第 4 章　*k*-近邻算法

本章主要介绍 *k*-近邻算法。它是一个有监督的机器学习算法，也称为 KNN 算法，它可以解决分类问题，也可以解决回归问题。本章涵盖的主要内容如下：

- *k*-近邻算法的原理、优缺点及参数 *k* 取值对算法性能的影响；
- 使用 *k*-近邻算法处理分类问题的示例；
- 使用 *k*-近邻算法解决回归问题的示例；
- 使用 *k*-近邻算法进行糖尿病检测的实例；
- 基于统计学的特征选择；
- 扩展阅读之 *k*-近邻算法性能优化；
- 扩展阅读之卡方检测及 F 值检测。

4.1　算 法 原 理

k-近邻算法的核心思想是未标记样本的类别，由距离其最近的 *k* 个邻居投票来决定。

假设我们有一个已经标记的数据集，即已经知道数据集中每个样本所属的类别。此时有一个未标记的数据样本，我们的任务是预测出这个数据样本所属的类别。*k*-近邻算法的原理是，计算待标记的数据样本和数据集中每个样本的距离，取距离最近的 *k* 个样本。待标记的数据样本所属的类别，就由这 *k* 个距离最近的样本投票产生。

假设 X_test 为待标记的数据样本，X_train 为已标记的数据集，算法原理的伪代码如下：

- 遍历 X_train 中的所有样本，计算每个样本与 X_test 的距离，并把距离保存在 Distance 数组中。
- 对 Distance 数组进行排序，取距离最近的 *k* 个点，记为 X_knn。
- 在 X_knn 中统计每个类别的个数，即 class0 在 X_knn 中有几个样本，class1

在 X_knn 中有几个样本等。

❑ 待标记样本的类别，就是在 X_knn 中样本个数最多的那个类别。

4.1.1　算法的优缺点

k-近邻算法的优点是准确性高，对异常值和噪声有较高的容忍度；缺点是计算量较大，对内存的需求也较大。从算法原理中可以看出，每次对一个未标记样本进行分类时，都需要全部计算一遍待标记的数据样本和数据集中每个样本的距离。

4.1.2　算法的参数

k-近邻算法参数是 *k*，参数选择需要根据数据来决定。*k* 值越大，模型的偏差越大，对噪声数据越不敏感，当 *k* 值很大时，可能会造成模型欠拟合；*k* 值越小，模型的方差就会越大，如果 *k* 值太小，就会造成模型过拟合。

4.1.3　算法的变种

k-近邻算法有一些变种，其中之一就是可以增加邻居的权重。默认情况下，在计算距离时，都是使用相同的权重。实际上，可以针对不同的邻居指定不同的距离权重，距离越近则权重越高，这可以通过指定算法的 weights 参数来实现。

另外一个变种是，使用一定半径内的点取代距离最近的 *k* 个点。在 scikit-learn 中，RadiusNeighborsClassifier 类实现了这个算法的变种。当数据采样不均匀时，该算法变种可以取得更好的性能。

4.2　示例：使用 *k*-近邻算法进行分类

在 scikit-learn 中，使用 *k*-近邻算法进行分类处理的是 sklearn.neighbors.KNeighbors Classifier 类。

（1）生成已标记的数据集：

```
from sklearn.datasets import make_blobs
# 生成数据
centers = [[-2, 2], [2, 2], [0, 4]]
X, y = make_blobs(n_samples=60, centers=centers,
```

```
                    random_state=0, cluster_std=0.60)
```

我们使用 sklearn.datasets 包下的 make_blobs()函数来生成数据集，在上面的代码中，生成 60 个训练样本，这 60 个样本分布在 centers 参数指定的中心点周围。cluster_std 是标准差，用来指明生成的点分布的松散程度。生成的训练数据集放在变量 *X* 中，数据集的类别标记放在 *y* 中。

读者可以输出 *X* 和 *y* 的值看一下，一个更直观的方法是使用 Matplotlib 库，它可以很容易地把生成的点画出来。

```
# 画出数据
plt.figure(figsize=(16, 10), dpi=144)
c = np.array(centers)
plt.scatter(X[:, 0], X[:, 1], c=y, s=100, cmap='cool');  # 画出样本
# 画出中心点
plt.scatter(c[:, 0], c[:, 1], s=100, marker='^', c='orange');
```

