1. KNN 算法简介

最近邻居法（KNN 算法，又译 K-近邻算法）是一种用于分类和回归的非参数统计方法。在这两种情况下，输入包含特征空间中的 k 个最接近的训练样本。在 KNN 分类中，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（k 为正整数，通常较小）中最常见的分类决定了赋予该对象的类别。若 k = 1，则该对象的类别直接由最近的一个节点赋予。在 KNN 回归中，输出是该对象的属性值。该值是其 k 个最近邻居的值的平均值。

最近邻居法采用向量空间模型来分类，概念为相同类别的案例，彼此的相似度高，而可以借由计算与已知类别案例之相似度，来评估未知类别案例可能的分类或属性值。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 Lazy 学习方法。k-近邻算法的缺点是对数据的局部结构非常敏感。

2. 存在的问题

k-近邻算法的预测结果与 k 的大小相关。同样的数据，K 值不同可能导致不同的预测结果。k-近邻算法存在归纳偏置：一个实例的分类与在欧氏空间中它附近的实例的分类相似。当数据集的维度较高时，就会出现实例间距离会被大量的不相关属性所支配，可能导致相关属性的值很接近的实例相距很远。也有可能存在这样一种情况，由于 KNN 算法的距离度量一般使用欧式距离。在 N 维空间中，待测样本点到训练样本点可以看作一个 N 维向量，该向量的长度就是训练样本到待测样本的距离。当待测样本与训练样本的夹角最小时，距离不一定最小，此时，有可能夹角最小的值更合适。从这个角度切入的话，也许某些系统的数据集适合这种方式。

下面几个例子用到的数据来自 sleep.arff(http://www.cs.waikato.ac.nz/ml/weka/datasets.html)。其中，属性从左到右依次为 body_weight、brain_weight、max_life_span、gestation_time、predation_index、sleep_exposure_index、danger_index、total_sleep、距离和夹角。需要求取的属性是 total_sleep。

实验所使用的软件平台为：

Windows 7 SP1 64bit

Java 1.8 update 45

Weka 3.6.12

IntelliJ IDEA 14.1.3

测试条件为 IB1 算法，66%训练，33%测试

待测样本	3.385	44.5	14	60	1	1	1	12.5		
训练样本	36.33	119.5	16.2	63	1	1	1	13	0.000726	0.092242
	4.235	50.4	9.8	52	1	1	1	9.8	0.002009	0.128392
	0.01	0.25	24	50	1	1	1	19.9	0.010786	0.305307
	100	157	22.4	100	1	1	1	10.8	0.011984	0.084494
	1.7	6.3	5	12	2	1	1	19.4	0.076781	1.477736
	3.5	10.8	6.5	120	2	1	1	17.4	0.077412	1.140567

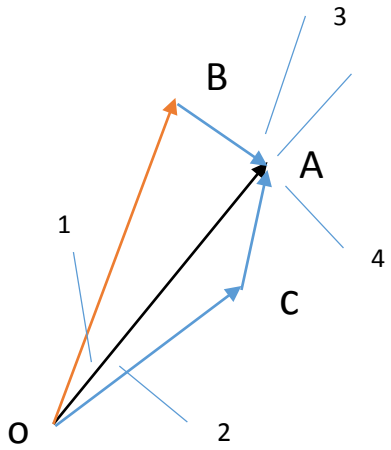
待测样本	0.75	12.3	7	225	2	2	2	6.6		
训练样本	0.48	15.5	12	140	2	2	2	17	0.020651	0.250394
	1.4	12.5	12.7	90	2	2	2	11	0.048888	0.404808
	4.288	39.2	13.7	63	2	2	2	12.5	0.070222	0.499235
	3.6	21	6	225	3	2	3	5.4	0.125107	0.336791
	0.9	2.6	4.5	60	2	1	2	13.3	0.131103	0.707295
	0.104	2.5	2.3	46	3	2	2	15.8	0.144782	0.656431

