This subunit will walk you through step three of the Data Science Method: **exploratory data analysis.** Exploratory Data Analysis (EDA) is an approach for summarizing and visualizing the important characteristics and statistical properties of a dataset. Visualizing the data will help you make sense of it to identify emerging themes. Identifying these trends will help you to form hypotheses about the data.

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----86eb12060eac--------------------------------)

[[](https://guymaskall.medium.com/?source=post_page-----86eb12060eac--------------------------------)](https://guymaskall.medium.com/?source=post_page-----86eb12060eac--------------------------------)

[Guy Maskall](https://guymaskall.medium.com/?source=post_page-----86eb12060eac--------------------------------)

Jul 15, 2020

·

5 min read

·

**Exploratory data analysis**

**Visualize! Visualize! Visualize! Oh, and don’t waste ink!**

A picture containing electronics, circuit

Description automatically generated

Image by [Gerd Altmann](https://pixabay.com/users/geralt-9301/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=3706562) from [Pixabay](https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=3706562)

**Data quality**

We previously wrote about issues in data quality. Certainly, many issues can be spotted quickly and clearly right at the start, those missing river level values for example (even if you’re not immediately sure how to handle them). But many issues will only come to light once you dig a little deeper into the data and start to consider the distributions and relationships within and between variables. A person’s height, for example, might very reasonably be somewhere between 50 cm (toddler) and 1.80 m. Their weight may also very reasonably be anywhere between 10 kg and 100 kg. But only when considering both together could you realise there’s a “toddler” weighing 100 kg or a tall adult weighing 15 kg. You suddenly have some outliers on your hands! Thus, exploratory data analysis (EDA) is a step when data quality issues are still very much being explored.

**Complex relationships**

We just brought up an example where you need to compare two variables in unison in order to spot outliers. By implication this was easy; you’d just plot one variable on the x-axis and the other on the y-axis and see what “sticks out”. What if you had age as well? Well, there are graphing tools that allow you to define a z-axis as well, and even rotate the axes dynamically to visualize the data from different directions. What if you had salary as well? It’s getting tricky now, isn’t it? You could start plotting the different combinations of variables as x and y over multiple graphs. But even that’s only considering pairwise relationships.

What if your data had 10 variables? 20? 1000? Fortunately, there are a number of techniques to reduce the dimensionality of your data and visualize patterns. One common technique is principle components analysis (PCA), although there are many others.

**Visualizations**

A skill we can’t emphasise enough when it comes to EDA is data visualization. And this doesn’t always have to be a figure! If it’s a reasonably small number of summary statistics, then a table can often be the most effective, and efficient, way to communicate. We don’t tend to print out so much these days, but imagine how much ink you’d use printing out a moderate sized bar chart that merely conveyed three data values. Perhaps that would be a good candidate to present in a simple table.

There are more and more plot types readily available in packages now, offering many and varied ways to represent your data. But just because they exist, doesn’t mean you need to use them. Don’t be the datavis equivalent of that guy who gives a powerpoint presentation using every single animation and sound effect he could find just because they were there! Think carefully about what message you want to draw out in your figure and craft it accordingly. Some basic types to get you started:

* scatter (or point) plots — great for showing relationships between two variables. Effective use can be made of point colour, size, and even shape. Give some consideration to which features you map to which aesthetic. Don’t map a feature with 20 distinct values to shape unless you’re confident your reader has a very sharp eye! Do you even want to have 20 different values represented anyway? Perhaps just colour the top five categories and lump the rest into ‘Other’. Distinct colours can be informative for discrete (categorical) features. Beware having too many! Tone (of a single colour, e.g. black through to ever lighter shades of grey) can be useful for adding an additional continuous variable to your figure. Many graphics libraries will choose between colours or tones according to data type. Take care what data type your plotting function thinks a feature is! A 1/0 for True/False is a classic example that can appear represented as a tone, when it would be clearer as distinct colours. Above all, explore different options that bring out *your* message effectively and without redundancy. Play around with the alpha value if you have a dense mass of points, and ask yourself whether you really need to put every data point on the page — could you show the same message about the data distribution by plotting one in every thousand points?
* bar plot —a useful favourite for showing a set of discrete values, e.g. counts of categories. Don’t colour each category separately if the x-axis is also the category just because it makes a pretty rainbow! For example, say the x-axis is month and the y-value is butterfly count. Don’t colour the bars by month. But do consider colouring by season, perhaps. That could add something. Or perhaps it would be relevant to colour according to whether the month was particularly wet or dry. That would be an effective use of colour. If you have so many categories (on the x-axis) that the labels are illegible, ask yourself whether you really need to show all of them. Perhaps just a figure showing the top twenty categories gets your point across? And here we come to the “waste of ink” point. If you have two categories, say male and female, then perhaps just put them in a table.
* histogram — great for summarising distributions. Do explore different bin widths to pull out just the right amount of structure in the data that makes sense. No, I can’t tell you what that is.
* line plots — great for plotting a continuous sequence of values. Avoid them if intermediate x-values don’t make sense, such as when dealing with categories. The eye will naturally want to assume the trail of ink joining points is a valid interpolation. Reconsider their use if you don’t have many x-values. Line plots comprising three vertices are not very informative. A personal bugbear is lines that comprise many values (points), but they form something very, very close to a straight line. Does that line plot comprise two points only, one at each end, in which case it’s a poor plot that gives an illusion of a high correlation, or are there 1000 points, which is much stronger evidence for (a very suspicious) correlation. Would a scatter plot be better here? Or at least a plot with points and lines! A single plot with a number of lines coloured by different categories can be an effective way to highlight different trends between the categories.

This is an incredibly abridged selection of plot types, but they are good stalwarts that, with some careful thought, you can make effective use of time and time again. Above all, and this cannot be stressed enough, **think about what it is you’re trying to communicate in your plot** and try to effectively convey it, and only it, with a minimum of ink and extraneous fuss and bother.

**About this article**

This is the fourth article of a linked series written to provide a straightforward introduction to getting started with the data science process. You can find the introduction [here](https://medium.com/@guymaskall/the-data-science-method-dsm-35200eb4984), the previous article [here](https://medium.com/@guymaskall/data-wrangling-69fe7f4aa820), and the next article in the series [here](https://medium.com/@guymaskall/pre-processing-and-training-data-d16cc12dfbac).

How do we get from data to answers? Exploratory data analysis is a process for exploring datasets, answering questions, and visualizing results. This course presents the tools you need to clean and validate data, to visualize distributions and relationships between variables, and to use regression models to predict and explain. You'll explore data related to demographics and health, including the National Survey of Family Growth and the General Social Survey. But the methods you learn apply to all areas of science, engineering, and business. You'll use Pandas, a powerful library for working with data, and other core Python libraries including NumPy and SciPy, StatsModels for regression, and Matplotlib for visualization. With these tools and skills, you will be prepared to work with real data, make discoveries, and present compelling results.

#### Read, clean, and validate

0%

The first step of almost any data project is to read the data, check for errors and special cases, and prepare data for analysis. This is exactly what you'll do in this chapter, while working with a dataset obtained from the National Survey of Family Growth.

#### Read, clean, and validate

0%

The first step of almost any data project is to read the data, check for errors and special cases, and prepare data for analysis. This is exactly what you'll do in this chapter, while working with a dataset obtained from the National Survey of Family Growth.

**Daily XP100**

##### Exercise

##### Exercise

# Exploring the NSFG data

To get the number of rows and columns in a DataFrame, you can read its shape attribute.

To get the column names, you can read the columns attribute. The result is an Index, which is a Pandas data structure that is similar to a list. Let's begin exploring the NSFG data! It has been pre-loaded for you into a DataFrame called nsfg.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Calculate the number of rows and columns in the DataFrame nsfg.

Nice job! Remember these attributes and methods; they are useful when you are exploring a new dataset. It's now time to check for errors and prepare the data for analysis. Keep going!

# Display the number of rows and columns

nsfg.shape

# Display the names of the columns

nsfg.columns

# Select column birthwgt\_oz1: ounces

ounces = nsfg['birthwgt\_oz1']

# Print the first 5 elements of ounces

print(ounces.head())

# Display the number of rows and columns

nsfg.shape

# Display the names of the columns

nsfg.columns

# Select column birthwgt\_oz1: ounces

ounces = nsfg['birthwgt\_oz1']

# Print the first 5 elements of ounces

print(ounces.head())

0 4.0

1 12.0

2 4.0

3 NaN

4 13.0

Name: birthwgt\_oz1, dtype: float64

<script.py> output:

0 4.0

1 12.0

2 4.0

3 NaN

4 13.0

Name: birthwgt\_oz1, dtype: float64

# Display the number of rows and columns

nsfg.shape

(9358, 10)

# Display the number of rows and columns

nsfg.shape

# Display the names of the columns

nsfg.columns

Index(['caseid', 'outcome', 'birthwgt\_lb1', 'birthwgt\_oz1', 'prglngth', 'nbrnaliv', 'agecon', 'agepreg', 'hpagelb', 'wgt2013\_2015'], dtype='object')

# Display the number of rows and columns

nsfg.shape

# Display the names of the columns

nsfg.columns

# Select column birthwgt\_oz1: ounces

ounces = nsfg['birthwgt\_oz1']

# Display the number of rows and columns

nsfg.shape

# Display the names of the columns

nsfg.columns

# Select column birthwgt\_oz1: ounces

ounces = nsfg['birthwgt\_oz1']

# Print the first 5 elements of ounces

print(ounces).head()

0 4.0

1 12.0

2 4.0

3 NaN

4 13.0

...

9353 11.0

9354 7.0

9355 6.0

9356 3.0

9357 5.0

Name: birthwgt\_oz1, Length: 9358, dtype: float64

**Daily XP200**

**Clean and Validate**

**50 XP**

**1. Clean and Validate**

In the previous lesson, we read data from the National Survey of Family Growth and selected a column from a DataFrame. In this lesson, we'll check for errors and prepare the data for analysis.

**2. Selecting columns**

We'll use the same DataFrame we used in the previous lesson, nsfg, which contains one row for each pregnancy in the survey. I'll select the variable birthwgt\_lb1, which contains the pound part of birth weight, and assign it to pounds. And birthwgt\_oz1 contains the ounce part of birth weight, so I'll assign that to ounces.

**3. Value counts**

Before we do anything with this data, we have to validate it. One part of validation is confirming that we are interpreting the data correctly. We can use value\_counts() to see what values appear in pounds and how many times each value appears. By default, the results are sorted with the most frequent value first, so I use sort\_index() to sort them by value instead, with the lightest babies first and heaviest babies last. As we'd expect, the most frequent values are 6-8 pounds, but there are some very light babies, a few very heavy babies, and two values, 98, and 99, that indicate missing data.

**4. Value counts**

We can validate the results by comparing them to the codebook, which lists the values and their frequencies. The results here agree with the codebook, so we have some confidence that we are reading and interpreting the data correctly.

**5. Describe**

Another way to validate the data is with describe(), which computes summary statistics like the mean, standard deviation, min, and max. Here are the results for pounds. count is the number of values. The minimum and maximum values are 0 and 99, and the 50th percentile, which is the median, is 7. The mean is about 8.05, but that doesn't mean much because it includes the special values 98 and 99. Before we can really compute the mean, we have to replace those values with NaN to represent missing data.

**6. Replace**

The replace() method does what we want; it takes a list of values we want to replace and the value we want to replace them with. np dot nan means we are getting the special value NaN from the NumPy library, which is imported as np. The result from replace() is a new Series, which I assign back to pounds. Remember that the mean of the original series was about 8 point 05 pounds. The mean of the new series is about 6 point 7 pounds. It makes a big difference when you remove a few 99-pound babies! Instead of making a new Series, you can call replace() with inplace=True, which modifies the existing Series "in place", that is, without making a copy. Here's what that looks like for ounces. Since we didn't make a new series, we don't have to assign it back to ounces.

**7. Arithmetic with Series**

Now we want to combine pounds and ounces into a single Series that contains total birth weight. Arithmetic operators work with Series objects; so, to convert from ounces to pounds, we can divide by 16 (there are 16 ounces in a pound). Then we can add the two Series objects to get the total. Here are the results. The mean is about 7 point 1, which is a little more than what we got before we added in the ounces part. Now we're close to answering our original question, the average birth weight for babies in the U.S., but as we'll see in the next lesson, we're not there yet.

**8. Let's practice!**

Let's first practice what we learned in this video.

##### Exercise

##### Exercise

# Validate a variable

In the NSFG dataset, the variable 'outcome' encodes the outcome of each pregnancy as shown below:

| **value** | **label** |
| --- | --- |
| 1 | Live birth |
| 2 | Induced abortion |
| 3 | Stillbirth |
| 4 | Miscarriage |
| 5 | Ectopic pregnancy |
| 6 | Current pregnancy |

The nsfg DataFrame has been pre-loaded for you. Explore it in the IPython Shell and use the methods Allen showed you in the video to answer the following question: How many pregnancies in this dataset ended with a live birth?

##### Instructions

**50 XP**

##### Possible Answers

* 

6489

* 

9538

* 

1469

* 

6

# Display the number of rows and columns nsfg.shape # Display the names of the columns nsfg.columns # Select column birthwgt\_oz1: ounces ounces = nsfg['birthwgt\_oz1'] # Print the first 5 elements of ounces print(ounces.head())

[5]:

outcome = nsfg['outcome']

In [10]:

print(outcome.value\_counts().sort\_index())

1 6489

2 947

3 86

4 1469

5 118

6 249

Name: outcome, dtype: int64

[5]:

outcome = nsfg['outcome']

**Daily XP100**

##### Exercise

##### Exercise

# Clean a variable

In the NSFG dataset, the variable 'nbrnaliv' records the number of babies born alive at the end of a pregnancy.

If you use .value\_counts() to view the responses, you'll see that the value 8 appears once, and if you consult the codebook, you'll see that this value indicates that the respondent refused to answer the question.

Your job in this exercise is to replace this value with np.nan. Recall from the video how Allen replaced the values 98 and 99 in the ounces column using the .replace() method:

ounces.replace([98, 99], np.nan, inplace=True)

##### Instructions

**100 XP**

* In the 'nbrnaliv' column, replace the value 8, in place, with the special value NaN.
* Confirm that the value 8 no longer appears in this column by printing the values and their frequencies.
* # Replace the value 8 with NaN
* nsfg['\_\_\_\_'].\_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_)
* # Print the values and their frequencies
* print(nsfg['\_\_\_\_'].\_\_\_\_())

# Display the number of rows and columns nsfg.shape

# Display the names of the columns nsfg.columns

# Select column birthwgt\_oz1: ounces ounces = nsfg['birthwgt\_oz1']

# Print the first 5 elements of ounces print(ounces.head())

# Replace the value 8 with NaN

nsfg['nbrnaliv'].replace(8, np.nan, inplace=True)

# Print the values and their frequencies

print(nsfg['nbrnaliv'].value\_counts())

# Replace the value 8 with NaN

nsfg['nbrnaliv'].replace(8, np.nan, inplace=True)

# Print the values and their frequencies

print(nsfg['nbrnaliv'].value\_counts())

1.0 6379

2.0 100

3.0 5

Name: nbrnaliv, dtype: int64

<script.py> output:

1.0 6379

2.0 100

3.0 5

Name: nbrnaliv, dtype: int64

# Compute a variable

For each pregnancy in the NSFG dataset, the variable 'agecon' encodes the respondent's age at conception, and 'agepreg' the respondent's age at the end of the pregnancy.

Both variables are recorded as integers with two implicit decimal places, so the value 2575 means that the respondent's age was 25.75.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* Select 'agecon' and 'agepreg', divide them by 100, and assign them to the local variables agecon and agepreg.

# Select the columns and divide by 100

agecon = \_\_\_\_

agepreg = \_\_\_\_

Compute the difference, which is an estimate of the duration of the pregnancy. Keep in mind that for each pregnancy, agepreg will be larger than agecon.

# Select the columns and divide by 100

agecon = nsfg['agecon'] / 100

agepreg = nsfg['agepreg'] / 100

# Compute the difference

preg\_length = \_\_\_\_

# Select the columns and divide by 100

agecon = nsfg['agecon'] /100

agepreg = nsfg['agepreg'] /100

print(agecon)

print(agepreg)

0 20.00

1 22.91

2 32.41

3 36.50

4 21.91

...

9353 17.58

9354 17.41

9355 20.91

9356 34.50

9357 36.83

Name: agecon, Length: 9358, dtype: float64

0 20.75

1 23.58

2 33.08

3 NaN

4 22.66

...

9353 18.25

9354 18.16

9355 21.58

9356 35.25

9357 37.58

Name: agepreg, Length: 9358, dtype: float64

<script.py> output:

0 20.00

1 22.91

2 32.41

3 36.50

4 21.91

...

9353 17.58

9354 17.41

9355 20.91

9356 34.50

9357 36.83

Name: agecon, Length: 9358, dtype: float64

0 20.75

1 23.58

2 33.08

3 NaN

4 22.66

...

9353 18.25

9354 18.16

9355 21.58

9356 35.25

9357 37.58

Name: agepreg, Length: 9358, dtype: float64

Use .describe() to compute the mean duration and other summary statistics.

# Select the columns and divide by 100

agecon = nsfg['agecon'] / 100

agepreg = nsfg['agepreg'] / 100

# Compute the difference

preg\_length = agepreg - agecon

# Compute summary statistics

print(\_\_\_\_)

# Select the columns and divide by 100

agecon = nsfg['agecon'] / 100

agepreg = nsfg['agepreg'] / 100

# Compute the difference

preg\_length = agepreg - agecon

# Compute summary statistics

print(preg\_length.describe())

# Select the columns and divide by 100

agecon = nsfg['agecon'] / 100

agepreg = nsfg['agepreg'] / 100

# Compute the difference

preg\_length = agepreg - agecon

# Compute summary statistics

print(preg\_length.describe())

count 9109.000

mean 0.552

std 0.271

min 0.000

25% 0.250

50% 0.670

75% 0.750

max 0.920

dtype: float64

Good job. A variable that's computed from other variables is sometimes called a 'recode'. It's now time to get back to the motivating question for this chapter: what is the average birth weight for babies in the U.S.? See you in the next video!

**Daily XP300**

**Filter and visualize**

**50 XP**

**1. Filter and Visualize**

Let's get back to the motivating question for this chapter: what is the average birth weight for babies in the U.S.?

**2. Histogram**

In the previous lesson, we used data from the NSFG to compute birth weight in pounds and we stored the result in a Series called birth\_weight. Let's see what the distribution of those values looks like. We'll use the pyplot submodule from the matplotlib visualization library, which we import as plt. Pyplot provides hist(), which takes a Series and plots a histogram; that is, it shows the values and how often they appear. However, pyplot doesn't work with NaNs, so we have to use dropna(), which makes a new Series that contains only the valid values. The second argument, bins, tells hist to divide the range of weights into 30 intervals, called "bins", and count how many values fall in each bin. hist() takes other arguments that specify the type and appearance of the histogram; you will have a chance to explore these options in the next exercise. To label the axes we'll use xlabel() and ylabel(), and finally, to display the plot, we'll use plt dot show().

**3. Histogram**

And here's what the results look like. The x-axis is birth weight in pounds, divided into 30 bins. The y-axis is the number of births in each bin. The distribution looks a little like a bell curve, but the tail is longer on the left than on the right; that is, there are more light babies than heavy babies. That makes sense because the distribution includes some babies that were born preterm. The most common duration for pregnancy is 39 weeks, which is "full term"; "preterm" is usually defined to be less than 37 weeks.

**4. Boolean Series**

To see which babies are preterm, we can use the prglngth column, which records pregnancy length in weeks. When you compare a Series to a value, the result is a Boolean Series; that is, each element is a Boolean value, True or False. In this case, it's True for each preterm baby and False otherwise. We can use head() to see the first 5 elements.

**5. Boolean Series**

If you compute the sum of a Boolean Series, it treats True as 1 and False as 0, so the sum is the number of Trues, which is the number of preterm babies, about 3700. If you compute the mean, you get the fraction of Trues; in this case, it's close to 0.4; that is, about 40% of the births in this dataset are preterm.

**6. Filtering**

We can use a Boolean Series as a filter; that is, we can select only rows that satisfy a condition or meet some criterion. For example, we can use preterm and the bracket operator to select values from birth\_weight, so preterm\_weight contains birth weights for preterm babies. To select full-term babies, we can use the tilde operator, which is "logical NOT" or inverse; it makes the Trues false and the Falses true. Not surprisingly, full term babies are heavier, on average, than preterm babies.

**7. Filtering**

If you have two Boolean Series, you can use logical operators to combine them; ampersand is the logical AND operator, and the vertical bar or pipe is logical OR.

**8. Resampling**

There's one more thing we have to do before we can answer our question: resampling. The NSFG is not exactly representative of the U.S. population; by design, some groups are more likely to appear in the sample than others; they are "oversampled". Oversampling helps to ensure that you have enough people in every subgroup to get reliable statistics, but it makes the analysis a little more complicated. However, we can correct for oversampling by resampling. I won't get into the details here, but I have provided a function called resample\_rows\_weighted() that you can use for the exercises. If you are interested in learning more about resampling, check out DataCamp's statistics courses.

**9. Finish it off!**

Now we have everything we need to answer the motivating question. Let's get to it.

**Daily XP350**

##### Exercise

##### Exercise

# Make a histogram

Histograms are one of the most useful tools in exploratory data analysis. They quickly give you an overview of the distribution of a variable, that is, what values the variable can have, and how many times each value appears.

As we saw in a previous exercise, the NSFG dataset includes a variable 'agecon' that records age at conception for each pregnancy. Here, you're going to plot a histogram of this variable. You'll use the bins parameter that you saw in the video, and also a new parameter - histtype - which you can read more about [here](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html) in the matplotlib documentation. Learning how to read documentation is an essential skill. If you want to learn more about matplotlib, you can check out DataCamp's [Introduction to Data Visualization with Matplotlib](https://www.datacamp.com/courses/introduction-to-data-visualization-with-matplotlib) course.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))

Plot a histogram of agecon with 20 bins.

 [2](javascript:void(0))

Adapt your code to make an unfilled histogram by setting the parameter histtype to be 'step'.

