K-Means Clustering 과제 보고서

2020. 11. 9 (월) 5PM / 20181202 김수미

1. 코드 및 알고리즘 / 자료구조 설명

K-means Clustering은 주어진 데이터셋을 K개의 군집으로 묶는 clustering 알고리즘이다. 기계학습 중 비지도학습의 일종이며, 그 중에서도 비계층적 군집분석 알고리즘에 속한다. 비지도학습이기 때문에 지난 과제의 KNN 알고리즘과는 달리 정답 레이블이 주어지지 않으며, 비슷한 특성을 가지는 데이터(해당 알고리즘에서는 거리가 가까운 데이터끼리는 비슷한 특성을 가짐으로 간주)끼리 묶어 군집을 형성한다. K-means Clustering은 데이터를 K개의 군집으로 묶으며, K값을 모델이 스스로 정하지는 않으므로 직접 정해서 미리 알려줘야 한다.

K-means Clustering은 아래와 같은 과정으로 작동한다.

1. K개의 centroid 점들을 랜덤하게 샘플링한다.
2. 데이터셋의 각 데이터들을 가장 가까운 centroid와 같은 군집이라고 분류한다.
3. 각 군집의 centroid들을 해당 군집에 속한 데이터들의 평균 점으로 업데이트 한다.
4. 2단계, 3단계를 더 이상 군집 분류에 변화가 없거나 사용자가 정해놓은 maximum iteration수에 도달할 때 까지 반복한다.

해당 과제에서는 K-Means Clustering을 singly linked list 자료구조를 사용하여 구현한다. K개의 singly linked list의 head pointer를 저장하고 있는 pointer array가 있고, 각 linked list는 첫 번째 node에 centroid 정보가, 그 이후의 노드들에 해당 군집에 속하는 데이터들의 정보가 저장되도록 구현하자.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #define MAX_ITER 50
5  #define MAX_SIZE 500
```

코드 작성에 필요한 헤더파일들과 상수값을 설정해 주었다. MAX_ITER는 Clustering 작업의 최대 반복 횟수이며, MAX_SIZE는 데이터를 저장할 배열의 최대 크기이다.

```
7     float dataset[MAX_SIZE][MAX_SIZE];
8     float centroid[MAX_SIZE][MAX_SIZE];
9     int data_num, feature_num, K;
10    int random = 1000;
11
12    /* ***** Singly Linked List를 구성하는 노드 ***** */
13    typedef struct node {
14        float data[MAX_SIZE];
15        struct node *next;
16    } node;
```

dataset[MAX_SIZE][MAX_SIZE] : input.txt의 데이터들을 저장하는 배열이다.데이터가 몇 번째 데이터인지에 대한 정보와 해당 데이터가 가지고 있는 feature값들을 모두 저장한다.

centroid[MAX_SIZE][MAX_SIZE] : 랜덤한 centroid 점들의 값을 저장하는 배열이다.

총 K개의 centroid 값들을 저장하므로 몇 번째 centroid 값인지에 대한 정보와 해당 centroid가 가지고 있는 feature값들을 모두 저장한다.

data_num : 데이터 개수를 저장 / feature_num : 데이터의 feature 개수를 저장 / K : K값 저장

typedef struct node : Singly Linked List를 구성하는 노드. 데이터 값과 다음 노드를 가리키는 포인터 next 값을 가진다. 데이터는 feature가 여러개면 여러개의 값을 가질 수 있으므로 일차원 배열의 형태로 잡아 주었다.

```
18    /* ***** 노드 뒤에 노드 하나를 추가하는 함수 ***** */
19    void addFirst(struct node *target, float data[MAX_SIZE]) {
20        struct node *newNode = malloc(sizeof(struct node));
21        newNode->next = target->next;
22        for (int i = 0; i < feature_num; i++)
23            newNode->data[i] = data[i];
24        target->next = newNode;
25    }
26
27    /* ***** 노드 뒤에 노드 하나를 삭제하는 함수 ***** */
28    void removeFirst(struct node *target) {
29        struct node *removeNode = target->next;
30        target->next = removeNode->next;
31        free(removeNode);
32    }
```

addFisrt 함수 : 노드 뒤에 노드 하나를 추가하는 함수. 새로운 노드는 언제나 헤더노드 바로 뒤에 삽입해 주기 때문에 간편하게 사용할 수 있다.

removeFisrt 함수 : 노드 뒤에 노드 하나를 삭제하는 함수. 바로 뒤에 노드를 삭제하기 때문에 현재 살펴보고 있는 노드를 삭제하려면 해당 노드 바로 앞 노드의 주소를 가지고 있어야 한다.

