华东师范大学计算机科学技术系上机实践报告

课程名称：知识分析与应用基础	**年级**：2016 级	**上机实践成绩**：
指导教师：杨燕、贺樑	**姓名**：张臻炜	**创新实践成绩**：
上机实践名称：KNN	**学号**：10165102154	**上机实践日期**：
上机实践编号：1	**组号**：	**上机实践时间**：

1 实验内容

1.1 名词解释

KNN 是一种基于实例的学习，或者是局部近似和将所有计算推迟到分类之后的惰性学习。

KNN 算法是所有的机器学习算法中最简单的之一。

1.2 基本原理

基于实例的学习方法中最基本的是 KNN 算法。

这个算法假定所有的实例对应于 N 维欧氏空间 Ân 中的点。

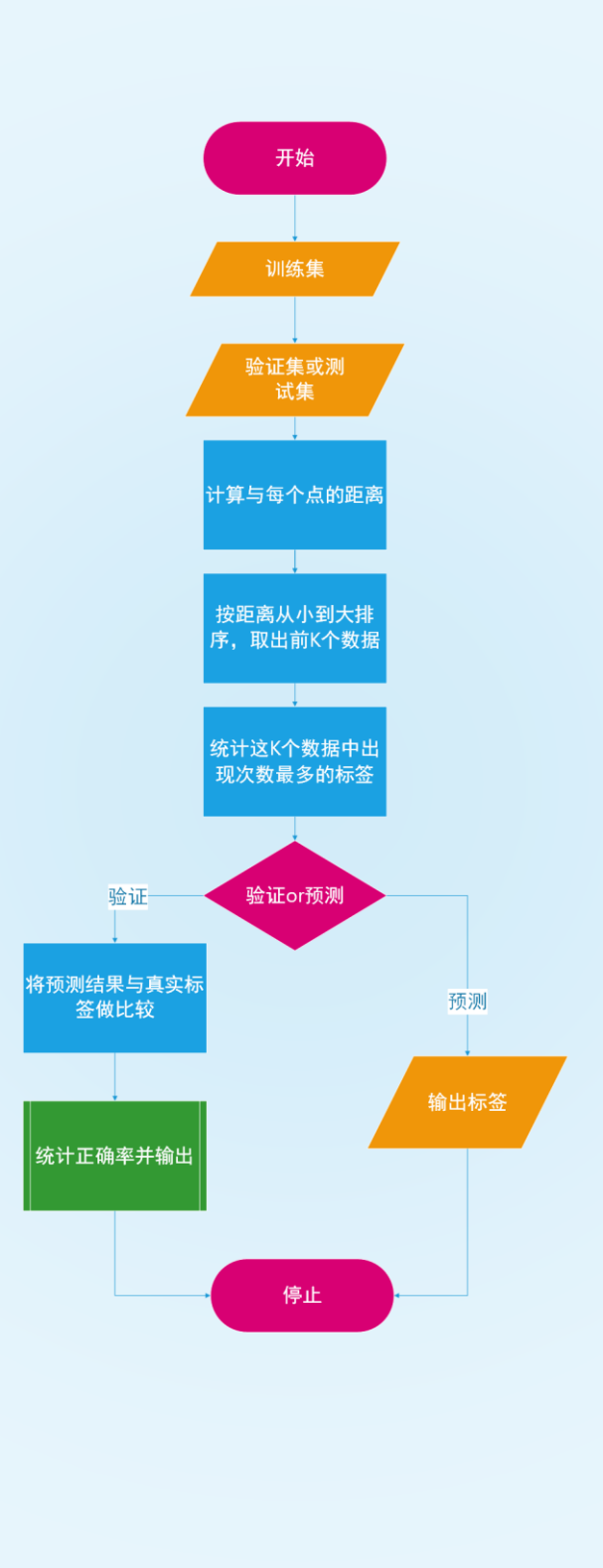
一个实例的最近邻是根据标准欧氏距离定义的。

在 KNN 分类中，输入包含特征空间中的 K 个最接近的训练样本，输出是一个分类族群。

一个对象的分类是由其邻居的“多数表决”确定的，K 个最近邻居中最常见的分类决定了赋予该对象的类别。

特别的，若 K=1，则该对象的类别直接由最近的一个节点赋予。

1.3 算法流程


开始
训练集
验证集或测试集
计算与每个点的距离
按距离从小到大排序，取出前K个数据
统计这K个数据中出现次数最多的标签
验证or预测
验证
预测
将预测结果与真实标签做比较
输出标签
统计正确率并输出
停止

