Feb 19, 2021 · 7 min read · Listen

8 Best Data Transformation in Pandas

A guide on how to do data transformation with toy datasets in Pandas.

Photo by Corinne Kutz on Unsplash

Most real-world dataset is dirty. Before analyzing the dataset, you need to transform this dataset. This process is called data transformation. In this post, I'll talk about data transformation. In short, I'll cover the following topics:

- 1. Finding the duplicate values
- 2. Mapping
- 3. Replacing
- 4. Renaming
- 5. Cutting
- 6. Finding the specific values
- 7. Selecting
- 8. Creating dummy variables

Let’s get started.

First of all, let me import Pandas.

```
In [1]:  1  import pandas as pd
```

1. Finding the duplicate values

Let’s create a dataset.

In [2]:

```
data=pd.DataFrame({"a":["one","two"]*3,
                   "b":[1,1,2,3,2,3]})
data
```

Out[2]:

	a	b
0	one	1
1	two	1
2	one	2
3	two	3
4	one	2
5	two	3

To see whether the row is repeated or not, you can use the duplicated method.

In [3]:

```
data.duplicated()
```

Out[3]:

```
0    False
1    False
2    False
3    False
4     True
5     True
dtype: bool
```

To remove duplicate rows, you can use the drop_duplicates method which returns a data frame.

```
In [4]:  1 data.drop_duplicates()
```

Out[4]:

	a	b
0	one	1
1	two	1
2	one	2
3	two	3

To find duplicates on specific columns, you can use the duplicated method. To show this, let me add a column to the dataset.

```
In [5]:  1 data["c"]=range(6)
         2 data
```

Out[5]:

	a	b	c
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	2	4
5	two	3	5

For example, let's find the duplicate values according to column c.

```
In [6]:  1 data.duplicated(["a","b"],keep="last")
```

```
Out[6]: 0    False
        1    False
        2     True
        3     True
        4    False
        5    False
        dtype: bool
```

2. Mapping

You can transfer the values of data to the dataset with a function or a mapping. To show that, let me create a dataset.

```
In [7]:  1 df=pd.DataFrame({"names":["Tim","tom","Sam",
         2                          "kate","Kim"],
         3                "scores":[60,50,70,80,40]})
         4 df
```

Out[7]:

	names	scores
0	Tim	60
1	tom	50
2	Sam	70
3	kate	80
4	Kim	40

Let's add a column to show each student's class.

```
In [8]:  1 classes={"Tim":"A","Tom":"A","Sam":"B",
         2          "Kate":"B","Kim":"B"}
```

Let's convert the first character of the names to a capital (uppercase) letter so that the names are the same.

```
In [9]:  1  n=df["names"].str.capitalize()
```

Now, let's add a branch variable to the df dataset using the map() method.

```
In [10]:  1  df["branches"]=n.map(classes)
```

```
In [11]:  1  df
```

Out[11]:

	names	scores	branches
0	Tim	60	A
1	tom	50	A
2	Sam	70	B
3	kate	80	B
4	Kim	40	B

3. Replacing

As you know, you can use the fillna method for missing data. You can also the replace method for missing data. To show this, let me create a series named s.

```
In [12]:  1  s=pd.Series([80,70,90,60])
          2  s
```

```
Out[12]: 0    80
         1    70
         2    90
         3    60
         dtype: int64
```

I'm going to add missing data to this data. To do this, let me use Numpy. First, I'm going to import NumPy.

```
In [13]:  1  import numpy as np
```

Let's assign the missing data to 70 with the replace method.

```
In [14]:  1 s.replace(70,np.nan)

Out[14]: 0    80.0
         1     NaN
         2    90.0
         3    60.0
         dtype: float64
```

Using the replace method, you can assign different values to each value. For example, let's assign missing data instead of 70 and 0 instead of 60.

```
In [15]:  1 s.replace([70,60],[np.nan,0])

Out[15]: 0    80.0
         1     NaN
         2    90.0
         3     0.0
         dtype: float64
```

To replace, you can use the dictionary structure.

```
In [16]:  1 s.replace({90:100,60:0})

Out[16]: 0     80
         1     70
         2    100
         3      0
         dtype: int64
```

4. Renaming

You can rename axes in the dataset with a function or mapping. To show this, let's create a dataset.

