第 3 章

数据预处理

现实世界的应用中，通过网络爬虫、调查问卷等途径收集到的原始数据，往往存在缺失值、重复记录、异常点及数据单位不一致性等一系列问题。为了将这些未经处理的原始数据转变为适合深入分析和挖掘的形式，需要对数据进行预处理，以确保数据的质量和一致性，从而提高后续分析的准确性和可靠性。

本章将详细介绍数据预处理的相关技术和方法，并基于心脑血管数据进行数据预处理的综合案例演示。

本章要点：

- 数据清洗。
- 数据转换。
- 数据规约。

3.1 数据预处理概述

数据预处理是进行数据分析与挖掘之前必不可少的准备阶段，旨在通过集成、清理和转换等操作提升原始数据的质量、完整性和适用性。这一过程涉及识别并修正错误、填补或处理缺失值与异常值、实施数据转换及归一化、执行合并与去重、降维及特征提取等多项任务，确保数据集更适合后续算法的需求。

具体而言，数据预处理通常包括数据集成、数据清洗、数据转换、数据脱敏与隐私保护、数据变换和数据规约等一系列操作，旨在将原始数据转化为适合进行数据分析和挖掘的格式。①数据集成是将来自不同来源的数据合并成一个统一的数据集，以便进行综合分析。②数据清洗包括填补缺失值、消除重复记录、处理异常点等，以确保数据的完整性和准确性。③数据转换包括对数据进行标准化、归一化和转换等操作，以提高数据的可解释性和模型的性能。④数据脱敏与隐私保护是指在保留数据一定有用性的同时，通过移除或修改个人信息，确保在数据分析和使用过程中个人隐私得到妥善保护的一系列措施。⑤数据变换通常涉及将数据从原始的线性空间或域转换到另一个线性空间或域，以便更好地进行分析或提取特征。⑥数据规约则是通过降维、数据压缩和特征选择等方法减少数据的规模，减少数据集的复杂度和大小，从而提高数据挖掘的效率。

这些预处理步骤有助于提高数据的质量和可用性，为后续的数据分析和挖掘任务打下坚实的基础。

3.2 数据集成

数据集成通过合并来自不同来源的数据，构建出统一的数据集合，以便于后续的数据分析和利用。在这个过程中，确保数据的一致性、完整性和准确性是至关重要的，这对于保证数据分析结果的可靠性和有效性具有决定性作用。

在本章中，默认所有实例均已导入 Pandas 和 NumPy 模块包，导入语句如下：

```
import numpy as np
import pandas as pd
```

3.2.1 数据集成概述

在现实应用场景中，数据常分散在多个不同的数据源中。在进行数据分析与挖掘之前，需考虑各数据源的字段表述是否统一、属性是否存在重复等问题。数据集成的作用在于将来自不同源头、不同格式或结构的数据融合，形成统一且一致的数据集合，以支持后续的分析工作。数据集成的目标是简化后续的数据分析和处理流程，更有效地满足业务需求。在众多大数据项目中，数据集成占据了相当大的工作量，虽然具体比例可能因项目而异，但普遍认为数据集成是项目中最为耗时和关键的环节之一。

以下是 3 种常见的数据集成方式。

（1）手动导入/导出：这是最基本的数据集成方法，通过手动将数据从一个系统导出，然后手动导入另一个系统。虽然简单易行，但效率低下且容易出错，适用于小规模数据或临时需求。

（2）批量导入/导出：这种方法是通过定期批量导入或导出数据，通常使用文件格式（如 CSV、Excel 等）。可以使用 ETL（Extract-Transform-Load）工具或脚本来自动化这个过程，适用于中小规模的数据集成需求。

（3）数据库连接：将多个应用程序的数据库连接到同一个中央数据库，以实现数据的集中存储和共享。这种方法可以通过数据库连接器、API 或数据库复制等技术实现。

3.2.2 数据集成的主要方法

在实践中运用 Python 进行数据集成时，通常会涉及数据的合并与连接。Pandas 库内置了丰富的函数或方法来实现数据的合并与连接，主要是 merge、concat 及 join 等。这些函数和方法能够针对 Series 对象和 DataFrame 对象，执行各种符合逻辑关系的合并或连接。其中，merge 方法主要基于两个 DataFrame 的共同列进行合并，join 方法主要基于两个 DataFrame 的索引进行合并，concat 方法是对 Series 或 DataFrame 进行行连接或列连接。表 3-1 所示为 Pandas 中用于数据集成的主要函数或方法。

表 3-1 Pandas 中用于数据集成的主要函数或方法

函数/方法	说　明
merge()	Pandas 中进行数据集合并的主要函数，用于将两个或多个 DataFrame 按照某些共同的键（如列名）合并在一起。merge()函数可以实现内连接（inner）、外连接（outer）、左连接（left）或右连接（right）。它允许根据指定的键来组合数据，类似于关系数据库中的连接操作
concat()	主要用于将多个 DataFrame 或 Series 沿指定的轴（行轴 axis=0 或列轴 axis=1）连接起来。默认保留每个对象的索引和列标签，支持不同的索引对齐方式

续表

函数/方法	说　明
join()	针对 DataFrame 或 Series 类型数据的连接方法，可以将多个 DataFrame 或 Series 根据索引对齐并连接起来
combine_first()	是一个实例方法，用于填充缺失值。它会用另一个 DataFrame 或 Series 中的非空值替换当前对象的缺失值

表 3-1 中的每种方法都有特定的使用场景。merge 和 concat 是最常用的两个合并函数，但它们在某些细节上有所不同，选择哪种方法取决于具体的数据合并需求。例如，当需要根据共同的列合并数据时，通常使用 merge；而当需要简单地将数据附加在一起时，则使用 concat 更合适。在处理大数据集时，需要注意 merge 中的 sort 参数，如果不需要排序，可以设置为 False 以提高性能。

下面将结合实例介绍 Python 中通过数据合并和连接进行数据集成的方法。

1. 运用 merge 函数合并数据

merge 函数是 Pandas 库中的一个重要数据合并工具，其作用类似于 Excel 中的 VLOOKUP 函数或 SQL 中的 JOIN 操作。merge 函数主要用于将两个或多个 DataFrame 对象基于某些共同的键进行横向连接，从而生成一个新的 DataFrame。

在 Pandas 中，merge 函数支持多种连接方式，包括但不限于以下 4 种。内连接(inner)：只保留两个 DataFrame 都有的键的记录。左连接(left)：保留左边 DataFrame 的所有记录，右边 DataFrame 中不存在的键则填充 NaN。右连接(right)：保留右边 DataFrame 的所有记录，左边 DataFrame 中不存在的键则填充 NaN。外连接(outer)：保留所有记录，对于任一边 DataFrame 中没有匹配的键，则填充 NaN。默认情况下，如果不指定连接方式，则默认为内连接。merge 函数原型如下：

```
pandas.merge(left, right, how = 'inner', on = None, left_on = None, right_on = None, left_index = 
False, right_index = False, sort = True, suffixes = ('_x', '_y'), copy = True, indicator = False, 
validate = None)
```

merge 函数的参数及作用如下。

left：第一个要合并的 DataFrame。

right：第二个要合并的 DataFrame。

how：指定如何进行合并。可以是以下几种——left、right、outer、inner。

on：指定用于合并的键。如果设置为 None，则使用默认键。

left_on，right_on：分别指定左、右 DataFrame 的合并键。

left_index，right_index：如果设置为 True，则使用索引作为合并键。

sort：如果设置为 True，则根据合并键进行排序。

suffixes：当两个 DataFrame 合并时如果存在重复键，则使用这个参数来区分重复的列。默认是('_x', '_y')，分别对应左、右 DataFrame 的列。

copy：如果设置为 True，则在合并时复制数据。如果设置为 False，则可能修改原始数据。

indicator：如果设置为 True，则在结果中添加一个指示列，以表明哪些行来自哪个 DataFrame。

validate：用于验证合并操作是否符合预期关系，若不符合设定的关系且设置了该参数，将抛出异常。

例 3-1 merge 函数合并数据实例。

```
In:
    d1 = pd.DataFrame({'cat':['a','b','c','d'],'color':['red','yellow','blue','black']})
    d2 = pd.DataFrame({'cat':['a','b','c','d'],'age':[2,4,6,8]})
    print(pd.merge(d1, d2))
Out:
      cat   color  age
    0  a      red    2
    1  b   yellow    4
    2  c     blue    6
    3  d    black    8
```

例 3-1 使用 merge()函数将两个 DataFrame 对象进行合并。例子中的两个 DataFrame 都有 cat 列,其中,d1 还有 color 列,d2 有 age 列。pd.merge(d1,d2)函数将 d1 和 d2 两个 DataFrame 对象沿着 cat 列进行合并。如果两个 DataFrame 中 cat 列的值相同,则对应的 color 和 age 列的值会组合到一起,形成一个新的 DataFrame。

2. 用 concat 函数连接数据

在 Pandas 中,concat()函数用于灵活拼接数据。默认行为类似于"全外连接",保留所有输入 DataFrame 的行。若只想保留两个 DataFrame 共有的行,可将 join 参数设置为 inner 以实现内连接。内连接只保留共有的行,而外连接(默认或设置为 outer)则包含所有行,未匹配的行以 NaN 填充。concat()可根据需求调整参数进行数据拼接,而更复杂的连接操作(如左连接或右连接)则通常使用 merge()函数。

concat 函数原型如下:

```
pd.concat(objs, axis = 0, join = 'outer', ignore_index = False, keys = None, levels = None, names =
None, verify_integrity = False, sort = False, copy = True)
```

concat 函数的参数及作用如下。

objs:一个或多个 Pandas 对象,可以是 Series 或 DataFrame。

axis:指定连接的轴,0 表示按行连接,1 表示按列连接。

join:连接方式,outer 表示全外连接,会保留所有原始数据;inner 表示内连接,只保留共有键的数据;如果为 None,则使用全外连接。

ignore_index:如果为 True,则忽略现有的索引,并为合并后的对象创建新的整数索引。

keys:为输入数据添加标签,生成多层索引(如 keys=['A','B'])。

levels:配合 keys,用于指定多层索引的级别。

names:用于指定连接后的列名或索引名。

verify_integrity:如果为 True,则检查合并后的索引是否唯一,如果有重复,则抛出异常。

sort:如果为 True,根据非拼接轴的列名对结果进行排序,默认为 False(不排序)。

copy:如果为 False,则在可能的情况下不复制数据。

例 3-2 concat 函数沿行方向使用外连接数据实例。

```
In:
    # 创建第一个 DataFrame
    df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                        'B': ['B0', 'B1', 'B2'],
                        'C': ['C0', 'C1', 'C2']},
```

```
                        index = [0, 1, 2])
    # 创建第二个 DataFrame
    df2 = pd.DataFrame({'A': ['A3', 'A4', 'A5'],
                         'B': ['B3', 'B4', 'B5'],
                         'D': ['D3', 'D4', 'D5']},
                        index = [3, 4, 5])
    # 使用 concat()沿行方向执行全外连接数据
    result = pd.concat([df1, df2], join = 'outer',axis = 0)
    # 打印合并后的 DataFrame
    print(result)
Out:
        A   B    C    D
    0  A0  B0   C0  NaN
    1  A1  B1   C1  NaN
    2  A2  B2   C2  NaN
    3  A3  B3  NaN   D3
    4  A4  B4  NaN   D4
    5  A5  B5  NaN   D5
```

例 3-2 中使用 pd.concat()函数将两个 DataFrame 进行合并。join='outer'表示进行全外连接,axis=0 表示沿行方向进行合并。合并后的 DataFrame 将包含 df1 和 df2 的所有行,即使某些行在一个 DataFrame 中没有匹配。未匹配的行将被添加到合并后的 DataFrame 中,并填充为缺失值(NaN)。

3. 运用 combine_first()函数合并数据

使用 combine_first()函数合并两个 DataFrame 时,它取两个 DataFrame 中对应索引的部分,将它们组合在一起,对于不重叠的部分,会填充 NaN 值。

例 3-3 combine_first()函数合并数据实例。

```
In:
    # 创建两个 DataFrame
    df1 = pd.DataFrame({
        'A': [1, 2, None, 4],
        'B': [5, None, 7, 8]
    })
    df2 = pd.DataFrame({
        'A': [None, None, 3, 4],
        'B': [5, 6, 7, None]
    })
    # 使用 combine_first()合并 DataFrame
    # 结果 DataFrame 会优先使用 df1 的数据,然后用 df2 的数据填充 df1 中的 NaN
    result = df1.combine_first(df2)
Out:
         A    B
    0  1.0  5.0
    1  2.0  6.0
    2  3.0  7.0
    3  4.0  8.0
```

例 3-3 中两个 DataFrame 对象 df1 和 df2 中都有 A 和 B 列。使用 combine_first()方法时,结果 result 保留了 df1 中的所有非 NaN 值,并用 df2 中的相应值填充了 df1 中的 NaN。例如,df1 中 A 列的第三个值和 B 列的第二个值是 NaN,这些值在 result 中被 df2 中相应的值 3 和 6 所替换。

3.2.3 数据集成的关键问题

在数据集成过程中，实体识别、冗余识别及数据值冲突检测是3个核心问题，需要得到有效解决。实体识别的主要目标是识别并确定数据源中的各种实体，如个人、地点、组织等。不同数据源可能对同一实体有不同的描述方式，如人名的不同翻译版本，实体识别的任务就是识别这些等价实体，并将它们统一整合。冗余识别是减少数据集成过程中冗余数据的关键步骤。不同数据源可能包含对同一实体的重复记录，这些冗余不仅浪费存储空间，还可能降低数据分析的准确性。因此，识别并去除这些重复记录是很有必要的。数据值冲突检测是确保集成数据一致性和准确性的重要环节。由于数据源的质量参差不齐，某些数据可能已经过时或不准确，检测数据值冲突就是识别并解决这些数据源之间的差异，确保数据的正确性和最新性。

通过解决实体识别、冗余识别和数据值冲突检测等关键问题，可以提高数据集成的效率和准确性，从而更好地支持数据分析、业务决策和其他应用场景的需求。

1. 实体识别

在数据集成过程中，实体识别问题是一个关键问题，它涉及如何准确识别不同数据源中的实体并进行匹配。例如，对于具有不同命名方式但实际表示同一实体的属性，如一个数据库中的 customer_id 与另一个数据库中的 customer_number，数据集成需要确定它们是否代表相同的实体。实体识别中的单位不统一也会带来问题，例如，重量属性在一个系统中采用公制，而在另一个系统中却采用英制；价格属性在不同地点采用不同的货币单位。这些语义的差异为数据集成带来许多问题。

实体识别的主要目的是解决不同数据源之间的矛盾和不一致性，如同名异义、异名同义、单位不统一等问题。同名异义是指同一属性对应着不同的实体。例如，两个数据源中的属性 id 分别描述的是商品编号和订单编号，这两个属性实际上代表了不同的实体。实体识别技术可以帮助用户识别这种情况，并将具有相同名称但不同含义的属性进行区分。异名同义是指不同属性对应着同一实体。例如，两个数据源中的属性 sale_dt 和 sale_date 描述的都是销售日期，只是属性名称不同。实体识别技术可以通过比对不同数据源中的属性，发现并解决这种问题。单位不统一是指同一个实体分别用不同标准的容积单位表示。例如，两个数据源中的属性 fuel_consumption 分别描述的是以升和加仑为容积单位的燃料消耗量。实体识别技术有助于发现并解决这种单位不一致的问题。

在大数据环境下，实体识别更加复杂，因为数据源更加多样化，数据量也更加庞大。因此，大数据环境下的实体识别需要采用更加高级的技术和方法。一种常见的方法是利用元数据进行实体匹配。这种方法通过分析不同数据源中的元数据，找出相同或相似的实体，然后进行匹配和整合。这种方法需要先对元数据进行清洗和整理，以确保匹配的准确性和可靠性。另一种常见的方法是利用自然语言处理技术进行实体识别。这种方法通过分析文本中的语言结构和语义关系，识别出其中的实体信息。常用的技术包括词向量、语义分析和机器学习等。在大数据环境下，实体识别还需要考虑如何处理大量的数据和如何优化算法以提高效率。因此，需要采用分布式计算和并行处理等技术，同时还需要对算法进行优化和改进，以适应大数据环境下的需求。

例 3-4 实体识别实例，解决不同数据源中异名同义问题。

```
In:
    # 读取数据
    df1 = pd.read_csv('data/data1.csv')
```

```
    df2 = pd.read_csv('data/data2.csv')
    # 定义映射关系
    map_dict = {
        'customer_no': 'customer_id'
    }
    #对第二个数据集进行映射,解决不同数据源中异名同义的问题
    df2 = df2.rename(columns = map_dict)
    # 合并数据集
    df_merged = pd.merge(df1, df2, on = 'customer_id')
    # 打印结果
    print(df_merged)
Out:
            customer_id  customer_name  customer_age
    0            1            John            25
    1            2            Tom             33
```

例 3-4 中,使用 rename 函数将第二个数据集中的 customer_no 字段映射为与第一个数据集中同名的 customer_id,然后使用 pd.merge 函数将两个数据集连接在一起,连接字段是 customer_id。

实体识别是数据集成中非常重要的一步,它有助于将不同数据源的实体进行统一和整合,提高数据的质量和可用性,为后续的数据分析和知识图谱构建等任务提供基础支持。

2. 冗余识别

数据冗余也是数据集成中需要特别关注的问题。如果一个属性能由另一个或另一组属性推导出,则这个属性可能是冗余的,例如,一个顾客数据表中的平均月收入就是冗余属性,因为它可以根据月收入属性计算出来。另外,属性命名的不一致也会导致集成后的数据集中出现数据冗余问题。数据冗余可能会导致数据集的体积增大,同时也可能会引入错误。例如,同一属性在不同的数据库中会有不同的字段名,如“年薪”和“月薪”可能就是描述同一种信息的两种不同字段。在集成多个数据库时,这种冗余数据是需要被检测和处理的。相关分析是一种可以用来检测数据冗余问题的方法。例如,给定两个属性 A 和 B,根据这两个属性的数值可分析出这两个属性间的相互关系。如果一个属性的变化总是伴随着另一个属性的变化,那么这两个属性可能存在冗余。对于数值属性,可以使用相关系数(Correlation Coefficient)和协方差(Covariance)来评估一个属性的值如何随另一个属性变化。

除了检查属性的冗余,有时还需要检查元组的冗余情况。例如,对于某个唯一的数据实体,有可能存在来自不同数据源的重复元组,此时需要对元组的冗余情况进行检测和处理。

综上,在进行数据集成时,可以利用相关分析等方法来检测并处理可能出现的数据冗余问题,以提高数据集成后的数据质量。

例 3-5　冗余元组识别实例。

```
In:
    # 读取两个 CSV 文件
    df1 = pd.read_csv('data/data1.csv')
    df2 = pd.read_csv('data/data3.csv')
    # 将两个 DataFrame 进行连接,并去除重复的客户信息
    merged_df = pd.merge(df1, df2, on = 'customer_id', how = 'outer')
    merged_df = merged_df.drop_duplicates()
    # 将连接后的 DataFrame 存储为新的 XLSX 文件
    merged_df.to_excel('data/1_BasicInfo_new.xlsx')
```

图 3-1 所示为冗余元组识别实例结果。

	customer_id	customer_name	customer_email
0	1	John	John@htomail.com
1	2	Tom	Tom@gmail.com
2	3	Jack	NaN
3	6	NaN	Lucky@hotmail.com

图 3-1 冗余元组识别实例结果

在例 3-5 中,首先通过 customer_id 字段将两个 DataFrame 合并。随后,应用 drop_duplicates()函数移除了重复的客户信息。合并时采用了外连接(outer join)方式,确保了两个数据源的所有记录都被保留,包括重复项。在删除重复项时,仅保留了首次出现的记录。若数据源中存在重复的客户信息,还可依据其他字段如 customer_name 和 customer_email 进行匹配和处理。若这些字段的值完全相同,则判定为重复记录。最终,合并后的 DataFrame 被保存为新的 XLSX 文件。

3. 数据值冲突问题

在数据集成过程中,由于不同数据源可能存在表示方法、比例尺度或编码方式的差异,因此,数据值可能会出现冲突,这种情况在实际应用中比较常见。例如,在一个数据源中,重量属性可能采用公制单位,如克或千克,而在另一个数据源中,相同的属性却可能采用英制单位,如磅或盎司。另外,价格属性在不同地点可能采用不同的货币单位,如美元、欧元或人民币等。

这些来自不同数据源的语义或数据结构方面的差异为数据集成带来了许多问题。首先,这些差异可能导致数据集成过程中的数据不匹配问题,使得合并后的数据集准确性降低。其次,这些差异可能会对数据分析和决策产生影响,从而降低数据分析结果的准确性和可靠性。此外,这些差异还可能增加数据清洗和转换的复杂性,浪费大量的人力和时间资源。

因此,在进行数据集成时,需要关注和解决这些潜在的差异问题。这可能包括对数据进行标准化、统一编码方式、转换货币单位等操作,以确保集成后的数据集的准确性和一致性。同时,还需要采用合适的数据集成策略,如数据映射、数据转换和数据融合等,以尽可能减小这些差异对数据集成的负面影响。

例 3-6 数据值冲突检测与处理实例。

```
In:
    # 创建第一个 DataFrame,价格单位为人民币
    data1 = {
        '商品': ['商品 A', '商品 B'],
        '价格(元)': [100, 200]
    }
    df1 = pd.DataFrame(data1)
    # 创建第二个 DataFrame,价格单位为美元
    data2 = {
        '商品': ['商品 C', '商品 D', '商品 E'],
        '价格(美元)': [10, 20, 30]
    }
    df2 = pd.DataFrame(data2)
    # 将美元转换为人民币,假设汇率为 1 美元 = 7 元
    df2['价格(元)'] = df2['价格(美元)'] * 7
    # 使用 outer join 合并两个 DataFrame
    df_merged = pd.merge(df1, df2, on='商品', how='outer', suffixes=('_元', '_美元'))
    # 使用 combine_first 来合并价格列,优先选择 '价格(元)' 列的值
    df_merged['统一后价格(单位: 元)'] = df_merged['价格(元)_元'].combine_first(df_merged['价
格(元)_美元'])
```

```
    # 删除不需要的列
    df_merged.drop(columns = ['价格(元)_美元'], inplace = True)
    # 显示处理后的 DataFrame
    print(df_merged)
Out:
       商品    价格(元)_元  价格(美元)  统一后价格(单位: 元)
    0  商品 A     100.0     NaN            100.0
    1  商品 B     200.0     NaN            200.0
    2  商品 C     NaN       10.0           70.0
    3  商品 D     NaN       20.0           140.0
    4  商品 E     NaN       30.0           210.0
```

在例 3-6 中，首先创建了包含商品和价格的 DataFrame 对象 df1 和 df2，并将 df2 中的美元价格转换为人民币。然后，通过外连接合并 df1 和 df2，并使用 combine_first()函数合并价格列，最后删除了不需要的列"价格(元)_美元"以得到一个统一的商品价格列表 df_merged。

3.2.4　案例——心脑血管数据集成

本案例使用国家人口健康科学数据中心提供的两个心脑血管数据集进行数据集成，分别是"心脑血管疾病高危人群随访人员基本信息"和"心脑血管疾病高危人群随访人员健康信息"。

例 3-7　心脑血管数据集成实例。

```
In:
  # 读取 Excel 文件
  df1 = pd.read_excel(r'data/T1_BasicInfo.xlsx')
  df2 = pd.read_excel(r'data/T2_HealthInfo.xlsx')
  # 集成两个 DataFrame
  df3 = pd.merge(df1, df2, on = ['编号', 'name', 'ID'], how = 'inner', left_index = True)
  # 重置索引,使其从 1 开始
  df3.reset_index(drop = True, inplace = True)   # 先丢弃旧的索引
  df3.index = df3.index + 1                      # 将索引值加 1,使索引从 1 开始
  print(df3.head())
  df3.to_excel('data/result.xlsx')
```

集成后数据的前 5 行如图 3-2 所示。

	编号	name	ID	sex	民族	文化程度	身高	体重	收缩压	舒张压	...	吸烟	饮酒	运动习惯	高血压病	糖尿病	房颤或瓣膜性心脏病	冠心病或脑卒中家族史	既往脑卒中或短暂性脑缺血发作(TIA)	既往冠心病或心绞痛发作	危险分层
1	1	menghui	19521118018852	2	1	1.0	178.0	81.0	180.0	87.0	...	2.0	1.0	1.0	1.0	1.0	0.0	0.0	0	0	3
2	2	gejiachen	19630320015172	1	1	1.0	170.0	77.0	136.0	82.0	...	2.0	1.0	2.0	1.0	1.0	0.0	0.0	0	0	3
3	3	weishi	19770103011135	1	1	1.0	157.0	72.0	125.0	80.0	...	2.0	1.0	1.0	0.0	0.0	0.0	0.0	0	0	1
4	4	bianning	19740715011858	1	1	2.0	153.0	60.0	125.0	79.0	...	2.0	1.0	2.0	1.0	1.0	0.0	NaN	0	0	3
5	5	pangyuke	19610305012216	2	1	1.0	156.0	63.0	140.0	90.0	...	2.0	1.0	2.0	0.0	0.0	0.0	0.0	0	0	1

5 rows × 25 columns

图 3-2　心血管数据集成实例结果

例 3-7 中，使用 merge()函数将心血管基本数据和健康信息数据进行了集成，其中用 on 参数指定了用两个表共有的多关键字"编号"、name 及 ID，以内连接的方式进行合并，并将数据集成的结果写入新的数据文件。然后，重置 df3 的索引，并通过将每个索引值加 1 来确保索引从 1 而不是 0 开始。最后，将数据集成结果写入结果文件 result.xlsx。

3.3 数据清洗

数据清洗是数据预处理的重要步骤，旨在识别并纠正异常数据，检测并处理缺失值和重复记录，以提高数据质量，确保数据的完整性、唯一性、一致性和合规性等。数据清洗能够发现并修正数据中可识别的错误，过滤掉不符合要求的数据，确保数据的准确性和可靠性。在数据分析和数据挖掘过程中，异常数据可能会对分析结果产生不利影响。因此，识别并移除这些数据不仅有助于简化数据集、减少存储需求并提升处理效率，还有利于构建更稳健、精确和高效的分析模型，从而改善数据挖掘的效果。数据清洗的常规操作包括缺失值、重复值和异常值的检测与处理。

3.3.1 缺失值的检测与处理

缺失值是指数据集中某个或某些属性不完整的值。产生缺失值的原因主要分为两类：人为原因和机械原因。其中，人为原因是指数据由于人为因素未被记录或丢失，而机械原因是设备故障导致数据无法收集或存储失败。数据中存在大量缺失值会导致数据分析和挖掘建模过程中有用信息的丢失，降低输出结果的可靠性。因此，对缺失值进行检测并处理是数据清洗中重要的一环。缺失值的处理方法主要包括删除、替换和插补等，处理方法的具体选择取决于数据分布特征和实际问题需求。

1. 缺失值的检测

数据中的缺失值通常用 None 或者 numpy.NaN 来表示。二者的区别在于，numpy.NaN 是一个浮点类型的数据，而 None 是 NoneType 类型的唯一值。

Pandas 中常用的缺失值检测方法及说明如表 3-2 所示。

表 3-2 Pandas 中常用的缺失值检测方法及说明

方　法	说　明
isnull()	检测 Series 或 DataFrame 中是否存在缺失值
notnull()	检测 Series 或 DataFrame 中是否不存在缺失值
isna()	检测 Series 或 DataFrame 中是否存在缺失值
notna()	检测 Series 或 DataFrame 中是否不存在缺失值

Series 和 DataFrame 类型中的 isnull()和 isna()方法可以直接判断数据集中是否存在缺失值，若发现缺失值，则返回值为 True。而 notnull()和 notna()方法在发现缺失值时，返回值为 False。需要注意的是，isnull()方法与 isna()方法的作用相同，用于检查数据中是否存在空值(包括 None 和 NaN)。notnull()和 notna()方法的作用相同，用于筛选出数据中不为空的元素。下面以 isnull()和 notnull()方法为例进行缺失值检测。

例 3-8 使用 isnull()方法检测缺失值实例。

```
In:
    ser = pd.Series([1, np.NaN, 3, None])
    df = pd.DataFrame(dict(number = [1, 2, np.NaN],
                           fruits = ['Apple', 'Peach', None],
                           price = [np.NaN, 2.5, 3]))
    display(ser.isnull())    # 使用 isnull()方法检测 Series 类型对象 ser 中是否存在缺失值
Out:
    0    False
```

```
    1    True
    2    False
    3    True
    dtype: bool
In:
    display(df.isnull()) # 使用 isnull()方法检测 DataFrame 类型对象 df 中是否存在缺失值
Out:
        number  fruits  price
    0   False   False   True
    1   False   False   False
    2   True    True    False
```

例 3-8 中，首先分别构造了带有缺失值的 Series 和 DataFrame 类型的对象并输出。其中，Series 对象包含 1、np. NaN、3、None 4 个值，DataFrame 对象共包含 3 个缺失值。而后调用 isnull()方法检查 Series 和 DataFrame 对象中的数据。当结果显示数据为缺失值时映射为 True，其余为 False。

例 3-9 使用 notnull()方法检测缺失值实例。

```
In:
    ser = pd.Series([1, np.NaN, 3, None])
    df = pd.DataFrame(dict(number = [1, 2, np.NaN],
                           fruits = ['Apple', 'Peach', None],
                           price = [np.NaN, 2.5, 3]))
    display(ser.notnull()) # 使用 notnull ()方法检测 Series 类型对象 ser 中是否存在缺失值
Out:
    0    True
    1    False
    2    True
    3    False
    dtype: bool
In:
    display(df.notnull()) # 使用 notnull ()方法检测 DataFrame 类型对象 df 中是否存在缺失值
Out:
        number  fruits  price
    0   True    True    False
    1   True    True    True
    2   False   False   True
```

例 3-9 创建了对象，并通过调用 notnull()方法进行了缺失值检测。结果显示，notnull()方法和 isnull()方法一样，都是用于判断数据中是否存在缺失值。但与 isnull()方法不同的是，当 notnull()方法发现数据中存在缺失值时，它会将该值映射为 False。

