**과제 1: 군집화**

20232907 정현승

**1. 분석하는 데이터 및 결론 활용 방안**

분석하려는 데이터는 고가 휴대전화의 구조나 성능 데이터로, kaggle의 Dataset[1] 에서 가져온 후 가격이 높은 데이터만 추출하여 군집화를 진행하였다. 이 과정을 통해 고가 휴대전화를 구조나 성능을 기준으로 분류하여, 저가형 휴대전화만 제작하던 중소 휴대전화 제조사가 고가 휴대전화 시장에 진출하려 할 때 기존에 존재하던 휴대전화의 크기, 성능 등의 데이터를 몇 종류로 나누어 이 중에 만만한 것을 똑같이 따라 만드는 방향으로 가도록 도울 수 있다.
이를 페르소나 설정을 이용해 쉽게 설명하면 다음과 같다. 중소기업 A사는 휴대전화를 만들어 판매하는 일을 한다. 지금까지 A사는 저가형 휴대전화만 판매해 왔다. 이제 A사는 기술이 어느 정도 쌓여, 고가 휴대전화 시장에 진출을 희망한다. A사는 단순히 고가의 휴대전화를 팔아봤다는 경험뿐 아니라, 삼성이나 애플 같은 대기업과 어깨를 나란히 할 정도로 성장하는 것을 희망한다. 그러나 A사는 아직 고가 휴대전화를 판매해 본 적이 없기에, 고가 휴대전화의 특징을 알지 못한다. 그렇다고 휴대전화를 아무렇게나 만들어 팔자니 판매량이 부진할 것이 우려된다. 따라서 A사는 **다른 기업에서 출시한 고가 휴대전화 데이터를 토대로 고가에 팔리는 휴대전화의 종류는 무엇이 있는지 알아보고**, 이 중 A사의 기술이나 재산 등을 이용해 따라 만들기 만만한 것을 따라 만들려 한다. 이때 고가에 팔리는 휴대전화의 구조나 성능 데이터를 이용해 고가에 팔리는 휴대전화의 종류를 몇 가지로 추려내는 과정이 이번에 진행할 군집화 과정이다.

데이터의 출처를 보면 이 데이터는 분류 알고리즘, 그중에서도 지도 알고리즘에 사용할 목적으로 만들어져 있으며, 데이터 파일을 보면 전처리 작업이 되어 있다는 것을 알 수 있다. 다만 원래 데이터의 목적이 휴대전화의 성능과 구조 등을 이용해 휴대전화의 가격대를 예측하는 것인데, 이번에는 이 데이터를 목적 그대로 사용하지 않고, 고가의 휴대전화 데이터를 추출해 이들을 몇 개의 그룹으로 나누어 각 그룹의 특징(고가에 팔리는 휴대전화의 특징)을 알아보았다.

데이터 추출은 train.csv에 저장된 2000개의 전체 데이터에서 price_range 필드 값이 3인 항목만 뽑아내는 방식으로 진행하였다. 군집화에는 데이터 추출에 사용된 필드를 제외하고 총 20개의 필드 데이터를 사용하며, 그 종류는 다음과 같다. 위 단락에서 설명했듯 이미 이 데이터는 전처리가 되어 있는 상태이다. 여기서 최소, 최댓값과 표준편차는 추출한 고가 휴대전화 데이터뿐 아니라 추출 이전 휴대전화 데이터 전체를 대상으로 계산한 것이다.

