사용자 만족도 예측 프로젝트

전현욱

1. 프로젝트 개요

20000개의 만족도 조사 응답 데이터 세트가 있다. 6가지 특성에 따라서 사용자의 특성을 구별하고 만족도 조사의 class는 "만족"과 "불만족"이 있다. 이 데이터를 사용해서 새로운 사용자의 만족도를 예측한다. k-NN알고리즘을 이용해서 새로운 사용자에 대해 만족과 불만족을 분류한다.

1. 데이터 세트를 9:1로 나누어서 Training data와 Testing data로 사용한다. 각 데이터는 배타적으로 나누어서 사용하며, 랜덤으로 선택된 10세트에 대해서 실험한다.

2. k-NN알고리즘을 직접 코드로 구현하고, 테스트 데이터에 대해 분류 결과를 도출한다. 분류기의 성능은 훈련/테스트 데이터세트의 조합을 이용하여 측정하고 합산한다.

2. 프로젝트 구현 환경 및 실행방법

2.1 사용언어: Java

2.2 실행 방법

1번, 2번, 3번에 따라서 수행한다. 단 2, 3번에 대해서는 콘솔에 bestK에 관한 결과는 도출되지 않고 result.txt파일에 저장된다.

1. 이클립스 [File] -> Open Project from file Systems -> Directory -> 프로젝트 선택 -> 실행
2. 프로젝트 폴더 -> command에 java -jar knn.jar 입력 -> 실행됨.
3. knn.exe를 실행 ->

2.3 실행 조건

자바 1.8버전 이상이 설치되어 있어야 한다. 자바가 설치되어 있지 않다면 제공한 knn.exe파일을 실행시킨다. 훈련, 테스트 데이터 셋은 프로젝트 하위폴더인 classification 폴더에 존재해야 한다.

2.4. 전체 수행시간

k값을 계속 증가시키면서 수행하기 때문에 한 데이터셋에 대해서 약 30초가량의 시간이 필요하다. 10개에 대해서 수행하면 대략적으로 5~6분 정도가 걸리게 될 것이다.

3. 구현 내용

자바 클래스 디자인

이번 프로젝트로 구현한 자바 프로젝트에는 5개의 클래스가 존재한다.

- **knn:** 이번 프로젝트의 메인 프로젝이다. k-NN알고리즘을 수행하며, k값을 증가시키면서 10개의 데이터셋에 대해서 자동으로 결과를 도출한다.
- **FileManager:** 훈련 데이터 및 테스트 데이터 csv파일을 읽어오기 위한 메소드가 정의되어 있다. 또한 최종적으로 <학번>.csv파일을 생성하기 위한 메소드도 정의되어있다. 행이 2000개인 테스트 데이터셋 10개에 대한 클래스 예측값을 저장하기 때문에 <학번>.csv파일은 20000개의 row를 가지게 된다.
- **Record:** double[]c attributes, int classLabel 변수가 존재한다. 데이터의 class를 분류할 때 'satisfied', 'unsatisfied'값을 각각 1, 0으로 변경한 뒤에 사용하기 위해서 int형 변수를 사용했다.
- **TrainRecord:** Record를 상속한 자식 클래스이다. 거리를 저장하기 위한 `double distance` 변수를 선언하였다.
- **TestRecord:** Record를 상속한 자식 클래스이다. knn알고리즘이 테스트 데이터에 대해 예측한 class값을 저장하기 위해 `int predictedLabel`변수를 선언하였다.

4. k-NN 알고리즘

4.1. 전체 알고리즘 흐름

1쌍의 훈련, 테스트 데이터셋에 대해서 k를 1,3,5...,29까지 증가시키면서 각 데이터 셋과 k값에 대한 정확도, 수행시간을 저장한다. 1개의 훈련, 데이터셋에 대해서 k값을 모두 적용해 보았을 때, 더 이상 정확도가 늘어나지 않는 k값을 최적의 k값으로 설정한다(`int bestK`). 그리고 그 k

값에 대한 class 분류결과를 "20176867.csv"파일에 저장한다. 이 과정을 10번 반복하여 최종적으로 10개의 데이터셋에 대한 class분류결과를 도출하고, 콘솔창에서 각 훈련, 테스트 데이터 쌍에 대한 최적의 k값 및 정확도를 출력한다.

