实验报告

1 实验简介

本次实验主要的工作是在 MNIST 数据集上使用 KNN 算法完成多分类任务，并比较了使用不同的 k 值和距离度量对分类准确率的影响。同时完成了带权值的 KNN 算法和用 KD 树实现 KNN 两个实验，并将准确率和程序运行时间与原始 KNN 算法进行比较并加以分析。

本次实验的代码采用 Python 3 实现，IDE 是 Jupyter Notebook。代码文件为 KNN_final.py, 里面有比较详细的注释，还有 Jupyter Notebook 下的 KNN_final.ipynb，可以直接在 Jupyter Notebook 中运行。这两个文件和实验报告一起打包。

2 实验原理

1.KNN 的基本原理

KNN 算法的基本原理是为了判定未知样本的类别，以全部训练样本作为代表点，计算未知样本与所有训练样本的距离，选择与未知样本距离最近的确定个数的 K 个样本，根据这 K 个样本的类别进行投票表决，得票最多的类别即为未知样本的预测类别。KNN 算法中两个比较重要的超参数是 k 值的选择和距离度量方式的选择。k 值太小会使算法对噪声过于敏感，k 值太大会使类别之间的界限变得模糊。距离度量方式通常需要根据具体的问题选择合适的度量。

2. 带权值的 KNN

普通的 KNN 算法在投票过程中对于距离最近的 k 个样本中的每一项都同等对待，即对于每一项都在对应类别数量上加 1。一种合理的思路是给与未知样本距离更近的样本赋予更高的权重，即根据每个样本与未知样本的距离给该样本赋予一个权重 w_j，其取值范围是 0-1，与未知样本距离更近

的样本在投票中的话语权更大。该权重 w_j 定义为：

$$w_j = \begin{cases} \dfrac{d_k - d_j}{d_k - d_1}, & d_k \neq d_1 \\ 1, & d_k = d_1 \end{cases}$$

其中 $d_1, ...d_k$ 是按从小到大排列的 k 个样本与未知样本的距离。对于距离最近的 k 个样本中的每个样本 j，其对应的类别加上权重 w_j，然后选取和最大的类别作为预测类别。

3. 使用 KD 树实现 KNN

实现 k 近邻法时，一个重要的问题是如何对训练数据进行快速 k 近邻搜索。普通的 KNN 算法使用穷举搜索，即计算输入实例与每一个训练实例的距离。计算并存储好以后，再查找 k 近邻。当训练集很大时，计算非常耗时。为了提高 KNN 搜索的效率，可以考虑使用特殊的结构存储训练数据，以减小计算距离的次数。kd 树是一种对 k 维空间中的实例点进行存储以便对其进行快速检索的树形数据结构。kd 树是一种二叉树，表示对 k 维空间的一个划分，构造 kd 树相当于不断地用垂直于坐标轴的超平面将 k 维空间切分，构成一系列的 k 维超矩形区域。kd 树的每个结点对应于一个 k 维超矩形区域。利用 kd 树可以省去对大部分数据点的搜索，从而减少搜索的计算量。

使用 kd 树实现 KNN 首先要把所有的训练数据存储到 kd 树中，即完成 kd 树的构造。构造 kd 树的过程为：构造根结点，使根结点对应于 K 维空间中包含所有实例点的超矩形区域；通过递归的方法，不断地对 k 维空间进行切分，生成子结点。在超矩形区域上选择一个坐标轴和在此坐标轴上的一个切分点，确定一个超平面，这个超平面通过选定的切分点并垂直于选定的坐标轴，将当前超矩形区域切分为左右两个子区域（子结点）；这时，实例被分到两个子区域，这个过程直到子区域内没有实例时终止（终止时的结点为叶结点）。在此过程中，将实例保存在相应的结点上。选择坐标轴时通常选择数据最分散 (方差最大) 的维度，然后在此维度上对数据进行排序，选择中位数对应的数据点作为切分点。