| 종류 | C언어 필드명 | 단위 | 최솟값 | 최댓값 | 표준편차 |
|---|---|---|---|---|---|
| 최대 베터리 저장 용량 | batteryPower | mAh | 501 | 1998 | 439 |
| 블루투스 기능 포함 여부 | hasBluetooth | 0 or 1 | 0 | 1 | 0.5 |
| 프로세서 처리 속도 | clockSpeed | GHz | 0.5 | 3 | 0.82 |
| 듀얼심(투넘버) | supportDualSim | 0 or 1 | 0 | 1 | 0.5 |
| 전면 카메라 화소 수 | frontCamPixel | megapixel | 0 | 19 | 4.34 |
| 4G 지원 여부 | support4G | 0 or 1 | 0 | 1 | 0.5 |
| 내장 메모리 | internalMemory | GB | 2 | 64 | 18.1 |
| 휴대전화 두께 | depth | cm | 0.1 | 1 | 0.29 |
| 휴대전화 무게 | weight | g | 80 | 200 | 35.4 |
| 프로세서 코어 수 | cores | 1개 | 1 | 8 | 2.29 |
| 카메라 화소 수 | camPixel | megapixel | 0 | 20 | 6.06 |
| 화면 화소 세로 | pixelHeight | pixel | 0 | 1960 | 444 |
| 화면 화소 가로 | pixelWidth | pixel | 500 | 1998 | 432 |
| Ram | ram | MB | 256 | 3998 | 1080 |
| 화면 세로 길이 | screenHeight | cm | 5 | 19 | 4.21 |

1) https://www.kaggle.com/datasets/iabhishekofficial/mobile-price-classification

| | | | | | |
|---|---|---|---|---|---|
| 화면 가로 길이 | screenWidth | cm | 0 | 18 | 4.36 |
| 배터리 지속시간 | batteryLife | h | 2 | 20 | 5.46 |
| 3G 지원 여부 | support3G | 0 or 1 | 0 | 1 | 0.43 |
| 터치스크린 유무 | hasTouchscreen | 0 or 1 | 0 | 1 | 0.5 |
| Wifi 지원 여부 | hasWifi | 0 or 1 | 0 | 1 | 0.5 |

표 1

## 2. 결과 도출 조건

군집화는 k-means, k-medoids 2가지 방법을 이용했다. 데이터에서 사용하지 않는 부분을 제외해도 데이터의 차원이 20차원이 나오지만, 20가지 종류의 데이터 변수가 전부 숫자로 정의되어 있어, 평균을 정의할 수 있기에 k-means를 사용했다. True 또는 false 값이 들어가 있는 데이터 변수는 true를 1, false를 0으로 놓고 사용해 평균을 정의할 수 있었다.

이때 데이터 간 거리를 정의해야 하는데, 이 데이터 간 거리는 어떻게 설정하느냐에 따라 결과가 크게 달라지는 중요한 요소이다. 그러나 여기에는 정해져 있는 답이 없어, 거리를 확실하게 정의할 방법이 없다. 예를 들어 몇 명의 사람을 키와 몸무게를 이용해 군집화할 때 두 사람이 얼마나 차이가 있는지 거리(스칼라값)로 나타내려 한다. 키와 몸무게 총 2차원으로 구성된 데이터 간 거리를 1차원으로 줄이는 방법은 딱 정의할 수 없다. 키와 몸무게를 단순히 Euclidean Distance를 사용해 구한다 해도 단위를 고려해야 한다. 키의 단위가 cm이냐 m이냐에 따라서도 거릿값은 완전히 달라진다. 또 키가 1cm 차이 나는 것과 몸무게가 1kg 차이 나는 것은 같다고 말할 수는 없을 것이다. 키가 5cm나 10cm 정도는 차이 나야 1kg 차이 나는 것과 비슷할 수도 있다. 이 키와 몸무게 간의 비율(가중치)은 정해진 게 없다. 어느 것 하나 정답이라 할 수 없다. 데이터의 분포를 고려한 거리인 Mahalanobis Distance를 사용한다고 하더라도, 데이터의 분포가 꼭 데이터 간 실제 가중치와 비슷하다는 의미는 될 수 없다. 따라서 더 중요한 데이터에 가중치를 더 주어야 한다는 점을 제외하고는 명확한 거리 측정 방법을 알 수 없다. 그렇기 때문에 이번에는 거리 조건을 2개 설정하고 군집화를 시행해, 더 좋은 결과가 나온 쪽을 선택하였다. 거리 조건 자체는 Euclidean Distance 기반으로, 가중치를 달리해 결과를 산출하였다.

