

# ID3 과제 보고서

2020. 12. 20 (일) 5PM / 20181202 김수미

## 1. 코드 및 알고리즘 / 자료구조 설명

이번 과제에서는 Decision Tree 알고리즘 중 하나인 ID3 알고리즘을 코드로 작성해 보았다. ID3 알고리즘은 기계학습 중 지도학습(Supervised Learning)에 속한다. Decision Tree 는 Tree 자료구조를 사용해 가장 변별력이 좋은 feature 순으로 데이터를 나누어 label 을 예측하는 알고리즘이며, ID3 알고리즘은 변별력을 판단하는 기준으로 엔트로피(Entropy)라는 정보 이론 개념을 사용한다. 아래는 작성한 코드의 세부 설명이다.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #define log2(x) log(x)/log(2)
5  int data_num, feature_num, test_num;
6  int data[500][500]; int test[500][500];
7  int data_class[500]; int used_feature[500];
8  int nodes[200000] = { NULL };
```

먼저 코드 작성에 필요한 헤더파일들을 불러오고, 매크로와 전역변수들을 선언해준다. log2 매크로는 엔트로피 계산에 사용되는 연산이며, data\_num 과 feature\_num 은 학습 데이터의 전체 데이터 개수와 feature 개수를, test\_num 은 테스트 데이터의 전체 개수를 저장하는 변수이다. 두 2 차원 배열 data 와 test 는 각각 학습데이터셋과 테스트데이터셋의 내용을 저장하며, data\_class 는 학습데이터셋의 클래스 정보만을 저장한다. 일차원 배열 nodes 는 Decision Tree 의 각 노드를 생성할 때 조상 노드의 분류기준으로 사용된 feature 가 다시 채택되는 것을 방지하기 위해 매번 분류 기준 feature 번호를 선택할 때 마다 조상 노드에 사용된 feature 번호를 확인하기 위한 배열이다. Random access 는 Linked List 보다 일차원배열에서 훨씬 유리하기 때문에 사용했다.

```
10  // ----- 1. 트리를 구성하는 노드. 리프노드는 자식노드가 NULL -----
11  typedef struct treeNode *treePtr;
12  typedef struct treeNode {
13      int data;
14      treePtr leftchild;
15      treePtr rightchild;
16  } treeNode;
```

트리를 구성하는 노드 구조체이다. 정수 데이터 필드와 leftchild, rightchild 포인터 필드로 구성되어 있다.

```

18 // ----- 2. 트리에 새로운 노드를 추가하는 함수 -----
19 treePtr insert_node(treePtr node, int num) {
20     treePtr ptr = (treePtr)malloc(sizeof(*ptr));
21     ptr->data = num;
22     ptr->leftchild = ptr->rightchild = NULL;
23     if (node->leftchild == NULL) node->leftchild = ptr;
24     else node->rightchild = ptr;
25     return ptr;
26 }
27
28 // ----- 3. 모두 같은 class를 가지는지 check하는 함수 -----
29 int check_class(int data_idx[], int num_data) {
30     int yes = 0, no = 0;
31     for (int i = 0; i < num_data; i++) {
32         if (data_class[data_idx[i]] == 1) yes++;
33         else if (data_class[data_idx[i]] == 0) no++;
34     }
35     if (yes == 0) { return 0; }
36     else if (no == 0) { return 1; }
37     else { return 2; }
38 }

```

하나의 노드 다음에 새로운 노드를 추가하는 함수이다. 전체적으로 트리 구조를 구성할 것이기 때문에 트리를 점점 확장해 나가는 데에 사용되는 함수라고 할 수 있겠다.

입력으로 받은 데이터셋이 모두 같은 class 를 가지는지 확인하는 함수이다. 모두 같은 class 를 가지는 경우 분류를 중단하고 리프노드를 생성한다.

