MNIST 手写字符识别

班级:	自动化 1904
学号:	20194285
姓名:	杨振宇

目录

1.模型阐述：

1.1 数据集解析：

MNIST 数据集从 openml.org 网站下载的数据总计 70000 样本，下载数据为 arff 格式，为了便于数据提取，使用 weka 软件将其转化为 csv 格式，并将 label 列从最后一列转为第一列。当需要图像格式提取时，使用 np.reshape 将 784*1 转为 28*28 的格式。数据文件下载链接如下
https://pan.baidu.com/s/1x6sqgfvll2m6vBR6p09fdQ?pwd=8rqd 提取码: 8rqd
复制这段内容后打开百度网盘

1.2KNN 算法：

参考了李航的《统计学习方法》，p37 的内容，
算法流程如下：

算法 3.1（k 近邻法）

输入：训练数据集

$$T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\}$$

其中，$x_i\in\mathcal{X}\subseteq\mathbf{R}^n$ 为实例的特征向量，$y_i\in\mathcal{Y}=\{c_1,c_2,\cdots,c_K\}$ 为实例的类别，$i=1,2,\cdots,N$；实例特征向量 x；

输出：实例 x 所属的类 y.

（1）根据给定的距离度量，在训练集 T 中找出与 x 最邻近的 k 个点，涵盖这 k 个点的 x 的邻域记作 $N_k(x)$；

（2）在 $N_k(x)$ 中根据分类决策规则（如多数表决）决定 x 的类别 y：

$$y=\arg\max_{c_j}\sum_{x_i\in N_k(x)}I(y_i=c_j),\quad i=1,2,\cdots,N;\quad j=1,2,\cdots,K \tag{3.1}$$

式 (3.1) 中，I 为指示函数，即当 $y_i=c_j$ 时 I 为 1，否则 I 为 0. ■

k 近邻法的特殊情况是 $k=1$ 的情形，称为最近邻算法. 对于输入的实例点（特征向量）x，最近邻法将训练数据集中与 x 最邻近点的类作为 x 的类.

1.3 决策树算法：

参考李航的《统计学习方法》，p63 到 p65 的内容，
ID3 算法流程如下：
输入训练数据集 D. 特征集 A ， 阈值ϵ，
输出：决策树 T
(1)若 D 中所有实例属于同一类 C_k 则 T 为单结点树，并将类 C_k 作为该结点的类标记，返回 T ，
(2) 若$A=\emptyset$，则 T 为单结点树，并将 D 中实例数最大的类 C_k 作为该结点的类标记，返回 T ，
(3) 否则， 计算 A 中各特征对 D 的信息增益， 选择信息增益最大的特征 Ag ；

(4) 如果 Ag 的信息增益小于阈值ϵ， 则置 T 为单结点树， 并将 D 中实例数最大的类 C_k 作为该结点的类标记，返回 T ，
(5) 否则，对 Ag 的每一可能值 ai，依 Ag =ai 将 D 分割为若干非空子集 Di，将 Di 中实例数最大的类作为标记，构建子结点，由结点及其于结点构成树 T. 返回 T。
(6) 对第 i 个子结点，以 Di 为训练集，以 A -{Ag}为特征集，递归地调用步(1)- 步(5)，得到子树 Ti，返回 Ti。

C4.5 算法流程如下
输入训练数据集 D. 特征集 A ， 阈值ϵ，
输出：决策树 T
(1)若 D 中所有实例属于同一类 C_k 则 T 为单结点树，并将类 C_k 作为该结点的类标记，返回 T ，
(2) 若$A=\emptyset$，则 T 为单结点树，并将 D 中实例数最大的类 C_k 作为该结点的类标记，返回 T ，
(3) 否则， 计算 A 中各特征对 D 的信息增益比， 选择信息增益比最大的特征 Ag ；
(4) 如果 Ag 的信息增益比小于阈值ϵ， 则置 T 为单结点树， 并将 D 中实例数最大的类 C_k 作为该结点的类标记，返回 T ，
(5) 否则，对 Ag 的每一可能值 ai，依 Ag =ai 将 D 分割为若干非空子集 Di，将 Di 中实例数最大的类作为标记，构建子结点，由结点及其于结点构成树 T. 返回 T。
(6) 对第 i 个子结点，以 Di 为训练集，以 A -{Ag}为特征集，递归地调用步(1)- 步(5)，得到子树 Ti，返回 Ti。
在编程中也按照上述算法流程进行编写

2.关键程序解析：

2.1KNN 程序

储存训练集前 k 个点，并计算欧氏距离，计算除了前 k 个点之外点的方法与之相同：

```
for i in range(k):
    label = train_labels[i]
    train_vec = trainset[i]
    dist = np.linalg.norm(train_vec - test_vec)        # 计算两个点的欧氏距离
    knn_list.append((dist,label))
```