 Plot the histogram

# Label the axes

plt.xlabel('Age at conception')

plt.ylabel('Number of pregnancies')

# Show the figure

plt.show()

# Select the columns and divide by 100 agecon = nsfg['agecon'] / 100 agepreg = nsfg['agepreg'] / 100 # Compute the difference preg\_length = agepreg - agecon # Compute summary statistics print(preg\_length.describe())

# Plot the histogram plt.hist(agecon.dropna(), bins=20) # Label the axes plt.xlabel('Age at conception') plt.ylabel('Number of pregnancies') # Show the figure plt.show()

# Plot the histogram

plt.hist(agecon, bins=20, histtype='step')

# Label the axes

plt.xlabel('Age at conception')

plt.ylabel('Number of pregnancies')

# Show the figure

plt.show()

# Select the columns and divide by 100

agecon = nsfg['agecon'] / 100

agepreg = nsfg['agepreg'] / 100

# Compute the difference

preg\_length = agepreg - agecon

# Compute summary statistics

print(preg\_length.describe())

# Plot the histogram

plt.hist(agecon.dropna(), bins=20)

# Label the axes

plt.xlabel('Age at conception')

plt.ylabel('Number of pregnancies')

# Show the figure

plt.show()

# Plot the histogram

plt.hist(agecon, bins=20, histtype='step')

# Label the axes

plt.xlabel('Age at conception')

plt.ylabel('Number of pregnancies')

# Show the figure

plt.show()

Nice job! matplotlib functions provide a lot of options; be sure to read the documentation so you know what they can do.

# Compute birth weight

Now let's pull together the steps in this chapter to compute the average birth weight for full-term babies.

I've provided a function, resample\_rows\_weighted, that takes the NSFG data and resamples it using the sampling weights in wgt2013\_2015. The result is a sample that is representative of the U.S. population.

Then I extract birthwgt\_lb1 and birthwgt\_oz1, replace special codes with NaN, and compute total birth weight in pounds, birth\_weight.

# Resample the data

nsfg = resample\_rows\_weighted(nsfg, 'wgt2013\_2015')

# Clean the weight variables

pounds = nsfg['birthwgt\_lb1'].replace([98, 99], np.nan)

ounces = nsfg['birthwgt\_oz1'].replace([98, 99], np.nan)

# Compute total birth weight

birth\_weight = pounds + ounces/16

##### Instructions

**100 XP**

* Make a Boolean Series called full\_term that is true for babies with 'prglngth' greater than or equal to 37 weeks.
* Use full\_term and birth\_weight to select birth weight in pounds for full-term babies. Store the result in full\_term\_weight.
* Compute the mean weight of full-term babies.
* # Create a Boolean Series for full-term babies
* full\_term = \_\_\_\_
* # Select the weights of full-term babies
* full\_term\_weight = \_\_\_\_
* # Compute the mean weight of full-term babies
* print(\_\_\_\_)

# Plot the histogram plt.hist(agecon, bins=20, histtype='step') # Label the axes plt.xlabel('Age at conception') plt.ylabel('Number of pregnancies') # Show the figure plt.show()

# Create a Boolean Series for full-term babies

full\_term = nsfg['prglngth']>= 37

# Select the weights of full-term babies

full\_term\_weight = birth\_weight[full\_term]

# Compute the mean weight of full-term babies

print(full\_term\_weight.mean())

# Create a Boolean Series for full-term babies

full\_term = nsfg['prglngth']>= 37

# Select the weights of full-term babies

full\_term\_weight = birth\_weight[full\_term]

# Compute the mean weight of full-term babies

print(full\_term\_weight.mean())

7.392597951914515

Nice job. You're almost done, but there's one last thing we have to check...

**Daily XP550**

##### Exercise

##### Exercise

# Filter

In the previous exercise, you computed the mean birth weight for full-term babies; you filtered out preterm babies because their distribution of weight is different.

The distribution of weight is also different for multiple births, like twins and triplets. In this exercise, you'll filter them out, too, and see what effect it has on the mean.

##### Instructions

**100 XP**

* Use the variable 'nbrnaliv' to make a Boolean Series that is True for single births (where 'nbrnaliv' equals 1) and False otherwise.
* Use Boolean Series and logical operators to select single, full-term babies and compute their mean birth weight.
* For comparison, select multiple, full-term babies and compute their mean birth weight.
* # Filter full-term babies
* full\_term = nsfg['prglngth'] >= 37
* # Filter single births
* single = \_\_\_\_
* # Compute birth weight for single full-term babies
* single\_full\_term\_weight = birth\_weight[\_\_\_\_ & \_\_\_\_]
* print('Single full-term mean:', single\_full\_term\_weight.mean())
* # Compute birth weight for multiple full-term babies
* mult\_full\_term\_weight = birth\_weight[\_\_\_\_ & \_\_\_\_]
* print('Multiple full-term mean:', mult\_full\_term\_weight.mean())

# Create a Boolean Series for full-term babies full\_term = nsfg['prglngth']>= 37 # Select the weights of full-term babies full\_term\_weight = birth\_weight[full\_term] # Compute the mean weight of full-term babies print(full\_term\_weight.mean())

# Filter full-term babies

full\_term = nsfg['prglngth'] >= 37

# Filter single births

single = nsfg['nbrnaliv']==1

# Compute birth weight for single full-term babies

single\_full\_term\_weight = birth\_weight[single & full\_term]

print('Single full-term mean:', single\_full\_term\_weight.mean())

# Compute birth weight for multiple full-term babies

mult\_full\_term\_weight = birth\_weight[~single & full\_term]

print('Multiple full-term mean:', mult\_full\_term\_weight.mean())

# Filter full-term babies

full\_term = nsfg['prglngth'] >= 37

# Filter single births

single = nsfg['nbrnaliv']==1

# Compute birth weight for single full-term babies

single\_full\_term\_weight = birth\_weight[single & full\_term]

print('Single full-term mean:', single\_full\_term\_weight.mean())

# Compute birth weight for multiple full-term babies

mult\_full\_term\_weight = birth\_weight[~single & full\_term]

print('Multiple full-term mean:', mult\_full\_term\_weight.mean())

Single full-term mean: 7.40297320308299

Multiple full-term mean: 5.784722222222222

<script.py> output:

Single full-term mean: 7.40297320308299

Multiple full-term mean: 5.784722222222222

Congratulations on completing Chapter 1! Now that we have clean data, we're ready to explore. Coming up in Chapter 2, we'll look at distributions of variables in the General Social Survey and explore the relationship between education and income.

**Daily XP120**

##### Exercise

##### Exercise

# Make a PMF

The GSS dataset has been pre-loaded for you into a DataFrame called gss. You can explore it in the IPython Shell to get familiar with it.

In this exercise, you'll focus on one variable in this dataset, 'year', which represents the year each respondent was interviewed.

The Pmf class you saw in the video has already been created for you. You can access it outside of DataCamp via the [empiricaldist](https://pypi.org/project/empiricaldist/) library.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Make a PMF for year with normalize=False and display the result.
* # Compute the PMF for year
* pmf\_year = Pmf(\_\_\_\_, \_\_\_\_=\_\_\_\_)
* # Print the result
* print(pmf\_year)

# Filter full-term babies full\_term = nsfg['prglngth'] >= 37 # Filter single births single = nsfg['nbrnaliv']==1 # Compute birth weight for single full-term babies single\_full\_term\_weight = birth\_weight[single & full\_term] print('Single full-term mean:', single\_full\_term\_weight.mean()) # Compute birth weight for multiple full-term babies mult\_full\_term\_weight = birth\_weight[~single & full\_term] print('Multiple full-term mean:', mult\_full\_term\_weight.mean())

# Compute the PMF for year

pmf\_year = Pmf(gss['year'], normalize=False)

# Print the result

print(pmf\_year)

1972 1613

1973 1504

1974 1484

1975 1490

1976 1499

1977 1530

1978 1532

1980 1468

1982 1860

1983 1599

1984 1473

1985 1534

1986 1470

1987 1819

1988 1481

1989 1537

1990 1372

1991 1517

1993 1606

1994 2992

1996 2904

1998 2832

2000 2817

2002 2765

2004 2812

2006 4510

2008 2023

2010 2044

2012 1974

2014 2538

2016 2867

Name: Pmf, dtype: int64

##### nstructions 2/2

**50 XP**

* [2](javascript:void(0))

#### Question

How many respondents were interviewed in 2016?

##### Possible Answers

* 

**2867 This is the correct answer no. of respondents in 2016**

* 

1613

* 

2538

* 

0.045897

# Compute the PMF for year

pmf\_year = Pmf('year', normalize=False)

# Print the result

print(pmf\_year)

# Compute the PMF for year

pmf\_year = Pmf('year', normalize=False)

# Print the result

print(pmf\_year)

year 1

Name: Pmf, dtype: int64

In [1]:

print(gss['year'])

0 1972

1 1972

2 1972

3 1972

4 1972

...

62461 2016

62462 2016

62463 2016

62464 2016

62465 2016

Name: year, Length: 62466, dtype: int64

# Compute the PMF for year

pmf\_year = Pmf(gss['year'], normalize=False)

# Print the result

print(pmf\_year)

1972 1613

1973 1504

1974 1484

1975 1490

1976 1499

1977 1530

1978 1532

1980 1468

1982 1860

1983 1599

1984 1473

1985 1534

1986 1470

1987 1819

1988 1481

1989 1537

1990 1372

1991 1517

1993 1606

1994 2992

1996 2904

1998 2832

2000 2817

2002 2765

2004 2812

2006 4510

2008 2023

2010 2044

2012 1974

2014 2538

**2016 2867**

Name: Pmf, dtype: int64

Correct. The PMF makes it easy to extract insights like this. Time now to visualize the PMF for the 'age' variable of this GSS dataset!

Correct. The PMF makes it easy to extract insights like this. Time now to visualize the PMF for the 'age' variable of this GSS dataset!

# Select the age column

age = \_\_\_\_

# Compute the PMF for year pmf\_year = Pmf(gss['year'], normalize=False) # Print the result print(pmf\_year)

Make a normalized PMF of age. Store the result in pmf\_age.

# Select the age column

age = gss['age']

# Make a PMF of age

pmf\_age = Pmf(\_\_\_\_)

Plot pmf\_age as a bar chart.

# Select the age column

age = gss['age']

# Make a PMF of age

pmf\_age = Pmf(age)

# Plot the PMF

# Label the axes

plt.xlabel('Age')

plt.ylabel('PMF')

plt.show()

Please rate this exercise:

Press enter to

# Select the age column

age = gss['age']

# Make a PMF of age

pmf\_age = Pmf(age, normalize=False)

# Select the age column

age = gss['age']

# Make a PMF of age

pmf\_age = Pmf(age)

# Select the age column

age = gss['age']

# Make a PMF of age

pmf\_age = Pmf(age)

# Plot the PMF

pmf\_age.bar(label='age')

# Label the axes

plt.xlabel('Age')

plt.ylabel('PMF')

plt.show()

**Cumulative distribution functions**

**50 XP**

**1. Cumulative distribution functions**

In the previous lesson, we saw the probability mass function, or PMF, which represents the possible values in a distribution and their probabilities. In this lesson, we'll see another way to represent a distribution, the cumulative distribution function, or CDF. CDFs are useful for some computations; they are also a great way to visualize and compare distributions.

**2. From PMF to CDF**

You might remember that a PMF tells you - if you draw a random value from a distribution - what's the chance of getting x, for any given value of x. The CDF is similar; if you draw a random value from a distribution, it tells you the chance of getting a value less than or equal to x.

**3. Example**

As a small example, suppose the distribution only has 5 elements, 1, 2, 2, 3, and 5. The PMF says that the probability of value 1 is 1/5; the probability of value 2 is 2/5, and the probabilities for 3 and 5 are 1/5. The CDF is the cumulative sum of the probabilities from the PMF. For example, the CDF of 2 is three fifths, because three out of 5 values in the distribution are less than or equal to 2. The CDF of 5 is 1, or 100%, because all of the values in the distribution are less than or equal to 5.

**4. Make and plot a CDF**

In the code for this course, I provide a Cdf class which is similar to the Pmf class we've seen. As a function, Cdf takes any kind of sequence and returns a new Cdf object. In this example, the sequence is the ages of respondents in the General Social Survey. The Cdf provides plot, which plots the CDF as a line. Here's what it looks like. The x-axis is the ages, from 18 to 89. The y-axis is the cumulative probabilities, from 0 to 1.

**5. Evaluating the CDF**

The Cdf object can be used as a function, so if you give it an age, it returns the corresponding probability. In this example, the age is the quantity, q, which is 51. The corresponding probability is p, which is 0.66. That means that about 66% of the respondents are 51 years old or younger. The arrow in the figure shows how you could read this value from the CDF, at least approximately.

**6. Evaluating the inverse CDF**

The CDF is an invertible function, which means that if you have a probability, p, you can look up the corresponding quantity, q. In this example, I look up the probability 0.25, which returns 30. That means that 25% of the respondents are age 30 or less. Another way to say the same thing is "age 30 is the 25th percentile of this distribution". I also look up probability 0.75, which returns 57, so 75% of the respondents are 57 or younger. Again, the arrows in the figure show how you could read these values from the CDF. By the way, the distance from the 25th to the 75th percentile is called the interquartile range, or IQR. It measures the spread of the distribution, so it is similar to standard deviation or variance. Because it is based on percentiles, it doesn't get thrown off by extreme values or outliers, the way variance does. So IQR can be more "robust" than variance, which means it works well even if there are errors in the data or extreme values.

**7. Let's practice!**

In the next lesson, we'll use CDFs to compare distributions between groups. But first, you can practice making and plotting CDFs, and reading them forward and backward

**Daily XP400**

##### Exercise

##### Exercise

# Make a CDF

In this exercise, you'll make a CDF and use it to determine the fraction of respondents in the GSS dataset who are OLDER than 30.

The GSS dataset has been preloaded for you into a DataFrame called gss.

As with the Pmf class from the previous lesson, the Cdf class you just saw in the video has been created for you, and you can access it outside of DataCamp via the [empiricaldist](https://pypi.org/project/empiricaldist/) library.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Select the 'age' column. Store the result in age.
* # Select the age column
* age = \_\_\_\_
* script.py

1

2



* IPython Shell
* Slides
* Notes

# Filter full-term babies

full\_term = nsfg['prglngth'] >= 37

# Filter single births

single = nsfg['nbrnaliv']==1

# Compute birth weight for single full-term babies

single\_full\_term\_weight = birth\_weight[single & full\_term]

print('Single full-term mean:', single\_full\_term\_weight.mean())

# Compute birth weight for multiple full-term babies

mult\_full\_term\_weight = birth\_weight[~single & full\_term]

print('Multiple full-term mean:', mult\_full\_term\_weight.mean())

Compute the CDF of age. Store the result in cdf\_age.

# Select the age column

age = gss['age']

# Compute the CDF of age

cdf\_age = Cdf(gss['age'])

# Select the age column

age = gss['age']

# Select the age column

age = gss['age']

# Compute the CDF of age

cdf\_age = Cdf(gss['age'])

Calculate the CDF of 30

# Select the age column

age = gss['age']

# Compute the CDF of age

cdf\_age = Cdf(age)

# Calculate the CDF of 30

print(\_\_\_\_)

# Select the age column

age = gss['age']

# Compute the CDF of age

cdf\_age = Cdf(gss['age'])

# Select the age column

age = gss['age']

# Compute the CDF of age

cdf\_age = Cdf(age)

# Calculate the CDF of 30

print(cdf\_age(30))

0.2539137136526388

#### Question

What fraction of the respondents in the GSS dataset are OLDER than 30?

##### Possible Answers

* 

**Approximately 75% This is the answer**

* 

Approximately 65%

* 

Approximately 45%

* 

Approximately 25%

**Daily XP493**

##### Exercise

##### Exercise

# Compute IQR

Recall from the video that the interquartile range (IQR) is the difference between the 75th and 25th percentiles. It is a measure of variability that is robust in the presence of errors or extreme values.

In this exercise, you'll compute the interquartile range of income in the GSS dataset. Income is stored in the 'realinc' column, and the CDF of income has already been computed and stored in cdf\_income.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))

Calculate the 75th percentile of income and store it in percentile\_75th.

# Calculate the 75th percentile

age=gss['age'] cdf\_age= Cdf(gss['age']) print(cdf\_age(.45))

Calculate the 25th percentile of income and store it in percentile\_25th.

# Calculate the 75th percentile

percentile\_75th = cdf\_income.inverse(0.75)

print(percentile\_75th)

# Calculate the 25th percentile

percentile\_25th = cdf\_income.inverse(0.25)

print(percentile\_25th)

43426.0

13750.0

Calculate the interquartile range of income. Store the result in iqr.

# Calculate the 75th percentile

percentile\_75th = cdf\_income.inverse(0.75)

# Calculate the 25th percentile

percentile\_25th = cdf\_income.inverse(0.25)

# Calculate the interquartile range

iqr = \_\_\_\_ - \_\_\_\_

# Print the interquartile range

print(iqr)

# Calculate the 75th percentile

percentile\_75th = cdf\_income.inverse(0.75)

# Calculate the 25th percentile

percentile\_25th = cdf\_income.inverse(0.25)

# Calculate the interquartile range

iqr = percentile\_75th - percentile\_25th

# Print the interquartile range

print(iqr)

# Calculate the 75th percentile

percentile\_75th = cdf\_income.inverse(0.75)

print(percentile\_75th)

# Calculate the 25th percentile

percentile\_25th = cdf\_income.inverse(0.25)

print(percentile\_25th)

43426.0

13750.0

<script.py> output:

43426.0

13750.0

# Calculate the 75th percentile

percentile\_75th = cdf\_income.inverse(0.75)

# Calculate the 25th percentile

percentile\_25th = cdf\_income.inverse(0.25)

# Calculate the interquartile range

iqr = percentile\_75th - percentile\_25th

# Print the interquartile range

print(iqr)

29676.0

#### Question

What is the interquartile range (IQR) of income in the GSS dataset?

##### Possible Answers

* 

**Approximately 29676 This is the answer**

* 

Approximately 26015

* 

Approximately 34702

* 

Approximately 30655

**Daily XP593**

##### Exercise

##### Exercise

# Plot a CDF

The distribution of income in almost every country is long-tailed; that is, there are a small number of people with very high incomes.

In the GSS dataset, the variable 'realinc' represents total household income, converted to 1986 dollars. We can get a sense of the shape of this distribution by plotting the CDF.

##### Instructions

**100 XP**

* Select 'realinc' from the gss dataset.
* Make a Cdf object called cdf\_income.
* Create a plot of cdf\_income using .plot().
* # Select realinc
* income = \_\_\_\_
* # Make the CDF
* cdf\_income = \_\_\_\_
* # Plot it
* \_\_\_\_
* # Label the axes
* plt.xlabel('Income (1986 USD)')
* plt.ylabel('CDF')
* plt.show()

# Select realinc

income = gss['realinc']

# Make the CDF

cdf\_income = Cdf(income)

# Plot it

cdf\_income.plot()

# Label the axes

plt.xlabel('Income (1986 USD)')

plt.ylabel('CDF')

plt.show()

# Select realinc income = gss['realinc'] # Make the CDF cdf\_income = Cdf(income) # Plot it cdf\_income.plot() # Label the axes plt.xlabel('Income (1986 USD)') plt.ylabel('CDF') plt.show()

**Daily XP693**

**Comparing distributions**

**50 XP**

**1. Comparing distributions**

So far we've seen two ways to represent distributions, PMFs and CDFs. In this lesson, we'll use PMFs and CDFs to compare distributions, and we'll see the pros and cons of each.

**2. Multiple PMFs**

One way to compare distributions is to plot multiple PMFs on the same axes. For example, suppose we want to compare the distribution of age for male and female respondents. First I'll create a boolean Series that's true for male respondents. And I'll extract the age column. Now I can select ages for the male and female respondents. And plot a Pmf for each. Of course I always remember to label the axes!

**3. Age PMFs**

Here's the result. It looks like there are more men in their twenties, maybe. And there are more women in their 70s and 80s. In between, the plot is pretty noisy; most of these differences are just random variations.

**4. Multiple CDFs**

We can do the same thing with CDFs. Here's the code: everything is the same except I replaced Pmf with Cdf.

**5. Age CDFs**

And here is the result. In general, CDFs are smoother than PMFs. Because they smooth out randomness, we can often get a better view of real differences between distributions. In this case, the lines overlap over the whole range; that is, the distributions are nearly identical. But we can see the blue line to the left of the orange line across the distribution, which shows that men are younger at every percentile. Or, another way to think of it: for every age, the fraction of men below that age is more than the fraction of women below that age. But not by very much.

**6. Income distribution**

As another example, let's look at household income and compare the distribution before and after 1995 (I chose 1995 because it's roughly the midpoint of the survey). The variable realinc represents household income in 1986 dollars. I'll make a boolean Series to select respondents interviewed before 1995. Now I can plot the PMFs. And label the axes.

**7. Income PMFs**

Here's what it looks like. There are a lot of unique values in this distribution, and none of them appear very often. The PMF is so noisy, we can't really see the shape of the distribution. It looks like there are more people with high incomes after 1995, but it's hard to tell. We can get a clearer picture with a CDF.

**8. Income CDFs**

Here's the code to generate the CDFs.

**9. Income CDFs**

And here are the results. Below $30,000 the CDFs are almost identical; above that, we can see that the orange distribution is shifted to the right. In other words, the fraction of people with high incomes is about the same, but the income of high earners has increased.

**10. Let's practice!**

In general, I recommend CDFs for exploratory analysis. They give you a clear view of the distribution, without too much noise, and they are good for comparing distributions, especially if you have more than two. In the exercises for this lesson, you'll have a chance to compare incomes for respondents with different education levels.

**Daily XP743**

##### Exercise

##### Exercise

# Distribution of education

Let's begin comparing incomes for different levels of education in the GSS dataset, which has been pre-loaded for you into a DataFrame called gss. The variable educ represents the respondent's years of education.

What fraction of respondents report that they have 12 years of education or fewer?

##### Instructions

**50 XP**

##### Possible Answers

* 

Approximately 22%

* 

Approximately 31%

* 

Approximately 47%

* ****

**Approximately 53% this is the answer**

Select realinc income = gss['realinc'] # Make the CDF cdf\_income = Cdf(income) # Plot it cdf\_income.plot() # Label the axes plt.xlabel('Income (1986 USD)') plt.ylabel('CDF') plt.show()

Correct. If you evaluate the CDF at 12, you get the fraction of respondents with 12 or fewer years of eduction.

n [4]:

cdf= Cdf(gss['educ'])

In [5]:

cdf(12)

Out[5]:

**array(0.53226117)**

**Daily XP793**

##### Exercise

##### Exercise

# Extract education levels

Let's create Boolean Series to identify respondents with different levels of education.

In the U.S, 12 years of education usually means the respondent has completed high school (secondary education). A respondent with 14 years of education has probably completed an associate degree (two years of college); someone with 16 years has probably completed a bachelor's degree (four years of college).