待测样本	1.62	11.4	13	17	2	1	2	13.7		
训练样本	0.9	2.6	4.5	60	2	1	2	13.3	0.012186	0.301788
	4.288	39.2	13.7	63	2	2	2	12.5	0.067856	0.608436
	1.7	6.3	5	12	2	1	1	19.4	0.069268	0.790189
	0.12	1	3.9	16	3	1	2	14.4	0.071181	0.416673
	3.5	3.9	3	14	2	1	1	19.4	0.073001	0.817971
	1.4	12.5	12.7	90	2	2	2	11	0.075809	0.637453

上面使用的是 Weka 中的 IB1 算法，该算法使用最近的邻居的值来推断待测样本的值。对该算法简单修改，让其记录夹角。上面三个例子都能反应，夹角较小的样本不一定距离最近，在第一个和第三个例子中，使用夹角最小的两个样本值的平均值的话会更接近真实值。第二个例子也指出，仅仅使用夹角来判断也是不够的。所以应该结合两者来判断。

3. 分析

可能存在这样的系统，当系统的某一个属性值发生改变时，目标值会发生较大改变，而当系统的大部分属性值都发生改变时，目标值变化较小。也可能存在与之相反的系统，当某一个属性值改变时，目标值变化较小，而当系统大部分属性值发生改变时，目标值变化较大。前一种情况应该是不常见的。最一般的，用欧式距离度量，然后求平均值的做法适合后一种情况。即便是在后一种情况中，考虑夹角也许能降低误差。



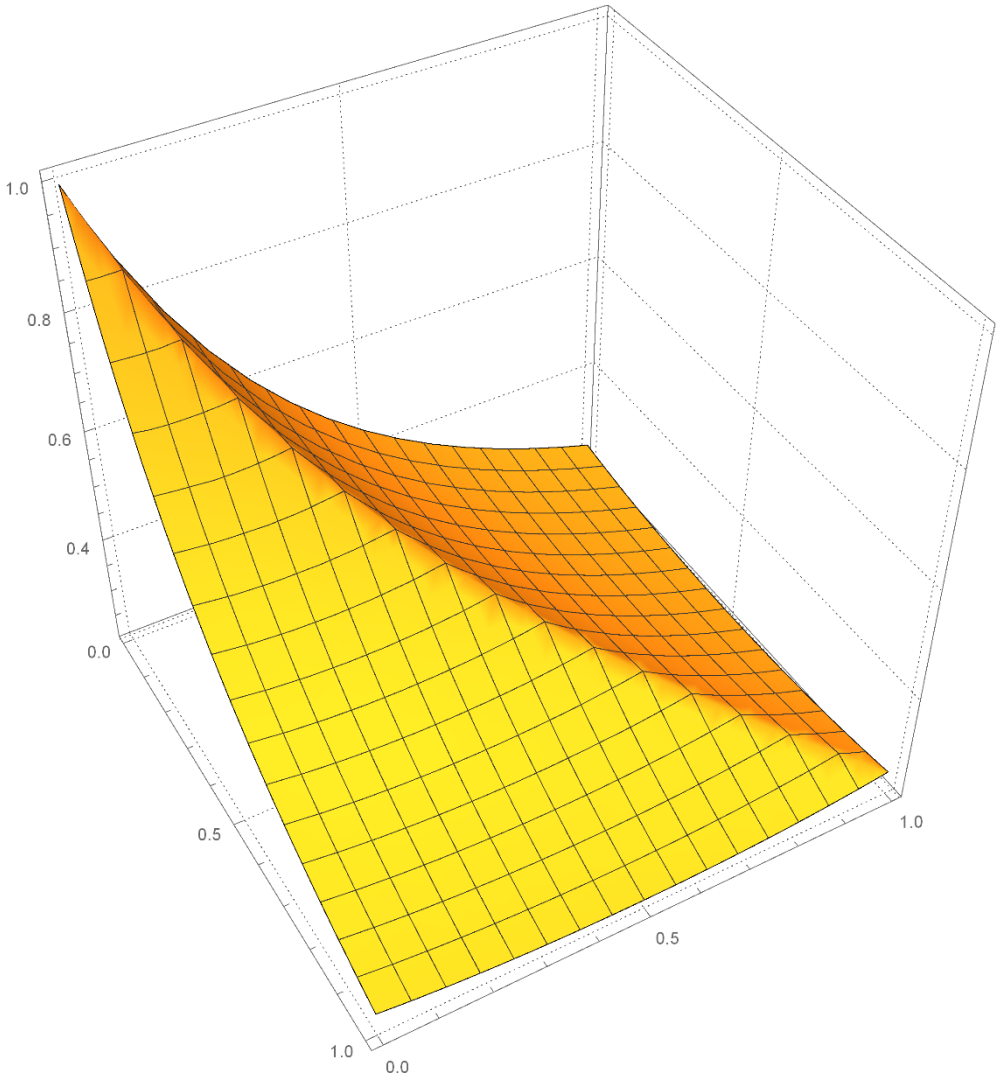
关于距离的度量。把所有的属性值做归一化，使其范围在 0——1。距离量化使用欧式距离。如果数据集有三个属性（不包括要预测的目标属性）的话，那么所有的样本点在一个半径为 1 的球体内。如右图所示，假设 O 点是球体的中心，B、C 是训练样本，A 是待测样本。OA 就是 A 样本归一化后所有属性值的欧式距离。采用 KNN 方法，我们要度量的是距离 AB、AC。夹角的话有两种，第一种是角 1 和角 2。这两个角与 AB 和 AC 的长度有关系，所以在大部分情况下，按距离排序后，角 1 和角 2 的顺序也排好了。角 3 和角 4 与 AB 和 AC 的长度基本没有关系。两者的参考标准不同，前者都是相对于 O 点的，后者相对于待测样本点 A 度量的。