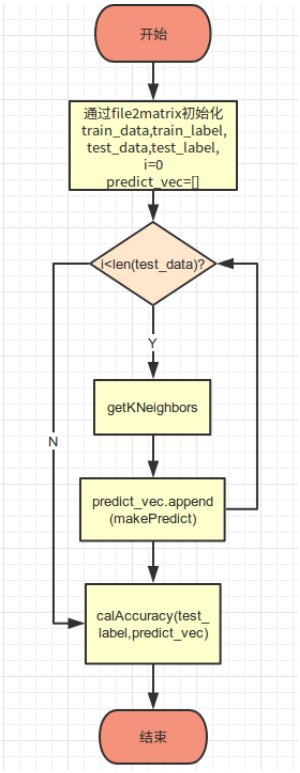
构造好 kd 树后，当给出一个输入数据时就可以通过 kd 树查找出与其距离最近的 k 个训练数据。查找 kd 树算法可以通过递归和非递归两种方式实现，其具体算法在下一部分中详细介绍。kd 树减少搜索计算量的核心在于在每一个有两个子结点的父结点处都先搜索输入数据所在的子节点，再比较当前 k 个近邻中的最大距离和输入数据与当前父结点定义的分割超平

面之间的距离的相对大小，如果 k 个近邻中的最大距离小于输入数据和分割超平面之间的距离，那么就不需要搜索另外一个子结点了。这样就可以减少搜索的范围，提高搜索速度。

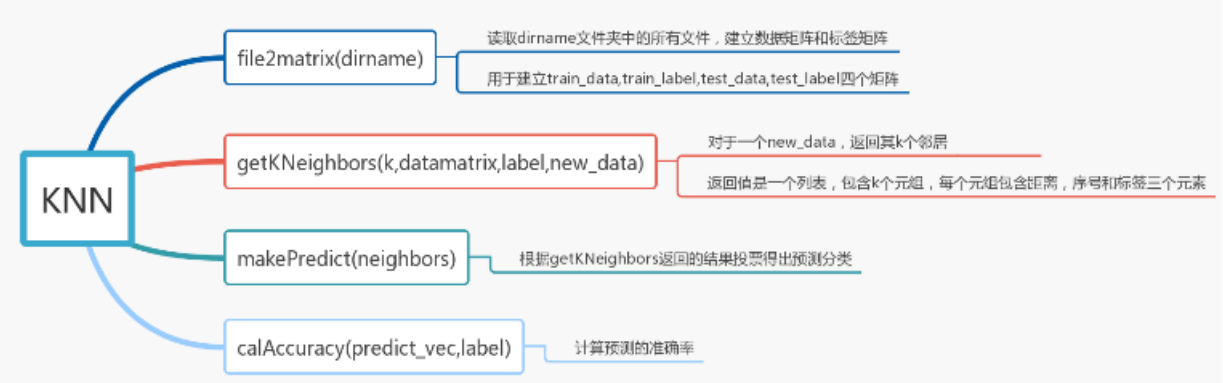
3 实验代码

1. 基本 KNN

普通 KNN 算法首先用 file2matrix 函数读取文件，构造 train_data, train_label, test_data, test_label 四个矩阵，每个训练样本矩阵展开成行向量作为 train_data 矩阵的一行，每个训练样本的标签作为 train_label 列向量的一个元素，同理构造 test_data 和 test_label。然后对于每一个测试样本，根据指定的距离度量选择 k 个邻居，再根据邻居的类别投票确定预测类别。所有测试样本均预测完成后，根据预测向量和标签向量计算预测的准确率。同时统计出对所有测试样本进行预测并计算预测准确率所需要的时间。程序运行流程图如下：



程序中每个函数的功能如下图所示：



其中 getKNeighbors 函数调用了求距离的函数，本实验中使用欧式距离和曼哈顿距离作为距离度量。

2. 带权重的 KNN

与普通 KNN 算法相比，带权重的 KNN 仅在作出预测时的投票环节有所不同，普通 KNN 算法在投票时对于 k 个邻居中的每一个都在对应类别上加 1, 带权重的 KNN 则对 k 个邻居中的每一个根据距离求出权重，在对应类别上加上权重的值。因此，只需要把原来的 makePredict(neighbors) 替换成重写的 makePredict_weighted(neighbors) 即可，其余部分均不变。