1.4 需要求解的问题

给定一个训练集，包含若干数据，每个数据有 4 个向量和 1 个标签；

给定一个测试集，测试集与训练集格式类似，包含若干数据，每个数据有 4 个向量，需要用 KNN 算法预测测试集的每个数据的标签，即所属类别。

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
```

1.5 实验目的

完成测试集的标签预测。

2 算法实现

2.1 验证集的随机选取

在本题中，只有训练集没有验证集，这种情况下应该从训练集随机抽取一部分数据作为验证集。

常见的划分是 80%的数据作为训练集、20%的数据作为验证集。

同时考虑到，在一般场合，测试集和验证集是分开给出的，而不是需要随机从训练集抽取，所以为了程序的可扩展性，我没有选择在读入训练集的时候就随机 20%存入验证集，而是将训练集完完整整读入。

这种情况下，只需要分别指定训练集和验证集的文件名就可以开始训练了，而不会导致读入训练集的时候硬编码地随机选取了 20%的验证数据。

因此，对于只有训练集的数据（例如本题），需要额外（并且也只需要额外）写一个函数来完成随机抽取验证数据到验证集并将验证数据从训练集删除的操作，不会影响其他部分的代码。

这个操作对于本题来说不是核心部分，从略，只给出我抽取出的两个 TXT 文件。

需要注意的是，随机抽取的验证集不同，准确率会有浮动，大约在 91%~96%之间（偶尔也可能是 100%正确）。

2.2 Data 类的实现

考虑到封装，Data 类向量的个数做了静态数据域，这样就只要改这一个地方就可以了，for 循环什么都不用改，都做了可移植性的处理。

Data 当然最好实现 Comparable 接口，最最好还是实现了泛型的 Comparable<T>接口。

我没有，偷懒了，需要不同的比较方法的时候自己 new 一个 Comparator 传进去吧。

Data 类的数据域成员，以及各方法的含义（包括接受参数的含义和返回值的含义），我都以 Doc 的形式注释在了源代码中。

对于部分比较关键的实现语句，我也在源代码中给出了注释。

我相信助教你看源代码是看得懂的，看不懂再来问我吧。

```
/**
 * 归一化
 *
 * @param raw
 *            原数据点
 * @param newValue
 *            归一化后的新值
 * @return 归一化后的数据点
 */
public static Data normal(Data raw, double[] newValue) {
    Data normal = new Data();
    // 依次赋归一化后的值，并复制标签
    for (int i = 0; i < NUMBER; ++i)
        normal.values[i] = newValue[i];
    normal.label = raw.label;
    return normal;
}
```

实现的时候遇到了一个问题是一开始总是输出 Iris-setosa，后来判断的原因是 Data 的构造器错了，直接 new 一个不带参数的会导致标签为 null，所以 keySet 就一直是空集，所以应该写一个静态的工厂类返回归一化后的数据，或 super 或直接调用别的构造器。

2.3 输入输出处理

没有硬编码，考虑到数据的可扩展性，double，考虑到标签的可扩展性，考虑String，考虑到数据数量级不一定一样，都做了归一化的处理。

归一化的处理需要计算极差等，这部分数学公式的解释从略。

只需要给定文件名，不需要关心如何读入的。

请阅读源代码和 Doc 了解具体的实现思路。