对于 B 点和 C 点，AB 和 AC 长度基本相同，但是角 1 和角 2 与角 3 和角 4 就有明显的差异了。我认为角 1 和角 2 适合当某一个属性值改变时，目标值变化较小，而当大部分属性值发生改变时，目标值变化较大的系统。角 3 和角 4 适合当某一个属性值发生改变时，目标值会发生较大改变，而当大部分属性值都发生改变时，目标值变化较小的系统。后面会用数据集测试下。

4. 方法

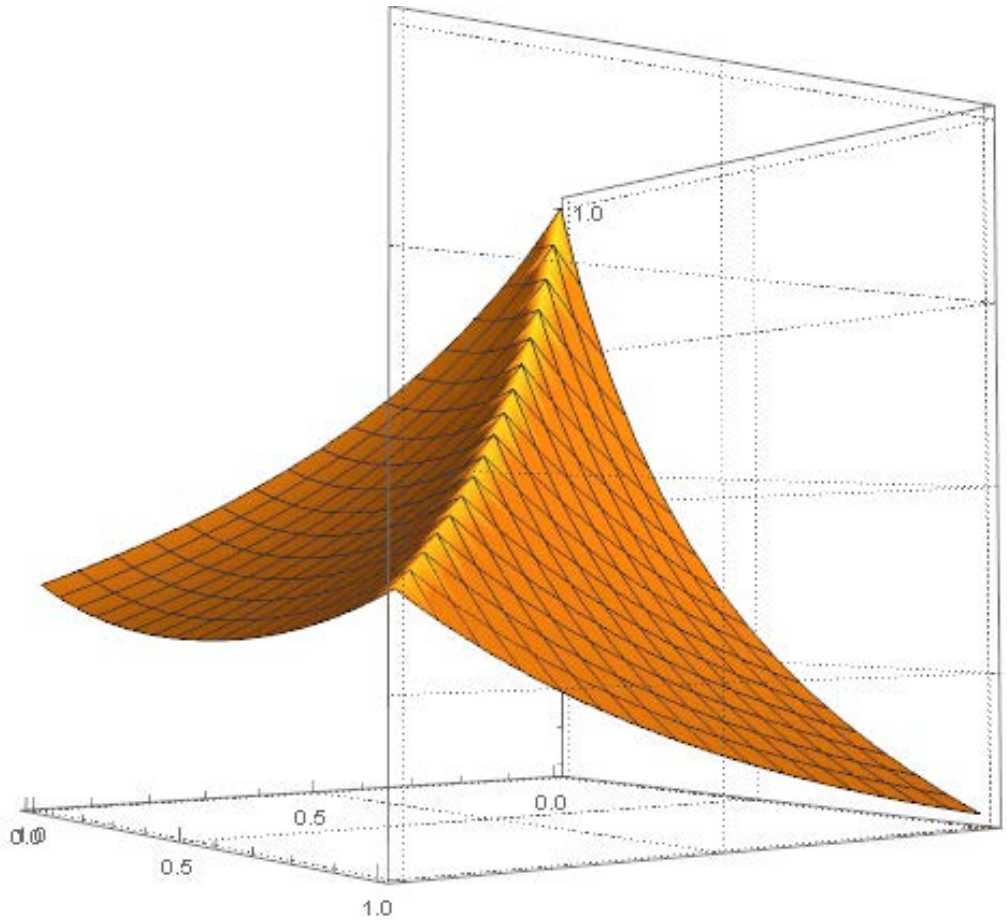
之前是只关注距离，现在加了角度，这两个量应该怎样影响目标属性的取值呢。可以参考用距离加权平均的方法，只不过这里权值由距离和夹角共同决定。当采用角 1 和角 2 时，我们假设夹角和距离同时比较小的训练样本更接近真实值。也就是说这样的样本的权值应该大一些。为此设法寻找一个函数，当 x 和 y 都比较小时，函数值较大，x 和 y 中至少有一个偏大，就会导致函数值变小。

$$\frac{1}{(|x-y|+1)\,(x+1)(y+1)}$$



从三维图上很容易看到，这个函数满足我们的要求。
当我们需要权值偏向于夹角或者距离时，可以简单修改下函数，例如

$$\frac{100}{(|x-y|+1)\,(x+100)(y+1)}$$



这样，权值的计算就解决了。

5.上机实习

首先将 IB1.java 中的代码复制到自己新建的包中，然后删去用不到的代码，knn 类中只剩下 buildClassifier、classifyInstance 和 main 这三个方法。

在 buildClassifier 方法中，新建两个属性“distance”和“angle”用来存储距离和夹角，并把这两个属性放到最后面。

```
m_Train = new Instances(instances, 0, instances.numInstances());
m_Train.insertAttributeAt(new Attribute("distance"),
m_Train.numAttributes());
m_Train.insertAttributeAt(new Attribute("angle"), m_Train.numAttributes());
```

写一个 angle 函数来计算夹角，供 classifyInstance 方法调用。

```
private double angle(Instance first, Instance second) {
    double diff = 0, diff1 = 0, diff2 = 0, angle, innerProduct = 0,
moduleProduct, module1 = 0, module2 = 0;
    for(int i = 0; i < m_Train.attribute("distance").index(); i++) {
        if (i == m_Train.classIndex()) {
            continue;
        }
        if (!first.isMissing(i) && !second.isMissing(i)){
            diff = norm(first.value(i), i) * norm(second.value(i), i);
            diff1 = norm(first.value(i), i) * norm(first.value(i), i);
            diff2 = norm(second.value(i), i) * norm(second.value(i), i);
        }
        innerProduct += diff;
        module1 += diff1;
        module2 += diff2;
    }
    moduleProduct = Math.sqrt(module1) * Math.sqrt(module2);
    angle=Math.acos(innerProduct/moduleProduct);
    return angle;
}
```

