華中科技大學

课程实验报告

题目: KNN 的 Python 简单实现

课程名称:	机器学习
专业班级:	CS1703
学　　号:	U201714668
姓　　名:	葛松
指导教师:	李玉华
报告日期:	2020 年 5 月 28 日

计算机科学与技术学院

目录

1 实验一

1.1 实验目的与要求

1. 理解 KNN 算法，及其具体实现
2. 熟悉 Python 语言的使用
3. 熟悉图片的处理方法

1.2 实验内容

1. 动手实现 KNN 算法，语言限定为 python。数据集使用 MNIST 数据集。
2. 最终需要实现以下功能
   - 输入若干测试图片, 输出对应每张图片 k 近邻的图片。
   - 绘制 knn 算法的训练 misclassification rate 曲线, 并做出分析
   - 可以自由发挥 (人脸数据集等等)

1.3 实验方案

以下将整个实验内容分为以下几个部分逐一介绍

- 整体设计
- KNN 核心算法
- 数据集读取
- 输出每张图片 k 临近的图片
- 计算 misclassification rate 并绘图

1.3.1 整体设计

整个实现过程的流程图如下图所示

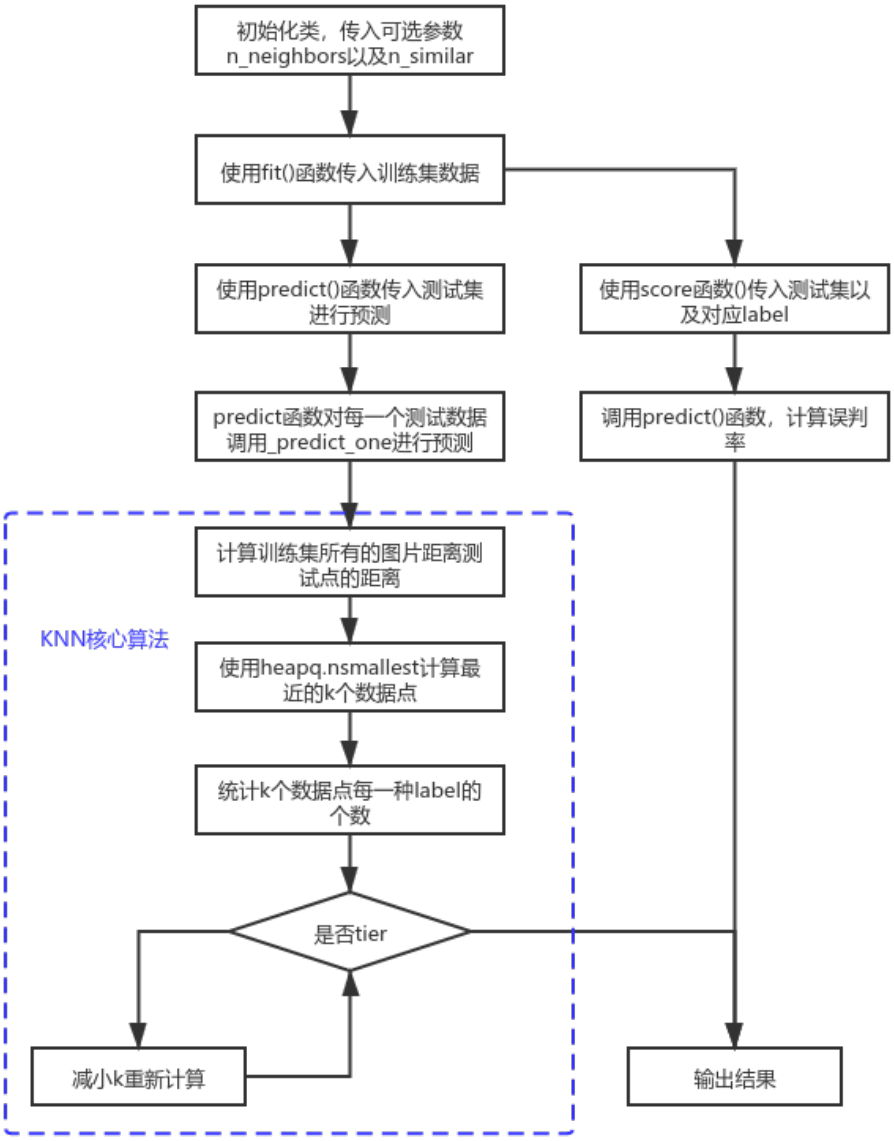


图 1: KNNClassifier流程图

该实验中对 KNN 的实现接口主要参考了sklearn 中的 KNeighborsClassifier ，将 KNN 的实现包装在一个类当中，并实现相关方法作为接口开放，类 `KNNClassifier`的 UML 类图如下所示

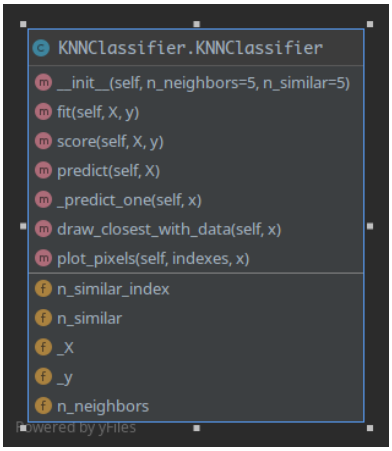


图 2: `KNNClassifier`的 UML 图

以下分别介绍`KNNClassifier`的属性以及方法

- 属性 `_X`
  用于存放数据集的数据部分，也即图片的 numpy 表示
- 属性 `_y`
  用于存放数据集的标签部分，也即每一个图片的数字
- 属性 `n_similar`
  用来配置显示最临近的`n_similar`个图片的具体数目
- 属性 `n_similar_index`
  保存距离最近的`n_similar`个图片在`_X`中的下标
- 方法 `fit(X,y)`
  用于传入数据，实际上就是将内部的`_X`以及`_y`设置为对应值
- 方法 `score(X,y)`
  对测试数据`X`，`y`进行打分，输出 misclassification rate
- 方法 `predict(X)`
  预测数据`X`，其中参数`X`为 numpy 数组，表示多个图片
- 方法 `_predict_one(x)`
  与方法`predict(X)`配合，用于预测单个图片，其中参数`x`为单个图片的 numpy 表示
- 方法 `draw_cloest_with_data(x)`
