KNN

课前准备

- 下载Anaconda软件，请点击这里进行下载。

本节要点

- KNN算法的基本原理。
- KNN算法实现分类与回归任务。
- 超参数调整。
- KD树的构建与邻居选择。【扩展】

KNN算法

引言

在讲解KNN之前，我们来看如下的数据集：

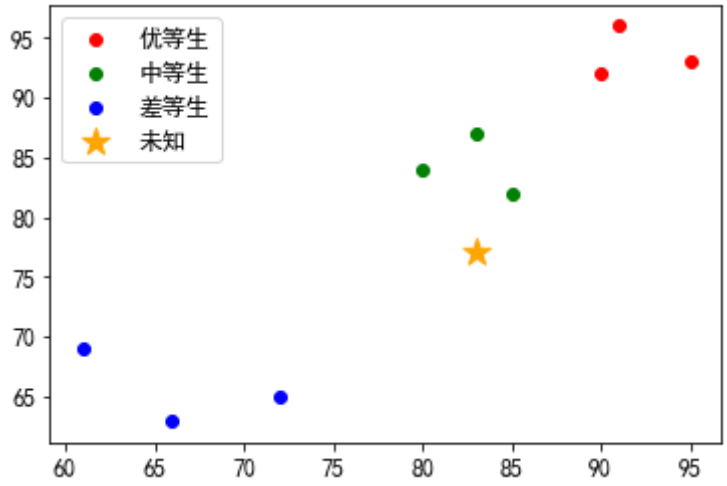
语文	数学	学生
95	93	好
90	92	好
91	96	好
85	82	中
83	87	中
80	84	中
61	69	差
66	63	差
72	65	差
83	77	?

我们将以上数据映射到空间中，进行绘制。因为数据具有两个特征，因此，每条数据对应二维空间中的一个点。

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["font.family"] = "SimHei"
plt.rcParams["axes.unicode_minus"] = False
plt.rcParams["font.size"] = 12

```

```
good = np.array([[95, 93], [90, 92], [91, 96]])
medium = np.array([[85, 82], [83, 87], [80, 84]])
bad = np.array([[61, 69], [66, 63], [72, 65]])
unknown = np.array([[83, 77]])