由于增加了两个属性，所以在 distance 函数和 updateMinMax 函数中的 for 循环停止条件要做修改。

```
for (int j = 0;j < m_Train.attribute("distance").index(); j++)
```

在 classifyInstance 方法中我们一开始并没有修改 classValue 的值的算法，只是把距离和夹角保存起来，并输出出来，本文一开始的三个例子就是这样输出出来的。

```
while (enu.hasMoreElements()) {
    Instance trainInstance = (Instance) enu.nextElement();
    if (!trainInstance.classIsMissing()) {
        distance = distance(instance, trainInstance);
        angle = angle(instance, trainInstance);
        trainInstance.setValue(m_Train.attribute("distance"), distance);
        trainInstance.setValue(m_Train.attribute("angle"), angle);
        if (distance < minDistance) {
            minDistance = distance;
            classValue = trainInstance.classValue();
        }
    }
}
m_Train.sort(m_Train.attribute("distance").index());
System.out.println("fly");
System.out.println(instance);
for (int i=0; i<4; i++){
    System.out.println(m_Train.instance(i));
}

return classValue;
```

下面给出的是完整的 classifyInstance 方法

```
public double classifyInstance(Instance instance) throws Exception {

    if (m_Train.numInstances() == 0) {
        throw new Exception("No training instances!");
    }

    double distance, angle, classWeightSum=0, classValue = 0;
    Instances m_MinAngel, m_MinDistance;

    updateMinMax(instance);

    Enumeration enu = m_Train.enumerateInstances();
    while (enu.hasMoreElements()) {
        Instance trainInstance = (Instance) enu.nextElement();
        if (!trainInstance.classIsMissing()) {
            distance = distance(instance, trainInstance);
            angle = angle(instance, trainInstance);
            //保存计算结果
            trainInstance.setValue(m_Train.attribute("distance"), distance);
            trainInstance.setValue(m_Train.attribute("angle"), angle);
        }
    }
    //按夹角排序
    m_Train.sort(m_Train.attribute("angle").index());
    //将前 2 个复制到 m_MinAngle
    m_MinAngel = new Instances(m_Train, 0, 2);
    m_MinAngel.insertAttributeAt(new Attribute("classWeight"), m_MinAngel.n
umAttributes());
    m_Train.sort(m_Train.attribute("distance").index());
    m_MinDistance = new Instances(m_Train, 0, 2);
    m_MinDistance.insertAttributeAt(new Attribute("classWeight"), m_MinDist
ance.numAttributes());

    //将夹角和距离对象合并为一个新对象
    Instances m_Weight = new Instances(m_MinAngel);
    for (int i=0; i<m_MinDistance.numInstances()-1; i++){
        m_Weight.add(m_MinDistance.instance(i));
    }
    //将夹角归一化
    m_Weight.sort(m_Weight.attribute("angle").index());
    for (int i=0; i<m_Weight.numInstances()-1; i++){
        angle = m_Weight.instance(i).value(m_Weight.attribute("angle").index
())/m_Weight.instance(m_Weight.numInstances()-1).value(m_Weight.attribute
("angle").index());
        m_Weight.instance(i).setValue(m_Weight.attribute("angle").index(), a
ngle);
    }
    //将距离归一化
    m_Weight.sort(m_Weight.attribute("distance").index());
```

```
    for (int i=0; i<m_Weight.numInstances()-1; i++){
        distance = m_Weight.instance(i).value(m_Weight.attribute("distance
").index())/m_Weight.instance(m_Weight.numInstances()-1).value(m_Weight.at
tribute("distance").index());
        m_Weight.instance(i).setValue(m_Weight.attribute("distance").index
(), distance);
    }
    //计算权值
    for (int i=0; i<m_Weight.numInstances()-1; i++){
        angle = m_Weight.instance(i).value(m_Weight.attribute("angle").inde
x());
        distance = m_Weight.instance(i).value(m_Weight.attribute("distance
").index());
        m_Weight.instance(i).setValue(m_Weight.attribute("classWeight").ind
ex(),100/((Math.abs(angle-distance)+1)*(angle+1)*(distance+100)));
    }
    //将权值归一化
    m_Weight.sort(m_Weight.attribute("classWeight").index());
    for (int i=0; i<m_Weight.numInstances()-1; i++){
        classWeightSum += m_Weight.instance(i).value(m_Weight.attribute("cl
assWeight").index());
    }
    for (int i=0; i<m_Weight.numInstances()-1; i++){
        double classWeight = m_Weight.instance(i).value(m_Weight.attribute("
classWeight").index());
        m_Weight.instance(i).setValue(m_Weight.attribute("classWeight").ind
ex(), classWeight/classWeightSum);
    }
    //计算 classValue
    for (int i=0; i<m_Weight.numInstances()-1; i++){
        classValue += m_Weight.instance(i).value(m_Weight.attribute("classW
eight").index()) * m_Weight.instance(i).classValue();
    }

    return classValue;
}
```