这些点的分布情况在坐标轴上一目了然，其中，三角形的点即各个类别的中心点，如图 4-1 所示。

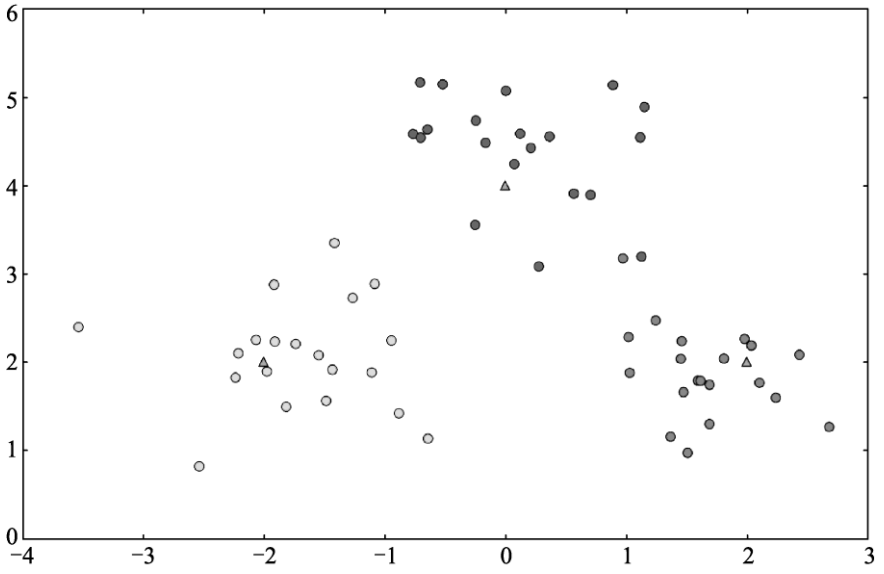

图 4-1　类别数据

（2）使用 KNeighborsClassifier 对算法进行训练，我们选择的参数是 k = 5。

```
from sklearn.neighbors import KNeighborsClassifier
# 模型训练
k = 5
clf = KNeighborsClassifier(n_neighbors=k)
clf.fit(X, y);
```

（3）对一个新的样本进行预测。

```
# 进行预测
X_sample = [0, 2]
X_sample = np.array(X_sample).reshape(1, -1)
y_sample = clf.predict(X_sample);
neighbors = clf.kneighbors(X_sample, return_distance=False);
```

我们要预测的样本是[0, 2]，使用 kneighbors()方法把这个样本周围距离最近的 5 个点取出来。取出来的点是训练样本 X 中的索引，从 0 开始计算。

（4）把待预测的样本及和其最近的 5 个点标记出来。

```
# 画出示意图
plt.figure(figsize=(16, 10))
plt.scatter(X[:, 0], X[:, 1], c=y, s=100, cmap='cool');   # 样本
plt.scatter(c[:, 0], c[:, 1], s=100, marker='^', c='k'); # 中心点
plt.scatter(X_sample[0][0], X_sample[0][1], marker="x",
           s=100, cmap='cool')       # 待预测的点

for i in neighbors[0]:
    plt.plot([X[i][0], X_sample[0][0]], [X[i][1], X_sample[0][1]],
          'k--', linewidth=0.6);     # 预测点与距离最近的 5 个样本的连线
```