4.2. Z-score normalization

주어진 데이터 셋을 관찰하면 6개의 속성값의 범위가 모두 제각각이다. 즉 feature의 데이터 스케일이 심한 차이가 나는 상황이다. 모든 데이터 포인트가 비슷한 정도의 스케일로 반영되도록 하기를 원해서 정규화를 도입했다. 데이터의 규모가 크고 이상치(outlier)의 존재 가능성을 염두에 두고 Z-score 정규화를 수행하였다.

각 훈련, 데이터 쌍의 2만개의 데이터에 대해 "(Xi – 평균) / 표준편차" 를 수행해서 모든 feature에 대해 비슷한 스케일이 되도록 처리 과정을 거쳤다. 아래는 관련 코드이다.

```
//z-score-normalize trainingSet
double[] avg = new double[6];
double[] std = new double[6];
for (int i=0; i<FileManager.NumOfAttributes; i++) {
    long total=0;
    for (int j=0; j<FileManager.NumOfSamples; j++) {
        total += trainingSet[j].attributes[i];
    }
    avg[i] = total / FileManager.NumOfSamples; //평균

    double temp=0;
    for (int j=0; j<FileManager.NumOfSamples; j++) {
        temp += Math.pow((trainingSet[j].attributes[i] - avg[i]),2);
    }

    double var = (double)temp / FileManager.NumOfSamples; //분산
    std[i] = Math.sqrt(var); //표준편차

    //z-scoring
    for(int j=0; j<FileManager.NumOfSamples; j++) {
        trainingSet[j].attributes[i] = (trainingSet[j].attributes[i] - avg[i]) / std[i];
    }

}
```

4.3. k-nearest neighbor 계산

각각의 테스트 데이터에 대해 훈련데이터 내에서 k개의 이웃을 찾았다. 이 중에서 빈도수가 더 높은 클래스(satisfied, unsatisfied)를 결과값으로 갖게 된다. 거리를 측정할 때에는 Euclidian distance를 이용하였다. 아래는 관련 코드이다.

```java
public static double calcDistance(Record training, Record test) {
    int numOfAttributes = training.attributes.length;
    double dist = 0;

    for(int i = 0; i < numOfAttributes; i ++){
        dist += Math.pow(training.attributes[i] - test.attributes[i], 2);
    }

    return Math.sqrt(dist);
}
```

Euclidian distance를 구하는 메소드이다. 6개의 속성에 대해서 각각 distance를 구한 뒤에 더한 값을 제곱근에 넣어서 계산한다.

```java
// test데이터에 대해 k-nearest neighbor계산
static TrainRecord[] findKNearestNeighbors(TrainRecord[] trainingSet, TestRecord testRecord,int K){
    int NumOfTrainingSet = trainingSet.length;
    TrainRecord[] neighbors = new TrainRecord[K];

    //초기화
    int index;
    for(index = 0; index < K; index++){
        trainingSet[index].distance = calcDistance(trainingSet[index], testRecord);
        neighbors[index] = trainingSet[index];
    }


    //나머지 데이터에 대해서 k이웃 구하기
    for(index = K; index < NumOfTrainingSet; index ++){
        trainingSet[index].distance = calcDistance(trainingSet[index], testRecord);
        // 배열에서 distance가 가장 큰 인덱스 구하기
        int maxIndex = 0;
        for(int i = 1; i < K; i ++){
            if(neighbors[i].distance > neighbors[maxIndex].distance)
                maxIndex = i;
        }
        // 현재 계산한 거리와, 배열에서의 가장 큰 distance값 비교 -> 이웃 업데이트
        if(neighbors[maxIndex].distance > trainingSet[index].distance)
            neighbors[maxIndex] = trainingSet[index];
    }

    return neighbors;
}
```