##### Instructions

**100 XP**

* Complete the line that identifies respondents with associate degrees, that is, people with 14 or more years of education but less than 16.
* Complete the line that identifies respondents with 12 or fewer years of education.
* Confirm that the mean of high is the fraction we computed in the previous exercise, about 53%.
* # Select educ
* educ = gss['educ']
* # Bachelor's degree
* bach = (educ >= 16)
* # Associate degree
* assc = \_\_\_\_
* # High school (12 or fewer years of education)
* high = \_\_\_\_
* print(high.mean())

# Select realinc income = gss['realinc'] # Make the CDF cdf\_income = Cdf(income) # Plot it cdf\_income.plot() # Label the axes plt.xlabel('Income (1986 USD)') plt.ylabel('CDF') plt.show()

# Select educ

educ = gss['educ']

# Bachelor's degree

bach = (educ >= 16)

# Associate degree

assc = (bach <= 14)

# High school (12 or fewer years of education)

high = (assc  <= 12)

print(high.mean())

# Select educ

educ = gss['educ']

# Bachelor's degree

bach = (educ >= 16)

# Associate degree

assc = (bach <= 14)

# High school (12 or fewer years of education)

high = (assc <= 12)

print(high.mean())

1.0

Excellent. Remember, you can use logical operators to make Boolean Series and select rows from a DataFrame or Series.

# Select educ

educ = gss['educ']

# Bachelor's degree

bach = (educ >= 16)

# Associate degree

assc = (educ >=14) & (educ <16)

# High school (12 or fewer years of education)

high = (educ  <= 12)

print(high.mean())

# Select educ

educ = gss['educ']

# Bachelor's degree

bach = (educ >= 16)

# Associate degree

assc = (educ >=14) & (educ <16)

# High school (12 or fewer years of education)

high = (educ  <= 12)

print(high.mean())

# Select educ

educ = gss['educ']

# Bachelor's degree

bach = (educ >= 16)

# Associate degree

assc = (educ >=14) & (educ <16)

# High school (12 or fewer years of education)

high = (educ <= 12)

print(high.mean())

0.5308807991547402

<script.py> output:

0.5308807991547402

**Daily XP70**

##### Exercise

##### Exercise

# Plot income CDFs

Let's now see what the distribution of income looks like for people with different education levels. You can do this by plotting the CDFs. Recall how Allen plotted the income CDFs of respondents interviewed before and after 1995:

Cdf(income[pre95]).plot(label='Before 1995')

Cdf(income[~pre95]).plot(label='After 1995')

You can assume that Boolean Series have been defined, as in the previous exercise, to identify respondents with different education levels: high, assc, and bach.

##### Instructions

**100 XP**

* Fill in the missing lines of code to plot the CDFs.
* income = gss['realinc']
* # Plot the CDFs
* \_\_\_\_(label='High school')
* \_\_\_\_(label='Associate')
* \_\_\_\_(label='Bachelor')
* # Label the axes
* plt.xlabel('Income (1986 USD)')
* plt.ylabel('CDF')
* plt.legend()
* plt.show()

income = gss['realinc']

# Plot the CDFs

Cdf(income).plot(label='High school')

Cdf(income).plot(label='Associate')

Cdf(income).plot(label='Bachelor')

# Label the axes

plt.xlabel('Income (1986 USD)')

plt.ylabel('CDF')

plt.legend()

plt.show()

income = gss['realinc'] # Plot the CDFs Cdf(income).plot(label='High school') Cdf(income).plot(label='Associate') Cdf(income).plot(label='Bachelor') # Label the axes plt.xlabel('Income (1986 USD)') plt.ylabel('CDF') plt.legend() plt.show()

Nice job. It might not be surprising that people with more education have higher incomes, but looking at these distributions, we can see where the differences are.

**Daily XP220**

##### Exercise

##### Exercise

# Distribution of income

In many datasets, the distribution of income is approximately lognormal, which means that the logarithms of the incomes fit a normal distribution. We'll see whether that's true for the GSS data. As a first step, you'll compute the mean and standard deviation of the log of incomes using NumPy's np.log10() function.

Then, you'll use the computed mean and standard deviation to make a norm object using the [scipy.stats.norm()](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html) function.

##### Instructions

**100 XP**

* Extract 'realinc' from gss and compute its logarithm using np.log10().
* Compute the mean and standard deviation of the result.
* Make a norm object by passing the computed mean and standard deviation to norm().
* # Extract realinc and compute its log
* income = gss['realinc']
* log\_income = \_\_\_\_
* # Compute mean and standard deviation
* mean = \_\_\_\_
* std = \_\_\_\_
* print(mean, std)
* # Make a norm object
* from scipy.stats import norm
* dist = \_\_\_\_

income = gss['realinc'] # Plot the CDFs Cdf(income[high]).plot(label='High school') Cdf(income[assc]).plot(label='Associate') Cdf(income[bach]).plot(label='Bachelor') # Label the axes plt.xlabel('Income (1986 USD)') plt.ylabel('CDF') plt.legend() plt.show()

# Extract realinc and compute its log

income = gss['realinc']

log\_income = np.log10(income)

# Compute mean and standard deviation

mean = log\_income.mean()

std = log\_income.std()

print(mean, std)

# Make a norm object

from scipy.stats import norm

dist = norm(mean, std)

#print(dist)

# Extract realinc and compute its log

income = gss['realinc']

log\_income = np.log10(income)

# Compute mean and standard deviation

mean = log\_income.mean()

std = log\_income.std()

print(mean, std)

# Make a norm object

from scipy.stats import norm

dist = norm(mean, std)

print(dist)

4.371148677934171 0.4290082383271385

<scipy.stats.\_distn\_infrastructure.rv\_frozen object at 0x7efcd1132520>

Nice work. Now we can plot the model and the observed distribution and see where they differ.

**Daily XP320**

##### Exercise

##### Exercise

# Comparing CDFs

To see whether the distribution of income is well modeled by a lognormal distribution, we'll compare the CDF of the logarithm of the data to a normal distribution with the same mean and standard deviation. These variables from the previous exercise are available for use:

# Extract realinc and compute its log

log\_income = np.log10(gss['realinc'])

# Compute mean and standard deviation

mean, std = log\_income.mean(), log\_income.std()

# Make a norm object

from scipy.stats import norm

dist = norm(mean, std)

dist is a scipy.stats.norm object with the same mean and standard deviation as the data. It provides .cdf(), which evaluates the normal cumulative distribution function.

Be careful with capitalization: Cdf(), with an uppercase C, creates Cdf objects. dist.cdf(), with a lowercase c, evaluates the normal cumulative distribution function.

##### Instructions

**100 XP**

* Evaluate the normal cumulative distribution function using dist.cdf.
* Use the Cdf() function to compute the CDF of log\_income.
* Plot the result.
* # Evaluate the model CDF
* xs = np.linspace(2, 5.5)
* ys = \_\_\_\_
* # Plot the model CDF
* plt.clf()
* plt.plot(xs, ys, color='gray')
* # Create and plot the Cdf of log\_income
* \_\_\_\_
* # Label the axes
* plt.xlabel('log10 of realinc')
* plt.ylabel('CDF')
* plt.show()

# Evaluate the model CDF

xs = np.linspace(2, 5.5)

ys = dist.cdf(xs)

# Plot the model CDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Create and plot the Cdf of log\_income

Cdf(log\_income).plot()

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('CDF')

plt.show()

# Evaluate the model CDF

xs = np.linspace(2, 5.5)

ys = dist.cdf(xs)

# Plot the model CDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Create and plot the Cdf of log\_income

Cdf(log\_income).plot()

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('CDF')

plt.show()

# Extract realinc and compute its log

income = gss['realinc']

log\_income = np.log10(income)

# Compute mean and standard deviation

mean = log\_income.mean()

std = log\_income.std()

print(mean, std)

# Make a norm object

from scipy.stats import norm

dist = norm(mean, std)

print(dist)

# Evaluate the model CDF

xs = np.linspace(2, 5.5)

ys = dist.cdf(xs)

# Plot the model CDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Create and plot the Cdf of log\_income

Cdf(log\_income).plot()

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('CDF')

plt.show()

Good job. The lognormal model is a pretty good fit for the data, but clearly not a perfect match. That's what real data is like; sometimes it doesn't fit the model.

**Daily XP420**

##### Exercise

##### Exercise

# Comparing PDFs

In the previous exercise, we used CDFs to see if the distribution of income is lognormal. We can make the same comparison using a PDF and KDE. That's what you'll do in this exercise!

As before, the norm object dist is available in your workspace:

from scipy.stats import norm

dist = norm(mean, std)

Just as all norm objects have a .cdf() method, they also have a .pdf() method.

To create a KDE plot, you can use Seaborn's [kdeplot()](https://seaborn.pydata.org/generated/seaborn.kdeplot.html) function. Here, Seaborn has been imported for you as sns.

##### Instructions

**100 XP**

* Evaluate the normal PDF using dist, which is a norm object with the same mean and standard deviation as the data.
* Make a KDE plot of the logarithms of the incomes, using log\_income, which is a Series object.

# Evaluate the normal PDF

xs = np.linspace(2, 5.5)

ys = \_\_\_\_

# Plot the model PDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Plot the data KDE

\_\_\_\_

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('PDF')

plt.show()

# Evaluate the model CDF xs = np.linspace(2, 5.5) ys = dist.cdf(xs) # Plot the model CDF plt.clf() plt.plot(xs, ys, color='gray') # Create and plot the Cdf of log\_income Cdf(log\_income).plot() # Label the axes plt.xlabel('log10 of realinc') plt.ylabel('CDF') plt.show()

# Evaluate the normal PDF

xs = np.linspace(2, 5.5)

ys = dist.pdf(xs)

# Plot the model PDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Plot the data KDE

sns.kdeplot(log\_income)

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('PDF')

plt.show()

# Evaluate the model CDF

xs = np.linspace(2, 5.5)

ys = dist.cdf(xs)

# Plot the model CDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Create and plot the Cdf of log\_income

Cdf(log\_income).plot()

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('CDF')

plt.show()

# Evaluate the normal PDF

xs = np.linspace(2, 5.5)

ys = dist.pdf(xs)

# Plot the model PDF

plt.clf()

plt.plot(xs, ys, color='gray')

# Plot the data KDE

sns.kdeplot(log\_income)

# Label the axes

plt.xlabel('log10 of realinc')

plt.ylabel('PDF')

plt.show()

Congratulations on completing Chapter 2! We've seen several ways to vizualize and compare distributions: PMFs, CDFs, and KDE plots. In the next Chapter we'll explore relationships between variables, starting with heights and weights from a large survey of adults in the U.S. See you there!

##### Exercise

##### Exercise

# PMF of age

Do people tend to gain weight as they get older? We can answer this question by visualizing the relationship between weight and age. But before we make a scatter plot, it is a good idea to visualize distributions one variable at a time. Here, you'll visualize age using a bar chart first. Recall that all PMF objects have a .bar() method to make a bar chart.

The BRFSS dataset includes a variable, 'AGE' (note the capitalization!), which represents each respondent's age. To protect respondents' privacy, ages are rounded off into 5-year bins. 'AGE' contains the midpoint of the bins.

##### Instructions

**100 XP**

* Extract the variable 'AGE' from the DataFrame brfss and assign it to age.
* Get the PMF of age and plot it as a bar chart.

# Extract age

age = \_\_\_\_

# Plot the PMF

pmf\_age = \_\_\_\_

pmf\_age.\_\_\_\_

# Label the axes

plt.xlabel('Age in years')

plt.ylabel('PMF')

plt.show()

# Evaluate the normal PDF xs = np.linspace(2, 5.5) ys = dist.pdf(xs) # Plot the model PDF plt.clf() plt.plot(xs, ys, color='gray') # Plot the data KDE sns.kdeplot(log\_income) # Label the axes plt.xlabel('log10 of realinc') plt.ylabel('PDF') plt.show()

# Extract age age = brfss['AGE'] # Plot the PMF pmf\_age = Pmf(age) pmf\_age.bar() # Label the axes plt.xlabel('Age in years') plt.ylabel('PMF') plt.show()

Ok, we're off to a good start. Notice that the last age range is bigger than the others. That's the kind of thing you see when you plot distributions.

**Daily XP670**

##### Exercise

##### Exercise

# Scatter plot

Now let's make a scatterplot of weight versus age. To make the code run faster, I've selected only the first 1000 rows from the brfss DataFrame.

weight and age have already been extracted for you. Your job is to use plt.plot() to make a scatter plot.

##### Instructions

**100 XP**

* Make a scatter plot of weight and age with format string 'o' and alpha=0.1.

# Extract age age = brfss['AGE'] # Plot the PMF pmf\_age = Pmf(age) pmf\_age.bar() # Label the axes plt.xlabel('Age in years') plt.ylabel('PMF') plt.show()

# Select the first 1000 respondents

brfss = brfss[:1000]

# Extract age and weight

age = brfss['AGE']

weight = brfss['WTKG3']

# Make a scatter plot

plt.xlabel('Age in years')

plt.ylabel('Weight in kg')

plt.show()

So far so good. By adjusting alpha we can avoid saturating the plot. Next we'll jitter the data to break up the columns.

 Select the first 1000 respondents

brfss = brfss[:1000]

# Extract age and weight

age = brfss['AGE']

weight = brfss['WTKG3']

# Make a scatter plot

plt.plot(age, weight, 'o', alpha=0.1)

plt.xlabel('Age in years')

plt.ylabel('Weight in kg')

plt.show()

# Extract age and weight

age = brfss['AGE']

weight = brfss['WTKG3']

# Make a scatter plot

plt.plot(age, weight, 'o')

plt.xlabel('Age in years')

plt.ylabel('Weight in kg')

plt.show()

# Select the first 1000 respondents

brfss = brfss[:1000]

# Extract age and weight

age = brfss['AGE']

weight = brfss['WTKG3']

# Make a scatter plot

plt.plot(age, weight, 'o', alpha=0.1)

plt.xlabel('Age in years')

plt.ylabel('Weight in kg')

plt.show()

##### Exercise

# Jittering

In the previous exercise, the ages fall in columns because they've been rounded into 5-year bins. If we jitter them, the scatter plot will show the relationship more clearly. Recall how Allen jittered height and weight in the video:

height\_jitter = height + np.random.normal(0, 2, size=len(brfss))

weight\_jitter = weight + np.random.normal(0, 2, size=len(brfss))

##### Instructions

**100 XP**

* Add random noise to age with mean 0 and standard deviation 2.5.
* Make a scatter plot between weight and age with marker size 5 and alpha=0.2. Be sure to also specify 'o'.

# Select the first 1000 respondents

brfss = brfss[:1000]

# Add jittering to age

age = brfss['AGE'] + \_\_\_\_

# Extract weight

weight = brfss['WTKG3']

# Make a scatter plot

plt.xlabel('Age in years')

plt.ylabel('Weight in kg')

plt.show()

# Select the first 1000 respondents

brfss = brfss[:1000]

# Add jittering to age

age = brfss['AGE'] + np.random.normal(0, 2.5, size=len(brfss))

# Extract weight

weight = brfss['WTKG3']

# Make a scatter plot

plt.plot(age, weight, 'o', markersize= 5, alpha=0.2)

plt.xlabel('Age in years')

plt.ylabel('Weight in kg')

plt.show()

# Select the first 1000 respondents brfss = brfss[:1000] # Add jittering to age age = brfss['AGE'] + np.random.normal(0, 2.5, size=len(brfss)) # Extract weight weight = brfss['WTKG3'] # Make a scatter plot plt.plot(age, weight, 'o', markersize= 5, alpha=0.2) plt.xlabel('Age in years') plt.ylabel('Weight in kg') plt.show()

Excellent. By smoothing out the ages and avoiding saturation, we get the best view of the data. But in this case the nature of the relationship is still hard to see. In the next lesson, we'll see some other ways to visualize it.

**Daily XP870**

**Visualizing relationships**

**50 XP**

**1. Visualizing relationships**

In the previous lesson we used scatter plots to visualize relationships between variables, and in the exercise, you explored the relationship between age and weight. In this lesson, we'll see other ways to visualize these relationships, including boxplots and violin plots.

**2. Weight and age**

In the previous exercises, you made a scatter plot of weight versus age. Your code probably looked like this. And the results looked like this. It looks like older people might be heavier, but it is hard to see clearly.

**3. More data**

For the exercises, you worked with a small subset of the data. Now let's see what it looks like with more data. Here's the code. And here's the plot. I made a few changes in the code: \* First, I reduced the marker size, because we have more data now, \* Second, I jittered the weights, so the horizontal rows are not visible. \* I jitter the ages, too, but less than in the exercise, so the data points are spread out, but there's still space between the columns. That makes it possible to see the shape of the distribution in each age group, and the differences between groups. If we take this idea one step farther, we can use KDE to estimate the density function in each column and plot it.

**4. Violin plot**

And there's a name for that; it's called a violin plot. Seaborn provides a function that makes violin plots, but before we can use it, we have to get rid of any rows with missing data. Here's how. dropna() creates a new DataFrame that contains the rows from brfss where AGE and WTKG3 are not NaN. Now we can call violinplot(). The x and y parameters mean we want AGE on the x-axis and WTKG3 on the y-axis. data is the DataFrame we just created, which contains the variables we're going to plot. The parameter inner=None simplifies the plot a little. Here's what it looks like. Each column is a graphical representation of the distribution of weight in one age group. The width of these shapes is proportional to the estimated density, so it's like two vertical PDFs plotted back to back, and filled in with nice colors. There's one other way to look at data like this, called a box plot.

**5. Box plot**

The code to generate a box plot is very similar. I put in the parameter whis=10 to turn off a feature we don't need. If you are curious about it, you can read the documentation or check out DataCamp's Seaborn courses. Here's what it looks like. Each box represents the interquartile range, or IQR, from the 25th to the 75th percentile. The line in the middle of each box is the median. The spines sticking out of the top and bottom show the minimum and maximum values. In my opinion, this plot gives us the best view of the relationship between weight and age. Looking at the medians, it seems like people in their 40s are the heaviest; younger and older people are lighter. Looking at the sizes of the boxes, it seems like people in their 40s have the most variability in weight, too. These plots also show how skewed the distribution of weight is; that is, the heaviest people are much farther from the median than the lightest people.

**6. Log scale**

For data that skews toward higher values, it is sometimes useful to look at it on a logarithmic scale. We can do that with the pyplot function yscale(). Here's what it looks like. To show the relationship between age and weight most clearly, this is probably the figure I would use.

**7. Let's practice!**

Now let's get some practice with violin and box plots.

**Daily XP870**

**Visualizing relationships**

**50 XP**

**1. Visualizing relationships**

In the previous lesson we used scatter plots to visualize relationships between variables, and in the exercise, you explored the relationship between age and weight. In this lesson, we'll see other ways to visualize these relationships, including boxplots and violin plots.

**2. Weight and age**

In the previous exercises, you made a scatter plot of weight versus age. Your code probably looked like this. And the results looked like this. It looks like older people might be heavier, but it is hard to see clearly.

**3. More data**

For the exercises, you worked with a small subset of the data. Now let's see what it looks like with more data. Here's the code. And here's the plot. I made a few changes in the code: \* First, I reduced the marker size, because we have more data now, \* Second, I jittered the weights, so the horizontal rows are not visible. \* I jitter the ages, too, but less than in the exercise, so the data points are spread out, but there's still space between the columns. That makes it possible to see the shape of the distribution in each age group, and the differences between groups. If we take this idea one step farther, we can use KDE to estimate the density function in each column and plot it.

**4. Violin plot**

And there's a name for that; it's called a violin plot. Seaborn provides a function that makes violin plots, but before we can use it, we have to get rid of any rows with missing data. Here's how. dropna() creates a new DataFrame that contains the rows from brfss where AGE and WTKG3 are not NaN. Now we can call violinplot(). The x and y parameters mean we want AGE on the x-axis and WTKG3 on the y-axis. data is the DataFrame we just created, which contains the variables we're going to plot. The parameter inner=None simplifies the plot a little. Here's what it looks like. Each column is a graphical representation of the distribution of weight in one age group. The width of these shapes is proportional to the estimated density, so it's like two vertical PDFs plotted back to back, and filled in with nice colors. There's one other way to look at data like this, called a box plot.

**5. Box plot**

The code to generate a box plot is very similar. I put in the parameter whis=10 to turn off a feature we don't need. If you are curious about it, you can read the documentation or check out DataCamp's Seaborn courses. Here's what it looks like. Each box represents the interquartile range, or IQR, from the 25th to the 75th percentile. The line in the middle of each box is the median. The spines sticking out of the top and bottom show the minimum and maximum values. In my opinion, this plot gives us the best view of the relationship between weight and age. Looking at the medians, it seems like people in their 40s are the heaviest; younger and older people are lighter. Looking at the sizes of the boxes, it seems like people in their 40s have the most variability in weight, too. These plots also show how skewed the distribution of weight is; that is, the heaviest people are much farther from the median than the lightest people.

**6. Log scale**

For data that skews toward higher values, it is sometimes useful to look at it on a logarithmic scale. We can do that with the pyplot function yscale(). Here's what it looks like. To show the relationship between age and weight most clearly, this is probably the figure I would use.

**7. Let's practice!**

Now let's get some practice with violin and box plots.

# Drop rows with missing data

data = brfss.dropna(subset=['\_HTMG10', 'WTKG3'])

# Make a box plot

sns.boxplot(\_\_\_\_)

# Plot the y-axis on a log scale

# Remove unneeded lines and label axes

sns.despine(left=True, bottom=True)

plt.xlabel('Height in cm')

plt.ylabel('Weight in kg')

plt.show()

# Select the first 1000 respondents brfss = brfss[:1000] # Add jittering to age age = brfss['AGE'] + np.random.normal(0, 2.5, size=len(brfss)) # Extract weight weight = brfss['WTKG3'] # Make a scatter plot plt.plot(age, weight, 'o', markersize= 5, alpha=0.2) plt.xlabel('Age in years') plt.ylabel('Weight in kg') plt.show()

# Drop rows with missing data

data = brfss.dropna(subset=['\_HTMG10', 'WTKG3'])

# Make a box plot

sns.boxplot(x='\_HTMG10', y='WTKG3', data=data, whis=10)

# Plot the y-axis on a log scale

plt.yscale('log')

# Remove unneeded lines and label axes

sns.despine(left=True, bottom=True)

plt.xlabel('Height in cm')

plt.ylabel('Weight in kg')

plt.show()

# Drop rows with missing data data = brfss.dropna(subset=['\_HTMG10', 'WTKG3']) # Make a box plot sns.boxplot(x='\_HTMG10', y='WTKG3', data=data, whis=10) # Plot the y-axis on a log scale plt.yscale('log') # Remove unneeded lines and label axes sns.despine(left=True, bottom=True) plt.xlabel('Height in cm') plt.ylabel('Weight in kg') plt.show()

Very good. These box plots provide a good view of the relationship between the variables. They also show the spread of the values in each column.

**Daily XP1020**

##### Exercise

##### Exercise

# Distribution of income

In the next two exercises we'll look at relationships between income and other variables. In the BRFSS, income is represented as a categorical variable; that is, respondents are assigned to one of 8 income categories. The variable name is 'INCOME2'. Before we connect income with anything else, let's look at the distribution by computing the PMF. Recall that all Pmf objects have a .bar() method.