결과 도출 방식은 k-means와 k-medoids 알고리즘 2개를, 각각 중심점(center) 개수와 거리 정의 방법을 달리해 결과를 도출하고, 각 결과의 intra distance 평균을 inter distance 평균으로 나눈 값이 작을수록 더 좋은 결과라고 생각하였다. 각 결과 도출 시에도 초기 random point를 1000번 달리해 1000개의 결과를 얻었고, 이 중 완전히 동일한 결과가 많이 나온 것을 최종 결과로 보았다. 과정을 1000번 반복하는 이유는 결과가 local minima에 빠질 수 있기 때문이다. 이때 intra distance는 최적화를 위해 center에서 점까지의 거리 중 최댓값을 사용했고, inter distance는 2개의 군집에서 각각 데이터 하나씩을 뽑을 때 나올 수 있는 모든 거릿값 중 최솟값으로 설정하되 center 간의 거리가 다른 값보다 작으면 이 값으로 했다.

## 3. 소프트웨어 구조

소스코드에서 “//do_something”이라 적힌 부분은 데이터의 종류가 달라질 때마다 변경해야 하는 부분이다. 알고리즘을 먼저 짜고 데이터를 넣었기에 필요했으며, 또 이 알고리즘을 다른 데이터를 군집화할 때도 사용할 수 있도록 삭제하지 않고 놔두었다.

데이터는 20개의 멤버 변수를 가진 구조체로 선언하였다. 이때 실제 데이터에는 부동 소수점 자료형이 2종밖에 없는데 20개의 멤버 변수 모두 부동 소수점 자료형으로 선언한 이유는 평균을 정의해야 하기 때문이다. 특히 0과 1만 존재하는 데이터는 그 평균이 0과 1 사이일 텐데, int나 bool 자료형으로 선언하면 0과 1 사이의 값을 담을 수 없어지기 때문에 사용할 수 없다. 따라서 0과 1만 존재하는 자료형을 포함한 대부분의 멤버가 float 형으로 선언되었으며, 원래 데이터가 부동 소수점 자료형이거나 데이터의 범위(최댓값과 최솟값의 차이)가 적은 멤버는 정확한 평균 계산을 위해 double로 선언했다. 단 0과 1만 존재하는 자료형은 double이 아닌 float 형으로 두었다.

구조체 Assign은 k-means나 k-medoids를 사용한 후 그 결과를 담는 구조체로, 데이터와

이 데이터에 해당하는 center를 가리키는 포인터 하나씩으로 이루어져 있다.

나머지 2개의 구조체는 단순히 함수에서 여러 개의 값을 반환받기 위해 선언한 구조체이다.

많은 양의 데이터를 다루는 특성상 동적 할당이 매우 많이 사용되는데, 동적 할당 후 성공 여부는 매번 확인하고 실패 시 함수 내부에서 exit(1)을 진행하였다. 다른 교수님은 동적할당 해제를 신경 쓰지 말라고 하셨는데, 함수 내부에서 exit(1)이 호출되는 상황을 제외하고는 동적할당 해제도 진행하였다.