k개의 이웃을 구하는 부분이다. 훈련, 테스트 데이터셋, k값을 인자로 받은 뒤 calcDistance를 호출해서 거리를 얻어내고 가까운 이웃을 찾아나간다.테스트 데이터 1개가 들어올 때마다 distance를 계산하고, 기존의 neighbor배열을 순회하면서 distance를 비교한다. 만일 새로 들어온 데이터의 distance가 더 작다면 neighbor배열을 업데이트한다. 최종적으로 k개의 이웃을 반환한다.

4.4. classification

```java
// neighbor사용해서 class 구하기
static int classify(TrainRecord[] neighbors){
    HashMap<Integer, Double> map = new HashMap<Integer, Double>();
    int num = neighbors.length;
    System.out.println(num);

    for(int idx = 0; idx < num; idx ++){
        TrainRecord temp = neighbors[idx];
        int key = temp.classLabel;

        if(!map.containsKey(key)) {
            map.put(key, 1 / temp.distance);
        }
        else{
            double value = map.get(key);
            value += 1 / temp.distance;
            map.put(key, value);
        }
    }
```

```java
    // 좀 더 가까운 클래스 선택
    double maxSimilarity = 0;
    int returnLabel = -1;
    Set<Integer> labelSet = map.keySet();
    Iterator<Integer> it = labelSet.iterator();

    //해시맵을 순회 -> 가장 weight가 큰 key가 class가 됨.
    while(it.hasNext()){
        int label = it.next();
        double value = map.get(label);
        if(value > maxSimilarity){
            maxSimilarity = value;
            returnLabel = label;
        }
    }

    return returnLabel;
}
```

위는 classification 코드이다. 위에서 구한 이웃들을 이용해서 class를 판별하는 과정이다. 각 이웃들에 대해서 HashMap에 <class, weight>를 저장한다. weight로는 1/distance를 사용하였다. 만일 K=5일 때, 이웃의 class가 A,A,B,B,B이고, distance가 1,2,4,6,7 이라고 해보자. 일반적인 상황에서는 B를 클래스로 판별할 것이다. 그렇지만 실제로는 A로 결정하는 것이 좀 더 타당하다. 왜냐하면 A클래스가 테스트 데이터와 훨씬 가깝게 존재하기 때문이다. 그렇기 때문에 각 이웃에 대해서 가중치를 주는 방식을 선택하였다. 좀 더 정확성을 높일 수 있게 된다.

HashMap을 생성한 후에, 프로그램은 전체 HashMap을 탐색해서 가장 큰 weight를 갖는 클래스를 선택하게 된다(예측값 결정).