从图 4-2 中可以清楚地看到 *k*-近邻算法的原理。

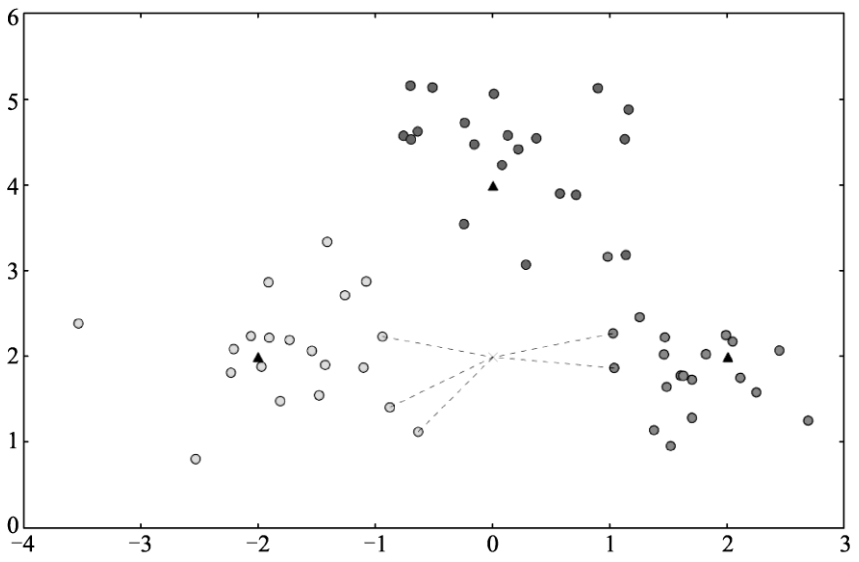

图 4-2　投票机制

本节的示例代码可参阅随书代码 ch04.01.ipynb。

4.3 示例：使用 *k*-近邻算法进行回归拟合

分类问题的预测值是离散的，可以使用 *k*-近邻算法在连续区间内对数值进行预测，然后进行回归拟合。在 scikit-learn 中，使用 *k*-近邻算法进行回归拟合的算法是 sklearn.neighbors. KNeighborsRegressor 类。

（1）生成数据集并在余弦曲线的基础上加入噪声。

```
import numpy as np
n_dots = 40
X = 5 * np.random.rand(n_dots, 1)
y = np.cos(X).ravel()

# 添加一些噪声
y += 0.2 * np.random.rand(n_dots) - 0.1
```

（2）使用 KNeighborsRegressor 训练模型。

```
# 训练模型
from sklearn.neighbors import KNeighborsRegressor
k = 5
knn = KNeighborsRegressor(k)
knn.fit(X, y);
```

我们要怎样进行回归拟合呢？

一个方法是，在 *X* 轴上的指定区间内生成足够多的点，针对这些足够密集的点，使用训练出来的模型进行预测，得到预测值 y_pred，然后在坐标轴上把所有的预测点连接起来，这样就画出了拟合曲线。

针对足够密集的点进行预测。

```
# 生成足够密集的点并进行预测
T = np.linspace(0, 5, 500)[:, np.newaxis]
y_pred = knn.predict(T)
knn.score(X, y)
```

可以用 score()方法计算拟合曲线针对训练样本的拟合准确性，笔者的计算机输出的结果如下：

```
0.99000494130215722                    # 在读者的环境运行时，值会略有差异
```

（3）把这些预测点连起来，构成拟合曲线。

```
# 画出拟合曲线
plt.figure(figsize=(16, 10), dpi=144)
plt.scatter(X, y, c='g', label='data', s=100)          # 画出训练样本
```

```
plt.plot(T, y_pred, c='k', label='prediction', lw=4)  # 画出拟合曲线
plt.axis('tight')
plt.title("KNeighborsRegressor (k = %i)" % k)
plt.show()
```