```
34      /* ***** 파일을 오픈해 배열에 데이터셋을 저장하는 함수 ***** */
35      void fileopen(char *filename) {
36          FILE* fp;
37          fp = fopen(filename, "r");
38          if (fp == NULL) {
39              fprintf(stderr, "file open error ₩n");
40              exit(1); }
41
42          fscanf(fp, "%d", &data_num); // 데이터 개수
43          fscanf(fp, "%d", &feature_num); // 데이터 feature 개수
44          fscanf(fp, "%d", &K); // K값
45
46          // 데이터를 2차원 배열에 저장
47          for (int i = 0; i < data_num; i++) {
48              for (int k = 0; k < feature_num; k++)
49                  fscanf(fp, "%f", &dataset[i][k]);
50          }
51          fclose(fp);
52      }
```

fileopen 함수 : 파일을 오픈해 배열에 데이터셋을 저장하는 함수이다. 헷갈리거나 중간에 빠지는 데이터가 발생하는 것을 방지하기 위해 먼저 배열에 데이터들을 저장해둔 다음 singly linked list 에 데이터 값을 할당해 주었다.

```
54      /* ***** random centroid 값을 생성하는 함수(범위 : 0~3 사이) ***** */
55      float rand_cent() {
56          srand(random);
57          float result = rand() % 3 + rand() % 100000 / 100000.0;
58          random *= 2;
59          return result;
60      }
```

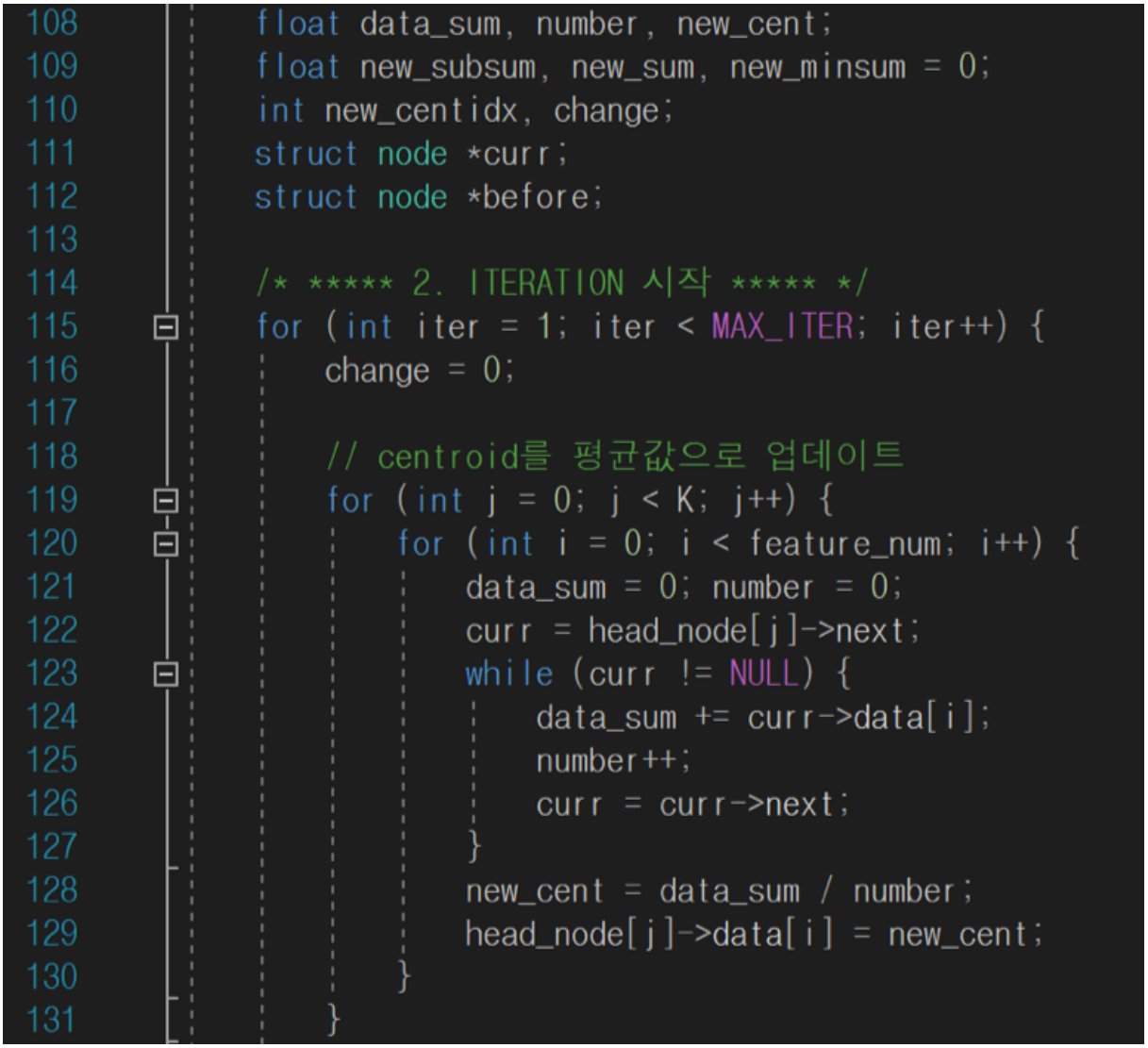
rand_cent 함수 : random centroid 값을 생성하는 함수이다. 무난하게 float 0 에서 3 사이의 값을 생성해 centroid 로 사용하여 clustering 을 진행했다.