3.KD 树实现 KNN

使用 KD 树实现 KNN 是通过 KD 树找出与给定的目标数据距离最近的 k 个邻居，这一任务主要分为两个阶段：第一个阶段是构建 KD 树，即把训练集中所有训练样本存储到 KD 树中；第二个阶段是查找 KD 树，即找出距离最近的 k 个邻居。下面分别对两个阶段的具体实现进行说明。

构建 KD 树：

KD 树用 kdTree 类表示,它由很多个结点组成,每个结点用一个 kdNode 类来表示。kdNode 类中包含的属性有 data(存储数据向量),left(存储左孩子结点),right(存储右孩子结点),split(存储当前切分超平面的切割维度)。把所有训练样本存储到 KD 树中需要调用 ___init___ 构造方法中的 createNode 函数，该函数采用递归的思想，首先计算出 dataset 中所有数据各个维度的方差，选择方差最大的维度为分割维度，然后根据分割维度对所有数据排序，取中位数对应的数据为分割点，左边一半数据集建立左结点，右边一半数据集建立右结点，如此递归地建立结点，直到数据集长度为 0, 即所有数

据均存储到结点上为止。以上过程的代码如下所示：

```
from functools import reduce
class kdNode:
    def __init__(self,data,left,right,split):
        self.data=data
        self.left=left
        self.right=right
        self.split=split

class kdTree:
    def __init__(self,data_set):#data_set的格式是[(i,data[i]) for i in range(len(data))]
        def createNode(data_set):
            if len(data_set)==0:#数据集长度为0时终止
                return None
            else:
                dim=[]
                for i in range(len(data_set[0][1])):#求出数据矩阵的所有列向量
                    dim.append([data_set[j][1][i] for j in range(len(data_set))])
                variance=list(map(getVariance,dim))#求出所有列的方差
                split=variance.index(max(variance))#选择方差最大的维度为切分维度
                data_set.sort(key=lambda x:x[1][split])#根据切分维度对数据排序
                split_data=data_set[len(data_set)//2]
                #把排序后数据集分成两半，分别用来建立当前结点的左右孩子，如此递归地建立kd树
                root=kdNode(split_data,createNode(data_set[:len(data_set)//2]),createNode(data_set[len(data_set)//2
                return root

        def getVariance(l):
            sum_1=reduce((lambda x,y:x+y),l)
            mean=sum_1/len(l)
            sum_2=reduce((lambda x,y:x+y),list(map((lambda x:x**2),l)))
            return (sum_2/len(l)-mean**2)

        self.root=createNode(data_set)
```

查找 KD 树：

查找 KD 树可以采用递归和非递归的查找方法。对于递归的查找方法，首先需要维护一个长度为 k 的优先队列，优先队列中的每个元素是一个元组，代表当前与目的数据距离最近的 k 个邻居之一，元组中包含该邻居与目的数据的距离的相反数和该训练数据。由于 python 中提供的优先队列是最小优先队列，而此处需要最大优先队列，所以距离取相反数。对于每一个结点，首先计算距离并更新优先队列，然后递归查找目的数据所在的子空间并更新优先队列，此时若优先队列中的最大距离小于目的数据到分割超平面的距离，就不需要再查找另一个子空间，否则要查找另一个子空间。该递归过程直到叶子节点处终止。以上过程的代码如下：

```
from heapq import *
def search_knn(tree,point,k):
    result=[]
    def search_node(node,point,result,k):
        if not node:#结点为空时终止
            return
        else:
            node_dist=getEuclidDist(node.data[1],point)#求当前结点保存的数据与目的数据的距离
            item=(-node_dist,node.data)#把距离的相反数和数据打包成元组
            #当优先队列元素已经达到k时，只有当前结点的距离小于优先队列中的最大距离时才把当前结点加入优先队列，替换队列中距离最大的结点
            if len(result)>=k:
                if -node_dist>result[0][0]:
                    heapreplace(result, item)
            else:
                heappush(result, item)#队列元素数量小于k时直接把当前结点加入
            #递归搜索目的数据所在的孩子结点，更新优先队列
            if node.data[1][node.split]>=point[node.split]:
                search_node(node.left,point,result,k)
                next_node=node.right
            else:
                search_node(node.right,point,result,k)
                next_node=node.left
            #如果优先队列中的最大距离小于目的结点到分割超平面的距离，就不需要搜索另一个孩子结点，否则搜索另一个孩子节点
            if -abs(node.data[1][node.split]-point[node.split])>result[0][0] or len(result)<k:
                search_node(next_node,point,result,k)

    search_node(tree.root,point,result,k)
    return result
```