找出 k 个点中距离最远的点 max，如果 k 之外的点与之比较距离小于 max，则替换它：循环进行完所有的点即可找出距离测试样本最近的 k 个点

```
if max_index < 0:
    for j in range(k):
        if max_dist < knn_list[j][0]:
            max_index = j
            max_dist = knn_list[max_index][0]
```

```
if dist < max_dist:
    knn_list[max_index] = (dist,label)
    max_index = -1
    max_dist = 0
```

统计 k 个最近的点的标签，选择数量最多的标签作为预估值:

```
# 统计选票
class_total = 10
class_count = [0 for i in range(class_total)]
for dist,label in knn_list:
    class_count[label] += 1

# 找出最大选票
max_vote = max(class_count)

# 找出最大选票标签
for i in range(class_total):
    if max_vote == class_count[i]:
        predict.append(i)
        break
```

2.2 决策树程序

数据集的预处理:

```
# 二值化
def binaryzation(img):
    cv_img = img.astype(np.uint8)
    cv2.threshold(cv_img, 50, 1, cv2.THRESH_BINARY_INV, cv_img)
    return cv_img

def binaryzation_features(trainset):
    features = []
#将 784*1 的数据转化为 28*28 的数据并二值化
    for img in trainset:
        img = np.reshape(img, (28, 28))
        cv_img = img.astype(np.uint8)
        img_b = binaryzation(cv_img)
        features.append(img_b)

    features = np.array(features)
    features = np.reshape(features, (-1, feature_len))
```

```
    return features
```

经验熵的计算:

```
def calc_ent(x):
    x_value_list = set([x[i] for i in range(x.shape[0])])
    ent = 0.0
    for x_value in x_value_list:
        p = float(x[x == x_value].shape[0]) / x.shape[0]
        logp = np.log2(p)
        ent -= p * logp
    return ent
```

条件熵的计算:

```
def calc_condition_ent(x, y):
    x_value_list = set([x[i] for i in range(x.shape[0])])
    ent = 0.0
    for x_value in x_value_list:
        sub_y = y[x == x_value]
        temp_ent = calc_ent(sub_y)
        ent += (float(sub_y.shape[0]) / y.shape[0]) * temp_ent
    return ent
```

信息增益计算:

```
def calc_ent_grap(x, y):
    base_ent = calc_ent(y)
    condition_ent = calc_condition_ent(x, y)
    ent_grap = base_ent - condition_ent
    return ent_grap
```

决策树类的设计

```
class Tree(object):
    def __init__(self, node_type, Class=None, feature=None):
        self.node_type = node_type  # 节点类型为叶子节点或内部节点
        self.dict = {}  # dict 的键表示特征 Ag 的可能值 ai，值表示根据 ai 得到的子树
        self.Class = Class  # 叶节点表示的类，若是内部节点则为 none
        self.feature = feature  # 表示当前的树即将由第 feature 个特征划分（即第 feature 特征是使得当前树中信息增益最大的特征）

    def add_tree(self, key, tree):
        self.dict[key] = tree

    def predict(self, features):
        if self.node_type == 'leaf' or (features[self.feature] not
```

```
in self.dict):
            return self.Class

        tree = self.dict.get(features[self.feature])
        return tree.predict(features)
```

ID3 算法主体

```
def recurse_train(train_set, train_label, features):
    LEAF = 'leaf'
    INTERNAL = 'internal'

    # 步骤 1—如果训练集 train_set 中的所有实例都属于同一类 Ck
    label_set = set(train_label)
    if len(label_set) == 1:
        return Tree(LEAF, Class=label_set.pop())

    # 步骤 2—如果特征集 features 为空
    class_len = [(i, len(list(filter(lambda x: x == i,
train_label)))) for i in range(class_num)]  # 计算每一个类出现的个数
    (max_class, max_len) = max(class_len, key=lambda x: x[1])

    if len(features) == 0:
        return Tree(LEAF, Class=max_class)

    # 步骤 3—计算信息增益,并选择信息增益最大的特征
    max_feature = 0
    max_gda = 0
    D = train_label
    for feature in features:
        A = np.array(train_set[:, feature].flat)  # 选择训练集中的第
feature 列（即第 feature 个特征）
        gda = calc_ent_grap(A, D)
        if gda > max_gda:
            max_gda, max_feature = gda, feature

    # 步骤 4—信息增益小于阈值
    if max_gda < epsilon:
        return Tree(LEAF, Class=max_class)

    # 步骤 5—构建非空子集
    sub_features = list(filter(lambda x: x != max_feature,
features))
    tree = Tree(INTERNAL, feature=max_feature)
```

```
    max_feature_col = np.array(train_set[:, max_feature].flat)
    feature_value_list = set(
        [max_feature_col[i] for i in
range(max_feature_col.shape[0])])  # 保存信息增益最大的特征可能的取值
(shape[0]表示计算行数)
    for feature_value in feature_value_list:

        index = []
        for i in range(len(train_label)):
            if train_set[i][max_feature] == feature_value:
                index.append(i)

        sub_train_set = train_set[index]
        sub_train_label = train_label[index]

        sub_tree = recurse_train(sub_train_set, sub_train_label,
sub_features)
        tree.add_tree(feature_value, sub_tree)

    return tree
```