最终生成的拟合曲线及训练样本数据如图 4-3 所示。

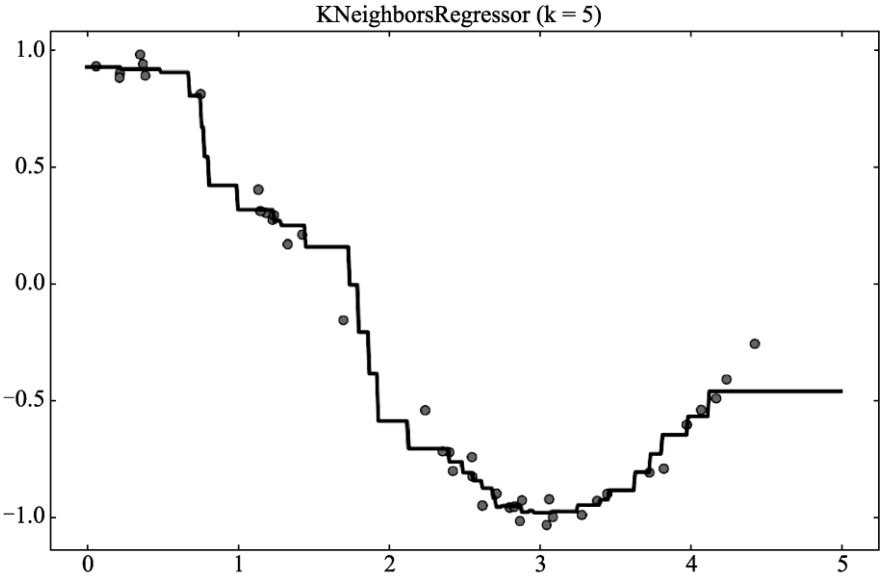

图 4-3　拟合曲线

本节的示例代码可以参阅 ch04.02.ipynb。

4.4　实例：糖尿病预测

本节使用 *k*-近邻算法及其变种，对 Pima 印第安人的糖尿病进行预测。数据来源于 kaggle.com，网址为 https://www.kaggle.com/uciml/pima-indians-diabetes-database，读者可自行前往下载，也可以使用随书代码中笔者下载好的数据 code/ dataset/pima-indians-diabetes。

4.4.1　加载数据

使用 pandas 加载数据：

```
# 加载数据
data = pd.read_csv('datasets/pima-indians-diabetes/diabetes.csv')
```

```
print('dataset shape {}'.format(data.shape))
data.head()
```

笔者的计算机输出的结果如下：

```
dataset shape (768, 9)
Out[2]:
Pregnancies Glucose BloodPressure   SkinThickness
Insulin BMI DiabetesPedigreeFunction    Age Outcome
0   6   148 72  35  0   33.6    0.627   50  1
1   1   85  66  29  0   26.6    0.351   31  0
2   8   183 64  0   0   23.3    0.672   32  1
3   1   89  66  23  94  28.1    0.167   21  0
4   0   137 40  35  168 43.1    2.288   33  1
```

从输出结果中可以看到，总共有 768 个样本、8 个特征，其中，Outcome 为标记值，0 表示没有糖尿病，1 表示有糖尿病。这 8 个特征分别如下：

- **Pregnancies**：怀孕的次数。
- **Glucose**：血浆葡萄糖浓度，采用 2 小时口服葡萄糖耐量试验测得。
- **BloodPressure**：舒张压（mmHg，即毫米汞柱）。
- **SkinThickness**：肱三头肌皮肤褶皱厚度（mm）。
- **Insulin**：两个小时血清胰岛素（μU/mL）。
- **BMI**：身体质量指数，即体重除以身高的平方。
- **DiabetesPedigreeFunction**：糖尿病血统指数，糖尿病和家庭遗传相关。
- **Age**：年龄。

我们可以进一步观察数据集里阳性和阴性样本的个数：

```
data.groupby("Outcome").size()
```

输出如下：

```
Outcome
0    500
1    268
dtype: int64
```

其中，阴性样本 500 例，阳性样本 268 例。接着需要对数据集进行简单处理，把 8 个特征值分离出来作为训练数据集，把 Outcome 列分离出来作为目标值，然后把数据集划分为训练数据集和测试数据集。

```
X = data.iloc[:, 0:8]
Y = data.iloc[:, 8]
print('shape of X {}; shape of Y {}'.format(X.shape, Y.shape))
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2);
```

4.4.2　模型比较

使用普通的 *k*-近邻算法、带权重的 *k*-近邻算法及指定半径的 *k*-近邻算法分别对数据集进行拟合并计算评分。

```
from sklearn.neighbors import KNeighborsClassifier,
    RadiusNeighborsClassifier

# 构造 3 个模型
models = []
models.append(("KNN", KNeighborsClassifier(n_neighbors=2)))
models.append(("KNN with weights", KNeighborsClassifier(
    n_neighbors=2, weights="distance")))
models.append(("Radius Neighbors", RadiusNeighborsClassifier(
    n_neighbors=2, radius=500.0)))

# 分别训练 3 个模型，并计算评分
results = []
for name, model in models:
    model.fit(X_train, Y_train)
    results.append((name, model.score(X_test, Y_test)))
for i in range(len(results)):
    print("name: {}; score: {}".format(results[i][0],results[i][1]))
```

