中山大学移动信息工程学院本科生实验报告

（2017年秋季学期）

课程名称:*Artificial Intelligence*

年级	1501	专业（方向）	移动互联网
学号	15352015	姓名	曹广杰
电话	13727022190	Email	1553118845@qq.com

一、实验题目

K近邻模型处回归问题

二、实现内容

1. 使用KNN处理回归问题，在验证集上，通过调节K值、选择不同距离等方式得到一个相关系数最优的模型参数，并将该过程记录在实验报告中。这一步可以通过使用“validation相关度评估.xlsx”文件辅助验证（也可以自己写代码）。
2. 在测试集上应用步骤1中得到的模型参数（K，距离类型等），将输出结果保存为“学号*姓名拼音*KNN_regression.csv”。

算法原理

KNN模型在应对大量的数据集的时候，只选择与测试数据相近的信息，缩小计算范围由此减少计算量——这种方式减少了与测试样本无关的数据对样本可能产生的影响。

Knn处理数据回归。Knn会根据定义的距离计算方式获得与当前处理信息最贴近的数据，并根据已有的数据使用**特定的算法**获得每种情感的比例。

Knn方法的不足。Knn方法当然不是完美的，这种方法在消除了无关数据对数学模型的影响之后，同时放弃了其他有关模型或者关联性较弱的数据对于当前模型的加权意义。这就使得Knn方法可以在大致方向上与其同类的消息匹配得很好，但是在数据的调整上很少有发挥的余地。

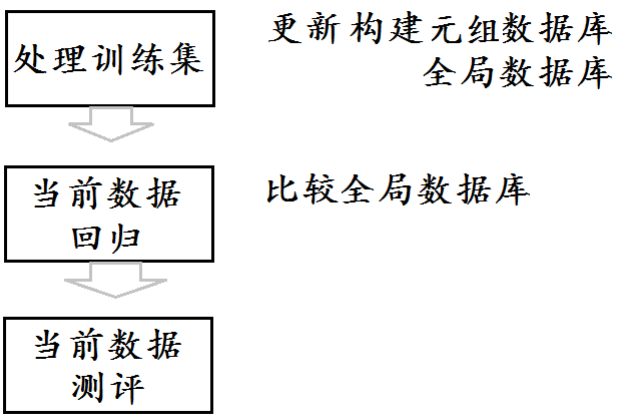
- 距离的计算：距离的计算可能有多种方式。包括曼哈顿距离、欧式距离以及余弦距离等。所谓距离计算方式是对当前的分类操作建立一个数学模型，对于实际问题而言，不同的数学模型可能会有不同的效果，这也取决于我们的模型的构建以及数学关系的嵌套是否可以与实际问题符合得更好。
- 缩小计算范围。当笔者在分析当前的一条数据与数据集中的所有数据时，我们不敢保证我们的算法可以发现每一条信息隐藏的关系——这种情形下，如果一定要将与当前处理的数据关系微弱的信息纳入当前的数学模型，十有八九会使得模型为了迁就该部分数据而导致数学模型的波动与失衡。所以，Knn就设计模型以避开这个问题，Knn只选择关系最紧密的几个近邻值纳入数学模型，这就十分有效地避免了无谓数据对模型的影响。
- 特定的算法：与距离的计算很类似，需要根据具体的情况，结合实际问题，创建数学模型以解释当前的情况。

伪代码

在实现数据处理之前，我们需要构建几个全局通用的数据库。

- 所有词的数据库，词之间互不重复。该数据库将用于构建one hot矩阵。
- 训练集每一行的信息，元组集合。该数据库将用于在训练集巨大的数据量中筛选与当前数据最靠近的几个近邻值，用于作为遍历时候的索引。
- validation文件的句子信息集合。保存了validation文件中表达信息的句子，并以字符串格式保存在vector中。

整体流程



笔者在分类的基础上，对原有的主函数内部的实现进行了打包与封装，这里流程图可以找到与之对应的伪代码，如下：

```
int main(int argc, char** argv) {
    train("train_set.csv"); // 分析训练数据集
    for(int i=0; i<元组个数; i++){ // 更新元组集合
        当前元组更新TF矩阵的一行;
    }

    string file_name="validation_set.csv";
    valid(file_name);// 分析validation文件
    for(int i=0; i<validation文件的数据数目; i++){
        由数据字符串注册新的元组tp;
        tp.taste(5);// tp元组尝出感情，更新感情成员变量
    }
    /*
    输出数据
    */
    return 0;
}
```