가장 먼저 파일로부터 데이터를 읽었다. 먼저 파일을 열고 파일의 길이를 확인한다. 이후 이 길이를 전부 받을 수 있는 문자열과 이 문자열을 잘라 저장할 배열을 동적으로 할당받는다. 이때 파일을 읽기 전 데이터 개수만큼 배열을 동적으로 할당받는바, 여기에서만큼은 데이터의 개수를 미리 알려줄 필요가 있다. realloc 함수를 사용하는 방안도 있으나 이 또한 실행시간 측면에서 큰 낭비라고 생각해 시행하지 않았다. 따라서 #define을 이용해 미리 데이터의 개수를 넣어주었으며, 이 값은 여기에서만 사용하고 다른 데에서는 데이터를 직접 읽으며 구한 데이터의 개수를 사용한다. 파일을 읽을 때는 각 멤버 요소를 나타내는 첫 번째 줄은 사용하지 않고, 그다음 줄부터 파일의 끝까지 입력받는다(입력받을 때는 pin 포인터를 이용한다). 한 줄씩 입력받으며 입력받은 모든 줄을 문자열의 배열(datas)에 저장한다(저장 시에는 datas 포인터를 직접 사용하지 않고 pdatas 포인터를 이용해 저장한다). 이렇게 파일을 다 읽었으면 파일은 닫고, 읽은 데이터를 ‘,’ 기준으로 잘라 부동 소수점 자료형으로 저장한다. 저장해야 하는 멤버 변수만 20개라 코드가 매우 길어 보이지만 잘 보면 같은 것을 반복하는 코드이다. 위 과정은 반복문 사용이 불가능해 20줄을 전부 작성했다. 마지막으로 파일에서 읽어 저장한 공간과 이를 잘라 저장한 배열 2개의 동적할당을 해제하고, 저장된 멤버 변수와 데이터 개수를 반환한다. 데이터 중에 문자열이 있을 것을 고려해 문자열 포인터도 반환할 수 있게(이후 적절한 시점에 동적할당을 해제할 수 있도록) 코드를 작성했지만, 현재 사용하는 데이터에는 문자열이 없어 활용하지 않았다.

이후 1000번 k-means 또는 k-medoids의 결과를 산출한다. 두 과정의 결괏값 자료형도 같고 진행 과정도 상당히 유사해, 두 과정을 다른 함수로 빼지 않고 하나의 함수로 만들었다. enum mode 자료형의 인자를 사용해 두 과정 중 하나를 선택하게 했다. 우선 center를 무작위로 찍는다. 단 무작위로 찍되 각 멤버 변수가 실제 존재 가능한 데이터가 되도록(그러니까 0과 1만 들어가는 멤버에 2나 0.5 같은 값을 넣지 않는다는 말이다) 최댓값과 최솟값, 그리고 범위와 단위를 조정했다. 이후 데이터별로 center를 mapping 한 결과가 들어갈 배열을 5개 선언하고(assignResult, 이 자료형을 생성하는 과정은 while 문 직전에 존재하는 for 문까지 포함한다), 데이터를 하나하나 넣어주는 작업을 진행한다(이제 데이터는 전체 데이터가 저장된 배열에서 직접 읽는 게 아니라 이 mapping 결과(구조체 Assign 배열) 중 data 필드를 이용해 확인한다).

이제 Assign과 Refitting을 반복한다. Assignment Step은 하나의 data 별로 어떤 center가 가장 가까운지 확인하여, 가장 가까운 center에 해당하는 점을 이 data에 해당하는 center로 보고 그 값(포인터 값)을 mapping 결과로 넣었다. 이 작업을 모든 data에 대해 시행하면 Refitting Step으로 넘어갔다. Refitting Step은 한 center에 대해 이 center에 mapping 된 모든 데이터의 평균(k-means) 또는 다른 모든 데이터와의 거리 합이 가장 작은 데이터 하나(k-medoids)를 구해야 한다. 이때 하나의 center에 mapping 된 모든 데이터를 하나로 모으는 게 더 효율적일 것으로 판단하여, **center 조정 작업 전 mapping 데이터를 정렬**하였다. 이때 정렬은 mapping 정보가 저장된 데이터의 위치(포인터)나 데이터의 center(포인터)나 center의 순서가 달라도 내용이 같으면 같은 것으로 취급되도록 내부에 저장된 값을 확인했다. center의 내용이 다를 때는 이후 과정을 대비해 특정 순서로 정렬을 해 주어야 하는데, center 내용이 일관된 순서대로 정렬되면 되기 때문에(이는 후술한다) memcmp 함수에서 받은 반환 값을 그대로 이용했다. 다만 파일에서 읽은 데이터는 그냥 포인터 값으로 같은지 다른지를 판단했는데, 어차피 파일에서 읽은 데이터가 저장된 위치(포인터)는 한 곳이고 그 포인터 값만 계속 돌려쓰고 있는 것이기 때문이다. 이렇게 데이터를 정렬한 후 같**은 center에 mapping 된 데이터를 모두 찾아, center를 조정하였다.** 조정 작업도 후술한다. 마지막으로 4번째 전까지의 mapping 결과와 현재 결과를 비교해 기존 mapping 결과와 현재 결과가 같은지 확인해 같으면 Assign 및 Refitting 작업을 중단하고, 아니면 이 작업을 반복