```
/**
 * 读入原始数据，将每个数据点构造成一个Data泪，保存ArrayList的形式
 *
 * @return 所有读入的数据
 */
public ArrayList<Data> input_raw() {
    ArrayList<Data> list = new ArrayList<Data>();
    while (in.hasNextLine()) {
        String[] s = in.nextLine().split(","); // 以逗号分隔，保存成字
        Data d = new Data(s); // 构造，并加入list
        list.add(d);
    }
    return list;
}

/**
 * 读入数据，并作归一化处理，将每个数据点构造成一个Data泪，保存ArrayList的
 *
 * @return 归一化后的所有数据
 */
public ArrayList<Data> input_normalized() {
    ArrayList<Data> raw = input_raw(); // 先读入原始数据
    ArrayList<Data> normalized = new ArrayList<Data>();
    // 遍历所有数据，找到最小和最大的值
    double[] mins = new double[Data.NUMBER];
```

2.4 KNN 核心代码

非常简单，按照上面流程图实现即可。

计算记录就是遍历。

排序就是直接调用 Collection 的 sort()。

统计次数用 Map，Key 和 Value 一一对应。

找出现次数最多的标签，直接遍历。

```java
/**
 * 预测
 *
 * @param data
 *             给定的数据，没有标签
 * @return 预测的标签，字符串形式
 */
public String predict(Data data) {
    // 遍历KNN中的每一个点，计算到给定点的距离
    for (Data d : list) {
        double dist = calculateDistance(data, d);
        d.setDistance(dist);
    }
    // 排序，排序的标准是距离从小到大，需要构造一个Comparator（因为没有实现Comparable接口）
    Collections.sort(list, new Comparator<Data>() {
        @Override
        public int compare(Data d1, Data d2) {
            if (d1.getDistance() < d2.getDistance())
                return -1;
            else if (d1.getDistance() == d2.getDistance())
                return 0;
            else
                return 1;
        }
    });

