3.4 Three Variables and Beyond

May 9, 2019

1 Chapter 3.4 Three Variables and Beyond

So far in this chapter, we have seen a few ways to summarize and visualize the relationship between two variables. But what if we are working with three or more variables? This section discusses some strategies for dealing with multivariate data.

Let's first read in the Ames housing data set, which will be a working example throughout this section.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import pandas as pd

        housing_df = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/mas
        housing_df.head()
```

```
Out[1]:    Order        PID  MS SubClass MS Zoning  Lot Frontage  Lot Area Street  \
        0      1  526301100           20        RL         141.0     31770   Pave
        1      2  526350040           20        RH          80.0     11622   Pave
        2      3  526351010           20        RL          81.0     14267   Pave
        3      4  526353030           20        RL          93.0     11160   Pave
        4      5  527105010           60        RL          74.0     13830   Pave

          Alley Lot Shape Land Contour    ...     Pool Area Pool QC  Fence  \
        0   NaN       IR1          Lvl    ...             0     NaN    NaN
        1   NaN       Reg          Lvl    ...             0     NaN  MnPrv
        2   NaN       IR1          Lvl    ...             0     NaN    NaN
        3   NaN       Reg          Lvl    ...             0     NaN    NaN
        4   NaN       IR1          Lvl    ...             0     NaN  MnPrv

          Misc Feature Misc Val Mo Sold Yr Sold Sale Type  Sale Condition  SalePrice
        0          NaN        0       5    2010       WD           Normal     215000
        1          NaN        0       6    2010       WD           Normal     105000
        2         Gar2    12500       6    2010       WD           Normal     172000
        3          NaN        0       4    2010       WD           Normal     244000
        4          NaN        0       3    2010       WD           Normal     189900

        [5 rows x 82 columns]
```

1.1 Mapping Aesthetics to Variables

In Section 3.3, we made a scatterplot showing the relationship between living area and sale price. What if we also want to understand how number of bedrooms enters into the equation?

One possibility is to make a three-dimensional scatterplot. However, 3D plots are often misleading when represented in two dimensions, and they don't generalize well to even higher dimensions. A better approach is to use other *aesthetics* of the plot, such as the color or size of the points, to represent additional variables.

The `.plot.scatter()` function in `pandas` allows us to control four aesthetics of a scatterplot. We've seen two already:

- `x=`: the variable on the x-axis
- `y=`: the variable on the y-axis

but there are two more:

- `c=`: the colors of the points (either the name of a variable in the `DataFrame` or an array specifying the color of each point)
- `s=`: the sizes of the points (must be an array specifying the size of each point)

For example, to use color to represent the number of bedrooms, we could do the following:

```
In [ ]: housing_df.plot.scatter(x="Gr Liv Area", y="SalePrice",
                                c="Bedroom AbvGr", alpha=.5)
```

Notice how the colors become darker as you move down the plot. This means that, holding living area constant, a house is less expensive the *more* bedrooms it has. (Why do you think this is?)

Now, number of bedrooms is a quantitative variable. What if we wanted to visualize how a categorical variable, such as building type, interacts with these two quantitative variables (living area and sale price)? We have to manually construct the array of colors using the `.map()` function we learned in Chapter 1.

```
In [ ]: cols = housing_df["Bldg Type"].map({
            "1Fam": "blue",
            "TwnhsE": "green",
            "Twnhs": "green",
            "Duplex": "red",
            "2fmCon": "orange"
        })

        housing_df.plot.scatter(x="Gr Liv Area", y="SalePrice",
                                c=cols, alpha=.3)
```

1.2 Small Multiples

Rather than try to pack all of the variables into a single plot, we can juxtapose several plots or "facets", each showing the data from a slightly different angle. Edward Tufte coined the term "small multiples" for this type of graphic.

For example, rather than use color to represent building type, we could have made 5 separate scatterplots, one for each building type, and arranged them in a row for easy comparison.

```
In [ ]: # Specifies a 1 x 5 grid of plots, figsize in inches
        fig, axes = plt.subplots(1, 5, figsize=(10, 4))

        bldg_types = housing_df["Bldg Type"].unique()
        for ax, bldg_type in zip(axes, bldg_types):
            housing_type = housing_df[housing_df["Bldg Type"] == bldg_type]
            housing_type.plot.scatter(x="Gr Liv Area", y="SalePrice", ax=ax)
            ax.set_title(bldg_type)
```

Of course, the goal of such a graphic is to facilitate comparison, which is difficult when the x- and y-axes of the facets are so different. Since the facets are aligned horizontally, it makes sense to use a common y-axis for all of them. We can do this by specifying `sharey=True` in `plt.subplots()`. (There is also a corresponding `sharex=` argument.)