2. 缺失值的处理

1）删除缺失值

当缺失值的样本在整体数据样本中所占比例较小时，可以采用删除缺失值的方法进行处理。但删除法存在很大的局限性。例如，处理数据量庞大、存在较多缺失值的样本时，删除缺失值会减少大量样本，从而影响信息的客观性和结果的正确性。Pandas 中常用的缺失值删除方法为 dropna()，此方法可以删除含有缺失值的行或列。

dropna()方法的语法如下：

```
dropna(axis = 0, how = 'any', thresh = None, subset = None, inplace = False)
```

dropna()方法的主要参数及说明如表 3-3 所示。

表 3-3　dropna()方法的主要参数及说明

参　　数	说　　明
axis	axis 用来指定过滤行或列，一般默认为 0。当 axis=0 时，删除包含缺失值的行，当 axis=1 时，删除包含缺失值的列
how	how 用来指定过滤的标准，默认值为 any。how='any'表示只要有缺失值存在就删除该行或列。how='all'表示如果该行或列全部为缺失值，就予以删除
thresh	用于设定阈值，表示有效数据量的最小要求
subset	表示在特定的子集中寻找缺失值
inplace	表示是否在原数据上进行操作，默认值为 False。当 inplace='False'时，表示修改在原数据的副本上进行，返回的是新数据。当 inplace='True'时，表示修改的是原数据

例 3-10　缺失值删除实例。

```
In:
    #创建一个包含缺失值的数据集
    df = pd.DataFrame({'A': [1, 2, None, 4, 5],
                       'B': [None, 6, 7, 8, None],
                       'C': [9, 10, 11, None, 12]})
    df.dropna()                          #删除数据集中的缺失值
Out:
        A    B     C
    1  2.0  6.0  10.0
```

例 3-10 演示了如何使用 dropna()方法处理数据中的缺失值。首先，创建了一个包含缺失值的 DataFrame 对象数据，然后使用 dropna()方法删除了包含缺失值的行并输出结果。

2）替换缺失值

当数据中存在缺失值时，可以采用一定的规则或方法对缺失值进行替换，从而使数据变得完整。通常情况下，如果缺失值为数值型数据，可以使用均值(mean)、中位数(median)及众数(mode)等描述数据趋势的统计量替换缺失值，也可以使用常数填充。如果缺失值为类别型数据，可以使用众数进行替换。Pandas 库中的 fillna()方法可实现缺失值替换。fillna()方法的语法如下：

```
pandas.DataFrame.fillna(value = None, method = None, axis = None, inplace = False, limit = None)
```

fillna()方法的主要参数及说明如表 3-4 所示。

表 3-4　fillna()方法的主要参数及说明

参　　数	说　　明
value	用来填充缺失值的标量、字典、Series 或者 DataFrame
method	替换的方式。常用的值有 bfill 或 ffill。bfill 表示用后面的值填充前面的缺失值，ffill 表示用前面的值填充后面的缺失值
axis	指定填充的轴向。0 或 index 表示按列填充，1 或 columns 表示按行填充
inplace	布尔值，如果为 True，则在原 DataFrame 上进行填充，不返回新的对象

注意：method 参数和 value 参数不能同时使用。

除此之外，还可以使用 sklearn.impute 中的 SimpleImputer 填补缺失值，首先根据数据类型选择合适的填充策略（如均值、中位数、众数等），然后对指定的列应用 fit_transform 方法，该方法会计算填充值并用这些值替换掉原有的缺失值。

例 3-11　替换缺失值实例。

```
In:
    from sklearn.impute import SimpleImputer
    # 创建原始 DataFrame
    df = pd.DataFrame({
        'A': [1, 2, np.NaN, 4, 5],
        'B': [np.NaN, np.NaN, 7, 8, np.NaN],
        'C': [9, 10, 10, 12, np.NaN]
    })
    # 使用 fillna 填充 A 列的缺失值,用均值
    df['A'].fillna(df['A'].mean(), inplace=True)
    # 使用 fillna 填充 B 列的缺失值,使用中位数
    df['B'].fillna(df['B'].median(), inplace=True)
    # 使用 SimpleImputer 填充 C 列的缺失值,用众数填充
    cat_imputer = SimpleImputer(strategy='most_frequent')
    df['C'] = cat_imputer.fit_transform(df[['C']])
    # 打印填充后的数据
    print(df)
Out:
         A    B     C
    0  1.0  7.5   9.0
    1  2.0  7.5  10.0
    2  3.0  7.0  10.0
    3  4.0  8.0  12.0
    4  5.0  7.5  10.0
```

例 3-11 中,首先创建了一个包含缺失值的 DataFrame 对象,使用 fillna 方法和每列的均值来填充 A 列的缺失值。然后,使用 fillna 方法和中位数来填充 B 列的缺失值。对于 C 列,使用 SimpleImputer 的众数策略来填充缺失值。最后,输出替换缺失值后的 DataFrame。

3）插补缺失值

缺失值插补是一种常用的数据清洗方法,其目的是通过指定的算法估计和替换数据中的缺失值,以便更好地进行数据分析和建模。与简单预测单个缺失值不同,插补的目标是确定缺失数据所服从的分布。常见的插补方法包括线性插补、多项式插补等,而更复杂的方法如多重插补则是通过多次迭代来估计缺失值,并在每次迭代中使用已知的值来填补缺失值。选择何种插补方法应根据数据特性和分析需求来确定。插补是一种估计缺失值的方法,通过已知的值来估计未知的值,而替换缺失值通常是直接用一个固定的值来代替缺失值。

Pandas 库中的 interpolate()方法可用于插补缺失值,语法如下:

```
pandas.DataFrame.interpolate(method = 'linear', axis = 0, limit = None, inplace = False, limit_
direction = ('forward', 'backward'), limit_area = None, downcast = None, **kwargs)
```

interpolate()方法的主要参数及说明如表 3-5 所示。

表 3-5　interpolate()方法的主要参数及说明

参　数	说　明
method	插值方法,可选值为 linear(线性插值)、polynomial(多项式插值)、spline(样条插值)等。默认为 linear
axis	指定沿哪个轴进行插值,默认为 0。如果为 1,则表示按列插值
limit	插值的最大间隔,即连续 NaN 值之间的最大差值。默认为 None,表示不限制插值范围
limit_direction	指定插值的方向,可选值为 forward(向前填充)和 backward(向后填充)。默认为 forward

例 3-12　线性插补缺失值实例。

```
In:
    #创建一个包含缺失值的DataFrame
    data = {'A': [1, np.nan, 3, 4, np.nan, 6],
            'B': [4, 5, 6, np.nan, 8, 9]}
    df = pd.DataFrame(data)
    df_interpolated = df.interpolate(method='linear')   # 对缺失值进行线性插补
    print(df_interpolated)
Out:
        A    B
    0  1.0  4.0
    1  2.0  5.0
    2  3.0  6.0
    3  4.0  7.0
    4  5.0  8.0
    5  6.0  9.0
```

例3-12创建了一个包含缺失值的DataFrame对象df,并用interpolate()方法分别对其A、B列中的缺失值进行了线性插补并输出结果。

3.3.2 重复值的检测与处理

重复值指数据集中存在多个完全或部分相同的记录或样本。导致重复值出现的原因有很多,如数据录入错误、系统故障重复采集等。数据集中大量重复值的存在会导致实际数据分析和建模结果与预期产生较大的偏差,模型准确性降低,从而造成对数据的错误解读,产生误导性的结论。因此,在数据预处理阶段,对重复值进行检测和处理是非常重要的一个步骤。

1. 重复值的检测

Pandas库中的duplicated()方法可以检测数据中是否有重复值,语法如下:

```
pandas.DataFrame.duplicated(subset=None, keep='first', inplace=False)
```

表3-6所示为duplicated()方法的主要参数及说明。

表3-6 duplicated()方法的主要参数及说明

参数	说明
subset	可选参数,可以是列名的字符串或列名的列表。如果指定,则只考虑这些列来检测重复项。如果不指定,则使用所有列
keep	可选参数,指定如何处理重复项。默认值是first,表示保留首次出现的重复项。其他可选值包括last:保留最后出现的重复项,去除之前的重复项;False:去除所有重复项;True:保留所有重复项
inplace	布尔值,可选参数。如果设置为True,则直接在原始DataFrame上进行操作,返回None。如果设置为False(默认值),则返回一个布尔序列的新Series,其中True表示重复项,False表示非重复项

如果inplace=False(默认),返回一个布尔Series,表示每个行是否是重复的。如果inplace=True,则修改原始DataFrame,移除重复行,并返回None。

例3-13 完全重复值的检测实例。

```
In:
    df = pd.DataFrame(dict(number=[1, 2, 3, 4, 5, 6, 3, 3],
                           name=['Tom', 'Kate', 'Tom','Bob','Kate','Lily','Tom','Tom'],
                       birthday=[pd.Timestamp('1995-09-21'),pd.Timestamp('1980-01-08'),
```

```
pd.Timestamp('1995 - 09 - 21'),pd.Timestamp('1980 - 01 - 03'),pd.Timestamp('1980 - 01 - 08'),
pd.Timestamp('1980 - 01 - 05'),pd.Timestamp('1995 - 09 - 21'),pd.Timestamp('1995 - 09 - 21')],
role = [None, 'Student', 'Teacher', 'Student', 'Student', 'Student','Teacher','Teacher'], score =
[100,90,80,70,60,50,80,80]))
    df.duplicated()
Out:
    0       False
    1       False
    2       False
    3       False
    4       False
    5       False
    6       True
    7       True
    dtype: bool
```

例 3-13 首先构建了一个包含 8 行数据的 DataFrame 对象 df,其中第 6 和第 7 行数据完全相同,第 0 行和第 2 行某些列上数据相同。调用 duplicated()方法,返回一个布尔序列,其中 True 表示对应的行是完全重复的,False 表示对应的行不是完全重复的。

例 3-14 不完全重复值的检测实例。

```
In:
  df = pd.DataFrame(dict(number = [1, 2, 3, 4, 5, 6, 3, 3],
                         name = ['Tom', 'Kate', 'Tom','Bob','Kate','Lily','Tom','Tom'],
                         birthday = [pd.Timestamp('1995 - 09 - 21'),pd.Timestamp('1980 - 01 - 08'),
pd.Timestamp('1995 - 09 - 21'),pd.Timestamp('1980 - 01 - 03'),pd.Timestamp('1980 - 01 - 08'),
pd.Timestamp('1980 - 01 - 05'),pd.Timestamp('1995 - 09 - 21'),pd.Timestamp('1995 - 09 - 21')],
role = [None, 'Student', 'Teacher', 'Student', 'Student', 'Student','Teacher','Teacher'],
                           score = [100,90,80,70,60,50,80,80]))
  df.duplicated(subset = ['name','birthday'])
Out:
  0    False
  1    False
  2    True
  3    False
  4    True
  5    False
  6    True
  7    True
  dtype: bool
```

例 3-14 调用 duplicated()方法,并在函数中添加 subset 标签,用于检测指定标签列中数据完全重复的行。输出结果中,重复值映射为 True,非重复值映射为 False。其中,第 4 行与第 1 行在 name 和 birthday 列上的值完全相同,即都是 Kate 和 1980-01-08 的组合,因此,duplicated()函数返回 True,表示第 4 行是重复的。同样,第 2 行、第 6 行和第 7 行在 name 和 birthday 列上的值都是 Tom 和 1995-09-21,与第 0 行相同,即这 3 行在 name 和 birthday 列与第 0 行上完全相同,因此这 3 行也是重复的。

2. 重复值的处理

1) 识别重复数据与数据排重

识别重复数据是指在数据集中找出具有相同特征的多个数据,而数据排重是在识别出重复数据后,将这些重复数据从数据集中移除,只保留其中一个或几个样本。这两个步骤可以提高数据质量和准确性,避免分析错误和决策偏差,同时减小数据集规模,提高数据处理效率。

例 3-15 重复数据的识别实例。

```
In:
  df[df.duplicated()]
Out:
   number  name   birthday       role    score
  6  3     Tom   1995-09-21   Teacher    80
  7  3     Tom   1995-09-21   Teacher    80
In:
  df[df.duplicated(['name','birthday'])]
Out:
     number  name     birthday       role    score
  2    3      Tom     1995-09-21    Teacher     80
  4    5      Kate    1980-01-08    Student     60
  6    3      Tom     1995-09-21    Teacher     80
  7    3      Tom     1995-09-21    Teacher     80
```

例 3-15 调用 duplicated()方法筛选出了所有重复数据。第一种情况未指定列,默认基于所有列进行重复判断;第二种情况指定了 name 和 birthday 两列,基于这两列进行重复判断。

2)去除重复数据

在标记出重复数据后,可以使用 drop_duplicates()方法删除这些重复值。

例 3-16 重复数据的去除实例。

```
In:
  # 删除重复行,保留每组重复行中的最后一行
  df1 = df.drop_duplicates(keep = 'last')
  display(df1)
Out:
   number  name    birthday      role     score
  0  1     Tom     1995-09-21   None      100
  1  2     Kate    1980-01-08   Student   90
  3  4     Bob     1980-01-03   Student   70
  4  5     Kate    1980-01-08   Student   60
  5  6     Lily    1980-01-05   Student   50
  7  3     Tom     1995-09-21   Teacher   80
In:
  df2 = df1.drop_duplicates(subset =  ['name','birthday'])
  display(df2)
Out:
     number  name  birthday       role       score
  0  1       Tom   1995-09-21   None        100
  1  2       Kate  1980-01-08   Student     90
  3  4       Bob   1980-01-03   Student     70
  5  6       Lily  1980-01-05   Student     50
```

例 3-16 进行了两次 drop_duplicates()调用。第一次调用时,通过设置 keep='last'参数保留了每组重复行的最后一行。第二次调用时,指定了 subset=['name', 'birthday']参数,如果两行在 name 和 birthday 列上的值相同,则只保留其中的一行。

3.3.3 异常值的检测与处理

异常值是指在样本数据中与其他样本显著不同的离群点,其来源主要包括人为误差和自然误差,具体可以分为以下几类:数据输入错误、测量误差、实验误差、故意异常值、数据处理错误、抽样错误及自然异常值。异常值往往会改变数据的统计特征,如平均值、方差等,进而导致错误的结论或预测结果。此外,异常值还可能影响模型的拟合效果,降低模型对数据的解释能力。

因此,在进行数据分析前,通常需要进行异常值的检测与处理,以确保数据的质量和准确性。

1. 异常值的检测

异常值的检测方法有很多,常用的方法包括 Z 分数(Z-Score)法、箱线图(Box Plot)分析法、散点图法、DBSCAN 检测法等,下面结合实例进行介绍。

例 3-17 异常值检测实例——Z 分数(Z-Score)法。

```
In:
    from scipy import stats
    # 收缩压数据,包括一些极端值
    sbp_data = np.array([
        120, 125, 130, 135, 140, 145, 150, 155,   # 正常范围内的数据
        160, 165, 170, 175, 180, 185, 190,        # 边缘高血压的数据
        250, 260, 270, 280, 290, 300, 310,        # 明显异常高的数据
        50, 45, 40, 35, 30, 25, 20                # 明显异常低的数据
    ])
    # 计算每个数据点的 Z 分数
    z_scores = stats.zscore(sbp_data)
    # 定义阈值,通常如果|Z| > 3,则认为该点为异常值
    #如果数据不是严格符合正态分布,或者希望更敏感地检测到异常值,可以适当降低这个阈值,本
    #例设置为 1
    threshold = 1
    outliers = np.where(np.abs(z_scores) > threshold)
    # 打印原始数据中的异常值及其对应的 Z 分数
    if outliers[0].size > 0:
        print("异常值:")
        for index in outliers[0]:
            print(f"收缩压: {sbp_data[index]}, Z 分数: {z_scores[index]:.2f}")
    else:
        print("没有检测到异常值")
    # 如果需要,也可以打印出所有非异常的数据点
    non_outliers = sbp_data[np.abs(z_scores) <= threshold]
    print("\n 非异常值:")
    print(non_outliers)
Out:
    异常值:
    收缩压: 250, Z 分数: 1.08
    收缩压: 260, Z 分数: 1.19
    收缩压: 270, Z 分数: 1.30
    ……
    收缩压: 25, Z 分数: -1.50
    收缩压: 20, Z 分数: -1.56
    非异常值:
    [120 125 130 135 140 145 150 155 160 165 170 175 180 185 190]
```

例 3-17 中,采用 Z 分数法检测异常值。首先,定义了一个包含收缩压测量值的数组 sbp_data。然后,使用 stats.zscore 函数来计算每个数据点相对于整体数据集均值和标准差的 Z 分数。接着,通过设定一个阈值(这里设置为 1),可以找出哪些数据点被认为是异常值。最后,输出被标记为异常值的收缩压数值及它们相应的 Z 分数,部分结果如上所示。

通常可以设置 $|Z|>3$,如果数据不是严格符合正态分布,或希望更敏感地检测到异常值,可以适当降低这个阈值。通过以下代码,可以检查并输出不同阈值下的异常值检测结果。

```
In:
  def detect_outliers(data, threshold):
      z_scores = stats.zscore(data)
      outliers = np.where(np.abs(z_scores) > threshold)
```

```
        return outliers
    # 测试不同的阈值
    thresholds = [1, 1.5, 2, 2.5, 3]
    for t in thresholds:
        outliers = detect_outliers(sbp_data, t)
        if outliers[0].size > 0:
            print(f"阈值: {t}")
            print("异常值:")
            for index in outliers[0]:
                print(f"收缩压: {sbp_data[index]}, Z 分数: {stats.zscore(sbp_data)[index]:.2f}")
        else:
            print(f"阈值: {t} - 没有检测到异常值")
```

上述代码可以检查不同阈值下的异常值检测结果,输出略。此外,还可以结合直方图和箱线图的可视化结果,以及业务需求,选择一个合适的阈值。通常情况下,如果希望减少误报并且数据大致符合正态分布,可以选择较高的阈值(如 3)。如果希望更敏感地检测到异常值,可以选择较低的阈值(如 1 或 1.5)。注意,选择阈值是一个迭代过程,可能需要多次调整和测试才能找到最佳值。

例 3-18 异常值检测实例——IQR 分数法结合箱线图(Box Plot)分析。

```
In:
    import numpy as np
    import matplotlib.pyplot as plt
    # 设置 matplotlib 支持中文字体
    plt.rcParams['font.sans-serif'] = ['SimHei']       # 设置字体为黑体
    plt.rcParams['axes.unicode_minus'] = False         # 正确显示负号
    # 正常范围的血红蛋白水平数据
    normal_hb_data = np.random.normal(13, 1.5, 100)   # 假设正常范围是平均值为 13,标准差为 1.5
    # 添加异常值: 过低的血红蛋白水平(贫血)和过高的血红蛋白水平(多血症)
    anomalous_low_hb = np.random.normal(8, 0.5, 10)   # 贫血患者的血红蛋白水平
    anomalous_high_hb = np.random.normal(18, 0.5, 10) # 多血症患者的血红蛋白水平
    # 合并数据集
    hb_data = np.concatenate([normal_hb_data, anomalous_low_hb, anomalous_high_hb])
    # 计算 IQR
    Q1 = np.percentile(hb_data, 25)
    Q3 = np.percentile(hb_data, 75)
    IQR = Q3 - Q1
    # 计算异常值阈值
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    # 打印四分位数和边界
    print(f"Q1: {Q1}")
    print(f"Q3: {Q3}")
    print(f"IQR: {IQR}")
    print(f"Lower Bound: {lower_bound}")
    print(f"Upper Bound: {upper_bound}")
    # 检测异常值
    outliers = hb_data[(hb_data < lower_bound) | (hb_data > upper_bound)]
    # 打印异常值
    print("异常值:")
    for outlier in outliers:
        print(outlier)
    # 绘制箱线图
    plt.figure(figsize=(10, 6))
    plt.boxplot(hb_data, vert=False, patch_artist=True, showmeans=True, labels=['血红蛋白水平'])
    plt.title('血红蛋白水平数据的箱线图')
    plt.xlabel('血红蛋白水平 (g/dL)')
    plt.grid(True)
    plt.show()
```

异常值输出结果略，箱线图结果如图 3-3 所示。

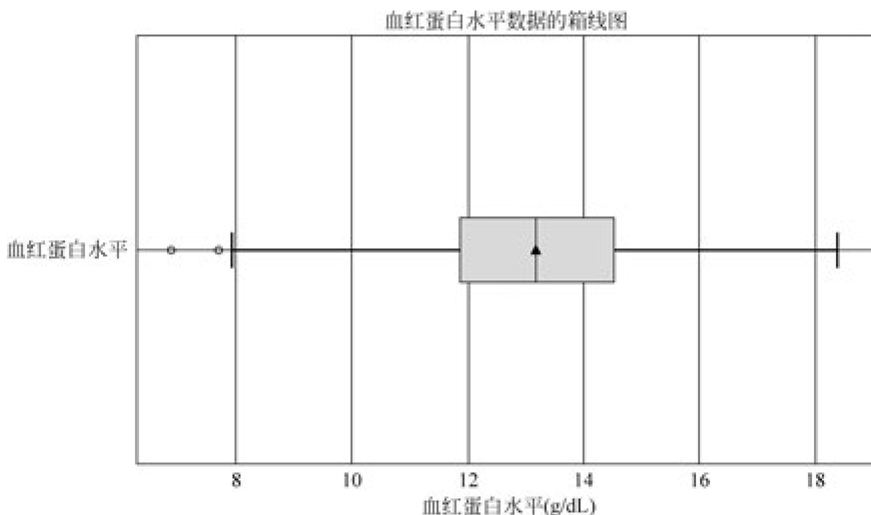


图 3-3 IQR 分数法结合箱线图检测异常值

例 3-18 使用四分位数范围(IQR)分数法结合箱线图(Box Plot)进行异常值检测。首先，通过 NumPy 库生成了一组模拟的正常血红蛋白水平数据，并添加了两组模拟的异常值数据，分别代表贫血(血红蛋白水平过低)和多血症(血红蛋白水平过高)的情况。接着，计算了数据的第一和第三、四分位数(Q1 和 Q3)及四分位数范围(IQR)，并据此确定了异常值的上下界。然后，打印出了四分位数和异常值边界，并检测并打印出了所有异常值。最后，使用 matplotlib 库绘制了一个箱线图，直观地展示了数据的分布情况，包括中位数、四分位数和异常值，以识别数据中的异常情况。

例 3-19 异常值检测实例——散点图法。

```
In:
  wdf = pd.DataFrame(np.arange(20),columns = ['W'])
  wdf['Y'] = wdf['W'] * 1.5 + 2
  #在数据中增加两个异常值
  wdf.iloc[3,1] = 128
  wdf.iloc[18,1] = 150
  wdf.plot(kind = 'scatter',x = 'W',y = 'Y')
  plt.show()
```

结果如图 3-4 所示。

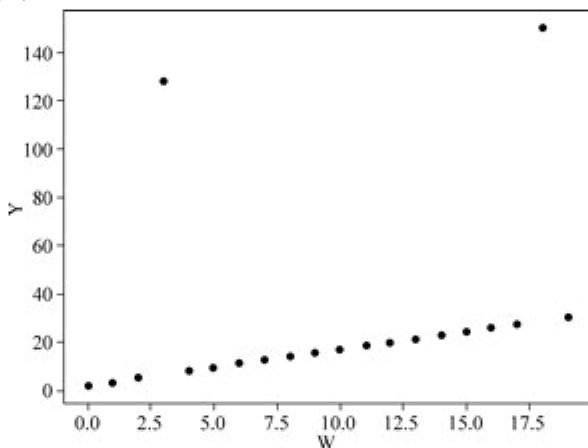


图 3-4 散点图检测异常值

例 3-19 中,采用了散点图法进行异常值检测。以 W 为 x 轴,Y 为 y 轴,绘制散点图。从图 3-4 中可以看出,当 x 为 3 和 18 时,y 轴的值出现了显著的偏差,表明此位置存在异常值。

请注意,采用散点图检测异常值的有效性取决于数据的分布和异常值的类型。对于高维数据或者更复杂的分布情况,可能需要更高级的统计方法或机器学习技术来检测异常值。

例 3-20 异常值检测实例——DBSCAN 检测法。

```
In:
  from sklearn.preprocessing import StandardScaler
  from sklearn.cluster import DBSCAN
  data = {'name':['张三','李四','王五','赵六','钱七','刘九','李十','阿旺'],
          'age':[20,24,19,150,21,30,35,28],
          'income':[20000,18000,1122000,21000,23000,18000,19000,11000]}
  df = pd.DataFrame(data)
  X = df[['age', 'income']].values
  scaler = StandardScaler()
  X_scaled = scaler.fit_transform(X)
  db = DBSCAN(eps = 2, min_samples = 3).fit(X_scaled)
  labels = db.labels_
  outliers = np.where(labels == -1)[0]
  print('Detected outliers:')
  for i in outliers:
    print('Index:', i)
    print(df.loc[i])
Out:
  Detected outliers:
  Index: 2
  name          王五
  age             19
  income      1122000
  Name: 2, dtype: object
  Index: 3
  name          赵六
  age            150
  income       21000
  Name: 3, dtype: object
```

例 3-20 中,采用 DBSCAN 检测法对异常值进行检测。首先,构建包含年龄和收入两个特征的 DataFrame 对象。提取特征进行标准化处理后,使用 DBSCAN 算法对标准化后的数据进行聚类,将无法聚类的噪声点标记为-1,作为异常点。最后遍历所有样本,并输出标签为-1 的样本索引和内容,即为检测到的异常点。

2. 异常值的处理

例 3-21 异常值处理实例——删除异常值。

```
In:
  df.drop(index = outliers, inplace = True)
  print('Data after removing outliers:')
  print(df)
Out:
  Data after removing outliers:
      name   age  income
  0   张三    20   20000
  1   李四    24   18000
  4   钱七    21   23000
  5   刘九    30   18000
```

```
  6   李十   35   19000
  7   阿旺   28   11000
```

例 3-21 中,在异常值的处理中,采用直接删除的方法。利用 drop()方法直接删除异常值所在的对应行并输出新的数据集。

例 3-22 异常值处理实例——回归预测。

```
In:
  from sklearn.linear_model import LinearRegression
  data = {'name':['张三','李四','王五','赵六','钱七','刘九','李十','闫五'],
          'age':[20,24,19,150,21,30,35,28],
          'income':[20000,18000,1122000,21000,23000,18000,19000,11000]}
  df = pd.DataFrame(data)
  #预测年龄的异常值
  X_train = df[df['age']< 60][['income']]
  y_train = df[df['age']< 60]['age']
  age_model = LinearRegression()
  age_model.fit(X_train, y_train)
  df.loc[df['age']>=60, 'age'] = age_model.predict(df[df['age']>=60][['income']])
  #预测收入的异常值
  X_train = df[df['income']< 80000][['age']]
  y_train = df[df['income']< 80000]['income']
  income_model = LinearRegression()
  income_model.fit(X_train, y_train)
  df.loc[df['income']>=80000, 'income'] =
        income_model.predict(df[df['income']>=80000][['age']])
  df['age'] = df['age'].astype(int)
  df['income'] = df['income'].astype(int)
  print(df)
Out:
      name  age  income
  0   张三   20   20000
  1   李四   24   18000
  2   王五   19   20450
  3   赵六   26   21000
  4   钱七   21   23000
  5   刘九   30   18000
  6   李十   35   19000
  7   闫五   28   11000
```

例 3-22 中,采用了线性回归模型来预测并修正样本数据中的异常值。首先,分别针对年龄和收入构建了回归模型,利用数据集中正常范围的样本进行拟合。随后,使用这些模型对异常值部分进行预测,并将预测结果替换原有的异常值。最终,输出了经过修正的样本数据。异常值处理的具体成效如下:原本异常的年龄值 150 岁(赵六)被修正为预测值 26 岁;原本异常的收入值 1122000 元(王五)也被调整至预测值 20450 元。

3.3.4 案例——心脑血管数据值清洗

本节中的心脑血管疾病高危人群数据分为"心脑血管疾病高危人群随访人员基本信息"和"心脑血管疾病高危人群随访人员健康信息"两个表,共 5000 例。本节将对心脑血管疾病高危人群数据逐步进行缺失值检测预处理、重复值检测预处理和异常值检测预处理等数据清洗任务。

1. 缺失值检测与处理案例

例 3-23 心脑血管疾病高危人群数据缺失值检测与处理。

首先，导入例3-7中的数据并整合后得到的数据文件result.xlsx，将数据中的NULL值视为缺失值处理。接着，将字符类型列中的空值替换为NaN，检测并统计数据中的缺失值。

```
In:
    #导入已集成的心血管数据
    df = pd.read_excel('data/result.xlsx')
    display(df)
    # 将数据中的#NULL!值作为缺失值处理，将其替换为NaN
    # 只对字符串类型的列进行操作
    for column in df.select_dtypes(include = ['object']).columns:
        df[column] = df[column].replace('#NULL!', np.NaN)
    # 检查数据集中的缺失值
    missing_values = df.isnull().sum()
    # 打印每个字段的缺失值数量
    print("缺失值统计：")
    print(missing_values)
Out:
    name                              0
    ID                                0
    sex                               0
    民族                              0
    文化程度                          113
    身高                              0
    体重                              0
    收缩压                            91
    舒张压                            91
    总胆固醇                          57
    甘油三酯                          7
    高密度脂蛋白胆固醇                61
    低密度脂蛋白胆固醇                6
    空腹血糖                          157
    吸烟                              33
    饮酒                              24
    运动习惯                          29
    高血压病                          26
    糖尿病                            18
    房颤或瓣膜性心脏病                13
    冠心病或脑卒中家族史              383
    既往脑卒中或短暂性脑缺血发作(TIA) 0
    既往冠心病或心绞痛发作            0
    危险分层                          0
    dtype: int64
```

对于变量类型为“类别型”的缺失值，采用众数替换的方式进行填充。首先，定义包含想要计算众数的字段名列表fields，然后将众数转换为列表，打印字段名和众数，如果众数有多个，则会打印出所有的众数。

```
In:
  # 列表包含想要计算众数的"类别型"字段的字段名
  fields = ['文化程度', '吸烟', '饮酒', '运动习惯', '高血压病', '糖尿病', '房颤或瓣膜性心脏病',
'冠心病或脑卒中家族史']
  for field in fields:
      mode = df[field].mode().tolist()  # 将众数转换为列表
      print(f"{field}: {mode[0] if len(mode) == 1 else mode}")  # 打印字段名和众数，如果众数有
                                                                # 多个，则会打印出所有的众数
Out:
    文化程度：1.0
    吸烟：2.0
```

```
    饮酒: 1.0
    运动习惯: 1.0
    高血压病: 0.0
    糖尿病: 0.0
    房颤或瓣膜性心脏病: 0.0
    冠心病或脑卒中家族史: 0.0
```

计算 fields 列表中每个字段的众数,使用众数填充缺失值,并将结果存储在新的 DataFrame 对象 df_n 中,输出结果略。

```
In:
  # 计算 fields 列表中每个字段的众数
  mode_values = df[fields].mode().iloc[0]
  # 使用众数填充缺失值,并将结果存储在新的 DataFrame 对象 df_n 中
  df_n = df.copy()
  df_n[fields] = df_n[fields].fillna(mode_values)
  display(df_n)
```