    // 统计出现次数最多的标签，直接用MAP来统计（红黑树）
    HashMap<String, Integer> map = new HashMap<String, Integer>();
    // 遍历距离最小的K个点，获取他们的标签，对应标签计数
    for (int i = 0; i < K; ++i) {
        Data d = list.get(i);
        String label = d.getLabel();
        if (map.containsKey(label)) {
            int count = map.get(label);
            map.put(label, count + 1);
        } else {
            map.put(label, 1);
        }
    }
```

2.5 其他

封装的非常好了，所以 Main 函数就非常简单。

最多就是处理一个输入输出。

至于 Handler，我只做了输入的解析，也就是说，用户不需要关心解析的过程，只需要知道，调用会返回一个 ArrayList。

至于输出，其实我本来是想直接打结果的，但是助教一定要输出格式和原来一样。

但是文件又不能又读又写，我读入总归是要读的，要 test 嘛，那么又不能写出，所以只能把答案放在另一个文件了。

我是 Windows，换行符是\r\n，但是助教指不定用什么操作系统，搞不好他那里打开的时候，Unix 和 Windows 换行符（LF 和 CRLF）不同，最后倒都是粘连在一起了。

所以，一方面我没有直接用\n 作为换行符，我是调用了 Java 虚拟机的 newLine()，这样就可以自动根据操作系统的版本决定是用什么作为换行符，另一方面，除了附上 ans.txt，我直接复制一份答案在文末。

3 进一步的优化

3.1 K 值的优化

3.1.1 优化方案

如何选择一个最佳的 K 值取决于数据。

一般情况下，在分类时较大的 K 值能够减小噪声的影响，但会使类别之间的界限变得模糊。

一个较好的 K 值能通过各种启发式技术来获取。

3.1.2 可行性与必要性讨论

可行，但是没有必要。

如果选择较小的 K 值，就相当于用较小的邻域中的训练实例进行预测，近似误差会减小，只有与输入实例较近或相似的训练实例才会对预测结果起作用，特别的，若 K=1，则该对象的类别直接由最近的一个节点赋予。换句话说，K 值的减小就意味着整体模型变得复杂，容易发生过拟合。

如果选择较大的 K 值，就相当于用较大邻域中的训练实例进行预测，近似误差会增大，不相似的训练实例也会对预测器作用，使预测发生错误，特别的，若 K=N，则此时无论输入实例是什么，都只是简单的预测它属于在训练实例中最多的累，模型过于简单，忽略了训练实例中大量有用信息。

鉴于本题数据规模只有 120，去除 20%的验证数据，数据规模不足以支撑启发式搜索自动调整 K 值。

事实上，对于本题来说，K=5、6、7……已经没有什么变化了。

3.2 权重的优化

3.2.1 优化方案

对 KNN 算法的一个显而易见的改进是对 K 个近邻的贡献加权，根据它们相对查询点 Q 的距离，将较大的权值赋给较近的近邻。

一种常见的做法是根据每个近邻与 Q 的距离平方的倒数加权这个近邻的“选举权”。

$$\hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

3.2.2 可行性与必要性讨论

可行，为每个点的距离增加一个权重，使得距离近的点可以得到更大的权重。

最简单的形式是返回距离的倒数（反函数）。

$$\frac{1}{Dx - Dy}$$

但是为了避免完全一样的点造成的权重无穷大，可以加上一个较小的常数项。

$$\frac{1}{(Dx - Dy) + (const)}$$

这种方法的潜在问题是，它为近邻分配很大的权重，稍远一点的会衰减的很快，会使算法对噪声数据变得更加敏感。

另一个常见的方法是，高斯函数，高斯函数比较复杂，但克服了前述函数的缺点。

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

但是没有必要。

这是我和助教存在争议、讨论很久的地方。

助教举出一个例子，K=3，距离当前点最近的是距离为 1，标签为 A，距离当前点第 2 近的距离为 102，标签为 B，距离当前点第 3 近的距离为 103，标签为

B，那么 K=3 条件下，显然出现次数最多的是第 2 个簇，也就是标签 B，但是事实上距离给定点距离最近的是标签 A，是不是应该预测为 A？

我给出如下若干反驳：

第一，如果是有效的训练集，训练集中不同标签的出现的数量应该是类似的，或者说至少在同一个数量级。

如果预测的标签确实是 A，那么，应该有很多的训练数据会有 A 这个标签，那么这些标签如果是合法的，一定是会距离当前点的距离为形如 2、3、5、8……

所以，给定的 K，出现次数最多的应该还是 A，上述情况不可能出现，因为不可能只有一个 A 距离当前点是最近的，其他都是另一个簇的。

第二，假设上述情况真的出现了，那么应该认为这个孤立点是噪声数据，应该被排除。

所以最合理的做法，其实是判断出现这种情况的时候，应该将预测点直接标记为噪声，但是这样做也有实现的困难。

（方法肯定是有的，遍历所有的 K 个点的标签，发现一个标签出现 1 次、其他出现了 K-1 次，那么这个就是噪声点，不过，你说不清到底距离多远才算远，K 足够大的时候，可能会出现很多簇，分别是 1 个、B 个、C 个，那就不好说了。距离多远才算远？1 和 100 算不算远？那 1 和 5 呢？）

第三，按照助教给出的建议，其实慢慢已经走向了另一个算法：聚类。例如层次聚类、密度聚类……

也就是，将需要预测的点，加入到整个测试集中，全部放在一起聚类，聚成一个一个的簇，然后看一下预测点在哪个簇中，得到答案。

然而这么做就是另一个算法了，这是 K-Means 算法，K-Means 和 KNN 是两个完全不同的算法。我认为这不是 KNN 的优化范畴，你总不能说优化着、优化着，就到了另一个地方，那干脆直接学另一个算法好了。

当然，K-Means 聚类我又不是没写过：

