Pandas Report

Author: Rodrigo Santos de Carvalho - Published 2022-02-15 03:06 - (0 Reads)

Pandas, Python library for data manipulation and analysis

Pandas

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data. It allows us to analyze big data and make conclusions based on statistical theories. Pandas can also clean messy data sets, and make them readable and relevant. For those reasons, Pandas is really important for Machine Learning, since the latter usually deals with large amounts of data, which often need to be prepared before the algorithm is run.

Pandas is usually imported as pd:

```
import pandas as pd
```

Series and Labels in Pandas

In Pandas, a series is a one-dimensional array, which represents a column in a table.

```
import pandas as pd

array = ["a", "b", "c"]
series = pd.Series(array)

print(series)
```

output:

```
0    a
1    b
2    c
```

Labels work just like the indexes of the array which represents a series. If the value for a label isn't specified, it is defined as the index of that value in the array. The example below shows how the labels (0, 1 ,2) return the values of the series above:

```
import pandas as pd

array = ["a", "b", "c"]
series = pd.Series(array)

print(series[0])
print(series[1])
print(series[2])
```

output:

```
a
b
c
```

Labels, as stated before, can be specified, instead of using the indexes array which represents the series:

```
import pandas as pd

array = ["a", "b", "c"]
series = pd.Series(array, index = ["x", "y", "z"])

print(series)
```

output:

```
x    a
y    b
z    c
```

Series can be created with a dictionary too, with the keys as labels. This is an easier way to specify the labels.

```
import pandas as pd

dict = {
"x": 1,
 "y":  2,
"z": 3
}
series = pd.Series(dict)

print(series)
```

output:

```
x    1
y    2
z    3
```

DataFrames in Pandas

In Pandas, DataFrames are multi-dimensional tables, whereas series are the columns of that table. Therefore, a DataFrame can be created from two or more series.

```
import pandas as pd

data = {
  "numbers": [1, 2, 3],
  "letters": ["a", "b", "c"],
  "colors": ["red", "green", "blue"]
}
data_frame = pd.DataFrame(data)

print(data_frame)
```

output:

```
numbers letters colors
0        1       a    red
1        2       b  green
2        3       c   blue
```

Since DataFrames are like a table of values, they are made of rows and columns. With Pandas, those rows can be easily located. The bigger the DataFrame being analyzed is, the more important it is to be able to accurately locate its rows.

Rows can be located through the loc attribute:

```
import pandas as pd

data = {
  "numbers": [1, 2, 3],
  "letters": ["a", "b", "c"],
  "colors": ["red", "green", "blue"]
}
data_frame = pd.DataFrame(data)

rows_0_and_1 = data_frame.loc[[0, 1]]

print(rows_0_and_1 )
```

output:

```
numbers letters colors
0        1       a    red
1        2       b  green
```

The loc attribute can receive [x:y] as a parameter, which will locate all the rows numbered from x to y.

Labels can also be named in DataFrames:

```
import pandas as pd

data = {
  "height": [1.71, 1.60, 1.80],
  "weight": [63, 50, 72]
}
data_frame = pd.DataFrame(data, index = ["person1", "person2", "person3"])

print(data_frame)
```

output:

```
height  weight
person1    1.71      63
person2    1.60      50
person3    1.80      72
```

Rows can be located through named labels as well:

```
import pandas as pd

data = {
  "height": [1.71, 1.60, 1.80],
  "weight": [63, 50, 72]
}
data_frame = pd.DataFrame(data, index = ["person1", "person2", "person3"])
person1_and_2 = data_frame.loc[["person1", "person2"]]

print(person1_and_2)
```

output:

```
height  weight
person1    1.71      63
person2    1.60      50
```

Pandas Read CSV

Big datasets are commonly stored in CSV files (comma separated values). CSV files are really simple and easy to read. Pandas can read a CSV file, transform it into a DataFrame and manipulate it.

```
import pandas as pd

dataset = pd.read_csv('credit_data.csv')

print(dataset)
```

output:

```
clientid        income        age         loan  default
0            1  66155.925095  59.017015  8106.532131        0
1            2  34415.153966  48.117153  6564.745018        0
2            3  57317.170063  63.108049  8020.953296        0
3            4  42709.534201  45.751972  6103.642260        0
4            5  66952.688845  18.584336  8770.099235        1
...        ...           ...        ...          ...      ...
1995      1996  59221.044874  48.518179  1926.729397        0
1996      1997  69516.127573  23.162104  3503.176156        0
1997      1998  44311.449262  28.017167  5522.786693        1
1998      1999  43756.056605  63.971796  1622.722598        0
1999      2000  69436.579552  56.152617  7378.833599        0
```