对于变量类型为数值型的缺失值,采用中位数的方式进行填充。下面的代码中,首先创建了一个包含所有数值型字段名的列表 numeric_fields。然后,通过循环遍历这个列表,计算每个字段的中位数,并将这些中位数存储在一个字典 median_values 中,并输出这个字典。

```
In:
    # 列表包含想要填充中位数的"数值型"字段名
    numeric_fields = ['收缩压', '舒张压', '总胆固醇', '甘油三酯', '高密度脂蛋白胆固醇', '低密度脂蛋白胆固醇', '空腹血糖']
    median_values = {} # 创建一个字典来存储每个字段的中位数
    for field in numeric_fields: # 计算每个字段的中位数并存储在字典中
        median_values[field] = df_n[field].median()
    print(median_values) #输出放中位数的字典
Out:
    {'收缩压': 130.0, '舒张压': 80.0, '总胆固醇': 5.12, '甘油三酯': 1.38, '高密度脂蛋白胆固醇': 1.6, '低密度脂蛋白胆固醇': 3.2, '空腹血糖': 5.56}
```

使用 fillna()方法将缺失值替换为对应字段的中位数,并将结果存储在新的 DataFrame 对象 df_new 中,输出结果略。

```
In:
    df_new = df_n.fillna(median_values)    # 使用中位数填充缺失值,并将结果存储在 df_new 中
    display(df_new)                        # df_new 包含了使用中位数填充缺失值后的数据
```

再次检测数据中是否还有缺失值,发现各个字段均没有缺失值。

```
In:
    df_new.isnull().sum()      # 再次检测数据中是否还有缺失值
Out:
    name                   0
    ID                     0
    ......
    既往冠心病或心绞痛发作   0
    危险分层               0
```

本例中,首先对数据进行了集成处理,在集成后的完整数据中,先将数据中的 NULL 值视作缺失值进行处理,然后使用 isnull().sum()方法检测数据中的缺失值。经统计发现,数据中的缺失值有类别型和数值型两种。最后,使用 fillna()方法对类别型的缺失值使用众数进行替换,对数值型的缺失值使用其中位数进行替换,替换完成后再次检测发现数据中已无缺失值。

2. 重复值检测与处理案例

在对数据缺失值进行检测与处理后，使用上个案例中处理好的 df_new 对象调用 duplicated()方法，进行完全重复值、身份信息重复值和健康信息重复值的筛查。

例 3-24 心脑血管疾病高危人群数据重复值检测与处理。

（1）完全重复值的检测。

```
In:
   df_new[df_new.duplicated()]
Out:
   name  ID  sex  民族    文化程度    身高    体重    收缩压  ...  危险分层    编号
   0 rows × 24 columns
```

（2）身份信息重复值的检测。

```
In:
    df_new[df_new.duplicated(['name','ID'])]
Out:
name  ID    sex  民族    文化程度    身高    体重    收缩压  ...  危险分层    编号
    0 rows × 24 columns
```

（3）健康信息重复值的检测。

```
In:
    df_new[df_new.duplicated(subset=['收缩压','舒张压','总胆固醇','甘油三酯','高密度脂蛋白胆
固醇','低密度脂蛋白胆固醇','空腹血糖'])]
Out:
name ID    sex  民族    文化程度       身高 体重  收缩压  ...  危险分层  编号
346 cengzhonghuan    19400629016215  2  1      1.0  172.0  85.0    130.0   80.0  3.36
...  2.0  1.0  1.0  0.0  0.0  0.0  0.0  0   0   1
......
2207    tongbei  19790914016449  2    1    2.0 160.0  65.0  120.0    80.0    5.00    ...
2.0 1.0 2.0 0.0 0.0 0.0 NaN  0    0    1
35 rows × 24 columns
```

对健康信息重复值的检测输出了 35 行结果，说明有 35 行在健康信息列的取值上与其他行存在重复值。

3. 异常值检测与处理案例

例 3-25 心脑血管疾病高危人群数据异常值检测与处理。

（1）IQR 结合箱线图法检测“身高”和“体重”数据异常值。

首先，使用 IQR 结合箱线图的方法进行个人“身高”和“体重”数据的异常值检测，代码如下：

```
In:
    import matplotlib.pyplot as plt
    # 设置 matplotlib 支持中文字体
    plt.rcParams['font.sans-serif'] = ['SimHei']  # 设置字体为黑体
    plt.rcParams['axes.unicode_minus'] = False     # 正确显示负号
    def calculate_iqr_and_detect_outliers(df, column_name):
        Q1 = df[column_name].quantile(0.25)
        Q3 = df[column_name].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
```

```
        outliers = df[(df[column_name] < lower_bound) | (df[column_name] > upper_bound)]
        print(f"{column_name}列四分位数和边界: Q1 = {Q1}, Q3 = {Q3}, IQR = {IQR}, 下界 = {lower_
bound}, 上界 = {upper_bound}")
        print(f"\n{column_name}列异常值:")
        print(outliers[['编号', column_name]])
        print(f"{column_name}异常值个数: {len(outliers)}")
        return Q1, Q3, IQR, lower_bound, upper_bound, outliers

    def plot_boxplot(df, column_names, titles, figsize = (14, 6), dpi = 300):
        plt.figure(figsize = figsize, dpi = dpi)
        for i, col in enumerate(column_names, start = 1):
            plt.subplot(1, len(column_names), i)
            plt.boxplot(df[col], vert = False, patch_artist = True, showmeans = True, labels =
[titles[i - 1]])
            plt.title(titles[i - 1])
            plt.grid(True)
        plt.tight_layout()
        plt.show()

    # 对身高列执行异常值检测
    Q1_height, Q3_height, IQR_height, lower_bound_height, upper_bound_height, outliers_height =
calculate_iqr_and_detect_outliers(df_new, '身高')

    # 对体重列执行异常值检测
    Q1_weight, Q3_weight, IQR_weight, lower_bound_weight, upper_bound_weight, outliers_weight =
calculate_iqr_and_detect_outliers(df_new, '体重')

    # 绘制箱线图
    plot_boxplot(df_new, ['身高', '体重'], ['身高数据的箱线图', '体重数据的箱线图'])
Out:
    身高列四分位数和边界: Q1 = 156.0, Q3 = 168.0, IQR = 12.0, 下界 = 138.0, 上界 = 186.0
    体重列四分位数和边界: Q1 = 57.0, Q3 = 71.0, IQR = 14.0, 下界 = 36.0, 上界 = 92.0
```

检测结果中，身高异常值个数为 23，体重异常值个数为 41，具体异常值输出结果略。身高和体重的异常值箱线图检测结果如图 3-5 所示。

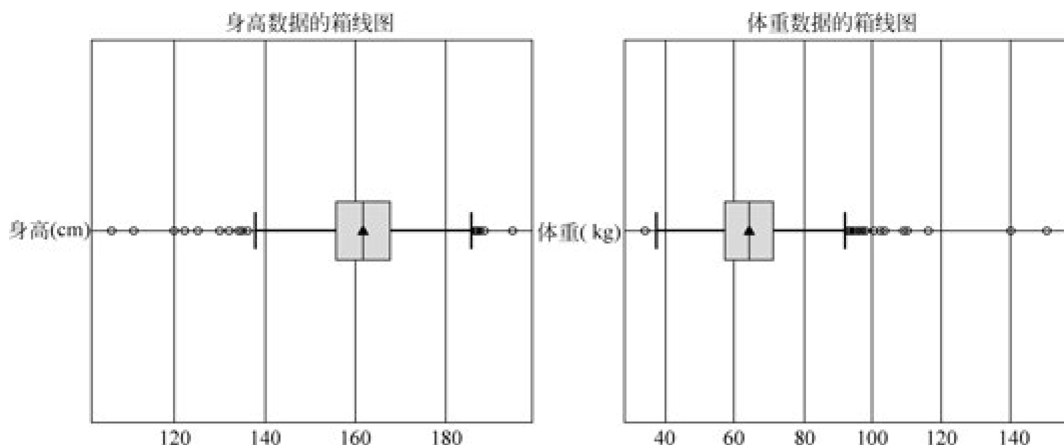


图 3-5　身高和体重的异常值箱线图检测结果

（2）使用 Z 分数方法检测指定的数值型列中的异常值，并打印出异常值及其个数，代码如下：

```
In:
  from scipy import stats
```

```
import matplotlib.pyplot as plt
# 选择数值型列
numeric_columns = ['收缩压', '舒张压', '总胆固醇', '甘油三酯', '高密度脂蛋白胆固醇', '低密度脂蛋白胆固醇', '空腹血糖']
# 初始化一个字典来存储异常值及其数量
outliers_info = {}
# 对每列应用 Z-score 方法来检测异常值
for column in numeric_columns:
    # 计算 Z 分数,并排除 NaN 值
    z_scores = stats.zscore(df_new[column].dropna())
    # 计算绝对值 Z 分数
    abs_z_scores = np.abs(z_scores)
    # 确定异常值的索引
    outliers_index = np.where(abs_z_scores > 3)[0]
    # 使用索引从原始 DataFrame 中获取异常值
    outliers_values = df_new.loc[outliers_index, column]
    # 存储异常值及其数量
    outliers_info[column] = {
        'outliers': outliers_values,
        'count': len(outliers_values)
    }
# 准备数据用于绘制柱状图
columns = list(outliers_info.keys())
outliers_counts = [outliers_info[col]['count'] for col in columns]
# 绘制柱状图
plt.figure(figsize=(10, 6),dpi=300)
bars = plt.bar(columns, outliers_counts, color='skyblue')
plt.xlabel('列名')
plt.ylabel('异常值个数')
plt.title('各列异常值的个数')
plt.xticks(rotation=45)
plt.tight_layout()
# 在柱状图上添加异常值个数标签
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 1), va='bottom')
    # va=bottom 表示垂直对齐方式为底部
# 显示图表
plt.show()
```

输出结果如图 3-6 所示。

首先计算了各检查指标(数值型)列的 Z 分数,并确定了哪些值是异常值(Z 分数绝对值大于 3)。然后,收集了每个列的异常值数量,并使用 matplotlib 绘制了柱状图。在柱状图柱子顶部显示异常值数量的标签。

(3) 类别型数据的异常值检测,代码如下:

```
In:
  cols = ['高血压病','吸烟', '饮酒']
  mask0 = ~df_new[cols[0]].isin([0.0, 1.0])
  mask1 = ~df_new[cols[1]].isin([1.0, 2.0, 3.0])
  mask2 = ~df_new[cols[2]].isin([1.0, 2.0, 3.0])
  mask = mask0 | mask1 | mask2
  # 过滤
  df_filtered = df_new[mask]
  df_filtered
Out:
name  ID  sex  民族  文化程度  身高  体重  收缩压  ...  危险分层  编号
    0 rows × 24 columns
```

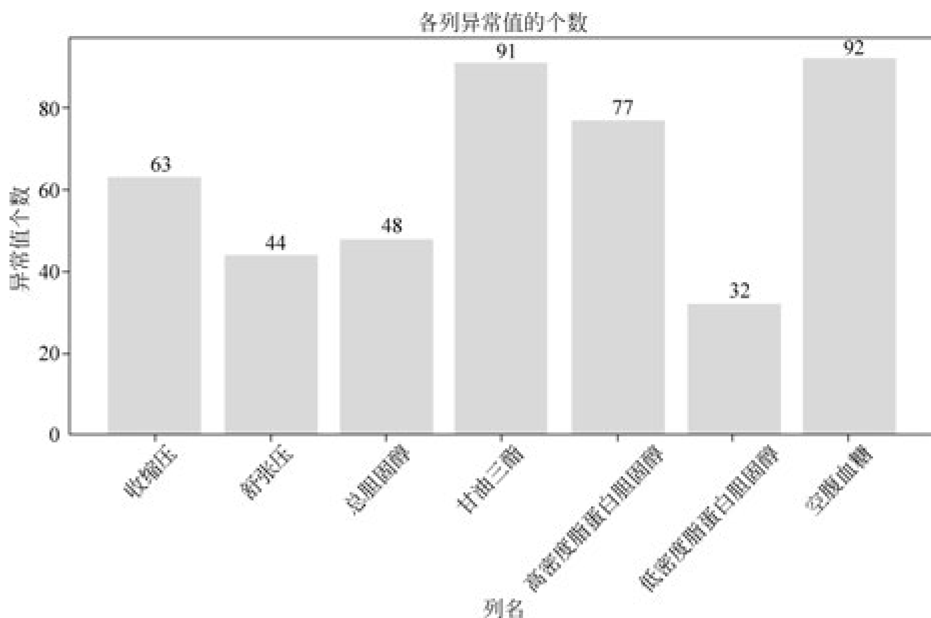


图 3-6　检查指标列的异常值检测结果

数据中,'高血压病'列有两种取值,'吸烟'和'饮酒'列分别有 3 种取值,上例旨在检查数据中是否存在超出正常取值范围的异常情况。

(4) ID 列数据异常值筛选,代码如下:

```
In:
  mask = df['ID'].str.len() != 14
  # 筛选出符合条件的数据
  result = df[mask]
  result
Out:
name    ID  sex  民族    文化程度    身高    体重    收缩压  ...  危险分层    编号
    0 rows × 24 columns
```

上例中创建了一个布尔值数组 mask,用于筛选出 df_new 中 ID 列的值长度不等于 14 的行。

(5) 对身高、体重列的异常值进行检测并进行处理,代码如下:

```
In:
    def get_outliers_count(data, columns):
        """
        获取指定列的处理前异常值数量
        参数:
        - data (pd.DataFrame): 输入的 DataFrame
        - columns (list): 要处理的列名列表
        返回:
        - list: 每列异常值的数量
        """
        before_counts = []
        for column in columns:
            # 计算 IQR
            iqr = data[column].quantile(0.75) - data[column].quantile(0.25)
            # 找到下界和上界
            lower_bound = data[column].quantile(0.25) - 1.5 * iqr
```

```
            upper_bound = data[column].quantile(0.75) + 1.5 * iqr
            # 找到异常值
            outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
            before_counts.append(len(outliers))
        return before_counts
    def detect_and_replace_outliers(data, columns):
        """
        使用 IQR 方法检测并替换指定列的异常值
        参数:
        - data (pd.DataFrame): 输入的 DataFrame
        - columns (list): 要处理的列名列表
        返回:
        - pd.DataFrame: 替换异常值后的 DataFrame
        - dict: 处理前后每列异常值的数量
        """
        outliers_count_dict = {}
        for column in columns:
            # 计算 IQR
            iqr = data[column].quantile(0.75) - data[column].quantile(0.25)
            # 找到下界和上界
            lower_bound = data[column].quantile(0.25) - 1.5 * iqr
            upper_bound = data[column].quantile(0.75) + 1.5 * iqr
            # 找到异常值
            outliers = data[(data[column] < lower_bound) | (data[column] > upper_bound)]
            outliers_index = outliers.index
            outliers_count = len(outliers_index)
            # 替换异常值
            median_value = data[column].median()
            data.loc[outliers_index, column] = median_value
            # 更新异常值数量字典
            outliers_count_dict[column] = {
                'before': outliers_count,
                'after': 0      # 因为已经替换了异常值
            }
        return data, outliers_count_dict
    before_counts = get_outliers_count(df_new, ['身高', '体重'])
    processed_data, outliers_count_dict = detect_and_replace_outliers(df_new, ['身高', '体重'])
    after_counts = [outliers_count_dict[col]['after'] for col in ['身高', '体重']]
    print('处理前身高, 体重异常值数量',before_counts)
    print('处理后身高, 体重异常值数量',after_counts)
Out:
    处理前身高, 体重异常值数量 [23, 41]
    处理后身高, 体重异常值数量 [0, 0]
```

上述以身高、体重列为例，展示了如何对指定列的异常值进行检测和处理。其中，get_outliers_count 函数用于计算 DataFrame 中指定列处理前的异常值数量。它首先计算 IQR（四分位数范围），然后确定每列的正常值范围，最后统计超出这个范围的值的数量。detect_and_replace_outliers_optimized 函数使用 IQR 方法检测并替换数据框中指定列的异常值。它首先计算 IQR 和正常值范围，然后找出异常值并替换为中位数，最后输出处理前后身高、体重列的异常值情况。

以上综合案例通过对心血管数据进行缺失值的检测和处理、重复值的检测和处理及异常值的检测和处理等操作，实现了对心血管数据的清洗。清洗后的数据保存在 df_new 中，如有需要可以将其导出为 Excel 文件。

3.4 数据转换

数据转换是数据预处理的一个重要方面，它涵盖多种操作，如标准化、归一化和各种形式的数据转换，旨在提升数据的可解释性和增强模型的性能。数据转换能够使数据更加清洁、结构更加合理，从而为后续的数据分析和建模提供坚实的基础。

3.4.1 数据标准化

数据标准化(Standardization)是一种数学转换过程，它将原始数据转换为无量纲的相对数值，确保数据集具有零均值和单位方差。这一过程有助于消除数据间的量纲差异和不公平性，减少异常值对分析结果的负面影响，并增强机器学习算法对数据的拟合能力，进而提高数据分析的准确性。例如，在进行回归预测时，数据标准化的目的是确保各特征值具有均衡的权重；在执行主成分分析时，数据标准化可以促进模型更快地拟合数据，加快收敛速度。Z-Score 是常用的数据标准化方法。

Z-Score 标准化也称为标准差标准化，通过将数据转换为标准正态分布(均值为 0，标准差为 1)来进行标准化，计算公式为

$$Z=\frac{X-\mu}{\sigma} \tag{3-1}$$

其中，Z 表示标准化后的值，X 表示原始值，μ 表示原始值的均值，σ 表示原始值的标准差。

例 3-26 使用 Z-Score()方法进行数据标准化。

```
In:
    from sklearn.datasets import load_iris
    from sklearn.preprocessing import StandardScaler
    iris = load_iris()              # 加载鸢尾花数据集
    X = iris.data                   # 调取鸢尾花样本集
    zscore = StandardScaler()       # 创建一个标准化(Z-Score)对象
    zscore.fit(X)                   # 计算均值和方差
    X_std = zscore.transform(X)     # 标准化操作
    print(X_std)
Out:
    [[-9.00681170e-01  1.01900435e+00 -1.34022653e+00 -1.31544430e+00]
     [-1.14301691e+00 -1.31979479e-01 -1.34022653e+00 -1.31544430e+00]
     ......
     [ 6.86617933e-02 -1.31979479e-01  7.62758269e-01  7.90670654e-01]]
```

例 3-26 使用 sklearn 中的 StandardScaler()类创建了 zscore 对象，对加载的鸢尾花数据集实现数据标准化操作。标准化后的数据中每个特征值都转换为具有均值为 0、标准差为 1 的值。

3.4.2 数据归一化

数据归一化(Normalization)也称为向量归一化或长度归一化，其核心目的是将数据从不同的量纲映射到一个统一的区间内。这样做可以消除特征间的量纲差异，防止某些特征在模型训练过程中占据过大的影响力，进而促进模型快速收敛并提升整体性能。例如，当使用神经网络时，输入层的数据经常会被归一化，这有助于加速权重更新的过程，并且可以使梯度下降算法更加稳定。

在统计学和机器学习领域，"标准化"一词通常特指将数据转换成具有零均值和单位方差的形式，即 Z-Score 标准化。相对而言，"归一化"是一个更为宽泛的概念，它涵盖将数据缩放

到特定范围或分布的各种方法，例如，将数据限制在[0，1]区间(最小-最大归一化)或确保数据向量的长度为1(L1或L2归一化)。尽管这两个术语有时可以互换使用，但具体指代哪种技术还需依据上下文来明确。数据标准化主要针对连续变量，通过调整使其平均值为0、标准差为1，特别适用于数值范围广泛的特征。数据归一化则将连续变量压缩至[0，1]区间，适用于那些数值本身较为集中，但需要进一步统一到特定区间的特征。标准化的重点在于调整数据的分布，使其更接近正态分布，并减少异常值的影响。归一化侧重于按比例缩放，确保数据落在预定范围内，但这一过程可能会受到异常值的影响。

1. L1 和 L2 归一化

L1 和 L2 归一化是常用的归一化方法。L1 归一化将向量中的每个元素除以向量元素的绝对值之和，具体计算公式如下：

$$Z=\frac{X}{\sum |X|} \tag{3-2}$$

其中，Z 表示归一化后的向量，X 表示原始向量，Σ表示求和操作。L2 归一化将向量中的每个元素除以向量的欧几里得范数(即向量元素平方和的平方根)，具体计算公式如下：

$$Z=\frac{X}{\|X\|} \tag{3-3}$$

其中，Z 表示归一化后的向量，X 表示原始向量，$\|X\|$ 表示向量 X 的欧几里得范数。

2. Min-Max 归一化

Min-Max 的原理是通过最小值和最大值来缩放数据，该方法通过线性变换将数据映射到[0,1]区间，计算公式为

$$Z=\frac{X-X_{\min}}{X_{\max}-X_{\min}} \tag{3-4}$$

其中，Z 表示原始数据经过数据归一化后的值，X 表示原始值，$X_{\min}$ 表示原始值的最小值，$X_{\max}$ 表示原始值的最大值。

3. 小数定标归一化

小数定标归一化通过移动数据的小数点位置进行归一化，将原始数据映射在[−1,1]区间，计算公式为

$$Z=\frac{X}{10^{k}} \tag{3-5}$$

其中，Z 表示原始数据经过数据归一化后的值；X 表示原始值；k 为超参数，一般设置为大于或等于原始值的绝对值的最小整数。

例 3-27 数据归一化实例。

```
In:
    from sklearn.preprocessing import MinMaxScaler
    # 创建示例数据
    data = {
        '患者 ID': [1, 2, 3, 4, 5, 6],
        '性别': ['男', '女', '男', '女', '男', '女'],
        '年龄': [55, 60, 45, 50, 58, 62],
        '血压': [140, 120, 130, 150, 135, 145],
        '胆固醇': [220, 280, 200, 260, 210, 270]
```

```
    }
    df = pd.DataFrame(data)
    # 显示原始数据
    print("原始数据:")
    print(df)
    # 选择需要归一化的数值列
    numeric_columns = ['年龄', '血压', '胆固醇']
    # 初始化 MinMaxScaler
    scaler = MinMaxScaler()
    # 对选定的数值列进行归一化
    df[numeric_columns] = scaler.fit_transform(df[numeric_columns])
    # 显示归一化后的数据
    print("\n归一化后的数据:")
    print(df)
    # 保存处理后的数据到新的 CSV 文件
    output_file_path = 'normalized_patient_data.csv'
    df.to_csv(output_file_path, index=False)
    print(f"处理后的数据已保存到 {output_file_path}")
```

例 3-27 中,首先使用 Pandas 库读取并处理了一个包含患者年龄、血压和胆固醇水平的医疗数据集。然后,利用 sklearn 库中的 MinMaxScaler 对这些数值特征进行归一化,将它们的值缩放到[0,1],从而确保不同尺度的特征在机器学习模型中具有相同的权重。最终,归一化后的数据被保存到一个新的 CSV 文件中。

3.4.3 数据编码

在机器学习的数据预处理阶段,为了使算法能够处理非数值型数据,通常需要进行映射变换,将这类数据转换为数值形式,这一转换过程称为数据编码。常见的数据编码技术包括标签编码、独热编码、序号编码、频数编码、目标编码等。

1. 标签编码

标签编码(Label Encoding)是一种将分类数据转换为整数的方法,通常从 0 开始递增。例如,可以将['beijing','hangzhou','guangzhou']编码为[1,2,3]。

2. 独热编码

独热编码(One-Hot Encoding)通过为每个离散属性的每个类别创建一个新的二进制特征来实现数据的编码。在这些特征向量中,对于每个样本,只有一个特征对应的值为 1,表示该样本属于该类别,而其他所有特征值均为 0。例如,可以将['beijing','hangzhou','guangzhou']中的'beijing'编码为[1,0,0],'hangzhou'编码为[0,1,0],'guangzhou'编码为[0,0,1]。

3. 序号编码

序号编码(Ordinal Encoding)是一种将离散特征的各个类别映射为整数序号的方法。序号编码适用于有序特征,其中类别之间存在一定的顺序关系,但没有明确的意义。例如,可以将['高中','本科','硕士','博士']编码为{'高中':0,'本科':1,'硕士':2,'博士':3}。

序号编码和标签编码都是用整数替代类别标签,区别在于序号编码保留了类别的顺序,适用于有序数据(如"低""中""高"),序号编码可以反映这种顺序。而标签编码则用于无序数据(如"红色""蓝色""绿色"),仅区分不同类别而不表示顺序。

4. 频数编码

频数编码(Frequency Encoding)将每个类别替换为该类别在数据集中的频数或出现次数。这种编码方法可以提供关于类别的频率信息,但可能引入一定的信息误差。例如,['beijing','hangzhou','guangzhou','beijing']被编码为{'beijing':2,'hangzhou':1,'guangzhou':1}。

5. 目标编码

目标编码(Target Encoding)将离散属性的每个类别编码为其在目标变量上的平均值或其他统计信息。该方法能够捕获类别与目标变量之间的关联性。例如,原始数据 $X=[X_1,X_2]$,其中 X_1='beijing','beijing','beijing','hangzhou','hangzhou',$X_2=9,8,10,9,7$。X 经过目标编码后的值为 $\hat{X}$={'beijing':9,'hangzhou':8},其中,'beijing'的运算过程为 $(9+8+10)/3=9$,'hangzhou'的运算过程为 $(9+7)/2=8$。

例 3-28 标签编码实例。

```
In:
    from sklearn.preprocessing import LabelEncoder
    le = LabelEncoder()
    city_list = ["Beijing", "Nanjing", "Nantong", "Beijing"]
    le.fit(city_list)
    city_list_le = le.transform(city_list)               # 进行编码
    city_list_new = le.inverse_transform(city_list_le)   # 进行编码
    print(f'原始数据为:{city_list}')
    print(f'list 中含有的类别有:{le.classes_}')
    print(f'编码后:{city_list_le}')
    print(f'解码后:{city_list_new}')
Out:
    原始数据为:['Beijing', 'Nanjing', 'Nantong', 'Beijing']
    list 中含有的类别有:['Beijing' 'Nanjing' 'Nantong']
    编码后:[0 1 2 0]
    解码后:['Beijing' 'Nanjing' 'Nantong' 'Beijing']
```

例 3-28 使用 sklearn.preprocessing.LabelEncoder 实现标签编码,将'Beijing'、'Nanjing'、'Nantong'分别编码为 0,1,2。

例 3-29 序号编码实例。

```
In:
    import pandas as pd
    from sklearn.preprocessing import OrdinalEncoder
    # 创建一个包含分类数据的简单 DataFrame
    data = {
        'Color': ['red', 'blue', 'green'],
        'Size': ['S', 'M', 'L']
    }
    df = pd.DataFrame(data)
    # 初始化 OrdinalEncoder
    encoder = OrdinalEncoder()
    # 对整个 DataFrame 进行编码
    df_encoded = encoder.fit_transform(df)
    # 将编码后的结果转换回 DataFrame 格式
    df_encoded = pd.DataFrame(df_encoded, columns = df.columns)
    # 查看每个类别的对应编码值
    for i, col in enumerate(df.columns):
        print(f"\n{col} 类别及其对应的编码值:")
```

```
        for j, category in enumerate(encoder.categories_[i]):
            print(f"{category}: {j}")
Out:
    Color 类别及其对应的编码值:
    blue: 0
    green: 1
    red: 2
    Size 类别及其对应的编码值:
    L: 0
    M: 1
    S: 2
```

例 3-29 首先创建了一个包含两列分类数据(Color 和 Size)的 DataFrame。然后,使用 OrdinalEncoder 对这两列进行序号编码,并将结果存储在一个新的 DataFrame 中。

例 3-30　独热编码实例。

```
In:
    from sklearn.preprocessing import OneHotEncoder
    # 创建一个包含类别数据的数据集
    data = {
        '颜色': ['红色', '绿色', '蓝色', '绿色', '红色']
    }
    # 将数据转换为 DataFrame
    df = pd.DataFrame(data)
    # 初始化 OneHotEncoder
    encoder = OneHotEncoder(sparse = False)  # sparse = False 表示输出为密集矩阵
    # 对颜色列进行独热编码
    encoded_data = encoder.fit_transform(df[['颜色']])
    # 将编码结果转换为 DataFrame,并设置列名
    encoded_df = pd.DataFrame(encoded_data, columns = encoder.get_feature_names_out(['颜色']))
    # 将编码结果与原始数据合并
    result_df = pd.concat([df, encoded_df], axis = 1)
    # 打印结果
    print(result_df)
Out:
       颜色  颜色_红色  颜色_绿色  颜色_蓝色
    0  红色     1.0       0.0       0.0
    1  绿色     0.0       1.0       0.0
    2  蓝色     0.0       0.0       1.0
    3  绿色     0.0       1.0       0.0
    4  红色     1.0       0.0       0.0
```

例 3-30 中首先创建了一个包含不同颜色名称的数据集。然后,利用 OneHotEncoder 对这些颜色名称进行独热编码,每个颜色都转换成了一个二进制向量。例如,"蓝色""绿色"和"红色"分别对应独热编码列"颜色_蓝色""颜色_绿色"和"颜色_红色"。每行对应一个数据点,其中的 1 表示该数据点属于相应的颜色类别。

例 3-31　频数编码实例。

```
In:
    # 创建一个包含分类数据的简单 DataFrame
    data = [['苹果'], ['香蕉'], ['苹果'], ['橙子'], ['香蕉'], ['苹果']]
    df = pd.DataFrame(data, columns = ['水果'])
    # 使用 value_counts()来获取每个类别的频数
    frequency = df['水果'].value_counts()
    # 创建一个字典,将每个类别映射到其频数
    frequency_dict = frequency.to_dict()
    # 应用频数编码
```

```
    df['频数编码'] = df['水果'].map(frequency_dict)
    df
Out:
         水果    频数编码
    0    苹果    3
    1    香蕉    2
    2    苹果    3
    3    橙子    1
    4    香蕉    2
    5    苹果    3
```

例 3-31 展示了如何使用 Pandas 对分类数据进行频数编码，通过将每个类别映射到其在数据集中出现的频数。

3.4.4 案例——心脑血管数据转换

本节主要对心脑血管疾病高危人群数据的各项指标(如收缩压、舒张压、总胆固醇等)进行数据标准化、数据归一化处理。由于数据编码旨在将非数值型数据转换为数值型数据，心血管数据中的数据除姓名外不包含非数值型数据，因此，省去数据编码对心脑血管数据进行处理的实验。

例 3-32 数据标准化。

```
In:
    from sklearn.preprocessing import StandardScaler
    # 文件路径
    file_path = r'data/T2_HealthInfo.xlsx'
    # 需要选择的列名称
    columns_to_select = ['收缩压', '舒张压', '总胆固醇', '甘油三酯', '高密度脂蛋白胆固醇', '低
密度脂蛋白胆固醇', '空腹血糖']
    # 读取 Excel 文件并选择特定列
    df = pd.read_excel(file_path, usecols = columns_to_select)
    # 查看含有'#NULL!'的行的数量
    print(df.isin(['#NULL!']).count())
    # 将'#NULL!'替换为 np.nan
    df = df.replace('#NULL!', np.nan)
    # 处理缺失值,这里可以选择删除含有缺失值的行或填充缺失值
    #可以删除含有缺失值的行
    # df.dropna(inplace = True)

    # 本例采用填充缺失值,用每列的平均值填充
    df.fillna(df.mean(), inplace = True)
    # 标准化数据
    scaler = StandardScaler()
    scaled_data = scaler.fit_transform(df)
    # 将标准化后的数据转换为 DataFrame
    df_scaled = pd.DataFrame(scaled_data, columns = df.columns)
    # 显示标准化后的 DataFrame
    display(df_scaled)
```

例 3-32 首先读取心脑血管疾病高危人群原始数据中的部分数值型列，并对其中的缺失值进行处理，然后调用 sklearn 中的 StandardScaler 对数据进行标准化操作，最后显示标准化处理后的数据，部分结果如图 3-7 所示。

例 3-33 数据归一化。

```
In:
    from sklearn.preprocessing import MinMaxScaler
```

```
#归一化数据
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df)
df_scaled = pd.DataFrame(scaled_data, columns = df.columns)
# 显示归一化后的 DataFrame
display(df_scaled)
```

	收缩压	舒张压	总胆固醇	甘油三酯	高密度脂蛋白胆固醇	低密度脂蛋白胆固醇	空腹血糖
0	2.468573	0.504238	-3.794389	2.644453	2.686660	-2.074804	-0.772485
1	0.040502	0.047304	-3.664802	2.921614	0.411218	0.287269	0.430983
2	-0.566515	-0.135470	-3.612968	2.757049	0.220462	-0.802214	-0.717782
3	-0.566515	-0.226857	-3.587050	3.146807	1.869136	-1.617038	-0.772485
4	0.261236	0.778398	-3.569772	2.618469	-0.270053	4.224057	0.184819
...	...	...	...	...	...	...	...