C4.5 算法主体与 ID3 相同，区别体现在步骤三信息增益比的计算

```
# 步骤 3——计算信息增益,并选择信息增益最大的特征
max_feature = 0
max_gda = 0
D = train_label
for feature in features:
    A = np.array(train_set[:, feature].flat)  # 选择训练集中的第
feature 列（即第 feature 个特征）
    gda = calc_ent_grap(A, D)
    if calc_ent(A) != 0:  # 计算信息增益比(与 ID3 的区别)
        gda /= calc_ent(A)
    if gda > max_gda:
        max_gda, max_feature = gda, feature
```

训练和预测环节

```
#训练环节
def train(train_set, train_label, features):
    return recurse_train(train_set, train_label, features)

#预测环节
def predict(test_set, tree):
    result = []
```

```
    for features in test_set:
        tmp_predict = tree.predict(features)
        result.append(tmp_predict)
    return np.array(result)
```

2.3 五折交叉验证（以 knn 为例）

设定 k 折交叉验证的折数，并设定随机状态为随机

```
kf = KFold(n_splits=5, random_state=None)
```

展示划分情况，将原训练集进行分配，并在分配好新训练集和验证集上进行训练和验证。

```
# 显示 5 折交叉验证的具体划分情况
for train_index, test_index in kf.split(x_train,y_train):
    print("Train:", train_index, "Validation:", test_index)
    X_train, X_test = x_train[train_index], x_train[test_index]
    Y_train, Y_test = y_train[train_index], y_train[test_index]

i = 1
for train_index, test_index in kf.split(x_train,y_train):
    print('\n{} of kfold {}'.format(i, kf.n_splits))
    X_train, X_test = x_train[train_index], x_train[test_index]
    Y_train, Y_test = y_train[train_index], y_train[test_index]
    pred_test = KNN(X_test, X_train, Y_train)
    score = accuracy_score(Y_test, pred_test)
    recall = recall_score(Y_test, pred_test,average='macro')
    precision = precision_score(Y_test, pred_test,average='macro')
    kfoldscore.append(score)
    kfoldrecall.append(recall)
    kfoldpre.append(precision)
    print("此折的准确率为 %f" % score)
    print("此折的召回率为 %f" % recall)
    print("此折的为精确度 %f" % precision)
    i += 1
```

3 结果分析:

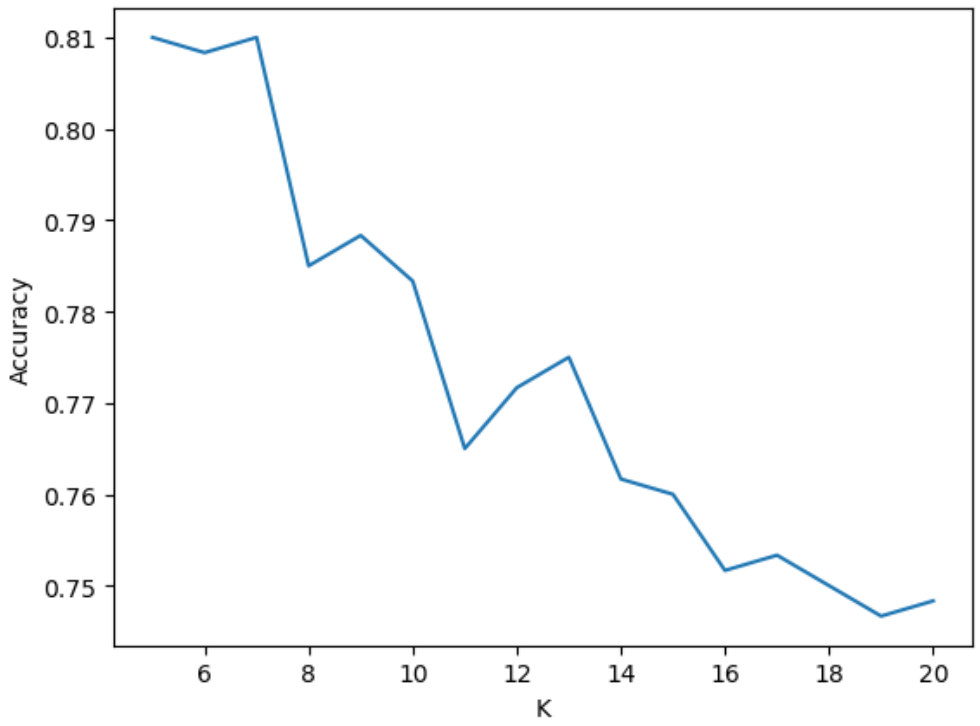
（注：下文的图片均与图片上方段落对应）

3.1 KNN 训练结果:

投票过程：
例子如下，表示 k=10，训练集数量为 600 时第一折前两个验证数据投票结果

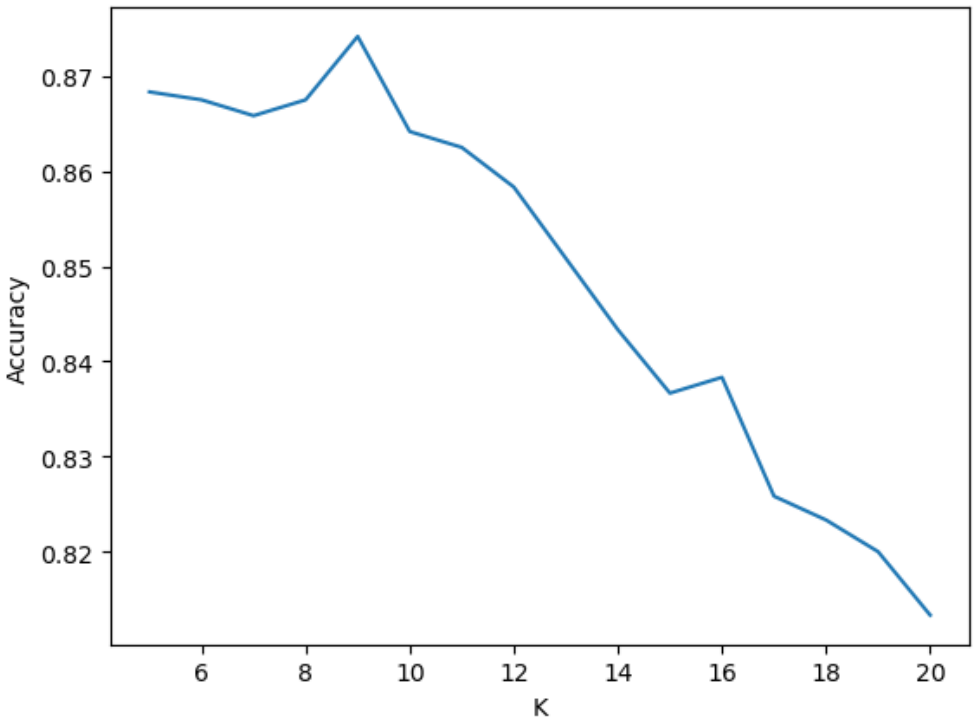
1 of kfold 5
0
[0, 0, 0, 4, 0, 3, 2, 0, 1, 0]
1
[0, 6, 3, 0, 0, 0, 0, 0, 0, 1]

KNN 训练过程：
Train 数量 600
最优 k: 5
最优 k 的测试集准确率: 0.81
[0.81, 0.8083333333333333, 0.81, 0.785, 0.7883333333333333, 0.7833333333333333, 0.765, 0.7716666666666666, 0.775, 0.7616666666666667, 0.76, 0.7516666666666667, 0.7533333333333333, 0.75, 0.7466666666666667, 0.7483333333333333]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

