人工智慧概論

0613413 蔡怡君

－實驗目的：自建Decision Tree，並且使用random forest去預測，CART為base tree。

－資料來源：UCI Machine Learning Repository－wine dataset

－實驗方法：

最一開始要將資料切割成80% training用、20% validation用，因為要random forest每一顆Tree都需要長的不同，( tree bagging )所以取80%中的training = 64%的原始data去自建class CART model，( attribute bagging )透過每一層給的attribute(自訂原始6個)，看每一個attribute去跑threshold找到最佳的gini index，再去挑選最好的attribute透過已經計算好的gini index，跑的同時要去限制minimum number 跟Tree depth。

Random forest中的50顆CART，在這裡取0.8\*50 = 40 去predict，而0.2\*50 = 10顆為out of bag (OOB)，跑predict的時候決定最後的class分類是由40顆去投票投出來的結果。

－結果觀察：

在實驗的過程都是使用correct classification rate為y軸。實驗結果為跑100次的平均。

**實驗變因：**

1. Size of Training and Validation
2. number of trees
3. how many attributes to consider at each node splitting. (當attribute = 1 為特殊情形)
4. 樹的max depth、樹的minimum number

原始資料：

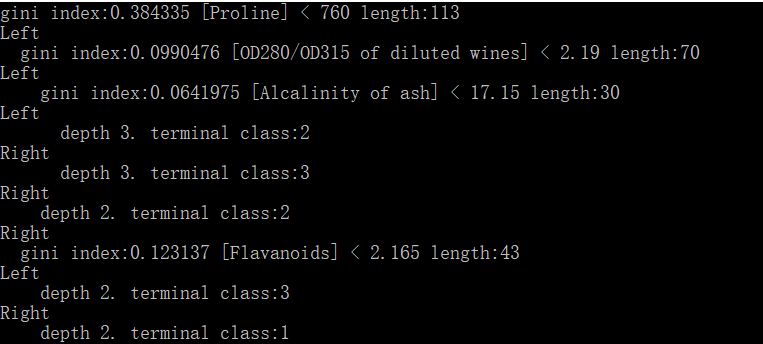
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Percent of Training | Number of tree | Attribute(node split) | Max depth | Minimum number |
| 0.8 | 50 | 6 | 5 | 5 |

1. Size of Training and Validation 與 correct classification rates的關係：

變因：train在整體的percentage。

1. Number of trees in the forest與correct classification rates的關係：
2. Number of attributes to consider at each node splitting與correct classification rates的關係：
3. 樹的max depth：

在檢查的過程發現其實跑到depth 3或是4的時候，樹就已經完整建立好了。



1. 樹的minimum
2. Extremely random forest中attribute為random給的，去比較跟random forest中的tree數量成長有沒有影響到correct rate：

－結果分析：

1. 先講一下這次測試的dataset－wine，可以從測試的時候去得知說是還蠻容易就學習的dataset，連只給30%data去train就可以有80%以上的成功率了，還有在建樹的過程中，只需要大約3、4層就可以建好了。
2. Train在整體data占比例占的越少，正確率越低。但是這裡蠻訝異的一點就是不會有比例較大(0.9)，導致validation data overfitting的問題，我猜想應該是因為dataset太簡單太好預測了。
3. Number of tree in random forest在5~20顆時，正確度成長率很快，差不多在近20顆時到達平衡，樹的多寡不會導致overfitting反而是正確率增加、錯誤率變小。
4. Number of attribute in node splitting這邊當attribute = 1時為extreme random forest tree特殊情形，可以看到attribute數量越多正確率越低，是因為Attribute給的越多，相對而言樹的diversity就越低，因為每一顆樹在選的時候都會選擇最好的attribute去長，所以使得最後投票的時候偏好類似，導致正確率下降。
5. 樹的tree depth，正確率主要在1~3階段成長性增加而後穩定，因為在後圖可以看出其實因為這個dataset蠻簡單的，所以在建tree時主要在3、4就已經完成了。
6. Minimum number of a node可以看出沒有太大的幅度，主要應該是因為dataset本身的資料class本來差異就蠻大。
7. Extremely random forest tree可以看到大約正確率在~20顆tree的時候大幅度的增加，就可以把random給attribute的正確率給恢復像attribute 6個時的正常情況，在只有2~5顆的時候可以看出正確率相較於20顆蠻低的。