图 3-7　心脑血管数据标准化实例结果

部分输出结果如图 3-8 所示。

	收缩压	舒张压	总胆固醇	甘油三酯	高密度脂蛋白胆固醇	低密度脂蛋白胆固醇	空腹血糖
0	0.673077	0.365854	0.000000	0.223896	0.519820	0.076667	0.105100
1	0.391026	0.325203	0.009791	0.235109	0.369369	0.291667	0.183149
2	0.320513	0.308943	0.013708	0.228451	0.356757	0.192500	0.108647
3	0.320513	0.300813	0.015666	0.244219	0.465766	0.118333	0.105100
4	0.416667	0.390244	0.016971	0.222845	0.324324	0.650000	0.167184
...	...	...	...	...	...	...	...

图 3-8　心脑血管数据归一化实例结果

例 3-33 首先读取心脑血管疾病高危人群原始数据中的部分数值型列，并对其中的缺失值进行处理（方法与例 3-32 相同，本例此处省略），然后调用 sklearn 中的 MinMaxScaler 对数据进行归一化操作，最后显示归一化处理后的数据。

以上综合案例通过对心血管数据进行数据标准化和数据归一化操作，实现了数据量纲的统一。这样做可以使机器学习模型更容易拟合这些数据，从而提升数据分析的效果。在实际应用中，选择具体的数据转换方法时，应根据具体任务的需求进行分析和决策。

3.5　数据脱敏与隐私保护

3.5.1　数据脱敏与隐私保护概述

在大数据时代，数据开放共享是挖掘数据价值的重要基础。各类大数据中包含个人隐私等敏感数据，一旦在数据使用过程中发生敏感信息泄露的问题，将会对国家、社会和个人带来严重损失。因此，在保障数据供给的同时进行数据脱敏是实现数据安全开放共享的重要举措。

数据脱敏又称数据漂白、数据去隐私化或数据变形，是指对某些敏感信息（如身份证号、电话、日期、密码等）进行数据变形，以防止未经授权的访问者获取到真实敏感数据，从而实现数据保护。数据脱敏的目的是在进行数据开放共享的同时，在保持数据可用性的前提下防止隐私数据泄露和滥用，保证数据安全。数据脱敏在医疗、金融等领域应用较早，这类数据中通常包含大量个人隐私信息，一旦泄露将会对个人带来不利影响。如今数据脱敏在日常生活中也比较常见，如在进行网购时，平台会将快递单中的收件人手机号码用星号（*）进行部分遮挡，以免不法分子窃取个人信息，从而保护了用户隐私安全。

数据脱敏的方法是按照特定的脱敏规则对敏感数据进行处理，如数据替换、无效化、乱序、数据遮挡、数据加密等。在进行数据脱敏后，真实数据得以隐藏或修改，只允许相关授权人员通过一定的工具对原数据进行访问。在保证不影响数据使用的情况下，数据脱敏实现了隐私保护，扩展了原有数据的可使用范围，有利于大数据的建设和发展。

3.5.2 数据脱敏原则

大数据时代，数据脱敏是保障数据安全的重要方式。数据脱敏如果只是做到抹去敏感的数据内容，那么处理数据的各个环节将会变得杂乱无章。因此，在抹去数据敏感内容的同时，通常需要遵守以下原则，以保证处理数据各个环节的协调统一。

（1）保证脱敏后的数据具有原来的数据特征。数据脱敏是将敏感数据变成不敏感数据，而不是改变其脱敏后的属性及特征。脱敏后的数据应保持原有数据的特征分布，以适用于目标研究领域。

（2）遵循最小化原则。只脱敏必要的数据，从而尽可能地减少对数据完整性和可用性的影响。

（3）遵循不可逆原则。通常情况下，为了防止数据信息相关人员通过脱敏后的数据还原出原始敏感数据，数据脱敏应当遵循不可逆原则，即无法通过脱敏后的数据还原或者推断出任何原始敏感信息，从而保证数据的安全性。当然，在一些特定的或者特殊的使用情况下，也同样存在可恢复式数据信息的复敏需求。

（4）对其他所有已知的有可能生成敏感数据信息的非敏感字段同样进行脱敏处理。遵循这项原则可以防止相关人员根据可看到的数据或者信息推断出敏感内容，从而引发不良行为。例如，在某种情况下，家庭住址为非敏感数据，但若可以根据家庭住址进一步推断出个人及其他身份信息，那么家庭住址就是可以生成敏感数据的非敏感数据，因此需要被完全隐藏。

（5）脱敏过程应是可自动化可无限重复的。大量的数据千变万化，且由于数据的自动生成速度极快，脱敏过程必须在系统规定的条件下自动连续进行，这样系统才会提高数据处理的效率。同时数据脱敏也应该具有可重复性，以满足数据的可测性、正确性、安全性等指标。

（6）保持数据的关联性。数据与数据之间在不同情况下具有一定的关联，因此，在数据脱敏计算时，需要确保经过脱敏操作后得到的数据仍保持与其原始数据之间具有相关部分的高度关联性，保证后续数据处理在实际活动中产生有效的结果。对于全结构化信息和半结构化数据，如果某一个字段数据与另外一个字段信息有对应关系，而脱敏算法破坏了这种映射关系，那么该字段的使用价值将基本不复存在。因此，在数据脱敏的同时需要保持数据的关联性。

（7）保持数据的一致性。如果数据需要多次处理或者需要在不同的环境下进行脱敏，那么需要保证多次脱敏数据之间的一致性，从而在数据的不断变更及相关领域不断发展的条件下，保证数据的持续一致性。

遵循以上原则可以进行有效的数据脱敏，既保持了原有的数据结构特点和数据关联性，又保证大数据研究的各个环节及大数据相关业务不受数据脱敏的影响。在进行数据脱敏时，需要根据具体情况采用合适的原则来保护数据的安全和隐私，降低数据泄露或被盗用、滥用所带来的风险，从而可以更放心地将数据公开呈现在大众面前，增加数据分享和共享的可行性而不危及数据的安全和隐私，在各种测试、研究和合作等场景中使用并进行深入研究和探索，寻找数据的无限可能。

3.5.3 数据脱敏方法

1. 数据替换

数据替换是将数据集中的敏感信息替换为模拟值、模糊值或其他逻辑合理的替代值，以实现隐私保护的脱敏方法。

例 3-34 数据替换实例。

```
In:
    # 创建一个包含姓名和电话号码的 DataFrame
    data = {
        '姓名': ['张三', '李四', '王五'],
        '电话号码': ['13812345678', '13987654321', '13612349876']
    }
    df = pd.DataFrame(data)
    # 数据替换
    # 用固定的匿名名称替换姓名
    df['姓名'] = '匿名'
    # 用固定的匿名电话号码替换电话号码
    df['电话号码'] = '12345678901'
    # 打印替换后的数据
    print("替换后的数据:")
    print(df)
Out:
        姓名    电话号码
    0   匿名  12345678901
    1   匿名  12345678901
    2   匿名  12345678901
```

例 3-34 对一组包含个人姓名和电话号码的数据进行了脱敏处理，具体做法是通过简单的替换操作，姓名和电话号码等敏感信息都被统一替换为“匿名”和“12345678901”。通过数据替换有效地隐藏了数据中的敏感信息。

2. 随机化

随机化是一种通过使用随机数据来代替真值以改变敏感数据的方法，这样可以保持替换值的随机性以模拟样本的真实性。在随机化过程中，字母会被替换为随机字母，数字会被替换为随机数字，而文字可能会被随机替换成其他文字，这种方式改变了敏感数据，且用户不易察觉。

例 3-35 随机化实例。

```
In:
    import random
    import string
    # 生成随机字符串
    def generate_random_string(length):
        return ''.join(random.choice(string.ascii_letters + string.digits) for _ in range(length))
    # 生成随机姓名
    def generate_random_name():
        first_names = ["张", "李", "王", "赵"]
        last_names = ["伟", "芳", "娜", "杰"]
        return random.choice(first_names) + random.choice(last_names)
    # 生成随机电子邮件地址
    def generate_random_email():
```

```
        username = generate_random_string(8)
        domain = random.choice(["example.com", "test.com"])
        return f"{username}@{domain}"
    # 生成随机电话号码
    def generate_random_phone():
        return f"({random.randint(100, 999)}) {random.randint(100, 999)} - {random.randint(1000,
9999)}"
    # 脱敏函数
    def mask_data(data):
        masked_data = []
        for record in data:
            masked_record = {
                "name": generate_random_name(),
                "email": generate_random_email(),
                "phone": generate_random_phone()
            }
            masked_data.append(masked_record)
        return masked_data
    # 示例数据
    data = [
        {"name": "张三", "email": "zhangsan@example.com", "phone": "(123) 456 - 7890"},
        {"name": "李四", "email": "lisi@example.com", "phone": "123 - 456 - 7890"},
        {"name": "王五", "email": "wangwu@example.com", "phone": "1234567890"}
    ]
    # 脱敏处理
    masked_data = mask_data(data)
    # 打印脱敏后的数据
    print("脱敏后的数据:")
    for record in masked_data:
        print(record)
Out:
    脱敏后的数据:
    {'name': '赵杰', 'email': '8th2q1YK@example.com', 'phone': '(663) 115 - 8529'}
    {'name': '赵芳', 'email': 'Fdmuumvl@example.com', 'phone': '(400) 596 - 4096'}
    {'name': '张杰', 'email': 'teHjyLQz@example.com', 'phone': '(517) 911 - 7330'}
```

在例 3-35 中，通过生成随机的姓名、电子邮件地址和电话号码来替换原始数据中的敏感信息，并分别输出了结果。这样可以保护真实数据不被泄露，同时保留了原数据的结构和格式。

3. 数据遮挡

数据遮挡是一种常见的隐私保护技术，它通过只显示部分信息，将敏感数据的一部分进行遮挡，以防止未经授权的人员获取完整的敏感信息，确保隐私信息不被轻易泄露。数据遮挡通常用于处理账户类数据，如银行卡号、身份证号码或手机号码。

与数据替换不同，数据遮挡通常会保留数据的部分结构和格式，但掩盖具体的敏感信息。

例 3-36 数据遮挡实例。

```
In:
    def mask_id(id_number):
        masked_id = id_number[:-4] + '****'
    #函数通过切片操作将身份证号码的前面部分保留，而将后面 4 位替换为****
        return masked_id
    id_number = '110112200404280318'
    masked_id_number = mask_id(id_number)
    print(masked_id_number)
```

```
Out:
    11011220040428****
```

在例 3-36 中，通过对用户的身份证号码最后 4 位进行遮挡，只显示身份证号码前面部分的数字，从而保护了用户的隐私数据。

4. 偏移和取整

偏移和取整通过随机移位或取整改变数字数据，使得数据无法被还原和识别，在保持了数据安全性的同时保证了范围的大致真实性。例如，将数据列中的数值进行随机的加减操作，使得数据的总体趋势不变，但具体数值发生了变化。

例 3-37 数据偏移实例。

```
In:
    from datetime import datetime, timedelta
    import random
    # 原始日期列表
    date = ['2024-01-01', '2023-03-16']
    # 创建一个新列表来存储调整后的日期
    date_new = []
    # 将日期字符串转换为 NumPy 数组
    arr = np.array(date)
    # 遍历数组中的每个日期
    for i in arr:
        # 将日期字符串转换为 datetime 对象
        y = datetime.strptime(i, '%Y-%m-%d')
        # 生成 24～240 的随机数,步长为 24
        random_hours = random.choice(range(24, 241, 24))
        # 将随机小时数加到原始日期上
        s = y + timedelta(hours=random_hours)
        # 将调整后的日期转换回字符串格式并添加到新列表中
        date_new.append(s.strftime('%Y-%m-%d'))
    # 打印原始日期和调整后的日期
    print("原始日期:", date)
    print("调整后的日期:", date_new)
Out:
    原始日期: ['2024-01-01', '2023-03-16']
    调整后的日期: ['2024-01-06', '2023-03-23']
```

在例 3-37 中，对原始日期列表中的敏感数据进行了随机偏移处理，对原日期数据增加 24～240 小时的一个随机数(步长为 24)，并返回新日期列表，在保持日期大致真实性情况下提高了数据安全性。

5. 加密处理

数据加密是指通过加密算法和密钥将明文转变为密文的数据脱敏方法。例如，对银行账号、身份证号等敏感数据使用加密算法进行脱敏，只有经过授权的人员才能够解密，使个人信息不易被窃取，提高数据安全性。数据加密技术包括对称加密(如 AES、DES)、非对称加密(如 RSA)、同态加密等。

例 3-38 加密处理实例。

```
In:
    from pyDes import des, CBC, PAD_PKCS5
    import binascii
```

```
    key = '12345678'
    data = '20250101'
    # 确保 key 和 data 都是字节串
    key = bytes(key, 'utf-8')
    data = bytes(data, 'utf-8')
    # 初始化向量 iv,这里使用一个简单的 8 B 的 iv
    # 在实际应用中,iv 应该是随机的且不可预测的
    iv = bytes([0] * 8)
    def des_encrypt(secret_key, s, iv):
        k = des(secret_key, CBC, iv, padmode=PAD_PKCS5)
        en = k.encrypt(s)
        return binascii.b2a_hex(en)
    def des_decrypt(secret_key, s, iv):
        k = des(secret_key, CBC, iv, padmode=PAD_PKCS5)
        de = k.decrypt(binascii.a2b_hex(s))
        return de.decode('utf-8')              # 解码为字符串
    # 加密
    secret_str = des_encrypt(key, data, iv)
    print(secret_str)
    # 解密
    clear_str = des_decrypt(key, secret_str, iv)
    print(clear_str)
Out:
    b'8464ea9b81584dfe31d0d4d431e12086'
    20250101
```

在例 3-38 中,使用 DES 对称加密算法对敏感数据进行了一次加密和解密处理。首先,通过调用 des_encrypt()加密函数,并设置密钥“12345678”对敏感信息“20250101”进行加密处理,得到密文“8464ea9b81584dfe31d0d4d431e12086”。其次,使用 des_decrypt()解密函数和相同的密钥“12345678”对密文解密,得到明文(即原敏感信息)“20250101”。

3.5.4 案例——心脑血管数据脱敏

1. 数据替换

例 3-39 数据替换。

```
In:
    # 读取 Excel 表格数据
    df = pd.read_excel("data/T2_HealthInfo.xlsx")
    column_name = "name"                    # 姓名列的列名
    def desensitize_name(name):             # 数据脱敏函数
        name_length = len(name)
        if name_length >= 2:
            return name[0] + '*'            # 脱敏长度大于 2 的姓名,隐藏后面所有字
        else:
                return name                 # 对姓名列进行数据脱敏
    # 对姓名列进行数据脱敏
    df[column_name] = df[column_name].apply(desensitize_name)
    # ID 列的列名
    column_name = "ID"
    # 数据脱敏函数
    def mask_id(id):
      # 不脱敏长度小于 9 的 ID
        if len(id) >= 9:
                return id[:3] + '*' * (len(id) - 6) + id[-3:]   # 隐藏中间 3~8 位数字
        else:
```

```
            return id
    # 对 ID 列进行数据脱敏
    df[column_name] = df[column_name].apply(mask_id)
    # 将修改后的数据保存到新的 Excel 文件中
    display(df.head())
```

表 3-7 所示为原数据。表 3-8 所示为掩码屏蔽结果。

表 3-7 原数据

编 号	name	ID
1	menghui	19521118018852
2	gejiachen	19630320015172
3	weishi	19770103011135
4	bianning	19740715011858
5	pangyuke	19610305012216
6	qiyi	19670429012400
7	maoyuan	19391121018195
8	fengbao	19681029017591
9	huamao	19680330019489
10	kemaliang	1943102901393X

表 3-8 掩码屏蔽结果

编 号	name	ID
1	m ******	195 ******** 852
2	g ********	196 ******** 172
3	w *****	197 ******** 135
4	b *******	197 ******** 858
5	p *******	196 ******** 216
6	q ***	1968 ******* 400
7	m ******	193 ******** 195
8	f ******	196 ******** 591
9	h *****	196 ******** 489
10	k ********	194 ******** 93X

例 3-39 中对姓名和 ID 的敏感部分进行了掩码屏蔽，降低了个人身份信息被泄露的风险，使得数据安全性增强。通过掩码屏蔽敏感信息，可以减少未经授权的访问或不当使用敏感数据的可能性，有助于保护数据的安全性和完整性。

2. 随机化

例 3-40 随机化实例。

```
In:
    import random
    import string
    # 读取 Excel 文件
    file_path = r'data/T2_HealthInfo.xlsx'
    df = pd.read_excel(file_path)
    # 定义用于随机化的字符集
    characters = string.ascii_letters
    # 定义随机化"姓名"列的函数
```

```
def randomize_name(name):
    return ''.join(random.choice(characters) for _ in range(len(name)))
# 将随机化函数应用于"姓名"列
df['name'] = df['name'].apply(randomize_name)
# 输出处理后的数据
display(df.name.head())
```

原数据与随机化结果如表 3-9 所示。

表 3-9 原数据与随机化结果

编 号	name(原数据)	name(随机化结果)
1	menghui	SBqoQAf
2	gejiachen	JIihzkJOp
3	weishi	tXtaAj
4	bianning	EWfSJNMh
5	pangyuke	fyYsRrxC

例 3-40 使用了 Python 的 random 库和 string 库来生成随机字母。首先,定义了一个函数 randomize_name,该函数将名字替换为随机生成的字母。然后,将这个函数应用于 name 列,将每个人的名字替换为随机字符串,从而保护隐私。

3. 数据遮挡

例 3-41 数据遮挡实例。

```
In:
  def mask_last_four(digits):
      if len(digits) >= 4:
          masked_digits = digits[:-4] + '****'
      else:
          # 如果长度小于 4,可以选择填充 X 或其他字符,或者返回原始 digits
          # 这里选择填充 X 直到长度为 4,然后进行遮蔽
          masked_digits = digits + 'X' * (4 - len(digits)) + '****'
      return masked_digits

  # 读取 Excel 文件
  df = pd.read_excel('data/T1_BasicInfo.xlsx')

  # 假设 ID 列是字符串类型,并且想要遮蔽最后 4 位
  # 如果 ID 列不是字符串类型,可能需要先将其转换为字符串
  df['ID'] = df['ID'].astype(str).apply(mask_last_four)

  # 保存修改后的 Excel 文件
  df.to_excel('data/数据遮挡.xlsx', index=False)
```

原数据与数据遮挡结果如表 3-10 所示。

表 3-10 原数据与数据遮挡结果

编 号	ID(原数据)	ID(数据遮挡结果)
1	19521118018852	1952111801****
2	19630320015172	1963032001****
3	19770103011135	1977010301****
4	19740715011858	1974071501****
5	19610305012216	1961030501****

在例 3-41 中,只有前面部分的信息被显示,后面 4 位被遮挡起来,实现了敏感信息保护,用户的隐私安全得到提升。

4. 偏移和取整

例 3-42 偏移和取整实例。

```
In:
   from datetime import datetime,timedelta
   data = np.genfromtxt("data/T2_HealthInfo.xlsx",dtype = 'str', delimiter = ',',skip_header = (1),
usecols = ( - 1))
   date_list = []
   data

   for i in data:
       date = i[:8]
       rest = i[8:]
       y = datetime.strptime(date, '%Y%m%d')
       s = y + timedelta(hours = 48)
       nd = s.strftime('%Y%m%d') + rest
       date_list.append(nd)
   date_list
```

数据偏移结果如表 3-11 所示。

表 3-11 数据偏移结果

编号	ID	New ID
1	19521118018852	19521120018852
2	19630320015172	19630322015172
3	19770103011135	19770105011135
4	19740715011858	19740717011858
5	19610305012216	19610307012216

例 3-42 中,对心脑血管疾病数据集中样本 ID 进行了脱敏。ID 前 8 位是患者的出生日期,属于敏感信息,需要进行脱敏处理。通过数据偏移,将出生日期进行了"加两天"的处理,例如,将第 1 行中的"19521118018852"变为"19521120018852",较原日期增加了 2 天,在保证数据大致真实性情况下提高了数据安全性。

5. 加密处理

例 3-43 加密处理实例。

```
In:
   from Crypto.Cipher import DES
   import binascii
   # 读取 Excel 文件
   file_path = "data/T2_HealthInfo.xlsx"
   df = pd.read_excel(file_path)
   # 获取需要加密的列的数据
   data = df['ID'].astype(str).values
   # 设置加密解密函数
   def pad(text):
       while len(text) % 8 != 0:
           text += ''
       return text
```

```
def des_encrypt(secret_key, s):
    cipher = DES.new(secret_key.encode('utf-8'), DES.MODE_CBC, secret_key.encode('utf-8'))
    padded_text = pad(s)
    encrypted_text = cipher.encrypt(padded_text.encode('utf-8'))
    return binascii.b2a_hex(encrypted_text).decode('utf-8')
def des_decrypt(secret_key, s):
    cipher = DES.new(secret_key.encode('utf-8'), DES.MODE_CBC, secret_key.encode('utf-8'))
    encrypted_text = binascii.a2b_hex(s)
    decrypted_text = cipher.decrypt(encrypted_text).decode('utf-8')
    return decrypted_text.strip()
# 设置密钥
key = '12345678'
# 对敏感数据加密
secret_data = [des_encrypt(key, i) for i in data]
# 对密文解密
clear_data = [des_decrypt(key, i) for i in secret_data]
# 创建新的 DataFrame 以显示 ID 和密文
result_df = pd.DataFrame({'ID': data, '密文': secret_data})
# 将结果保存到 Excel 文件
result_df.to_excel("data/encrypted_data.xlsx", index=False)
# 显示前几行加密和解密后的数据(用于验证)
clear_data = [des_decrypt(key, i) for i in secret_data]
print("加密数据: ", secret_data[:5])
print("解密数据: ", clear_data[:5])
```

数据加密结果如表 3-12 所示。

表 3-12 数据加密结果

ID	密 文
195 ******** 852	76c3f081fae3eb2cef419897ee5bbf1e
196 ******** 172	fa517f1837210f09233ba0615e810820
197 ******** 135	76280b667684ea5e611fd71119b471bc
197 ******** 858	76280b667684ea5e909adca0c1f2f3e5
196 ******** 216	fa517f1837210f0968933eb1272177ec

例 3-43 中,使用 DES 对称加密算法对敏感数据 ID 进行了一次加、解密处理。首先,调用 des_encrypt()加密函数并设置密钥“12345678”对 ID 值进行加密处理;其次,使用 des_decrypt()解密函数和相同的密钥对密文解密,得到原明文。加密后,原 ID 内容不可识别,个人信息得到了有效保护。

3.6 数据变换

数据变换旨在解决原始数据中可能存在的不一致性、冗余和噪声等问题。通过数据清洗、转换和规范化等手段,可以显著提升数据的一致性和准确性。本节将重点介绍线性空间变换及域空间变换技术。

3.6.1 数据变换概述

数据变换是数据预处理中的一个重要环节,它通过对数据进行数学操作来改善数据的质量和适用性,目标是将原始数据转换为更适合分析或模型训练的形式。这一过程能够消除数据中的不一致性、重复性和噪声等,从而增强数据的一致性和准确性。在数据变换中,线性空间变换和域空间变换是两种主要方法。

在探讨线性代数时,线性空间构成了对某些对象集合在数量特征上的抽象,而线性变换则揭示了这些空间内元素间的根本线性关系。线性变换维护了向量的线性组合与标量乘法,确保在保持向量间关系和线性属性的同时,对数据进行高效的转换。这种变换通常借助矩阵运算来完成,涉及旋转、缩放和剪切等多种操作。通过这种方式,线性变换能够在特征提取步骤中发挥关键作用,特别是在从数据中挖掘关键信息时,如提取图像的边缘和纹理特征,这一点尤为重要。与线性空间的概念相对,域空间是由一系列定义了加、减、乘、除运算的元素组成的集合,如实数和复数域。在域空间的框架下,数据变换可以通过非线性方法完成,即利用非线性函数将数据从一个域映射到另一个域。非线性变换使得数据的转换和特征提取更为灵活,从而更有效地匹配数据的复杂结构和关联。在应用线性与域空间变换的过程中,Python 提供了众多高效能的库和工具,大幅便利了编程和数据分析中的变换实现。

对于线性空间变换,数据变换主要包括如下 9 种。

(1) 求逆矩阵:在 Python 中,可以使用 NumPy 库的 inv()函数来求解矩阵的逆。求逆矩阵的目的是解线性方程组、进行坐标变换、求解参数估计等。逆矩阵在线性模型求解、解决约束问题,以及在物理、工程和计算机图形学中具有广泛的应用。

(2) 求矩阵积:使用 NumPy 库的 matmul()等函数可以计算矩阵的乘积。矩阵积是线性代数中的重要运算,它用于描述线性空间中向量变换的效果,以及计算多个线性变换的复合效果。矩阵积在计算几何、图形学、机器学习等领域具有广泛的应用。

(3) 点积:在 Python 中,可以使用 NumPy 库的 dot()函数来计算向量的点积。点积是一种向量运算,它衡量了两个向量之间的相似性。点积可以用来计算向量的模长、计算向量之间的角度、计算向量之间的投影等。点积在数据分析、模式识别、计算机图形学等领域具有广泛的应用。

(4) 内积与外积:在 Python 中,使用 NumPy 库进行向量运算可以很方便地进行内积和外积的计算。内积描述了线性空间中向量之间的相似性和夹角关系,而外积则返回垂直于两个向量的新向量。内积和外积在几何计算、计算工作和能量等方面具有广泛的应用。

(5) 叉乘:在 Python 中,叉乘是一种线性空间变换方法,用于计算两个向量的叉乘结果。叉乘的目的是衡量两个向量的相对方向和平面的法向量。通过叉乘运算,可以获取一个垂直于原始向量的新向量,它的方向和大小都与输入向量有关。叉乘在计算几何、物理学、计算机图形学等领域具有广泛的应用。

(6) 计算行列式:使用 NumPy 库的 det()函数可以计算矩阵的行列式。行列式衡量了线性空间中矩阵变换对向量面积、体积的影响。计算矩阵的行列式可以用于分析线性变换的性质、判断矩阵的可逆性,并在计算几何、特征值计算等方面发挥作用。

(7) 计算特征值与特征向量:使用 NumPy 库的 linalg 一系列函数可以计算矩阵的特征值和特征向量。特征值和特征向量在线性代数和矩阵分析中应用广泛,用于描述线性变换的特征和性质。通过计算特征值和特征向量,可以实现降维、模式识别、数据压缩等任务。

(8) 奇异值分解:在 Python 中,可以使用 NumPy 库的 svd()函数进行奇异值分解。奇异值分解是一种将矩阵分解为奇异值、左奇异向量和右奇异向量的方法。奇异值分解在数据降维、噪声去除、图像压缩等方面具有广泛的应用。

(9) 最小二乘:最小二乘是一种用于拟合数据和求解回归问题的优化方法。最小二乘在回归分析、机器学习和优化问题中都有广泛的应用。

对于域空间变换,数据变换主要包括傅里叶变换和滤波。

(1) 傅里叶变换。傅里叶变换是一种用于分析信号和波形的数学变换。它将一个信号在时间域上表示的函数转换为频率域上的函数。傅里叶变换可以帮助分析信号的频谱特性,从

而在音频处理、图像处理、通信系统等领域中发挥重要作用。

(2) 滤波。滤波是一种常见的信号处理技术，用于强调或削弱信号中的某些频率成分。滤波可以通过傅里叶变换得到信号的频率表示，然后在频率域上对信号进行滤波操作，最后通过反变换将滤波后的信号转换回时间域。滤波在信号处理、图像处理、音频处理等领域具有广泛的应用。

3.6.2 线性空间变换

本节结合实例，介绍线性空间数据变换的主要方法。

1. 求逆矩阵

numpy.linalg.inv()是 NumPy 库中的一个函数，在线性代数计算中用于计算矩阵的逆矩阵。numpy.linalg.inv()函数的语法如下：

```
numpy.linalg.inv(a)
```

参数 a 是一个输入的方阵或者具有相同的行列数的矩阵。该函数将计算矩阵 a 的逆矩阵。如果矩阵不可逆，将会引发一个异常。如果 a 不是正方形或反转失败，将会抛出 LinAlgError 异常。

numpy.linalg.inv()函数在数学、科学和工程等领域的实际应用中有着广泛的用途，主要包括如下 3 种。

(1) 线性方程组求解。逆矩阵可以用于求解线性方程组。给定一个方程组 Ax=b，其中 A 是系数矩阵，x 是未知变量向量，b 是常量向量。通过计算 A 的逆矩阵，可以通过乘法操作求解未知变量向量 x。