```

39 // ----- 4. 데이터 feature별 entropy 계산 함수 -----
40 float feature_entropy(int feature_class[], int data_idx[], int data_number) {
41     float yes, no, yes_entropy, no_entropy, result;
42     float yes_yes = 0, yes_no = 0;
43     float no_yes = 0, no_no = 0;
44
45     // 현재 feature에서 참/불참이면서 class가 yes/no
46     for (int i = 0; i < data_number; i++) {
47         if (feature_class[i] == 1 && data_class[data_idx[i]] == 1) yes_yes++;
48         else if (feature_class[i] == 1 && data_class[data_idx[i]] == 0) yes_no++;
49         else if (feature_class[i] == 0 && data_class[data_idx[i]] == 1) no_yes++;
50         else if (feature_class[i] == 0 && data_class[data_idx[i]] == 0) no_no++;
51     }
52     yes = yes_yes + yes_no;
53     no = no_yes + no_no;
54 }

```

```

54 // yes_entropy를 계산한다
55 if (yes_yes != 0 && yes_no != 0)
56     yes_entropy = (yes / (float)data_number) * ((yes_yes / yes) * (log2(yes / yes_yes)) + (yes_no / yes) * (log2(yes / yes_no)));
57 else if (yes_yes == 0 && yes_no != 0)
58     yes_entropy = (yes / (float)data_number) * ((yes_no / yes) * (log2(yes / yes_no)));
59 else if (yes_yes != 0 && yes_no == 0)
60     yes_entropy = (yes / (float)data_number) * ((yes_yes / yes) * (log2(yes / yes_yes)));
61 else if (yes == 0)
62     yes_entropy = 0;
63 // no_entropy를 계산한다
64 if (no_yes != 0 && no_no != 0)
65     no_entropy = (no / (float)data_number) * ((no_yes / no) * (log2(no / no_yes)) + (no_no / no) * (log2(no / no_no)));
66 else if (no_yes == 0 && no_no != 0)
67     no_entropy = (no / (float)data_number) * ((no_no / no) * (log2(no / no_no)));
68 else if (no_yes != 0 && no_no == 0)
69     no_entropy = (no / (float)data_number) * ((no_yes / no) * (log2(no / no_yes)));
70 else if (no == 0)
71     no_entropy = 0;
72
73 result = yes_entropy + no_entropy;
74 return result;
75 }

```

데이터셋에서 각 feature 로 나누었을 때의 엔트로피를 계산하는 함수이다. 가장 정보량이 큰 feature 번호를 판별하고 이를 분류기준으로 삼는 연산을 하는 데에 사용되는 함수이다.

현재 feature 에서의 데이터가 참여(1)인지 불참(0)인지, 참여이면서 데이터의 클래스는 참여인지 불참인지, 불참이면서 데이터의 클래스는 참여인지 불참인지에 따라 엔트로피 값이 달라진다.

```

77 // ----- 5. 정보 획득량이 가장 큰 feature의 index를 찾는 함수 -----
78 int max_info(int data_idx[], int data_len, int for_node) {
79     float min_entropy = 100, flag = 0;
80     int min_feature_idx = 0, for_for = for_node;
81     int *feature = malloc(sizeof(int) * data_len);
82     for (int feature_idx = 0; feature_idx < feature_num; feature_idx++) {
83         for_for = for_node;
84         for (int i = 0; i < data_len; i++)
85             feature[i] = data[data_idx[i]][feature_idx];
86         if (feature_entropy(feature, data_idx, data_len) < min_entropy) {
87             while (for_for != 0) {
88                 if (feature_idx == nodes[for_for]) {
89                     flag = 1;
90                     break;
91                 }
92                 for_for /= 2;
93             }
94             if (flag == 1) { flag = 0; continue; }
95             min_entropy = feature_entropy(feature, data_idx, data_len);
96             min_feature_idx = feature_idx;
97         }
98     }
99     free(feature);
100     return min_feature_idx;
101 }

```

위의 엔트로피 계산 함수를 이용하여 가장 정보획득량이 큰 feature 번호를 찾아주는 함수이다. 가장 정보 획득량이 많아 선택된 feature 번호는 해당 노드의 분류기준이 되며 자식노드들에게 해당 feature 번호에서의 데이터값이 참여인지 불참인지에 따라 데이터가 나뉘어 전달되게 된다.