－心得：

這次自建random forest + base tree CART的實驗讓我收穫蠻多的，以前我有寫過python直接使用套建的結果，但那時候常常定義、參數還是會有點模糊，不是很清楚裡面套件裏頭主要是在做甚麼，透過這次的實驗，可以很清楚的知道每個步驟在做甚麼，而且在寫之前原本覺得應該會困難重重，結果還行，而且預測出來的模型也蠻佳的，所以還蠻開心的，這些概念對我未來使用套件會對於裏頭的參數有更多的了解，希望也可以預測模型更佳。

－Reference：只有讀檔string處理有使用到source code。

1. #include <iostream>
2. #include <string>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <sstream>
6. #include <fstream>
7. #include <time.h> // time
8. #include <math.h>
9. #include <algorithm> // std::random\_shuffle
10. #include <vector>
11. #define N 178
12. #define type 13
14. **using** **namespace** std;
16. **float** data[N][type+1];
17. **float** train[N][type+1];
18. **float** valid[N][type+1];
20. **int** TN,VN;
22. string printAttr[] = {"class",
23. "Alcohol",
24. "Malic acid",
25. "Ash",
26. "Alcalinity of ash",
27. "Magnesium",
28. "Total phenols",
29. "Flavanoids",
30. "Nonflavanoid phenols",
31. "Proanthocyanins",
32. "Color intensity",
33. "Hue",
34. "OD280/OD315 of diluted wines",
35. "Proline"};
37. **void** print\_data(**float** input[][type+1], **int** n){
38. **for** (**int** j = 0 ; j < n ; j ++) {
39. **for** ( **int** i = 0 ; i < type+1 ; i ++)
40. cout << input[j][i] <<",";
41. cout << endl;
42. }
43. }
44. vector<string> \_csv(string);
45. **void** load\_data(string);
47. // split data to training / validation
48. **void** split\_data(**float** p){
49. **int** n = N \* p;
50. TN = n;
51. VN = N - TN;
52. //cout << "train N:" << FN << endl;
53. //cout << "Valid N:" << VN << endl;
54. random\_shuffle(data,data+N);
56. **for** (**int** i = 0 ; i < TN ; i ++)
57. **for** (**int** j = 0 ; j < type+1 ; j ++)
58. train[i][j] = data[i][j];
60. **int** Vindex = 0;
61. **for** (**int** i = TN ; i < N ; i ++) {
62. **for** ( **int** j = 0 ; j < type+1 ; j ++)
63. valid[Vindex][j] = data[i][j];
64. Vindex++;
65. }
66. }
68. **float** gini\_index(**float** input[][type+1],**int** n){
69. **int** count[3]={0}; // 分別計算class 1,2,3
70. **float** gini = 0;
71. **for**( **int** i = 0 ; i < n ; i ++) {
72. count[ **int**(input[i][0]-1)] ++;
73. }
74. **for** (**int** i = 0 ; i < 3 ; i ++) {
75. gini += pow(count[i]\*1.0/n,2);
76. }
77. gini = 1.0 - gini;
78. **return** gini;
79. }
81. **struct** split\_info{
82. **int** attribute;
83. **float** threshold;
84. **float** gini;
85. split\_info(**void**){
86. gini = 1.0;
87. }
88. };
90. **class** CART{
91. **private**:
92. **float** data[N][type+1];
93. **int** length;
94. **int** num\_attr; // num of random attribute
95. **public**:
96. CART \*Left;
97. CART \*Right;
98. **bool** is\_terminal;
99. **int** end\_class;
100. **struct** split\_info result; // save best attr, threshold, gini
102. CART(**float**[][type+1],**int** l); // constructor
103. **int** get\_length();
104. **void** set\_num\_attr(**int**); // set num\_attr
105. **float** test\_split(**int**,**float**);
106. **struct** split\_info select\_threshold(**int**);  // find best value (lowest gini) according to threshold
107. **void** find\_best\_attr(); // select the threshold with the lowest total impurity
108. **void** get\_split();
109. **bool** same\_class();
110. **int** to\_terminal();
111. };
113. CART::CART(**float** input[][type+1],**int** l) {