plt.scatter(good[:, 0], good[:, 1], color="r", label="优等生")
plt.scatter(medium[:, 0], medium[:, 1], color="g", label="中等生")
plt.scatter(bad[:, 0], bad[:, 1], color="b", label="差等生")
plt.scatter(unknown[:, 0], unknown[:, 1], color="orange", marker="*",
s=200, label="未知")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1554b1bec48>
```



课堂练习

上图中橙色的点（样本），最可能的类别是（ ）。

A 优等生

B 中等生

C 差等生

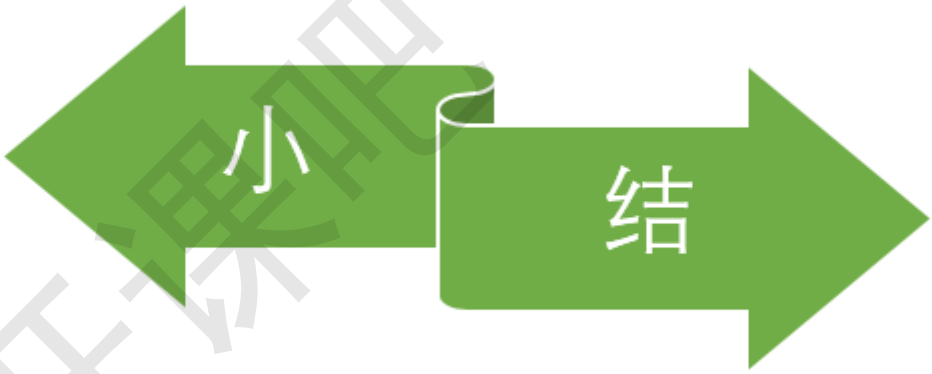
D 三种可能概率均等

算法原理

从上例的运行结果中，我们发现，相似度较高的样本，映射到 n 维空间后，其距离会比相似度较低的样本在距离上更加接近，这正是KNN算法的核心思维。

KNN（K-Nearest Neighbor），即K近邻算法。K近邻就是K个最近的邻居，当需要预测一个未知样本的时候，就由与该样本最接近的K个邻居来决定。KNN既可以用于分类问题，也可以用于回归问题。当进行分类预测时，使用K个邻居中，类别数量最多（或加权最多）者，作为预测结果。当进行回归预测时，使用K个邻居的均值（或加权均值），作为预测结果。

KNN算法的原理在于，样本映射到多维空间时，相似度较高的样本，其距离也会比较接近，反之，相似度较低的样本，其距离也会比较疏远。我们可以将该算法理解为“近朱者赤，近墨者黑”。



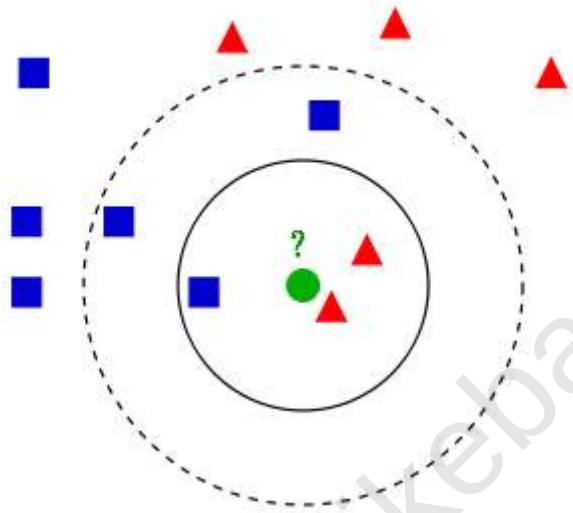
算法超参数

超参数，是指我们在训练模型之前，需要人为指定的参数。该参数不同于模型内部的参数，模型内部的参数是通过训练数据，在训练过程中计算得出的。超参数的不同，可能会对模型的效果产生很大影响。

K值

K值的选择，会直接影响到预测结果。当K值较小时，模型会依赖于附近的邻居样本，具有较好敏感性，但是稳定性会较弱，容易导致过拟合。当K值较大时，稳定性增加，但是敏感性会减弱，容易导致欠拟合。

通常情况下，我们可以通过交叉验证的方式，选择最合适的K值。



距离度量方式

在scikit-learn中，距离默认使用闵可夫斯基距离（minkowski），p的值为2。假设n维空间中的两个点为X与Y：

$$X = (x_1, x_2, \ldots\ldots, x_n)$$
$$Y = (y_1, y_2, \ldots\ldots, y_n)$$

则闵可夫斯基距离为：

$$D(X,Y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p}$$

可知：当p为1时，距离就是曼哈顿距离，当p为2时，距离就是欧几里得距离（欧氏距离）。

权重计算方式

权重可以分为两种：

- 统一权重：所有样本的权重相同。
- 距离加权权重：样本的权重与待预测样本的距离成反比。

课堂练习

使用KNN预测未知样本A，假设K的值为3，三个最近的邻居是B，C与D。A距离B，C，D的距离分别是2，3，5。如果按照距离加权的方式计算权重，则B，C，D的权重分别是（ ）。

A $\frac{1}{2}$，$\frac{1}{3}$，$\frac{1}{5}$
B $\frac{1}{3}$，$\frac{1}{3}$，$\frac{1}{3}$
C $\frac{1}{2}$，$\frac{3}{10}$，$\frac{1}{5}$
D $\frac{15}{31}$，$\frac{10}{31}$，$\frac{6}{31}$

算法步骤

KNN算法的执行过程如下：

1. 确定算法超参数。
    - 确定近邻的数量K。
    - 确定距离度量方式。
    - 确定权重计算方式。
    - 其他超参数。
2. 从训练集中选择离待预测样本A最近的K个样本。

3. 根据这K个样本预测A。
    - 对于分类，使用K个样本的类别（或加权类别）预测A。
    - 对于回归，使用K个样本目标值（y）的均值（或加权均值）预测A。

给定待预测样本A，如果在训练集中，存在两个（或更多）不同的邻居N_1与N_2（N_1与N_2的标签不同），二者距离A的距离相等，假设K的值为1，则此时选择哪个邻居较为合理?

A 选择N_1

B 选择N_2

C 随机选择

D 根据N_1与N_2在训练集中出现的先后顺序选择。

E C或D

程序示例

使用KNN实现分类

建模预测

我们以鸢尾花数据集为例，通过KNN算法实现分类任务。同样，为了方便可视化，我们只取其中的两个特征。

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

iris = load_iris()
X = iris.data[:, :2]
y = iris.target
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
# n_neighbors：邻居的数量。
# weights：权重计算方式。可选值为uniform与distance。
#     uniform：所有样本统一权重。
#     distance：样本权重与距离成反比。
knn = KNeighborsClassifier(n_neighbors=3, weights="uniform")
knn.fit(X_train, y_train)
y_hat = knn.predict(X_test)
print(classification_report(y_test, y_hat))
```

