2.4 Data Cube Operations

May 8, 2019

1 2.4 Data Cube Operations

Data cubes are d-dimensional hypercubes. We can answer questions about a data set by manipulating this hypercube. In this section, we will study three basic operations: slicing, dicing, and roll-ups.

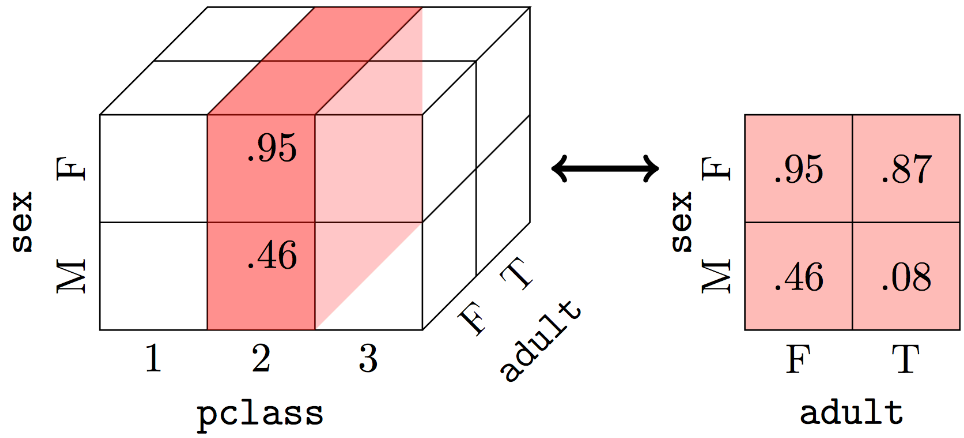
```
In [1]: %matplotlib inline
        import numpy as np
        import pandas as pd
        titanic_df = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/mas

        titanic_df["adult"] = (titanic_df["age"] >= 18)

        survival_cube = titanic_df.pivot_table(
            index="sex", columns=["pclass", "adult"],
            values="survived", aggfunc=np.mean)
        survival_cube
```

```
Out[1]: pclass         1                   2                   3
        adult      False     True      False     True      False     True
        sex
        female  0.947368  0.968000  0.952381  0.870588  0.536364  0.443396
        male    0.400000  0.326389  0.464286  0.083916  0.147059  0.155709
```

2 Slicing

Slicing a data cube refers to fixing the value of one dimension of the hypercube. For example, suppose we only want to know the survival rates of passengers in second class. To do this, we fix the value of `pclass` at 2 and look at the survival rates over the other dimensions.

It is easy to see why this operation is called "slicing" if you imagine a three-dimensional cube. When we fix the value of one dimension, we are essentially slicing the cube at that value, as shown in the figure below.



Each slice reduces the dimension of a data cube by one. If the original data cube had d dimensions, then the slice has $d - 1$ dimensions.

To slice a pivot table in `pandas`, we simply access the corresponding row or column in the `DataFrame`. For example, to get the survival rates for the passengers in second class from the data cube above, we can simply select the column labeled 2. The result is a two-dimensional data cube:

```
In [2]: survival_cube[2]
```

```
Out[2]: adult      False     True
        sex
        female  0.952381  0.870588
        male    0.464286  0.083916
```

Depending on how the pivot table is arranged, the slice is sometimes not in data cube form. For example, if we slice the data cube to get only the data for male passengers, the output is two-dimensional but not in data cube form:

```
In [3]: survival_cube.loc["male"]
```

```
Out[3]: pclass  adult
        1       False    0.400000
                True     0.326389
        2       False    0.464286
                True     0.083916
        3       False    0.147059
                True     0.155709
        Name: male, dtype: float64
```

But it is easy to convert this tabular data into a data cube; we simply unstack the `Series` so that each value of `adult` is a separate column.

```
In [4]: survival_cube.loc["male"].unstack()
```

```
Out[4]: adult      False     True
        pclass
        1       0.400000  0.326389
        2       0.464286  0.083916
        3       0.147059  0.155709
```

Slicing was easy in the two examples above because pclass and sex were both the outermost (i.e., first) level in their respective indexes. But what if we want to slice on a dimension that is buried in some intermediate level of a MultiIndex? We can use the "cross-section" function (.xs) of pandas. For example, the following code returns the survival rates for the children on the Titanic:

```
In [5]: survival_cube.xs(False, level="adult", axis=1)

Out[5]: pclass         1         2         3
        sex
        female  0.947368  0.952381  0.536364
        male    0.400000  0.464286  0.147059
```

This code tells pandas to return all columns (because axis=1) of survival_cube where adult is equal to False.