  用于绘出与图片`x`最接近的`n_similar`个图片
- 方法 `plot_pixels(indexes,x)`
  用于和`draw_cloest_with_data(x)`配合

1.3.2 KNN 核心算法

KNN 的核心算法主要由_predict_one来实现，主要代码以及注释如下

_predict_one方法

```
def _predict_one(self, x):
    k = self.n_neighbors
    # 只判断一个,x为728维的数组
    dist_list = list(np.sqrt(((self._X - x) ** 2).sum(axis=1)))
    # (距离，标签，index)三元组，其中index用于定位原来
    dist_list = list(zip(dist_list, self._y, list(x for x in range(0, len(self._X)))))
    # 暂时不考虑第k和第k+1个元素的距离相等的情况
    # 首先通过nsmallest函数获得最近的k个元素
    dist_list = nsmallest(k, dist_list)
    # 再通过np.unique函数返回前k个数据点中每一种标签的数目
    label, counts = np.unique([x[1] for x in dist_list], return_counts=True)
    lable_counts = list(zip(counts, label))
    # 找到数目最多的标签
    lable_counts.sort(reverse=True)
    result = lable_counts[0][1]
    while k > 1 and len(lable_counts) > 1 and lable_counts[0][0] == lable_counts[1][0]:
        # 当最大和第二大相等的时候，需要缩小k，重新进行计算
        if dist_list[k - 1] == lable_counts[0][1]:
            result = lable_counts[1][1]
            break
        elif dist_list[k - 1] == lable_counts[1][1]:
            result = lable_counts[0][1]
            break
        k -= 1
    self.n_similar_index = [x[2] for x in dist_list[:self.n_similar]] #
        选出最近的n的点的index
    return result
```