```
62      /* ***** main 함수 시작 ***** */
63      int main() {
64          // 파일 오픈 및 데이터 저장
65          fileopen("input.txt");
66
67          // K개의 centroid 초기값을 랜덤으로 설정해준다.
68          for (int i = 0; i < K; i++) {
69              for (int k = 0; k < feature_num; k++)
70                  centroid[i][k] = rand_cent();
71          }
72
73          // K개의 헤더노드를 저장하는 배열 동적할당 후 초기값 삽입
74          node **head_node;
75          head_node = (node**)malloc(sizeof(node*) * K);
76          for (int i = 0; i < K; i++) {
77              struct node *head = malloc(sizeof(struct node));
78              for (int k = 0; k < feature_num; k++) {
79                  head->data[k] = centroid[i][k];
80              }
81              head->next = NULL;
82              head_node[i] = head;
83          }
```

main 함수의 시작 부분이다. k-means clustering 작업은 따로 함수를 만들지 않고 모두 main 함수 내부에서 작성해주었다. 먼저 파일을 오픈해 배열에 데이터들을 저장해 둔 다음, K 개의 centroid 초기값을 랜덤으로 설정해 주었다. 그 다음 K 개의 헤더노드의 pointer 를 저장하는 포인터배열을 동적으로 할당해 준 뒤 centroid 값을 저장하고 있는 헤더노드들을 가리키도록 설정해 주었다.

```
85          /* ***** 1. 초기 링크드리스트를 생성한다 ***** */
86          int cent_idx;
87          float sum, subsum, min_distance = 0;
88          for (int i = 0; i < data_num; i++) {
89              for (int j = 0; j < K; j++) {
90                  sum = 0, subsum = 0;
91                  // 가장 가까운 centroid를 찾는다
92                  for (int k = 0; k < feature_num; k++) {
93                      subsum = (head_node[j]->data[k] - dataset[i][k]);
94                      subsum = pow(subsum, 2);
95                      sum += subsum;
96                  } sum = sqrt(sum);
97                  if (j == 0) {
98                      min_distance = sum;
99                      cent_idx = j; }
100                 else if (sum < min_distance) {
101                     min_distance = sum;
102                     cent_idx = j; }
103             }
104             // 가장 가까운 centroid값의 헤더노드 뒤에 삽입
105             addFirst(head_node[cent_idx], dataset[i]);
106         }
```

초기 링크드리스트를 생성하는 부분이다. 현재 데이터값들이 배열에 저장되어 있기 때문에 배열의 데이터 값을들 centroid 값과의 거리를 비교한 뒤, 해당 데이터값을 저장하는 노드를 새로 생성해 가장 가까운 centroid 값을 저장하고 있는 헤더 노드 바로 뒤에 연결해 주었다. 새로운 노드는 리스트의 맨 뒤에 붙이지 않고 매번 헤더노드 뒤에 끼워넣었다.



```
    float data_sum, number, new_cent;
    float new_subsum, new_sum, new_minsum = 0;
    int new_centidx, change;
    struct node *curr;
    struct node *before;

    /* ***** 2. ITERATION 시작 ***** */
    for (int iter = 1; iter < MAX_ITER; iter++) {
        change = 0;

        // centroid를 평균값으로 업데이트
        for (int j = 0; j < K; j++) {
            for (int i = 0; i < feature_num; i++) {
                data_sum = 0; number = 0;
                curr = head_node[j]->next;
                while (curr != NULL) {
                    data_sum += curr->data[i];
                    number++;
                    curr = curr->next;
                }
                new_cent = data_sum / number;
                head_node[j]->data[i] = new_cent;
            }
        }
```

이제 singly linked list 가 생성되었기 때문에, 본격적으로 clustering 작업을 시작할 차례이다. 먼저 필요한 변수들을 선언해준 다음, clustering loop 내부에 작업 코드를 작성해 주었다.
change 변수는 군집의 업데이트가 발생했는지 여부를 체크해주는 변수이며, 군집의 업데이트가 더이상 발생하지 않는 경우 작업을 계속하는 의미가 없어지기 때문에 해당 경우에는 반복 횟수가 50 번이 차지 않아도 clustering 작업을 중단시킬 것이다.

가장 먼저 해줄 작업은 centroid 값을 데이터들의 평균값으로 업데이트 시켜 주는 작업이다. 각 군집의 리스트를 처음부터 끝까지 훑으며 데이터 값의 합을 구하고, 군집에 속한 데이터 개수로 나누어 평균값을 구해 centroid 값을 업데이트해 주었다.

