaf 加入 heap (省略初始化)
    new_nbr = NN()
    min_dist = euclidean_distance(leaf.ecoli, target)
    heappush(heap, new_nbr)
```

```
    while len(path) != 0:
        back = path.pop()
        b_node.attr[back.dim] = back.pivot # hyperPlain
        b_dist = euclidean_distance(b_node, target)
        if b_dist <= heap[0].dist or len(heap) < k:
            if target.attr[back.dim] > back.pivot:
                heappush(back.getLeftChild(), target)
            else:
                heappush(back.getRightChild(), target)
            # remove the remain leaves
            while len(heap) > k:
                heappop(heap)

    return heap
```

k-NN classify（由 k-NN heap 回傳的 heap predict 類別）

```
def kNN_classify(heap):

    class_num = {'cp': 0, 'im': 0, 'pp': 0, 'imU': 0, 'om': 0,
                 'omL': 0, 'imL': 0, 'imS': 0}

    for e in heap:
        class_num[e.ecoli.ecoliClass] += 1

    # 1 為各 class 個數，0 為 class name
    sorted_class = sorted(class_num.items(),
                          key=operator.itemgetter(1, 0))

    sorted_class.reverse()

    # 回傳 k 個最近鄰中出現最多次的 class
    return sorted_class[0][0]
```