笔者的计算机输出的结果如下：

```
name: KNN; score: 0.681818181818
name: KNN with weights; score: 0.636363636364
name: Radius Neighbors; score: 0.62987012987
```

对于权重算法，我们选择的是距离越近，权重就越高。RadiusNeighborsClassifier 模型的半径，选择了 500。从输出结果中可以看出，普通的 *k*-近邻算法性能是最好的。问题来了，这个判断准确吗？答案是不准确。因为我们的训练样本和测试样本是随机分配的，不同的训练样本和测试样本组合可能导致计算出来的算法准确性有差异。读者可以试着多次运行本书的示例代码，观察一下输出值是否有变化。

怎样更准确地对比算法的准确性呢？一个方法是，多次随机分配训练数据集和交叉验证数据集，然后求模型准确性评分的平均值。所幸，我们不需要从头实现这个过程，scikit-learn 提供了 KFold 和 cross_val_score()函数来处理这种问题。

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

results = []
```

```
for name, model in models:
    kfold = KFold(n_splits=10)
    cv_result = cross_val_score(model, X, Y, cv=kfold)
    results.append((name, cv_result))
for i in range(len(results)):
    print("name: {}; cross val score: {}".format(
        results[i][0],results[i][1].mean()))
```

在上述代码中，我们通过 KFold 把数据集分成 10 份，其中，1 份作为交叉验证数据集来计算模型的准确性，剩余的 9 份作为训练数据集。cross_val_score()函数总共计算出 10 次不同训练数据集和交叉验证数据集组合得到的模型准确性评分，最后求平均值。这样的评价结果相对更准确一些。

输出结果如下：

```
name: KNN; cross val score: 0.714764183185
name: KNN with weights; cross val score: 0.677050580998
name: Radius Neighbors; cross val score: 0.6497265892
```

看起来还是普通的 *k*-近邻算法性能更优。

4.4.3 模型训练与分析

本节我们就使用普通的 *k*-均值算法模型对数据集进行训练，并查看对训练样本的拟合情况及对测试样本的预测准确性情况。

```
knn = KNeighborsClassifier(n_neighbors=2)
knn.fit(X_train, Y_train)
train_score = knn.score(X_train, Y_train)
test_score = knn.score(X_test, Y_test)
print("train score: {}; test score: {}".format(train_score,
test_score))
```

笔者的计算机输出的结果如下：

```
train score: 0.842019543974; test score: 0.727272727273
```

从这个输出中可以看到两个问题：一是对训练样本的拟合情况不佳，评分 0.84 左右，这说明算法模型太简单了，无法很好地拟合训练样本；二是模型的准确性欠佳，不到 73%的预测准确性。我们可以进一步画出学习曲线，证实结论。

```
from sklearn.model_selection import ShuffleSplit
from common.utils import plot_learning_curve

knn = KNeighborsClassifier(n_neighbors=2)
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
plt.figure(figsize=(10, 6), dpi=200)
```