##### Instructions

**100 XP**

* Extract 'INCOME2' from the brfss DataFrame and assign it to income.
* Plot the PMF of income as a bar chart.
* # Extract income
* income = \_\_\_\_
* # Plot the PMF
* # Label the axes
* plt.xlabel('Income level')
* plt.ylabel('PMF')
* plt.show()

# Drop rows with missing data data = brfss.dropna(subset=['\_HTMG10', 'WTKG3']) # Make a box plot sns.boxplot(x='\_HTMG10', y='WTKG3', data=data, whis=10) # Plot the y-axis on a log scale plt.yscale('log') # Remove unneeded lines and label axes sns.despine(left=True, bottom=True) plt.xlabel('Height in cm') plt.ylabel('Weight in kg') plt.show()

# Extract income

income = brfss['INCOME2']

# Plot the PMF

Pmf(income).bar()

# Label the axes

plt.xlabel('Income level')

plt.ylabel('PMF')

plt.show()

# Extract income income = brfss['INCOME2'] # Plot the PMF Pmf(income).bar() # Label the axes plt.xlabel('Income level') plt.ylabel('PMF') plt.show()

Good work. Almost half of the respondents are in the top income category, so this dataset doesn't distinguish between the highest incomes and the median. But maybe it can tell us something about people with incomes below the median.

Good work. Almost half of the respondents are in the top income category, so this dataset doesn't distinguish between the highest incomes and the median. But maybe it can tell us something about people with incomes below the median.

# Drop rows with missing data

data = brfss.dropna(subset=['INCOME2', 'HTM4'])

# Make a violin plot

# Remove unneeded lines and label axes

sns.despine(left=True, bottom=True)

plt.xlabel('Income level')

plt.ylabel('Height in cm')

plt.show()

# Extract income income = brfss['INCOME2'] # Plot the PMF Pmf(income).bar() # Label the axes plt.xlabel('Income level') plt.ylabel('PMF') plt.show()

Interesting. It looks like there is a weak positive relationsip between income and height, at least for incomes below the median. In the next lesson we'll see some ways to quantify the strength of this relationship.

**Daily XP200**

**Correlation**

**50 XP**

**1. Correlation**

In the previous lesson, we visualized relationships between pairs of variables. In this lesson we'll learn about the coefficient of correlation, which quantifies the strength of these relationships.

**2. Correlation coefficient**

When people say "correlation" casually, they might mean any relationship between two variables. In statistics, it usually means Pearson's correlation coefficient, which is a number between -1 and 1 that quantifies the strength of a linear relationship between variables. To demonstrate, I'll select three columns from the BRFSS dataset, like this. The result is a DataFrame with just those columns. Now we can use the corr() method, like this.

**3. Correlation matrix**

The result is a "correlation matrix". Reading across the first row, the correlation of HTM4 with itself is 1. That's expected; the correlation of anything with itself is 1. The next entry is more interesting; the correlation of height and weight is about 0 point 47. It's positive, which means taller people are heavier, and it is moderate in strength, which means it has some predictive value. If you know someone's height, you can make a better guess about their weight, and vice versa. The correlation between height and age is about -0 point 09. It's negative, which means that older people tend to be shorter, but it's weak, which means that knowing someone's age would not help much if you were trying to guess their height. The correlation between age and weight is even smaller. It is tempting to conclude that there is no relationship between age and weight, but we have already seen that there is. So why is the correlation so low?

**4. Weight and age**

Remember that the relationship between weight and age looks like this. People in their 40s are the heaviest; younger and older people are lighter. So this relationship is nonlinear.

**5. Nonlinear relationships**

But correlation only works for linear relationships. If the relationship is nonlinear, correlation generally underestimates how strong it is. To demonstrate, I'll generate some fake data: xs contains equally-spaced points between -1 and 1. ys is xs squared plus some random noise. Here's the scatter plot of xs and ys. It's clear that this is a strong relationship; if you are given `x`, you can make a much better guess about y. But here's the correlation matrix; the computed correlation is close to 0. In general, if correlation is high -- that is, close to 1 or -1, you can conclude that there is a strong linear relationship. But if correlation is close to 0, that doesn't mean there is no relationship; there might be a strong, non-linear relationship. This is one of the reasons I think correlation is not such a great statistic.

**6. You keep using that word**

There's another reason to be careful with correlation; it doesn't mean what people take it to mean. Specifically, correlation says nothing about slope. If we say that two variables are correlated, that means we can use one to predict the other. But that might not be what we care about.

**7. Strength of relationship**

For example, suppose we are concerned about the health effects of weight gain, so we plot weight versus age, from 20 to 50 years old. Here are two fake datasets I generated. The one on the left has higher correlation, about 0 point 76 compared to 0 point 47. But on the left, the average weight gain over 30 years is less than 1 kg; on the right, it is almost 10 kilograms! In this scenario, the relationship on the right is probably more important, even though the correlation is lower. The statistic we really care about is the slope of the line.

**8. Let's practice!**

In the next lesson, you'll learn how to estimate that slope. But first, let's practice with correlation.

**Daily XP250**

##### Exercise

##### Exercise

# Computing correlations

The purpose of the BRFSS is to explore health risk factors, so it includes questions about diet. The variable '\_VEGESU1' represents the number of servings of vegetables respondents reported eating per day.

Let's see how this variable relates to age and income.

##### Instructions

**100 XP**

* From the brfss DataFrame, select the columns 'AGE', 'INCOME2', and '\_VEGESU1'.
* Compute the correlation matrix for these variables.
* # Select columns
* columns = \_\_\_\_
* subset = \_\_\_\_
* # Compute the correlation matrix
* print(subset.\_\_\_\_())

# Drop rows with missing data data = brfss.dropna(subset=['INCOME2', 'HTM4']) # Make a violin plot sns.violinplot(x='INCOME2', y='HTM4', data=data, inner=None) # Remove unneeded lines and label axes sns.despine(left=True, bottom=True) plt.xlabel('Income level') plt.ylabel('Height in cm') plt.show()

# Select columns

columns = ['AGE', 'INCOME2', '\_VEGESU1']

subset = brfss[columns]

# Compute the correlation matrix

print(subset.corr())

# Select columns

columns = ['AGE', 'INCOME2', '\_VEGESU1']

subset = brfss[columns]

# Compute the correlation matrix

print(subset.corr())

AGE INCOME2 \_VEGESU1

AGE 1.000 -0.015 -0.01

INCOME2 -0.015 1.000 0.12

\_VEGESU1 -0.010 0.120 1.00

So far, so good. In the next exercise you'll think about how to interpret these results.

# Interpreting correlations

In the previous exercise, the correlation between income and vegetable consumption is about 0.12. The correlation between age and vegetable consumption is about -0.01.

Which of the following are correct interpretations of these results:

* A: People with higher incomes eat more vegetables.
* B: The relationship between income and vegetable consumption is linear.
* C: Older people eat more vegetables.
* D: There could be a strong nonlinear relationship between age and vegetable consumption.

##### Answer the question

**50XP**

#### Possible Answers

* 

A and C only.

press1

* 

B and D only.

press2

* 

B and C only.

press3

* 

**A and D only This is the answer.**

Correct! The correlation between income and vegetable consumption is small, but it suggests that there is a relationship. But a correlation close to 0 does mean there is no relationship.

**Daily XP400**

**Simple regression**

**50 XP**

**1. Simple regression**

In the previous lesson we saw that correlation does not always measure what we really want to know. In this lesson, we look at an alternative - simple linear regression.

**2. Strength of relationship**

Let's look again at an example from the previous lesson, a hypothetical relationship between weight and age. I generated two fake datasets to make a point: The one on the left has higher correlation, about 0 point 76 compared to 0 point 48. But in the one on the left, the average weight gain over 30 years is less than 1 kg; on the right, it is almost 10 kilograms! In this context, the statistic we probably care about is the slope of the line, not the correlation coefficient.

**3. Strength of effect**

To estimate the slope of the line, we can use linregress() from the SciPy stats module. The result is a LinRegressResult object that contains five values: slope is the slope of the line of best fit for the data; intercept is the intercept. For Hypothetical #1, the estimated slope is about 0.019 kilograms per year or about 0.6 kilograms over the 30-year range.

**4. Strength of effect**

Here are the results for Hypothetical #2. The estimated slope is about 10 times higher: about 0 point 18 kilograms per year or 6 kilograms per 30 years, What's called rvalue here is correlation, which confirms what we saw before; the first example has higher correlation, about 0 point 76 compared to 0 point 48. But the strength of the effect, as measured by the slope of the line, is about 10 times higher in the second example.

**5. Regression lines**

We can use the results from linregress() to compute the line of best fit: first we get the min and max of the observed xs; then we multiply by the slope and add the intercept. And plot the line. Here's what that looks like for the first example. And the same thing for the second example. The visualization here might be misleading unless you look closely at the vertical scales; the slope on the right is almost 10 times higher.

**6. Height and weight**

Now let's look at an example with real data. Here's the scatter plot of height and weight again, from Lesson 1.

**7. Regression line**

Now we can compute the regression line. linregress() can't handle NaNs, so we have to use dropna() to remove rows that are missing the data we need. Now we can compute the linear regression. And here are the results. The slope is about 0 point 9 kilograms per centimeter, which means that we expect a person one centimeter taller to be almost a kilogram heavier. That's quite a lot.

**8. Line of best fit**

As before, we can compute the line of best fit and plot it. And here's what that looks like. The slope of this line seems consistent with the scatter plot.

**9. Linear relationships**

However, linear regression has the same problem as correlation; it only measures the strength of a linear relationship. Here's the scatter plot of weight versus age, which you saw in a previous exercise. People in their 40s are the heaviest; younger and older people are lighter. So the relationship is nonlinear.

**10. Nonlinear relationships**

If we don't look at the scatter plot and blindly compute the regression line, here's what we get. The estimated slope is only 0 point 02 kilograms per year, or 0 point 6 kilograms in 30 years.

**11. Not a good fit**

And here's what the line of best fit looks like. A straight line does not capture the relationship between these variables well.

**12. Let's practice!**

In the next lesson, we'll learn how to use multiple regression to estimate non-linear relationships. But first, let's practice simple regression.

**Daily XP450**

##### Exercise

##### Exercise

# Income and vegetables

As we saw in a previous exercise, the variable '\_VEGESU1' represents the number of vegetable servings respondents reported eating per day.

Let's estimate the slope of the relationship between vegetable consumption and income.

##### Instructions

**100 XP**

* Extract the columns 'INCOME2' and '\_VEGESU1' from subset into xs and ys respectively.
* Compute the simple linear regression of these variables.
* from scipy.stats import linregress
* # Extract the variables
* subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])
* xs = \_\_\_\_
* ys = \_\_\_\_
* # Compute the linear regression
* res = \_\_\_\_
* print(res)

# Select columns columns = ['AGE', 'INCOME2', '\_VEGESU1'] subset = brfss[columns] # Compute the correlation matrix print(subset.corr())

from scipy.stats import linregress

# Extract the variables

subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])

xs = subset['INCOME2']

ys = subset['\_VEGESU1']

# Compute the linear regression

res = linregress(xs, ys)

print(res)

from scipy.stats import linregress

# Extract the variables

subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])

xs = subset['INCOME2']

ys = subset['\_VEGESU1']

# Compute the linear regression

res = linregress(xs, ys)

print(res)

LinregressResult(slope=0.06988048092105019, intercept=1.5287786243363106, rvalue=0.11967005884864099, pvalue=1.378503916248713e-238, stderr=0.002110976356332333, intercept\_stderr=0.013196467544093607

Good job. The estimated slope tells you the increase in vegetable servings from one income group to the next.

**Daily XP550**

##### Exercise

##### Exercise

# Fit a line

Continuing from the previous exercise:

* Assume that xs and ys contain income codes and daily vegetable consumption, respectively, and
* res contains the results of a simple linear regression of ys onto xs.

Now, you're going to compute the line of best fit. NumPy has been imported for you as np.

##### Instructions

**100 XP**

* Set fx to the minimum and maximum of xs, stored in a NumPy array.
* Set fy to the points on the fitted line that correspond to the fx.
* # Plot the scatter plot
* plt.clf()
* x\_jitter = xs + np.random.normal(0, 0.15, len(xs))
* plt.plot(x\_jitter, ys, 'o', alpha=0.2)
* # Plot the line of best fit
* fx = \_\_\_\_
* fy = \_\_\_\_
* plt.plot(fx, fy, '-', alpha=0.7)
* plt.xlabel('Income code')
* plt.ylabel('Vegetable servings per day')
* plt.ylim([0, 6])
* plt.show()

# Extract the variables subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1']) xs = subset['INCOME2'] ys = subset['\_VEGESU1'] # Compute the linear regression res = linregress(xs, ys) print(res)

# Plot the scatter plot

plt.clf()

x\_jitter = xs + np.random.normal(0, 0.15, len(xs))

plt.plot(x\_jitter, ys, 'o', alpha=0.2)

# Plot the line of best fit

fx = np.array([xs.min(), xs.max()])

fy = res.intercept + res.slope \* fx

plt.plot(fx, fy, '-', alpha=0.7)

plt.xlabel('Income code')

plt.ylabel('Vegetable servings per day')

plt.ylim([0, 6])

plt.show()

# Compute the linear regression

res = linregress(xs, ys)

print(res)

# Plot the scatter plot

plt.clf()

x\_jitter = xs + np.random.normal(0, 0.15, len(xs))

plt.plot(x\_jitter, ys, 'o', alpha=0.2)

# Plot the line of best fit

fx = np.array([xs.min(), xs.max()])

fy = res.intercept + res.slope \* fx

plt.plot(fx, fy, '-', alpha=0.7)

plt.xlabel('Income code')

plt.ylabel('Vegetable servings per day')

plt.ylim([0, 6])

plt.show()

Congratulations on completing Chapter 3! We've seen several ways to visualize relationships between variables and quantify their strength. In the next chapter we use regression to explore relationships among more than two variables.

**Daily XP650**

**Limits of simple regression**

**50 XP**

**1. Limits of simple regression**

In this chapter we'll get farther into regression, including multiple regression and one of my all-time favorite tools, logistic regression. But first let's understand the limits of simple regression.

**2. Income and vegetables**

In a previous exercise, you made a scatter plot of vegetable consumption as a function of income, and plotted a line of best fit. Here's what it looks like (note that this version includes more data than you had for the exercise). The slope of the line is 0 point 07, which means that the difference between the lowest and highest income brackets is about 0 point 49 servings per day. So that's not a very big difference.

**3. Vegetables and income**

But it was an arbitrary choice to plot vegetables as a function of income. We could have plotted it the other way around, like this. The slope of this line is 0 point 23, which means that the difference between 0 and 8 servings per day is about 2 income codes, roughly from code 5 to code 7. If we check the codebook, income code 5 is about $30,000 per year; income code 7 is about $65,000. So if we use vegetable consumption to predict income, we see a big difference. But when we used income to predict vegetable consumption, we saw a small difference. This example shows that regression is not symmetric; the regression of A onto B is not the same as the regression of B onto A.

**4. Regression is not symmetric**

We can see that more clearly by putting the two figures side by side and plotting both regression lines on both figures. They are different because they are based on different assumptions. On the left, we treat income as a known quantity and vegetable consumption as random. On the right, vegetable consumption is known and income is random. When you run a regression model, you make decisions about how to treat the data, and those decisions affect the result you get.

**5. Regression is not causation**

This example is meant to demonstrate another point, which is that regression doesn't tell you much about causation. If you think people with lower income can't afford vegetables, you might look at the figure on the left and conclude that it doesn't make much difference. If you think better diet increases income, the figure on the right might make you think it does. But in general, regression can't tell you what causes what. In this example, A might cause B, or B might cause A, or there might be other factors that cause both A and B. Regression alone can't tell you which way it goes.

**6. Multiple regression**

However, we have tools for teasing apart relationships among multiple variables; one of the most important is multiple regression. SciPy doesn't do multiple regression, so we have to switch to a new library, StatsModels. Here's the import statement. And here's how we use it. `ols` stands for "ordinary least squares", another name for regression. The first argument is a formula string that specifies that we want to regress income as a function of vegetable consumption. The second argument is the BRFSS DataFrame. The names in the formula correspond to columns in the DataFrame. The result from ols() represents the model; we have to run dot fit() to get the results. The results object contains a lot of information, but the first thing we'll look at is params, which contains the estimated slope and intercept. And we get the same results we got from SciPy, so that's good!

**7. Let's practice!**

In the next lesson we'll move on to multiple regression. But first, let's practice simple regression with statsmodels.

# Regression and causation

In the BRFSS dataset, there is a strong relationship between vegetable consumption and income. The income of people who eat 8 servings of vegetables per day is double the income of people who eat none, on average.

Which of the following conclusions can we draw from this data?

A. Eating a good diet leads to better health and higher income.

B. People with higher income can afford a better diet.

C. People with high income are more likely to be vegetarians.

##### Answer the question

**50XP**

#### Possible Answers

* 

A only.

press1

* 

B only.

press2

* 

B and C.

press3

* 

**None of them. This is the correct answer**

That's right. This data is consistent with all of these conclusions, but it does not provide conclusive evidence for any of them.

That's right. This data is consistent with all of these conclusions, but it does not provide conclusive evidence for any of them.

# Plot the scatter plot plt.clf() x\_jitter = xs + np.random.normal(0, 0.15, len(xs)) plt.plot(x\_jitter, ys, 'o', alpha=0.2) # Plot the line of best fit fx = np.array([xs.min(), xs.max()]) fy = res.intercept + res.slope \* fx plt.plot(fx, fy, '-', alpha=0.7) plt.xlabel('Income code') plt.ylabel('Vegetable servings per day') plt.ylim([0, 6]) plt.show()

from scipy.stats import linregress

import statsmodels.formula.api as smf

# Run regression with linregress

subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])

xs = subset['INCOME2']

ys = subset['\_VEGESU1']

res = \_\_\_\_

print(res)

# Run regression with StatsModels

results = smf.ols('\_\_\_\_', \_\_\_\_ = \_\_\_\_).fit()

print(results.params)

from scipy.stats import linregress

import statsmodels.formula.api as smf

# Run regression with linregress

subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])

xs = subset['INCOME2']

ys = subset['\_VEGESU1']

res = linregress(xs, ys)

print(res)

# Run regression with StatsModels

results = smf.ols('\_VEGESU1 ~ INCOME2', data= brfss).fit()

print(results.params)

from scipy.stats import linregress

import statsmodels.formula.api as smf

# Run regression with linregress

subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])

xs = subset['INCOME2']

ys = subset['\_VEGESU1']

res = linregress(xs, ys)

print(res)

# Run regression with StatsModels

results = smf.ols('\_VEGESU1 ~ INCOME2', data= brfss).fit()

print(results.params)

LinregressResult(slope=0.06988048092105019, intercept=1.5287786243363106, rvalue=0.11967005884864099, pvalue=1.378503916248713e-238, stderr=0.002110976356332333, intercept\_stderr=0.013196467544093607)

Intercept 1.529

INCOME2 0.070

dtype: float64

Nice job. When you start working with a new library, checks like this help ensure that you are doing it right.

**Daily XP850**

**Multiple regression**

**50 XP**

**1. Multiple regression**

Now that we are using StatsModels, getting from simple to multiple regression is easy. As an example, we'll use data from the General Social Survey, which we saw in Chapter 2, and we'll explore variables that are related to income.

**2. Income and education**

First, we load the GSS data. Then we run a regression of real income as a function of years of education. The first argument of ols() is a formula that specifies the variables in the regression. On the left, realinc is the variable we are trying to predict; on the right, educ is the variable we are using to inform the predictions. And here are the results. The estimated slope is 3586, which means that each additional year of education is associated with an increase of almost $3600 of income. But income also depends on age, so it would be good to include that in the model, too.

**3. Adding age**

Here's how. On the right side of the formula, you can list as many variables as you like, in this case, we have educ and age. The plus sign indicates that we expect the contributions of the two variables to be additive, which is a common assumption for models like this. Here are the results. The estimated slope for education is 3655, a little more than in the previous model. The estimated slope for age is only about $80 per year, which is surprisingly small.

**4. Income and age**

To see what's going on, let's look more closely at the relationship between income and age. I'll use groupby(), which is a Pandas feature we haven't seen before, to divide the DataFrame into age groups. The result is a GroupBy object that contains one group for each value of age. The GroupBy object behaves like a DataFrame in many ways. You can use brackets to select a column, like realinc in this example, and then invoke a method like mean(). The result is a Pandas series that contains the mean income for each age group, which we can plot like this.

**5. Mean income over age**

Here's the result. Average income increases from age 20 to age 50, then starts to fall. And that explains why the estimated slope is so small, because the relationship is non-linear. Remember that correlation and simple regression can't measure non-linear relationships. But multiple regression can!

**6. Adding a quadratic term**

To describe a non-linear relationship, one option is to add a new variable that is a non-linear combination of other variables. As an example, I'll create a new variable called age2 that equals age squared. Now we can run a regression with both age and age2 on the right side. And here are the results. The slope associated with age is substantial, about $1700 per year. The slope associated with age2 is about -17, which is harder to interpret.

**7. Whew!**

In the next lesson, we'll see methods to visualize regression results. But first, let's practice multiple regression.

**Daily XP50**

##### Exercise

##### Exercise

# Plot income and education

To get a closer look at the relationship between income and education, let's use the variable 'educ' to group the data, then plot mean income in each group.

Here, the GSS dataset has been pre-loaded into a DataFrame called gss.

##### Instructions

**100 XP**

* Group gss by 'educ'. Store the result in grouped.
* From grouped, extract 'realinc' and compute the mean.
* Plot mean\_income\_by\_educ as a scatter plot. Specify 'o' and alpha=0.5.

# Group by educ

grouped = \_\_\_\_

# Compute mean income in each group

mean\_income\_by\_educ = \_\_\_\_

# Plot mean income as a scatter plot

plt.plot(\_\_\_\_)

# Label the axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.show()

from scipy.stats import linregress import statsmodels.formula.api as smf # Run regression with linregress subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1']) xs = subset['INCOME2'] ys = subset['\_VEGESU1'] res = linregress(xs, ys) print(res) # Run regression with StatsModels results = smf.ols('\_VEGESU1 ~ INCOME2', data= brfss).fit() print(results.params)

# Group by educ

grouped = gss.groupby('educ')

# Compute mean income in each group

mean\_income\_by\_educ = grouped['realinc'].mean()

# Plot mean income as a scatter plot

plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5)

# Label the axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.show()

* ython Shell
* Slides
* Notes

from scipy.stats import linregress

import statsmodels.formula.api as smf

# Run regression with linregress

subset = brfss.dropna(subset=['INCOME2', '\_VEGESU1'])

xs = subset['INCOME2']

ys = subset['\_VEGESU1']

res = linregress(xs, ys)

print(res)

# Run regression with StatsModels

results = smf.ols('\_VEGESU1 ~ INCOME2', data= brfss).fit()

print(results.params)

# Group by educ

grouped = gss.groupby('educ')

# Compute mean income in each group

mean\_income\_by\_educ = grouped['realinc'].mean()

# Plot mean income as a scatter plot

plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5)

# Label the axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.show()

Well done. It looks like the relationship between income and education is non-linear.