하였다. 이때 직전 결과만 비교하지 않는 것은 데이터 몇 개가 한 번 작업을 반복할 때마다 mapping 된 center 값이 왔다 갔다 하는 사례가 있을 것을 대비한 것이다. 비록 mapping 결과만을 비교하게 되지만(center 값은 Assign 및 Refitting 작업을 한 번 할 때마다 값이 덮어써지기 때문에 활용할 수 없다) 어떤 데이터가 하나의 center에 mapping이 되는지 알고 있으면 center를 구할 수 있고, 같은 데이터가 center에 mapping 되면 center 계산 결과는 계산을 언제 하나 같을 것이기 때문에 문제는 생기지 않는다. 직전 mapping 결과와 이번 mapping 결과가 같다면 center 또한 변하지 않았을 것임을 충분히 예상할 수 있다. 따라서 center의 값을 직접 비교하지 않아도 문제가 없다.

위 작업이 끝나고 while 문을 나왔다면 군집화 작업은 거의 끝났다. 다만 앞에서 언급했던 mapping 된 center 값이 특정 데이터 사이에서 왔다 갔다 할 때, 왔다 갔다 하는 결과 중 아무거나 최종 결과로 고르면 나중에 같은 결과가 몇 번 나왔는지 확인하는 과정에서 각기 다른 결과로 판정되는 문제가 발생한다. 하나의 군집화 과정을 1000번 실행하여 가장 많이 나온 것을 결과로 취할 때 이런 상황이 발생하면 원래 최종 결과가 되어야 하는 결과 대신 다른 결과가 최종 결과가 되는 문제가 발생한다. 따라서 이런 상황을 미리 예방하기 위해 mapping 된 center 값이 특정 데이터 사이에서 왔다 갔다 할 때 나오는 여러 mapping 결과 중 하나를 골라 결과로 삼는다. 이런 상황에서 둘 중 특정한 것을 고른다고 데이터가 크게 달라지지 않으니 무엇을 고르는지는 상관이 없으나, 이런 상황이 또 발생하면 일관되게 같은 결과를 골라야 한다. 따라서 mapping의 데이터 포인터 순서로 골랐고, 이 데이터에 기준이 되는 center와 현재 덮어써진 center 결과가 다를 수 있으니, Refitting만 한 번 진행하였다. 위의 작업이 끝났으면 결과를 반환하였다.

k-means에서 center 구하는 건 단순히 평균을 이용했다. 하나의 center에 mapping 된 모든 데이터를 멤버별로 더한 뒤 데이터의 개수로 나누어주었다. 이는 데이터의 모든 자료형이 부동소수점이기에 가능하다. 다만 원래 데이터의 멤버 변수가 정수 또는 소수점 한 자리수 밖에 되지 않는다는 점을 고려해, 평균 계산 시 부동소수점의 precision은 고려하지 않았다.

