实验 3 KNN 算法的实现与比较

1. 实验目的

通过本次实验，学习使用 KNN 算法进行空间向量的分类任务，比较分析不同 KNN 算法之间的复杂度。

1. 实验步骤

a) 阅读并理解 KNN 示例代码；

b) 实现一个质朴 KNN 算法；

c) 比较质朴算法和示例代码中的算法复杂度，并作分析。

1. 实验结果展示

3.1 代码实现与说明

**# 加载标准数据**

iris = load\_iris()

X = iris.data[:, :2]

y = iris.target

**# 设置近邻数为 15**

n\_neighbors = 15

**# 获取测试点（采样点）**

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

x\_create = np.arange(x\_min, x\_max, 0.02)

y\_create = np.arange(y\_min, y\_max, 0.02)

xx, yy = np.meshgrid(x\_create, y\_create)

array\_in\_col = np.c\_[xx.ravel(), yy.ravel()]

target\_labels = []

**# 计算测试点和所有标准点之间的欧式距离，**

**# 找到与此测试点距离最近的 15 个标准点**

**# 统计标准点中各种标签的数量，取最大者作为该测试点的标签**

for target in array\_in\_col:

**# 计算测试点与各个标准点之间的欧式距离**

dists = []

for x in X:

dist = np.linalg.norm(target - x)

dists.append(dist)

**# 找到与此测试点距离最近的 15 个标准点**

labels = []

for i in range(n\_neighbors):

index = dists.index(min(dists))

labels.append(y[index])

dists.remove(dists[index])

**# 统计计算获取测试点标签**

counters = [0, 0, 0]

for lab in labels:

counters[lab] += 1

target\_labels.append(counters.index(max(counters)))

Z = np.reshape(np.array(target\_labels), xx.shape)

**# 设置标准点和背景的颜色并绘图**

cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])

cmap\_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

plt.figure()

plt.pcolormesh(xx, yy, Z, cmap=cmap\_light)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap\_bold)

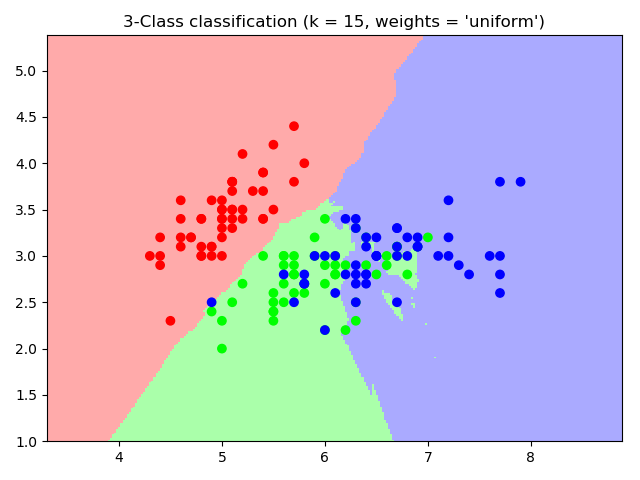
plt.xlim(xx.min(), xx.max())

plt.ylim(yy.min(), yy.max())

plt.title("3-Class classification (k = 15, weights = 'uniform')")

plt.show()

3.2 运行结果



3.3 复杂度分析

质朴的 KNN 算法耗时的地方在需要对每个测试点都计算一遍它与各个标准点之间的空间距离。

Sklearn 提供的 KNN 算法的优化表现在两个方面，一个是通过在源数据的基础上创建新的数据结构（树状等），来提前完成大量的需要重复计算的工作（例如在 fit 方法中使用 K-D Tree 、 Ball Tree 等），以此提高最后分类阶段的分类效率，在当前的小数据集中表现并不明显；另一个是更多地使用 C/C++ 、Cpython 来编写计算代码，避免 CPU 在 Python 解释器上花费过多的时间。

1. 实验总结与收获

通过这次实验，学习使用 KNN 算法进行空间向量的分类任务，并且比较分析两种 KNN 算法的复杂度。