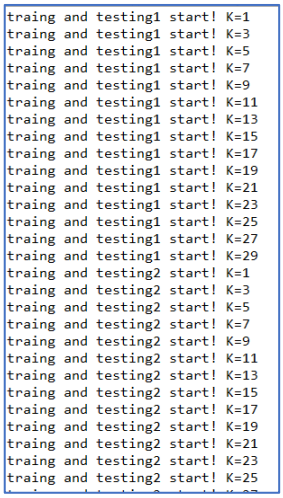
4.5. 메인 메소드

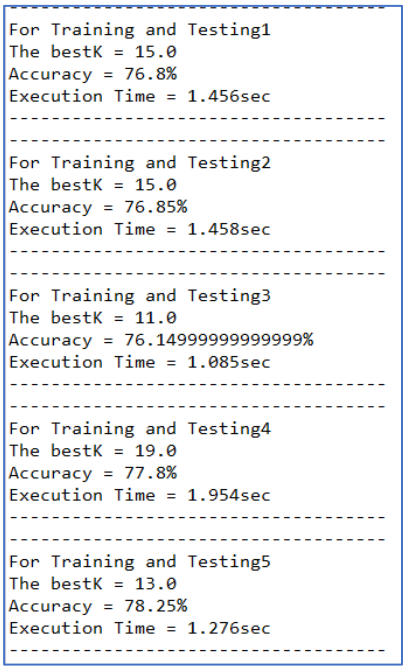
10개의 훈련, 테스트 데이터셋에 대해서 k값을 1에서 29까지 증가시키면서 최적의 k값을 찾아낸다. 그 뒤 최적의 k값에 대한 알고리즘 정확도 및 수행시간을 출력한다. class분류 결과는 학번.csv파일에 저장된다. 세부적인 수행 내용에 대해서는 result.txt파일에 따로 저장해둔다. 필요시 참고하면 된다. 아래는 관련 코드이다.

```java
public class knn {
    static double[] accuracyList = new double[15]; //1개의 테스트에 대해 15개의 k를 실험해서 정확도 저장
    static double[] executionTime = new double[15];
    static double[][] performance = new double[10][3]; //각 테스트의 최적k일 때의 정확도 저장
    static int kNum = 15; //15개의 k사용 1~30사이의 홀수

    public static void main(String[] args) throws IOException {
        PrintWriter pw = new PrintWriter("classification\\result.txt"); //파일에 결과 저장
        int index=0;
        //knn알고리즘 수행
        for(int i=0; i<10; i++) {
            int idx=0;
            for(int k=1; k<kNum*2; k=k+2) {

                System.out.println("traing and testing"+(i+1)+" start! K="+k);
                pw.println("traing and testing"+(i+1)+" start! K="+k);
                final long startTime = System.currentTimeMillis();
                double accuracy = knn_result("classification\\training"+Integer.toString(i+1)+".csv" ,"classification\\testing" + Integer.toString(i+1)+".csv", k); //알고리즘 수행
                final long endTime = System.currentTimeMillis();

                double execTime = (endTime-startTime)/(double)1000;
                executionTime[idx] = execTime; //1~29까지 k에 대한 실행시간 저장
                accuracyList[idx] = accuracy; //1~29까지 k에 대한 정확도 저장
                idx++;
                //System.out.println("Total excution time: "+execTime+" sec\n");
                pw.println("The accuracy is "+accuracy+"%");
                pw.println("Total excution time: "+execTime+" sec\n");
            }

            // 최적의 k값 찾기
            int bestK=0;

            for(int j=0; j<kNum; j++) {
                if(Double.compare(accuracyList[j], accuracyList[j+1])>0) { // k를 늘렸는데 정확도가 떨어진 경우
                    bestK = j*2+1;
//                  System.out.println("bestK: "+bestK);
//                  System.out.println("accuracy: "+accuracyList[j]+"%");
//                  System.out.println("execTime: "+executionTime[j]+"sec\n\n");
                    pw.println("bestK: "+bestK);
                    pw.println("accuracy: "+accuracyList[j]+"%");
                    pw.println("execTime: "+executionTime[j]+"sec\n\n");

                    performance[index][0] = bestK;
                    performance[index][1] = accuracyList[j];
                    performance[index][2] = executionTime[j];

                    knn_store("classification\\training"+Integer.toString(i+1)+".csv" ,"classification\\testing" + Integer.toString(i+1)+".csv", bestK);
                    index++;


                    break;
                }
            }
        }
```

```java
// 최종결과 도출
for(int num=0; num<10; num++) {
    System.out.println("-------------------------------------");
    System.out.println("For Training and Testing"+(num+1)+"");
    System.out.println("The bestK = "+performance[num][0]);
    System.out.println("Accuracy = "+performance[num][1]+"%");
    System.out.println("Execution Time = " + performance[num][2] + "sec");
    System.out.println("-------------------------------------");
}

System.out.println();
```