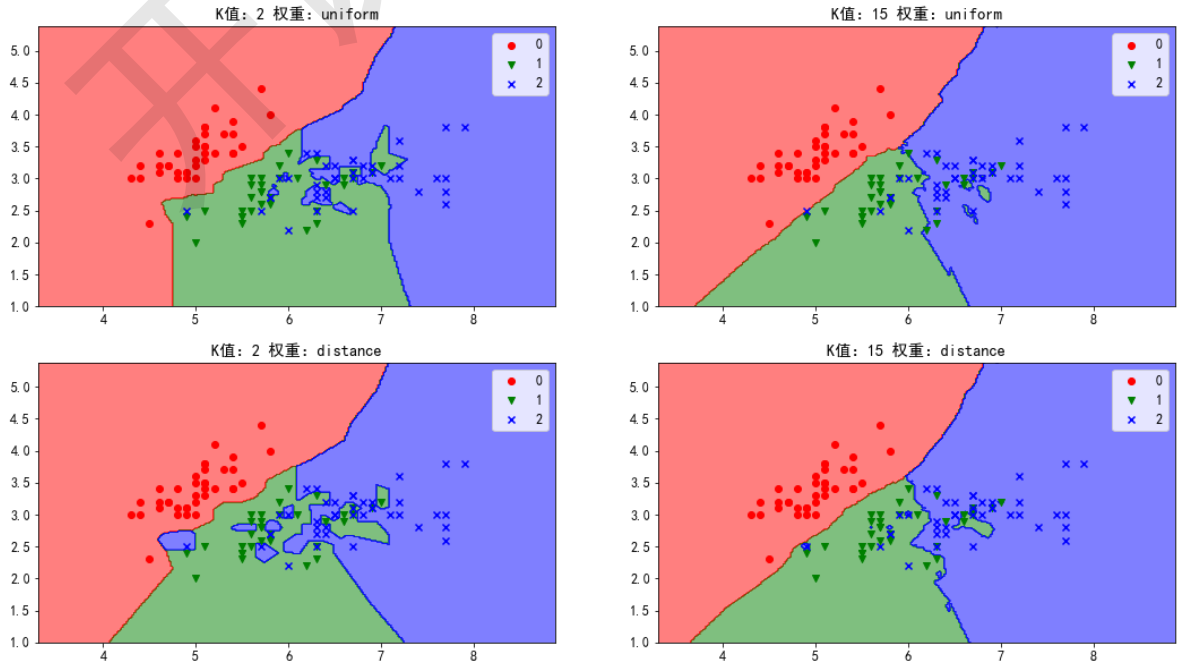
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        13
           1       0.78      0.44      0.56        16
           2       0.44      0.78      0.56         9

    accuracy                           0.71        38
   macro avg       0.74      0.74      0.71        38
weighted avg       0.77      0.71      0.71        38
```

超参数对模型的影响

当然，不同的超参数值，会直接影响模型的分类效果。我们可以从决策边界中发现这一点。