```

103 // ----- 6. leftchild와 rightchild를 만드는 함수 -----
104 void make_child(treePtr node, int feature_idx, int for_node, int data_idx[], int data_len) {
105     treePtr temp1, temp2;
106     int for_node1 = for_node, for_node2 = for_node;
107     int *yes_data = malloc(sizeof(int) * 500);
108     int *no_data = malloc(sizeof(int) * 500);
109     int check, y_idx = 0, n_idx = 0;
110     int left_feature = 0, right_feature = 0;
111     int y_class_num = 0, n_class_num = 0;
112
113     // leftchild 데이터 집합과 rightchild 집합을 분류
114     for (int i = 0; i < data_len; i++) {
115         if (data[data_idx[i]][feature_idx] == 1) { yes_data[y_idx] = data_idx[i]; y_idx++; }
116         else if (data[data_idx[i]][feature_idx] == 0) { no_data[n_idx] = data_idx[i]; n_idx++; }
117     }
118
119     // 모든 데이터의 class count
120     for (int i = 0; i < data_len; i++) {
121         if (data_class[data_idx[i]] == 1) y_class_num++;
122         else if (data_class[data_idx[i]] == 0) n_class_num++;
123     }
124
125     // 전달 받은 데이터의 class가 모두 같은 경우
126     if (y_class_num == 0 || n_class_num == 0) return;

```

다음은 노드의 leftchild와 rightchild를 만드는 함수이다. 더이상 자식 노드를 만들 수 있는지를

check 하고, 만들 수 있는 경우 어떤 데이터 값(분류기준 feature 혹은 리프노드)을 넣어 노드를 생성할 것인지를 정한 다음 노드를 생성한다.

더이상 자식 노드를 만들 수 없는 경우는 더이상 사용할 수 있는 feature 가 없거나, 전달 받은 데이터셋에 포함된 데이터들의 클래스가 모두 같은 경우이다.

```
127 // 더이상 데이터 분류에 사용할 feature가 없는 경우
128 if (for_node * 2 > pow(2, feature_num) - 1) {
129     if (y_class_num > n_class_num) {
130         temp1 = insert_node(node, 1);
131         for_node1 *= 2;
132         nodes[for_node1] = 1;
133         return; }
134     else if (n_class_num >= y_class_num) {
135         temp2 = insert_node(node, 0);
136         for_node2 = (for_node2 * 2) + 1;
137         nodes[for_node2] = 0;
138         return; }}
```

먼저 더이상 데이터 분류에 사용할 feature 가 없는 경우를 체크해 주는 부분이다. 트리의 높이가 feature 개수보다 커지는 경우, 모든 feature 를 트리 노드에서 분류기준으로 사용했음을 의미하게 된다. 따라서 노드 번호가 2 의 feature\_num(feature 개수)제곱보다 커지는 경우 더이상 자식 노드를 생성하지 않고 해당 노드는 리프 노드 처리를 해 주었다.

```
140 else { // leftchild 노드 생성
141     check = check_class(yes_data, y_idx);
142     if (check == 0 || check == 1) { // 데이터의 class가 모두 같은 경우
143         temp1 = insert_node(node, check);
144         for_node1 *= 2;
145         nodes[for_node1] = check;
146         make_child(temp1, left_feature, for_node1, yes_data, y_idx); }
147     else {
148         left_feature = max_info(yes_data, y_idx, for_node1);
149         for_node1 *= 2;
150         nodes[for_node1] = left_feature;
151         temp1 = insert_node(node, left_feature);
152         make_child(temp1, left_feature, for_node1, yes_data, y_idx); }
```

분류에 사용할 feature 가 없는게 아닌 경우, leftchild 또는 rightchild 를 생성하는 부분으로 넘어가준다. 이 때 데이터의 class 가 모두 같은 경우에는 feature 번호가 아닌 클래스 번호(0 또는 1)을 해당 노드의 데이터 필드 값으로 설정해준다. 해당 값을 할당 받은 경우에는 다음 recursion 에서 더 이상의 분류를 중단하고 함수를 빠져나오게 된다.

```

154 // rightchild 노드 생성
155 check = check_class(no_data, n_idx);
156 if (check == 0 || check == 1) { // 데이터의 class가 모두 같은 경우
157     temp2 = insert_node(node, check);
158     for_node2 = (for_node2 * 2) + 1;
159     nodes[for_node2] = check;
160     make_child(temp2, right_feature, for_node2, no_data, n_idx); }
161 else {
162     right_feature = max_info(no_data, n_idx, for_node2);
163     for_node2 = (for_node2 * 2) + 1;
164     nodes[for_node2] = right_feature;
165     temp2 = insert_node(node, right_feature);
166     make_child(temp2, right_feature, for_node2, no_data, n_idx); }
167 }
168 free(yes_data);
169 free(no_data);
170 }

```

leftchild 와 절차는 동일하다. 매번 노드 생성 후에는 make\_child 함수를 다시 호출해 recursive 한 형태로 노드를 생성하게 된다.