```
133             // 각 군집 업데이트
134             for (int j = 0; j < K; j++) { // 0~K번째 군집까지
135 
136                 before = head_node[j]; // 한단계 전 노드 가리킴
137                 curr = head_node[j]->next; // 현재 노드 가리킴
138 
139                 while (curr != NULL) { // 노드가 더이상 없을때까지 각 군집 탐색
140                     for (int k = 0; k < K; k++) {
141                         new_sum = 0;
142                         for (int i = 0; i < feature_num; i++) {
143                             new_subsum = head_node[k]->data[i] - curr->data[i];
144                             new_subsum = pow(new_subsum, 2);
145                             new_sum += new_subsum;
146                         } new_sum = sqrt(new_sum);
147                         if (k == 0) {
148                             new_minsum = new_sum;
149                             new_centidx = k;
150                         }
151                         else if (new_sum < new_minsum) {
152                             new_minsum = new_sum;
153                             new_centidx = k;
154                         }
155                     }
156 
157                     if (new_centidx != j) { // 소속된 군집을 변경 하는 경우
158                         change++; // 변경이 발생했음을 표시해줌
159                         addFirst(head_node[new_centidx], curr->data); // 새 노드 생성
160                         curr = curr->next; // 다음 데이터로 이동
161                         removeFirst(before); // 노드 삭제
162                     }
163 
164                     else { // 군집 이동 없는 경우
165                         before = before->next; // 군집 이동 없으면 next
166                         curr = curr->next; // 다음 데이터로 이동
167                     }
168 
169                 }
170 
171             }
172 
173             if (change == 0) break; // 더이상 군집에 변화가 없으면 작업 중단
174         }
```

그 다음 작업은 데이터들의 군집을 업데이트하는 작업이다. 업데이트된 centroid 값을 기준으로 군집에 속해있는 데이터들이 계속 해당 군집에 포함될 것인지, 아니면 다른 군집으로 이동시켜야 하는지 확인한 다음 이동이 필요한 경우 해당 군집에 데이터 노드를 삽입한 뒤 기존 군집에서 데이터 노드를 삭제해주었다. (자세한 과정은 코드의 주석 참고 바람)

한 군집에서 다른 군집으로 데이터가 이동하는 경우 change 값을 양수로 만들어주어 이를 체크해 두었고, 더이상 데이터의 이동이 발생하지 않으면 change 값이 계속 0 으로 남아있기 때문에, 모든 군집의 데이터들의 확인이 끝난 뒤에서 change 값이 0 으로 남아 있으면 작업을 중단시켜 주었다.

```
176     /* ***** 3. Clustering 결과 및 파일 출력 ***** */
177     FILE *fp = fopen("output.txt", "wt");
178     if (fp == NULL) {
179         fprintf(stderr, "file open error ₩n");
180         exit(1);}
181
182     // k-means clusteing 결과를 확인한다
183     for (int j = 0; j < K; j++) {
184         struct node *curr2 = head_node[j]->next;
185         fprintf(fp, "%d₩n", j); printf("%d₩n", j);
186         while (curr2 != NULL) {
187             for (int i = 0; i < feature_num; i++) {
188                 fprintf(fp, "%f ", curr2->data[i]);
189                 printf("%f ", curr2->data[i]);
190             } fprintf(fp, "₩n"); printf("₩n");
191             curr2 = curr2->next;
192         }
193     } fclose(fp);
194
195     system("pause");
196     return 0;
197 }
```

위 반복문이 끝나면 모든 clustering 작업이 완료된 것이다. 이제 헤더노드 뒤에 연결된 노드들을 출력해줌으로써 데이터들이 어떻게 군집을 이루었는지 확인하기만 하면 된다.

각 군집의 centroid 값을 저장하고 있는 헤더노드부터 시작해 해당 리스트의 끝까지 탐색하며 모든 군집의 데이터들을 콘솔창에 출력함과 동시에 동일한 내용을 저장하는 output.txt 파일을 생성해 주었다.