PART 1: Implementation of Basic K-Nearest Neighbor Classification algorithm

Introduction

The k-nearest neighbor algorithm is a method for classifying objects based on closest training objects: an object is classified according to a majority vote of its neighbors. Here the basic KNN algorithm is implemented with the assistance of object-oriented design, and it supports different metric methods and different choices of K. The rest of Part 1 is organized as follows: Implementation describes the class design and talks about the details of how KNN is implemented. The experimental result is shown in Evaluation. In Discussion, items including running time, the choice of K, the choice of metric type and normalization are discussed. Finally, a summary which concludes part 1 is done in Conclusion.

Implementation

Class design

- **Record**: a basic class which simulates records containing a double array for attributes and an integer for label. Also, this is the argument type for Metric interface which defines a method to get distance between two records.

- **TrainRecord**: a subclass of Record with a double type number storing its distance to test record as an addition.

- **TestRecord**: a subclass of Record with an integer storing the predicted label worked by KNN algorithm as an addition.

- **Metric**: an interface defining a method which calculates the distance between two records.

- **CosineSimilarity**: a class which implements Metric and provides the way to get the cosine similarity between two records. Note that at the end of getDistance() method, 1 / cosineSimilarity is returned in order to be unified with Euclidean distance and L1 distance. The reason why we do this is because that the distance becomes larger when the cosine similarity is smaller. The case when cosine similarity equals to 0 is also taken into consideration.

- **L1Distance**: a class which implements Metric and provides the way to calculate the L1 distance between two records.

- **EuclideanDistance**: a class which implements Metric and provides the way to calculate the Euclidean distance between two records.

- **FileManager**: a class providing static methods for three purposes: reading training data, reading testing data and outputting predicted labels.

- **KNN1**: the main class implementing the KNN algorithm. Details will be discussed in the next section.

Algorithm

In this implementation, KNN is performed according to the below two phases:

1. **Find K nearest neighbors**
   For a specific test record, an array with size K is created to keep the K nearest neighbors, and the way to achieve this goal is to go through the whole training set and update the array if there is a training record whose distance is smaller than the largest distance in the original array. At the end of this process, the K nearest neighbors are found. The below Java codes demonstrate how it is implemented:

```
// Find K nearest neighbors of testRecord within trainingSet
static TrainRecord[] findKNearestNeighbors(TrainRecord[] trainingSe,
TestRecord    testRecord,int K, Metric metric) throws Exception{
    int NumOfTrainingSet = trainingSet.length;
    if(K > NumOfTrainingSet){
      throw new Exception("K is larger than the length of training
         set!");
    }

   //Create an array with size K to store K nearest neighbors
   TrainRecord[] neighbors = new TrainRecord[K];

    //initialization, put the first K trainRecords into the above array
    int index;
    for(index = 0; index < K; index++){
      //calculate the distance between TrainingRecord and TestRecord
      trainingSet[index].distance =
         metric.getDistance(trainingSet[index], testRecord);
      neighbors[index] = trainingSet[index];
    }

    //go through the remaining records in the trainingSet to find K
    //nearest neighbors
    for(index = K; index < NumOfTrainingSet; index ++){
      //calculate the distance between TrainingRecord and TestRecord
       trainingSet[index].distance =
          metric.getDistance(trainingSet[index], testRecord);

      //get the index of the neighbor with the largest distance to
      //testRecord in array neighbors
```

```
        int maxIndex = 0;
        for(int i = 1; i < K; i ++){
           if(neighbors[i].distance > neighbors[maxIndex].distance)
              maxIndex = i;
        }

        //add the current trainingSet[index] into neighbors when its
        //distance is smaller than the distance of neighbors[maxIndex]
        //By this way, the array keeps the K nearest training records
        if(neighbors[maxIndex].distance > trainingSet[index].distance)
           neighbors[maxIndex] = trainingSet[index];
        }

        return neighbors;
}
```