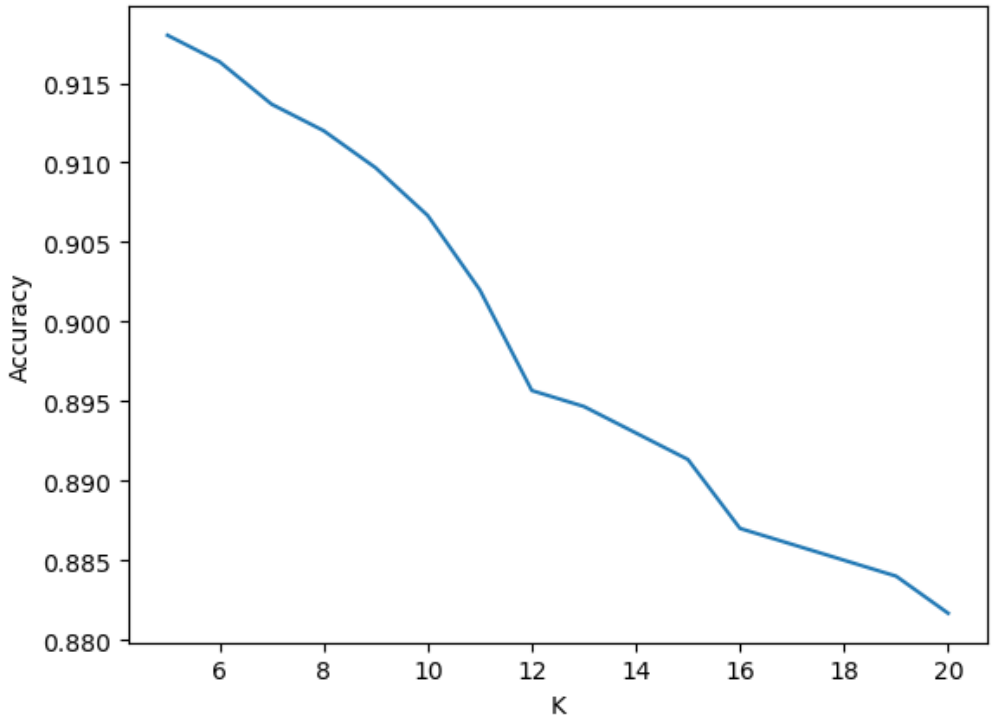


Train 数量 1200
最优 k: 9
最优 k 的测试集准确率: 0.8741666666666666
[0.8683333333333333, 0.8675, 0.8658333333333333, 0.8675, 0.8741666666666666, 0.8641666666666666, 0.8625, 0.8583333333333333, 0.8508333333333333, 0.8433333333333334, 0.8366666666666667, 0.8383333333333334, 0.8258333333333333, 0.8233333333333334, 0.82, 0.8133333333333334]

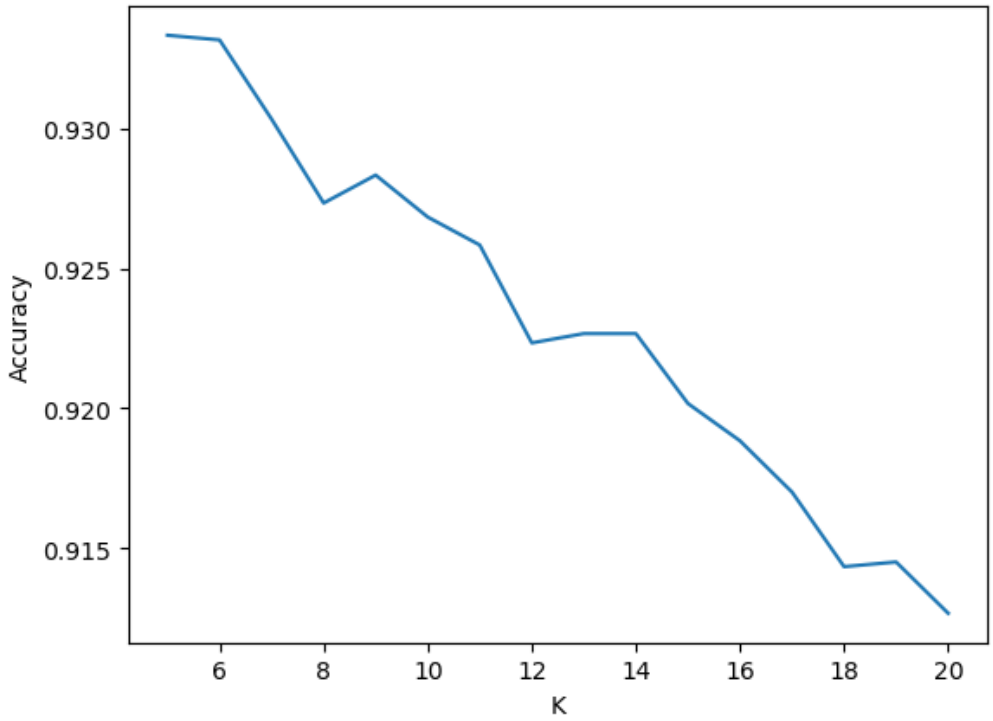
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]



Train 数量 3000
最优 k: 5
最优 k 的测试集准确率: 0.9179999999999999
[0.9179999999999999, 0.9163333333333333, 0.9136666666666666, 0.912, 0.9096666666666666, 0.9066666666666666, 0.902, 0.8956666666666667, 0.8946666666666667, 0.893, 0.8913333333333333, 0.887, 0.886, 0.885, 0.884, 0.8816666666666667]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]



Train 数量 6000 时：
最优 k: 5
最优 k 的测试集准确率: 0.9333333333333333
[0.9333333333333333, 0.9331666666666667, 0.9303333333333333, 0.9273333333333333, 0.9283333333333333, 0.9268333333333333, 0.9258333333333334, 0.9223333333333333, 0.9226666666666666, 0.9226666666666666, 0.9201666666666667, 0.9188333333333333, 0.917, 0.9143333333333333, 0.9145, 0.9126666666666667]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]



（注，当训练集数量为 6000 时，每轮次的训练时间为 240 秒，每一折平均 48s，但是时间是等比例增长的）

我们通过模型阐述可以知道，KNN 模型当数据集数量很大时，耗时会大幅度增加，所以我选择在较小数据集量，进行五折交叉验证，以选择合适的参数，通过上述图像的对比，我们可以看出，当 k 为 5 时，knn 的决策效果最好，因此我们选择的 K=5 作为测试的最优解。