5. 수행 결과





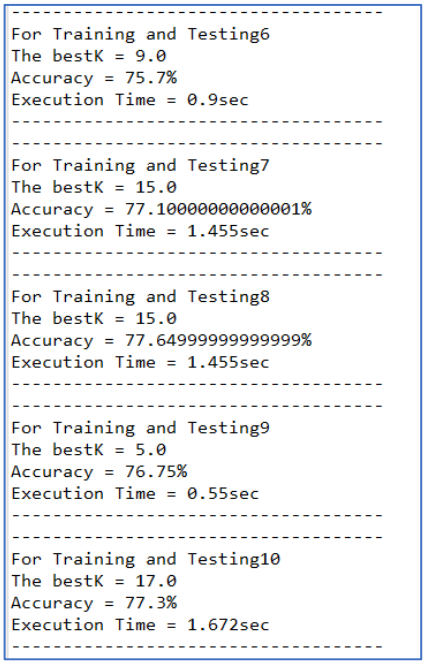
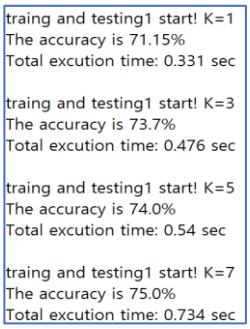
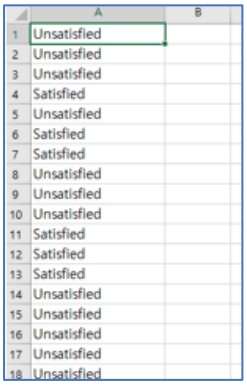


Figure1 **Figure2** **Figure3**

위의 사진은 프로그램 수행 후의 콘솔 결과창이다. Figure1에서 훈련 및 테스트 데이터셋에 대해서 k값을 증가시키면서 최적의 k값을 찾고 있는 것을 확인할 수 있다. 이후 모든 데이터셋에 대해 이 과정이 끝나면, 각각의 데이터셋에 대해서 최적의 k값과 정확도, 수행시간을 출력해준다.

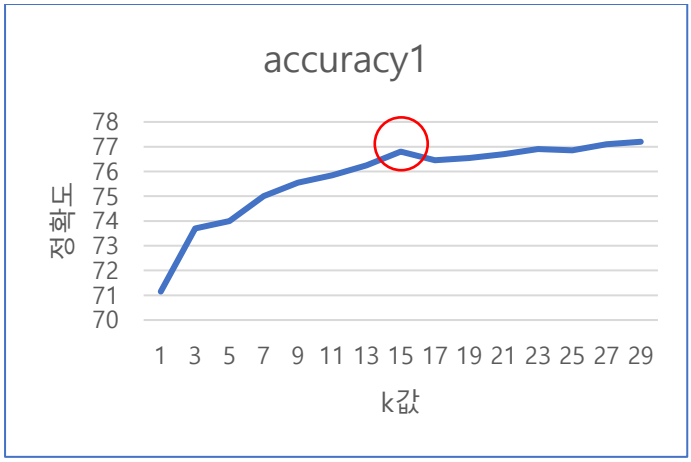


콘솔창에 출력되지 않은 모든 k에 대한 결과값은 classification폴더의 result.txt에 저장된다. 왼쪽의 사진은 텍스트 결과의 일부를 캡쳐한 것이다. 이렇게 각 테스트의 각 k값에 대해서 결과가 저장된다.



왼쪽의 사진은 "20176867.csv"에 출력된 결과를 일부 캡쳐해서 가져온 것이다. 1~2000줄까지는 첫번째 테스트 데이터, 2001~4000줄까지는 두번째 테스트 데이터에 대한 classfication결과를 저장한다. 즉 파일 전체는 20000줄을 가지게 된다.