k-medoid에서 center를 구하려면 우선 모든 데이터 간 거리를 구해야 하므로 미리 전부 구해 2차원 배열에 넣었다. 단 하나의 center에 하나의 데이터만 들어오면 그것은 예외상황으로, 예외처리를 진행하였다. 그때그때 데이터 간의 거리를 구하는 것보다 미리 한 번에 구해놓고 사용하는 게 좋을 것 같아 이렇게 진행하였다. 이때 데이터 간 거리를 저장하는 2차원 배열도 동적 할당으로 받았고, 실제 거릿값이 저장되는 공간은 한 번에 할당받아 필요한 만큼 적당히 나누어 사용했다. 이는 처음 파일에서 데이터를 읽어올 때 문자열을 받을 공간을 넉넉히 잡아두고 이를 나누어 문자열 배열에 저장했던 것과 같다. 거리 저장은 사용의 편의를 위해 x>y일 때 distanceTable[x][y]의 값이 x번째 데이터와 y번째 데이터 간 거리가 되도록, x<=y일 때는 존재할 수 없도록, 즉 오른쪽 그림처럼 계단식으로 2차원 배열의 구조를 만들어 저장했다. distanceTable[0]인 1차원 배열은 거리를 저장하는 공간이 아닌 임시 배열로 사용하였다. 여기서 한 번 더 사용을 편리하게 하기 위해 매크로 함수 getDistance를 정의해 x>y 조건을 신경 쓰지 않고 거리를 가져올 수 있도록 하였다. 위와 같이 거리를 다 구했으면 각 데이터에서 다른 모든 데이터 간 거리의 합을 구하고, 이 값이 가장 적은 데이터를 center로 했다. 이때 거리의 합을 구하는 과정에서는 부동소수점의 precision을 고려하여, 작은 수부터 하나씩 하나씩 더하도록 하였다. 이 과정에서 더할 대상 전체를 담은 배열이 필요한데, 이때 임시 배열 distanceTable[0]을 사용했다.

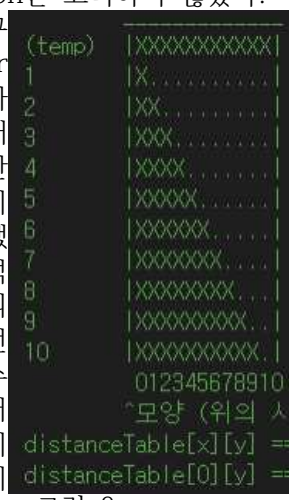


그림 2

부동소수점의 precision을 고려해 더하는 과정은 부동소수점 변수를 오름차순으로 정렬하고, 맨 앞의 두 수를 더한 뒤 그 수를 나머지 변수 사이에 오름차순에 맞게 삽입하는 행위를 반복하는 방식으로 진행하였다. 이번에는 프로그램 전체에서 음수를 취급하지 않으므로 절댓값을 취할 필요는 없다.

이렇게 k-means 또는 k-medoids 군집화를 1000번 반복하는 것까지 끝나면, 이제는 그 1000번의 데이터 중 하나를 고를 차례이다. 우선 1000번의 군집화 과정을 전부 정렬한다. 앞

에서 언급했듯, mapping 결과를 보고 첫 번째 mapping의 데이터 포인터가 일치하는지, center의 값이 같은지 확인하고 이를 기준으로 정렬하고 같은 것을 찾는다. 이때 군집화 과정에서 mapping 데이터 정렬은 끝냈으므로 mapping의 정렬은 신경쓰지 않아도 된다. 이번에도 정렬 순서는 큰 상관이 없고 같은 것으로 취급받는 데이터가 한 곳에 몰려 있기만 하면 된다. 이때 데이터 값이 아주 조금만 달라도 다른 것으로 취급하는데, center 위치가 비슷하면 결과가 대부분 그 근처에 몰려 있을 것이라고, 그리고 그 중 하나가 최다 결과로 최종 결과가 될 것이라 생각하여 별도의 처리를 진행하지 않았다(군집화 과정에서 점 몇 개가 왔다갔다할 때의 처리는 이미 진행했기도 하다). 이후 **최빈 등장 결과를 찾아 최종 결과로 가져온다.** 최종 결과는 intra distance 평균을 inter distance 평균으로 나눈 값과 함께 콘솔에도 출력하고 파일에도 저장한다. 이것이 소스코드가 하는 역할이며, center의 개수를 바꾸거나 거리의 정의를 바꾸거나, 아니면 k-means와 k-medoid 간 모드를 전환한다든가 하는 역할은 전부 소스코드를 직접 수정하면서 진행하였다. **이때 center 개수 변경은 코드 12번째 줄의 NUMOFGROUP 값을, k-means와 k-medoid 간 전환은 88번째 줄의 k_means_medoids 함수의 4번째 인자를, 거리 정의는 767줄에 정의된 distanceBetweenData 함수의 코드를 직접 바꾸면서 변경할 수 있다.**

**4. 결과**

k-means, k-medoids로 모드를 바꾸어 보고, center의 수를 2부터 4까지 바꾸어 보고, 또 거리의 정의를 바꾸어 보기도 하면서 여러 번 실행해 보았다.