(2) 概率和统计。逆矩阵在概率和统计学中有重要的应用。例如，多元正态分布的协方差矩阵的逆矩阵称为精度矩阵。精度矩阵在概率论和统计学中有很多重要用途。

(3) 机器学习和数据科学。逆矩阵在机器学习和数据科学中也有广泛的应用。例如，在线性回归中，通过计算输入矩阵的逆矩阵可以得到最小二乘解。逆矩阵还可以用于主成分分析(PCA)和线性判别分析(LDA)等降维技术。

通过 numpy.linalg.inv()函数，可以计算数组(Array)或矩阵(Matrix)的逆矩阵。

例 3-44 数组的逆矩阵求解。

```
In:
    import numpy as np
    # 创建一个 3×3 的二维数组
    arr = np.array([[1, 2, 3], [3, 4, 1], [3, 2, 3]])
    # 计算逆矩阵
    inv_arr = np.linalg.inv(arr)
    # 打印逆矩阵
    print(inv_arr)
Out:
    [[-5.00000000e-01   7.40148683e-17   5.00000000e-01]
     [ 3.00000000e-01   3.00000000e-01  -4.00000000e-01]
     [ 3.00000000e-01  -2.00000000e-01   1.00000000e-01]]
```

例 3-44 首先创建了一个 3×3 的二维数组，然后使用 np.linalg.inv 函数计算该矩阵的逆矩阵并输出。

例 3-45 矩阵的逆矩阵求解。

```
In:
    import numpy as np
    mtx = np.matrix([[1, 2, 3,4], [3, 4, 1,2], [3, 2, 3,1],[3,2,1,4]])
    inv_mtx = np.linalg.inv(mtx)
    print(inv_mtx)
Out:
    [[-0.3125      -0.125        0.25         0.3125    ]
     [ 0.125        0.41666667  -0.16666667  -0.29166667]
     [ 0.1875      -0.125        0.25        -0.1875    ]
     [ 0.125       -0.08333333  -0.16666667   0.20833333]]
```

例 3-45 计算并查看一个 4×4 矩阵的逆矩阵。注意：矩阵必须是方阵(行数等于列数)才能计算其逆矩阵。此外，矩阵必须是非奇异的(行列式不为 0)，否则，无法计算逆矩阵，np.linalg.inv 会抛出 LinAlgError 异常。

np.linalg.inv 还可以计算多个矩阵的逆矩阵，如例 3-46 所示。

例 3-46 多个矩阵的逆矩阵求解。

```
In:
    import numpy as np
    a = np.array([[[1.,2.],[2.,3.]],[[4,2],[3,2]]])
    inv_a = np.linalg.inv(a)
    print(inv_a)
Out:
    [[[-3.    2. ]
      [ 2.   -1. ]]

     [[ 1.   -1. ]
      [-1.5   2. ]]]
```

例 3-47 求解逆矩阵实例。利用 numpy.linalg.inv()函数模拟通信中对矩阵进行加密和解密的应用，同时利用逆矩阵求解可以进行解密算法的计算。

加密密钥矩阵 a 为 $\begin{bmatrix} -1 & 1 & -1 & 1 \\ 0 & 2 & 0 & 3 \\ 3 & 1 & -1 & 4 \\ 2 & -1 & 2 & 1 \end{bmatrix}$，得到的密文矩阵是 $\begin{bmatrix} -15 & -15 & -23 & 11 \\ 64 & 29 & 23 & 19 \\ 111 & 32 & 36 & 32 \\ 89 & 58 & 59 & 7 \end{bmatrix}$，求解明文矩阵和明文矩阵内容。

约定明文矩阵内容每个英文字母用一个整数来表示：a:1，b:2…z:26。

```
In:
    #输入加密密钥矩阵 a
    a = np.array([[-1,1,-1,1],[0,2,0,3],[3,1,-1,4],[2,-1,2,1]])
    print("a 的逆矩阵为")
    #求加密密钥矩阵的逆矩阵，即解密密钥矩阵 inv_a
    inv_a = np.linalg.inv(a)
    print(inv_a)
    #输入密文矩阵 R
    R = np.array([[-15,-15,-23,11],[64,29,23,49], [111,32,36,32],[89,58,59,7]])
    print("明文矩阵为")
    #将解密密钥矩阵 inv_a 与密文矩阵 R 相乘，得到明文矩阵
    print(np.dot(inv_a,R))
Out:
    a 的逆矩阵为
    [[-1.25         0.25         0.25        -0.5        ]
     [-2.            1.           0.          -1.         ]
```

```
 [-0.41666667    0.41666667  -0.25         0.16666667]
 [  1.33333333  -0.33333333   0.           0.66666667]]
明文矩阵为
[[18.  5. 14.  3.]
 [ 5.  1. 10. 20.]
 [20. 20. 20.  9.]
 [18.  9.  1.  3.]]
```

例 3-47 通过求解解密密钥矩阵 inv_a 求解出了明文矩阵，通过明文矩阵和明文内容加密规则，可以得出明文内容为 retreating tactic。

例 3-48 将利用逆矩阵对一组身高、体重数据进行处理。表 3-13 所示为一组包含身高和体重的数据集，需要计算其协方差矩阵和逆矩阵。协方差矩阵能揭示数据中变量之间的相关性，为数据处理和特征选择提供依据。逆矩阵则用于在需要时还原数据，便于进一步分析或重建原始数据。

表 3-13　病人信息数据集

病　　人	身高/cm	体重/kg
Patient_1	170	60
Patient_2	165	55
Patient_3	180	80
Patient_4	158	54
Patient_5	175	67
Patient_6	168	58

例 3-48　身高和体重数据集协方差矩阵和逆矩阵的求解。

```
In:
    import numpy as np
    data = np.array([[170, 60], [165, 55], [180, 80], [158, 54], [175, 67],[168,58]])
    a = np.cov(data, rowvar=False)
    inv_a = np.linalg.inv(a)
    print("协方差矩阵:")
    print(a)
    print("逆矩阵:")
    print(inv_a)
Out:
    协方差矩阵:
    [[59.06666667 69.06666667]
     [69.06666667 96.26666667]]
    逆矩阵:
    [[  0.10510073  -0.07540468]
     [-0.07540468    0.06448702]]
```

例 3-48 中，首先定义包含病人身高和体重的数据集 data。接着使用 np.cov()函数计算协方差矩阵 a；然后，使用 np.linalg.inv()函数计算协方差矩阵的逆矩阵 inv_a；最后输出协方差矩阵和逆矩阵的结果。

2. 矩阵积

求矩阵积可以采用多种方法，如调用 numpy.matmul()、点积运算、运用运算符等求解矩阵积，如表 3-14 所示。

表 3-14　求矩阵积的方法

方　　法	方法说明	适用范围
numpy.matmul()	NumPy 库提供的函数,用于计算两个矩阵的乘积。它可以处理不同形状的矩阵,并按矩阵乘法规则进行计算。基于矩阵乘法的标准定义进行操作,因此能够应用于多维数组的乘法	处理高维数组或不同形状的矩阵之间的乘积
点积运算	点积是两个向量之间的乘积,将对应元素相乘并求和。可以用于计算两个相同长度的向量的乘积。对于矩阵而言,可以将它们视为向量的扁平化版本并执行相同的计算。在 NumPy 中,可以使用 numpy.dot 函数进行点积运算	用于计算两个相同长度的向量的乘积
运算符	NumPy 支持使用运算符@来执行矩阵乘法运算。如 A@B 表示将矩阵 A 和 B 相乘。这种运算符方法提供了简洁、直观的语法来计算矩阵的乘积。和 dot 函数用法一致,只不过@是运算符号,dot 是函数。a@b 和 np.dot(a,b)输出表示含义一致	用于计算两个相同长度的矩阵的乘积(当运算符两边的数据维度无法满足矩阵运算时会报错),使用运算符形式的矩阵乘法提供了一种更简洁的语法形式

numpy.matmul()方法用于计算两个矩阵的乘积。它可以处理不同形状的矩阵,并按矩阵乘法规则进行计算。它是基于矩阵乘法的标准定义进行操作的,因此,能够应用于处理高维数组或不同形状的矩阵之间的乘积运算。

例 3-49　numpy.matmul()方法求解矩阵积。

```
In:
    #定义两个矩阵数组 a,b
    a = np.array([[1, 2, 3], [2, 3, 4]])
    b = np.array([[1, 2], [2, 3],[3,4]])
    #调用矩阵内积 matmul 函数,打印结果为矩阵
    print(np.matmul(a,b))
Out:
    [[14 20]
    [20 29]]
```

例 3-50　假设有一幅 3 像素×3 像素的图像矩阵表示为[[135,146,120],[112,118,130],[160,155,165]],将图像逆时针旋转 90°得到新的矩阵。将图像矩阵与旋转矩阵进行矩阵乘法,实现矩阵旋转操作。

```
In:
    #a 为 3 像素×3 像素的图像矩阵
    a = np.array([[135, 146, 120],[112, 118, 130],[160, 155, 165]])
    #b 为旋转矩阵
    b = np.array([[0, -1, 0],[1, 0, 0],[0, 0, 1]])
    #调用矩阵内积 matmul 函数,将两个矩阵进行矩阵乘法,打印结果为逆时针旋转 90°的新矩阵
    print(np.matmul(b,a))
Out:
    [[-112  -118  -130]
     [  135   146   120]
     [  160   155   165]]
```

3. 点积

点积计算可以采用 numpy.dot 方法。numpy.dot 是 NumPy 库中的一个函数,用于计算两个数组的点积,或是一个数组与一个矩阵的乘积。

numpy.dot 方法的用途如下。

(1) 矩阵乘法：当两个数组都是二维矩阵时，dot 方法计算它们的矩阵乘积，即将第一个矩阵的行与第二个矩阵的列进行对应元素相乘并求和的运算。

(2) 计算向量内积：当两个数组都是一维数组(向量)时，dot 方法计算它们的内积，即将对应元素相乘并求和。

(3) 数组之间的乘法：可以使用 dot 方法进行数组之间的乘法操作，其中第一个数组作为标量而第二个数组作为矩阵。除此之外，numpy.dot 方法对于多维矩阵的操作也是适用的，但需要注意维度的匹配。

例 3-51 假设有两个矩阵 X 和 Y，X=[[2,4,6],[1,3,5]]；Y=[[1,2],[3,4],[5,6]]。使用 numpy.dot 计算矩阵 X 和 Y 的点积。

```
In:
    X = [[2, 4, 6],[1, 3, 5]]
    Y = [[1, 2],[3, 4],[5, 6]]
    print(np.dot(X, Y))
Out:
    [[44 56]
    [35 44]]
```

例 3-52 多维矩阵点积。

假设有两个三维矩阵 A 和 B：

```
A = numpy.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
B = numpy.array([[[2, 3], [4, 5]], [[6, 7], [8, 9]], [[10, 11], [12, 13]]])
```

计算矩阵 A 和 B 的点积。

```
In:
    #A、B 为两个三维矩阵
    A = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]])
    B = np.array([[[2, 3], [4, 5]], [[6, 7], [8, 9]], [[10, 11], [12, 13]]])
    #调用点积 dot 函数,打印结果为新矩阵
    print(np.dot(A, B))
Out:
    [
    [[[ 10  13],[ 22  25],[ 34  37]]
     [[ 22  29],[ 50  57],[ 78  85]]]
    [[[ 34  45],[ 78  89],[122 133]]
     [[ 46  61],[106 121],[166 181]]]
    [[[ 58  77],[134 153],[210 229]]
     [[ 70  93],[162 185],[254 277]]]
    ]
```

4. 内积与外积

NumPy 库提供了 numpy.inner()和 numpy.outer()方法，用于向量的内积和外积运算。numpy.inner()方法用于计算两个数组的内积，它将两个数组对应位置的元素相乘，并将乘积相加得到一个标量值。numpy.outer()方法用于计算两个数组的外积，它返回的结果是一个二维矩阵。

例 3-53 两个一维数组 a 和 b：a=np.array([10,13,35])，b=np.array([24,45,16])，计算这两个一维数组的内积。

```
In:
    a = np.array([10, 13, 35])
    b = np.array([24, 45, 16])
    print(np.inner(a, b))
Out:
    #结果为标量值
    1385
```

例 3-54 两个一维数组 a 和 b：a＝np.array([10,13,35])，b＝np.array([24,45,16])，计算这两个一维数组的外积。

```
In:
    #a、b 为两个一维数组
    a = np.array([10, 13, 35])
    b = np.array([24, 45, 16])
    print(np.outer(a, b))
Out:
    #结果为二维矩阵
    [[ 240  450  160]
     [ 312  585  208]
     [ 840 1575  560]]
```

例 3-55 两个一维数组 a 和 b：a＝np.array([10,2,5])，b＝np.array([3,12,6])，计算这两个向量的余弦相似度。

```
In:
    a = np.array([10, 2, 5])
    b = np.array([3, 12, 6])
    #numpy.linalg.norm()函数用于计算数组或向量的范数(或模)。对于一维向量,范数等于向量的
    #模长,对于二维矩阵,范数等于矩阵的最大奇异值
    cosine_similarity = np.inner(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
    print(cosine_similarity)
Out:
    0.5379643898572859
```

说明：通过计算两个向量内积并除以向量的模长的乘积得到余弦相似度。余弦相似度是衡量两个向量方向一致程度的度量(取值范围为－1～1)。

5. 叉乘

向量叉乘又称向量叉积或叉乘，是在三维空间中两个向量之间定义的一种运算。叉乘是将两个向量相乘得到一个新的向量，其方向垂直于原始向量所在的平面，其大小等于原始向量的长度与夹角的正弦值的乘积。向量叉乘通常用符号×表示。

在三维几何中，给定两个向量 u 和 v，它们的叉乘表示为 u×v。叉乘的结果是一个新的向量，可用右手法则来确定它的方向。如果将右手的拇指指向 u 的方向，将食指指向 v 的方向，那么中指的方向就是 u×v 的方向。叉乘符合右手螺旋定则，如图 3-9 所示。

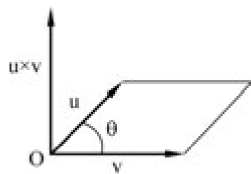


图 3-9 向量叉乘示意图

向量叉乘在物理学、工程学和计算机图形学等领域广泛应用。向量叉乘可以用于计算平面的法向量、向量之间的夹角及三角形的面积等。向量叉乘的概念也可以推广到更高维的向量空间中。

向量叉乘的计算方法如下：

$$\vec{a}\times\vec{b}=(a_1,a_2,a_3)\times(b_1,b_2,b_3)=(a_2b_3-a_3b_2,a_3b_1-a_1b_3,a_1b_2-a_2b_1) \quad (3\text{-}6)$$

numpy.cross()函数是 NumPy 库中用于计算向量叉乘的函数。该函数接收两个输入参数,分别是表示两个向量的 NumPy 数组。它返回一个新的 NumPy 数组,表示两个输入向量的叉乘结果。

numpy.cross()函数的基本语法如下:

```
numpy.cross(a, b, axisa = -1, axisb = -1, axisc = -1, axis = None)
```

参数说明如下。

a:表示第一个向量的 NumPy 数组。

b:表示第二个向量的 NumPy 数组。

axisa,axisb,axisc,axis:这些参数用于指定向量数组的轴,以便在多维数组中进行计算。默认情况下,向量的最后一个轴被视为表示向量分量的轴。

例 3-56 使用 numpy.cross()函数计算两个向量的叉乘。

```
In:
    # 定义两个向量
    a = np.array([1, 5, 9])
    b = np.array([3, 5, 7])
    # 计算叉乘
    result = np.cross(a, b)
    print(result)
Out:
    [-10  20 -10]
```

numpy.cross()函数在处理不同领域的数据时也非常有用,例如,计算图形学中的法线向量、直线方程等,如例 3-57 和例 3-58 所示。

例 3-57 在三维空间中,有一个直线 L,它由一个点 P=(2,1,−3)和一个方向向量 V=[1,−2,4]确定。假设有另外一条直线 M,它与 L 相交于点 Q,并且 M 和 L 的方向向量相同。给定点 Q=(4,−1,1),求直线 M 的方程。

基本思路:首先,定义已知点 P、Q 和方向向量 V;然后,使用 np.cross 构造一个辅助向量 N,用于后续几何分析;最后,计算直线 M 的方程并输出。

```
In:
    from sympy import Symbol
    P = np.array([2, 1, -3])
    V = np.array([1, -2, 4])
    Q = np.array([4, -1, 1])
    # 使用叉乘构造一个辅助向量 N
    N = np.cross(V, [0, 0, 1])
    s = Symbol('s')
    x = Q[0] + s * N[0]
    y = Q[1] + s * N[1]
    z = Q[2] + s * N[2]
    print(f"M: x = {x}, y = {y}, z = {z}")
Out:
    M: x = 4 + s, y = -1 - 2*s, z = 1 + 4*s
```

在例 3-57 中,直线 M 的方程为 x=4,y=−1,z=1。这表示直线 M 经过点(4,−1,1)并且方向向量为(1,−2,4)。

叉乘还能在医学数据处理中用于计算磁共振成像(MRI)中的梯度磁场,如例 3-58 所示。

例 3-58 MRI 技术通过在人体组织中施加外部磁场来创建详细的图像。在 MRI 中,不同组织对磁场的响应不同。梯度磁场是在 MRI 扫描期间施加的额外磁场,用于定位和编码图

像。在 MRI 数据处理中，梯度磁场的方向和大小非常重要，因为它们影响最终生成的图像。一种常见的处理步骤是通过计算梯度磁场的旋转方向或大小来检测异常情况。假设有 3 个矢量 A、B 和 C，A(1,2,1)，B(2,3,4)，D(1,0,1)，它们表示 MRI 扫描期间的梯度磁场读数，现需计算这些矢量的旋转方向。

基本思路：首先，输入梯度磁场 A、B、C 的度数向量；然后，计算 np.cross(B,C)得到中间结果；接着，计算 np.cross(A,np.cross(B,C))；最终得到 direction。

```
In:
    A = np.array([1, 2, 1])
    B = np.array([2, 3, 4])
    C = np.array([1, 0, 1])
    direction = np.cross(A, np.cross(B, C))
    print(direction)
Out:
    [-8  6 -4]
```

在例 3-58 中，得出这些矢量的旋转方向，利用 direction 的结果来分析 MRI 数据中梯度磁场的旋转方向，以更详细和准确的方式进行图像分析。

6. 计算行列式

numpy.linalg.det()函数用于计算矩阵的行列式值。该函数接收一个矩阵作为输入，并返回该矩阵的行列式值。行列式值是一个标量值，用于描述矩阵的性质。注意：numpy.linalg.det()函数仅对方阵有效。如果输入的矩阵不是方阵，则函数将会引发一个异常。

例 3-59 有一个 2×2 的矩阵 x，x=np.array([[23,11],[19,20]])，计算该矩阵的行列式值。

```
In:
    #x 为 2×2 矩阵
    x = np.array([[23,11], [19,20]])
    print(np.linalg.det(x))
Out:
    #结果为标量值
    250.9999999999
```

例 3-60 一个 3×3 的矩阵 x，x=np.array([[22,19,47],[54,25,16],[17,38,29]])，计算该矩阵的行列式值。

```
In:
    #x 为 3×3 的矩阵
    x = np.array([[22, 19, 47], [54, 25, 16], [17, 38, 29]])
    print(np.linalg.det(x))
Out:
    #结果为标量值
    54456.99999999
```

例 3-61 有一个矩阵 x，x=np.array([[3,5],[1,9]])，请判断该矩阵是否可逆。

```
In:
    x = np.array([[3, 5], [1, 9]])
    #判断行列式是否为 0
    determinant = np.linalg.det(x)
    if determinant != 0:
      print("行列式不为 0,矩阵是可逆的")
    else:
      print("行列式为 0,矩阵是不可逆的")
```

```
Out:
    #矩阵可逆
    行列式不为 0,矩阵是可逆的
```

说明,numpy.linalg.det()函数可以用来计算行列式的值,进而根据其是否为 0 来判断矩阵是否可逆。

7. 计算特征值和特征向量

在数据变换中,计算特征值和特征向量是一种常用的方法。特征值和特征向量提供了有关矩阵或数据集的关键信息,可以用于降维、提取重要特征和理解数据的结构。

对于一个仿真矩阵,可以调用 numpy.linalg.eigvals()函数计算其特征值,或调用 numpy.linalg.eig()函数计算其特征值和特征向量。

例 3-62 numpy.linalg.eigvals()计算其特征值。

```
In:
    matrix = np.array([[1, 2], [3, 4]])
    eigenvalues = np.linalg.eigvals(matrix)
    print(eigenvalues)
Out:
    [-0.37228132  5.37228132]
In:
    matrix = np.array([[1, 2], [3, 4]])
    eigenvalues, eigenvectors = np.linalg.eig(matrix)
    print(eigenvalues)
    print(eigenvectors)
Out:
    [-0.37228132  5.37228132]
    [[-0.82456484 -0.41597356]
     [ 0.56576746 -0.90937671]]
```

当涉及医学方面的数据处理时,numpy.linalg.eigvals()和 numpy.linalg.eig()函数可以用于分析和理解医学数据中的特征值和特征向量,如例 3-63 所示。

例 3-63 为了帮助诊断患者的心脏情况,医生使用特征值和特征向量对 ECG 数据进行分析。频谱特征可以从特征值和特征向量中得出。通过分析特征值和特征向量,可以获得心电图信号的频谱特征。这些特征可以用于诊断心脏疾病、检测异常频率组件及评估信号中的频率成分。

```
In:
    numbers = 10
    samples = 1000
    data = np.random.rand(samples, numbers)
    matrix = np.cov(data.T)
    values, vectors = np.linalg.eig(matrix)
    print("特征值:")
    print(values)
    print("特征矩阵:")
    print(vectors)
```

特征值和特征矩阵结果如图 3-10 所示。

在例 3-63 中,首先使用 np.random.rand 函数生成一个随机的二维 ECG 数据矩阵。然后,通过将数据转置并使用 np.cov 函数计算协方差矩阵。接下来,使用 np.linalg.eig 函数计算协方差矩阵的特征值和特征向量。最后,使用 print 语句打印特征值和特征向量的结果。

```
特征值:
[0.15100147 0.12182839 0.11138375 0.09464866 0.08583894 0.07238465
 0.06099987 0.0441891  0.04961077 0.0525481 ]
特征矩阵:
[[ 2.83059095e-01 -3.12221146e-01 -3.37737712e-01 -1.64388017e-01
   5.48542711e-01  4.32782799e-01 -2.77315269e-01  1.21179231e-01
   1.98739531e-01  2.49037548e-01]
 [-3.93906624e-01  2.79911316e-01 -3.04565075e-01 -4.98212679e-01
  -1.26570355e-02  2.66928384e-01  3.25824740e-01 -1.96955997e-01
  -3.15436938e-01  3.31129307e-01]
 [-1.63642154e-01 -6.23042738e-01  5.61329544e-02  1.00690017e-04
  -1.81979893e-01  3.38601149e-01  5.59969226e-01  2.35357359e-01
   6.26298129e-02 -2.47466241e-01]
 [-1.22701415e-01 -5.97743369e-02 -4.34460008e-01  6.46470841e-01
   7.02354517e-03  2.51125303e-02  1.91841741e-01 -5.23352209e-01
   2.01999552e-01  1.50022560e-01]
 [ 3.64425621e-01  4.98122264e-01 -6.82210248e-02  1.23441216e-01
  -1.73126382e-02  5.53100967e-02  4.92201416e-01  4.40350926e-01
   3.46907867e-01  1.98245202e-01]
 [ 2.47116113e-01  1.41052476e-01  4.59532790e-01 -1.92715369e-02
   5.21763103e-01  1.75646052e-01  3.35795500e-01 -4.58837420e-01
  -1.10980719e-01 -2.62298801e-01]
 [ 6.33686608e-01 -5.16589971e-02 -4.67390407e-01 -1.53397766e-01
  -2.87949748e-01 -6.23556835e-02  7.72452615e-02 -1.55265757e-01
  -3.56243563e-01 -3.31663855e-01]
 [-6.63007413e-02  2.08643204e-02 -5.60079406e-02  4.69630712e-01
   2.94060786e-01  2.27165401e-02  3.43477760e-02  3.98616436e-01
  -7.20965637e-01  6.80068174e-02]
 [ 3.50356208e-01 -2.49548845e-01  3.91273153e-01  6.10149353e-02
  -3.30357366e-01  9.09715177e-02  2.10267619e-02 -1.71711634e-01
  -1.90938254e-01  6.88739549e-01]
 [ 1.74454742e-02 -3.14662039e-01 -1.03716392e-01 -2.08375430e-01
   3.38696659e-01 -7.61298941e-01  3.22422264e-01  1.74696238e-02
   2.13469765e-02  2.17939317e-01]]
```

图 3-10　特征值和特征矩阵结果

在实际应用中,对导入患者的 ECG 数据进行变换,可以用于后续的治疗诊断。

对于厄米特矩阵(共轭对称矩阵)或实对称矩阵,可以调用 numpy. linalg. eigvalsh()函数来计算其特征值,调用 numpy. linalg. eigh()函数来计算其特征值和特征向量。

例 3-64　调用 numpy. linalg. eigvalsh()函数和 numpy. linalg. eigh()函数计算特征值和特征向量。

```
In:
    a = np.array([[1, 2, 3],
                  [2, 5, 6],
                  [3, 6, 9]])
    eigenvalues = np.linalg.eigvalsh(a)
    print("特征值: ")
    print(eigenvalues)
    eigenvalues, eigenvectors = np.linalg.eigh(a)
    print("特征值: ")
    print(eigenvalues)
    print("特征向量: ")
    print(eigenvectors)
Out:
    特征值:
    [5.35409526e-17 6.99264746e-01 1.43007353e+01]
    特征值:
    [9.69980627e-16 6.99264746e-01 1.43007353e+01]
```

```
特征向量:
[[-9.48683298e-01   1.77819106e-01   -2.61496397e-01]
 [  2.89601067e-16  -8.26924214e-01   -5.62313386e-01]
 [  3.16227766e-01   5.33457318e-01   -7.84489190e-01]]
```

对于 numpy. linalg. eigvals()、numpy. linalg. eig()和 numpy. linalg. eigvalsh()、numpy. linalg. eigh()这些计算特征值和特征向量函数的用法,表 3-15 进行了功能和适用范围的对比。

表 3-15　计算特征值和特征向量函数的功能和适用范围

函　　数	功　　能	适 用 范 围
numpy. linalg. eigvals()	计算特征值	普通(非对称)矩阵
numpy. linalg. eig()	计算特征值和特征向量	普通(非对称)矩阵
numpy. linalg. eigvalsh()	计算特征值	对称或复 Hermite 矩阵
numpy. linalg. eigh()	计算特征值和特征向量	对称或复 Hermite 矩阵

相比 numpy. linalg. eigvals()和 numpy. linalg. eig()函数,numpy. linalg. eigvalsh()和 numpy. linalg. eigh()函数在计算特征值和特征向量方面通常更加高效,尤其是针对对称或厄米矩阵。这些函数在特征值分析、量子力学问题、信号处理、PCA 主成分分析等领域都有广泛的应用。

例 3-65　分析某个金融市场中的股票组合优化问题。根据收集的股票的历史收益率数据,计算出这些股票的协方差矩阵。计算协方差矩阵的特征值和特征向量,并根据特征值和特征向量来确定最优的投资组合权重。

```
In:
    # 定义股票的历史收益率数据矩阵
    matrix = np.array([[0.010, 0.030, 0.050],
                       [0.030, 0.070, 0.090],
                       [0.050, 0.090, 0.110]])
    # 计算特征值和特征向量(适用于实对称矩阵)
    eigenvalues, eigenvectors = np.linalg.eigh(matrix)
    # 只计算特征值(适用于实对称矩阵)
    values = np.linalg.eigvalsh(matrix)
    # 打印特征值
    print("特征值:", values)
    print("特征值: ", eigenvalues)
    # 打印特征向量
    print("特征向量: ", eigenvectors)
    # 计算最大特征值对应的特征向量,并归一化为投资组合权重
    a = eigenvectors[:, -1] / np.sum(eigenvectors[:, -1])
    print("最优的投资组合权重 (方法 1): ", a)
    # 计算最大特征值对应的特征向量,并乘以最大特征值
    a = eigenvectors[:, -1] * eigenvalues[-1]
    # 归一化为投资组合权重
    b = a / np.sum(a)
    print("最优的投资组合权重 (方法 2): ", b)
Out:
    特征值:[-0.0129104   0.00310124   0.19980917]
    特征值:  [-0.0129104   0.00310124   0.19980917]
    特征向量:  [[-0.78344549   0.54879962  -0.29160272]
     [-0.30455922  -0.74806527  -0.58961176]
     [  0.54171658   0.37311838  -0.75321034]]
    最优的投资组合权重 (方法 1):  [0.17841305 0.36074572 0.46084123]
    最优的投资组合权重 (方法 2):  [0.17841305 0.36074572 0.46084123]
```

例 3-65 首先输入股票的历史收益率数据 matrix ,然后使用 np. linalg. eigh 计算特征值和

特征向量，使用 np. linalg. eigvalsh 只计算特征值。最后，通过两种方法计算权重：一种是直接归一化特征向量(eigenvectors[:,−1] / np. sum(eigenvectors[:,−1]))；另一种是将特征向量乘以最大特征值后再归一化(eigenvectors[:,−1] * eigenvalues[−1]并除以总和)。这两种方法都用于找到最优的投资组合权重。

8. 奇异值分解

奇异值分解是一种线性代数的分解方法，将矩阵分解成 3 个矩阵的乘积：U、Σ 和 V^T。其中，U 和 V^T 是正交矩阵，Σ 是对角矩阵，对角线上的元素称为奇异值。numpy. linalg. svd()函数接收一个矩阵作为输入，并返回 3 个输出矩阵：U、Σ 和 V^T，这些矩阵可以用于重新构造原始矩阵。奇异值分解在数据降维、推荐系统和图像压缩等领域广泛应用。它提供了对数据的低秩逼近，有助于理解数据的结构和特征。

numpy. linalg. svd()函数可以用于计算矩阵的奇异值分解。此外，scipy 库也提供了 scipy. linalg. svd()和 scipy. linalg. svdvals()等函数来进行奇异值分解。

例 3-66　对矩阵进行奇异值分解。

```
In:
    x = np.array([[1, 2], [3, 4], [5, 6]])
    # 将矩阵 x 分解为 U、S 和 Vt 三个矩阵
    U, S, Vt = np.linalg.svd(x)
    print("U:")
    print(U)
    print("S:")
    print(S)
    print("Vt:")
    print(Vt)
Out:
    U:
    [[-0.2298477   0.88346102  0.40824829]
     [-0.52474482  0.24078249 -0.81649658]
     [-0.81964194 -0.40189603  0.40824829]]
    S:
    [9.52551809 0.51430058]
    Vt:
    [[-0.61962948 -0.78489445]
     [-0.78489445  0.61962948]]
```