```java
public void solve() throws Exception {
    // 随机选择k个数据作为初始的中心，因为这k个不能重复，所以放在
    HashSet<Integer> set = new HashSet<Integer>();
    while (set.size() != k)
        set.add((int) (Math.random() * allData.size()));
    // 取出这k个数据，放到初始的中心ArrayList中
    ArrayList<Data> initialMeans = new ArrayList<Data>();
    for (int i : set)
        initialMeans.add(allData.get(i));
    ClusterK clusterK = null;
    // initialMeans为null说明上一次更新中心，没有任何一个中心发
    while (initialMeans != null) {
        // 用当前的中心构造一个ClusterK
        clusterK = new ClusterK(k, initialMeans);
        // 对于每一个数据，加入到距离它最近的那个中心所在的簇
        for (Data data : allData)
            clusterK.autoAdd(data);
        // 更新中心，直到稳定
        initialMeans = clusterK.updateMeans();
    }
    showResult(clusterK);
}
```

按照助教的权重优化，其实权重都不要了，甚至还能直接排除噪声点。

但是助教说，聚类时无监督了，标签就派不上用处。但是要知道，如果标签正确，那么聚类结果，属于同一个簇的，标签应该是一样的，或者说绝大部分是一样的，如果标签是乱的，那么只能说是标签本身是完全混乱的，所以，聚类一做，就不需要标签了，标签最多只是用来验证了。

所以，综上，KNN 的权重是可行的，也是有意义的，但是没有必要。

这只是把近的点变得更近，相对的更近，权重更大。

效果绝不明显。

总的来说，还是错在数据规模太小，规模大了，就好办了，例如对于我使用 MNIST 训练集构建的 BP 神经网络的权重调整：

```
            # 对于每一个神经元，计算误差，并对应的更新权重
            for i in range(neuronNumbers):
                weight = neuron.getWeight(i)  # 权重数组
                # 神经元的输出作为参数的反向传播的值乘上对应的误差
                error = errors[i] * self.backActive(outActivations[i])

                # 根据当前的误差和学习的速率，更新权重
                # 更新除了最后一个输入以外的所有的权重
                for j in range(inputNumbers):
                    if oldErrors is not None:  # 输入层没有前一层，需要特别判断不为 None
                        # 更新前一层的误差，加和为当前的误差和权重的乘积
                        oldErrors[j] = oldErrors[j] + error * weight[j]
                        # 根据反向传播的规则更新误差
                    weight[j] = weight[j] + error * self.learningRate * oldOutActivations[j]
                # 偏移量
                weight[inputNumbers] = weight[inputNumbers] + error * self.learningRate * self.BIAS
```

3.3 复杂度降低

因为存在排序，所以复杂度至少是O(NlogN)的。

其他的操作都是线性操作，都是O(K)或者O(N)。

Map（红黑树）是O(KlogK)，由于 K 远小于 N，所以不计。

所以忽略常系数，可以认为复杂度是近似的相当于排序的复杂度。

其实有方法可以降下来，因为我只关心前 K 个值，所以很容易可以做线性的处理，复杂度降低到O(KN)。

不过比较复杂，还是直接排序来的无脑。

而且，就算降下来了，因为上面说，K 远小于 N，甚至 K 远小于 logN，所以这样就只相差一个很小的常数$\frac{C1}{C2}$，优化意义不大。

3.4 正确率提高

需要注意的是，验证集的不同，准确率会发生变化。

同时，如果验证集过大，那么训练集数据就小，即样本规模小，这样的预测结果没有可参考价值。

如果验证集太小，那么验证的准确率容易受到极端数据的影响，偏向极大或者极小，没有参考价值。

归根结底还是数据规模不够呀，要是有 10000 个数据就好办了。

对于目前的情况，准确率已经很好了，但是还有提升的空间。

根据验证集的不同，准确率在 91%~96%浮动。

对于我附件上的这份训练和测试集，准确率是 93.75%。

应该可以更高，但是我觉得可以接受了，毕竟是简单的算法。

3.5 进一步思考

3.5.1 优点

简单。

3.5.2 缺点

KNN 算法的缺点是对数据的局部结构非常敏感。

原始朴素的算法通过计算测试点到存储样本点的距离是比较容易实现的，但它属于计算密集型的，特别是当训练样本集变大时，计算量也会跟着增大

4 实验结果

4.1 程序运行环境

JDK1.8。

没有包，编译时不需要指定包名，用的 default package。

文件名在 Main.java 源代码中修改，没有放在命令行参数，也没有用一个 Scanner 去读入。

（换言之，需要更换文件名，需要修改源代码并重新编译，我承认这是我偷懒了，当然你也可以直接把文件名改成我规定的文件名）

文件路径应该与程序工作目录处于同一层。

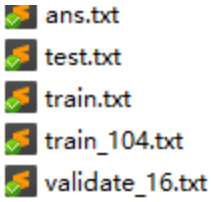
对于默认的 Eclipse，应该是 Project 所在目录，即与 bin、src 同级。

对于 InteliJ 或者其他 IDE，可能会是在 bin 目录下。

对于修改过配置文件的 IDE，遵循配置文件所规定的工作目录。

输入文件：train 是原始训练集，train_104 是我随机选取的 104 个训练数据，validate_16 是我随机选取的 16 个验证数据，test 是原始测试集。

输出文件：ans 是我得到的答案，格式与输入文件格式一致。



4.2 程序运行结果截图