2. **Classify according to weights**
   After the K nearest neighbors are found, a HashMap is created to keep the pairs <label, weight> by going through all the neighbors. If the HashMap happens to meet a new label, <label, 1 / distance> is directly added into the map. Otherwise, an updated version which adds the original weight with 1 / distance is put into the map. After the HashMap is correctly constructed, this program will go through the whole HashMap and find the label associated with the largest weight. Below are the Java codes which demonstrate the implementation:

```
// classify the class label by using neighbors
static int classify(TrainRecord[] neighbors){
        //construct a HashMap to store <classLabel, weight>
        HashMap<Integer, Double> map = new HashMap<Integer, Double>();
        int num = neighbors.length;

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

              //if this classLabel does not exist in the HashMap, put
              //<key, 1/(temp.distance)> into the HashMap
              if(!map.containsKey(key))
                    map.put(key, 1 / temp.distance);

              //else, update the HashMap by adding the weight associating
              //with that key
              else{
                    double value = map.get(key);
                    value += 1 / temp.distance;
                    map.put(key, value);
              }
        }
        //Find the most likely label
        double maxSimilarity = 0;
        int returnLabel = -1;
        Set<Integer> labelSet = map.keySet();
        Iterator<Integer> it = labelSet.iterator();

        //go through the HashMap by using keys
        //and find the key with the highest weight
        //the key with the highest weight is the returned class label
        while(it.hasNext()){
            int label = it.next();
            double value = map.get(label);
            if(value > maxSimilarity){
                maxSimilarity = value;
                returnLabel = label;
            }
        }

        return returnLabel;
}
```

Evaluation

Dataset	iris	glass	vowel	vehicle	letter	DNA
K	1	1	3	3	3	5
metric_type	1	0	2	1	0	2
Accuracy	96%	67.29%	94.95%	66.19%	94.32%	80.29%
Time(seconds)	0.15	0.14	0.39	0.13	59.46	10.61

Discussion

- **Running time is O(m*d*n), m = # examples in training set, d = # dimensions, n = # examples in testing set**
  For instance, there are 10000 training samples, 16 dimensions and 10000 testing samples in letter dataset. As a result, the running time for letter dataset is significantly larger than the others.

- **The choice of K**
  For DNA data set, because the data is made up of 1s and 0s, it is very likely that each test record may have several training records whose distances to that test record are the same. As a result, a large K will generate a relatively better result for DNA case. In order to better illustrate the above idea, the experiment with different K is done, and the result is shown as follows:

K	2	3	4	5
dna accuracy	71.75%	78.41%	79.03%	80.29%

However, for some datasets, a relatively larger K may not lead to better results. For instance, the experimental with different K is done on dataset vowel, and the result is shown as follows:

K	1	2	3	4
vowel accuracy	96.36%	96.36%	94.95%	93.54%

As the above table suggests, the accuracy decreases with K increasing. In all, the choice of K depends on the characteristics of the dataset, and it is better to test different K in order to achieve good results.

- **The choice of metric type**
  Different metric methods also have impacts on performing classification tasks. Here the vehicle data set is tested, and the following result is worked out:

metric_type	CosineSimilarity	L1Distance	EuclideanDistance
vehicle accuracy	65.48%	66.19%	64.30%

According to the above table, L1 distance achieves the best result while Euclidean distance performs the worst. Hence, the metric type should be set according to the characteristics of the target data set such that optimal results can be obtained.

- **Normalization**
  Normalization is also discussed here, and the experiment on glass and vehicle data sets is conducted to check how their accuracies change before and after normalization. Also, please note that the testing set is normalized according to the maximum and minimum values in training set. The following table shows the result:

	Without Normalization	With Normalization
glass accuracy	67.29%	64.49%
vehicle accuracy	66.19%	68.56%

For vehicle data set, the accuracy increases by 2.37% after normalization. However, such increase does not apply to glass data set. The accuracy of glass data set decreases by 2.8% after normalization. As a result, whether to normalize or not also depends the target data set. But in general, normalization helps to increase the accuracy of classification.

Conclusion

The k-nearest neighbor algorithm, which classifies objects according to the majority vote of its neighbors, is successfully implemented. The details about how k-nearest neighbor algorithm is implemented are also presented in this report. Furthermore, items such as the running time, the choice of K, the choice of metric type and normalization are discussed according to experiment results. The accuracy varies with different K, different metric types and the option of normalization. In order to achieve good results, proper setting based on the characteristics of data set is needed.