例 3-67　热力图是一种可视化矩阵数据的方式，而奇异值分解是一种将矩阵分解为 3 个矩阵的方法。可以使用 numpy 中的 numpy. linalg. svd() 函数进行矩阵的奇异值分解。本例利用 numpy. linalg. svd()函数对热力图进行奇异值分解。

```
In:
    import matplotlib.pyplot as plt
    # 生成一个随机的热力图
    heatmap = np.random.rand(10, 10)
    # 绘制原始热力图
    plt.imshow(heatmap, cmap='hot', interpolation='nearest')
    plt.title('Original Heatmap')
    plt.colorbar()
    plt.show()
    # 对热力图进行奇异值分解
    U, Σ, Vt = np.linalg.svd(heatmap)
    # 取前 k 个奇异值来重构热力图
```

```
    k = 5 #还可以更改成10个奇异值
    reconstructed_heatmap = U[:, :k] @ np.diag(Σ[:k]) @ Vt[:k, :]
    # 绘制重构后的热力图
    plt.imshow(reconstructed_heatmap, cmap='hot', interpolation='nearest')
    plt.title(f'Heatmap Reconstructed with k={k}')
    plt.colorbar()
    plt.show()
```

奇异值分解代码结果如图 3-11 所示。

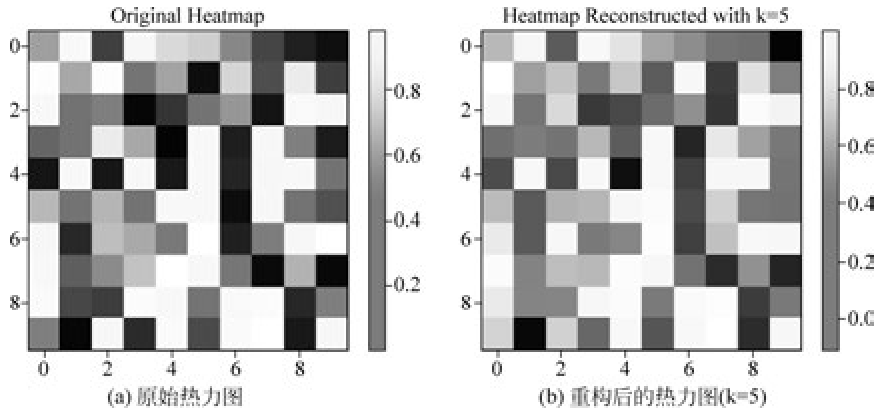


图 3-11　奇异值分解代码结果

9. 最小二乘

最小二乘法是统计学中常用的一种数学方法，用于拟合数据并找到最佳拟合线。它的主要目标是通过最小化观测数据与拟合线之间的残差平方和来找到最优解。

可以使用 numpy.linalg.lstsq()函数求解最小二乘法问题，函数原型如下：

```
numpy.linalg.lstsq(a, b, rcond='warn')
```

该函数的参数及作用如下。

a：(m,n)形状的数组，表示系数矩阵，其中 m 为观测值的数量，n 为未知数的数量。

b：(m,)或(m,k)形状的数组，表示观测值的目标值。当 b 为二维数组时，每列代表一个目标值，用于多变量拟合。

rcond：浮点数或 None，表示奇异值分解的截断阈值。默认为 warn，当 rcond 为 warn 时，为尽可能高的精度打印警告信息；当 rcond 为 None 时，将根据数组的特征值选择适当的阈值。

numpy.linalg.lstsq()函数的返回值如下。

x：形状为(n,)或(n,k)的数组，表示拟合的解。

residues：表示残差平方和，在每个观测值中计算目标值与拟合值之间的差值的平方和。

rank：表示系数矩阵 a 的秩。

s：表示奇异值分解的奇异值。

例 3-68　利用 numpy.linalg.lstsq()函数计算最小二乘解、秩、奇异值。

```
In:
    A = np.array([[1, 2], [3, 4], [5, 6]])
    b = np.array([7, 8, 9])
    x, residuals, rank, singular_values = np.linalg.lstsq(A, b, rcond=None)
```

```
        print("最小二乘解:")
        print(x)
        print("残差平方和:")
        print(residuals)
        print("矩阵 A 的秩:")
        print(rank)
        print("矩阵 A 的奇异值:")
        print(singular_values)
    Out:
        最小二乘解:
        [-6.   6.5]
        残差平方和:
        [1.98622668e-30]
        矩阵 A 的秩:
        2
        矩阵 A 的奇异值:
        [9.52551809 0.51430058]
```

利用 numpy.linalg.lstsq()函数可以进行计算最小二乘解,最后计算出斜率和截距。

例 3-69 输入正整数 n,生成 n 个数据点的 x 坐标和对应的 y 坐标,作为线性回归的数据集。使用 numpy.linalg.lstsq 函数拟合这些数据点,计算最小二乘解,并打印出斜率和截距。

```
    In:
        def data(n):
            x_coords = []
            y_coords = []
            print("请依次输入{}个 x 坐标: ".format(n))
            for i in range(n):
                x = float(input())
                x_coords.append(x)
            print("请依次输入{}个 y 坐标: ".format(n))
            for i in range(n):
                y = float(input())
                y_coords.append(y)
            return x_coords, y_coords
        n = int(input("请输入数据集的大小(正整数): "))
        x_coords, y_coords = data(n)
        A = np.vstack([x_coords, np.ones(len(x_coords))]).T
        b = y_coords
        x, residuals, rank, singular_values = np.linalg.lstsq(A, b, rcond=None)
        print("最小二乘解: ")
        print("斜率: {:.8f}".format(x[0]))
        print("截距: {:.8f}".format(x[1]))
    Out:
        请输入数据集的大小(正整数): 5
        请依次输入 5 个 x 坐标:
        2,4,6,8,10
        请依次输入 5 个 y 坐标:
        4,5,6,7,8
        最小二乘解:
        斜率: 0.50000000
        截距: 3.00000000
```

在例 3-69 中,运用函数计算出了斜率和截距。最小二乘解在数据处理中还有很多应用。最小二乘法可以用于医学图像处理中的去噪、边缘检测、图像恢复等任务。通过最小二乘优化,可以降低图像噪声、增强特定结构等。

例 3-70 模拟生成一个带噪声的图像数据 x 和 y,其中的 y 值是通过对正弦函数添加高

斯噪声生成的。然后使用 numpy. linalg. lstsq()函数对数据进行图像去噪处理。使用多项式拟合获得系数 coefficients 并根据获得的系数计算拟合曲线 y_fit,用 Matplotlib 绘制原始的带噪声数据和去噪后的曲线。

```
In:
    import matplotlib.pyplot as plt
    x = np.linspace(0, 10, 100)
    y = np.sin(x) + np.random.normal(0, 0.1, 100)
    A = np.vander(x, 5)
    coefficients = np.linalg.lstsq(A, y, rcond=None)[0]
    y_fit = np.dot(A, coefficients)
    plt.plot(x, y, 'b', label='Noisy Data')
    plt.plot(x, y_fit, 'r', label='Denoised Data')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Image Denoising')
    plt.show()
```

带噪声数据和去噪后的曲线如图 3-12 所示。

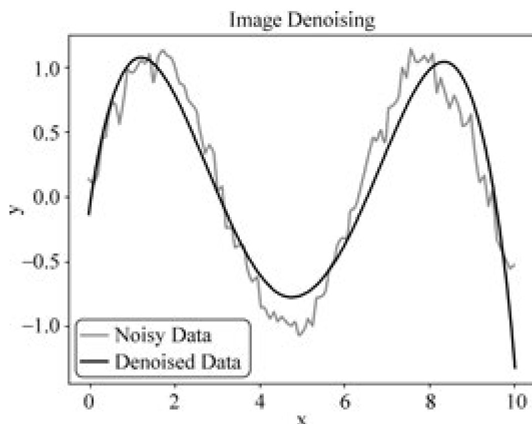


图 3-12　带噪声数据和去噪后的曲线

图 3-12 展示了降噪之后的曲线,可以比较带噪声数据和去噪后数据之间差异。本例中使用的是多项式拟合,通过不断调试代码,读者可以思考是否存在某个多项式阶数,可以获得最佳的去噪效果,以及除了多项式阶数,还有哪些因素可能会影响图像去噪的效果。

3.6.3　域空间变换

1. 傅里叶变换

傅里叶变换(Fourier Transform)是一种用于将信号从时域(时间域)转换为频域(频率域)的数学技术。傅里叶变换的主要思想是将一个信号(如声音、图像或任何其他时域信号)分解为一组基本的正弦和余弦函数,这些函数具有不同的频率和振幅。傅里叶变换可以帮助分析信号的频谱特性,从而在音频处理、图像处理、通信系统等领域中发挥重要作用。

可以使用 numpy. fft. rfft()函数来执行傅里叶变换,该函数的作用是将实数输入的信号从时域转换为频域,并返回一个包含频域复数结果的数组,函数表示如下:

```
numpy.fft.rfft(a, n=None, axis=-1, norm=None)
```

该函数的参数及作用如下。

a：要进行傅里叶变换的实数序列，可以是一维或多维数组，对于多维数组，可以通过 axis 参数指定在哪个轴上执行 RFFT。

n：可选参数，用于指定进行傅里叶变换时的 FFT 长度。

axis：可选参数，用于指定在多维数组中进行 RFFT 的轴。

norm：可选参数，用于指定归一化的方式。

该函数的返回值是一个包含复数结果的数组，表示输入信号在频域中的分量。输出数组的大小取决于输入信号和参数 n 的大小。

例 3-71 使用 numpy.fft.rfft()函数进行傅里叶变换。

```
In:
    data = [0, 1, 0, 0]
    rfft_result = np.fft.rfft(data)
    print(rfft_result)
Out:
    [ 1. + 0.j  0. - 1.j - 1. + 0.j]
```

在例 3-71 中，将一个包含 4 个实数元素的数组[0，1，0，0]作为输入，并对其进行了实数输入的傅里叶变换。输出结果是一个包含频域中的复数数组，只包含非负频率的部分。

例 3-72 给出一个实数信号，对其进行傅里叶变换并生成原始信号图和频谱图。

```
In:
    import matplotlib.pyplot as plt
    # 生成一个实数信号(频率为 5 Hz 的正弦波和频率为 10 Hz 的余弦波的叠加)
    sample_rate = 1000                # 采样率为 1000 Hz
    duration = 1                      # 信号持续时间为 1 s
    t = np.linspace(0, duration, int(sample_rate * duration), endpoint = False)
    signal = np.sin(2 * np.pi * 5 * t) + np.cos(2 * np.pi * 10 * t)
    # 进行傅里叶变换
    rfft_result = np.fft.rfft(signal)
    # 计算频率轴
    freqs = np.fft.rfftfreq(len(t), d = 1/sample_rate)
    # 绘制信号和频谱图
    plt.figure(figsize = (10, 6))
    # 绘制信号图
    plt.subplot(2, 1, 1)
    plt.plot(t, signal)
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.title('Time Domain Signal')
    # 绘制频谱图
    plt.subplot(2, 1, 2)
    plt.plot(freqs, np.abs(rfft_result))
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Amplitude')
    plt.title('Frequency Spectrum of the Signal')
    plt.tight_layout()
    plt.show()
```

信号和频谱图如图 3-13 所示。

在例 3-72 中，首先生成了一个包含频率为 5 Hz 的正弦波和频率为 10 Hz 的余弦波的实数信号，并对其进行了实数输入的傅里叶变换，然后绘制了原始信号和频谱图。

频域分析在医学信号处理中非常有用。频域分析通过傅里叶变换将心电图(ECG)信号从时域转换为频域，便于识别和分析信号中的频率成分，从而帮助检测心律不齐等问题，并通过去除噪声成分来提高信号质量。

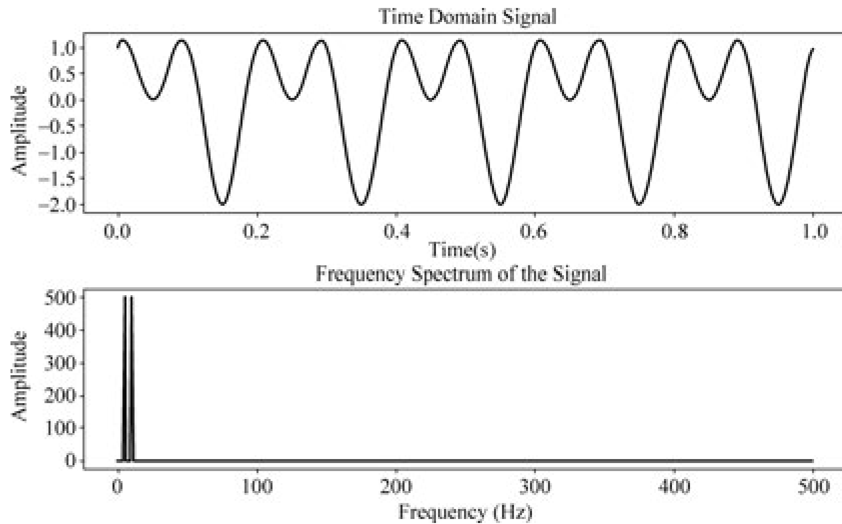


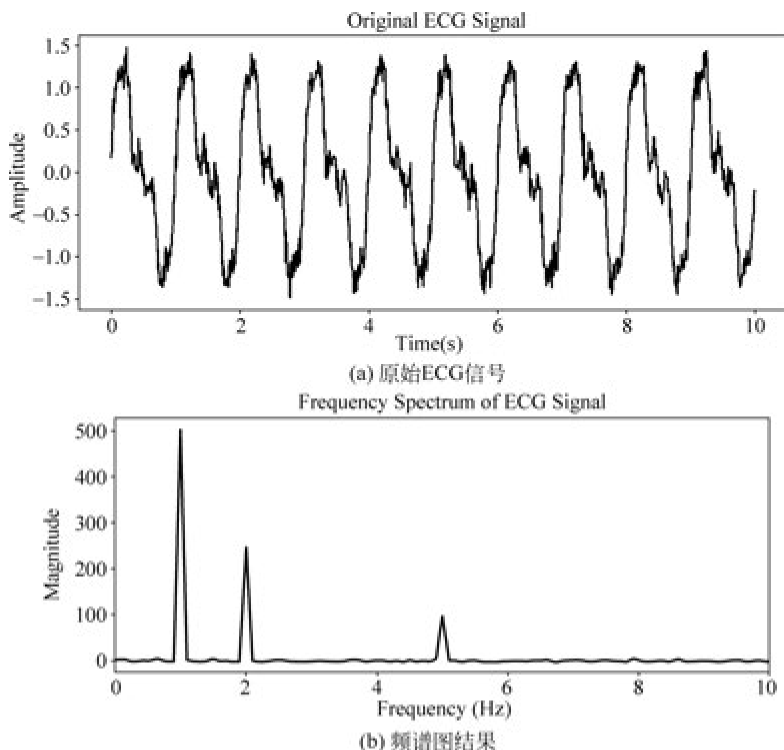
图 3-13 信号和频谱图

例 3-73 对心电图信号进行频域分析，以检测信号中的频率成分。

```
In:
    import matplotlib.pyplot as plt
    # 1. 生成或加载示例 ECG 信号
    # 这里生成一个简化的 ECG 信号作为示例
    np.random.seed(0)
    t = np.linspace(0, 10, 1000, endpoint=False)     # 时间轴,10 秒,1000 个采样点
    frequencies = [1, 2, 5]                          # 假设信号中有 3 个主要频率成分
    amplitudes = [1.0, 0.5, 0.2]                     # 每个频率成分的振幅
    signal = np.sum([amp * np.sin(2 * np.pi * freq * t) for amp, freq in zip(amplitudes,
frequencies)], axis=0)
    noise = 0.1 * np.random.randn(t.size)            # 添加一些噪声
    ecg_signal = signal + noise
    # 显示原始 ECG 信号
    plt.figure(figsize=(10, 4))
    plt.plot(t, ecg_signal)
    plt.title('Original ECG Signal')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.show()
    # 2. 使用 numpy.fft.rfft 进行傅里叶变换
    rfft_result = np.fft.rfft(ecg_signal)
    # 3. 计算频率轴
    n = len(ecg_signal)
    sample_rate = n / (t[-1] - t[0])                     # 采样率
    freqs = np.fft.rfftfreq(n, d=1/sample_rate)
    # 4. 绘制频域图
    plt.figure(figsize=(10, 4))
    plt.plot(freqs, np.abs(rfft_result))
    plt.title('Frequency Spectrum of ECG Signal')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude')
    plt.xlim(0, 10)                                      # 只显示 0～10Hz 的频率范围
    plt.show()
```

原始图像和频谱图结果如图 3-14 所示。

例 3-73 使用 numpy.fft.rfft()函数对生成的 ECG 信号进行傅里叶变换，将其从时域转换



(a) 原始ECG信号

(b) 频谱图结果

图 3-14　原始图像和频谱图结果

到频域。通过 numpy.fft.rfftfreq()计算频率轴，并绘制频谱图来观察信号中的主要频率成分。这有助于识别和分析 ECG 信号中的重要特征，如心律不齐等。

2. 滤波

滤波是一种信号处理技术，用于去除信号中不需要的频率成分或噪声。滤波可以应用于各种信号，包括音频、图像、传感器数据等。在滤波过程中，可以使用滤波器来改变信号的频谱特性，从而实现不同的信号处理目标。

可以使用 scipy.signal.filtfilt()函数实现滤波，该函数用于在时域中对信号进行滤波，可以有效地去除信号中的噪声或不需要的频率成分，同时保留信号的相位信息，函数表示如下：

```
scipy.signal.filtfilt(b, a, x, axis = - 1, padtype = 'odd', padlen = None, method = 'pad', irlen = None)
```

该函数的参数及作用如下。

b：滤波器的分子系数向量，即传递函数的分子部分。

a：滤波器的分母系数向量，即传递函数的分母部分。如果 a[0]不是 1，那么 a 和 b 都会被归一化，使得 a[0]为 1。

x：要过滤的数据数组。

axis：可选，应用过滤器的 x 轴，默认值为−1。

padtype：可选，必须是 odd、even、constant 或无，决定了用于应用滤波器的填充信号的扩展类型。如果 padtype 为 None，则不使用填充。默认值为 odd。

padlen：可选，要扩展的元素数 x 在两端轴应用过滤器之前。

method：可选，确定处理信号边的方法，可以是 pad 或 gust。当 method 为 pad 时，填充信号；当 method 为 gust 时，使用 Gustafsson 的方法，忽略 padtype 和 padlen。

irlen：可选，当 method 为 gust 时，irlen 指定滤波器的脉冲响应长度。如果 irlen 为 None，则不会忽略任何部分的脉冲响应。对于长信号，指定 irlen 可以显著提高滤波器的性能。

该函数的返回值是滤波后的输出信号，与输入信号具有相同的形状和类型。

例 3-74 模拟一个含有噪声的正弦波作为输入信号，设计一个滤波器对其进行滤波，并绘制原始信号和滤波后的信号。

```
In:
    import scipy.signal as signal
    import matplotlib.pyplot as plt
    # 生成一个含有噪声的正弦波作为示例输入信号
    np.random.seed(42)
    num_samples = 100
    time = np.linspace(0, 1, num_samples)
    noise = np.random.randn(num_samples) * 0.5
    signal_input = np.sin(2 * np.pi * 5 * time) + noise
    # 设计一个低通滤波器(巴特沃斯滤波器)并获取其分子系数和分母系数
    cutoff_freq = 0.2     # 截止频率为 0.2 Hz
    order = 4
    b, a = signal.butter(order, cutoff_freq, btype='low')
    # 使用 filtfilt 函数进行无相位滤波
    filtered_signal = signal.filtfilt(b, a, signal_input)
    # 绘制原始信号和滤波后的信号
    plt.figure(figsize=(10, 6))
    plt.plot(time, signal_input, label='Original Signal')
    plt.plot(time, filtered_signal, label='Filtered Signal')
    plt.xlabel('Time')
    plt.ylabel('Amplitude')
    plt.title('Original Signal and Filtered Signal')
    plt.legend()
    plt.grid(True)
    plt.show()
```

原始信号和滤波后的信号如图 3-15 所示。

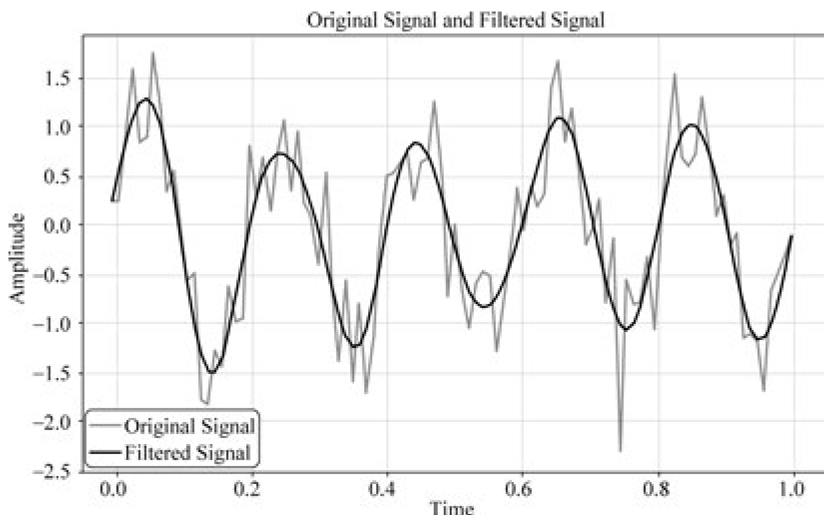


图 3-15 原始信号和滤波后的信号

例 3-74 中生成了一个包含噪声的正弦波信号，使用 scipy.signal.filtfilt()函数设计了一个低通滤波器(巴特沃斯滤波器)并获取了其分子系数 b 和分母系数 a。然后，使用 filtfilt()函

数对输入信号进行无相位低通滤波，去除噪声并平滑信号，并绘制了原始信号和滤波后的信号进行对比。

例 3-75 生成一个长度为 60 的随机信号，创建滤波器，使用两种不同的方法进行滤波，并绘制原始信号与两种滤波方法的对比图。

```
from scipy import signal
import matplotlib.pyplot as plt
# 创建滤波器
b, a = signal.ellip(4, 0.01, 120, 0.125)
# 使用随机数函数生成一个长度为 60 的随机信号
rng = np.random.default_rng()
n = 60
sig = rng.standard_normal(n) ** 3 + 3 * rng.standard_normal(n).cumsum()
# 分别使用 gust 和 pad 方法进行无相位滤波
fgust = signal.filtfilt(b, a, sig, method="gust")
fpad = signal.filtfilt(b, a, sig, padlen=50)
# 绘制对比结果
plt.plot(sig, label='Original Signal')
plt.plot(fgust,  label='Gust Signal')
plt.plot(fpad,  label='Pad Signal')
plt.title('Original Signal and Filtered Signal')
plt.legend()
plt.grid(True)
plt.show()
```

原始信号与两种滤波方法结果对比如图 3-16 所示。

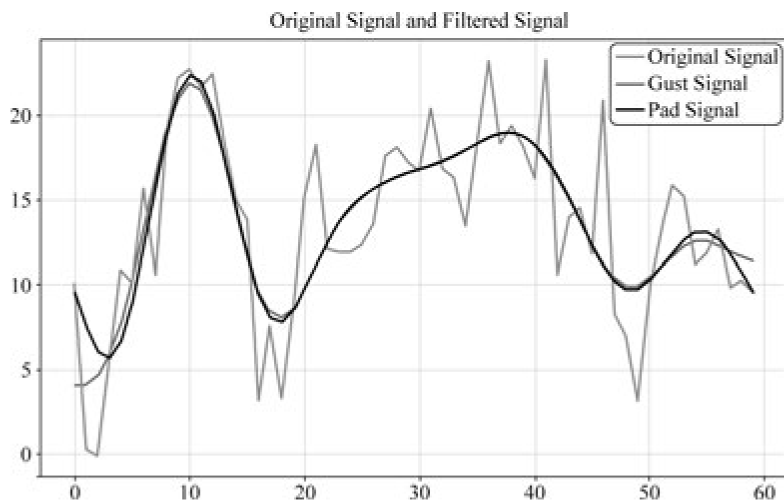


图 3-16 原始信号与两种滤波方法结果对比

在例 3-75 中，生成了一个长度为 60 的随机信号，并使用椭圆滤波器对该信号进行 gust 方法和 pad 方法的无相位滤波。然后，将原始信号和滤波后的信号一起绘制在同一张图上，方便对比观察滤波效果。

注意，在该例子中，使用 signal.ellip()函数设计了一个椭圆滤波器，其中参数 4 是滤波器的阶数，0.01 是通带最大衰减(dB)，120 是阻带最小衰减(dB)，0.125 是指截止频率为奈奎斯特频率的 0.125 倍(该例子中采样频率默认为 1Hz)。

3.6.4 案例——心脑血管数据变换

对心脑血管数据进行数据变换可以提取有价值的信息，减少特征的冗余，将数据降维并提

高机器学习模型的性能。在心脑血管数据分析中，进行特征向量分析可以发现不同特征之间的关系，降低数据维度并提取最具区分性的特征，更好地理解数据中的模式和趋势。进行PCA分析可以将原始特征通过线性变换映射到新的特征空间，使得新特征之间的相关性最小，达到降低数据维度、提取关键特征的目的。

例 3-76 计算特征值和特征向量和PCA分析。

```
In:
    from sklearn.preprocessing import StandardScaler
    from sklearn.decomposition import PCA
    import matplotlib.pyplot as plt
    df = pd.read_excel(r'Data/T2_HealthInfo.xlsx')

    df = df.replace('#NULL!', np.nan)
    df['舒张压'] = pd.to_numeric(df['舒张压'])
    df['收缩压'] = pd.to_numeric(df['收缩压'])
    df['空腹血糖'] = pd.to_numeric(df['空腹血糖'])
    # 特征选择
    # 选择'空腹血糖', '甘油三酯', '总胆固醇'作为特征
    X = df[['空腹血糖', '甘油三酯', '总胆固醇']].copy()
    # 使用平均值填充缺失值
    X_filled = X.fillna(X.mean())
    # 标准化
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X_filled)
    # PCA 分析
    pca = PCA()
    pca.fit(X_scaled)
    # 获取特征值和特征向量
    eigenvalues = pca.explained_variance_
    eigenvectors = pca.components_
    # 计算解释方差比例
    explained_variance_ratio = pca.explained_variance_ratio_

    # 绘制解释方差比例图
    plt.figure(figsize=(10, 5),dpi=300)
    plt.rcParams['font.family'] = 'SimHei'  # 使用思源黑体或其他支持中文的字体
    plt.bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, alpha=
0.5, align='center')
    plt.step(range(1, len(explained_variance_ratio) + 1), np.cumsum(explained_variance_
ratio), where='mid')
    plt.xlabel('主成分数量')
    plt.ylabel('解释方差比例')
    plt.title('解释方差比例图')
    plt.show()

    # 输出特征值和特征向量
    print("特征值:", eigenvalues)
    print("特征向量:", eigenvectors)
    print("解释方差比例:", explained_variance_ratio)
Out:
    特征值:[1.30422369 0.89217032 0.80420611]
    特征向量:[[ 0.5423216   0.56069267  0.62570841]
     [-0.74583926  0.66414723  0.05130547]
     [ 0.3867959   0.49450196 -0.778368  ]]
    解释方差比例:[0.43465428 0.29733063 0.26801509]
```

方差比例如图3-17所示。

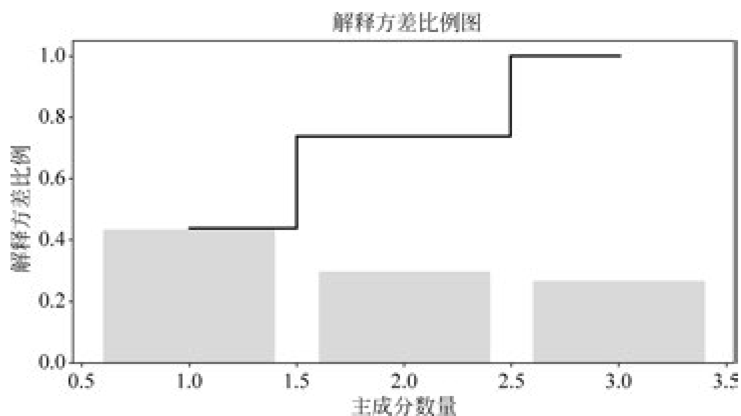


图 3-17 方差比例

对心脑血管数据进行奇异值分解可以降低数据维度，有助于减少冗余信息，降低数据噪声的影响，提高计算效率。除此之外，通过分析奇异值矩阵，可以发现数据中的潜在结构和模式，识别出在数据中具有显著影响的特征；而计算奇异值的解释方差比，可以确定哪些奇异值对数据的变异性有更大的解释能力，从而有助于深入研究和理解特定疾病或疾病进展的关键因素。

例 3-77 对心血管数据进行奇异值分解。

```
In:
    from scipy.linalg import svd
    import matplotlib.pyplot as plt
    # 加载数据集
    data = pd.read_excel(r'Data/T2_HealthInfo.xlsx')
    # 检查数据集的基本信息
    data_info = data.info()
    # 检查数据集中是否存在无穷大值
    infinite_values = data.isin([np.inf, -np.inf]).sum()
    # 删除含有缺失值的行
    data_cleaned = data.dropna()
    # 选择"收缩压"和"舒张压"列进行奇异值分解
    blood_pressure_data_cleaned = data_cleaned[['收缩压', '舒张压']]
    # 执行奇异值分解
    U_cleaned, s_cleaned, Vt_cleaned = svd(blood_pressure_data_cleaned, full_matrices=False)
    # 创建 Sigma 对角矩阵
    Sigma_cleaned = pd.DataFrame([[s_cleaned[0], 0], [0, s_cleaned[1]]])
    # 显示奇异值、左奇异向量和右奇异向量的头几行
    singular_values_cleaned = pd.DataFrame(s_cleaned, columns=['奇异值'])
    U_head_cleaned = pd.DataFrame(U_cleaned[:, :2], columns=['U1', 'U2'])
    Vt_head_cleaned = pd.DataFrame(Vt_cleaned.T[:, :2], columns=['V1', 'V2'])
    print(singular_values_cleaned, U_head_cleaned, Vt_head_cleaned)

    # 选择第一个和第二个奇异值进行降维
    selected_singular_values = s_cleaned[:2]
    # 构造新的 Sigma 对角矩阵
    Sigma_reduced = pd.DataFrame([[selected_singular_values[0], 0], [0, selected_singular_
values[1]]])
    # 构造降维后的数据矩阵
    X_reduced = U_cleaned @ Sigma_reduced @ Vt_cleaned
    # 绘制降维后的数据矩阵的散点图
    plt.figure(figsize=(10, 6))
    plt.scatter(X_reduced.iloc[:, 0], X_reduced.iloc[:, 1])
```

```
plt.title('Data scatter plot after dimensionality reduction')
plt.xlabel('Singular value 1')
plt.ylabel('Singular value 2')
plt.grid(True)
plt.show()
# 计算累积解释方差比
total_variance = sum(selected_singular_values ** 2)
cumulative_variance_ratio_reduced = np.cumsum(selected_singular_values ** 2) / total_variance
# 绘制累积解释方差比图像
plt.figure(figsize=(10, 6))
plt.plot(cumulative_variance_ratio_reduced, marker='o', linestyle='--', color='b')
plt.title('Cumulative explanatory variance ratio after dimensionality reduction')
plt.xlabel('Number of singular values')
plt.ylabel('Cumulative explanatory variance ratio')
plt.grid(True)
plt.show()
```