```
Validation Correct Rate = 93.75%
Predict result has been stored in: ans.txt
```

```
5.0,3.5,1.3,0.3,Iris-setosa
5.0,3.5,1.6,0.6,Iris-setosa
5.1,3.8,1.9,0.4,Iris-setosa
4.8,3.0,1.4,0.3,Iris-setosa
6.3,2.5,5.0,1.9,Iris-virginica
6.5,3.0,5.2,2.0,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
4.6,3.2,1.4,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
6.1,3.0,4.6,1.4,Iris-versicolor
5.8,2.6,4.0,1.2,Iris-versicolor
5.0,2.3,3.3,1.0,Iris-versicolor
6.7,3.0,5.2,2.3,Iris-virginica
5.6,2.7,4.2,1.3,Iris-versicolor
5.1,3.8,1.6,0.2,Iris-setosa
4.5,2.3,1.3,0.3,Iris-setosa
4.4,3.2,1.3,0.2,Iris-setosa
5.7,3.0,4.2,1.2,Iris-versicolor
5.7,2.9,4.2,1.3,Iris-versicolor
5.3,3.7,1.5,0.2,Iris-setosa
6.2,2.9,4.3,1.3,Iris-versicolor
5.1,2.5,3.0,1.1,Iris-versicolor
5.5,2.6,4.4,1.2,Iris-versicolor
5.7,2.8,4.1,1.3,Iris-versicolor
6.7,3.1,5.6,2.4,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
6.7,3.3,5.7,2.5,Iris-virginica
6.2,3.4,5.4,2.3,Iris-virginica
5.9,3.0,5.1,1.8,Iris-virginica
```

4.3 结果分析

得到了令我满意的结果。

5 程序源代码

Data 类

```
import java.util.Arrays;

/**
 * @author jxtxzzw 封装的数据类
 *
 */
public class Data {
    /**
     * 向量的个数，在本题中为 4
     */
    final static int NUMBER = 4;
    /**
     * 向量数组，存放每个向量的值
     */
    private double[] values = new double[NUMBER];
    /**
     * 标签，该数据对应的标签
     */
    private String label = null;
    /**
     * 距离，表示该点距离给定点的距离
     */
    private double distance;

    /**
     * 构造一个数据点
     *
     * @param rawData
     *            原始数据，字符串数组形式
     */
    public Data(String[] rawData) {
        for (int i = 0; i < NUMBER; ++i)
            values[i] = Double.parseDouble(rawData[i]);
        if (NUMBER < rawData.length)
            label = rawData[NUMBER]; // 标签仅在训练集和验证集有，所以需要校验
数组长度，测试集是没有标签的
    }

    /**
     * 归一化
     *
     * @param raw
     *            原数据点
     * @param newValue
```