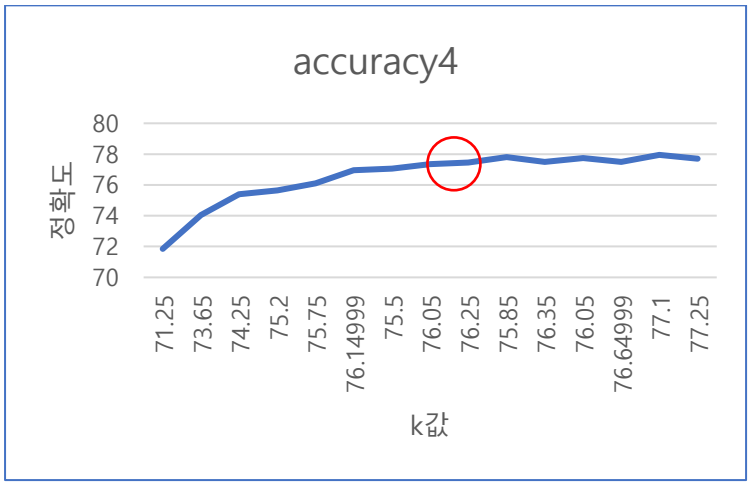
6. 성능 그래프 분석




```
For Training and Testing1
The bestK = 15.0
Accuracy = 76.8%
Execution Time = 1.465sec
```

k	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29
accuracy1	71.15	73.7	74	75	75.55	75.85	76.25	76.8	76.45	76.55	76.7	76.9	76.85	77.1	77.2

위의 그래프는 1번에서 10번 데이터셋 중에서 1번 데이터셋에 대해 수행한 결과이다. 빨강 원 부분을 보면 k값이 증가했을 때 오히려 정확도가 떨어지는 것을 알 수 있다. 이번에 수행한 알고리즘은 이런 부분에서 최적의 k값을 선택한다. 오른쪽 결과를 보면 최적의 k값은 15, 이때의 정확도와 수행시간을 확인할 수 있다.




```
bestK: 15
accuracy: 76.85%
execTime: 1.469sec
```

k	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29
accuracy	71.6	73.65	74.35	75.15	75.6	75.7	76.7	76.85	76.55	76.7	77.1	77.1	76.8	77	76.9

위의 그래프는 2번 데이터셋에 대한 결과이다. 빨강 원부분을 보면 k값이 15에서 17로 증가할 때 정확도가 떨어진다.

이런 방식으로 테스트를 수행하였다.

K값에 따른 평균 정확도 – 정규화 사용

k	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29
accuracy	71.38	73.70	74.9	75.59	76.18	76.58	76.89	77.18	76.87	77.22	77.21	77.36	77.36	77.56	77.54

k값이 증가할수록 평균적으로는 정확도가 올라가는 것을 확인하였다. 하지만 k값이 만약 이를 넘어서서 너무 커지게 된다면 overfitting이 될 가능성이 존재한다. 이번 경우에는 데이터 양이 많은 편이기 때문에 k가 29가 될때까지는 그런 현상은 발생하지 않은 것으로 보인다. K=7을 넘어서면 수행시간에 비해서 얻을수 있는 정확도의 향상이 크지 않은 것으로 보아, 적절한 K를 선택하는 것이 중요한 문제라고 할 수 있다.

K값에 따른 평균 정확도 – 정규화 사용 안함

k	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29
accuracy	62.05	62.9	63.16999	63.76	63.863	64.305	64.329	64.905	65.02	65.1285	65.145	65.235	64.98	65.42	65.59

정규화를 사용하지 않았을 때의 k값에 따른 평균 정확도이다. 평균적으로 약 10%이상의 예측률의 차이를 확인할 수 있다. 이를 보아서 데이터의 스케일이 일정하지 않을 때 정규화를 하는 것은 필수로 보인다. 데이터를 정규화하는 것이 적절한 K를 선택하는 것 이상으로 중요하다 점을 시사한다.

7. 결론

k-NN 알고리즘을 구현해보고, 20000개의 데이터셋에 대해서 어떻게 작동하는지 알아보았다. K값의 변화에 따른 정확도의 변화를 확인하였으며, 그에 따른 수행시간도 계산해보았다. 이번 프로젝트는 매우 중요한 점을 시사한다. 바로 K값의 변화에 따른 정확도의 개선보다 데이터 정규화를 통한 정확도의 개선점이 더 크다는 점이다.