降维后的数据散点图如图 3-18 所示。累积解释方差比图像如图 3-19 所示。

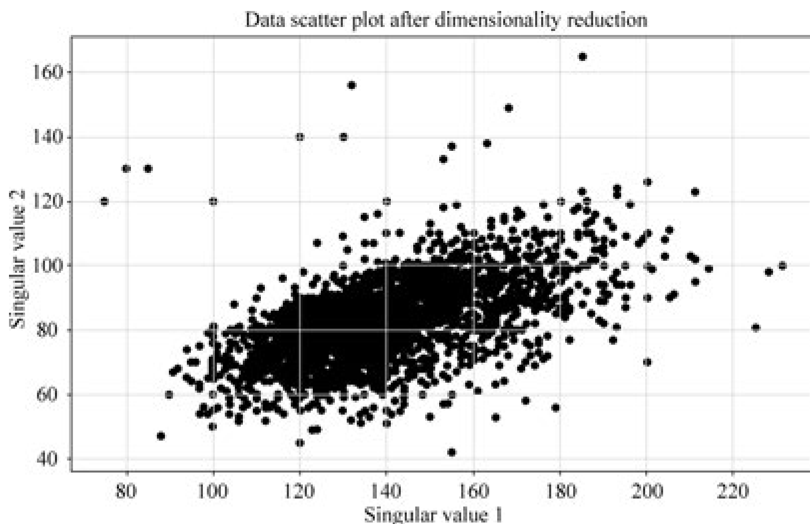


图 3-18 降维后的数据散点图

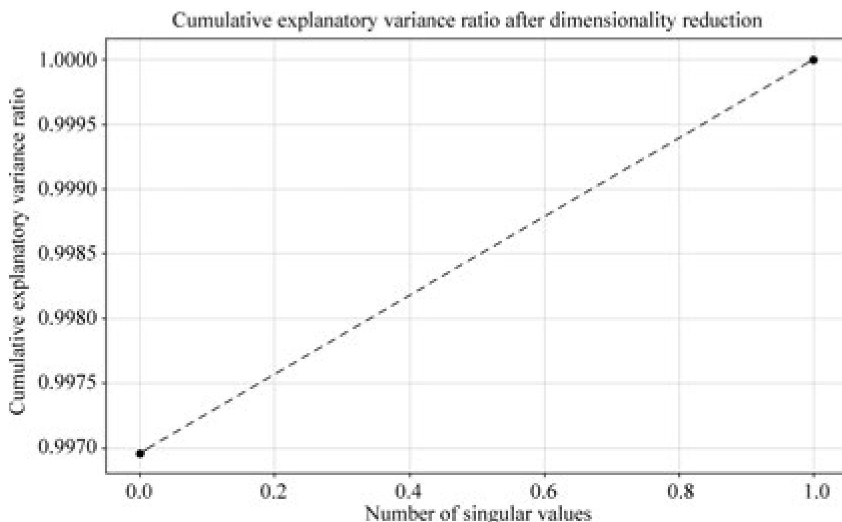


图 3-19 累积解释方差比图像

3.7 数据规约

数据规约(Data Reduction)是数据预处理的一个重要步骤,旨在通过降维、数据压缩和特征选择等方式,减少数据的维度和规模,从而提升数据挖掘的效率和可操作性。

3.7.1 数据规约概述

数据规约是处理大规模数据集的关键技术,它通过简化、压缩、概括和抽象等操作,有效减少数据量,同时提升数据的质量和可解释性。这一技术在不损害原始数据完整性的前提下,通过提炼数据的核心信息,增强了数据的可理解性,并确保了数据分析结果与使用完整数据集所得结果的高度一致性。

数据规约的核心目标是在保留关键信息的前提下,最大限度地减少数据冗余,优化数据分析和处理的效率。数据规约通过提升数据质量和精度、降低数据量、缩短分析时间、降低分析难度,从而提高数据的可重复性和可扩展性,为后续的数据挖掘和机器学习任务打下坚实的基础。

数据规约的方法主要包括 3 种:维度规约、数量规约和数据压缩。维度规约通过特征选择或特征提取,减少数据的维度,常用的方法有主成分分析(PCA)、线性判别分析(LDA)、小波变换等。数量规约通过参数或非参数方法,用更小的数据表示形式替换原始数据,如回归和对数-线性模型、直方图、聚类等。数据压缩通过变换,如小波变换,得到原始数据的压缩表示,分为无损压缩和有损压缩两种。

下面将深入探讨这 3 种数据规约方法的具体内容和应用案例,以便更好地理解其在数据处理中的作用和效果。

3.7.2 维度规约

维度规约(Dimensionality Reduction)是一种有效的数据预处理技术,旨在通过降低数据集中特征的数量来简化数据集。这种方法通过减少数据的维度,不仅简化了模型的复杂度,还提高了数据挖掘的效率,并降低了计算成本。维度规约主要可以分为两大类:特征选择和特征提取。

(1) 特征选择通过从原始特征集中筛选出一个子集,这个子集包含最具代表性和信息量的特征,从而提升模型的性能。这种方法特别适用于处理高维数据,尤其是当数据集中的特征存在冗余时,特征选择能够有效地剔除不相关或冗余的特征,保留最有价值的特征。

(2) 特征提取通过创建新的特征——这些新的特征是原始特征的某种组合(如线性组合),来达到降低维度的目的。特征提取通常用于数据降维,尤其适用于原始特征之间相关性较低或包含较多噪声的情况。特征提取不仅减少了数据的维度,还尝试保留了原始数据的主要信息。

维度规约通过特征选择和特征提取两种主要方式,有效地简化了数据集,提高了数据处理的效率和模型的预测性能。

1. 特征选择

特征选择通过从原始特征集中选择一个子集来提高模型的性能,降低计算成本。特征选择主要可以分为 3 类:过滤法(Filter Methods)、包装法(Wrapper Methods)和嵌入法

（Embedded Methods）。

1）过滤法

过滤法通过评估每个特征与目标变量之间的发散性或相关性来对特征进行评分，通过设定阈值或指定待选择特征的数量来进行特征筛选。这种方法独立于学习算法，通过评估每个特征与目标变量之间的关系来选择特征。评估标准可以是相关性系数、信息增益、卡方检验等。过滤法的优点是计算速度快、易于理解，但可能不会考虑特征之间的相互作用。下面主要介绍相关系数法和方差选择法。

（1）相关系数法。相关系数法主要是要计算出各个维度对目标值的相关系数和 p 值，根据相关系数的大小来判断两者的密切程度，如果某一特征与目标值之间线性相关性非常高，就可以选择这一维度作为特征。皮尔森相关系数是一种最常用的方法，其适用场景是呈正态分布的连续变量。相关系数较低有时可能是由异常值引起的，因此，在数据清洗阶段要处理好每个维度，以排除异常值的影响。

在 Python 当中，sklearn.feature_selection 库中的 SelectKBest 方法可以根据给定的选择器，选择出前 k 个与标签最相关的特征，参数主要是提供选择器和设置 k 的值。因此，可以选择先使用 Scipy 库中的 pearsonr 方法计算出相关系数矩阵，再使用 SelectKBest 类实现特征选择。另外，也可以使用 sklearn.feature_selection 库中的 f_regression 方法进行单变量线性回归测试，并结合 SelectKBest 类来实现特征选择。这两种方法均能有效识别出对模型预测能力贡献最大的特征。

例 3-78 用相关系数法对糖尿病数据集进行特征选择。

首先，导入糖尿病数据集，使用 iloc()方法将糖尿病数据集分割成特征矩阵 X 和目标向量 Y。X 包含 8 个特征变量，Y 包含一个二元分类变量。

```
In:
    from sklearn.feature_selection import f_regression
    from sklearn.feature_selection import SelectKBest
    # 读取数据
    path = "data/diabetes.csv"
    data = pd.read_csv(path)
    # 分离特征和目标变量
    X = data.iloc[:,:-1]
    Y = data['Outcome']
    X.head(5)
```

前 5 行糖尿病数据如表 3-16 所示。

表 3-16 前 5 行糖尿病数据

序号	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

创建 SelectKBest 对象，进行特征选择，代码如下：

```
In:
    # 创建 SelectKBest 对象,使用 f_regression 作为评分函数,并设置选择 5 个最佳特征
    skb = SelectKBest(f_regression, k=5)
    # 使用 fit_transform 方法选择最佳特征并转换特征集
```

```
    bestFeature = skb.fit_transform(X, Y)
    # 获取被选择的特征的索引
    selected_feature_indices = skb.get_support(indices = True)
    # 查看被选择的特征名称
    selected_features = X.columns[skb.get_support()]
    # 打印原始特征名称和被选择的特征索引
    print('原数据特征:', X.columns)
    print('筛选后特征序号: ',selected_feature_indices)
    print('筛选后特征: ',selected_features)
Out:
    原数据特征: Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI',
'DiabetesPedigreeFunction', 'Age'],
          dtype = 'object')
    筛选后特征序号: [0 1 5 6 7]
    筛选后特征: Index(['Pregnancies', 'Glucose', 'BMI', 'DiabetesPedigreeFunction', 'Age'], dtype =
'object')
```

在例3-78中,通过调用fit_transform方法对特征数据集X和目标向量Y进行了拟合,得到了经过筛选的特征矩阵。然后,使用get_support()方法来确定被选中的特征列的索引。根据输出结果,从原始的8个特征中筛选出了5个与目标变量最相关的特征。

(2) 方差选择法。方差选择法通过计算各特征的方差来筛选属性。具体操作是设定一个阈值,仅保留那些方差超过这一阈值的特征,从而剔除那些携带较少有用信息的属性。这种方法操作简便、处理迅速,但需要注意的是,方差的大小并不直接等同于属性与目标变量之间的相关性。因此,这种方法在筛选过程中可能忽视了目标变量的具体需求,导致未能充分关注属性与目标变量之间的关系。

在Python中,feature_selection库中的VarianceThreshold方法可以实现方差选择法。它的参数只有threshold,用于得到设置的阈值。如果不填,方法默认去除方差为0的属性。

例3-79 使用方差选择法对糖尿病数据集进行特征选择。

```
In:
    from sklearn.feature_selection import VarianceThreshold
    import numpy as np
    # 读取数据
    path = "data/diabetes.csv"
    data = pd.read_csv(path)
    #分离特征和目标变量
    X = data.iloc[:, :-1]
    Y = data.iloc[:, -1]
    # 计算每个特征的方差
    variances = X.var()
    # 选择方差的中位数作为阈值
    threshold = np.percentile(variances, 50)
    # 创建VarianceThreshold对象并拟合数据
    vt = VarianceThreshold(threshold = threshold)
    X_selected = vt.fit_transform(X)
    # 获取筛选后的特征名称
    selected_features = X.columns[vt.get_support()]
    # 打印原始特征名称
    print('原数据特征:', X.columns.tolist())
    # 打印筛选后的特征名称和数据集形状
    print('筛选后的特征名称:')
    print(selected_features.tolist())
    print('\n筛选后的数据集形状:', X_selected.shape)
```

```
Out:
    原数据特征：['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI',
'DiabetesPedigreeFunction', 'Age']
    筛选后的特征名称：
    ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin']
筛选后的数据集形状：(768, 4)
```

在例 3-79 中，首先计算糖尿病数据集中每个特征的方差，并设定一个基于方差中位数的阈值，使用方差选择法进行了特征筛选。然后，通过 VarianceThreshold 函数筛选出了方差大于阈值的特征，保留了可能对模型预测能力有更重要作用的数据维度，最终得到了筛选后的特征名称及筛选后数据集的形状。

2）包装法

包装法利用学习算法本身来评估特征子集的质量。这种方法通过递归地选择和排除特征，以寻找最优的特征组合。常见的技术包括前向选择、后向消除及递归特征消除(RFE)。尽管包装法能够提供较为精确的特征选择结果，但其主要缺点在于较高的计算成本，尤其是在处理大量特征时，这一问题尤为突出。

在 sklearn 库中提供了两种递归消除特征法的方法：一种是递归特征消除（Recursive Feature Elimination，RFE），另一种是通过交叉验证的递归特征消除（Recursive Feature Elimination with Cross-Validation，RFECV）。RFE 的工作原理是利用一个学习器来评估每个特征的重要性，然后从当前的特征集合中移除最不重要的特征。这个过程会重复进行，直到达到预设的特征数量为止。RFECV 则在此基础上增加了交叉验证的过程，以自动确定最优的特征数量。如果减少特征会导致模型性能下降，那么该方法将不会去除任何特征。这两种方法都是为了帮助用户选择出对模型预测能力贡献最大的特征组合，从而简化模型、提高泛化能力和减少过拟合的风险。

RFE 的主要参数如下。

n_features_to_select：指定最终希望保留的特征数量。

estimator：作为基础的学习模型，用于评估特征的重要性。

step：每次迭代中要删除的特征数目，默认值为 1。

RFECV 的主要参数如下。

estimator：提供的监督学习估计器，用来评估特征的重要性。

step：每次迭代中要删除的特征数目，默认值为 1。

min_features_to_select：设定的最少需要保留的特征数量，默认值为 1。

cv：指定交叉验证的折数，用于评估不同特征子集下的模型性能，默认值为 5。

例 3-80 使用递归消除特征法进行特征选择。

```
In:
    from sklearn.feature_selection import RFE
    from sklearn.linear_model import LogisticRegression
    import pandas as pd
    # 读取数据文件
    path = "data/diabetes.csv"
    data = pd.read_csv(path)
    # 假设最后一列是目标变量
    X = data.iloc[:,:-1]
    Y = data.iloc[:,-1]
    # 创建 RFE 对象并拟合数据
    rfe = RFE(estimator=LogisticRegression(max_iter=1000), n_features_to_select=6)
    #为 LogisticRegression 增加最大迭代次数(max_iter)
```

```
    bestFeature = rfe.fit_transform(X,Y)
    # 获取筛选后的特征名称
    selected_features = X.columns[rfe.get_support()]
    # 打印原始特征名称
    print('原数据特征:', X.columns.tolist())
    # 打印筛选后的特征名称和数据集形状
    print('筛选后的特征名称:')
    print(selected_features.tolist())
    print('\n筛选后的数据集形状:', bestFeature.shape)
Out:
    原数据特征:['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI',
'DiabetesPedigreeFunction', 'Age']
    筛选后的特征名称:
    ['Pregnancies', 'Glucose', 'BloodPressure', 'BMI', 'DiabetesPedigreeFunction', 'Age']
    筛选后的数据集形状:(768, 6)
```

在例3-80中,使用递归特征消除(RFE)方法结合逻辑回归模型来选择最重要的6个特征,通过读取CSV文件中的数据,分离特征和目标变量,然后应用RFE来拟合数据并输出所选特征的布尔掩码。

在RFE方法中,n_features_to_select参数用于指定要选择的最优特征数量。可以采用交叉验证的方式来设置n_features_to_select参数的最优值,通过将数据集分为多个子集,并在每个子集上分别进行特征选择和模型训练,然后计算平均性能指标来评估不同特征数量下的模型表现。此外,可以使用网格搜索(Grid Search)结合交叉验证,系统地遍历不同的n_features_to_select值,找到在多个验证集上均表现最佳的参数设置。最后,还需考虑到模型的复杂度和泛化能力,避免过拟合,确保所选特征数量既能提高模型性能,又不至于引入过多的噪声。

3)嵌入法

嵌入法是先使用某些机器学习的模型进行训练,得到各个特征的权值系数,根据系数从大到小选择,主要是采用带惩罚项的特征选择方法。可以使用feature_selection库中的SelectFromModel类实现,其中主要参数是estimator,为模型评估器。例如,逻辑回归就带有L1和L2惩罚项,线性支持向量机带有L2惩罚项。可以用L1正则项来选择特征,但L1没有选到的特征不代表不重要,原因是两个具有高相关性的特征可能只保留了一个。如想要进一步确定哪个特征重要,可以再通过L2正则项交叉检验得到。

例3-81 带惩罚项的特征选择法实例。

```
In:
    from sklearn.feature_selection import SelectFromModel
    from sklearn.linear_model import LogisticRegression
    # 读取数据
    path = "data/diabetes.csv"
    data = pd.read_csv(path)
    # 分离特征和目标变量
    X = data.iloc[:, :-1]
    Y = data.iloc[:, -1]
    # 创建Logistic回归模型
    logreg = LogisticRegression(penalty="l1", C=0.1, solver='liblinear', max_iter=1000)
    # 使用SelectFromModel进行特征选择
    sfm = SelectFromModel(logreg, threshold="1.5*mean")
    sfm.fit(X, Y)
    # 获取选择的特征
    selected_features = X.columns[sfm.get_support()]
```

```
    print('原数据特征:', X.columns)
    print('选择的特征:', selected_features)
Out:
    原数据特征: Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin','BMI',
'DiabetesPedigreeFunction', 'Age'], dtype = 'object')
    选择的特征: Index(['Pregnancies', 'BMI'], dtype = 'object')
```

在例 3-81 中，使用 SelectFromModel 对糖尿病数据集进行特征选择。首先使用 LogisticRegression 创建一个 L2 正则化的 Logistic 回归模型，指定 solver='liblinear'和 max_iter=1000 以确保模型能够收敛。再使用 SelectFromModel 对象根据 Logistic 回归模型的重要性权重来选择特征，其中的 threshold 参数通过设定重要性阈值(如特征系数绝对值均值 mean 的倍数)来控制保留的特征数量，阈值越高，选出的特征越少。然后，根据 get_support()方法结合特征名称获取选择的特征。最终，代码打印出原始数据特征和选择的重要特征。

2. 特征提取

特征提取是通过创建新的特征来简化数据集、减少维度并保留对模型训练最有价值的信息。这些新的特征通常是原始特征的某种组合(如线性组合)，从而改变原有的特征空间。其主要目的是在降低数据维度的同时，尽可能保留最重要的信息。特征提取特别适用于原始特征之间相关性较低或包含较多噪声的情况，能够有效提升模型的效率和性能。常用的特征提取方法包括主成分分析(Principal Component Analysis，PCA)、因子分析(Factor Analysis，FA)等。

1) 主成分分析

主成分分析是一种无监督的线性降维技术，通过识别数据中的主要变异方向，即主成分，将高维数据映射至更低维度的空间。主成分分析的核心在于最大化保留数据的方差，这样在减少数据维度的同时，能够最大限度地保留原始数据的信息。具体而言，主成分分析寻找一系列新的正交轴(即主成分)，使得数据在这些轴上的投影能够展现出最大的方差。这种方法确保了在降维之后，数据的类内变异最小化，而类间变异最大化，从而有效地保留了数据的关键特征。

主成分分析通过以下步骤实现降维。

(1) 中心化：对数据进行中心化处理。从每个特征中减去其均值，以确保数据集的平均值为零。中心化消除了特征之间的位置偏差，确保了协方差矩阵能够正确反映特征之间的相关性。

(2) 协方差矩阵：计算中心化后数据的协方差矩阵。协方差矩阵是一个统计量，描述了数据中不同特征之间的线性关系和变异性。

(3) 特征值分解：对协方差矩阵进行特征值分解，得到一组特征值和相应的特征向量。特征值量化了沿其对应特征向量方向上数据的方差大小。

(4) 选择主成分：根据特征值的大小，选择前 k 个最大的特征值对应的特征向量作为新的主成分。这些特征向量定义了一个新的空间，称为主成分空间。这些特征向量就是主成分，是数据中最重要的方向。

(5) 投影：将原始数据投影到这些主成分上，以获得降维后的数据表示。这一步实现了数据从原始高维空间到低维主成分空间的转换，同时尽可能保留数据中的信息。

主成分分析降维可以减少数据集的变量数量，同时保持数据的大部分信息，适用于数据量较大且特征之间相关性较高的情况。

sklearn.decomposition.PCA 类的主要参数如下：

```
sklearn.decomposition.PCA(n_components = None, *, copy = True, whiten = False, svd_solver =
'auto', tol = 0.0, iterated_power = 'auto', random_state = None)
```

具体参数名称、类型和含义如表 3-17 所示。

表 3-17 sklearn.decomposition.PCA 类的主要参数

参数名称	类　型	默认值	含　义
n_components	整数或浮点数	None	指定降维后的维度数量。如果为整数,则表示降维后的特征数量。如果为浮点数(0 < n_components < 1),则表示保留的方差百分比
copy	布尔值	True	如果为 True,则在执行中心化之前复制数据。如果设置为 False,则在原始数据上进行操作,这可以节省内存,但会修改输入数据
whiten	布尔值	False	如果为 True,则对降维后的数据进行白化处理,使其每个特征具有单位方差。有助于某些类型的后续处理,如独立成分分析(ICA)
svd_solver	字符串	auto	指定用于计算奇异值分解(SVD)的方法。auto:自动选择最佳方法。full:使用传统的全 SVD 分解(适用于 n_components 较小的情况)。arpack:使用 ARPACK 包实现的 SVD(适用于 n_components 较大的情况)。randomized:使用随机 SVD(适用于大数据集)
tol	浮点数	0.0	当 svd_solver='arpack' 时,用于控制收敛条件的容差。较小的 tol 值会导致更精确的结果,但计算时间可能更长
iterated_power	整数或 auto	auto	当 svd_solver='randomized'时,指定随机 SVD 的迭代次数。auto 表示根据数据大小自动选择合适的迭代次数
random_state	整数、随机状态实例或 None	None	控制随机数生成器的种子,确保结果的可重复性。对于 svd_solver='randomized'和 svd_solver='arpack'都有效

例 3-82 心脏病数据集的 PCA 实例。

```
In:
    import matplotlib.pyplot as plt
    from sklearn.decomposition import PCA
    from sklearn.preprocessing import StandardScaler
    import seaborn as sns
    # 加载心脏病数据集
    path = "data/heart-disease.csv"
    column_names = [
        'age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
        'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target'
    ]
    path = "data/heart-disease.csv"
    data = pd.read_csv(path, names=column_names)
    # 处理缺失值
    data.replace('?', np.nan, inplace=True)
    data.dropna(inplace=True)
    # 将非数值特征转换为数值
    data['ca'] = data['ca'].astype(float)
    data['thal'] = data['thal'].astype(float)
    # 分离特征和目标
    X = data.drop('target', axis=1)
    y = data['target']
    # 标准化数据
    scaler = StandardScaler()
```

```
    X_scaled = scaler.fit_transform(X)
    # 创建 PCA 对象,并指定降维后的维度为 2
    pca = PCA(n_components=2, random_state=42)
    # 对数据进行拟合并转换
    X_pca = pca.fit_transform(X_scaled)
    # 可视化结果
    plt.figure(figsize=(8, 6))
    colors = ['blue', 'red']
    labels = ['No Heart Disease', 'Heart Disease']
    for i, target_name in enumerate(labels):
        plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], color=colors[i], alpha=0.7, label=
target_name)
    plt.legend(loc='best', shadow=False, scatterpoints=1)
    plt.title('PCA of Heart Disease Dataset')
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.show()
    # 输出解释方差比
    explained_variance_ratio = pca.explained_variance_ratio_
    print(f'Explained Variance Ratio: {explained_variance_ratio}')
Out:
    Explained Variance Ratio: [0.23695056 0.12349486]
```

心脏病数据集主成分分析结果如图 3-20 所示。

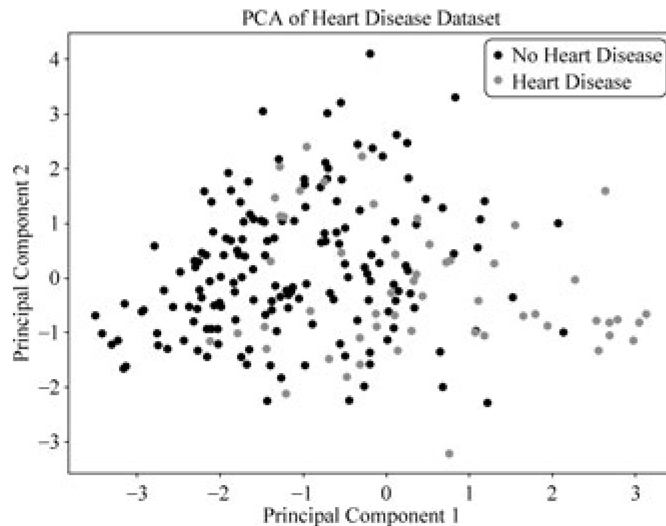


图 3-20 心脏病数据集主成分分析结果

在例 3-82 中,使用主成分分析对 UCI 的心脏病数据集(Heart Disease Dataset)进行降维,该数据集包括年龄、性别、胸痛类型、静息血压、血清胆固醇等多个与心脏病相关的特征,用于预测患者是否患有心脏病,可以从 https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data 进行数据下载。首先,加载并预处理数据,包括处理缺失值和使用 StandardScaler 进行特征标准化。接着,创建 PCA 对象并指定降维后的维度为 2,然后使用 fit_transform 方法将数据降至二维。最后,通过散点图可视化降维后的数据,并输出每个主成分的解释方差比,以评估信息保留情况。

2)因子分析

因子分析是一种统计方法,用于从一组相关变量中提取出较少的不相关变量,这些新变量被称为“因子”。因子分析在数据规约的特征提取中非常有用,因为它可以将原始的高维数据

转换为更低维度的表示，同时保留大部分原始信息。这有助于简化模型，提高解释性和可解释性，并在某些情况下改善预测性能。因子分析通常用于心理学、社会学、经济学和生物信息学等领域。

因子分析的基本思想是将每个原始变量表示为几个不可观测的潜在因子的线性组合。通过估计这些因子的权重（即因子载荷），可以捕捉到原始变量之间的相关性。然后，可以选择最有影响力的因子来代表原始数据，从而达到降低维度和简化问题的目的。在进行因子分析时，需要先收集一批样本数据，并对这些数据进行标准化处理。然后，使用某种算法（如主成分分析或最大似然估计）来估计因子载荷和 uniqueness（特殊性）。最后，根据因子的重要性对它们进行排序，只保留最重要的几个因子作为新的特征。

因子分析不仅可以帮助我们理解数据的内部结构，还可以用来检测异常值和处理缺失数据。此外，因子分析还可以与其他机器学习技术相结合，如聚类分析和回归分析，以进一步提高模型的准确性和可靠性。

sklearn.decomposition.FactorAnalysis 类的主要参数如下：

```
sklearn.decomposition.FactorAnalysis(n_components = None, *, tol = 0.01, copy = True, max_iter =
1000, noise_variance_init = None, svd_method = 'randomized', iterated_power = 3, random_state = 0)
```

具体参数名称和说明如表 3-18 所示。

表 3-18 sklearn.decomposition. FactorAnalysis 类的主要参数

参数名称	参数说明
n_components	设置要保留的因子个数，默认为 None（即保留所有特征，不降维）。实际应用中需手动指定以实现降维（如 n_components＝3）
tol	设置收敛容忍度，用于控制 EM 算法停止迭代的阈值
copy	控制是否复制输入数据 X，若为 True，则复制；否则在原始数据上操作
max_iter	设置 EM 算法的最大迭代次数，默认为 1000
noise_variance_init	初始化噪声方差，默认为 None（自动计算）
svd_method	指定 SVD（奇异值分解）方法，默认为 'randomized'
iterated_power	在随机 SVD 中使用的幂次迭代次数，用于加速计算，默认为 3
random_state	设置随机种子，使结果具有可重复性

例 3-83 糖尿病数据集的因子分析实例。

```
In:
    import pandas as pd
    from sklearn.decomposition import FactorAnalysis
    import matplotlib.pyplot as plt
    from sklearn.preprocessing import StandardScaler
    # 加载数据
    data = pd.read_csv('data/diabetes.csv')
    # 选择因子分析的列(假设除了 Outcome 列的所有列)
    columns = data.columns[:-1]                # 排除最后一列 Outcome
    X = data[columns]
    # 数据标准化
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    # 初始化因子分析对象,设置因子个数为 5(这个数值可能需要根据结果进行调整)
    fa = FactorAnalysis(n_components = 5, rotation = 'varimax')
    # 拟合数据
    fa.fit(X_scaled)
    # 获取因子载荷矩阵
    loadings = fa.components_
    print("因子载荷矩阵:")
```

```
    print(loadings)
    # 获取因子方差,以决定因子个数
    variance = fa.noise_variance_
    print("\n因子方差:")
    print(variance)
    # 可视化因子载荷矩阵
    plt.figure(figsize=(8, 6),dpi=300)
    plt.scatter(range(1, len(columns) + 1), variance)
    plt.plot(range(1, len(columns) + 1), variance)
    plt.title('Scree Plot')
    plt.xlabel('Factors')
    plt.ylabel('Eigenvalue')
    plt.grid()
    plt.show()
output:
    因子载荷矩阵:
    [[-0.04003498  0.07750925  0.15288018  0.74617219  0.62161366  0.27925932
       0.21098436 -0.08238436]
     [ 0.70676741  0.18856865  0.22617763 -0.07760309 -0.07247002  0.02146476
       0.00514206  0.77156285]
     [-0.00275255  0.71512212  0.06519577 -0.03522763  0.42481135  0.15028549
       0.15436705  0.16512707]
     [-0.0221748  -0.12909221 -0.25130941 -0.28403436  0.05784923 -0.63867506
      -0.09511995 -0.00866549]
     [ 0.03201174 -0.03821956 -0.50479726 -0.08232926  0.00912973 -0.12310513
       0.02305798 -0.12488437]]
    因子方差:
    [0.49631691 0.42952916 0.6050389  0.34877737 0.42347812 0.47651227
     0.9221105  0.35555515]
```