```
     *              归一化后的新值
     * @return 归一化后的数据点
     */
    public static Data normal(Data raw, double[] newValue) {
        Data normal = new Data();
        // 依次赋归一化后的值，并复制标签
        for (int i = 0; i < NUMBER; ++i)
            normal.values[i] = newValue[i];
        normal.label = raw.label;
        return normal;
    }

    private Data() {

    }

    public String getLabel() {
        return label;
    }

    public double getValueAt(int index) {
        return values[index];
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + NUMBER;
        result = prime * result + ((label == null) ? 0 :
label.hashCode());
        result = prime * result + Arrays.hashCode(values);
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Data other = (Data) obj;
        if (label == null) {
            if (other.label != null)
                return false;
        } else if (!label.equals(other.label))
            return false;
        if (!Arrays.equals(values, other.values))
            return false;
        return true;
    }

    @Override
    public String toString() {
```

```
        return "Data [label=" + label + ", distance=" + distance + "]";
    }

    /**
     * 设置当前点到给定点的距离
     *
     * @param dist
     *            距离
     */
    public void setDistance(double dist) {
        distance = dist;
    }

    public double getDistance() {
        return distance;
    }

}
```

InputHandler 类

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * @author jxtxzzw 输入输出处理
 *
 */
public class InputHandler {
    Scanner in;

    /**
     * 从给定文件读取数据
     *
     * @param fileName
     *            文件名
     */
    public InputHandler(String fileName) {
        try {
            in = new Scanner(new File(fileName));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    /**
     * 读入原始数据，将每个数据点构造成一个 Data 泪，保存 ArrayList 的形式
     *
     * @return 所有读入的数据
     */
    public ArrayList<Data> input_raw() {
        ArrayList<Data> list = new ArrayList<Data>();
        while (in.hasNextLine()) {
```

```
            String[] s = in.nextLine().split(","); // 以逗号分隔，保存成字
符串数组
            Data d = new Data(s); // 构造，并加入 list
            list.add(d);
        }
        return list;
    }

    /**
     * 读入数据，并作归一化处理，将每个数据点构造成一个 Data 泪，保存 ArrayList 的形
式
     *
     * @return 归一化后的所有数据
     */
    public ArrayList<Data> input_normalized() {
        ArrayList<Data> raw = input_raw(); // 先读入原始数据
        ArrayList<Data> normalized = new ArrayList<Data>();
        // 遍历所有数据，找到最小和最大的值
        double[] mins = new double[Data.NUMBER];
        double[] maxs = new double[Data.NUMBER];
        for (int i = 0; i < Data.NUMBER; ++i) {
            mins[i] = Double.POSITIVE_INFINITY;
            maxs[i] = Double.NEGATIVE_INFINITY;
            for (Data d : raw) {
                double v = d.getValueAt(i);
                mins[i] = Math.min(mins[i], v);
                maxs[i] = Math.max(maxs[i], v);
            }
        }
        // 归一化处理，遍历原始数据每一个点，计算归一化后的值
        for (Data d : raw) {
            double[] newValue = new double[Data.NUMBER];
            for (int i = 0; i < Data.NUMBER; ++i)
                newValue[i] = normalize(d.getValueAt(i), mins[i],
maxs[i]);
            Data n = Data.normal(d, newValue); // 构造归一化后的 Data
            normalized.add(n);
        }
        return normalized;

    }

    /**
     * 归一化
     *
     * @param value
     *            当前值
     * @param min
     *            最小值
     * @param max
     *            最大值
     * @return 归一化后的值
     */
    private double normalize(double value, double min, double max) {
```

```
        return (value - min) / (max - min);
    }

}
```

KNN 类

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Set;

/**
 * @author jxtxzzw
 *
 */
public class KNN {
    /**
     * KNN 的 K 的取值
     */
    private int K;
    /**
     * 所有的数据
     */
    private ArrayList<Data> list;