거리의 정의는 처음에는 정규화를 이용했다. 각 멤버 변수의 차를 멤버의 표준편차로 나누었고, 이를 제곱한 후 모두 더하고 제곱근을 취한 게 첫 번째 거리의 정의였다. 이때 표준편차는 저가형 휴대전화까지 전부 포함하여 계산된, 데이터 추출 이전의 값을 사용했다. 이렇게 하는 게 전체 휴대전화 제품 현황을 보여줘 조금 더 거리로 쓰기에는 좋겠다는 판단에 의한 것이며, 표준편차는 군집화 과정이 아니므로 외부에서 그 값을 계산해 값만 넣어주었다. 이렇게 되면 모든 데이터가 동일한 가중치를 가지게 된다. 그러나 실제로는 어떤 요소는 가격에 더 많은 영향을 줄 수도 있기에, 이를 직접 판단하여 가중치를 부여하였다. 가중치 부여 방법은 위쪽의 표 3과 같다. 예를 들어 clockSpeed 필드는 다른 필드보다 5배 더 민감하여, 이 필드가 값이 1 차이나면 다른 필드가 5 차이나는 것과 같은 효과를 가진다.

| 가중치 부여 필드 | 가중치(배) |
|---|---|
| clockSpeed | 5 |
| frontCamPixel | 10 |
| internalMemory | 5 |
| cores | 5 |
| camPixel | 10 |
| pixelHeight | 2 |
| pixelWidth | 2 |
| ram | 5 |
| screenHeight | 3 |
| screenWidth | 3 |

표 3

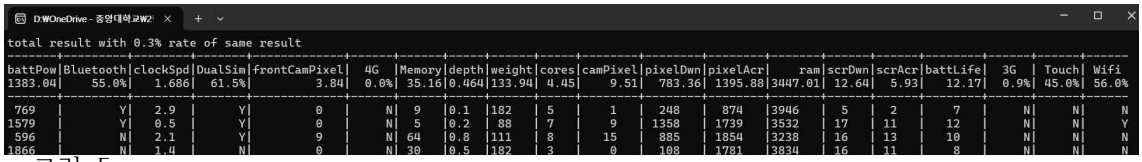
다음 표 4는 실행 결과를 평가 기준에 따라 정리한 표이다. 가장 많이 나온 결과가 몇 번 나온 결과인지, 걸리는 시간이 얼마인지는 보고자 하는 게 아니므로 생략했다.

| 거리 보정 없음(정규화) | | | 거리 보정 | | |
|---|---|---|---|---|---|
| center 수 | k-means | k-medoids | center 수 | k-means | k-medoids |
| 2 | 4.88377 | 5.324576 | 2 | 6.518834 | 9.145847 |
| 3 | 2.07307 | 3.118801 | 3 | 1.978950 | 2.645452 |
| 4 | 1.179668 | 1.849393 | 4 | 1.978950 | 1.698995 |

표 4

20차원 데이터라 시각화를 할 수 없어 직관적으로 어느 것이 좋다고 판단할 수는 없지만, 평가 기준만 놓고 보면 k-means가, 또 거리 보정이 없을 때, 그리고 center 수가 많을 때가 가장 좋았다. 이 사례에서는 오히려 거리 보정이 독이 되었다고 평가할 수 있다. 즉 **사람이 자기 생각대로 판단한 가중치는 오히려 역효과만을 불러올 수 있다는 결과**를 얻을 수 있다. 평가 기준은 아니지만 가장 많이 나온 결과가 등장한 횟수는 반대로 k-medoid가, 또 거리 보정이 있을 때, 그리고 center 수가 적을 때일수록 높았다. 다만 아래 그림 5처럼 1000번 시행 중 최다 등장 결과가 3번 정도만 등장한, 거리 보정 없을 때의 center 4개인 k-means 말고는, 모두 같은 결과가 100번 이상 등장하므로, 이는 크게 신경 쓰지 않아도 될 것으로 보인다(위 1개 경우는 실행을 여러 번 해도 결과가 크게 달라지지 않는 다른 경우와 달리 실행할 때마다 결과가 달라지는 현상을 쉽게 발견할 수 있다. 실제 테스트 중에도 여기서