完成代码后，测试数据集一共使用了三个，分别为 sleep.arff、autoPrice.arff 和 pollution.arff。

正如上面代码所示，使用的角 1 和角 2，权值分配时更倾向于距离。下面出现的 KNN 算法均指上面代码实现的算法，Weka 自带的算法会明确标出。

结果概括如下：

情景 1：KNN 算法

情景 2：IB1 算法，即 K=1

情景 3：IBK 算法，K=3，no distance weighting

情景 4：IBK 算法，K=3，weight by 1/distance

情景 5：IBK 算法，K=5，weight by 1/distance

对于数据集 sleep.arff，采用交叉验证，Folds=10

参数	*情景 1*	*情景 2*	*情景 3*	*情景 4*	*情景 5*
Correlation coefficient	0.7835	0.775	0.7483	0.7875	0.7541
Mean absolute error	2.1982	2.3627	2.551	2.3008	2.4838
Root mean squared error	3.1141	3.4373	3.0911	2.9093	3.0782
Relative absolute error	57.80%	62.13%	67.08%	60.50%	65.31%
Root relative squared error	66.05%	72.90%	65.56%	61.70%	65.28%
Total Number of Instances	51	51	51	51	51

对于数据集 autoPrice.arff，采用交叉验证，Folds=10

参数	*情景 1*	*情景 2*	*情景 3*	*情景 4*	*情景 5*
Correlation coefficient	0.8729	0.8738	0.862	0.8801	0.878
Mean absolute error	1629.1417	1609.8648	1798.3449	1668.0592	1666.5529
Root mean squared error	2920.2273	2902.5515	3010.9122	2888.5787	2953.8493
Relative absolute error	35.47%	34.83%	38.91%	36.09%	36.06%
Root relative squared error	49.49%	49.09%	50.92%	48.86%	49.96%
Total Number of Instances	159	159	159	159	159
UnClassified Instances	1 / 0.6289 %				

对于数据集 pollution.arff，采用交叉验证，Folds=10

参数	*情景 1*	*情景 2*	*情景 3*	*情景 4*	*情景 5*
Correlation coefficient	0.551	0.5507	0.5996	0.615	0.6687
Mean absolute error	44.9928	45.0788	38.4731	37.7571	35.2371
Root mean squared error	57.4651	57.5308	49.7635	48.993	45.9815
Relative absolute error	89.33%	89.50%	76.39%	74.97%	69.96%
Root relative squared error	92.03%	92.13%	79.69%	78.46%	73.64%
Total Number of Instances	60	60	60	60	60

6. 分析

由此可见考虑夹角后可以改善结果。但是对于数据集 pollution.arff，可能因为属性之间有某种关系，导致结果变差。

如果采用角 3 和角 4，三个数据集的结果会非常差。我认为出现这种情况可能是因为某些属性之间有内在的联系，预处理阶段采用属性选择或者属性规约也许能改善结果。