3.2 KNN 测试结果:

我们将参数调整为 K=5 之后，在 10000 个测试集上进行模型验证，结果如下：

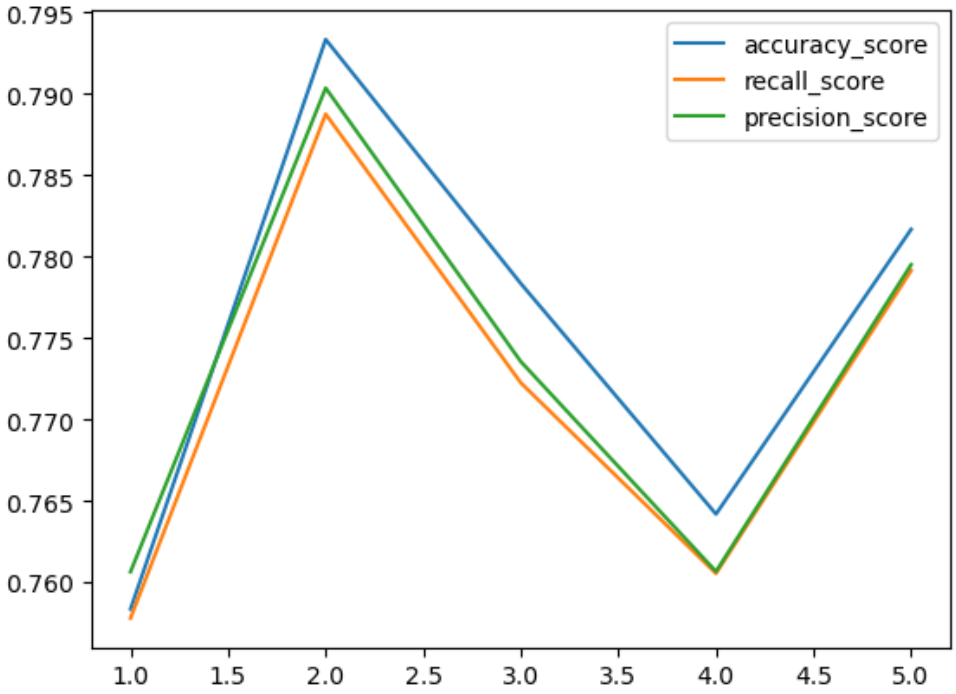
测试集上准确率为　0.973
测试集上总共用时　5819.09951543808　second

准确率为 97.3%，耗时为 97 分钟

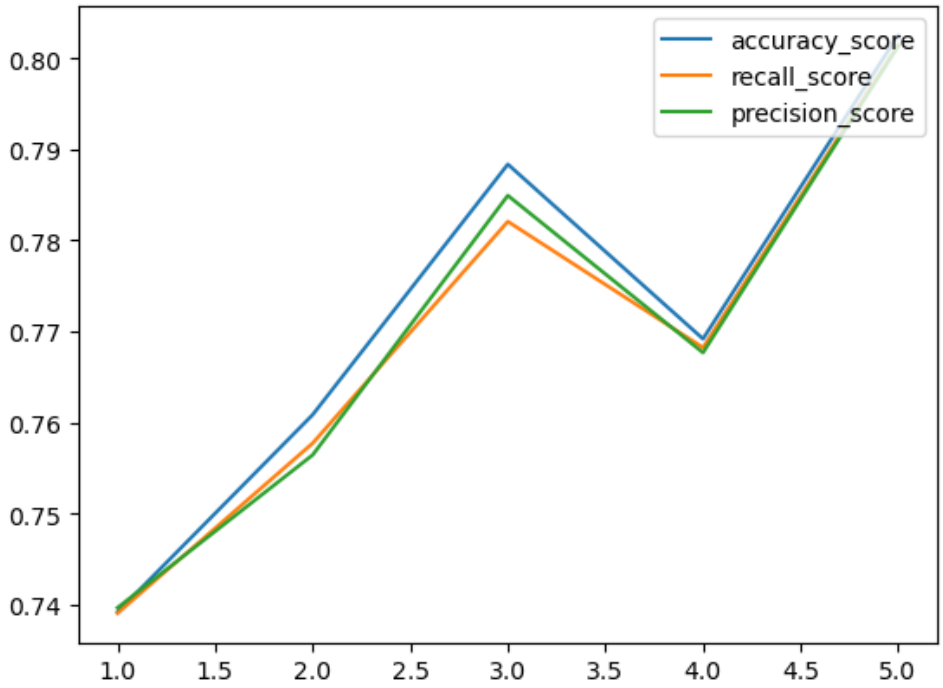
3.3 决策树训练结果：

决策树 ID3　训练数量 6000
其中一折的用时：
5 of kfold 5
训练耗时 40.823717 seconds
Start predicting...
predicting cost 0.007979 seconds
此折的准确率为 0.781667
此折的召回率为 0.779132
此折的为精确度 0.779497
平均准确率为 0.775167