**Daily XP150**

##### Exercise

##### Exercise

# Non-linear model of education

The graph in the previous exercise suggests that the relationship between income and education is non-linear. So let's try fitting a non-linear model.

##### Instructions

**100 XP**

* Add a column named 'educ2' to the gss DataFrame; it should contain the values from 'educ' squared.
* Run a regression model that uses 'educ', 'educ2', 'age', and 'age2' to predict 'realinc'.

# Group by educ grouped = gss.groupby('educ') # Compute mean income in each group mean\_income\_by\_educ = grouped['realinc'].mean() # Plot mean income as a scatter plot plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5) # Label the axes plt.xlabel('Education (years)') plt.ylabel('Income (1986 $)') plt.show()

mport statsmodels.formula.api as smf

# Add a new column with educ squared

gss['educ2'] = \_\_\_\_

# Run a regression model with educ, educ2, age, and age2

results = \_\_\_\_

# Print the estimated parameters

print(results.params)

import statsmodels.formula.api as smf

# Add a new column with educ squared

gss['educ2'] = gss['educ'] \*\*2

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Print the estimated parameters

print(results.params)

# Group by educ

grouped = gss.groupby('educ')

# Compute mean income in each group

mean\_income\_by\_educ = grouped['realinc'].mean()

# Plot mean income as a scatter plot

plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5)

# Label the axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.show()

import statsmodels.formula.api as smf

# Add a new column with educ squared

gss['educ2'] = gss['educ'] \*\*2

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Print the estimated parameters

print(results.params)

Intercept -23241.884

educ -528.309

educ2 159.967

age 1696.717

age2 -17.197

dtype: float64

Excellent. The slope associated with educ2 is positive, so the model curves upward.

**Daily XP250**

**Visualizing regression results**

**50 XP**

**1. Visualizing regression results**

In the previous lesson we ran a multiple regression model to characterize the relationship between income and age. Because the model is non-linear, the parameters are hard to interpret. In this lesson we'll see a way to interpret them visually, and to validate them against data.

**2. Modeling income and age**

Here's the model from the previous exercise. First, we created new variables for educ squared and age squared. Then we ran the regression model with educ, educ2, age, and age2. And here are the results. The parameters are hard to interpret. Fortunately, we don't have to -- sometimes the best way to understand a model is by looking at its predictions rather than its parameters.

**3. Generating predictions**

The regression results object provides a method called predict() that uses the model to generate predictions. It takes a DataFrame as a parameter and returns a Series with a prediction for each row in the DataFrame. To use it, I'll create a new DataFrame with age running from 18 to 85, and age2 set to age squared. Next, I'll pick a level for educ, like 12 years, which is the most common value. When you assign a single value to a column in a DataFrame, Pandas makes a copy for each respondent. Then we can use results to predict the average income for each age group, holding education constant.

**4. Plotting predictions**

The result from predict() is a Series with one prediction for each row. So we can plot it like this, with age on the x-axis and the predicted income for each age group on the y-axis. We can plot the data for comparison, like this; recall that we computed mean\_income\_by\_age in the previous lesson. And we should label the axes, as always.

**5. Comparing with data**

Here are the results. The blue dots show the average income in each age group. The orange line shows the predictions generated by the model, holding education constant. This plot shows the shape of the model, a downward-facing parabola.

**6. Levels of education**

We can do the same thing with other levels of education, like 14 years, which is the nominal time to earn an Associate's degree, and 16 years, which is the nominal time to earn a Bachelor's degree.

**7. Interpreting the results**

And here are the results. The lines show mean income, as predicted by the model, as a function of age, for three levels of education. This visualization helps validate the model since we can compare the predictions with the data. And it helps us interpret the model since we can see the separate contributions of age and education.

**8. Let's practice!**

In the exercises, you'll have a chance to run a multiple regression, generate predictions, and visualize the results. Have fun!

**Daily XP300**

##### Exercise

##### Exercise

# Making predictions

At this point, we have a model that predicts income using age, education, and sex.

Let's see what it predicts for different levels of education, holding age constant.

##### Instructions

**100 XP**

* Using np.linspace(), add a variable named 'educ' to df with a range of values from 0 to 20.
* Add a variable named 'age' with the constant value 30.
* Use df to generate predicted income as a function of education.
* # Run a regression model with educ, educ2, age, and age2
* results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()
* # Make the DataFrame
* df = pd.DataFrame()
* df['educ'] = \_\_\_\_
* df['age'] = \_\_\_\_
* df['educ2'] = df['educ']\*\*2
* df['age2'] = df['age']\*\*2
* # Generate and plot the predictions
* pred = \_\_\_\_
* print(pred.head())

import statsmodels.formula.api as smf # Add a new column with educ squared gss['educ2'] = gss['educ'] \*\*2 # Run a regression model with educ, educ2, age, and age2 results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit() # Print the estimated parameters print(results.params)

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Make the DataFrame

df = pd.DataFrame()

df['educ'] = np.linspace(0,20)

df['age'] = 30

df['educ2'] = df['educ']\*\*2

df['age2'] = df['age']\*\*2

# Generate and plot the predictions

pred = results.predict(df)

print(pred.head())

import statsmodels.formula.api as smf

# Add a new column with educ squared

gss['educ2'] = gss['educ'] \*\*2

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Print the estimated parameters

print(results.params)

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Make the DataFrame

df = pd.DataFrame()

df['educ'] = np.linspace(0,20)

df['age'] = 30

df['educ2'] = df['educ']\*\*2

df['age2'] = df['age']\*\*2

# Generate and plot the predictions

pred = results.predict(df)

print(pred.head())

0 12182.345

1 11993.359

2 11857.672

3 11775.286

4 11746.199

dtype: float64

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Print the estimated parameters

print(results.params)

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Make the DataFrame

df = pd.DataFrame()

df['educ'] = np.linspace(0,20)

df['age'] = 30

df['educ2'] = df['educ']\*\*2

df['age2'] = df['age']\*\*2

Nice job. Now let's see what the results look like.

**Daily XP400**

##### Exercise

##### Exercise

# Visualizing predictions

Now let's visualize the results from the previous exercise!

##### Instructions

**100 XP**

* Plot mean\_income\_by\_educ using circles ('o'). Specify an alpha of 0.5.
* Plot the prediction results with a line, with df['educ'] on the x-axis and pred on the y-axis.

# Run a regression model with educ, educ2, age, and age2 results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit() # Make the DataFrame df = pd.DataFrame() df['educ'] = np.linspace(0,20) df['age'] = 30 df['educ2'] = df['educ']\*\*2 df['age2'] = df['age']\*\*2 # Generate and plot the predictions pred = results.predict(df) print(pred.head())

# Plot mean income in each age group

plt.clf()

grouped = gss.groupby('educ')

mean\_income\_by\_educ = grouped['realinc'].mean()

\_\_\_\_

# Plot the predictions

pred = results.predict(df)

plt.plot(\_\_\_\_, \_\_\_\_, label='Age 30')

# Label axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.legend()

plt.show()

# Plot mean income in each age group

plt.clf()

grouped = gss.groupby('educ')

mean\_income\_by\_educ = grouped['realinc'].mean()

plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5)

# Plot the predictions

pred = results.predict(df)

plt.plot(df['educ'], pred, label='Age 30')

# Label axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.legend()

plt.show()

# Run a regression model with educ, educ2, age, and age2

results = smf.ols('realinc ~ educ + educ2 + age + age2', data=gss).fit()

# Make the DataFrame

df = pd.DataFrame()

df['educ'] = np.linspace(0,20)

df['age'] = 30

df['educ2'] = df['educ']\*\*2

df['age2'] = df['age']\*\*2

# Generate and plot the predictions

pred = results.predict(df)

print(pred.head())

# Plot mean income in each age group

plt.clf()

grouped = gss.groupby('educ')

mean\_income\_by\_educ = grouped['realinc'].mean()

plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5)

# Plot the predictions

pred = results.predict(df)

plt.plot(df['educ'], pred, label='Age 30')

# Label axes

plt.xlabel('Education (years)')

plt.ylabel('Income (1986 $)')

plt.legend()

plt.show()

Looks like this model captures the relationship pretty well. Nice job.

**Daily XP500**

**Logistic regression**

**50 XP**

**1. Logistic regression**

Congratulations on making it this far! I have saved one of my favorite topics for the end - logistic regression.

**2. Categorical variables**

To understand logistic regression, we have to start with categorical variables. Most of the variables we have used so far - like income, age, and education - are numerical. But variables like sex and race are categorical; that is, each respondent belongs to one of a specified set of categories.

**3. Sex and income**

With StatsModels, it is easy to include a categorical variable as part of a regression model. Here's how. In the formula string, the letter C indicates that sex is a categorical variable. And here are the results. The regression treats the value sex=1, which is male, as the default, and reports the difference associated with the value sex=2, which is female. So this result indicates that income for women is about $4100 less than for men, after controlling for age and education.

**4. Boolean variable**

If a categorical variable has only two values, it's called a boolean variable. For example, one of the questions in the General Social Survey asks "Would you favor or oppose a law which would require a person to obtain a police permit before he or she could buy a gun?" The variable is called gunlaw, and here are the values. 1 means yes and 2 means no, so most respondents are in favor. To explore the relationship between this variable and factors like age, sex, and education, we can use logistic regression. StatsModels provides logistic regression, but to use it, we have to recode the variable so 1 means yes and 0 means no. We can do that by replacing 2 with 0. And we can check the results.

**5. Logistic regression**

Now we can run the regression. Instead of ols(), we use logit(), which is named for the logit function, which is related to logistic regression. Other than that, everything is the same as what we have seen before. And here are the results. The parameters are in the form of "log odds", which you may or may not be familiar with. I won't explain them in detail here, except to say that positive values are associated with things that make the outcome more likely; negative values make the outcome less likely. For example, the parameter associated with sex=2 is 0.75, which indicates that women are more likely to support this form of gun control. To see how much more likely, we can generate and plot predictions, as we did with linear regression.

**6. Generating predictions**

As an example, I'll generate predictions for different ages and sexes, with education held constant. First we need a DataFrame with age and educ. Then we can compute age2 and educ2. We can generate predictions for men like this. And for women like this.

**7. Visualizing results**

Now, to visualize the results, I start by plotting the data. As we've done before, we'll divide the respondents into age groups and compute the mean in each group. The mean of a binary variable is the fraction of people in favor. Now we can plot the predictions, for men and women, as a function of age. And label the axes.

**8. Gun laws and age**

Here's what it looks like. According to the model, people near age 50 are least likely to support gun control (at least as this question was posed). And women are more likely to support it than men, by almost 15 percentage points.

**9. Let's practice!**

Logistic regression is a powerful tool for exploring relationships between a binary variable and the factors that predict it. In the exercises, you'll explore the factors that predict support for legalizing marijuana.

**Daily XP550**

##### Exercise

##### Exercise

# Predicting a binary variable

Let's use logistic regression to predict a binary variable. Specifically, we'll use age, sex, and education level to predict support for legalizing cannabis (marijuana) in the U.S.

In the GSS dataset, the variable grass records the answer to the question "Do you think the use of marijuana should be made legal or not?"

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Fill in the parameters of smf.logit() to predict grass using the variables age, age2, educ, and educ2, along with sex as a categorical variable.

# Plot mean income in each age group plt.clf() grouped = gss.groupby('educ') mean\_income\_by\_educ = grouped['realinc'].mean() plt.plot(mean\_income\_by\_educ, 'o', alpha=0.5) # Plot the predictions pred = results.predict(df) plt.plot(df['educ'], pred, label='Age 30') # Label axes plt.xlabel('Education (years)') plt.ylabel('Income (1986 $)') plt.legend() plt.show()

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit(\_\_\_\_).fit()

results.params

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

Intercept -1.685e+00

C(sex)[T.2] -3.846e-01

age -3.476e-02

age2 1.917e-04

educ 2.219e-01

educ2 -4.163e-03

dtype: float64

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Predicting a binary variable

Let's use logistic regression to predict a binary variable. Specifically, we'll use age, sex, and education level to predict support for legalizing cannabis (marijuana) in the U.S.

In the GSS dataset, the variable grass records the answer to the question "Do you think the use of marijuana should be made legal or not?"

##### Instructions 2/4

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Add a column called educ and set it to 12 years; then compute a second column, educ2, which is the square of educ.

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

Intercept -1.685e+00

C(sex)[T.2] -3.846e-01

age -3.476e-02

age2 1.917e-04

educ 2.219e-01

educ2 -4.163e-03

dtype: float64

<script.py> output:

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = \_\_\_\_

df['educ2'] = \_\_\_\_

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

Generate separate predictions for men and women.

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

# Generate predictions for men and women

df['sex'] = 1

pred1 = results.\_\_\_\_(\_\_\_\_)

df['sex'] = 2

pred2 = results.\_\_\_\_(\_\_\_\_)

* Fill in the missing code to compute the mean of 'grass' for each age group, and then the arguments of plt.plot() to plot pred2 versus df['age'] with the label 'Female'.

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

# Generate predictions for men and women

df['sex'] = 1

pred1 = results.predict(df)

df['sex'] = 2

pred2 = results.predict(df)

plt.clf()

grouped = gss.groupby('age')

favor\_by\_age = \_\_\_\_

plt.plot(favor\_by\_age, 'o', alpha=0.5)

plt.plot(df['age'], pred1, label='Male')

plt.plot(\_\_\_\_)

plt.xlabel('Age')

plt.ylabel('Probability of favoring legalization')

plt.legend()

plt.show()

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

<script.py> output:

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

# Generate predictions for men and women

df['sex'] = 1

pred1 = results.predict(df)

df['sex'] = 2

pred2 = results.predict(df)

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

<script.py> output:

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

# Generate predictions for men and women

df['sex'] = 1

pred1 = results.predict(df)

df['sex'] = 2

pred2 = results.predict(df)

plt.clf()

grouped = gss.groupby('age')

favor\_by\_age = grouped['grass'].mean()

plt.plot(favor\_by\_age, 'o', alpha=0.5)

plt.plot(df['age'], pred1, label='Male')

plt.plot(df['age'], pred2, label='Female')

plt.xlabel('Age')

plt.ylabel('Probability of favoring legalization')

plt.legend()

plt.show()

# Recode grass

gss['grass'].replace(2, 0, inplace=True)

# Run logistic regression

results = smf.logit('grass ~ age + age2 + educ + educ2 + C(sex)', data=gss).fit()

results.params

# Make a DataFrame with a range of ages

df = pd.DataFrame()

df['age'] = np.linspace(18, 89)

df['age2'] = df['age']\*\*2

# Set the education level to 12

df['educ'] = 12

df['educ2'] = df['educ']\*\*2

# Generate predictions for men and women

df['sex'] = 1

pred1 = results.predict(df)

df['sex'] = 2

pred2 = results.predict(df)

plt.clf()

grouped = gss.groupby('age')

favor\_by\_age = grouped['grass'].mean()

plt.plot(favor\_by\_age, 'o', alpha=0.5)

plt.plot(df['age'], pred1, label='Male')

plt.plot(df['age'], pred2, label='Female')

plt.xlabel('Age')

plt.ylabel('Probability of favoring legalization')

plt.legend()

plt.show()

Optimization terminated successfully.

Current function value: 0.588510

Iterations 6

You made it! Congratulations on completing this course. I hope you enjoyed it and learned a lot. Should you wish to use the Pmf and Cdf classes from this course in your own work, you can download the empiricaldist library [here](https://pypi.org/project/empiricaldist/).

### Step Three: Exploratory Data Analysis

This subunit will walk you through step three of the Data Science Method: **exploratory data analysis.** Exploratory Data Analysis (EDA) is an approach for summarizing and visualizing the important characteristics and statistical properties of a dataset. Visualizing the data will help you make sense of it to identify emerging themes. Identifying these trends will help you to form hypotheses about the data.

1

### Overview of Exploratory Data Analysis

Save



5 - 10 Minutes

2 Points

Now that you have your data wrangled, it’s time to do some exploratory analysis! In this article by Dr. Guy Maskall, you’ll be given an introduction to exploratory data analysis and why it is essential to any data science workflow! This is a complex topic that you will revisit throughout the course.

2

### EDA Cheat Sheet (6 pages)

Open in new tab

Save



3 - 5 Minutes

1 Points

Use this EDA cheat sheet as a quick document to refer to whenever you conduct exploratory data analysis (you'll find it especially handy when you work on step three of the guided capstone).

3

### Exploratory Data Analysis in Python

Save



2 - 3 Hours

45 Points

Please complete the **first two chapters** of this DataCamp course. These chapters cover the Python coding skills needed to complete step three of the guided capstone (which is coming up next). While working through these chapters, you'll learn how to explore data relationships, find correlations, and create histogram plots.

4

### Guided Capstone - Step Three

Save

4 - 8 Hours

110 Points

For this step of your guided capstone, you'll use EDA to review data relationships, perform summary statistics, and create data visualizations for the Big Mountain case study. While working on this task, you may run up against unfamiliar challenges. We encourage you to use StackOverflow and Google search to explore how other data scientists have dealt with similar challenges.

Steps

1. Open the notebook 03\_exploratory\_data\_analysis.ipynb, which you’ll find under the folder “Notebooks”
2. Complete the notebook
3. Push the updated notebook to GitHub
4. Submit a link to the notebook on GitHub. Specifically link the location of your notebook, and not just the general link to your GitHub repository.

**Note:** In this step, you need to use the cleaned datafile you generated from Step 2: Data Wrangling. If you haven’t properly run the previous step, please discuss with your mentor to get unblocked.   
  
You can find the rubric [here](https://www.springboard.com/archeio/download/3e9937718e3e443091b9e074d331777a/).

Top of Form

Paste URL(s) to your document(s):

Top of Form



Bottom of Form

+ Add another link

Bottom of Form

You can submit a project multiple times.

[Jupyter Notebook](http://localhost:8890/tree?token=34101dfefd012e8bd0b705731d01b2a4049a6063cf628fdf)

03\_exploratory\_data\_analysis (autosaved)

Python 3 (ipykernel)

* [File](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Edit](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [View](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Insert](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Cell](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Kernel](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Widgets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Help](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)

# 3 Exploratory Data Analysis

## 3.1 Contents

* [3 Exploratory Data Analysis](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3_Exploratory_Data_Analysis)
  + [3.1 Contents](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.1_Contents)
  + [3.2 Introduction](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.2_Introduction)
  + [3.3 Imports](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.3_Imports)
  + [3.4 Load The Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.4_Load_The_Data)
    - [3.4.1 Ski data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.4.1_Ski_data)
    - [3.4.2 State-wide summary data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.4.2_State-wide_summary_data)
  + [3.5 Explore The Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5_Explore_The_Data)
    - [3.5.1 Top States By Order Of Each Of The Summary Statistics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1_Top_States_By_Order_Of_Each_Of_The_Summary_Statistics)
      * [3.5.1.1 Total state area](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.1_Total_state_area)
      * [3.5.1.2 Total state population](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.2_Total_state_population)
      * [3.5.1.3 Resorts per state](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.3_Resorts_per_state)
      * [3.5.1.4 Total skiable area](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.4_Total_skiable_area)
      * [3.5.1.5 Total night skiing area](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.5_Total_night_skiing_area)
      * [3.5.1.6 Total days open](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.6_Total_days_open)
    - [3.5.2 Resort density](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.2_Resort_density)
      * [3.5.2.1 Top states by resort density](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.2.1_Top_states_by_resort_density)
    - [3.5.3 Visualizing High Dimensional Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3_Visualizing_High_Dimensional_Data)
      * [3.5.3.1 Scale the data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.1_Scale_the_data)
        + [3.5.3.1.1 Verifying the scaling](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.1.1_Verifying_the_scaling)
      * [3.5.3.2 Calculate the PCA transformation](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.2_Calculate_the_PCA_transformation)
      * [3.5.3.3 Average ticket price by state](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.3_Average_ticket_price_by_state)
      * [3.5.3.4 Adding average ticket price to scatter plot](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.4_Adding_average_ticket_price_to_scatter_plot)
    - [3.5.4 Conclusion On How To Handle State Label](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.4_Conclusion_On_How_To_Handle_State_Label)
    - [3.5.5 Ski Resort Numeric Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5_Ski_Resort_Numeric_Data)
      * [3.5.5.1 Feature engineering](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5.1_Feature_engineering)
      * [3.5.5.2 Feature correlation heatmap](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5.2_Feature_correlation_heatmap)
      * [3.5.5.3 Scatterplots of numeric features against ticket price](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5.3_Scatterplots_of_numeric_features_against_ticket_price)
  + [3.6 Summary](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.6_Summary)

## 3.2 Introduction

At this point, you should have a firm idea of what your data science problem is and have the data you believe could help solve it. The business problem was a general one of modeling resort revenue. The data you started with contained some ticket price values, but with a number of missing values that led to several rows being dropped completely. You also had two kinds of ticket price. There were also some obvious issues with some of the other features in the data that, for example, led to one column being completely dropped, a data error corrected, and some other rows dropped. You also obtained some additional US state population and size data with which to augment the dataset, which also required some cleaning.

The data science problem you subsequently identified is to predict the adult weekend ticket price for ski resorts.