关键代码

结构体框架

正如在分类中所描述的，笔者为导入的数据的每一行都设计了类以拟合当前的信息，方便后续的操作。表示当前文件的每一行，这种传统在回归问题上得到了传承，但是又有所发展：

```
class Tuple{ // 元组：如同train_set的一行
  public:
  void Collectvoc(string s, bool fix_base);// 从s中收集词，bool值决定是否添加到库中
  void Updaterow(); // 实现更新TF矩阵的一行
  void Register(string s){
    Collectvoc(s, false);
    Updaterow();
  }
  double Distanceof(Tuple tuple);
  void taste(int scope);//对当前元组的词库进行分析，品尝到底是什么情感，int为近邻域

  vector<bool> row;
  vector<string> vca; // 获得句子，拆分成词库
  Feel brain; // 表示人的6中情感，Feel为一个自定义的结构体
};
```

这其中的Feel是表示人的六欲的结构体：

```
class Feel{
public:
    void setfeel(vector<double> feelbase);
    vector<double> feelbar;
//  debug
    void visit_feelbar();
private:
    double anger;
    double disgust;
    double fear;
    double joy;
    double sad;
    double surprise;
};
```

笔者另为之添加了接口，所以内部的私有成员变量一般用不到，这里也就不做过多的解释了。

按照惯例，接下来按照整体流程的顺序，分析如何处理训练集并将训练集数据添加到数据库中的。

处理训练集

笔者在这里处理训练集的时候，是依照原有的分类函数进行修改的，整体格局相近，但是由于读入的字符串更加繁杂琐碎，使得字符串处理的部分占据了很大的篇幅，这种修改的结果就是可读性不强，凌乱。笔者这里就不过多地贴码了，因为这一部分与主体思路偏离。

回归计算在这里另外实现了一个函数，这样的修改使得之后的实现更为清晰、简洁与流畅。回归函数为taste，在Tuple类中。

近邻区域的选择

为了实现回归计算，首先要做的，还是对近邻域的选择。这次实现对于近邻域的计算要求更高，我们需要不仅仅是标签和距离，我们需要连同情感信息在内的元组的所有内容。那么这个问题就会变成：

Q:如何在元组集合中选择最接近当前数据的集合？

选项：

- 对当前的元组集合进行自定义排序：

  就算是使用快排——假如笔者会的话，计算复杂度至少要n*log(n)。

- 对当前集合进行遍历，获得符合条件的K个数据：

  一次遍历，更新容器，获得近邻值，计算复杂度为n。

这里笔者考虑的是对于与当前数据相差过多的元组，我们可以直接选择舍弃，而不使其参与任何的排序过程，这种方式可以节省下大量的计算资源与时间。

以下为对于第二种方式的实现：

```
map<double, int> findtp;// Tuple（元组）的索引号
vector<double> cdst;// candidate distance options
int update_cdst = 0;
for(int pin=0; pin<scope; pin++)    cdst.push_back(100);
// 对所有的元组进行遍历，查找到更为相近的元组
for(int tpin=0; tpin<Tuplebase.size(); tpin++){
  double dst = Distanceof(Tuplebase[tpin]);
  // 当前距离小于该容器内的最大距离
  if(dst < cdst[cdst.size()-1]){
    // 更新
    update_cdst++;
    cdst.push_back(dst); // 更新距离值
    findtp[dst] = tpin;  // 更新与之对应的索引号！！！
    sort(cdst.begin(), cdst.end());// 几十个数字的排序
    cdst.pop_back();    // 更新元素信息
    // 更新结束
  }
}
```

注意到初始化的时候，笔者将容器内的每一个数字都初始化为100了。这种情况下其实还是有可能出岔子的——比如，距离值都大于100，比如设定的K值甚至大于训练集容量。这种情况下，很可能多出一些莫名其妙的数据，这些数据将会给之后的debug工作带来不可预料的困难。