对于非递归的查找方法，首先需要从根结点开始找到目的数据所在的叶子结点，中间按路过的所有结点和叶子结点都压栈，然后对栈里的每个元素进行处理，求距离并更新优先队列。如果优先队列中的最大距离大于目的数据和分割超平面之间的距离，则把该结点的另一个子结点压栈。如果另一个子结点并不是叶子结点，则需要递归地找到目的数据所在的叶子结点，把路过的所有结点和叶子结点都压栈。如此循环直至栈空为止。以上过程代码如下：

```python
def search_knn_norecur(tree,point,k):
    result=[]
    def search_node(node,point,result,k):
        node_stack=[]
        temp_node=node
        #从根结点开始找到目的数据所在的叶子结点，中间所有路过的结点都压入node_stack
        while temp_node:
            node_stack.append(temp_node)
            if temp_node.data[1][temp_node.split]>=point[temp_node.split]:
                temp_node=temp_node.left
            else:
                temp_node=temp_node.right
        #分别处理node_stack中的每个结点，对于每个结点求距离并更新优先队列
        while len(node_stack)>0:
            node=node_stack.pop()
            node_dist=getEuclidDist(node.data[1],point)
            item=(-node_dist,node.data)
            if len(result)>=k:
                if -node_dist>result[0][0]:
                    heapreplace(result, item)
            else:
                heappush(result, item)
            #如果优先队列中的最大距离大于目的结点到分割超平面的距离，把未搜索的另一个子结点压栈
            if -abs(node.data[1][node.split]-point[node.split])>result[0][0] or len(result)<k:
                if node.data[1][node.split]>=point[node.split]:
                    if node.right:
                        node=node.right
                    else:
                        node=None
                else:
                    if node.left:
                        node=node.left
                    else:
                        node=None
            #如果压栈的另一个子结点不是叶子结点，则递归地查找到目的数据所在的叶子结点，路过的所有节点均压栈
            if node:
                while node.left or node.right:
                    node_stack.append(node)
                    if node.data[1][node.split]>=point[node.split]:
                        if not node.left:
                            break
                        else:
                            node=node.left
                    else:
                        if not node.right:
                            break
                        else:
                            node=node.right
                    if node.left==None and node.right==None:
                        node_stack.append(node)
```

在主程序中，需要先用训练数据构造出 kd 树，再把 getKNeighbors 函数替换成 search_knn 函数即可，其他部分不变。

4 实验结果

1. 基本 KNN 中不同 k 值的准确率和运行时间

在主程序的 getKNeighbors 函数中传入不同的 k 值并统计准确率。一次运行的结果如下所示：

```
from time import clock
if __name__=='__main__':
    train_data,train_label=file2matrix('trainingDigits')
    test_data,test_label=file2matrix('testDigits')
    predict_vec=[]
    t0=clock()
    for i in range(len(test_data)):
        neighbors=getKNeighbors(1,train_data,train_label,test_data[i])
        predict_vec.append(makePredict(neighbors))
    acc=calAccuracy(predict_vec,test_label)
    t1=clock()
    print('accuracy:%s' % acc)
    print('time:%s' % (t1-t0))
```

```
accuracy:0.9873150105708245
time:401.077532
```

取 k=1,3,5,7, 距离度量为欧式距离，得到的准确率和运行时间如下表所示：

k	1	3	5	7
Acc	0.9873	0.9894	0.9820	0.9778
Time	401.0775	407.1858	419.9877	401.8931