```
plot_learning_curve(plt, knn, "Learn Curve for KNN Diabetes",
                    X, Y, ylim=(0.0, 1.01), cv=cv);
```

笔者的计算机输出的结果如图 4-4 所示。

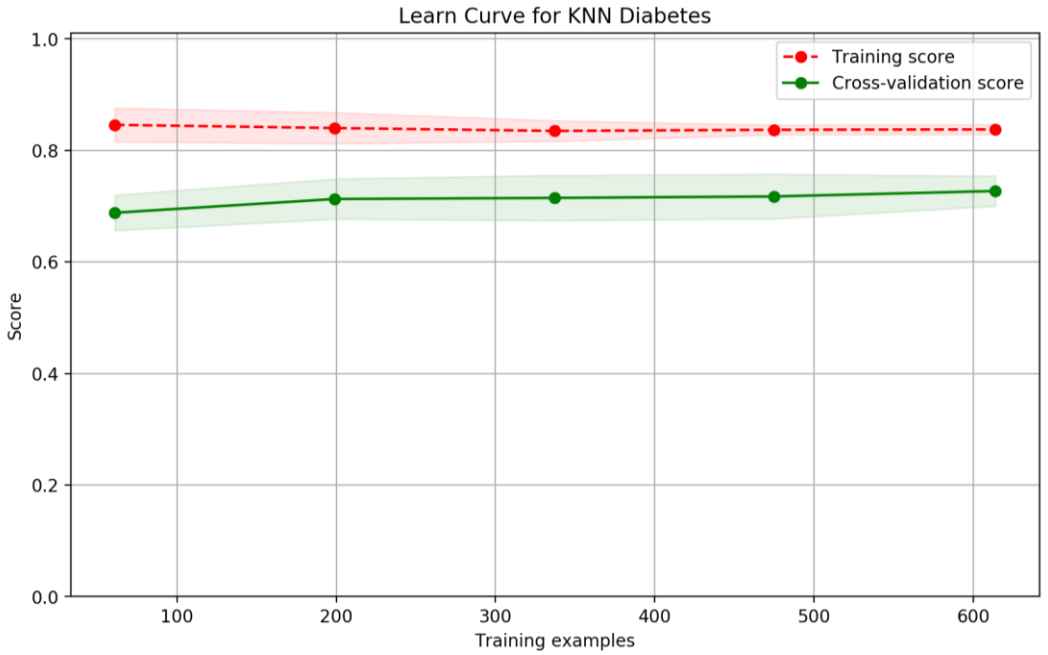

图 4-4　学习曲线

从图 4-4 中可以看出，训练样本评分较低且测试样本与训练样本距离较大，这是典型的欠拟合现象。*k*-近邻算法没有更好的方法来解决欠拟合问题，读者学完本书的其他章节后，可以试着用其他算法（如逻辑回归算法和支持向量机等）来对比不同模型的准确性情况。

4.4.4　特征选择与数据可视化

可能有读者会问，有没有直观的方法来揭示 *k*-近邻算法不是针对某个问题的好模型？一个办法是把数据画出来，可是有 8 个特征，无法在这么高的维度里画出数据并直观地观察。另一个办法是特征选择，即只选择 2 个**与输出值相关性最大**的特征，这样就可以在二维平面上画出输入特征值与输出值的关系了。

所幸，scikit-learn 在 sklearn.feature_selection 包中提供了丰富的特征选择方法。我们使用 SelectKBest 来选择相关性最大的两个特征：

```
from sklearn.feature_selection import SelectKBest
```

```
selector = SelectKBest(k=2)
X_new = selector.fit_transform(X, Y)
X_new[0:5]
```

把相关性最大的两个特征放在 X_new 变量中，同时输出了前 5 个数据样本。输出结果如下：

```
array([[ 148. ,   33.6],
       [  85. ,   26.6],
       [ 183. ,   23.3],
       [  89. ,   28.1],
       [ 137. ,   43.1]])
```

读者可能会好奇，相关性最大的特征到底是哪两个？对比本节开头的数据即可知道，它们分别是 BG（血糖浓度）和 BMI（身体质量指数）。血糖浓度和糖尿病的关系自不必说，身体质量指数是反映肥胖程度的指标，从业务角度来看，我们选择的 2 个相关性最高的特征还算合理。好学的读者可能想打破砂锅问到底：SelectKBest 到底使用什么神奇的方法选择出了这两个相关性最高的特征呢？这里涉及一些统计学的知识，感兴趣的读者可参阅下一节内容的延伸阅读。

我们来看看，如果只使用 2 个相关性最高的特征，那么在 3 种不同的 *k*-近邻算法中哪个准确性更高。

```
results = []
for name, model in models:
    kfold = KFold(n_splits=10)
    cv_result = cross_val_score(model, X_new, Y, cv=kfold)
    results.append((name, cv_result))
for i in range(len(results)):
    print("name: {}; cross val score: {}".format(
        results[i][0],results[i][1].mean()))
```

这次使用 X_new 作为输入，笔者的计算机输出的结果如下：