为此添加提示监听：

```
if(update_cdst < scope){
    printf("[Tuple.cpp/taste]:Scope or distance is too big.\n");
    printf("[Tuple.cpp/taste]:Program abort.\n");
    exit(0);
}//这样就万无一失啦哈哈哈
```

至此，我们就获得了我们需要的几个元组的序号，但是并没有提取出它们的信息。不过找到它们也只是探囊取物。因为我们把每一个距离的信息都与遍历时候元组的信息进行了映射，而且在不断地更新。其名字就是findtp。

在拥有了近邻域之后，就是对回归操作的计算了。

```
vector<double> fb; // 容量为6，储存6种情感值
for(int i=0; i<6; i++){
  double fp = 0; // 表示当前的情感值，之后会收集到容器内
  // 近邻域遍历
  for(int dstpin=0; dstpin<cdst.size(); dstpin++){
    double dst=cdst[dstpin];
    int tpin = findtp[dst];
    // 若与近邻域中的某一个元组完全符合
    if(Distanceof(Tuplebase[tpin]) == 0){
      fp = 0;
      fp += Tuplebase[tpin].brain.feelbar[i];
      break;
    }else{
     // 若不完全符合
      double key_feel = Tuplebase[tpin].brain.feelbar[i];
      fp = fp + key_feel/(0.01+Distanceof(Tuplebase[tpin]));// 公式
    }
  }
  // 循环外，收集集合参数
  fb.push_back(fp);
```

在获得了6种情感值之后，将这个容器归一化后的信息添加到当前数据的成员变量中即可。

```
brain.setfeel(fb);
```

创新点与优化

计算效率优化

很遗憾笔者在准确度上一直没有什么大的进展，但是笔者在运行效率上的修改是卓见成效的。从几十秒到20秒到十几秒。通过对无效计算的规避，消减了计算的复杂度。

debug处理

在编程过程中对可能发生的错误的预知可以在极大程度上减少后续修改中的困难。尤其在引入新的工具之后。笔者为减少排序的复杂度与计算量引入了vector，同时也需要对vector的困难引发的问题进行考虑。

情感值的计算

有一个不变的问题就是数学模型的套用，在本次实验中，笔者使用的数学模型是双曲线。当然，双曲线在趋势上的把握还是准确的。但是在应对较小值与极小值的时候，双曲线就显得犹豫不决优柔寡断了。

如我们所知，在validation集合中，有很多数值为0，这些数值在双曲线中表现为似有似无的极小值，可能精确到小数点后几位，尽管在一个数据中影响不明显，但是在数据量高达几百个甚至几千个的时候，这种影响的叠加就不可控了——它可能会相互抵消，也可能再次叠加。但是不可控的算法也就不可信。为此，笔者做了以下改进：

```
double key_feel = Tuplebase[tpin].brain.feelbar[i];
if(key_feel > 0.5){
  key_feel *= 1.5;
  key_feel = (int)(100*key_feel);
  key_feel /= 100;
}else if(key_feel < 0.5){
  key_feel *= 0.3;
  key_feel = (int)(100*key_feel);
  key_feel /= 100;
}
fp = fp + key_feel/(0.01+Distanceof(Tuplebase[tpin]));
```

笔者在原有公式的基础上，添加了阈值的限定——之前讨论过的双曲线收敛速度过低的问题，在这里由笔者手动解决。尽管这种方法并没有逃离小垃圾的噩梦，但是在一定程度上进行了修正。

而与之类似的，还可以使用对数收敛（对数在0-1区间收敛速度飞快的）、指数收敛（和对数一样的）和截断收敛等等，不一而足。

与此类似的还有使用TFIDF矩阵，断定在某一个文本中的最重要的词——个人感觉在本次实验中意义不大，实验数据量少暂且不谈，内容出现频率也非常有限，TFIDF矩阵的战场并不多。

三、实验结果与分析

1. 实验结果示例展示

	anger	disgust	fear	joy	sad	surprise
r	0.050976968	0.087537402	0.071502892	0.204536943	0.266760867	0.2003075
average	0.146937095					
evaluation	极弱相关 加油哦小辣鸡					

2. 评测指标展示及分析

	anger	disgust	fear	joy	sad	surprise
r	0.089547636	0.090751915	0.121242937	0.217373302	0.27280703	0.193898219
average	0.164270173					
evaluation	极弱相关 加油哦小辣鸡					