```

172 // ----- 7. decisiontree를 생성하는 함수 -----
173 treePtr create_tree(int data_index[]) {
174     int check, feature_idx, for_node = 0;
175     treePtr rootNode = (treePtr)malloc(sizeof(*rootNode));
176
177     // 처음 입력받은 데이터의 클래스가 모두 같은 경우
178     check = check_class(data_index, data_num);
179     if (check == 0 || check == 1) {
180         printf("아무런 분류도 이루어지지 않았습니다.");
181         rootNode->data = check;
182         rootNode->leftchild = NULL;
183         rootNode->rightchild = NULL;
184         return rootNode; }
185
186     // decision tree의 루트노드 생성
187     feature_idx = max_info(data_index, data_num, for_node);
188     rootNode->data = feature_idx;
189     rootNode->leftchild = NULL;
190     rootNode->rightchild = NULL;
191     for_node++;
192     nodes[for_node] = feature_idx;
193
194     // 루트 노드 밑으로 새로운 트리 생성
195     make_child(rootNode, feature_idx, for_node, data_index, data_num);
196     printf("Decision Tree가 생성되었습니다 . . .\n");
197     return rootNode;
198 }

```

다음은 본격적으로 decision tree의 생성을 시작하는 함수이다. 루트노드를 가장 먼저 생성한 다음 루트노드 밑으로 트리를 확장해 나간다. 모든 tree의 생성이 끝나고 make\_child 함수의 재귀적 호출이 끝나는 경우, Decision Tree가 생성되었다는 안내문구를 출력하게 된다.

```

200 // ----- 8. Decision Tree 구성 노드 메모리 할당 해제 함수 -----
201 void postorderTraverse(treePtr np) {
202     if (np == NULL) return;
203     postorderTraverse(np->leftchild);
204     postorderTraverse(np->rightchild); }
205 void delTree(treePtr np) {
206     if (np != NULL) {
207         postorderTraverse(np->leftchild);
208         postorderTraverse(np->rightchild);
209         free(np); }
210     printf("트리의 모든 노드가 삭제되었습니다 . . . \n");
211 }

```

트리의 사용이 끝난 다음 트리를 구성하는 노드의 메모리를 모두 해제하는 함수이다. postorder Traversal 을 사용하여 모든 노드들을 순회하며 할당된 메모리를 free 시킨다.

```

213 // ----- 9. main 함수 -----
214 int main() {
215     int data_feature, feature_idx;
216     // data.txt 파일 오픈 및 정보 저장
217     FILE *fp;
218     fp = fopen("data.txt", "r");
219     if (fp == NULL) {
220         fprintf(stderr, "file open error \n");
221         exit(1); }
222     fscanf(fp, "%d%d", &data_num, &feature_num);
223     for (int i = 0; i < data_num; i++) {
224         for (int j = 0; j < feature_num; j++)
225             fscanf(fp, "%d", &data[i][j]);
226         fscanf(fp, "%d", &data_class[i]); }
227     fclose(fp);
228
229     // 초기 데이터 idx 배열 생성
230     int *data_index = malloc(sizeof(int) * data_num);
231     for (int i = 0; i < data_num; i++)
232         data_index[i] = i;

```

작성한 함수들을 호출해 연산을 수행하는 메인 함수이다. 먼저 data.txt 파일을 읽어 Decision Tree 생성에 필요한 학습데이터들의 정보를 앞서 전역 변수 선언에서 선언한 여러 변수들과 배열에 저장한다.

그 다음 초기 데이터 idx 배열을 생성한다. 해당 배열을 현재 전달하는 데이터셋에 포함되어있는 데이터가 어떤 데이터들인지를 인덱스 번호를 통해 알려준다. 가장 먼저 전달되는 데이터셋은 모든 데이터를 포함하는 집합이므로 초기 데이터 idx 배열은 처음부터 끝까지의 데이터 인덱스번호를 모두 포함한다.