3 Dicing

Dicing is like slicing, except that we fix the values of two or more dimensions. For example, if we want to know the survival rates of males in second class, we could dice the data cube as follows:

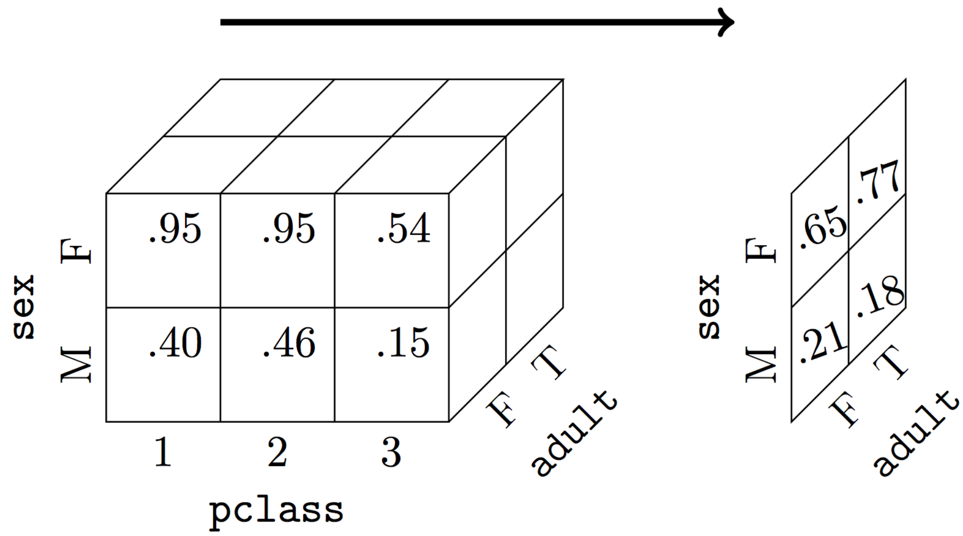
```
In [6]: survival_cube.loc["male", 2]

Out[6]: adult
        False    0.464286
        True     0.083916
        Name: male, dtype: float64
```

Notice that the result is one-dimensional, since there is only one remaining dimension that we have not fixed (adult). In general, if we dice a d-dimensional data cube along k dimensions, the result will be a $(d-k)$-dimensional data cube. In the example above, $d = 3$ and $k = 2$, so the output had $d - k = 1$ dimensions.

4 Roll-ups

The data cube above contains information about the survival rate by pclass, sex, and adult. But what if we are only interested in the survival rate by sex and adult? To do this, we have to **roll-up** data cube over the variable pclass.

The roll-up operation is diagrammed in the figure below. We want to collapse the pclass dimension, resulting in a two-dimensional data cube over sex and adult. Just as with the slicing operation, each roll-up operation reduces the dimension of the data cube by one.



What is the best way to calculate roll-ups? This is a trick question: the best way is to avoid calculating them at all! If we calculate the roll-ups when the pivot table is first created, then we can just look them up without having to calculate them. To get the roll-ups alongside the cell values, we specify `margins=True` in `.pivot_table()`.

```
In [7]: survival_cube_with_rollups = titanic_df.pivot_table(
            index=["sex", "adult"], columns="pclass",
            values="survived", aggfunc=np.mean,
            margins=True
        )

        survival_cube_with_rollups
```

```
Out[7]: pclass                1         2         3       All
        sex    adult
        female False   0.947368  0.952381  0.536364  0.646667
               True    0.968000  0.870588  0.443396  0.765823
        male   False   0.400000  0.464286  0.147059  0.213483
               True    0.326389  0.083916  0.155709  0.180556
        All            0.619195  0.429603  0.255289  0.381971
```

Compared with the default pivot table (i.e., without `margins=True`), this pivot table has an extra row and an extra column, both labeled "All". They are in the *margins* of the original table (which is why the extra argument to `.pivot_table()` was `margins=True`). This additional row and column contain various lower-dimensional roll-ups of the original data cube:

- The cell in the bottom right of this table is the roll-up of all three dimensions. In other words, it is the overall survival rate.
- The other values in the last column (labeled "All") represent the roll-up of `pclass`. In other words, they are the survival rates by `sex` and `adult`.
- The other values in the last row (labeled "All") represent the roll-up of both `sex` and `adult`. In other words, they are the survival rates by `pclass`.

However, this table does not store all the possible roll-ups of the three-dimensional datacube. For example, it does not store the roll-up of `sex`, nor does it store the roll-up of both `pclass` and `adult`. When designing a pivot table, it is a good idea to think about which roll-ups are most important and to choose the row indexes and columns accordingly so that those roll-ups are available.