114. **for** (**int** i = 0 ; i < l ; i ++) {
115. **for**(**int** j = 0 ; j < type+1 ; j ++)
116. data[i][j] = input[i][j];
117. }
118. length = l;
119. is\_terminal = **false**;
120. }
122. **int** CART::get\_length() {
123. **return** length;
124. }
126. **float** CART::test\_split(**int** attr,**float** threshold){
127. **float** left[N][type+1];
128. **float** right[N][type+1];
129. **int** Lindex = 0;
130. **int** Rindex = 0;
131. **for** (**int** i = 0 ; i < length ; i ++) {
132. **if** (data[i][attr] < threshold) {
133. **for** ( **int** j = 0 ; j < type +1 ; j ++)
134. left[Lindex][j] = data[i][j];
135. Lindex ++;
136. }
137. **else** {
138. **for** (**int** j = 0 ; j < type+1 ; j ++)
139. right[Rindex][j] = data[i][j];
140. Rindex ++;
141. }
142. }
143. **float** gini = Lindex\*1.0/length\*gini\_index(left,Lindex) + Rindex\*1.0/length\*gini\_index(right,Rindex);
144. **return** gini;
145. }
147. **void** CART::set\_num\_attr(**int** n) {
148. num\_attr = n;
149. }
151. // if threshold = 1 代表 Alcohol
152. **struct** split\_info CART::select\_threshold(**int** attr){
153. // th 存放這 attribute 所有值
154. **struct** split\_info best;
155. best.gini = 1.0;
156. vector<**float**>th;
157. **for** (**int** i = 0 ; i < length ; i ++)
158. th.push\_back(data[i][attr]);
159. sort(th.begin(),th.begin()+length);
161. **for** (**int** i = 0 ; i < length-1 ; i ++) {
162. **float** threshold = (th[i] + th[i+1]) / 2.0;
163. **float** resultgini = test\_split(attr,threshold);
164. **if**(resultgini < best.gini){
165. best.threshold = threshold;
166. best.gini = resultgini;
167. }
168. }
169. best.attribute = attr;
170. **return** best;
171. }
173. //select attribute from attr. bag
174. **void** CART::find\_best\_attr(){
175. **int** select [type+1];
176. **for** (**int** i = 0 ; i < type; i ++)
177. select[i] = i+1;
178. random\_shuffle(select,select+type);
180. **for** (**int** i = 0 ; i < num\_attr ; i ++) {
181. **struct** split\_info tmp;
182. //select threshold of the attribute
183. tmp = select\_threshold(select[i]);
184. //cout<<"attribute:"<<i<<endl;
185. **if** (tmp.gini < result.gini) {
186. result = tmp;
187. }
188. }
189. }
191. // check whether data in this class are the same class;
192. **bool** CART::same\_class() {
193. **int** first = data[0][0];
194. **for** (**int** i = 0 ; i < length ; i ++) {
195. **if** (data[i][0] != first)
196. **return** **false**;
197. }
198. is\_terminal = **true**;
199. end\_class = first;
200. **return** **true**;
201. }
203. // find the best attribute -> produce left & rihgt CART
204. **void** CART::get\_split(){
205. set\_num\_attr(6);
206. find\_best\_attr();
207. **float** left[N][type+1];
208. **float** right[N][type+1];
209. **int** Lindex = 0;
210. **int** Rindex = 0;
211. **for** (**int** i = 0 ; i < length ; i ++) {
212. **if** (data[i][result.attribute] < result.threshold) {
213. **for** ( **int** j = 0 ; j < type +1 ; j ++)
214. left[Lindex][j] = data[i][j];
215. Lindex ++;
216. }
217. **else** {
218. **for** (**int** j = 0 ; j < type+1 ; j ++)
219. right[Rindex][j] = data[i][j];
220. Rindex ++;
221. }
222. }
223. Left = **new** CART(left,Lindex);
224. Right = **new** CART(right,Rindex);
225. }