## 3.3 Imports



import pandas as pd

import numpy as np

import os

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.decomposition import PCA

from sklearn.preprocessing import scale

​

from library.sb\_utils import save\_file

## 3.4 Load The Data

### 3.4.1 Ski data



ski\_data = pd.read\_csv('../data/ski\_data\_cleaned.csv')



ski\_data.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 277 entries, 0 to 276

Data columns (total 25 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Name 277 non-null object

1 Region 277 non-null object

2 state 277 non-null object

3 summit\_elev 277 non-null int64

4 vertical\_drop 277 non-null int64

5 base\_elev 277 non-null int64

6 trams 277 non-null int64

7 fastSixes 277 non-null int64

8 fastQuads 277 non-null int64

9 quad 277 non-null int64

10 triple 277 non-null int64

11 double 277 non-null int64

12 surface 277 non-null int64

13 total\_chairs 277 non-null int64

14 Runs 274 non-null float64

15 TerrainParks 233 non-null float64

16 LongestRun\_mi 272 non-null float64

17 SkiableTerrain\_ac 275 non-null float64

18 Snow Making\_ac 240 non-null float64

19 daysOpenLastYear 233 non-null float64

20 yearsOpen 277 non-null float64

21 averageSnowfall 268 non-null float64

22 AdultWeekend 277 non-null float64

23 projectedDaysOpen 236 non-null float64

24 NightSkiing\_ac 163 non-null float64

dtypes: float64(11), int64(11), object(3)

memory usage: 54.2+ KB



ski\_data.head()

|  | **Name** | **Region** | **state** | **summit\_elev** | **vertical\_drop** | **base\_elev** | **trams** | **fastSixes** | **fastQuads** | **quad** | **...** | **TerrainParks** | **LongestRun\_mi** | **SkiableTerrain\_ac** | **Snow Making\_ac** | **daysOpenLastYear** | **yearsOpen** | **averageSnowfall** | **AdultWeekend** | **projectedDaysOpen** | **NightSkiing\_ac** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alyeska Resort | Alaska | Alaska | 3939 | 2500 | 250 | 1 | 0 | 2 | 2 | ... | 2.0 | 1.0 | 1610.0 | 113.0 | 150.0 | 60.0 | 669.0 | 85.0 | 150.0 | 550.0 |
| **1** | Eaglecrest Ski Area | Alaska | Alaska | 2600 | 1540 | 1200 | 0 | 0 | 0 | 0 | ... | 1.0 | 2.0 | 640.0 | 60.0 | 45.0 | 44.0 | 350.0 | 53.0 | 90.0 | NaN |
| **2** | Hilltop Ski Area | Alaska | Alaska | 2090 | 294 | 1796 | 0 | 0 | 0 | 0 | ... | 1.0 | 1.0 | 30.0 | 30.0 | 150.0 | 36.0 | 69.0 | 34.0 | 152.0 | 30.0 |
| **3** | Arizona Snowbowl | Arizona | Arizona | 11500 | 2300 | 9200 | 0 | 1 | 0 | 2 | ... | 4.0 | 2.0 | 777.0 | 104.0 | 122.0 | 81.0 | 260.0 | 89.0 | 122.0 | NaN |
| **4** | Sunrise Park Resort | Arizona | Arizona | 11100 | 1800 | 9200 | 0 | 0 | 1 | 2 | ... | 2.0 | 1.2 | 800.0 | 80.0 | 115.0 | 49.0 | 250.0 | 78.0 | 104.0 | 80.0 |

5 rows × 25 columns

### 3.4.2 State-wide summary data



state\_summary = pd.read\_csv('../data/state\_summary.csv')



state\_summary.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 35 entries, 0 to 34

Data columns (total 8 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 state 35 non-null object

1 resorts\_per\_state 35 non-null int64

2 state\_total\_skiable\_area\_ac 35 non-null float64

3 state\_total\_days\_open 35 non-null float64

4 state\_total\_terrain\_parks 35 non-null float64

5 state\_total\_nightskiing\_ac 35 non-null float64

6 state\_population 35 non-null int64

7 state\_area\_sq\_miles 35 non-null int64

dtypes: float64(4), int64(3), object(1)

memory usage: 2.3+ KB



state\_summary.head()

|  | **state** | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **state\_population** | **state\_area\_sq\_miles** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alaska | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 731545 | 665384 |
| **1** | Arizona | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 7278717 | 113990 |
| **2** | California | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 39512223 | 163695 |
| **3** | Colorado | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 5758736 | 104094 |
| **4** | Connecticut | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 3565278 | 5543 |

## 3.5 Explore The Data

### 3.5.1 Top States By Order Of Each Of The Summary Statistics

What does the state-wide picture for your market look like?



state\_summary\_newind = state\_summary.set\_index('state')

#### 3.5.1.1 Total state area



state\_summary\_newind.state\_area\_sq\_miles.sort\_values(ascending=False).head()

state

Alaska 665384

California 163695

Montana 147040

New Mexico 121590

Arizona 113990

Name: state\_area\_sq\_miles, dtype: int64

Your home state, Montana, comes in at third largest.

#### 3.5.1.2 Total state population



state\_summary\_newind.state\_population.sort\_values(ascending=False).head()

state

California 39512223

New York 19453561

Pennsylvania 12801989

Illinois 12671821

Ohio 11689100

Name: state\_population, dtype: int64

California dominates the state population figures despite coming in second behind Alaska in size (by a long way). The resort's state of Montana was in the top five for size, but doesn't figure in the most populous states. Thus your state is less densely populated.

#### 3.5.1.3 Resorts per state



state\_summary\_newind.resorts\_per\_state.sort\_values(ascending=False).head()

state

New York 33

Michigan 28

Colorado 22

California 21

Pennsylvania 19

Name: resorts\_per\_state, dtype: int64

New York comes top in the number of resorts in our market. Is this because of its proximity to wealthy New Yorkers wanting a convenient skiing trip? Or is it simply that its northerly location means there are plenty of good locations for resorts in that state?

#### 3.5.1.4 Total skiable area



state\_summary\_newind.state\_total\_skiable\_area\_ac.sort\_values(ascending=False).head()

state

Colorado 43682.0

Utah 30508.0

California 25948.0

Montana 21410.0

Idaho 16396.0

Name: state\_total\_skiable\_area\_ac, dtype: float64

New York state may have the most resorts, but they don't account for the most skiing area. In fact, New York doesn't even make it into the top five of skiable area. Good old Montana makes it into the top five, though. You may start to think that New York has more, smaller resorts, whereas Montana has fewer, larger resorts. Colorado seems to have a name for skiing; it's in the top five for resorts and in top place for total skiable area.

#### 3.5.1.5 Total night skiing area



state\_summary\_newind.state\_total\_nightskiing\_ac.sort\_values(ascending=False).head()

state

New York 2836.0

Washington 1997.0

Michigan 1946.0

Pennsylvania 1528.0

Oregon 1127.0

Name: state\_total\_nightskiing\_ac, dtype: float64

New York dominates the area of skiing available at night. Looking at the top five in general, they are all the more northerly states. Is night skiing in and of itself an appeal to customers, or is a consequence of simply trying to extend the skiing day where days are shorter? Is New York's domination here because it's trying to maximize its appeal to visitors who'd travel a shorter distance for a shorter visit? You'll find the data generates more (good) questions rather than answering them. This is a positive sign! You might ask your executive sponsor or data provider for some additional data about typical length of stays at these resorts, although you might end up with data that is very granular and most likely proprietary to each resort. A useful level of granularity might be "number of day tickets" and "number of weekly passes" sold.

#### 3.5.1.6 Total days open



state\_summary\_newind.state\_total\_days\_open.sort\_values(ascending=False).head()

state

Colorado 3258.0

California 2738.0

Michigan 2389.0

New York 2384.0

New Hampshire 1847.0

Name: state\_total\_days\_open, dtype: float64

The total days open seem to bear some resemblance to the number of resorts. This is plausible. The season will only be so long, and so the more resorts open through the skiing season, the more total days open we'll see. New Hampshire makes a good effort at making it into the top five, for a small state that didn't make it into the top five of resorts per state. Does its location mean resorts there have a longer season and so stay open longer, despite there being fewer of them?

### 3.5.2 Resort density

There are big states which are not necessarily the most populous. There are states that host many resorts, but other states host a larger total skiing area. The states with the most total days skiing per season are not necessarily those with the most resorts. And New York State boasts an especially large night skiing area. New York had the most resorts but wasn't in the top five largest states, so the reason for it having the most resorts can't be simply having lots of space for them. New York has the second largest population behind California. Perhaps many resorts have sprung up in New York because of the population size? Does this mean there is a high competition between resorts in New York State, fighting for customers and thus keeping prices down? You're not concerned, per se, with the absolute size or population of a state, but you could be interested in the ratio of resorts serving a given population or a given area.

So, calculate those ratios! Think of them as measures of resort density, and drop the absolute population and state size columns.



# The 100\_000 scaling is simply based on eyeballing the magnitudes of the data

state\_summary['resorts\_per\_100kcapita'] = 100\_000 \* state\_summary.resorts\_per\_state / state\_summary.state\_population

state\_summary['resorts\_per\_100ksq\_mile'] = 100\_000 \* state\_summary.resorts\_per\_state / state\_summary.state\_area\_sq\_miles

state\_summary.drop(columns=['state\_population', 'state\_area\_sq\_miles'], inplace=True)

state\_summary.head()

|  | **state** | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alaska | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 0.410091 | 0.450867 |
| **1** | Arizona | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 0.027477 | 1.754540 |
| **2** | California | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 0.053148 | 12.828736 |
| **3** | Colorado | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 0.382028 | 21.134744 |
| **4** | Connecticut | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 0.140242 | 90.203861 |

With the removal of the two columns that only spoke to state-specific data, you now have a Dataframe that speaks to the skiing competitive landscape of each state. It has the number of resorts per state, total skiable area, and days of skiing. You've translated the plain state data into something more useful that gives you an idea of the density of resorts relative to the state population and size.

How do the distributions of these two new features look?



state\_summary.resorts\_per\_100kcapita.hist(bins=30)

plt.xlabel('Number of resorts per 100k population')

plt.ylabel('count');

A picture containing text, crossword puzzle, public

Description automatically generated



state\_summary.resorts\_per\_100ksq\_mile.hist(bins=30)

plt.xlabel('Number of resorts per 100k square miles')

plt.ylabel('count');

Chart, histogram

Description automatically generated

So they have quite some long tails on them, but there's definitely some structure there.

#### 3.5.2.1 Top states by resort density



state\_summary.set\_index('state').resorts\_per\_100kcapita.sort\_values(ascending=False).head()

state

Vermont 2.403889

Wyoming 1.382268

New Hampshire 1.176721

Montana 1.122778

Idaho 0.671492

Name: resorts\_per\_100kcapita, dtype: float64



state\_summary.set\_index('state').resorts\_per\_100ksq\_mile.sort\_values(ascending=False).head()

state

New Hampshire 171.141299

Vermont 155.990017

Massachusetts 104.225886

Connecticut 90.203861

Rhode Island 64.724919

Name: resorts\_per\_100ksq\_mile, dtype: float64

Vermont seems particularly high in terms of resorts per capita, and both New Hampshire and Vermont top the chart for resorts per area. New York doesn't appear in either!

### 3.5.3 Visualizing High Dimensional Data

You may be starting to feel there's a bit of a problem here, or at least a challenge. You've constructed some potentially useful and business relevant features, derived from summary statistics, for each of the states you're concerned with. You've explored many of these features in turn and found various trends. Some states are higher in some but not in others. Some features will also be more correlated with one another than others.

One way to disentangle this interconnected web of relationships is via [principle components analysis](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA) (PCA). This technique will find linear combinations of the original features that are uncorrelated with one another and order them by the amount of variance they explain. You can use these derived features to visualize the data in a lower dimension (e.g. 2 down from 7) and know how much variance the representation explains. You can also explore how the original features contribute to these derived features.

The basic steps in this process are:

1. scale the data (important here because our features are heterogenous)
2. fit the PCA transformation (learn the transformation from the data)
3. apply the transformation to the data to create the derived features
4. (optionally) use the derived features to look for patterns in the data and explore the coefficients

#### 3.5.3.1 Scale the data

You only want numeric data here, although you don't want to lose track of the state labels, so it's convenient to set the state as the index.



#Code task 1#

#Create a new dataframe, `state\_summary\_scale` from `state\_summary` whilst setting the index to 'state'

state\_summary\_scale = state\_summary.set\_index(\_\_\_)

#Save the state labels (using the index attribute of `state\_summary\_scale`) into the variable 'state\_summary\_index'

state\_summary\_index = state\_summary\_scale.\_\_\_

#Save the column names (using the `columns` attribute) of `state\_summary\_scale` into the variable 'state\_summary\_columns'

state\_summary\_columns = state\_summary\_scale.\_\_\_

state\_summary\_scale.head()

The above shows what we expect: the columns we want are all numeric and the state has been moved to the index. Although, it's not necessary to step through the sequence so laboriously, it is often good practice even for experienced professionals. It's easy to make a mistake or forget a step, or the data may have been holding out a surprise! Stepping through like this helps validate both your work and the data!

Now use scale() to scale the data.



state\_summary\_scale = scale(state\_summary\_scale)

Note, scale() returns an ndarray, so you lose the column names. Because you want to visualise scaled data, you already copied the column names. Now you can construct a dataframe from the ndarray here and reintroduce the column names.



#Code task 2#

#Create a new dataframe from `state\_summary\_scale` using the column names we saved in `state\_summary\_columns`

state\_summary\_scaled\_df = pd.DataFrame(\_\_\_, columns=\_\_\_)

state\_summary\_scaled\_df.head()

##### 3.5.3.1.1 Verifying the scaling

This is definitely going the extra mile for validating your steps, but provides a worthwhile lesson.

First of all, check the mean of the scaled features using panda's mean() DataFrame method.



#Code task 3#

#Call `state\_summary\_scaled\_df`'s `mean()` method

state\_summary\_scaled\_df.\_\_\_

This is pretty much zero!

Perform a similar check for the standard deviation using pandas's std() DataFrame method.



#Code task 4#

#Call `state\_summary\_scaled\_df`'s `std()` method

state\_summary\_scaled\_df.\_\_\_

Well, this is a little embarrassing. The numbers should be closer to 1 than this! Check the documentation for [scale](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.scale.html) to see if you used it right. What about [std](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.std.html), did you mess up there? Is one of them not working right?

The keen observer, who already has some familiarity with statistical inference and biased estimators, may have noticed what's happened here. scale() uses the biased estimator for standard deviation (ddof=0). This doesn't mean it's bad! It simply means it calculates the standard deviation of the sample it was given. The std() method, on the other hand, defaults to using ddof=1, that is it's normalized by N-1. In other words, the std() method default is to assume you want your best estimate of the population parameter based on the given sample. You can tell it to return the biased estimate instead:



#Code task 5#

#Repeat the previous call to `std()` but pass in ddof=0

state\_summary\_scaled\_df.\_\_\_(\_\_\_)

There! Now it agrees with scale() and our expectation. This just goes to show different routines to do ostensibly the same thing can have different behaviours. Good practice is to keep validating your work and checking the documentation!

#### 3.5.3.2 Calculate the PCA transformation

Fit the PCA transformation using the scaled data.



state\_pca = PCA().fit(state\_summary\_scale)

Plot the cumulative variance ratio with number of components.



#Code task 6#

#Call the `cumsum()` method on the 'explained\_variance\_ratio\_' attribute of `state\_pca` and

#create a line plot to visualize the cumulative explained variance ratio with number of components

#Set the xlabel to 'Component #', the ylabel to 'Cumulative ratio variance', and the

#title to 'Cumulative variance ratio explained by PCA components for state/resort summary statistics'

#Hint: remember the handy ';' at the end of the last plot call to suppress that untidy output

plt.subplots(figsize=(10, 6))

plt.plot(state\_pca.explained\_variance\_ratio\_.\_\_\_)

plt.xlabel(\_\_\_)

plt.ylabel(\_\_\_)

plt.title(\_\_\_);

The first two components seem to account for over 75% of the variance, and the first four for over 95%.

**Note:** It is important to move quickly when performing exploratory data analysis. You should not spend hours trying to create publication-ready figures. However, it is crucially important that you can easily review and summarise the findings from EDA. Descriptive axis labels and titles are extremely useful here. When you come to reread your notebook to summarise your findings, you will be thankful that you created descriptive plots and even made key observations in adjacent markdown cells.

Apply the transformation to the data to obtain the derived features.



#Code task 7#

#Call `state\_pca`'s `transform()` method, passing in `state\_summary\_scale` as its argument

state\_pca\_x = state\_pca.\_\_\_(\_\_\_)



state\_pca\_x.shape

(35, 7)

Plot the first two derived features (the first two principle components) and label each point with the name of the state.

Take a moment to familiarize yourself with the code below. It will extract the first and second columns from the transformed data (state\_pca\_x) as x and y coordinates for plotting. Recall the state labels you saved (for this purpose) for subsequent calls to plt.annotate. Grab the second (index 1) value of the cumulative variance ratio to include in your descriptive title; this helpfully highlights the percentage variance explained by the two PCA components you're visualizing. Then create an appropriately sized and well-labelled scatterplot to convey all of this information.



x = state\_pca\_x[:, 0]

y = state\_pca\_x[:, 1]

state = state\_summary\_index

pc\_var = 100 \* state\_pca.explained\_variance\_ratio\_.cumsum()[1]

plt.subplots(figsize=(10,8))

plt.scatter(x=x, y=y)

plt.xlabel('First component')

plt.ylabel('Second component')

plt.title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained')

for s, x, y in zip(state, x, y):

plt.annotate(s, (x, y))

Timeline

Description automatically generated with medium confidence

#### 3.5.3.3 Average ticket price by state

Here, all point markers for the states are the same size and colour. You've visualized relationships between the states based on features such as the total skiable terrain area, but your ultimate interest lies in ticket prices. You know ticket prices for resorts in each state, so it might be interesting to see if there's any pattern there.



#Code task 8#

#Calculate the average 'AdultWeekend' ticket price by state

state\_avg\_price = ski\_data.groupby(\_\_\_)[\_\_\_].\_\_\_

state\_avg\_price.head()



state\_avg\_price.hist(bins=30)

plt.title('Distribution of state averaged prices')

plt.xlabel('Mean state adult weekend ticket price')

plt.ylabel('count');

A picture containing text, crossword puzzle

Description automatically generated

#### 3.5.3.4 Adding average ticket price to scatter plot

At this point you have several objects floating around. You have just calculated average ticket price by state from our ski resort data, but you've been looking at principle components generated from other state summary data. We extracted indexes and column names from a dataframe and the first two principle components from an array. It's becoming a bit hard to keep track of them all. You'll create a new DataFrame to do this.



#Code task 9#

#Create a dataframe containing the values of the first two PCA components

#Remember the first component was given by state\_pca\_x[:, 0],

#and the second by state\_pca\_x[:, 1]

#Call these 'PC1' and 'PC2', respectively and set the dataframe index to `state\_summary\_index`

pca\_df = pd.DataFrame({'PC1': \_\_\_, 'PC2': \_\_\_}, index=\_\_)

pca\_df.head()

That worked, and you have state as an index.



# our average state prices also have state as an index

state\_avg\_price.head()

state

Alaska 57.333333

Arizona 83.500000

California 81.416667

Colorado 90.714286

Connecticut 56.800000

Name: AdultWeekend, dtype: float64



# we can also cast it to a dataframe using Series' to\_frame() method:

state\_avg\_price.to\_frame().head()

|  | **AdultWeekend** |
| --- | --- |
| **state** |  |
| **Alaska** | 57.333333 |
| **Arizona** | 83.500000 |
| **California** | 81.416667 |
| **Colorado** | 90.714286 |
| **Connecticut** | 56.800000 |

Now you can concatenate both parts on axis 1 and using the indexes.



#Code task 10#

#Use pd.concat to concatenate `pca\_df` and `state\_avg\_price` along axis 1

# remember, pd.concat will align on index

pca\_df = \_\_\_([\_\_\_, \_\_\_], axis=\_\_\_)

pca\_df.head()

You saw some range in average ticket price histogram above, but it may be hard to pick out differences if you're thinking of using the value for point size. You'll add another column where you seperate these prices into quartiles; that might show something.



pca\_df['Quartile'] = pd.qcut(pca\_df.AdultWeekend, q=4, precision=1)

pca\_df.head()

|  | **PC1** | **PC2** | **AdultWeekend** | **Quartile** |
| --- | --- | --- | --- | --- |
| **Alaska** | -1.336533 | -0.182208 | 57.333333 | (53.1, 60.4] |
| **Arizona** | -1.839049 | -0.387959 | 83.500000 | (78.4, 93.0] |
| **California** | 3.537857 | -1.282509 | 81.416667 | (78.4, 93.0] |
| **Colorado** | 4.402210 | -0.898855 | 90.714286 | (78.4, 93.0] |
| **Connecticut** | -0.988027 | 1.020218 | 56.800000 | (53.1, 60.4] |



# Note that Quartile is a new data type: category

# This will affect how we handle it later on

pca\_df.dtypes

PC1 float64

PC2 float64

AdultWeekend float64

Quartile category

dtype: object

This looks great. But, let's have a healthy paranoia about it. You've just created a whole new DataFrame by combining information. Do we have any missing values? It's a narrow DataFrame, only four columns, so you'll just print out any rows that have any null values, expecting an empty DataFrame.



pca\_df[pca\_df.isnull().any(axis=1)]

|  | **PC1** | **PC2** | **AdultWeekend** | **Quartile** |
| --- | --- | --- | --- | --- |
| **Rhode Island** | -1.843646 | 0.761339 | NaN | NaN |

Ah, Rhode Island. How has this happened? Recall you created the original ski resort state summary dataset in the previous step before removing resorts with missing prices. This made sense because you wanted to capture all the other available information. However, Rhode Island only had one resort and its price was missing. You have two choices here. If you're interested in looking for any pattern with price, drop this row. But you are also generally interested in any clusters or trends, then you'd like to see Rhode Island even if the ticket price is unknown. So, replace these missing values to make it easier to handle/display them.

Because Quartile is a category type, there's an extra step here. Add the category (the string 'NA') that you're going to use as a replacement.



pca\_df['AdultWeekend'].fillna(pca\_df.AdultWeekend.mean(), inplace=True)

pca\_df['Quartile'] = pca\_df['Quartile'].cat.add\_categories('NA')

pca\_df['Quartile'].fillna('NA', inplace=True)

pca\_df.loc['Rhode Island']

PC1 -1.84365

PC2 0.761339

AdultWeekend 64.1244

Quartile NA

Name: Rhode Island, dtype: object

Note, in the above Quartile has the string value 'NA' that you inserted. This is different to numpy's NaN type.

You now have enough information to recreate the scatterplot, now adding marker size for ticket price and colour for the discrete quartile.

Notice in the code below how you're iterating over each quartile and plotting the points in the same quartile group as one. This gives a list of quartiles for an informative legend with points coloured by quartile and sized by ticket price (higher prices are represented by larger point markers).



x = pca\_df.PC1

y = pca\_df.PC2

price = pca\_df.AdultWeekend

quartiles = pca\_df.Quartile

state = pca\_df.index

pc\_var = 100 \* state\_pca.explained\_variance\_ratio\_.cumsum()[1]

fig, ax = plt.subplots(figsize=(10,8))

for q in quartiles.cat.categories:

im = quartiles == q

ax.scatter(x=x[im], y=y[im], s=price[im], label=q)

ax.set\_xlabel('First component')

ax.set\_ylabel('Second component')

plt.legend()

ax.set\_title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained')

for s, x, y in zip(state, x, y):

plt.annotate(s, (x, y))

Chart

Description automatically generated with low confidence

Now, you see the same distribution of states as before, but with additional information about the average price. There isn't an obvious pattern. The red points representing the upper quartile of price can be seen to the left, the right, and up top. There's also a spread of the other quartiles as well. In this representation of the ski summaries for each state, which accounts for some 77% of the variance, you simply do not seeing a pattern with price.