In [17]:

```
df=pd.DataFrame(
    np.arange(12).reshape(3,4),
    index=[0,1,2],
    columns=["tim","tom","kim","sam"])
df
```

Out[17]:

	tim	tom	kim	sam
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Now, let's create an s variable and pass it to this dataset.

In [18]:

```
s=pd.Series(["one","two","three"])
df.index=df.index.map(s)
```

Let's take a look at the dataset.

In [19]:

```
df
```

Out[19]:

	tim	tom	kim	sam
one	0	1	2	3
two	4	5	6	7
three	8	9	10	11

You can capitalize the row and column names with the rename method,

```
In [20]:  1 df.rename(index=str.title,columns=str.upper)
```

Out[20]:

	TIM	TOM	KIM	SAM
One	0	1	2	3
Two	4	5	6	7
Three	8	9	10	11

You can change the row or column names using the dictionary structure with the rename method.

```
In [21]:  1 df.rename(index={"one":"ten"},
          2             columns={"sam":"kate"},
          3             inplace=True)
          4 df
```

Out[21]:

	tim	tom	kim	kate
ten	0	1	2	3
two	4	5	6	7
three	8	9	10	11

5. Cutting

To group data periodically., you can use the cut method. To show this, let's create a data.

```
In [22]:  1 sc=[30,80,40,90,60,45,95,75,55,100,65,85]
```

Let's split this data into certain intervals. For this, let's create an x variable and specify the intervals.

```
In [23]:  1 x=[20,40,60,80,100]
```

Now, let's determine the interval of each value in sc according to these intervals.

```
In [24]:
1 y=pd.cut(sc,x)
2 y
```

```
Out[24]: [(20, 40], (60, 80], (20, 40], (80, 100], (40, 6
0], ..., (60, 80], (40, 60], (80, 100], (60, 8
0], (80, 100]]
Length: 12
Categories (4, interval[int64]): [(20, 40] < (4
0, 60] < (60, 80] < (80, 100]]
```

You can use the codes attribute to see the categorical code for each value.

```
In [25]:
1 y.codes
```

```
Out[25]: array([0, 2, 0, 3, 1, 1, 3, 2, 1, 3, 2, 3], dtyp
e=int8)
```

You can use the categories attribute to see intervals.

```
In [26]:
1 y.categories
```

```
Out[26]: IntervalIndex([(20, 40], (40, 60], (60, 80], (8
0, 100]],
              closed='right',
              dtype='interval[int64]')
```

Let's see the frequency of the categories.

```
In [27]:
1 pd.value_counts(y)
```

```
Out[27]: (80, 100]    4
(60, 80]     3
(40, 60]     3
(20, 40]     2
dtype: int64
```

Note that the intervals start with normal brackets and end with square brackets. To change this, you can use the option right = False.

```
In [28]:
1 y=pd.cut(sc,x,right=False)
2 y
```

```
Out[28]: [[20, 40), [80, 100), [40, 60), [80, 100), [60,
80), ..., [60.0, 80.0), [40.0, 60.0), NaN, [60.
0, 80.0), [80.0, 100.0)]
Length: 12
Categories (4, interval[int64]): [[20, 40) < [4
0, 60) < [60, 80) < [80, 100)]
```

You can name the intervals. Let me show this.

```
In [29]:
1 nm=["low", "medium", "high", "very high"]
2 pd.cut(sc,x,labels=nm)
```

```
Out[29]: ['low', 'high', 'low', 'very high', 'medium',
..., 'high', 'medium', 'very high', 'high', 'ver
y high']
Length: 12
Categories (4, object): ['low' < 'medium' < 'hig
h' < 'very high']
```