```

234 // test.txt 파일 오픈 및 정보 저장
235 fp = fopen("test.txt", "r");
236 if (fp == NULL) {
237     fprintf(stderr, "file open error \n");
238     exit(1); }
239 fscanf(fp, "%d", &test_num);
240 for (int i = 0; i < test_num; i++)
241     for (int j = 0; j < feature_num; j++)
242         fscanf(fp, "%d", &test[i][j]);
243 fclose(fp);
244
245 // decisiontree 생성
246 treePtr rootNode, temp;
247 rootNode = create_tree(data_index);
248 temp = rootNode;
249 feature_idx = temp->data;

```

그 다음 test.txt 파일을 입력받아 내부 데이터셋을 저장한다. 이는 Decision Tree 생성 이후 생성된 트리의 정확도를 판별하는데 사용된다.

그 다음 create\_tree 함수를 호출하여 Decision Tree 를 생성하고, rootNode 변수에 생성된 트리의 루트노드 포인터를 전달받는다. 해당 변수를 통해 생성된 트리 내부에 접근할 수 있다.

```

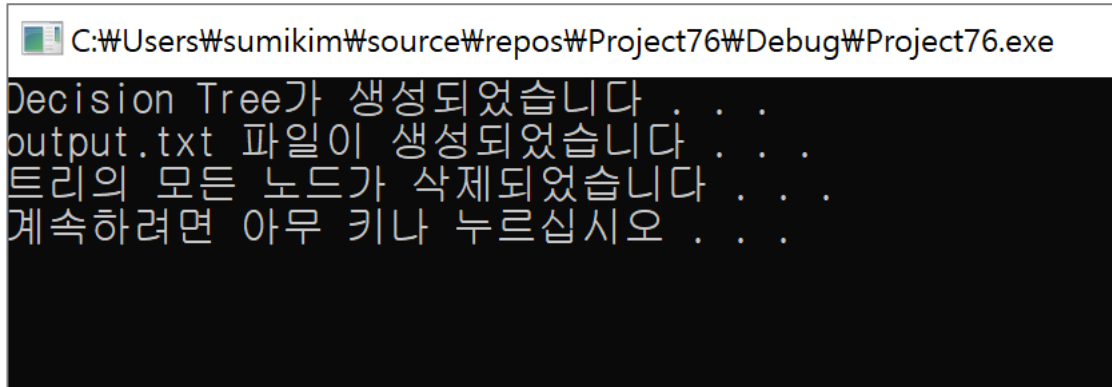
251 // test 데이터 분류
252 fp = fopen("output.txt", "wt");
253 if (fp == NULL) {
254     fprintf(stderr, "file open error \n");
255     exit(1); }
256 for (int i = 0; i < test_num; i++) {
257     feature_idx = rootNode->data;
258     temp = rootNode;
259     // 자식노드가 둘 다 NULL이 아니면 계속 반복
260     while (temp->leftchild != NULL && temp->rightchild != NULL) {
261         data_feature = test[i][feature_idx];
262         if (data_feature == 1) temp = temp->leftchild;
263         else if (data_feature == 0) temp = temp->rightchild;
264         feature_idx = temp->data; }
265     // 자식노드의 둘 중 하나가 NULL인 경우 (무조건 left)
266     if (temp->leftchild != NULL && temp->rightchild == NULL) {
267         temp = temp->leftchild;
268         feature_idx = temp->data; } // 리프노드의 데이터 값 반환
269     fprintf(fp, "%d\n", feature_idx);
270 } printf("output.txt 파일이 생성되었습니다 . . .\n");
271 fclose(fp);
272
273 // 생성된 트리의 모든 노드 메모리 할당 해제
274 delTree(rootNode);
275
276 free(data_index);
277 system("pause");
278 return 0;
279 }

```

이제 테스트 데이터셋의 클래스 분류를 생성한 Decision Tree 로 진행한다. 루트노드부터 시작해 매번 어떤 feature 번호의 분류기준을 만나는지에 따라 해당 feature 의 데이터값이 0 인지, 1 인지를

확인 후 leftchild 또는 rightchild 로 이동하고, 리프노드를 만날 때 까지 이를 반복한다.

모든 분류가 끝났으면 결과를 output.txt 파일에 출력 후 저장하고, 사용한 Decision Tree 의 모든 노드 메모리를 해제시켜준다.



```
C:\Users\sumikim\source\repos\Project76\Debug\Project76.exe
Decision Tree가 생성되었습니다 . . .
output.txt 파일이 생성되었습니다 . . .
트리의 모든 노드가 삭제되었습니다 . . .
계속하려면 아무 키나 누르십시오 . . .
```

해당 코드를 실행시키면 콘솔창에서 위와 같은 메시지들을 확인할 수 있다.