228. **int** CART::to\_terminal() {
229. **int** count[3] = {0}; // calculate type
230. **for** (**int** i = 0 ; i < length ; i ++) {
231. count[ **int**(data[i][0]-1) ] ++;
232. }
233. **int** maxx = count[0];
234. **int** ctype = 1;
235. **for** (**int** i = 1 ; i < 3 ; i ++) {
236. **if**(count[i] > maxx) {
237. ctype = i+1;
238. maxx = count[i];
239. }
240. }
241. is\_terminal = **true**;
242. end\_class = ctype;
243. **return** ctype;
244. }
246. **void** split(CART \*r, **int** max\_depth, **int** min\_size , **int** depth) {
247. //cout<<" Root split "<<endl;
248. r->get\_split();
249. // check for a no split
250. **if**(r->result.gini == 0) {
251. r->to\_terminal();
252. **return**;
253. }
254. // check for max depth
255. **if** (depth >= max\_depth) {
256. r->Left->to\_terminal();
257. r->Right->to\_terminal();
258. **return**;
259. }
260. // process left child
261. **if** (r->Left->get\_length() <= min\_size)
262. r->Left->to\_terminal();
263. **else** {
264. r->Left->get\_split();
265. split(r->Left, max\_depth, min\_size, depth+1);
266. }
267. // process Right child
268. **if** (r->Right->get\_length() <= min\_size)
269. r->Right->to\_terminal();
270. **else** {
271. r->Right->get\_split();
272. split(r->Right, max\_depth, min\_size, depth+1);
273. }
274. }
276. **void** print\_tree(CART \*root,**int** depth){
277. **if** ( root->is\_terminal == **false**) {
278. **for**(**int** i = 0 ; i < depth ; i ++)
279. cout<<"  ";
280. cout<<"gini index:"<< root->result.gini;
281. cout <<" ["<< printAttr[root->result.attribute]<<"] < " << root->result.threshold << " length:" << root->get\_length() << endl;
282. cout << "Left" << endl;
283. print\_tree(root->Left,depth+1);
284. cout << "Right" << endl;
285. print\_tree(root->Right,depth+1);
286. }
287. **else**{
288. **for**(**int** i = 0 ; i < depth ; i ++)
289. cout<<"  ";
290. cout << "depth " << depth <<". ";
291. cout << "terminal class:" <<  root->end\_class << endl;
292. }
293. }
295. CART build\_tree(**float** TRAIN[][type+1],**int** l, **int** max\_depth, **int** min\_size) {
296. CART root(TRAIN,l);
297. split(&root,max\_depth,min\_size,0);
298. **return** root;
299. }
301. **int** predict(CART \*root, **float** input[type+1]) {
302. CART \*cursor = root;
303. **while** ( !cursor->is\_terminal ) {
304. // < go to left
305. **if**( input[cursor->result.attribute] < cursor->result.threshold)
306. cursor = cursor->Left;
307. **else**
308. cursor = cursor->Right;
309. }
310. **return** cursor->end\_class;
311. }
313. **void** print\_result(vector<CART> RF,**float** input[][type+1],**int** n) {
314. **int** success = 0;
315. **for**(**int** i = 0 ; i < n ; i ++) {
316. **int** classCount[3] = {0};
317. **for** (**int** j = 0 ; j < RF.size() ; j ++) {
318. CART current = RF[j];
319. classCount[ predict(¤t,input[i]) - 1 ] ++;
320. }
321. // select most count to be the answer
322. **int** maxx = 0;
323. **int** finalclass;
324. **for**(**int** a = 0 ; a < 3 ; a ++) {
325. **if**(classCount[a] > maxx ) {
326. maxx = classCount[a];
327. finalclass = a+1; // 1 ~ 3
328. }
329. }
330. **if**(input[i][0] == finalclass)
331. success++;
332. }
333. cout << "success rate:" << success\*1.0/n <<endl;
334. }
336. **int** main()
337. {
338. // loading data
339. load\_data("wine.data");
340. // divide dataset into training & validation
341. split\_data(0.7);
343. // build K tree
344. **int** tree\_num = 200;
345. vector<CART> random\_forest;
346. **for**(**int** i = 0 ; i < tree\_num ; i ++) {
347. // split train to 0.8
348. random\_shuffle(train,train+TN);
349. **int** IN = TN \* 0.8;
350. **float** in[N][type+1];
351. **for** (**int** a = 0 ; a < IN ; a ++ )
352. **for** (**int** b = 0 ; b < type+1 ; b ++)
353. in[a][b] = train[a][b];
354. // build each tree
355. CART tmp = build\_tree(in,IN,5,5);
356. random\_forest.push\_back(tmp);
357. }
358. // Tree bagging
359. **int** TB = 0.8\*random\_forest.size();
360. vector<CART> selected(random\_forest.begin(),random\_forest.begin()+TB);
361. vector<CART> OOB(random\_forest.begin()+TB,random\_forest.end());
363. // testing by using OOB err. or validation data set
364. print\_result(selected,train,TN);
365. print\_result(selected,valid,VN);
366. cout<<"------------------OOB-----------------"<<endl;
367. print\_result(OOB,train,TN);
368. print\_result(OOB,valid,VN);
370. //print\_tree(&root,0);
371. system("pause");
372. **return** 0;
373. }
375. vector<string> \_csv(string s){
376. vector<string> arr;
377. istringstream delim(s);
378. string token;
379. **int** c = 0;
380. **while** (getline(delim,token,',')) {
381. arr.push\_back(token);
382. c ++;
383. }
384. **return** arr;
385. }
387. **void** load\_data(string f) {
388. ifstream inFile("wine.data");
389. **if** (!inFile){
390. cout << "檔案無法開啟\n";
391. exit(1);
392. }
393. string line;
394. **int** index = 0;
395. **while** (getline(inFile,line)) {
396. vector<string> a = \_csv(line);
397. **for** ( **int** i = 0 ; i < a.size() ; i ++)
398. data[index][i] = atof(a[i].c\_str());
399. index ++;
400. }
401. }