KNN 算法主要步骤就是

- 计算距离 (本次实验采用了欧氏距离)
- 排序
- 对前 k 个进行统计
- 如果出现个数最多的两个标签数目相等则缩小 k 值

其中计算距离通过numpy的广播机制可以非常方便的计算出来

```
dist_list = list(np.sqrt(((self._X - x) ** 2).sum(axis=1)))
```

排序这里采用了nsmallest函数, 而不是简单的调用sort方法，由于nsmallest函数通过构建最小堆的形式来得到 k 个最小的元组, 当 k 很小, 而整体的个数 N 很大的时候非常合适, 如本次试验中训练集为 6000 但是 k 只有 1 到 20, 而对每一种标签的统计也可以通过numpy的unique()函数得到

```
# 返回前k个数据点中每一种标签的数目
label, counts = np.unique([x[1] for x in dist_list[:k]], return_counts=True)
```

函数末尾的while循环来解决 tier 的情况，假设两个标签 A，B 的个数相等，且个数在所有的标签当中最大，那么依次从 k 个元素的末尾取出元素，直到检查到标签为 A 或者 B，如果第 i 个元素检查出 A，那么说明当 k 减小到 i-1 时，B 的个数将会超过 A，所以最终结果为 B，反之亦然，使用这种方法的目的是为了避免每一次都调用sort函数导致效率极低

1.3.3 数据集读取

这次实验中的数据集读取直接使用了sklearn中的fetch_openml方法 [1]，可以非常方便的导入numpy格式的 mnist 数据。代码如下所示

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')
```

其中mnist的成员data即为numpy格式的 70000 张手写图片，成员target即为每一张图片对应的标签

1.3.4 输出每张图片 k 临近的图片

在_predict_one方法中排序的同时设置了成员n_similar_index，保存了距离图片x最近的n_similar 个图片的下标，进而在方法draw_closest_with_data(x)使用

方法draw_cloest_with_data以及plot_pixels代码如下所示

draw_closest_with_data方法

```
def draw_closest_with_data(self, x):
    """
    x为数据
    """
    result = self._predict_one(x)
    self.plot_pixels(self.n_similar_index, x)
```

[1]在整个试验中仅此一处使用了sklearn, 仅仅用于导入数据

plot_pixels方法

```python
def plot_pixels(self, indexes, x):
    """
    x 目标图片的点位数据
    indexes 为最近的n个图片的index
    """
    pixels = np.reshape(x, (28, 28))
    plt.figure(figsize=(2, 2))
    plt.axis('off')
    plt.title(f"Target Image")
    plt.imshow(pixels, cmap='gray')
    count = len(indexes)
    rows = int(count ** 0.5)
    columns = int(count / rows + 1)
    fig = plt.figure(figsize=(columns, rows))
    for i in range(count):
        pixels = np.reshape(self._X[[indexes[i]]], (28, 28))
        ax = fig.add_subplot(rows, columns, i + 1)
        ax.title.set_text(f"No.{indexes[i]}")
        plt.axis('off')
        plt.imshow(pixels, cmap='gray')
    fig.tight_layout()
    plt.show()
```

输出结果如下图所示

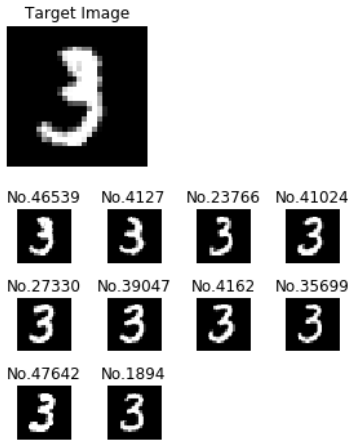


图 3: 绘制最近的n_similar个图片

1.3.5 计算 misclassification rate 并绘图

分别计算 k 从 1 到 20 时的score，也即misclassification rate，再通过matplotlib进行绘制即可具体代码如下

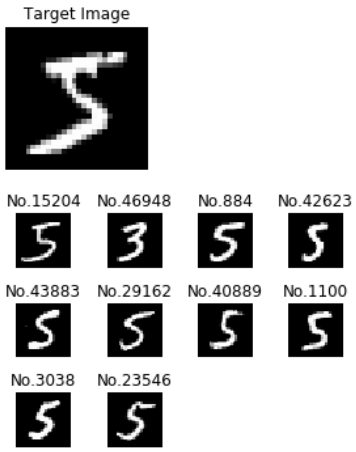
计算 misclassification rate