The above scatterplot was created using matplotlib. This is powerful, but took quite a bit of effort to set up. You have to iterate over the categories, plotting each separately, to get a colour legend. You can also tell that the points in the legend have different sizes as well as colours. As it happens, the size and the colour will be a 1:1 mapping here, so it happily works for us here. If we were using size and colour to display fundamentally different aesthetics, you'd have a lot more work to do. So matplotlib is powerful, but not ideally suited to when we want to visually explore multiple features as here (and intelligent use of colour, point size, and even shape can be incredibly useful for EDA).

Fortunately, there's another option: seaborn. You saw seaborn in action in the previous notebook, when you wanted to distinguish between weekend and weekday ticket prices in the boxplot. After melting the dataframe to have ticket price as a single column with the ticket type represented in a new column, you asked seaborn to create separate boxes for each type.



#Code task 11#

#Create a seaborn scatterplot by calling `sns.scatterplot`

#Specify the dataframe pca\_df as the source of the data,

#specify 'PC1' for x and 'PC2' for y,

#specify 'AdultWeekend' for the pointsize (scatterplot's `size` argument),

#specify 'Quartile' for `hue`

#specify pca\_df.Quartile.cat.categories for `hue\_order` - what happens with/without this?

x = pca\_df.PC1

y = pca\_df.PC2

state = pca\_df.index

plt.subplots(figsize=(12, 10))

# Note the argument below to make sure we get the colours in the ascending

# order we intuitively expect!

sns.\_\_\_(x=\_\_\_, y=\_\_\_, size=\_\_\_, hue=\_\_\_,

hue\_order=\_\_\_, data=pca\_df)

#and we can still annotate with the state labels

for s, x, y in zip(state, x, y):

plt.annotate(s, (x, y))

plt.title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained');

Seaborn does more! You should always care about your output. What if you want the ordering of the colours in the legend to align intuitively with the ordering of the quartiles? Add a hue\_order argument! Seaborn has thrown in a few nice other things:

* the aesthetics are separated in the legend
* it defaults to marker sizes that provide more contrast (smaller to larger)
* when starting with a DataFrame, you have less work to do to visualize patterns in the data

The last point is important. Less work means less chance of mixing up objects and jumping to erroneous conclusions. This also emphasizes the importance of getting data into a suitable DataFrame. In the previous notebook, you melted the data to make it longer, but with fewer columns, in order to get a single column of price with a new column representing a categorical feature you'd want to use. A **key skill** is being able to wrangle data into a form most suited to the particular use case.

Having gained a good visualization of the state summary data, you can discuss and follow up on your findings.

In the first two components, there is a spread of states across the first component. It looks like Vermont and New Hampshire might be off on their own a little in the second dimension, although they're really no more extreme than New York and Colorado are in the first dimension. But if you were curious, could you get an idea what it is that pushes Vermont and New Hampshire up?

The components\_ attribute of the fitted PCA object tell us how important (and in what direction) each feature contributes to each score (or coordinate on the plot). **NB we were sensible and scaled our original features (to zero mean and unit variance)**. You may not always be interested in interpreting the coefficients of the PCA transformation in this way, although it's more likely you will when using PCA for EDA as opposed to a preprocessing step as part of a machine learning pipeline. The attribute is actually a numpy ndarray, and so has been stripped of helpful index and column names. Fortunately, you thought ahead and saved these. This is how we were able to annotate the scatter plots above. It also means you can construct a DataFrame of components\_ with the feature names for context:



pd.DataFrame(state\_pca.components\_, columns=state\_summary\_columns)

|  | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.486079 | 0.318224 | 0.489997 | 0.488420 | 0.334398 | 0.187154 | 0.192250 |
| **1** | -0.085092 | -0.142204 | -0.045071 | -0.041939 | -0.351064 | 0.662458 | 0.637691 |
| **2** | -0.177937 | 0.714835 | 0.115200 | 0.005509 | -0.511255 | 0.220359 | -0.366207 |
| **3** | 0.056163 | -0.118347 | -0.162625 | -0.177072 | 0.438912 | 0.685417 | -0.512443 |
| **4** | -0.209186 | 0.573462 | -0.250521 | -0.388608 | 0.499801 | -0.065077 | 0.399461 |
| **5** | -0.818390 | -0.092319 | 0.238198 | 0.448118 | 0.246196 | 0.058911 | -0.009146 |
| **6** | -0.090273 | -0.127021 | 0.773728 | -0.613576 | 0.022185 | -0.007887 | -0.005631 |

For the row associated with the second component, are there any large values?

It looks like resorts\_per\_100kcapita and resorts\_per\_100ksq\_mile might count for quite a lot, in a positive sense. Be aware that sign matters; a large negative coefficient multiplying a large negative feature will actually produce a large positive PCA score.



state\_summary[state\_summary.state.isin(['New Hampshire', 'Vermont'])].T

|  | **17** | **29** |
| --- | --- | --- |
| **state** | New Hampshire | Vermont |
| **resorts\_per\_state** | 16 | 15 |
| **state\_total\_skiable\_area\_ac** | 3427 | 7239 |
| **state\_total\_days\_open** | 1847 | 1777 |
| **state\_total\_terrain\_parks** | 43 | 50 |
| **state\_total\_nightskiing\_ac** | 376 | 50 |
| **resorts\_per\_100kcapita** | 1.17672 | 2.40389 |
| **resorts\_per\_100ksq\_mile** | 171.141 | 155.99 |



state\_summary\_scaled\_df[state\_summary.state.isin(['New Hampshire', 'Vermont'])].T

|  | **17** | **29** |
| --- | --- | --- |
| **resorts\_per\_state** | 0.839478 | 0.712833 |
| **state\_total\_skiable\_area\_ac** | -0.277128 | 0.104681 |
| **state\_total\_days\_open** | 1.118608 | 1.034363 |
| **state\_total\_terrain\_parks** | 0.921793 | 1.233725 |
| **state\_total\_nightskiing\_ac** | -0.245050 | -0.747570 |
| **resorts\_per\_100kcapita** | 1.711066 | 4.226572 |
| **resorts\_per\_100ksq\_mile** | 3.483281 | 3.112841 |

So, yes, both states have particularly large values of resorts\_per\_100ksq\_mile in absolute terms, and these put them more than 3 standard deviations from the mean. Vermont also has a notably large value for resorts\_per\_100kcapita. New York, then, does not seem to be a stand-out for density of ski resorts either in terms of state size or population count.

### 3.5.4 Conclusion On How To Handle State Label

You can offer some justification for treating all states equally, and work towards building a pricing model that considers all states together, without treating any one particularly specially. You haven't seen any clear grouping yet, but you have captured potentially relevant state data in features most likely to be relevant to your business use case. This answers a big question!

### 3.5.5 Ski Resort Numeric Data



​

After what may feel a detour, return to examining the ski resort data. It's worth noting, the previous EDA was valuable because it's given us some potentially useful features, as well as validating an approach for how to subsequently handle the state labels in your modeling.



ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76 | 36 | 13 | 55 | 65 |
| **TerrainParks** | 2 | 1 | 1 | 4 | 2 |
| **LongestRun\_mi** | 1 | 2 | 1 | 2 | 1.2 |
| **SkiableTerrain\_ac** | 1610 | 640 | 30 | 777 | 800 |
| **Snow Making\_ac** | 113 | 60 | 30 | 104 | 80 |
| **daysOpenLastYear** | 150 | 45 | 150 | 122 | 115 |
| **yearsOpen** | 60 | 44 | 36 | 81 | 49 |
| **averageSnowfall** | 669 | 350 | 69 | 260 | 250 |
| **AdultWeekend** | 85 | 53 | 34 | 89 | 78 |
| **projectedDaysOpen** | 150 | 90 | 152 | 122 | 104 |
| **NightSkiing\_ac** | 550 | NaN | 30 | NaN | 80 |

#### 3.5.5.1 Feature engineering

Having previously spent some time exploring the state summary data you derived, you now start to explore the resort-level data in more detail. This can help guide you on how (or whether) to use the state labels in the data. It's now time to merge the two datasets and engineer some intuitive features. For example, you can engineer a resort's share of the supply for a given state.



state\_summary.head()

|  | **state** | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alaska | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 0.410091 | 0.450867 |
| **1** | Arizona | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 0.027477 | 1.754540 |
| **2** | California | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 0.053148 | 12.828736 |
| **3** | Colorado | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 0.382028 | 21.134744 |
| **4** | Connecticut | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 0.140242 | 90.203861 |



# DataFrame's merge method provides SQL-like joins

# here 'state' is a column (not an index)

ski\_data = ski\_data.merge(state\_summary, how='left', on='state')

ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76 | 36 | 13 | 55 | 65 |
| **TerrainParks** | 2 | 1 | 1 | 4 | 2 |
| **LongestRun\_mi** | 1 | 2 | 1 | 2 | 1.2 |
| **SkiableTerrain\_ac** | 1610 | 640 | 30 | 777 | 800 |
| **Snow Making\_ac** | 113 | 60 | 30 | 104 | 80 |
| **daysOpenLastYear** | 150 | 45 | 150 | 122 | 115 |
| **yearsOpen** | 60 | 44 | 36 | 81 | 49 |
| **averageSnowfall** | 669 | 350 | 69 | 260 | 250 |
| **AdultWeekend** | 85 | 53 | 34 | 89 | 78 |
| **projectedDaysOpen** | 150 | 90 | 152 | 122 | 104 |
| **NightSkiing\_ac** | 550 | NaN | 30 | NaN | 80 |
| **resorts\_per\_state** | 3 | 3 | 3 | 2 | 2 |
| **state\_total\_skiable\_area\_ac** | 2280 | 2280 | 2280 | 1577 | 1577 |
| **state\_total\_days\_open** | 345 | 345 | 345 | 237 | 237 |
| **state\_total\_terrain\_parks** | 4 | 4 | 4 | 6 | 6 |
| **state\_total\_nightskiing\_ac** | 580 | 580 | 580 | 80 | 80 |
| **resorts\_per\_100kcapita** | 0.410091 | 0.410091 | 0.410091 | 0.0274774 | 0.0274774 |
| **resorts\_per\_100ksq\_mile** | 0.450867 | 0.450867 | 0.450867 | 1.75454 | 1.75454 |

Having merged your state summary features into the ski resort data, add "state resort competition" features:

* ratio of resort skiable area to total state skiable area
* ratio of resort days open to total state days open
* ratio of resort terrain park count to total state terrain park count
* ratio of resort night skiing area to total state night skiing area

Once you've derived these features to put each resort within the context of its state,drop those state columns. Their main purpose was to understand what share of states' skiing "assets" is accounted for by each resort.



ski\_data['resort\_skiable\_area\_ac\_state\_ratio'] = ski\_data.SkiableTerrain\_ac / ski\_data.state\_total\_skiable\_area\_ac

ski\_data['resort\_days\_open\_state\_ratio'] = ski\_data.daysOpenLastYear / ski\_data.state\_total\_days\_open

ski\_data['resort\_terrain\_park\_state\_ratio'] = ski\_data.TerrainParks / ski\_data.state\_total\_terrain\_parks

ski\_data['resort\_night\_skiing\_state\_ratio'] = ski\_data.NightSkiing\_ac / ski\_data.state\_total\_nightskiing\_ac

​

ski\_data.drop(columns=['state\_total\_skiable\_area\_ac', 'state\_total\_days\_open',

'state\_total\_terrain\_parks', 'state\_total\_nightskiing\_ac'], inplace=True)

#### 3.5.5.2 Feature correlation heatmap

A great way to gain a high level view of relationships amongst the features.



#Code task 12#

#Show a seaborn heatmap of correlations in ski\_data

#Hint: call pandas' `corr()` method on `ski\_data` and pass that into `sns.heatmap`

plt.subplots(figsize=(12,10))

sns.\_\_\_(ski\_data.\_\_\_);

There is a lot to take away from this. First, summit and base elevation are quite highly correlated. This isn't a surprise. You can also see that you've introduced a lot of multicollinearity with your new ratio features; they are negatively correlated with the number of resorts in each state. This latter observation makes sense! If you increase the number of resorts in a state, the share of all the other state features will drop for each. An interesting observation in this region of the heatmap is that there is some positive correlation between the ratio of night skiing area with the number of resorts per capita. In other words, it seems that when resorts are more densely located with population, more night skiing is provided.

Turning your attention to your target feature, AdultWeekend ticket price, you see quite a few reasonable correlations. fastQuads stands out, along with Runs and Snow Making\_ac. The last one is interesting. Visitors would seem to value more guaranteed snow, which would cost in terms of snow making equipment, which would drive prices and costs up. Of the new features, resort\_night\_skiing\_state\_ratio seems the most correlated with ticket price. If this is true, then perhaps seizing a greater share of night skiing capacity is positive for the price a resort can charge.

As well as Runs, total\_chairs is quite well correlated with ticket price. This is plausible; the more runs you have, the more chairs you'd need to ferry people to them! Interestingly, they may count for more than the total skiable terrain area. For sure, the total skiable terrain area is not as useful as the area with snow making. People seem to put more value in guaranteed snow cover rather than more variable terrain area.

The vertical drop seems to be a selling point that raises ticket prices as well.

#### 3.5.5.3 Scatterplots of numeric features against ticket price

Correlations, particularly viewing them together as a heatmap, can be a great first pass at identifying patterns. But correlation can mask relationships between two variables. You'll now create a series of scatterplots to really dive into how ticket price varies with other numeric features.



# define useful function to create scatterplots of ticket prices against desired columns

def scatterplots(columns, ncol=None, figsize=(15, 8)):

if ncol is None:

ncol = len(columns)

nrow = int(np.ceil(len(columns) / ncol))

fig, axes = plt.subplots(nrow, ncol, figsize=figsize, squeeze=False)

fig.subplots\_adjust(wspace=0.5, hspace=0.6)

for i, col in enumerate(columns):

ax = axes.flatten()[i]

ax.scatter(x = col, y = 'AdultWeekend', data=ski\_data, alpha=0.5)

ax.set(xlabel=col, ylabel='Ticket price')

nsubplots = nrow \* ncol

for empty in range(i+1, nsubplots):

axes.flatten()[empty].set\_visible(False)



#Code task 13#

#Use a list comprehension to build a list of features from the columns of `ski\_data` that

#are \_not\_ any of 'Name', 'Region', 'state', or 'AdultWeekend'

features = [\_\_\_ for \_\_\_ in ski\_data.columns if \_\_\_ not in [\_\_\_, \_\_\_, \_\_\_, \_\_\_]]



scatterplots(features, ncol=4, figsize=(15, 15))

A picture containing text, electronics, display, several

Description automatically generated

In the scatterplots you see what some of the high correlations were clearly picking up on. There's a strong positive correlation with vertical\_drop. fastQuads seems very useful. Runs and total\_chairs appear quite similar and also useful. resorts\_per\_100kcapita shows something interesting that you don't see from just a headline correlation figure. When the value is low, there is quite a variability in ticket price, although it's capable of going quite high. Ticket price may drop a little before then climbing upwards as the number of resorts per capita increases. Ticket price could climb with the number of resorts serving a population because it indicates a popular area for skiing with plenty of demand. The lower ticket price when fewer resorts serve a population may similarly be because it's a less popular state for skiing. The high price for some resorts when resorts are rare (relative to the population size) may indicate areas where a small number of resorts can benefit from a monopoly effect. It's not a clear picture, although we have some interesting signs.

Finally, think of some further features that may be useful in that they relate to how easily a resort can transport people around. You have the numbers of various chairs, and the number of runs, but you don't have the ratio of chairs to runs. It seems logical that this ratio would inform you how easily, and so quickly, people could get to their next ski slope! Create these features now.



ski\_data['total\_chairs\_runs\_ratio'] = ski\_data.total\_chairs / ski\_data.Runs

ski\_data['total\_chairs\_skiable\_ratio'] = ski\_data.total\_chairs / ski\_data.SkiableTerrain\_ac

ski\_data['fastQuads\_runs\_ratio'] = ski\_data.fastQuads / ski\_data.Runs

ski\_data['fastQuads\_skiable\_ratio'] = ski\_data.fastQuads / ski\_data.SkiableTerrain\_ac



scatterplots(['total\_chairs\_runs\_ratio', 'total\_chairs\_skiable\_ratio',

'fastQuads\_runs\_ratio', 'fastQuads\_skiable\_ratio'], ncol=2)

Graphical user interface

Description automatically generated with medium confidence

At first these relationships are quite counterintuitive. It seems that the more chairs a resort has to move people around, relative to the number of runs, ticket price rapidly plummets and stays low. What we may be seeing here is an exclusive vs. mass market resort effect; if you don't have so many chairs, you can charge more for your tickets, although with fewer chairs you're inevitably going to be able to serve fewer visitors. Your price per visitor is high but your number of visitors may be low. Something very useful that's missing from the data is the number of visitors per year.

It also appears that having no fast quads may limit the ticket price, but if your resort covers a wide area then getting a small number of fast quads may be beneficial to ticket price.

## 3.6 Summary

**Q: 1** Write a summary of the exploratory data analysis above. What numerical or categorical features were in the data? Was there any pattern suggested of a relationship between state and ticket price? What did this lead us to decide regarding which features to use in subsequent modeling? What aspects of the data (e.g. relationships between features) should you remain wary of when you come to perform feature selection for modeling? Two key points that must be addressed are the choice of target feature for your modelling and how, if at all, you're going to handle the states labels in the data.

**A: 1** Your answer here



ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76 | 36 | 13 | 55 | 65 |
| **TerrainParks** | 2 | 1 | 1 | 4 | 2 |
| **LongestRun\_mi** | 1 | 2 | 1 | 2 | 1.2 |
| **SkiableTerrain\_ac** | 1610 | 640 | 30 | 777 | 800 |
| **Snow Making\_ac** | 113 | 60 | 30 | 104 | 80 |
| **daysOpenLastYear** | 150 | 45 | 150 | 122 | 115 |
| **yearsOpen** | 60 | 44 | 36 | 81 | 49 |
| **averageSnowfall** | 669 | 350 | 69 | 260 | 250 |
| **AdultWeekend** | 85 | 53 | 34 | 89 | 78 |
| **projectedDaysOpen** | 150 | 90 | 152 | 122 | 104 |
| **NightSkiing\_ac** | 550 | NaN | 30 | NaN | 80 |
| **resorts\_per\_state** | 3 | 3 | 3 | 2 | 2 |
| **resorts\_per\_100kcapita** | 0.410091 | 0.410091 | 0.410091 | 0.0274774 | 0.0274774 |
| **resorts\_per\_100ksq\_mile** | 0.450867 | 0.450867 | 0.450867 | 1.75454 | 1.75454 |
| **resort\_skiable\_area\_ac\_state\_ratio** | 0.70614 | 0.280702 | 0.0131579 | 0.492708 | 0.507292 |
| **resort\_days\_open\_state\_ratio** | 0.434783 | 0.130435 | 0.434783 | 0.514768 | 0.485232 |
| **resort\_terrain\_park\_state\_ratio** | 0.5 | 0.25 | 0.25 | 0.666667 | 0.333333 |
| **resort\_night\_skiing\_state\_ratio** | 0.948276 | NaN | 0.0517241 | NaN | 1 |
| **total\_chairs\_runs\_ratio** | 0.0921053 | 0.111111 | 0.230769 | 0.145455 | 0.107692 |
| **total\_chairs\_skiable\_ratio** | 0.00434783 | 0.00625 | 0.1 | 0.010296 | 0.00875 |
| **fastQuads\_runs\_ratio** | 0.0263158 | 0 | 0 | 0 | 0.0153846 |
| **fastQuads\_skiable\_ratio** | 0.00124224 | 0 | 0 | 0 | 0.00125 |



# Save the data

​

datapath = '../data'

save\_file(ski\_data, 'ski\_data\_step3\_features.csv', datapath)

[Jupyter Notebook](http://localhost:8890/tree?token=34101dfefd012e8bd0b705731d01b2a4049a6063cf628fdf)

03\_exploratory\_data\_analysis Last Checkpoint: 36 minutes ago (autosaved)

Python 3 (ipykernel)

* [File](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Edit](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [View](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Insert](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Cell](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Kernel](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Widgets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)
* [Help](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb)

# 3 Exploratory Data Analysis

## 3.1 Contents

* [3 Exploratory Data Analysis](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3_Exploratory_Data_Analysis)
  + [3.1 Contents](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.1_Contents)
  + [3.2 Introduction](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.2_Introduction)
  + [3.3 Imports](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.3_Imports)
  + [3.4 Load The Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.4_Load_The_Data)
    - [3.4.1 Ski data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.4.1_Ski_data)
    - [3.4.2 State-wide summary data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.4.2_State-wide_summary_data)
  + [3.5 Explore The Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5_Explore_The_Data)
    - [3.5.1 Top States By Order Of Each Of The Summary Statistics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1_Top_States_By_Order_Of_Each_Of_The_Summary_Statistics)
      * [3.5.1.1 Total state area](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.1_Total_state_area)
      * [3.5.1.2 Total state population](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.2_Total_state_population)
      * [3.5.1.3 Resorts per state](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.3_Resorts_per_state)
      * [3.5.1.4 Total skiable area](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.4_Total_skiable_area)
      * [3.5.1.5 Total night skiing area](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.5_Total_night_skiing_area)
      * [3.5.1.6 Total days open](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.1.6_Total_days_open)
    - [3.5.2 Resort density](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.2_Resort_density)
      * [3.5.2.1 Top states by resort density](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.2.1_Top_states_by_resort_density)
    - [3.5.3 Visualizing High Dimensional Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3_Visualizing_High_Dimensional_Data)
      * [3.5.3.1 Scale the data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.1_Scale_the_data)
        + [3.5.3.1.1 Verifying the scaling](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.1.1_Verifying_the_scaling)
      * [3.5.3.2 Calculate the PCA transformation](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.2_Calculate_the_PCA_transformation)
      * [3.5.3.3 Average ticket price by state](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.3_Average_ticket_price_by_state)
      * [3.5.3.4 Adding average ticket price to scatter plot](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.3.4_Adding_average_ticket_price_to_scatter_plot)
    - [3.5.4 Conclusion On How To Handle State Label](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.4_Conclusion_On_How_To_Handle_State_Label)
    - [3.5.5 Ski Resort Numeric Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5_Ski_Resort_Numeric_Data)
      * [3.5.5.1 Feature engineering](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5.1_Feature_engineering)
      * [3.5.5.2 Feature correlation heatmap](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5.2_Feature_correlation_heatmap)
      * [3.5.5.3 Scatterplots of numeric features against ticket price](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.5.5.3_Scatterplots_of_numeric_features_against_ticket_price)
  + [3.6 Summary](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/03_exploratory_data_analysis.ipynb#3.6_Summary)

## 3.2 Introduction

At this point, you should have a firm idea of what your data science problem is and have the data you believe could help solve it. The business problem was a general one of modeling resort revenue. The data you started with contained some ticket price values, but with a number of missing values that led to several rows being dropped completely. You also had two kinds of ticket price. There were also some obvious issues with some of the other features in the data that, for example, led to one column being completely dropped, a data error corrected, and some other rows dropped. You also obtained some additional US state population and size data with which to augment the dataset, which also required some cleaning.