You can categorize the data by entering the number of intervals.

```
In [30]:
1 pd.cut(sc,10)
```

```
Out[30]: [(29.93, 37.0], (79.0, 86.0], (37.0, 44.0], (86.
0, 93.0], (58.0, 65.0], ..., (72.0, 79.0], (51.
0, 58.0], (93.0, 100.0], (58.0, 65.0], (79.0, 8
6.0]]
Length: 12
Categories (10, interval[float64]): [(29.93, 37.
0] < (37.0, 44.0] < (44.0, 51.0] < (51.0, 58.0]
... (72.0, 79.0] < (79.0, 86.0] < (86.0, 93.0] <
(93.0, 100.0]]
```

You can use the qcut method to divide the data into quarter intervals. To show this, first, let's generate data from the normal distribution and then use the qcut method.

```
In [31]:
1 data=np.random.randn(100)
2 c=pd.qcut(data,4)
3 c
```

```
Out[31]: [(-0.135, 0.631], (0.631, 2.286], (-0.629, -0.13
5], (0.631, 2.286], (-2.186, -0.629], ..., (-0.6
29, -0.135], (0.631, 2.286], (0.631, 2.286], (-
0.135, 0.631], (-0.629, -0.135]]
Length: 100
Categories (4, interval[float64]): [(-2.186, -0.
629] < (-0.629, -0.135] < (-0.135, 0.631] < (0.6
31, 2.286]]
```

Now, let's take a look at how many values fall in each quarter interval.

```
In [32]:
1 pd.value_counts(c)
```

```
Out[32]: (0.631, 2.286]      25
(-0.135, 0.631]     25
(-0.629, -0.135]    25
(-2.186, -0.629]    25
dtype: int64
```

6. Finding the specific values in a dataset

You can find specific values in the dataset. To show this, let's generate data from the normal distribution.

```
In [33]:
1 data=pd.DataFrame(np.random.randn(1000,4))
2 data.head()
```

Out[33]:

	0	1	2	3
0	0.661162	-1.315550	0.138893	-2.186859
1	-0.422096	0.587658	-0.478577	-0.285737
2	-0.283092	-0.021623	1.194335	-0.197599
3	-1.545286	-0.219977	0.353704	0.424970
4	-0.196521	3.491917	0.016217	-0.464119

You can use the describe method to see summary statistics. Let me show this.

```
In [34]:
1 data.describe()
```

Out[34]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.048690	0.030168	-0.050958	-0.004881
std	0.991747	0.996656	0.992911	1.021762
min	-2.999425	-3.514219	-2.864107	-3.186480
25%	-0.723203	-0.622904	-0.726046	-0.752119
50%	-0.058605	0.027484	-0.036939	0.023319
75%	0.560961	0.654890	0.605512	0.739495
max	3.952527	3.491917	3.074154	3.002287

For example, let's find values whose absolute value exceeds 3 in a column with 1 index. To show this, I'm going to assign the column with the 1st index of the data to the col variable.

```
In [35]:
1 col=data[1]
```

Now, let's find the values whose absolute value is greater than 3.

```
In [36]:  1  col[np.abs(col)>3]
```

```
Out[36]: 4       3.491917
         117     3.098920
         311    -3.067095
         903    -3.514219
         Name: 1, dtype: float64
```

To find rows with at least one absolute value exceeding 3 in the entire data set, you can use the any method.

```
In [37]:  1  data[(np.abs(data)>3).any(1)]
```

Out[37]:

	0	1	2	3
4	-0.196521	3.491917	0.016217	-0.464119
117	-1.360727	3.098920	-0.902404	-1.874759
311	-1.408380	-3.067095	-0.034621	-1.377992
492	-0.508447	-0.264550	-2.246989	-3.096799
510	3.952527	-0.307437	1.160524	1.022866
547	-0.146700	-0.594890	3.074154	-0.198825
642	3.116662	-0.466845	-0.543486	0.038652
867	0.222411	-0.131135	0.618451	3.002287
903	-0.917298	-3.514219	-2.500715	0.259489
956	0.115918	1.314808	-0.220663	-3.186480

