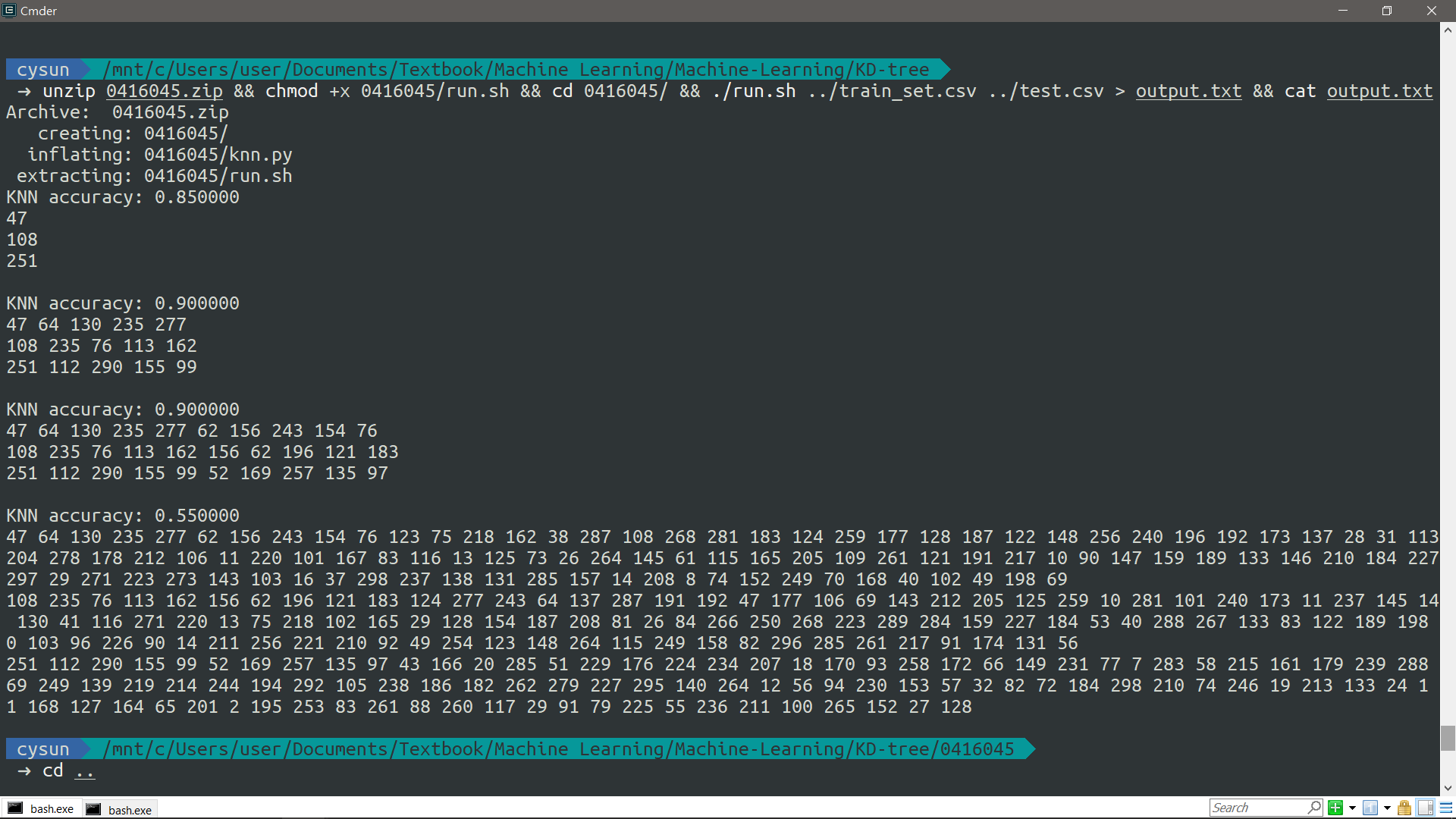
**Machine Learning - assignment #2**

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] |