k-means가 불확실하다는 것을 확인한 후 center의 수를 더 늘리지 않기로 했다). 이외에도 k-means는 1000번 시행에 1분 내외가 소요되고, k-medoids는 10분 내외가 소요되어 시간 측면에서는 k-means가 좋았다. **결론적으로 k-means는 빠르고 좋은 결과를 얻을 수 있으나 데이터 신뢰도가 떨어지고, k-medoids는 데이터를 신뢰할 수 있지만 느리고 결과가 좋다고 단정하기 어렵다. 또한 거리 보정은 매우 주의해야 하며, 거리 보정을 잘못할 경우 단순히 정규화한 것보다 나쁜 결과를 얻을 수 있다.**



total result with 0.3% rate of same result

| battPow | Bluetooth | clockSpd | DualSim | frontCamPixel | 4G | Memory | depth | weight | cores | camPixel | pixelDwn | pixelAcr | ram | scrDwn | scrAcr | battLife | 3G | Touch | Wifi |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1383.04 | 55.0% | 1.686 | 61.5% | 3.84 | 0.0% | 35.16 | 0.464 | 133.94 | 4.45 | 9.51 | 783.36 | 1395.88 | 3447.01 | 12.64 | 5.93 | 12.17 | 0.9% | 45.0% | 56.0% |
| 769 | Y | 2.9 | Y | 0 | N | 9 | 0.1 | 182 | 5 | 1 | 248 | 874 | 3946 | 5 | 2 | 7 | N | N | N |
| 1579 | Y | 0.5 | Y | 0 | N | 5 | 0.2 | 88 | 7 | 9 | 1358 | 1739 | 3532 | 17 | 11 | 12 | N | N | Y |
| 596 | N | 2.1 | Y | 9 | N | 64 | 0.8 | 111 | 8 | 15 | 885 | 1854 | 3238 | 16 | 13 | 10 | N | N | N |
| 1866 | N | 1.4 | N | 0 | N | 30 | 0.5 | 182 | 3 | 0 | 108 | 1781 | 3834 | 16 | 11 | 8 | N | N | N |

그림 5

이 데이터는 군집화의 **정답이나 이와 비슷한 무언가를 찾을 수 없다.** 답이 정해져 있지 않은 데이터이기도 하고, 답을 준다 해도 이에 해당하는 알고리즘을 짤 수 없으니 이 군집화의 결과가 옳은지 틀린지 확인할 방법이 없다. 맨 앞 데이터 활용 방안에 언급했듯 실제 군집화로 얻은 데이터 중 따라 만들기 만만한 하나를 골라 출시해 보고 판매량으로 평가할 수밖에 없다. 다만 실제 제작을 들어가려면 스펙 하나를 확실히 정해야 하는데, wifi 지원 여부, 터치스크린 지원 여부처럼 실제로는 0과 1 두 가지 선택만을 해야 하는데, 0.45 같은 애매한 결과를 보고 판단하기 어렵다면 k-means보다는 k-medoids가 더 좋은 선택이 될 것이다. 특히 k-medoids에서 나온 스펙을 그대로 따라 제작하면 이미 선례가 있었으므로, 미래(판매량)에 대한 어느 정도의 확신을 가지고 갈 수 있다. 애초에 이번 군집화의 목적이 이전 사례를 따라 하는 것이므로, **이 상황에서는 k-means보다는 k-medoids가 가치가 높고 성능이 더 좋다고 볼 수 있다. 따라서 최종으로 가중치 없이 center가 4개인 k-medoid를 선택하는 것이 바람직하다.**