4.1 Calculating Your Own Roll-Ups

Suppose you forgot to include the roll-ups when you first created the pivot table, or perhaps you need a roll-up that your pivot table does not provide. In most cases, there is no way to reconstruct the roll-ups from just the data cube. However, for some metrics, it is possible to reconstruct the roll-ups from the data cube.

For example, consider a data cube that stores the *number* of survivors by `sex`, `adult`, and `pclass`.

```
In [8]: num_survivors_cube = titanic_df.pivot_table(
            index=["sex", "adult"], columns="pclass",
            values="survived", aggfunc=np.sum
        )

        num_survivors_cube

Out[8]: pclass          1   2   3
        sex    adult
        female False   18  20  59
               True   121  74  47
        male   False   14  13  30
               True    47  12  45
```

Now, if we want to roll-up the `pclass` variable, we can calculate the total number of survivors by summing the numbers in first, second, and third class. In other words, we need to sum each row of the `DataFrame` above. This is possible using `.sum()`, but we have to specify an additional keyword argument, `axis=`, so that `pandas` knows which dimension to sum over:

- `axis=0` means aggregate *over* the rows (i.e., dimension 0), returning one number per column
- `axis=1` means aggregate *over* the columns (i.e., dimension 1), returning one number per row

Because we want the sum of each row, we are aggregating over the columns; thus we need to sum over `axis=1`:

```
In [9]: num_survivors_cube.sum(axis=1)

Out[9]: sex     adult
        female  False     97
                True     242
        male    False     57
                True     104
        dtype: int64
```

As a sanity check, let's make sure these numbers match the results from `.pivot_table()` when we set `margins=True`:

```
In [10]: titanic_df.pivot_table(
             index=["sex", "adult"], columns="pclass",
             values="survived", aggfunc=np.sum,
             margins=True
         )
```

```
Out[10]: pclass          1    2    3  All
         sex    adult
         female False   18   20   59   97
                True   121   74   47  242
         male   False   14   13   30   57
                True    47   12   45  104
         All           200  119  181  500
```

The numbers in the "All" column match exactly!

5 Exercises

Exercise 1. We saw one case where it was possible to manually reconstruct roll-ups using only the values in a data cube.

Is it possible to calculate the roll-up of `pclass` from just the values in `survival_cube` (a pivot table defined above)? In other words, can we reconstruct the survival rates by `sex` and `adult` from just the survival rates in `survival_cube`? Try a few different approaches and compare the results against the true answer, which you can obtain using `.groupby()` or `.pivot_table(..., margins=True)`.

```
In [23]: survival_cube
         survival_cube.stack("adult").mean(axis=1), survival_cube.stack("adult").mean(axis=1).u
         #this method averages over p-class equally instead of proportionally across the numbe
```

```
Out[23]: (sex     adult
          female  False    0.812038
                  True     0.760661
          male    False    0.337115
                  True     0.188671
          dtype: float64, adult      False     True
          sex
          female  0.812038  0.760661
          male    0.337115  0.188671)
```

```
In [24]: titanic_df.groupby(["sex", "adult"])["survived"].mean()
         #Using groupby will give us correct values; cannot calculate rollups after pivot tabl
```

```
Out[24]: sex     adult
         female  False    0.646667
```

```
        True     0.765823
male    False    0.213483
        True     0.180556
Name: survived, dtype: float64
```

```
In [25]: titanic_df.pivot_table(
             index=["sex","adult"], columns="pclass",
             values="survived", aggfunc="mean",
             margins=True)

         #margins=TRUE gives us the correct margin averages by weight
```

```
Out[25]: pclass               1         2         3       All
         sex    adult
         female False  0.947368  0.952381  0.536364  0.646667
                True   0.968000  0.870588  0.443396  0.765823
         male   False  0.400000  0.464286  0.147059  0.213483
                True   0.326389  0.083916  0.155709  0.180556
         All           0.619195  0.429603  0.255289  0.381971
```

```
In [26]: titanic_df.pivot_table(
             index=["sex","adult"], columns="pclass",
             values="survived", aggfunc="count",
             margins=True)
```

```
Out[26]: pclass          1    2    3   All
         sex    adult
         female False   19   21  110   150
                True   125   85  106   316
         male   False   35   28  204   267
                True   144  143  289   576
         All           323  277  709  1309
```

Exercises 2-4 deal with the Tips data set (https://raw.githubusercontent.com/dlsun/data-science-book/