```
from matplotlib.colors import ListedColormap

def plot_decision_boundary(model, X, y):
    color = ["r", "g", "b"]
    marker = ["o", "v", "x"]
    class_label = np.unique(y)
    cmap = ListedColormap(color[: len(class_label)])
    x1_min, x2_min = np.min(X, axis=0)
    x1_max, x2_max = np.max(X, axis=0)
    x1 = np.arange(x1_min - 1, x1_max + 1, 0.02)
    x2 = np.arange(x2_min - 1, x2_max + 1, 0.02)
    X1, X2 = np.meshgrid(x1, x2)
    Z = model.predict(np.c_[X1.ravel(), X2.ravel()])
    Z = Z.reshape(X1.shape)
    plt.contourf(X1, X2, Z, cmap=cmap, alpha=0.5)
    for i, class_ in enumerate(class_label):
        plt.scatter(x=X[y == class_, 0], y=X[y == class_, 1],
                    c=cmap.colors[i], label=class_, marker=marker[i])
    plt.legend()
```

```python
from itertools import product
weights = ['uniform', 'distance']
ks = [2, 15]

plt.figure(figsize=(18, 10))
# 计算weights与ks的笛卡尔积组合。这样就可以使用单层循环取代嵌套循环，
# 增加代码可读性与可理解性。
for i, (w, k) in enumerate(product(weights, ks), start=1):
    plt.subplot(2, 2, i)
    plt.title(f"K值：{k} 权重：{w}")
    knn = KNeighborsClassifier(n_neighbors=k, weights=w)
    knn.fit(X, y)
    plot_decision_boundary(knn, X_train, y_train)
```



通过决策边界，我们可以得出如下结论：

- K的值越小，模型敏感度越强（稳定性越弱），模型也就越复杂，容易过拟合。
- K的值越大，模型敏感度越弱（稳定性越强），模型也就越简单，容易欠拟合。

超参数调整

在实际应用中，我们很难单凭直觉，就能够找出合适的超参数，通常，我们可以通过网格交叉验证的方式，找出效果最好的超参数。

```
from sklearn.model_selection import GridSearchCV

knn = KNeighborsClassifier()
# 定义需要尝试的超参数组合。
grid = {"n_neighbors": range(1, 11, 1), "weights": ['uniform', 'distance']}
# estimator：评估器，即对哪个模型调整超参数。
# param_grid：需要检验的超参数组合。从这些组合中，寻找效果最好的超参数组合。
# scoring：模型评估标准。
# n_jobs：并发数量。
# cv：交叉验证折数。
# verbose：输出冗余信息，值越大，输出的信息越多。
gs = GridSearchCV(estimator=knn, param_grid=grid, scoring="accuracy",
n_jobs=-1, cv=3, verbose=10)
gs.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:    0.8s
[Parallel(n_jobs=-1)]: Done   8 tasks      | elapsed:    0.8s
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    0.9s
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:    0.9s
[Parallel(n_jobs=-1)]: Batch computation too fast (0.1930s.) Setting
batch_size=2.
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:    0.9s
[Parallel(n_jobs=-1)]: Done  44 out of  60 | elapsed:    0.9s remaining:
0.3s
[Parallel(n_jobs=-1)]: Done  51 out of  60 | elapsed:    0.9s remaining:
0.1s
[Parallel(n_jobs=-1)]: Done  58 out of  60 | elapsed:    0.9s remaining:
0.0s
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed:    0.9s finished
```

```
GridSearchCV(cv=3, error_score=nan,
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None,
n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='deprecated', n_jobs=-1,
             param_grid={'n_neighbors': range(1, 11),
                         'weights': ['uniform', 'distance']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=10)
```

当网格交叉验证结束后，我们就可以通过GridSearchCV对象的相关属性，来获取最好的参数，分值与模型。

```
# 最好的分值。
print(gs.best_score_)
# 最好的超参数组合。
print(gs.best_params_)
# 使用最好的超参数训练好的模型。
print(gs.best_estimator_)
```

```
0.7773826458036984
{'n_neighbors': 9, 'weights': 'uniform'}
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=9, p=2,
                     weights='uniform')
```

最后，我们使用最好的模型在测试集上进行测试，实现最后的检验。