```
# 分割测试集和训练集
X = mnist.data
y = mnist.target
X_train,X_test,y_train,y_test = X[:6000], X[6000:7000], y[:6000], y[6000:7000]
data_test = []
data_train = []
for k in range(1, 21):
    # 分别测试k从1到20
    KNN = KNNClassifier(n_neighbors=k)
    KNN.fit(X_train, y_train)
    mis_test = KNN.score(X_test, y_test)
    mis_train = KNN.score(X_train, y_train)
    print(mis_test)
    print(mis_train)
    data_test.append(mis_test)
    data_train.append(mis_train)
# 进行绘制
fig, ax = plt.subplots()
ax.plot([x+1 for x in range(20)], data_test, label='Test', marker='.',markersize=8)
ax.plot([x+1 for x in range(20)], data_train, label='Train', marker='*', markersize=8)
ax.set_ylabel('misclassfication rate', fontsize='medium')
ax.set_xlabel('k', fontsize='medium')
plt.xticks([x for x in range(1,21)], [x for x in range(1,21)])
plt.legend()
plt.show()
```

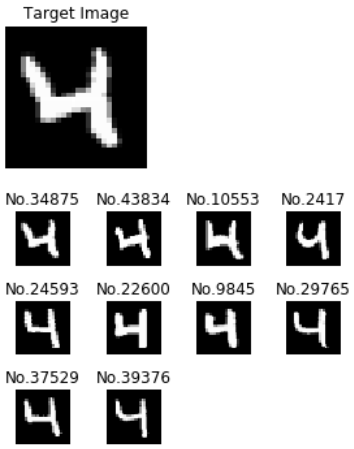
1.4 实验结果

1.4.1 输出每张图片 k 临近的图片

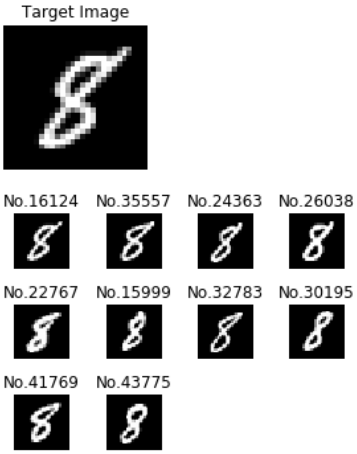
分别测试 mnist 数据集中的几个图片，输入结果如下



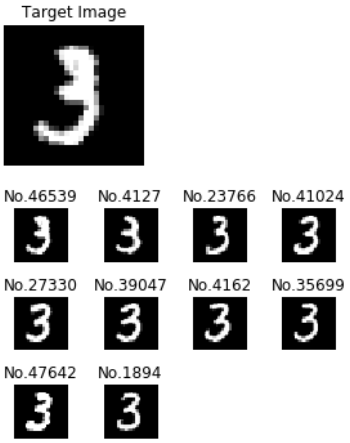
(a) 测试 1



(b) 测试 2



(c) 测试 3



(d) 测试 4

图 4: 最近的`n_similar`个图片输出

可见结果正确

1.4.2 输出 misclassification rate 曲线