Analyzing DataFrames in Pandas

The head() method helps in getting a quick overview of the DataFrame. It returns the headers and a specified number of rows, starting from the top.

```
dataset = pd.read_csv("credit_data.csv")

print(dataset.head(15))
```

output:

```
clientid        income        age         loan  default
0          1  66155.925095  59.017015  8106.532131        0
1          2  34415.153966  48.117153  6564.745018        0
2          3  57317.170063  63.108049  8020.953296        0
3          4  42709.534201  45.751972  6103.642260        0
4          5  66952.688845  18.584336  8770.099235        1
5          6  24904.064140  57.471607    15.498598        0
6          7  48430.359613  26.809132  5722.581981        0
7          8  24500.141984  32.897548  2971.003310        1
8          9  40654.892537  55.496853  4755.825280        0
9         10  25075.872771  39.776378  1409.230371        0
10        11  64131.415372  25.679575  4351.028971        0
11        12  59436.847123  60.471936  9254.244538        0
12        13  61050.346079  26.355044  5893.264659        0
13        14  27267.995458  61.576776  4759.787581        0
14        15  63061.960174  39.201553  1850.369377        0
```

Similarly, the tail() method returns the headers and a specified number of rows, starting from the bottom.

```
dataset = pd.read_csv("credit_data.csv")

print(dataset.tail(15))
```

output:

```
clientid        income        age         loan  default
1985      1986  22371.522191  39.142225  2291.856428        0
1986      1987  67994.988470  38.622259  7289.014109        0
1987      1988  49640.004702  20.542409  5760.858734        0
1988      1989  42067.246446  24.270612  4601.606183        0
1989      1990  43662.092688  25.252609  7269.596897        1
1990      1991  34237.575419  34.101654  2658.090632        0
1991      1992  26300.446554  45.539385  2317.393678        0
1992      1993  30803.806165  23.250084   623.024153        0
1993      1994  54421.410155  26.821928  3273.631823        0
1994      1995  24254.700791  37.751622  2225.284643        0
1995      1996  59221.044874  48.518179  1926.729397        0
1996      1997  69516.127573  23.162104  3503.176156        0
1997      1998  44311.449262  28.017167  5522.786693        1
1998      1999  43756.056605  63.971796  1622.722598        0
1999      2000  69436.579552  56.152617  7378.833599        0
```

There’s also the info() method. This method returns some information about the dataset.

```
dataset = pd.read_csv("credit_data.csv")

print(dataset.info())
```

output:

```
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   clientid  2000 non-null   int64
 1   income    2000 non-null   float64
 2   age       1997 non-null   float64
 3   loan      2000 non-null   float64
 4   default   2000 non-null   int64
dtypes: float64(3), int64(2)
memory usage: 78.2 KB
None
```

Moreover, the describe() method returns descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.

```
dataset = pd.read_csv("credit_data.csv")

print(dataset.describe())
```

output:

```
clientid        income          age          loan      default
count  2000.000000   2000.000000  1997.000000   2000.000000  2000.000000
mean   1000.500000  45331.600018    40.807559   4444.369695     0.141500
std     577.494589  14326.327119    13.624469   3045.410024     0.348624
min       1.000000  20014.489470   -52.423280      1.377630     0.000000
25%     500.750000  32796.459717    28.990415   1939.708847     0.000000
50%    1000.500000  45789.117313    41.317159   3974.719419     0.000000
75%    1500.250000  57791.281668    52.587040   6432.410625     0.000000
max    2000.000000  69995.685578    63.971796  13766.051239     1.000000
```

Cleaning Data with Pandas

It’s common that the data is not always correct (improper data or input errors), or just bad, in datasets. And since the accuracy of Machine Learning algorithms is dependent on the quality of the provided data, cleaning the data is essential.

Furthermore, empty cells, data in wrong format, wrong data and duplicates are examples of bad data that can “damage” the analysis of a dataset. In that manner, Pandas can be used to fix those examples of bad data.

The first example, empty data cells, can be easily fixed by simply removing the row that contains empty cells. Generally speaking, given that datasets are usually really big, if just a few rows are removed, the impact of removing them isn't significant.