图像如下

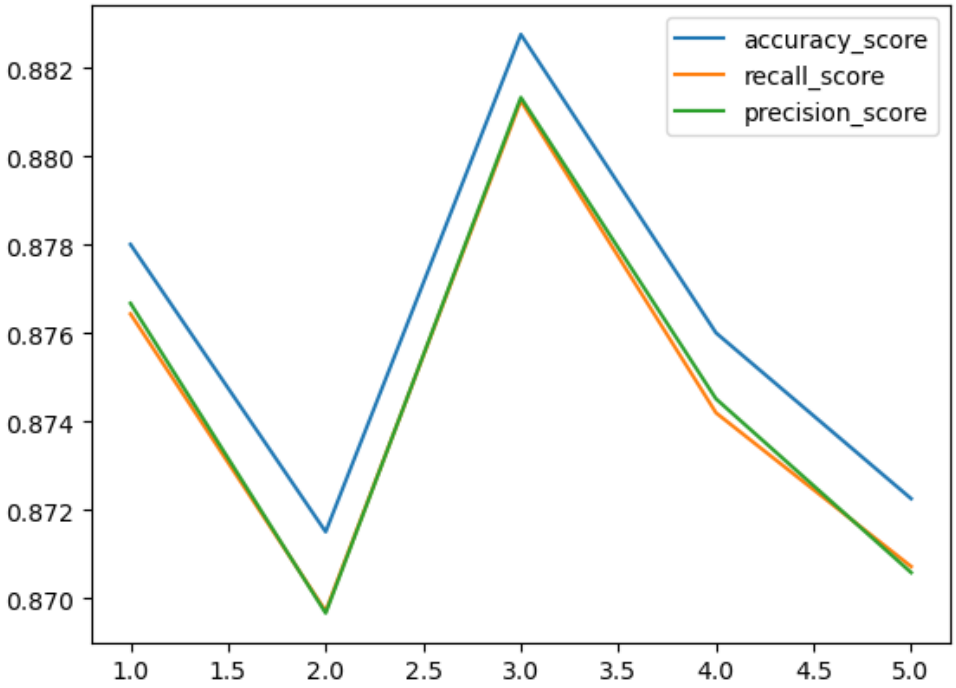


决策树 C4.5 数据集数量 6000
其中一折的用时
5 of kfold 5
训练耗时 192.743985 seconds
开始预测
预测耗时 0.034868 seconds
此折的准确率为 0.802500
此折的召回率为 0.801462
此折的为精确度 0.801301
平均准确度为 0.772000
图像如下



通过对两类决策树算法的比较，二者在相同数据集数量情况下结果相似，但是时间相差巨大，出于时间利益考量，我们选择 ID3 作为决策树的模型进行 60000 个训练集的训练，结果为：

决策树 ID3 训练数量 60000
5 of kfold 5
训练耗时 520.671932 seconds
开始预测
预测耗时 0.137660 seconds
此折的准确率为 0.872250
此折的召回率为 0.870715
此折的为精确度 0.870579
平均准确度为 0.876100



可以看出在五折交叉验证下，准确率相较于测试集为 6000 时显著增加

3.4 决策树测试结果:

在测试集数量为 10000，选择 ID3 算法时，决策树模型的训练结果为：

```
开始预测
预测耗时 667.139955 seconds
测试集上准确率为  0.8816
```

花费时间为 11 分钟，准确率为 88.16%

4.结论与讨论

4.1 结论

对比 KNN 与决策树 ID3 算法的，我们可以发现，KNN 的准确率 97.3%明显高于决策树 ID3 算法的 88.16%，但是 KNN 算法所消耗时间极其的长，比决策树时间长多近一个半小时。再对比与在较小数量的训练集上交叉验证的表现，我们可以得出如下结论：当数据集偏小时，KNN 优于决策树，但当数据集庞大，类似于 mnist 集时，更应该选用决策树算法，以更快地求解。

4.2 讨论

通过结论我们可以看出，如果能够缩短 knn 的运算时间，那么 knn 的算法会

明显优于决策树算法，为此查阅资料，得到如下解决办法：
在原始程序中我们选择的特征就是图像 28*28 共计 784 维的特征，通过之前图像识别课程学习，我了解到对于图片，可提取 hog 特征值，以削减特征值数量。
方向梯度直方图（Histogram of Oriented Gradient, HOG）特征是一种在计算机视觉和图像处理中用来进行物体检测的特征描述子。它通过计算和统计图像局部区域的梯度方向直方图来构成特征。
使用 python 提取 hog 特征值的方法如下：
从 xml 文件获取 hog 参数，本次设定的参数特征共 324 个，相原特征 784 个减少了 460 个

```
def get_hog_features(trainset):
    features = []

    hog = cv2.HOGDescriptor('../hog.xml')

    for img in trainset:
        img = np.reshape(img,(28,28))
        cv_img = img.astype(np.uint8)

        hog_feature = hog.compute(cv_img)
        # hog_feature = np.transpose(hog_feature)
        features.append(hog_feature)

    features = np.array(features)
    features = np.reshape(features,(-1,324))
```