```
In [ ]: fig, axes = plt.subplots(1, 5, figsize=(20, 8), sharey=True, sharex=True)

        bldg_types = housing_df["Bldg Type"].unique()
        for ax, bldg_type in zip(axes, bldg_types):
            housing_type = housing_df[housing_df["Bldg Type"] == bldg_type]
            housing_type.plot.scatter(x="Gr Liv Area", y="SalePrice", ax=ax)
            ax.set_title(bldg_type)
```

Sharing the y-axes between the facets also resolved another issue—the colliding y-axis labels—since now only the first plot in the figure has an y-axis label.

1.3 Grammar of Graphics

The **grammar of graphics** organizes the ideas above into a coherent philosophy. The key insight is that a graphic can be specified by mapping its "aesthetics" (e.g., color, size, x-axis, column facet) to variables in a data set. Although `pandas` provides some support for this philosophy (as we have seen above), the process is tedious and often requires writing boilerplate code. For example, in order to use color to represent building type, we had to manually map each building type to a color. Libraries based on the grammar of graphics provide a more friendly interface and hide this complexity from the user.

Software packages that implement the grammar of graphics include `ggplot2` in R and Altair in Python. Since we are working in Python, we will use Altair. The first step is to import the package.

```
In [2]: from altair import *
```

Now, let's use Altair to recreate the scatterplot from earlier, where each point was colored according to the number of bedrooms.

Every Altair command starts with `Chart(your_data_frame)`, followed by the two main elements of the graphic: - the mark (i.e., the geometric object being plotted, which for a scatterplot, is a circle) - the encoding channels (i.e., mappings between aesthetics and variables)

```
In [ ]: Chart(housing_df).mark_circle().encode(
            x="Gr Liv Area",
            y="SalePrice",
            color="Bedroom AbvGr"
        )
```

```
In [4]: housing_df["Bedroom AbvGr"]
```

```
Out[4]: 0       3
        1       2
        2       3
        3       3
        4       3
        5       3
        6       2
        7       2
        8       2
        9       3
        10      3
        11      3
        12      3
        13      2
        14      1
        15      4
        16      4
        17      1
        18      2
        19      3
        20      3
        21      3
        22      3
        23      2
        24      3
        25      3
        26      2
        27      3
        28      2
        29      2
               ..
        2900    3
        2901    2
        2902    3
        2903    3
        2904    2
        2905    2
        2906    2
        2907    3
        2908    3
        2909    4
        2910    5
        2911    3
        2912    4
        2913    1
        2914    3
```

```
2915    3
2916    1
2917    3
2918    3
2919    3
2920    3
2921    4
2922    4
2923    3
2924    4
2925    3
2926    2
2927    3
2928    2
2929    3
Name: Bedroom AbvGr, Length: 2930, dtype: int64
```

Now, what if we replace the number of bathrooms, a quantitative variable, with building type, a categorical variable?

```
In [ ]: Chart(housing_df).mark_circle().encode(
            x="Gr Liv Area",
            y="SalePrice",
            color="Bldg Type"
        )
```

Notice how Altair automatically inferred that building type was a categorical variable and chose a coloring scheme accordingly. (Compare the color schemes of the two plots above. For quantitative variables, Altair uses a color gradient, while for categorical variables, it uses distinct colors.)

On the other hand, if we had wanted to have building type be a column facet, using small multiples to show the different building types, we would map the `column` aesthetic to building type:

```
In [ ]: Chart(housing_df).mark_circle().encode(
            x="Gr Liv Area",
            y="SalePrice",
            column="Bldg Type"
        )
```

In the examples above, we mapped variables to aesthetics by simply specifying the column names. Although this is convenient, it does not allow the aesthetics to be customized any further. To customize an aesthetic, we have to specify the aesthetic as an object.

Every aesthetic in Altair has an associated Python class. The name of the class is usually just the name of the aesthetic, but with the first letter capitalized. For example, the `x` aesthetic is associated with the `X` class and the `color` aesthetic with the `Color` class. The first argument of the constructor is always the name of the variable to map to the aesthetic, but there are additional arguments that allow for customization.

For example, suppose we wanted the x-axis limits to range from 0 to 4000 and the tick labels on the y-axis to display numbers in scientific notation (i.e., 2e+5 instead of 200,000). Here's how we could achieve those customizations in Altair:

```
In [ ]: Chart(housing_df).mark_circle().encode(
            x=X("Gr Liv Area", scale=Scale(domain=(0, 4000))),
            y=Y("SalePrice", axis=Axis(format="e")),
            column="Bldg Type"
        )
```

1.4 Summary Statistics

When there are three or more variables, summarizing the relationships can be difficult, so it is usually more fruitful to produce a visualization. That said, there are a few commonly used summary statistics for multivariate data.

The **covariance matrix** is exactly what its name implies; it is a matrix whose (i, j)th entry is the correlation between variable i and variable j. It is obtained by calling `.cov()` on a `DataFrame`.

```
In [11]: variables = ["Gr Liv Area",
                      "Bedroom AbvGr",
                      "Full Bath",
                      "SalePrice"]

         housing_df[variables].cov()

Out[11]:                 Gr Liv Area  Bedroom AbvGr     Full Bath     SalePrice
         Gr Liv Area    2.555392e+05     216.245415    176.184999  2.854220e+07
         Bedroom AbvGr  2.162454e+02       0.685139      0.164533  9.516233e+03
         Full Bath      1.761850e+02       0.164533      0.305743  2.410074e+04
         SalePrice      2.854220e+07    9516.232724  24100.740965  6.381884e+09
```

Likewise, the **correlation matrix** is just the corresponding matrix of correlations. All of its entries are between -1 and $+1$. It is obtained by calling `.corr()` on a `DataFrame`.

```
In [12]: housing_df[variables].corr()

Out[12]:                Gr Liv Area  Bedroom AbvGr  Full Bath  SalePrice
         Gr Liv Area       1.000000       0.516808   0.630321   0.706780
         Bedroom AbvGr     0.516808       1.000000   0.359489   0.143913
         Full Bath         0.630321       0.359489   1.000000   0.545604
         SalePrice         0.706780       0.143913   0.545604   1.000000
```

Why does it make sense that all of the diagonal entries of this matrix are equal to 1.0?

2 Exercises

Exercises 1-3 deal with the Tips data set (`https://raw.githubusercontent.com/dlsun/data-science-book/mas`

Exercise 1. Make a scatterplot (using `pandas` and `matplotlib`) showing the relationship between the tip and the total bill. Use color to indicate whether the tipper was male or female and the size of each point to represent the party size.

```
In [ ]: tips = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/master/da

        cols = tips["sex"].map({
            "Male": "blue",
            "Female": "red"
        })

        tips.plot.scatter("total_bill", "tip", c=cols, s=[10,20,30,40,50])
        #tips["size"].unique()
```

Exercise 2. Repeat Exercise 1, but using Altair. Can you incorporate even more variables into this figure?

```
In [ ]: Chart(tips).mark_circle().encode(
            x="total_bill",
            y="tip",
            color="sex",
            size="size",
            column="day"
        ).interactive()
```

```
In [ ]: Chart(tips).mark_circle().encode(
            x=X("total_bill", title="Total Bill"),
            y=Y("tip", title="Tip"),
            color=Color("sex", title="Sex"),
            size=Size("size", title="Size"),
            column=Column("day", title="Day")
        )
```

Exercise 3. Calculate the correlation matrix summarizing the pairwise relationships between the quantitative variables in this data set. Interpret what you see.

```
In [33]: variables = ["total_bill",
                      "tip",
                      "size"]
         tips[variables].corr()

Out[33]:             total_bill       tip      size
         total_bill    1.000000  0.675734  0.598315
         tip           0.675734  1.000000  0.489299
         size          0.598315  0.489299  1.000000
```

```
In [103]: tips.corr()

Out[103]:             total_bill       tip      size
          total_bill    1.000000  0.675734  0.598315
          tip           0.675734  1.000000  0.489299
          size          0.598315  0.489299  1.000000
```

```
In [ ]: ?Chart
In [ ]: day_counts = tips.day.value_counts()
        day_order = day_counts[["Sun", "Thur", "Fri", "Sat"]]
        day_order.plot.bar()
```