Exercise 2. Create a pivot table that shows the average total bill by day, time, and table size. Include roll-ups with this pivot table that make it easy to answer questions like, "Is the average bill higher for lunch or dinner?"

```
In [12]: # TYPE YOUR CODE HERE.
         tips = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/master/

         avg_total_bill_pivot = tips.pivot_table(
             index=["time"], columns=["size","day"],
             values="total_bill", aggfunc=np.mean,
             margins=True)

         avg_total_bill_pivot
```

```
Out[12]: size       1                         2                                    3   \
         day      Fri    Sat    Thur         Fri        Sat      Sun        Thur     Fri
```

```
time
Dinner    NaN  5.16    NaN  17.799091  16.83717  17.56  18.780000    NaN
Lunch    8.58   NaN  10.07  13.072000       NaN    NaN  15.079787  15.98
All      8.58  5.16  10.07  16.321875  16.83717  17.56  15.156875  15.98

size                         ...       4                                \
day           Sat      Sun   ...     Fri        Sat        Sun   Thur
time                         ...
Dinner  25.509444  22.184    ...   40.17  29.876154  26.688333    NaN
Lunch         NaN      NaN   ...     NaN        NaN        NaN  29.95
All     25.509444  22.184    ...   40.17  29.876154  26.688333  29.95

size        5                   6                   All
day       Sat   Sun   Thur    Sun       Thur
time
Dinner  28.15  27.0    NaN  48.17        NaN  20.797159
Lunch     NaN   NaN  41.19    NaN  30.383333  17.168676
All     28.15  27.0  41.19  48.17  30.383333  19.785943

[3 rows x 21 columns]
```

Exercise 3. Create a pivot table that shows that average total bill by day and time for parties of size 2. (Don't do this by calling `.pivot_table()` on the original data. You should be able to do this using just the pivot table you created in Exercise 2.)

In [13]:
```
# TYPE YOUR CODE HERE.
avg_total_bill_pivot[2]

#if size and day was flipped, use avg_total_bill_pivot.xs(2, level="size", axis=1)
```

Out[13]:
```
day           Fri       Sat    Sun       Thur
time
Dinner  17.799091  16.83717  17.56  18.780000
Lunch   13.072000       NaN    NaN  15.079787
All     16.321875  16.83717  17.56  15.156875
```

Exercise 4. How would you create a pivot table that shows the average total bill by day and time? Is it possible to do this using just the pivot table you created in Exercise 2?

In [14]:
```
avg_total_bill_pivot
```

Out[14]:
```
size       1                       2                                     3  \
day      Fri   Sat   Thur        Fri       Sat    Sun       Thur    Fri
time
Dinner    NaN  5.16    NaN  17.799091  16.83717  17.56  18.780000    NaN
Lunch    8.58   NaN  10.07  13.072000       NaN    NaN  15.079787  15.98
All      8.58  5.16  10.07  16.321875  16.83717  17.56  15.156875  15.98

size                         ...       4                                \
```

```
day            Sat     Sun     ...        Fri        Sat        Sun   Thur
time                           ...
Dinner  25.509444  22.184     ...      40.17  29.876154  26.688333    NaN
Lunch         NaN     NaN     ...        NaN        NaN        NaN  29.95
All     25.509444  22.184     ...      40.17  29.876154  26.688333  29.95

size        5                  6                     All
day       Sat   Sun   Thur    Sun        Thur
time
Dinner  28.15  27.0    NaN  48.17         NaN  20.797159
Lunch     NaN   NaN  41.19    NaN   30.383333  17.168676
All     28.15  27.0  41.19  48.17   30.383333  19.785943

[3 rows x 21 columns]
```

In [15]:
```
# TYPE YOUR CODE HERE.
(avg_total_bill_pivot.drop("All", axis=0).drop("All", axis=1)).stack("day").mean(axis=
```

Out[15]:
```
time    day
Dinner  Fri     28.984545
        Sat     21.106554
        Sun     28.320467
        Thur    18.780000
Lunch   Fri     12.544000
        Thur    24.305520
dtype: float64
```

In [16]:
```
avg_total_bill_pivot.stack("day").mean(axis=1)
```

Out[16]:
```
time    day
Dinner          20.797159
        Fri     28.984545
        Sat     21.106554
        Sun     28.320467
        Thur    18.780000
Lunch           17.168676
        Fri     12.544000
        Thur    24.305520
All             19.785943
        Fri     20.262969
        Sat     21.106554
        Sun     28.320467
        Thur    24.318368
dtype: float64
```

In [17]:
```
tips.groupby(["time","day"]).total_bill.mean()

#This method is correct because in avg_total_bill_pivot (data cube), the method we us
#the average of an average which is incorrect. Each entry in the data cube is average
#given equal weight which is not valid.
```

```
Out[17]: time    day
         Dinner  Fri     19.663333
                 Sat     20.441379
                 Sun     21.410000
                 Thur    18.780000
         Lunch   Fri     12.845714
                 Thur    17.664754
         Name: total_bill, dtype: float64
```