应用之后训练集数据量为 6000 情况下五折交叉验证的时间为
测试集上准确率为 0.9873
测试集上总共用时 3955.5029673576355 second
时间从 90 分钟缩短 65 分钟，虽然有所改善，但依旧十分耗时，继续探究，发现可以使用 KD 树搜索算法来改进搜索 k 近邻点，
Kd 树算法流程如下：
输入 k 维空间数据集 T={x1,x2,x3,,,xn},
其中 xi= $(xi^{(1)}, xi^{(2)},...xi^{(k)})^T$ ， 其中 i =1.2,…n
输出 kd 树
(1)开始构造根结点，根结点对应于包含 T 的 k 维空阔的超矩形区域
选择.$x^{(1)}$ 为坐标轴，以 T 中所有实例的 $x^{(1)}$ 坐标的中位数为切分点，将根结点对应的超矩形区域切分为两个于区域.切分由通过切分点并与坐标轴 $x^{(1)}$ 垂直的超平面实现
由根结点生成深度为 1 的左、右于结点·左子结点对应坐标 x(1) /J、于切分点的于区域，右于结点对应于坐标 x(1)大于切分点的子区域.
将落在切分超平面上的实例点保存在根结点.
(2) 重复：对深度为 j 的结点，选择 $x^{(1)}$ 为切分的坐标轴， l=j(modk)+1，
以该结点的区域中所有实例的 $x^{(1)}$ 坐标的中位数为切分点，将该结点对应的超

矩形区域切分为两个于区域.切分自通过切分点并与坐标轴川}垂直的超平面实现
由该结点生成深度为 j+1 的左、右于结点:左于结点对应坐标 $x^{(1)}$ 小于切分点的子区域，右子结点对应坐标 $x^{(1)}$。大于切分点的子区域
将落在切分超平面上的实例点保存在该结点
(3)直到两个于区域没有实例存在时停止.从而形成 kd 树的区域划分

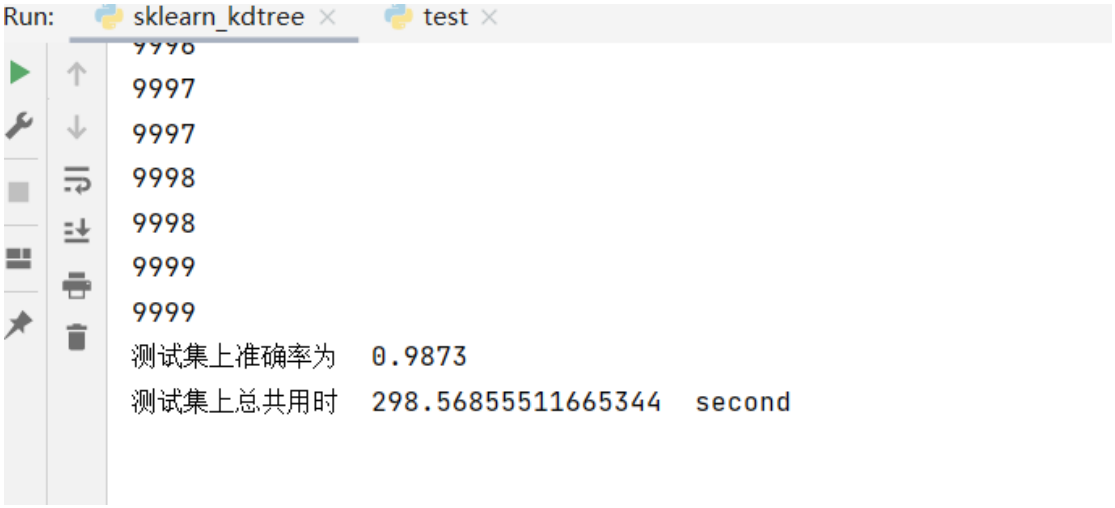
迫于时间限制，未能完成 kd 树的手写构造，故此通过对比 sklearn 中 KDTree 训练耗时来说明：
代码如下

```
def KNN_KDthree(testset,trainset,train_labels,test_labels):
    predict = []
    count = 0
    tree = KDTree(trainset)

    for i in range(0, len(test_labels)):
        print(i)
        test_vec = testset[[i]]
        # 输出当前运行的测试用例坐标，用于测试
        print(count)
        count += 1
        knn_list = []
        dist_to_knn, indx_knn = tree.query(test_vec, k=k_value)
        for i in range(0,5):
           labels = train_labels[indx_knn[0][i]]
           knn_list.append(labels)
        # 统计选票
        class_total = 10
        class_count = [0 for i in range(class_total)]
        for train_label in knn_list:
            class_count[train_label] += 1
        # 找出最大选票
        max_vote = max(class_count)
        # 找出最大选票标签
        for i in range(class_total):
            if max_vote == class_count[i]:
                predict.append(i)
                break


    return np.array(predict)
```

结合 hog 特征值提取，测试结果如下



通过这一结果，可以说明，经过特征值优化和 KD 树划分的 KNN 算法无论在时间上还是准确性上都会更优于决策树，此时选择 knn 做分类器更合适