```
name: KNN; cross val score: 0.7252050581
name: KNN with weights; cross val score: 0.690037593985
name: Radius Neighbors; cross val score: 0.651025290499
```

可以看出，还是普通的 *k*-近邻模型准确性较高，其准确性将近 73%，与所有特征一起训练的准确性相近。这也从侧面证明了 SelectKBest 特征选择的准确性。

回到目标上来，我们是想看看为什么 *k*-近邻算法无法很好地拟合训练样本。现在只有 2 个特征，可以很方便地在二维坐标上画出所有的训练样本并观察这些数据的分布情况。

```
# 画出数据
plt.figure(figsize=(10, 6), dpi=200)
plt.ylabel("BMI")
```

```
    plt.xlabel("Glucose")
    # 画出 Y == 0 的阴性样本，用圆圈表示
    plt.scatter(X_new[Y==0][:, 0], X_new[Y==0][:, 1], c='r', s=20,
    marker='o')
    # 画出 Y == 1 的阳性样本，用三角形表示
    plt.scatter(X_new[Y==1][:, 0], X_new[Y==1][:, 1], c='g', s=20,
    marker='^');
```

横坐标是血糖值，纵坐标是 BMI 值，反映身体的肥胖情况。从图 4-5 中可以看出，在中间数据集密集的区域，阳性样本和阴性样本几乎重叠在一起了。假设现在有一个待预测的样本在中间密集区域，它的阳性邻居多还是阴性邻居多呢？这真的很难说。这样就可以直观地看到，*k*-近邻算法在糖尿病预测方面无法达到很高的预测准确性。

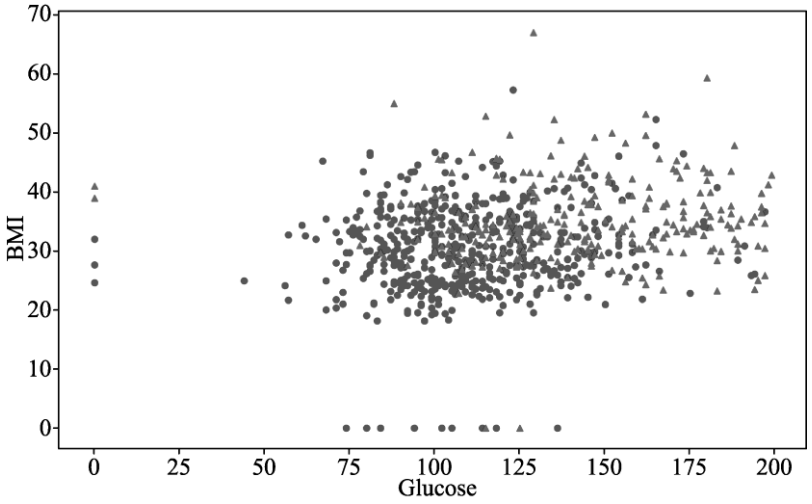

图 4-5　数据分布

4.5　拓 展 阅 读

本节首先介绍提高 *k*-近邻运算效率方面的知识，这里只给出一些通用的描述和参考资料，感兴趣的读者可以进一步深入研究。另外，在介绍特征选择时，本节还会介绍计算相关性大小的 SelectKBest()函数涉及的统计学知识。

4.5.1　如何提高 *k*-近邻算法的运算效率

根据算法原理，每次需要预测一个点时，都要计算训练数据集中每个点到这个

点的距离，然后选出距离最近的 k 个点进行投票。当数据集很大时，这个计算成本非常高。针对 N 个样本、D 个特征的数据集，其算法复杂度为 $O(DN^2)$。

为了解决这个问题，一种叫 *k-d* Tree 的数据结构被提出。为了避免每次都重新计算一遍距离，算法会把距离信息保存在一棵树中，这样在计算之前可以从树中查询距离信息，避免重新计算。*k-d* Tree 的基本原理是，如果 A 和 B 距离很远，B 和 C 距离很近，那么 A 和 C 的距离也很远。知道了这个信息，在合适的时候就可以跳过距离远的点，这样，优化后的算法复杂度可降低到 $O(DN\log(N))$。感兴趣的读者可参阅论文 *Multi Dimensional Binary Search Trees Used Associative Searching*，作者是 Bentley,J.L。

1989 年，另外一种称为 Ball Tree 的算法，在 *k-d* Tree 的基础上对性能进行了进一步优化。感兴趣的读者可以搜索 Five balltree construction algorithms 了解详细的算法信息。

4.5.2 相关性测试

先通过一个简单的例子了解一下**假设检验**问题，即判断假设的结论是否成立或成立的概率有多高。假设在一个城市随机采样到的程序员和性别的关系数据如下：

性别/职业	程序员	非程序员	小计
男	12	230	242
女	2	245	247
小计	14	475	489