因子载荷矩阵可视化如图 3-21 所示。

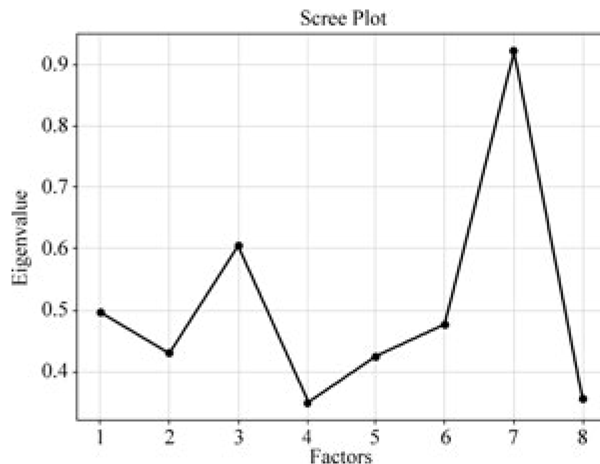


图 3-21 因子载荷矩阵可视化

例 3-83 首先加载 diabetes.csv 文件,然后选择除 Outcome 外的所有列作为因子分析的输入。数据被标准化处理,以确保每个特征的均值为 0,标准差为 1。然后,使用 FactorAnalysis 类进行因子分析,设置因子个数和旋转方法。最后,代码打印出因子载荷矩阵和因子方差,并绘制了一个 Scree Plot 来帮助决定保留多少个因子。

注意,因子分析的结果和解释需要结合具体的数据和业务背景来进行,因子载荷矩阵和因子方差提供了数据降维后的新特征表示,可以用于后续的数据分析或机器学习任务。

综上,主成分分析是一种通过线性组合将原始变量转换为一组不相关的主成分,以最大化数据总方差的方式来降维的技术,它不涉及任何潜在的理论模型,仅依赖于数据本身的协方差

结构。因子分析则是一种基于潜在变量模型的统计方法,它假设观察到的变量是由几个不可观测的潜在因子和独特误差共同作用的结果,旨在揭示这些潜在因子并理解它们与观察变量之间的关系。

3.7.3 数量规约

数量规约(Numerical Reduction)是数据规约的一种形式,它通过选择或构建更少的数据表示来减少数据量。这种方法特别适用于数据集非常大且数据点之间存在冗余或相似性的情况。数量规约的目标是在不显著损失信息的前提下,减少数据的规模,从而提高数据处理的效率和可管理性。常用的数据规约方法包括抽样和聚类。抽样是从原始数据集中选择一个代表性的子集;聚类是将数据点分组,使得同一组内的数据点彼此相似,而不同组之间的数据点差异较大。

常用的数量规约抽样方法有如下 4 种:①随机抽样,即每个个体被独立地随机选中。②分层抽样,用于将数据集分成不同的层,然后从每层中随机选择样本,以确保样本的代表性。③整群抽样,将数据集分成若干群组,然后随机选择整个群组作为样本。④系统抽样,按照固定的间隔从数据集中选择样本,这种方法适用于数据集有序排列的情况。

常用的数量规约聚类方法有如下两种:①K-Means 聚类,将数据点分组成 K 个簇的算法,使得簇内的数据点尽可能相似,簇间的数据点尽可能不同。②层次聚类,通过创建一个聚类树来组织数据点,可以是凝聚的(从单个数据点开始,逐步合并)或分裂的(从所有数据点开始,逐步分割)。

这些数量规约方法能够有效降低数据处理的计算成本和存储需求,提高数据分析的效率。在实际应用中,选择哪种方法取决于数据的特点、分析的目标及对信息损失的容忍度。下面结合实例对常用方法进行介绍。

1. 随机抽样

随机抽样是一种确保每个个体都有相同机会被选中的统计方法,通过这种无偏的样本选择方式,研究者能够从样本数据中推断出总体的特征,提高研究的代表性和准确性。随机抽样是科学研究和市场调查中常用的方法,有助于确保研究结果的普遍性和可靠性。

随机抽样的步骤通常包括确定总体范围、决定样本大小、选择适当的抽样框、应用随机化技术(如抽签、随机数表或计算机生成随机数)来选取样本单位,并最终执行抽样,确保每个单位都有非零的入选概率,以便收集数据进行分析。

在 Pandas 中,DataFrame.sample 方法可用于随机抽取样本,主要参数及说明如表 3-19 所示。

```
DataFrame.sample(self, n = None, frac = None, replace = False, weights = None, random_state = None,
axis = None)
```

表 3-19 DataFrame.sample 的主要参数及说明

参　数	参 数 说 明
self	DataFrame 对象本身
n	整数或无,可选。要抽取的行数或列数(取决于 axis 参数)
frac	浮点数或无,可选。要抽取的行或列的比例
replace	布尔值,默认为 False。是否允许抽样时替换,即是否允许重复抽取相同的行或列
weights	字符串或类似数组的对象,可选。如果指定,则必须是 DataFrame 的一个列名,或者是与 DataFrame 行数或列数相同的数组,用于给每行或每列分配一个抽样权重
random_state	整数或 np.random.RandomState 实例,可选。用于指定随机数生成器的种子
axis	整数或字符串,默认为 0(index)。0 或 index 表示按行抽样,1 或 columns 表示按列抽样

例 3-84 数据随机抽样实例。

```
In:
    # 创建一个示例 DataFrame
    data = {
        'A': range(1, 6),
        'B': range(10, 15),
        'C': range(100, 105)
    }
    df = pd.DataFrame(data)
    # 设置 sample 方法的参数
    sample_n = df.sample(n=3)                              # 抽取 3 行
    sample_frac = df.sample(frac=0.5)                      # 抽取 50% 的行
    weights = [0.1, 0.2, 0.3, 0.4, 0.5]                    # 设置权重
    sample_weights = df.sample(n=3, weights=weights)  # 使用权重抽取 3 行
    sample_n, sample_frac, sample_weights
Out:
    (   A   B    C
     1  2  11  101
     3  4  13  103
     0  1  10  100,
        A   B    C
     2  3  12  102
     3  4  13  103,
        A   B    C
     4  5  14  104
     3  4  13  103
     1  2  11  101)
```

例 3-84 演示了使用 Pandas 库中的 DataFrame. sample 方法来抽取样本。首先，使用 n＝3 参数，从原始数据帧中随机抽取了 3 行数据，这是无放回抽样，即抽取的行不会再次被选中。然后，使用 frac＝0.5 参数，这个参数指定了抽取的行数占原始数据帧总行数的比例。在这个例子中，由于原始数据帧有 5 行，所以抽取了 50％，即 2.5 行，实际操作中会四舍五入到最近的整数，因此，抽取了 2 行。最后，使用 weights 参数和 n＝3，这个参数允许用户为每行指定一个权重，从而进行加权抽样。在这个例子中，每行的权重分别为[0.1,0.2,0.3,0.4,0.5]，然后根据这些权重随机抽取了 3 行，权重较高的行有更高的概率被选中。

随机抽样可用于从大型数据集中抽取代表性样本进行进一步分析或建模。通过调整 n、frac 和 weights 参数，可以灵活地控制抽样过程以满足不同的需求。

2. 分层抽样

分层抽样是一种精细的概率抽样技术，它首先将总体划分为若干子群体，这些子群体在某个关键特征上具有一致性或相似性。然后，从每个子群体中独立地进行随机抽样，确保每个子群体在最终样本中得到充分的代表。这种方法的优势是能够保证样本在关键特征上的均衡分布，从而显著增强样本的多样性和代表性。

分层抽样特别适合于那些需要精确反映总体结构的研究，尤其是在总体中存在具有明显差异的群体时。通过分层抽样，研究者能够更有效地捕捉到这些群体间的差异，从而提高研究结果的准确性和可靠性。这种方法在社会科学研究、市场调查、医学研究等领域尤为有用，因为它有助于确保样本能够代表总体中的不同亚群体，使得研究结果更加全面和精确。

例 3-85 数据分层抽样实例。

```
In:
    # 1.数据集包含患者年龄和一些健康指标
    np.random.seed(0)
    data = {
        'Age': np.random.randint(0, 100, size=1000),
        'BloodPressure': np.random.randint(90, 160, size=1000),
        'Cholesterol': np.random.randint(120, 300, size=1000)
    }
    df = pd.DataFrame(data)
    # 2. 定义分层
    # 根据年龄将患者分为不同的层
    age_bins = [0, 18, 30, 50, 100]
    age_labels = ['0-18', '19-30', '31-50', '50+']
    df['AgeGroup'] = pd.cut(df['Age'], bins=age_bins, labels=age_labels, right=False)
    # 3. 进行分层抽样
    # 每个层中抽取10%的样本
    sample_fraction = 0.1
    stratified_sample = df.groupby('AgeGroup', group_keys=False).apply(lambda x: x.sample
(frac=sample_fraction))
    # 4. 打印抽样结果
    print("原始数据集形状:", df.shape)
    print("分层抽样后的数据集形状:", stratified_sample.shape)
    print("\n分层抽样后的数据集:")
    print(stratified_sample.head())
    # 5. 验证每个层的样本数量
    print("\n每个层的样本数量:")
    print(stratified_sample['AgeGroup'].value_counts())
Out:
    原始数据集形状:(1000, 4)
    分层抽样后的数据集形状:(101, 4)
    分层抽样后的数据集:
         Age  BloodPressure  Cholesterol  AgeGroup
    676   12            124          188      0-18
    831    9            112          160      0-18
    461    9            117          204      0-18
    535    5             94          139      0-18
    5      9            128          283      0-18
    每个层的样本数量:
    50+        50
    31-50      21
    0-18       18
    19-30      12
    Name: AgeGroup, dtype: int64
```

在例3-85中，首先生成了一个包含1000名患者的示例数据集，每个患者有年龄、血压和胆固醇水平3个特征。然后，使用pd.cut函数根据年龄将患者分为4个层：0～18岁、19～30岁、31～50岁和50岁以上。再通过groupby和apply方法对每个层进行抽样，从每个层中抽取10%的样本，其中lambda x：x.sample(frac=sample_fraction)是一个匿名函数，用于从每个组中抽取指定比例的样本。通过分层抽样，可以确保每个年龄段的患者在样本中都有代表性，从而提高分析的准确性和可靠性。分层抽样后的数据可以用于进一步的统计分析、建模或研究，有助于更好地理解和预测不同年龄段患者的健康状况。

3. 整群抽样

整群抽样是一种高效的数据采集策略，它首先根据特定属性将整个数据集划分为若干群体或簇，然后随机选取一定数量的簇，并将所选簇内的所有数据纳入抽样数据子集中。这种方

法特别适用于总体分布广泛、个体差异较大的情况，尤其是在进行大规模调查或基于地理区域划分的研究时。整群抽样的优势在于操作简便和成本效益高，但在群体内部个体相似性较高的情况下，可能会引入抽样误差，因此，需要仔细选择群体和确定样本量，以确保研究结果的准确性和可靠性。

整群抽样特别适用于群体间差异较大，而群体内个体间差异较小的情况，例如，在医学研究中，可能需要评估某种干预措施在不同医院的效果，这时可以将医院作为群体单位，随机抽取几家医院进行深入研究。

例 3-86 数据整群抽样实例。

```
In:
    import random
    # 假设有以下医院和患者数据
    data = {
        'Hospital': ['A', 'A', 'A', 'B', 'B', 'B', 'B', 'C', 'C', 'C'],
        'Patient': ['P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7', 'P8', 'P9', 'P10'],
        'Health Metric': [10, 12, 11, 9, 8, 7, 6, 5, 4, 3]  # 某个健康指标
    }
    # 将数据转换为 DataFrame
    df = pd.DataFrame(data)
    # 确定要抽取的群体数量
    # 先将唯一医院名称转换为列表
    unique_hospitals = df['Hospital'].unique().tolist()
    # 随机选择两家医院
    sample_hospitals = random.sample(unique_hospitals, 2)
    # 从选中的群体中抽取所有患者数据
    sample_data = df[df['Hospital'].isin(sample_hospitals)]
    print(sample_data)
Out:
          Hospital Patient  Health Metric
    0         A       P1              10
    1         A       P2              12
    2         A       P3              11
    7         C       P8               5
    8         C       P9               4
    9         C      P10               3
```

在例 3-86 中，对一个包含 3 家医院患者健康数据的数据集进行整群抽样。使用整群抽样，从这些医院中随机选取两家，并提取了这两家医院所有患者的健康指标数据，以便进行进一步的分析。这种方法适用于评估如医院间健康干预措施的效果等群体性研究。

4. 系统抽样

系统抽样也称为等距抽样，是一种概率抽样技术，其中总体被分为等距的间隔，然后按照固定的间隔从总体中抽取样本。这种方法在实施上比较简单，且能保证样本在总体中均匀分布。

系统抽样的关键步骤如下。

（1）确定抽样间隔：将总体大小除以所需的样本大小来计算间隔。

（2）随机选择起点：在间隔的第一个单位内随机选择一个起点。

（3）按照间隔选择样本：从起点开始，每隔一个间隔选择一个样本，直到达到所需的样本大小。

系统抽样的主要优势是简单性和高效性，特别适合于总体较大且有序的情况。然而，这种

方法假设总体中没有周期性的变化，否则，可能会导致样本选择上的偏差。例如，如果总体中的个体按照某种周期性模式排列，而抽样间隔恰好与这个模式相匹配，那么样本可能无法代表整个总体。因此，在实施系统抽样时，需要确保总体的分布是随机的，或者抽样间隔不会与总体中的任何周期性模式相吻合。

例 3-87　数据系统抽样实例。

```
In:
    # 假设有一份大型医院的门诊患者数据集，包含患者 ID 和他们的就诊日期
    data = {
        'PatientID': np.arange(1, 10001),                          # 患者 ID 从 1 到 10000
        'VisitDate': pd.date_range('20230101', periods = 10000)    # 假设的就诊日期
    }
    # 将数据转换为 DataFrame
    df = pd.DataFrame(data)
    # 确定抽样间隔
    sampling_interval = len(df)    // 100   # 从 10000 个患者中抽取 100 个样本
    # 随机选择一个起点
    start_index = np.random.randint(0, sampling_interval)
    # 使用系统抽样选择样本
    sampled_indices = np.arange(start_index, len(df), sampling_interval)
    # 选择样本
    sampled_data = df.iloc[sampled_indices]
    # 打印样本数据
    print(sampled_data)
Out:
    PatientID  VisitDate
    10           11 2023 - 01 - 11
    110         111 2023 - 04 - 21
    ...         ...         ...
    9810       9811 2049 - 11 - 10
    9910       9911 2050 - 02 - 18
```

在例 3-87 中，首先创建了一个包含 10000 个假设患者的数据集，每个患者都有一个唯一的 ID 和就诊日期。然后，计算了抽样间隔，这是通过将患者总数除以想要的样本大小来得到的。再随机选择一个起点，并每隔一定的间隔（该例中是 100）选择一个患者。最后，使用 Pandas 的 iloc()函数根据这些索引来选择样本，并打印出了样本数据。

5. 聚类规约

聚类分析也是一种数量规约，方法，它通过将数据集中的对象分组到较少的簇中，从而减少数据的规模，每个簇由具有相似特征的多个对象组成，这样就可以用簇的质心或代表点来代替簇内的所有数据点，达到减少数据量的目的，同时保留数据集中的主要结构和信息。

通过聚类对数据进行规约，不仅简化了数据分析过程，还降低了计算成本，适用于大规模数据集的处理和分析。在健康和医学领域，聚类可以帮助识别患者群体中的典型模式，例如，根据患者的年龄、血压和胆固醇水平等特征进行分组，进而选择每个组中的代表性患者，使得后续的研究和分析更加高效和有针对性。

例 3-88　K-means 聚类数量规约实例。

```
In:
    from sklearn.preprocessing import StandardScaler
    from sklearn.cluster import KMeans
    # 1. 生成包含患者年龄、血压和胆固醇水平的数据集
    np.random.seed(0)
```

```
    data = {
        'PatientID': np.arange(1, 1001),
        'Age': np.random.randint(0, 100, size=1000),
        'BloodPressure': np.random.randint(90, 160, size=1000),
        'Cholesterol': np.random.randint(120, 300, size=1000)
    }
    df = pd.DataFrame(data)
    # 2. 标准化数据
    features = df[['Age', 'BloodPressure', 'Cholesterol']]
    scaler = StandardScaler()
    scaled_features = scaler.fit_transform(features)
    # 3. 进行 K-Means 聚类,假设需要将数据分为 3 个簇
    num_clusters = 3
    kmeans = KMeans(n_clusters=num_clusters, random_state=0)
    kmeans.fit(scaled_features)
    df['Cluster'] = kmeans.labels_
    # 4. 从每个簇中选择代表性样本,选择每个簇的质心作为代表性样本
    centroids = kmeans.cluster_centers_
    representative_samples = []
    for i in range(num_clusters):
        # 找到最接近质心的样本
        closest_sample_index = np.argmin(np.linalg.norm(scaled_features - centroids[i], axis=1))
        representative_samples.append(df.iloc[closest_sample_index])
    representative_df = pd.DataFrame(representative_samples)
    # 5. 打印抽样结果
    print("原始数据集形状:", df.shape)
    print("聚类后的数据集形状:", representative_df.shape)
    print("\n聚类后的代表性样本:")
    print(representative_df)
    # 6. 验证每个簇的代表性样本
    print("\n每个簇的代表性样本:")
    print(representative_df[['PatientID', 'Age', 'BloodPressure', 'Cholesterol', 'Cluster']])
Out:
    原始数据集形状: (1000, 5)
    聚类后的数据集形状: (3, 5)
    聚类后的代表性样本:
         PatientID  Age  BloodPressure  Cholesterol  Cluster
    851        852  40             148          215        0
    249        250  39             109          171        1
    407        408  65             116          253        2
    每个簇的代表性样本:
         PatientID  Age  BloodPressure  Cholesterol  Cluster
    851        852  40             148          215        0
    249        250  39             109          171        1
    407        408  65             116          253        2
```

在例 3-88 中,生成了一个包含 1000 名患者的示例数据集,每个患者有患者 ID、年龄、血压和胆固醇水平 4 个特征。首先,使用 StandardScaler 对数据进行标准化处理,以确保各个特征在相同的尺度上。然后,使用 K-means 算法将数据分为 3 个簇,并将聚类结果存储在数据框中,增加一列 Cluster,表示每个样本所属的簇。接着,从每个簇中选择最接近质心的样本作为代表性样本。最后,打印原始数据集和聚类后的代表性样本的形状及详细信息。

通过聚类方法,可以将大量相似的患者数据归并到少数几个代表性样本中,从而减少数据量,简化后续分析,并有助于更好地理解和预测患者的健康状况。

3.7.4 数据压缩

数据压缩是数据规约的一种方法,它利用压缩算法减少数据的存储空间和传输带宽,同时

尽可能保留数据的重要信息。常见的数据压缩技术包括有损压缩(如 JPEG 图像压缩、MP3 音频压缩)和无损压缩(如 ZIP、GZIP)。

无损压缩是指在压缩过程中不会丢失任何原始数据的信息,解压缩后可以完全恢复原始数据。常见的无损压缩算法包括 ZIP、GZIP、LZ77/LZ78 系列(如 LZW 用于 GIF 图像)和 Huffman 编码。无损压缩广泛应用于需要精确还原的数据,如电子病历、基因序列和财务记录。

有损压缩在压缩过程中会丢失一些原始数据的信息,但这些丢失的信息对数据的主要特征影响较小,解压缩后的数据与原始数据非常接近,但不完全相同。常见的有损压缩算法包括 JPEG(用于图像压缩)、MP3(用于音频压缩)和 MPEG(用于视频压缩)。有损压缩通常用于可以容忍一定程度信息损失的数据,如医学图像、心电图和远程医疗中的大量图像或视频传输。

在健康和医学领域,无损压缩常用于保存和传输电子病历、基因序列等关键数据,确保其完整性和准确性。有损压缩适用于医学图像(如 X 光片、CT 扫描、MRI 图像),通过去除高频成分来减少数据量,同时保持足够的诊断质量。例如,使用 JPEG 压缩医学图像,通过调整压缩参数平衡图像质量和压缩率,可以显著减少文件大小,提高存储效率和传输速度,同时确保关键信息的保留和可用性。这种方法不仅简化了数据管理和分析过程,还降低了计算资源的需求。

例 3-89 使用 zlib 进行字符串数据压缩。

```
In:
    import zlib
    import base64
    # 假设有一个大型的字符串数据,如重复的日志记录
    large_string_data = "Log entry 123: This is a log message." * 1000
    # 压缩前的数据大小
    original_size = len(large_string_data)
    print(f"Original data size: {original_size} bytes")
    # 使用 zlib 压缩数据
    compressed_data = zlib.compress(large_string_data.encode('utf-8'))
    # 压缩后的数据大小
    compressed_size = len(compressed_data)
    print(f"Compressed data size: {compressed_size} bytes")
    # 为了便于展示,将压缩后的数据编码为 base64 字符串
    encoded_compressed_data = base64.b64encode(compressed_data).decode('utf-8')
    # 打印压缩后的数据(以 base64 编码的形式)
    print(f"Compressed data (base64 encoded): {encoded_compressed_data[:100]}...")
    # 解压缩数据以验证完整性
    decompressed_data = zlib.decompress(compressed_data).decode('utf-8')
    # 验证解压缩后的数据是否与原始数据相同
    assert decompressed_data == large_string_data
    print("Decompression successful, data integrity verified.")
Out:
    Original data size: 37000 bytes
    Compressed data size: 173 bytes
    Compressed data (base64 encoded): eJztyrsJgDAUQNFV3gSC2jmDpQukCFHwA8bG7c0Qlgdud898lcjnc7/
RD+MUy7rVaKXY2zhyrankboYgCIIgCIIgCIIgCIIgCIIg...
    Decompression successful, data integrity verified.
```

例 3-89 展示了如何使用 Python 的 zlib 库来压缩一个包含重复文本的大型字符串数据。通过压缩,可以显著减少数据存储和传输所需的空间。实例中,首先创建了一个重复的日志条目字符串,然后使用 zlib 的压缩功能对其进行压缩,并计算压缩前后的数据大小。为了便于展

示，将压缩后的数据转换为 base64 编码字符串。最后，通过解压缩并验证数据完整性，证明了压缩和解压缩过程的有效性。该例演示了如何在 Python 中进行数据压缩，并可用于处理需要减少存储空间或优化数据传输的场景。

3.7.5 案例——心脑血管数据规约

1. 数据预处理

对导入的 Excel 数据进行预处理操作，包括处理缺失值、转换为浮点数类型、填充缺失值、删除特定行和删除含空值的行，最后提取特征和标签，并输出特征的形状。

例 3-90 数据预处理。

```
In:
    import pandas as pd
    import numpy as np
    # 导入数据
    data = pd.read_excel("data/T2_HealthInfo.xlsx", header = 0)
    data = data.iloc[0:,3:]
    # 处理数据中的#NULL!
    data = data.replace("#NULL!",np.nan)
    # 数据格式转为 float 类型
    data = data.apply (pd.to_numeric, errors = 'coerce')
    # 数据预处理——缺失值处理
    cols = data.columns[:7]
    for col in cols:
        data[col].fillna(data[col].mean(), inplace = True)
    #删除"危险分层"一项的值为 yes 和 no 的数据
    data = data[~data['危险分层'].isin(['yes','no'])]
    #删除含空值的行
    data = data.dropna(axis = 0)
    data.head()
    X = data.iloc[:, : - 1]  #特征值
    y = data.iloc[:, - 1]    #类别标签
    X.shape
Out:
    (4475, 16)
```

经过数据预处理，得到了一个包含 4475 行、16 个特征的数据集。这些特征主要包括收缩压、舒张压、总胆固醇、甘油三酯、高密度脂蛋白胆固醇等。

2. 维度规约案例

使用方差阈值法对数据 X 进行特征选择，目的是去除方差过小的特征，因为这些特征对模型的预测效果帮助不大。设置方差的阈值为 0.1，表示只保留方差大于 0.1 的特征。得到了筛选后的 11 个特征名，它们是收缩压、舒张压、总胆固醇等。这些特征反映了心脑血管疾病高危人群的健康状况。将数据 X 更新为筛选后的数据集，以便于进行后续的分析和建模。

例 3-91 维度规约实例。

```
In:
    from sklearn.feature_selection import VarianceThreshold
    vt = VarianceThreshold(threshold = 0.1)     # 参数 threshold 为方差的阈值
    bestFeature = vt.fit_transform(X)
    vt_columns = X.columns[vt.get_support()]
    print(vt_columns)
```

```
    X = X[vt_columns]                    # 得到新的数据集
Out:
    Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
           'BMI', 'DiabetesPedigreeFunction', 'Age'],
          dtype='object')
```

3. 数量规约案例

使用 train_test_split()函数，按照随机划分法把数据 X 分成训练集和测试集。设置 test_size=0.2，让测试集占总数据的 20%；设置 random_state=30，保证每次划分的结果相同；设置 shuffle=True，让数据在划分前被打乱顺序。

例 3-92 数量规约实例。

```
In:
    from sklearn.model_selection import train_test_split
    X_train,X_test = train_test_split(X,test_size=0.2,random_state=30,shuffle=True)
    print('原数据维度: ',X.shape)
    print('分层后维度: ',X_train.shape)

Out:
    原数据维度: (768, 8)
    分层后维度: (614, 8)
```

4. 数据压缩案例

使用主成分分析对心血管数据集进行有损压缩。首先，使用 StandardScaler()函数对 X_train 进行标准化处理，并返回一个新的数组 X_scaled；其次，创建 PCA 对象，并指定保留两个主成分，即将原始数据降到二维空间；最后，对标准化后的数据进行拟合和转换，得到主成分矩阵 X_pca。

例 3-93 数据压缩实例。

```
In:
    # 导入所需的库
    from sklearn.preprocessing import StandardScaler
    from sklearn.decomposition import PCA
    # 标准化数据
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X_train)
    # 创建 PCA 对象,并指定保留两个主成分
    pca = PCA(n_components=2)
    # 对标准化后的数据进行拟合和转换,得到主成分矩阵
    X_pca = pca.fit_transform(X_scaled)

    print('降维后的结果: ',X_pca)
    print('方差贡献率: ',pca.explained_variance_ratio_)      #降维后的各主成分的方差值占
                                                              #总方差值的比例,即方差贡献率
    print('降维后各主成分的方差值: ',pca.explained_variance_) #降维后的各主成分的方差值

Out:
    降维后的结果: [[-0.4984698   0.2501628 ]
     [-0.58053703  1.26991359]
     [ 1.15237363  1.38663554]
     ...
     [-1.22510498  1.79222956]
     [-0.44570637 -0.69083847]
```

```
 [-0.86378322 -1.15120916]]
方差贡献率:[0.26196008 0.21428534]
降维后各主成分的方差值:[2.09909937 1.7170793 ]
```

降维后的结果：一个二维数组，每行代表一个样本，每列代表一个主成分。

方差贡献率：方差贡献率越大，说明主成分综合原始特征的信息的能力越强。方差贡献率为[0.16198577 0.14059534]，说明第一个主成分可以解释16.2%的数据方差，第二个主成分可以解释14.1%的数据方差。

降维后各主成分的方差值：降维后各主成分的方差值为[1.78234135 1.54698084]，说明第一个主成分的特征值为1.78，第二个主成分的特征值为1.55。两个特征值相差不大，说明两个主成分都有一定的影响力。

3.8 本章小结

数据预处理是提升数据质量和应用价值的关键环节，可使数据更好地满足数据挖掘算法的要求。这一过程对于提高数据分析与挖掘模型的性能和精确度，以获得准确和可靠的分析结果具有重要意义。

数据清洗是数据预处理中的关键环节，在此环节要通过一定的检测和处理方法，将原始数据清洗为质量较高的数据。

本章详细介绍了数据值清理、数据转换、数据脱敏与隐私保护等数据清洗技术，并结合心血管数据介绍了数据清洗的综合案例。

通过数据的集成、变换和规约，可以将多渠道的数据进行整合，转换数据的形式并筛选与目标有关的数据，以提高数据分析与挖掘的效率。本章结合实际案例，对数据预处理中的数据集成、数据变换和数据规约进行了详细介绍，为后续的分析挖掘工作打下了良好的基础。

扫一扫

自测题

习题 3

1. 对学生成绩数据集进行预处理。

说明：该数据集包含以下列：学号、班级、姓名、性别、C语言、计算机导论、数据库原理。

要求：

(1) 删除重复值。确保数据集中没有重复的学生记录，只考虑学号和姓名的组合来确定重复值。

(2) 补齐缺失值。对于数值型特征(如C语言、计算机导论、数据库原理成绩)，使用每列的均值进行填充。对于分类型特征(如性别)，使用所在班级的众数进行填充。对于性别列中的缺失值，使用所在班级的众数进行填充。

(3) 处理异常值。确保所有成绩都在0～100的合理范围内。对于超出这个范围的值，将其视为异常值并替换为NaN，然后重新使用均值进行填充。

(4) 添加“成绩等级”列。根据C语言、计算机导论、数据库原理3门课程的平均成绩，添加一个新的列——“成绩等级”，该列的值为“不及格”“及格”“良好”“优秀”，根据平均成绩的区间进行划分(0～60为“不及格”，60～75为“及格”，75～90为“良好”，90～100为“优秀”)。

(5) 将处理后的数据集保存为一个新的Excel文件，文件名为processed_stu_scores.xlsx。

2. 对breast_cancer.csv进行数据转换。具体要求如下：对['radius_mean','texture_

mean','perimeter_mean']列进行标准化处理,对['radius_worst','texture_worst','perimeter_worst']列进行归一化处理,对'diagnosis'列进行数据编码,显示处理后的数据前5行。

3. 对UCI机器学习库中的Adult数据集进行数据脱敏与隐私保护。

说明:在数据分析和机器学习中,保护个人隐私是非常重要的。数据脱敏是一种常用的方法,通过泛化、添加噪声等手段来减少数据中的敏感信息,同时保持数据的可用性。数据中字段包括 column_names =['age','workclass','fnlwgt','education','education-num','marital-status','occupation','relationship','race','sex','capital-gain','capital-loss','hours-per-week','native-country','income']

要求:

(1) 读取并理解adult.csv数据集。

(2) 对数据集中的敏感信息进行脱敏处理。泛化年龄:将年龄分为几个区间(如0～19、20～39、40～59、60+)。泛化职业:将职业分为高、中、低3个级别。添加噪声:在资本收益和资本损失列上添加正态分布的噪声,以增加数据的随机性。编码分类变量:使用LabelEncoder对分类变量进行标签编码,以便更好地应用于机器学习模型。

(3) 保存处理后的数据集。

4. 使用因子分析对乳腺癌的基因表达GSE2034数据集进行特征提取。

说明:GSE2034是一个关于乳腺癌的基因表达数据集,包含不同样本的基因表达水平。因子分析可以发现潜在的结构或因素,这些因素可以解释观测变量之间的相关性。通过因子分析,可以减少数据的维度,并提取出最重要的特征。

要求:

(1) 加载并预处理GSE2034数据集。

(2) 使用因子分析进行特征提取。

(3) 可视化因子得分。

(4) 解释每个因子的方差解释比例。