```
estimator = gs.best_estimator_
y_hat = estimator.predict(X_test)
print(classification_report(y_test, y_hat))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        13
           1       0.75      0.38      0.50        16
           2       0.41      0.78      0.54         9

    accuracy                           0.68        38
   macro avg       0.72      0.72      0.68        38
weighted avg       0.76      0.68      0.68        38
```

通过网格交叉验证，可以帮助我们完成调参工作，因此，即使我们不太了解超参数的含义，也可以顺利完成调参任务。这种说法正确吗?

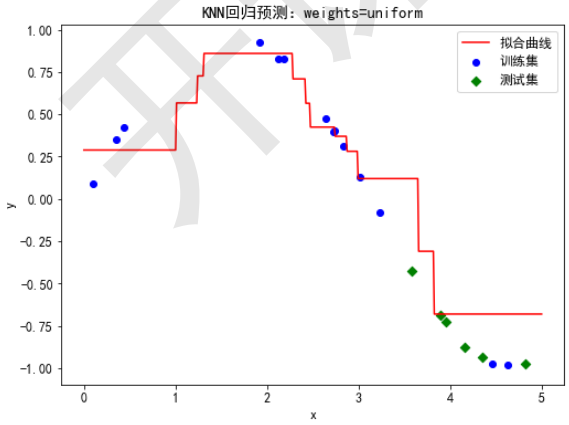
A 正确

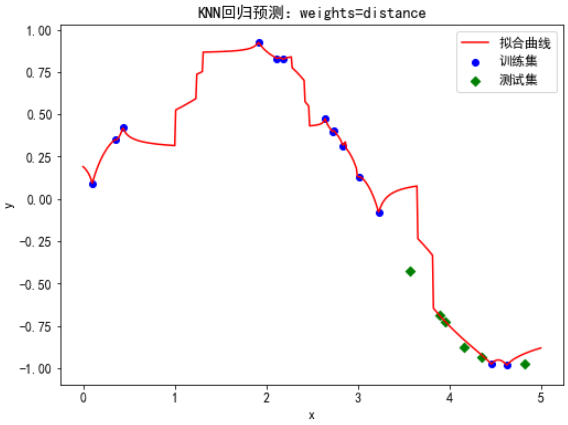
B 不正确

使用KNN回归预测

```
from sklearn.neighbors import KNeighborsRegressor

np.random.seed(0)
# 在[0, 5)的范围内，随机生成若干x点。
x = 5 * np.random.random(20)
X = x[:, np.newaxis]
y = np.sin(x) + np.random.normal(0, 0.01, size=len(x))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)
```

```
plt.figure(figsize=(18, 6))
for i, weights in enumerate(['uniform', 'distance']):
    plt.subplot(1, 2, i + 1)
    knn = KNeighborsRegressor(n_neighbors=3, weights=weights)
    knn.fit(X_train, y_train)
    plt.scatter(X_train, y_train, c="b", marker="o", label="训练集")
    plt.scatter(X_test, y_test, c="g", marker="D", label="测试集")
    t = np.linspace(0, 5, 500).reshape(-1, 1)
    plt.plot(t, knn.predict(t), c="r", label="拟合曲线")
    plt.legend()
    plt.title(f"KNN回归预测：weights={weights}")
    plt.xlabel("x")
    plt.ylabel("y")
```





KD树

当样本数量较少时，我们可以使用遍历所有样本的方式，找出最近的K的邻居，然而，如果数据集庞大，这种方式会造成大量的时间开销，此时，我们可以使用KD树的方式来选择K个邻居。
KD树算法中，首先是对训练数据进行建模，构建KD树，然后再根据建好的模型来获取邻近样本数据。

拓展点

- KD-Tree的构建。
- 使用KD-Tree寻找最近的邻居。

作业

1. 在数据集中，如果特征之间的量纲差异较大时，使用KNN算法之前，是否需要对数据进行标准化操作?
2. 在逻辑回归算法中，样本概率是通过sigmoid函数来计算的。KNN在分类时，是如何预测概率的。
3. 在个人信息数据集上，使用KNN算法建模，并预测一个人的收入是否会超过5万美元。