假设我们的结论是程序员和性别无关，这个假设称为原假设（Null Hypothesis）。问：通过随机采样观测到的数据，原假设是否成立？或者说原假设成立的概率有多高？

卡方检验（Chi-Squared Test）是检测假设成立与否的一个常用工具。它的计算公式如下：

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{4-1}$$

其中，卡方检验的值记为 χ^2，O_i 是观测值，E_i 是期望值。针对本例，如果原假设成立，即程序员的职业和性别无关，那么我们期望的男程序员数量应该为(14 / 489) × 242 = 6.928，女程序员数量应该为(14 / 489) × 247 = 7.072，同理可得到的期望值如下：

性别/职业	程序员	非程序员
男	6.928	235.072
女	7.072	237.928

根据卡方检验公式，可以算出卡方值如下：

$$\chi^2 = \frac{(12-6.928)^2}{6.928} + \frac{(230-235.072)^2}{235.072} + \frac{(2-7.072)^2}{7.072} + \frac{(245-237.928)^2}{237.928} = 7.67$$

算出卡方值后，怎么判断原假设成立的概率是多少呢？这里还涉及**自由度**和**卡方分布**的概念。简单地讲，自由度是$(r-1)\times(c-1)$，其中，r 是行数，c 是列数，针对我们的问题，其自由度为 1。卡方分布是指，若 n 个相互独立的随机变量均服从正态分布，则这 n 个随机变量的**平方和**构成一新的随机变量，其分布规律称为卡方分布。卡方分布的密度函数和自由度相关，知道了自由度和目标概率，就能求出卡方值。

针对我们的问题可以查表得到，自由度为 1 的卡方分布在 99%处的卡方值为 6.63。我们计算出来的卡方值为 7.670。由于 7.67 > 6.63，因此有 99%的把握可以推翻原假设。换个说法，如果原假设成立，即程序员的职业和性别无关，那么我们随机采样到的数据出现的概率将低于 1%。读者可以搜索“卡方表”或 Chi Squared Table，找到不同自由度对应的卡方值。

卡方值的大小可以反映变量与目标值的相关性，值越大，相关性越大。利用这个特性，通过 SelectKBest()函数就可以计算不同特征的卡方值，以此判断特征与输出值的相关性大小，从而完成特征选择。在 scikit-learn 中，计算卡方值的函数是 sklearn.feature_selection.chi2()。除了卡方检验外，F 值检验等算法也可以用来评估特征与目标值的相关性。SelectKBest 默认使用的就是 F 值检验算法，在 scikit-learn 中，使用 sklearn.feature_selection.f_classif 来计算 F 值。关于 F 值，感兴趣的读者可以在英文版维基百科上搜索 Fisher’s exact test，了解更多的信息。

4.6　习　　题

1．请用一句话描述 *k*-近邻算法的原理。

2．*k*-近邻算法有哪些变种？

3．参考 ch04.01.ipynb，使用 RadiusNeighborsClassifier 类来处理分类问题。

4．参考 ch04.02.ipynb，使用不同的算法参数 *k*，观察针对同一个数据集，拟合曲线有什么变化。

5．针对 ch04.02.ipynb 中的回归问题，试着画出算法的学习曲线。提示：关于学习曲线可参考第 3 章的 ch03.02.ipynb 例子，重点是复用 plot_learning_curve()函数。

6．运行 ch04.03.ipynb 的代码，验证一下，如果使用 SelectKBest 选择出 4 个相关性最高的特征，并把这 4 个特征作为输入来训练模型，那么模型的准确性是否会提高？为什么？

7．运行 ch04.03.ipynb 的代码看一下，使用 SelectKBest 选择特征时，如果把默认的 F 值换成卡方值则结果有什么不同。