2. 基本 KNN 中不同距离度量的准确率和运行时间

取 k=3, 分别使用欧式距离和曼哈顿距离作为距离度量，准确率和运行时间如下表所示：

Dist	Euclidean	Manhattan
Acc	0.9894	0.9894
Time	407.1858	175.7915

3. 带权重 KNN 中不同 k 值的准确率和运行时间

取 k=1,3,5,7, 距离度量为欧式距离，得到在不同 k 值下带权重 KNN 的准确率和运行时间，分别与基本 KNN 的准确率和运行时间绘制到同一表格中。以下两个表格分别表示了带权值和不带权值两种方法下 k 取不同值时的准确率和运行时间：

k / method	1	3	5	7
KNN	0.9873	0.9894	0.9820	0.9778
weighted_KNN	0.9873	0.9873	0.9905	0.9894

k / method	1	3	5	7
KNN	401.0775	407.1858	419.9877	401.8931
weighted_KNN	399.4201	402.3921	409.9475	413.7093

4.kd 树实现 KNN 不同 k 值的准确率和运行时间

用 kd 树实现 KNN 时，统计的运行时间包括构造 kd 树的时间和搜索 kd 树直至计算出预测准确率的时间两部分。取 k=3,5，距离度量为欧式距离，分别求出递归搜索和非递归搜索两种情况下的准确率和运行时间。一次运行的结果如下所示：

```
from time import clock
if __name__=='__main__':
    train_data,train_label=file2matrix('trainingDigits')
    test_data,test_label=file2matrix('testDigits')
    dataset=[(i,test_data[i]) for i in range(len(test_data))]
    t0=clock()
    kd = kdTree(dataset)
    predict_vec=[]
    for i in range(len(test_data)):
        result=search_knn(kd,test_data[i],3)
        neighbors=[(-j[0],j[1][0],test_label[j[1][0]]) for j in result]
        predict_vec.append(makePredict(neighbors))
    acc=calAccuracy(predict_vec,test_label)
    t1=clock()
    print('accuracy:%s' % acc)
    print('time:%s' % (t1-t0))
```

```
accuracy:0.9873150105708245
time:204.93587100000013
```

准确率和运行时间如下表所示：

method \ k	3	5
KNN	0.9894	0.9820
weighted_KNN	0.9873	0.9905
kd 树 (递归搜索)	0.9873	0.9810
kd 树 (非递归搜索)	0.9873	0.9810

method \ k	3	5
KNN	407.1858	419.9877
weighted_KNN	402.3921	409.9475
kd 树 (递归搜索)	204.9359	201.8857
kd 树 (非递归搜索)	202.6211	202.4897

5 实验结论

对实验结果进行分析，可以得出以下结论：

1. 不同 k 值对准确率和运行时间的影响

由基本 KNN 和带权重 KNN 中不同 k 值下的准确率表格可以发现，k 太大或太小均会导致准确率降低，因此 k 存在一个最优值使得准确率最大。这可以理解为 k 过小时算法对噪声过于敏感，k 过大时类别之间的界限变得模糊。对于基本 KNN 算法，k 的最优值为 3, 对于带权重的 KNN 算法，k 的最优值是 5。

对于基本 KNN 和带权重 KNN，基本上满足 k 越大运行时间越长，这可以理解为 k 越大投票时会花费更多的时间。

2. 不同距离度量对准确率和运行时间的影响

可以发现在基本 KNN 中 k=3 时使用欧式距离和曼哈顿距离作为距离度量时准确率完全一样，但是使用曼哈顿距离度量的运行时间远小于欧式距离度量的运行时间。这种情况出现的可能原因是由于数据向量中的每一位均为 0 或 1, 欧式距离和曼哈顿距离几乎是等效的，但是计算欧式距离更加耗费时间。

3. 加入权重对准确率和运行时间的影响

可以发现与基本 KNN 相比，加入权重之后准确率稍微有一点提高，但是提高的程度很小。运行时间基本没有太大变化。基本 KNN 在本分类任务上的准确率已经很好，加入权重并没有带来太多改善。

4. 使用 kd 树对准确率和运行时间的影响

可以发现与基本 KNN 相比，使用 kd 树搜索 k 近邻对准确率几乎没有影响，但运行时间降低为原来的一半，说明 kd 树可以减少搜索次数，提高运行效率。递归和非递归两种搜索方式几乎没有区别。