    /**
     * @param K
     *            K 值
     * @param list 数据
     */
    public KNN(int K, ArrayList<Data> list) {
        this.K = K;
        this.list = list;
    }

    /**
     * 预测
     *
     * @param data
     *            给定的数据，没有标签
     * @return 预测的标签，字符串形式
     */
    public String predict(Data data) {
        // 遍历 KNN 中的每一个点，计算到给定点的距离
        for (Data d : list) {
            double dist = calculateDistance(data, d);
            d.setDistance(dist);
        }
        // 排序，排序的标准是距离从小到大，需要构造一个 Comparator（因为没有实现
Comparable 接口）
        Collections.sort(list, new Comparator<Data>() {
            @Override
            public int compare(Data d1, Data d2) {
```

```
                if (d1.getDistance() < d2.getDistance())
                    return -1;
                else if (d1.getDistance() == d2.getDistance())
                    return 0;
                else
                    return 1;
            }
        });

        // 统计出现次数最多的标签，直接用 MAP 来统计（红黑树）
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        // 遍历距离最小的 K 个点，获取他们的标签，对应标签计数
        for (int i = 0; i < K; ++i) {
            Data d = list.get(i);
            String label = d.getLabel();
            if (map.containsKey(label)) {
                int count = map.get(label);
                map.put(label, count + 1);
            } else {
                map.put(label, 1);
            }
        }

        // 遍历所有的标签，找到出现次数最多的那个
        String ret = null;
        int count = 1;
        Set<String> keySet = map.keySet();
        for (String s : keySet) {
            if (map.get(s) > count) {
                count = map.get(s);
                ret = s;
            }
        }
        return ret;

    }

    /**
     * 计算距离
     *
     * @param d1
     *            数据 1
     * @param d2
     *            数据 2
     * @return 距离
     */
    private double calculateDistance(Data d1, Data d2) {
        double sum = 0;
        for (int i = 0; i < Data.NUMBER; ++i)
            sum += Math.pow(d1.getValueAt(i) - d2.getValueAt(i), 2);
        sum = Math.sqrt(sum);
        return sum;
    }

    /**
```

```
     * 验证
     *
     * @param validation
     *            验证集
     * @return 准确率
     */
    public double validate(ArrayList<Data> validation) {
        int pass = 0;
        int all = validation.size();
        // 依次遍历，预测，然后取出标签比较预测结果
        for (Data d : validation) {
            String p = predict(d);
            String l = d.getLabel();
            if (p.equals(l))
                pass++;
            // System.out.println("Predict: " + p + ", Label: " + l);
        }

        return pass * 1.0 / all;
    }

}
```

Main 类

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

/**
 * @author jxtxzzw
 *
 */
public class Main {
    public static void main(String[] args) {
        // 训练
        InputHandler in = new InputHandler("train_104.txt");
        ArrayList<Data> list = in.input_normalized();
        KNN knn = new KNN(15, list); // K=15
        // 验证
        in = new InputHandler("validate_16.txt");
        ArrayList<Data> validation = in.input_normalized();
        double x = knn.validate(validation);
        System.out.println("Validation Correct Rate = " + (x * 100) +
"%"); // 输出正确率
        // 测试
        in = new InputHandler("test.txt");
        ArrayList<Data> test = in.input_normalized();
        ArrayList<String> ans = new ArrayList<String>();
        for (Data d : test) {
```

```
            String s = knn.predict(d);
            // System.out.println(s);
            ans.add(s);
        }
        BufferedReader bf;
        BufferedWriter bw;
        // 输出结果
        try {
            bf = new BufferedReader(new FileReader(new
File("test.txt")));
            bw = new BufferedWriter(new FileWriter(new
File("ans.txt")));
            for (int i = 0; i < ans.size(); i++) {
                String f = bf.readLine();
                String w = ans.get(i);
                bw.write(f + "," + w);
                bw.newLine();
            }
            bw.flush();
            bw.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("Predict result has been stored in:
ans.txt");

    }

}
```