运行前述代码, 由于 mnist 数据集有 70000 个并且限于实现代码的运行速度, 所以选取了 6000 张图片作为训练集,1000 张图片作为测试集, 分别计算 k 从 1 到 20 时, 在训练集以及测试集上的 misclassification rate, 并绘制曲线

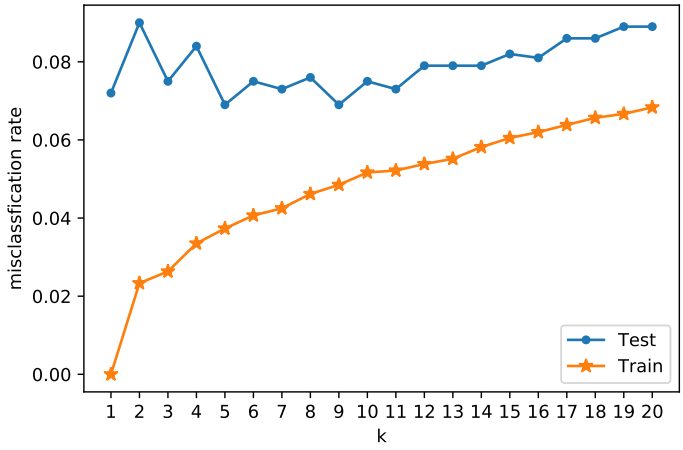


图 5: 绘制 misclassification rate 曲线

分析如下:

在训练集上的错误率随着 k 值的增大不断增大，由于 k 越大，考虑的点就越多，受到噪声影响就越大，导致错误率上升。而在测试集上的错误率上下波动，总体呈现先下降再上升的趋势，在 k 等于 5 以及 9 时达到最小值。综合来看当 k 等于 5 时兼顾了准确率以及泛化能力，在 1 到 20 中为最优解

1.5 实验总结

通过本次实验, 我对于 knn 算法有了更加深刻的理解, 尤其是对于 tier 情况的处理, 之前仅仅是有个大概的理解, 实际上实现起来还是会有很多细节上的问题, 其中最为严重的就是计算速度问题. 在和`sklearn`中的 KNN 实现比较之后, 发现我最开始的 knn 实现速度是真的很慢, 于是通过 pycharm 的分析功能尝试找出运行速度的瓶颈, 最终定位到了三个部分

- 距离计算

  由于最开始我是将每一个测试图片与训练集中的每一个向量计算距离, 导致速度非常慢, 而后利用了`numpy`的广播机制, 并且由于`numpy`的底层并不是使用 Python 而是 C 等效率更高的语言实现的, 所以很大的提高了运行速度

- 出现 tier 情况

  最初对于 tier 情况就是将 k 减小然后再一次调用整个过程, 而这就导致排序函数被重复调用, 之后改为了前文所述的使用`while`循环依次检查元素的方法, 对运行速度也有一定的提升

- 找出前 k 个最近的点

  最初是简单的将距离列表`dist_list`整体进行排序, 再取前 k 个元素, 但是后来注意到取出的元素个数 k 远小于排序集所包含元素个数, 如训练集有 6000 个但是 k 可能只有 1, 这就导致效率很低, 复杂度为 $O(NlogN)$. 而堆排序非常契合这种情况. 复杂度仅为 $O(klogN)$, 在使用堆排序之后又进一步提高了运行的效率

通过优化上述的三个瓶颈, 一定程度上提高了运行速度, 对于 6000 大小的训练集且 k 等于 5 时，计算大小为 1000 的测试集需要 20 秒，但是和 sklearn 还是有非常大的差距 (sklearn 的默认实现是 0.6 秒) , 后来也了解到可以通过使用`kd_tree`,`ball_tree`等特殊数据结构来优化运行速度, 但是限于能力和时间还是选择的最原始的方法.