The isnull() method returns the null values in the dataset, and the sum() returns the quantity of those values and to which column they belong.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
print(dataset.isnull().sum())
```

output:

```
clientid    0
income      0
age         3
loan        0
default     0
dtype: int64
```

Finally, the dropna() method removes the empty cells.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
dataset.dropna(inplace = True)

print(dataset.isnull().sum())
```

output:

```
clientid    0
income      0
age         0
loan        0
default     0
```

Moreover, instead of deleting entire rows, empty cells can be replaced through the fillna() method. This method allows any value to replace the empty cells, but it’s often better to replace them with the mean, median or mode of the column that empty cell belongs to.

The mean(), median() and mode() methods can be called for such.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
dataset_mean = dataset["age"].mean()
dataset_median = dataset["age"].median()
dataset_mode = dataset["age"].mode()[0]

print(dataset_mean)
print(dataset_median)
print(dataset_mode)
```

output:

```
40.80755937840458
41.3171591130085
-52.4232799196616
```

Now, the empty cells can be replaced by the mean value of the column they are part of. The output shows which rows had the empty value replaced.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
age_mean = dataset["age"].mean()
dataset.fillna(age_mean, inplace = True)

print(dataset.loc[dataset['age']==age_mean])
```

output:

```
clientid        income        age         loan  default
28        29  59417.805406  40.807559  2082.625938        0
30        31  48528.852796  40.807559  6155.784670        0
31        32  23526.302555  40.807559  2862.010139        0
```

The second example, data in wrong format, can be fixed by either removing the rows which contain them or converting all the cells in the column into the same format.

The most common methods for that are the following: astype(); to_numeric(); to_string(); to_datetime().

Since there's no data in the wrong format in the dataset being analyzed in this report, the example below just shows how pandas can convert data into another format. First, the age column is printed as it is (float), then it is converted to int.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")

print(dataset["age"].astype(float))
```

output:

```
0       59.017015
1       48.117153
2       63.108049
3       45.751972
4       18.584336
          ...
1995    48.518179
1996    23.162104
1997    28.017167
1998    63.971796
1999    56.152617
```

Now, converting the column to int:

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
print(dataset["age"].astype(int))
```

output:

```
0       59
1       48
2       63
3       45
4       18
        ..
1995    48
1996    23
1997    28
1998    63
1999    56
```

However, even if data is not in the wrong format and there isn't any empty cell, data can just be wrong (third example). Sometimes, by simply taking a look at a dataset, incoherent values can be identified.

There are two main ways to fix that: replace the wrong values or remove them. For small datasets, it's rather easy to replace values one by one. Still, for big datasets, it's too unproductive and just not effective.

In the analyzed dataset, there are negative values in the age column, which is clearly incoherent, despite the values being in the right format and the cells not being empty.

The loc attribute can be used to find those values:

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
negative_age = dataset["age"] less than 0 #The site didn't accept the less than symbol
dataset = dataset.loc[negative_age]

print(dataset)
```

output:

```
clientid        income        age         loan  default
15        16  50501.726689 -28.218361  3977.287432        0
21        22  32197.620701 -52.423280  4244.057136        0
26        27  63287.038908 -36.496976  9595.286289        0
```

With the wrong values identified, they can be easily replaced. And to keep the consistency between the age values in the dataset, the negative values will be replaced by the mean value for age (which won't consider the negative values).

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")

positive_age = dataset["age"]>0
dataset_means = dataset[positive_age].mean()
age_mean = dataset_means["age"]

print(age_mean)
```

output:

```
40.92770044906149
```

Finally, the negative values can be replaced:

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
age_mean = dataset["age"][dataset["age"]>0].mean()
dataset["age"][dataset['age'] < 0] = 40.92770044906149

print(dataset[dataset['age'] == 40.92770044906149])
```

output:

```
clientid        income      age         loan  default
15        16  50501.726689  40.9277  3977.287432        0
21        22  32197.620701  40.9277  4244.057136        0
26        27  63287.038908  40.9277  9595.286289        0
```

Finally, the last example of bad data, duplicate data, is easily fixed by removing duplicates. The drop_duplicates() method does just that.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")
dataset.drop_duplicates(inplace = True)
```

And to check if the duplicates have actually been removed, the duplicated() method can be used.

```
import pandas as pd

dataset = pd.read_csv("credit_data.csv")

def check_duplicate(dataset):
    for data in dataset.duplicated():
        if data == True:
            print(True)
            return
    return False

print(check_duplicate(dataset))
```

output:

```
False
```

Hence, the dataset has been completed cleaned.