对于笔者优化后的输出结果，相对于原本的信息都是"小辣鸡"的级别，我真是不想再说什么了。

测评指标以evaluation为准，二者相差不大，分析原因：

- 使用双曲线的数学模型符合度不高，测试数据与比较元组的距离不足以限定存在的情感系数。
- 传入的参数有限：
  - 在当前测试数据中可能会出现训练集中都未曾出现过的数据，这时候参考数据不再具有参考性
  - 距离作为衡量的唯一标准这样有失公正，对于不同的句子来说，不一样的词语甚至决定了该句子的情感系数的可能性——如果句子中出现了词语"die"其负面情绪的可能性就大大增加，除非是坏人死了——怎么会有绝对的坏人呢？

    在实际的处理中，应该找到这种关键词，并为之分离形成一个更有优势的数据库，再结合当前的距离信息，综合对当前数据进行考虑。

    distance仅仅表示相似度，用它作为衡量的唯一标准就已经犯了方向性的错误，这已经不再是凑几个模型或者尝试几次排列能够修正的了。

四、思考

- 为什么Knn相似度加权的时候，要将距离的倒数作为权重

  距离表示当前检查的元组与比较的元组之间的距离，距离越长则这两个元组的相似度越低，对照的元组参考性就越低，这种情况下，参考性与距离之间就呈负相关关系。因而，使用反比例函数是符合要求的。

- 如何归一化

  如果当前的情感系数不都为0，那么可以对所有的系数做和，再将每一个系数除以系数之和，以达到归一化的目的。

  如果当前的情感系数都未被探测到，可以都设置为1/6，但是这种方法的对该问题的符合度不高。

- 矩阵稀疏度不同的时候，曼哈顿距离与欧式距离有什么不同

  曼哈顿距离与欧式距离都是距离范式中的一种形式，曼哈顿距离取参数值为1，欧式距离取参数为2。

  当矩阵非常稀疏的时候，进行计算的数组中都具有很多的0，而非零的数据密度极低。在计算距离的时候，如果非零的值大于1，则曼哈顿距离计算得到的距离更大，而值小于1的时候，计算的距离更小——所谓波动都比欧式距离更小，因此，欧式距离的计算结构更加稳定。

  结合矩阵的稀疏度，对于不同的矩阵：

  - 使用曼哈顿距离计算得到的数据波动幅度更大，可以作为区分矩阵的特征量
  - 使用欧式距离计算得到的数据幅度波动较小，可以限定对数据的过度拟合分析

- 伯努利模型与多项式模型各有什么优缺点

  伯努利模型的重点集中于文本的数量，伯努利模型是以文本为计算单位的。而多项式模型是以词为单位的。

  伯努利模型在应对区分度大的文本集的时候，优势大于多项式模型，伯努利模型的处理方式决定了其能够很清楚地分清，当前这个词，在文本中到底有多大的价值，如果在此类文本中，出现这个词的概率过于小，那就说明该词并不能被人们接受地表达该类别文本的意思。但是伯努利模型在处理少量数据的时候就显现出巨大的劣势——如果我们的文本数量少得可怜，但是文本的内部有具有大量的信息可以挖掘，伯努利模型在这方面的表现很显然是不足的。

  多项式模型对文本之间的区分并不是非常敏感，多项式模型在分析一个词的时候，参考的主体是整个词库，这个词库表达的内容繁多而且值得挖掘，如此，对于这个词的理解就很深刻。但是同时多项式模型淡化了文本的概念，过度放大了出现次数所占的比重，这种特点的结果就是对介词副词等无实际意义的连接词没有任何抵抗力，非常容易混淆视听。

- 如果测试集中出现了一个全词典都没有出现的词应该如何处理

  常规来讲我们是没有任何办法的，在Bayes的regression实验中，有大量的测试数据都没有在训练集中出现过，这种情况下，就不得不放弃对于整个句子的句意的猜测。

  如果该句子中还有其他的词汇，我们可以降低这个未知的词在分析过程中所占的比重，将重点转移到其他已知内涵的词语中。

  但是正如Bayes的regression实验中，有的句子中一个词汇都没有出现过，这种情况下就真的是一筹莫展，就算是人，也鲜有办法。