You can use the sign method to see positive or negative values.

```
In [38]:
1 np.sign(data).head()
```

Out[38]:

	0	1	2	3
0	1.0	-1.0	1.0	-1.0
1	-1.0	1.0	-1.0	-1.0
2	-1.0	-1.0	1.0	-1.0
3	-1.0	-1.0	1.0	1.0
4	-1.0	1.0	1.0	-1.0

7. Selecting

You can use the permutation to sort randomly. To show this, I'm going to create a data frame.

```
In [39]:
1 data=pd.DataFrame(
2     np.arange(12).reshape(4,3))
3 data
```

Out[39]:

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

Now, let's use the permutation.

```
In [40]:
1 rw=np.random.permutation(4)
2 rw
```

```
Out[40]: array([2, 3, 0, 1])
```

To change the order of the row in the data, let's use the take method.

```
In [41]:  1 data.take(rw)
```

Out[41]:

	0	1	2
2	6	7	8
3	9	10	11
0	0	1	2
1	3	4	5

To randomly select a row, you can use the sample method.

```
In [42]:  1 data.sample()
```

Out[42]:

	0	1	2
3	9	10	11

You can select random rows.

```
In [43]:  1 data.sample(n=2)
```

Out[43]:

	0	1	2
0	0	1	2
2	6	7	8

8. Converts categorical data into dummy variables

You can use the get_dummy method to convert a categorical variable into a “dummy” or “indicator”. To show this, let me create a dataset.

In [44]:

```
data=pd.DataFrame(
    {"letter":["c","b","a","b","b","a"],
                    "number":range(6)})
data
```

Out[44]:

	letter	number
0	c	0
1	b	1
2	a	2
3	b	3
4	b	4
5	a	5

Let's convert the letter column into the dummy variables with the get_dummies method.

In [45]:

```
pd.get_dummies(data["letter"])
```

Out[45]:

	a	b	c
0	0	0	1
1	0	1	0
2	1	0	0
3	0	1	0
4	0	1	0
5	1	0	0

You can use the dummy variable to find the interval in which values fall in a dataset. To show this, let's generate data from the normal distribution.

```
In [46]:
1 data=np.random.randn(10)
2 data
```

```
Out[46]: array([-0.47399816, -0.4700948 , -0.10090975, -
0.49105714,  0.85748633,
       -2.17384891, -1.89062041,  1.15155524, -
1.12043372,  0.82935199])
```

Now let's divide the dataset into 4 intervals and see which interval each value falls into. To do this, I'm going to use the cut method.

```
In [47]:
1 pd.get_dummies(pd.cut(data,4))
```

Out[47]:

	(-2.177, -1.342]	(-1.342, -0.511]	(-0.511, 0.32]	(0.32, 1.152]
0	0	0	1	0
1	0	0	1	0
2	0	0	1	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0
6	1	0	0	0
7	0	0	0	1
8	0	1	0	0
9	0	0	0	1

That's it. In this post, I explained 8 useful data transformations. I hope you enjoy this post. Thanks for reading. You can find the notebook here.

Don’t forget to follow us on YouTube |Twitter | GitHub | Linkedin| Kaggle

Python Pandas Tutorial

Pandas is one of Python’s most important libraries. In this blog post, I will talk about the Pandas library and show...

medium.com

PRACTICAL DATA ANALYSIS with PANDAS

In my last post, I mentioned working with data in Pandas library. One of Python’s most important libraries is pandas...

levelup.gitconnected.com

If this post was helpful, please click the clap 👏 button below a few times to show me your support 👇

Python Pandas Pandas Tutorial Python Pandas Tutorial Pandas Dataframe Data Transformation

Get an email whenever Tirendaz AI publishes.

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Subscribe