The data science problem you subsequently identified is to predict the adult weekend ticket price for ski resorts.

## 3.3 Imports



import pandas as pd

import numpy as np

import os

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.decomposition import PCA

from sklearn.preprocessing import scale

​

from library.sb\_utils import save\_file

​

## 3.4 Load The Data

### 3.4.1 Ski data



ski\_data = pd.read\_csv('../data/ski\_data\_cleaned.csv')



ski\_data.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 277 entries, 0 to 276

Data columns (total 25 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Name 277 non-null object

1 Region 277 non-null object

2 state 277 non-null object

3 summit\_elev 277 non-null int64

4 vertical\_drop 277 non-null int64

5 base\_elev 277 non-null int64

6 trams 277 non-null int64

7 fastSixes 277 non-null int64

8 fastQuads 277 non-null int64

9 quad 277 non-null int64

10 triple 277 non-null int64

11 double 277 non-null int64

12 surface 277 non-null int64

13 total\_chairs 277 non-null int64

14 Runs 274 non-null float64

15 TerrainParks 233 non-null float64

16 LongestRun\_mi 272 non-null float64

17 SkiableTerrain\_ac 275 non-null float64

18 Snow Making\_ac 240 non-null float64

19 daysOpenLastYear 233 non-null float64

20 yearsOpen 277 non-null float64

21 averageSnowfall 268 non-null float64

22 AdultWeekend 277 non-null float64

23 projectedDaysOpen 236 non-null float64

24 NightSkiing\_ac 163 non-null float64

dtypes: float64(11), int64(11), object(3)

memory usage: 54.2+ KB



ski\_data.head()

|  | **Name** | **Region** | **state** | **summit\_elev** | **vertical\_drop** | **base\_elev** | **trams** | **fastSixes** | **fastQuads** | **quad** | **...** | **TerrainParks** | **LongestRun\_mi** | **SkiableTerrain\_ac** | **Snow Making\_ac** | **daysOpenLastYear** | **yearsOpen** | **averageSnowfall** | **AdultWeekend** | **projectedDaysOpen** | **NightSkiing\_ac** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alyeska Resort | Alaska | Alaska | 3939 | 2500 | 250 | 1 | 0 | 2 | 2 | ... | 2.0 | 1.0 | 1610.0 | 113.0 | 150.0 | 60.0 | 669.0 | 85.0 | 150.0 | 550.0 |
| **1** | Eaglecrest Ski Area | Alaska | Alaska | 2600 | 1540 | 1200 | 0 | 0 | 0 | 0 | ... | 1.0 | 2.0 | 640.0 | 60.0 | 45.0 | 44.0 | 350.0 | 53.0 | 90.0 | NaN |
| **2** | Hilltop Ski Area | Alaska | Alaska | 2090 | 294 | 1796 | 0 | 0 | 0 | 0 | ... | 1.0 | 1.0 | 30.0 | 30.0 | 150.0 | 36.0 | 69.0 | 34.0 | 152.0 | 30.0 |
| **3** | Arizona Snowbowl | Arizona | Arizona | 11500 | 2300 | 9200 | 0 | 1 | 0 | 2 | ... | 4.0 | 2.0 | 777.0 | 104.0 | 122.0 | 81.0 | 260.0 | 89.0 | 122.0 | NaN |
| **4** | Sunrise Park Resort | Arizona | Arizona | 11100 | 1800 | 9200 | 0 | 0 | 1 | 2 | ... | 2.0 | 1.2 | 800.0 | 80.0 | 115.0 | 49.0 | 250.0 | 78.0 | 104.0 | 80.0 |

5 rows × 25 columns

### 3.4.2 State-wide summary data



state\_summary = pd.read\_csv('../data/state\_summary.csv')



state\_summary.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 35 entries, 0 to 34

Data columns (total 8 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 state 35 non-null object

1 resorts\_per\_state 35 non-null int64

2 state\_total\_skiable\_area\_ac 35 non-null float64

3 state\_total\_days\_open 35 non-null float64

4 state\_total\_terrain\_parks 35 non-null float64

5 state\_total\_nightskiing\_ac 35 non-null float64

6 state\_population 35 non-null int64

7 state\_area\_sq\_miles 35 non-null int64

dtypes: float64(4), int64(3), object(1)

memory usage: 2.3+ KB



state\_summary.head()

|  | **state** | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **state\_population** | **state\_area\_sq\_miles** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alaska | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 731545 | 665384 |
| **1** | Arizona | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 7278717 | 113990 |
| **2** | California | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 39512223 | 163695 |
| **3** | Colorado | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 5758736 | 104094 |
| **4** | Connecticut | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 3565278 | 5543 |

## 3.5 Explore The Data

### 3.5.1 Top States By Order Of Each Of The Summary Statistics

What does the state-wide picture for your market look like?



state\_summary\_newind = state\_summary.set\_index('state')

#### 3.5.1.1 Total state area



state\_summary\_newind.state\_area\_sq\_miles.sort\_values(ascending=False).head()

state

Alaska 665384

California 163695

Montana 147040

New Mexico 121590

Arizona 113990

Name: state\_area\_sq\_miles, dtype: int64

Your home state, Montana, comes in at third largest.

#### 3.5.1.2 Total state population



state\_summary\_newind.state\_population.sort\_values(ascending=False).head()

state

California 39512223

New York 19453561

Pennsylvania 12801989

Illinois 12671821

Ohio 11689100

Name: state\_population, dtype: int64

California dominates the state population figures despite coming in second behind Alaska in size (by a long way). The resort's state of Montana was in the top five for size, but doesn't figure in the most populous states. Thus your state is less densely populated.

#### 3.5.1.3 Resorts per state



state\_summary\_newind.resorts\_per\_state.sort\_values(ascending=False).head()

state

New York 33

Michigan 28

Colorado 22

California 21

Pennsylvania 19

Name: resorts\_per\_state, dtype: int64

New York comes top in the number of resorts in our market. Is this because of its proximity to wealthy New Yorkers wanting a convenient skiing trip? Or is it simply that its northerly location means there are plenty of good locations for resorts in that state?

#### 3.5.1.4 Total skiable area



state\_summary\_newind.state\_total\_skiable\_area\_ac.sort\_values(ascending=False).head()

state

Colorado 43682.0

Utah 30508.0

California 25948.0

Montana 21410.0

Idaho 16396.0

Name: state\_total\_skiable\_area\_ac, dtype: float64

New York state may have the most resorts, but they don't account for the most skiing area. In fact, New York doesn't even make it into the top five of skiable area. Good old Montana makes it into the top five, though. You may start to think that New York has more, smaller resorts, whereas Montana has fewer, larger resorts. Colorado seems to have a name for skiing; it's in the top five for resorts and in top place for total skiable area.

#### 3.5.1.5 Total night skiing area



state\_summary\_newind.state\_total\_nightskiing\_ac.sort\_values(ascending=False).head()

state

New York 2836.0

Washington 1997.0

Michigan 1946.0

Pennsylvania 1528.0

Oregon 1127.0

Name: state\_total\_nightskiing\_ac, dtype: float64

New York dominates the area of skiing available at night. Looking at the top five in general, they are all the more northerly states. Is night skiing in and of itself an appeal to customers, or is a consequence of simply trying to extend the skiing day where days are shorter? Is New York's domination here because it's trying to maximize its appeal to visitors who'd travel a shorter distance for a shorter visit? You'll find the data generates more (good) questions rather than answering them. This is a positive sign! You might ask your executive sponsor or data provider for some additional data about typical length of stays at these resorts, although you might end up with data that is very granular and most likely proprietary to each resort. A useful level of granularity might be "number of day tickets" and "number of weekly passes" sold.

#### 3.5.1.6 Total days open



state\_summary\_newind.state\_total\_days\_open.sort\_values(ascending=False).head()

state

Colorado 3258.0

California 2738.0

Michigan 2389.0

New York 2384.0

New Hampshire 1847.0

Name: state\_total\_days\_open, dtype: float64

The total days open seem to bear some resemblance to the number of resorts. This is plausible. The season will only be so long, and so the more resorts open through the skiing season, the more total days open we'll see. New Hampshire makes a good effort at making it into the top five, for a small state that didn't make it into the top five of resorts per state. Does its location mean resorts there have a longer season and so stay open longer, despite there being fewer of them?

### 3.5.2 Resort density

There are big states which are not necessarily the most populous. There are states that host many resorts, but other states host a larger total skiing area. The states with the most total days skiing per season are not necessarily those with the most resorts. And New York State boasts an especially large night skiing area. New York had the most resorts but wasn't in the top five largest states, so the reason for it having the most resorts can't be simply having lots of space for them. New York has the second largest population behind California. Perhaps many resorts have sprung up in New York because of the population size? Does this mean there is a high competition between resorts in New York State, fighting for customers and thus keeping prices down? You're not concerned, per se, with the absolute size or population of a state, but you could be interested in the ratio of resorts serving a given population or a given area.

So, calculate those ratios! Think of them as measures of resort density, and drop the absolute population and state size columns.



# The 100\_000 scaling is simply based on eyeballing the magnitudes of the data

state\_summary['resorts\_per\_100kcapita'] = 100\_000 \* state\_summary.resorts\_per\_state / state\_summary.state\_population

state\_summary['resorts\_per\_100ksq\_mile'] = 100\_000 \* state\_summary.resorts\_per\_state / state\_summary.state\_area\_sq\_miles

state\_summary.drop(columns=['state\_population', 'state\_area\_sq\_miles'], inplace=True)

state\_summary.head()

|  | **state** | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alaska | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 0.410091 | 0.450867 |
| **1** | Arizona | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 0.027477 | 1.754540 |
| **2** | California | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 0.053148 | 12.828736 |
| **3** | Colorado | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 0.382028 | 21.134744 |
| **4** | Connecticut | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 0.140242 | 90.203861 |

With the removal of the two columns that only spoke to state-specific data, you now have a Dataframe that speaks to the skiing competitive landscape of each state. It has the number of resorts per state, total skiable area, and days of skiing. You've translated the plain state data into something more useful that gives you an idea of the density of resorts relative to the state population and size.

How do the distributions of these two new features look?



state\_summary.resorts\_per\_100kcapita.hist(bins=30)

plt.xlabel('Number of resorts per 100k population')

plt.ylabel('count');

Chart, histogram

Description automatically generated



state\_summary.resorts\_per\_100ksq\_mile.hist(bins=30)

plt.xlabel('Number of resorts per 100k square miles')

plt.ylabel('count');

Chart, histogram

Description automatically generated

So they have quite some long tails on them, but there's definitely some structure there.

#### 3.5.2.1 Top states by resort density



state\_summary.set\_index('state').resorts\_per\_100kcapita.sort\_values(ascending=False).head()

state

Vermont 2.403889

Wyoming 1.382268

New Hampshire 1.176721

Montana 1.122778

Idaho 0.671492

Name: resorts\_per\_100kcapita, dtype: float64



state\_summary.set\_index('state').resorts\_per\_100ksq\_mile.sort\_values(ascending=False).head()

state

New Hampshire 171.141299

Vermont 155.990017

Massachusetts 104.225886

Connecticut 90.203861

Rhode Island 64.724919

Name: resorts\_per\_100ksq\_mile, dtype: float64

Vermont seems particularly high in terms of resorts per capita, and both New Hampshire and Vermont top the chart for resorts per area. New York doesn't appear in either!

### 3.5.3 Visualizing High Dimensional Data

You may be starting to feel there's a bit of a problem here, or at least a challenge. You've constructed some potentially useful and business relevant features, derived from summary statistics, for each of the states you're concerned with. You've explored many of these features in turn and found various trends. Some states are higher in some but not in others. Some features will also be more correlated with one another than others.

One way to disentangle this interconnected web of relationships is via [principle components analysis](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA) (PCA). This technique will find linear combinations of the original features that are uncorrelated with one another and order them by the amount of variance they explain. You can use these derived features to visualize the data in a lower dimension (e.g. 2 down from 7) and know how much variance the representation explains. You can also explore how the original features contribute to these derived features.

The basic steps in this process are:

1. scale the data (important here because our features are heterogenous)
2. fit the PCA transformation (learn the transformation from the data)
3. apply the transformation to the data to create the derived features
4. (optionally) use the derived features to look for patterns in the data and explore the coefficients

#### 3.5.3.1 Scale the data

You only want numeric data here, although you don't want to lose track of the state labels, so it's convenient to set the state as the index.



#Code task 1#

#Create a new dataframe, `state\_summary\_scale` from `state\_summary` whilst setting the index to 'state'

state\_summary\_scale = state\_summary.set\_index('state')

#Save the state labels (using the index attribute of `state\_summary\_scale`) into the variable 'state\_summary\_index'

state\_summary\_index = state\_summary\_scale.index

#Save the column names (using the `columns` attribute) of `state\_summary\_scale` into the variable 'state\_summary\_columns'

state\_summary\_columns = state\_summary\_scale.columns

state\_summary\_scale.head()

|  | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **state** |  |  |  |  |  |  |  |
| **Alaska** | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 0.410091 | 0.450867 |
| **Arizona** | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 0.027477 | 1.754540 |
| **California** | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 0.053148 | 12.828736 |
| **Colorado** | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 0.382028 | 21.134744 |
| **Connecticut** | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 0.140242 | 90.203861 |

The above shows what we expect: the columns we want are all numeric and the state has been moved to the index. Although, it's not necessary to step through the sequence so laboriously, it is often good practice even for experienced professionals. It's easy to make a mistake or forget a step, or the data may have been holding out a surprise! Stepping through like this helps validate both your work and the data!

Now use scale() to scale the data.



state\_summary\_scale = scale(state\_summary\_scale)

Note, scale() returns an ndarray, so you lose the column names. Because you want to visualise scaled data, you already copied the column names. Now you can construct a dataframe from the ndarray here and reintroduce the column names.



#Code task 2#

#Create a new dataframe from `state\_summary\_scale` using the column names we saved in `state\_summary\_columns`

state\_summary\_scaled\_df = pd.DataFrame(state\_summary\_scale, columns=state\_summary\_columns)

state\_summary\_scaled\_df.head()

|  | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | -0.806912 | -0.392012 | -0.689059 | -0.816118 | 0.069410 | 0.139593 | -0.689999 |
| **1** | -0.933558 | -0.462424 | -0.819038 | -0.726994 | -0.701326 | -0.644706 | -0.658125 |
| **2** | 1.472706 | 1.978574 | 2.190933 | 2.615141 | 0.080201 | -0.592085 | -0.387368 |
| **3** | 1.599351 | 3.754811 | 2.816757 | 2.303209 | -0.164893 | 0.082069 | -0.184291 |
| **4** | -0.553622 | -0.584519 | -0.679431 | -0.548747 | -0.430027 | -0.413557 | 1.504408 |

##### 3.5.3.1.1 Verifying the scaling

This is definitely going the extra mile for validating your steps, but provides a worthwhile lesson.

First of all, check the mean of the scaled features using panda's mean() DataFrame method.



#Code task 3#

#Call `state\_summary\_scaled\_df`'s `mean()` method

state\_summary\_scaled\_df.mean()

resorts\_per\_state -6.344132e-17

state\_total\_skiable\_area\_ac -5.432163e-17

state\_total\_days\_open 9.754102e-17

state\_total\_terrain\_parks 4.282289e-17

state\_total\_nightskiing\_ac 6.344132e-17

resorts\_per\_100kcapita 5.075305e-17

resorts\_per\_100ksq\_mile 5.075305e-17

dtype: float64

This is pretty much zero!

Perform a similar check for the standard deviation using pandas's std() DataFrame method.



#Code task 4#

#Call `state\_summary\_scaled\_df`'s `std()` method

state\_summary\_scaled\_df.std()

resorts\_per\_state 1.014599

state\_total\_skiable\_area\_ac 1.014599

state\_total\_days\_open 1.014599

state\_total\_terrain\_parks 1.014599

state\_total\_nightskiing\_ac 1.014599

resorts\_per\_100kcapita 1.014599

resorts\_per\_100ksq\_mile 1.014599

dtype: float64

Well, this is a little embarrassing. The numbers should be closer to 1 than this! Check the documentation for [scale](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.scale.html) to see if you used it right. What about [std](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.std.html), did you mess up there? Is one of them not working right?

The keen observer, who already has some familiarity with statistical inference and biased estimators, may have noticed what's happened here. scale() uses the biased estimator for standard deviation (ddof=0). This doesn't mean it's bad! It simply means it calculates the standard deviation of the sample it was given. The std() method, on the other hand, defaults to using ddof=1, that is it's normalized by N-1. In other words, the std() method default is to assume you want your best estimate of the population parameter based on the given sample. You can tell it to return the biased estimate instead:



#Code task 5#

#Repeat the previous call to `std()` but pass in ddof=0

state\_summary\_scaled\_df.std(ddof=0)

resorts\_per\_state 1.0

state\_total\_skiable\_area\_ac 1.0

state\_total\_days\_open 1.0

state\_total\_terrain\_parks 1.0

state\_total\_nightskiing\_ac 1.0

resorts\_per\_100kcapita 1.0

resorts\_per\_100ksq\_mile 1.0

dtype: float64

There! Now it agrees with scale() and our expectation. This just goes to show different routines to do ostensibly the same thing can have different behaviours. Good practice is to keep validating your work and checking the documentation!

#### 3.5.3.2 Calculate the PCA transformation

Fit the PCA transformation using the scaled data.



state\_pca = PCA().fit(state\_summary\_scale)

Plot the cumulative variance ratio with number of components.



#Code task 6#

#Call the `cumsum()` method on the 'explained\_variance\_ratio\_' attribute of `state\_pca` and

#create a line plot to visualize the cumulative explained variance ratio with number of components

#Set the xlabel to 'Component #', the ylabel to 'Cumulative ratio variance', and the

#title to 'Cumulative variance ratio explained by PCA components for state/resort summary statistics'

#Hint: remember the handy ';' at the end of the last plot call to suppress that untidy output

plt.subplots(figsize=(10, 6))

plt.plot(state\_pca.explained\_variance\_ratio\_.cumsum())

plt.xlabel('Component #')

plt.ylabel('Cumulative ratio variance')

plt.title('Cumulative variance ratio explained by PCA components for state/resort summary statistics');

Chart, line chart

Description automatically generated

The first two components seem to account for over 75% of the variance, and the first four for over 95%.

**Note:** It is important to move quickly when performing exploratory data analysis. You should not spend hours trying to create publication-ready figures. However, it is crucially important that you can easily review and summarise the findings from EDA. Descriptive axis labels and titles are extremely useful here. When you come to reread your notebook to summarise your findings, you will be thankful that you created descriptive plots and even made key observations in adjacent markdown cells.

Apply the transformation to the data to obtain the derived features.



#Code task 7#

#Call `state\_pca`'s `transform()` method, passing in `state\_summary\_scale` as its argument

state\_pca\_x = state\_pca.transform(state\_summary\_scale)



state\_pca\_x.shape

(35, 7)

Plot the first two derived features (the first two principle components) and label each point with the name of the state.

Take a moment to familiarize yourself with the code below. It will extract the first and second columns from the transformed data (state\_pca\_x) as x and y coordinates for plotting. Recall the state labels you saved (for this purpose) for subsequent calls to plt.annotate. Grab the second (index 1) value of the cumulative variance ratio to include in your descriptive title; this helpfully highlights the percentage variance explained by the two PCA components you're visualizing. Then create an appropriately sized and well-labelled scatterplot to convey all of this information.



x = state\_pca\_x[:, 0]

y = state\_pca\_x[:, 1]

state = state\_summary\_index

pc\_var = 100 \* state\_pca.explained\_variance\_ratio\_.cumsum()[1]

plt.subplots(figsize=(10,8))

plt.scatter(x=x, y=y)

plt.xlabel('First component')

plt.ylabel('Second component')

plt.title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained')

for s, x, y in zip(state, x, y):

plt.annotate(s, (x, y))

A picture containing timeline

Description automatically generated

#### 3.5.3.3 Average ticket price by state

Here, all point markers for the states are the same size and colour. You've visualized relationships between the states based on features such as the total skiable terrain area, but your ultimate interest lies in ticket prices. You know ticket prices for resorts in each state, so it might be interesting to see if there's any pattern there.



#Code task 8#

#Calculate the average 'AdultWeekend' ticket price by state

state\_avg\_price = ski\_data.groupby('state')['AdultWeekend'].mean()

state\_avg\_price.head()

state

Alaska 57.333333

Arizona 83.500000

California 81.416667

Colorado 90.714286

Connecticut 56.800000

Name: AdultWeekend, dtype: float64



state\_avg\_price.hist(bins=30)

plt.title('Distribution of state averaged prices')

plt.xlabel('Mean state adult weekend ticket price')

plt.ylabel('count');

Chart, bar chart, histogram

Description automatically generated

#### 3.5.3.4 Adding average ticket price to scatter plot

At this point you have several objects floating around. You have just calculated average ticket price by state from our ski resort data, but you've been looking at principle components generated from other state summary data. We extracted indexes and column names from a dataframe and the first two principle components from an array. It's becoming a bit hard to keep track of them all. You'll create a new DataFrame to do this.



#Code task 9#

#Create a dataframe containing the values of the first two PCA components

#Remember the first component was given by state\_pca\_x[:, 0],

#and the second by state\_pca\_x[:, 1]

#Call these 'PC1' and 'PC2', respectively and set the dataframe index to `state\_summary\_index`

pca\_df = pd.DataFrame({'PC1': state\_pca\_x[:, 0], 'PC2': state\_pca\_x[:, 1]}, index=state\_summary\_index)

pca\_df.head()

|  | **PC1** | **PC2** |
| --- | --- | --- |
| **state** |  |  |
| **Alaska** | -1.336533 | -0.182208 |
| **Arizona** | -1.839049 | -0.387959 |
| **California** | 3.537857 | -1.282509 |
| **Colorado** | 4.402210 | -0.898855 |
| **Connecticut** | -0.988027 | 1.020218 |

That worked, and you have state as an index.



# our average state prices also have state as an index

state\_avg\_price.head()

state

Alaska 57.333333

Arizona 83.500000

California 81.416667

Colorado 90.714286

Connecticut 56.800000

Name: AdultWeekend, dtype: float64



# we can also cast it to a dataframe using Series' to\_frame() method:

state\_avg\_price.to\_frame().head()

|  | **AdultWeekend** |
| --- | --- |
| **state** |  |
| **Alaska** | 57.333333 |
| **Arizona** | 83.500000 |
| **California** | 81.416667 |
| **Colorado** | 90.714286 |
| **Connecticut** | 56.800000 |

Now you can concatenate both parts on axis 1 and using the indexes.



#Code task 10#

#Use pd.concat to concatenate `pca\_df` and `state\_avg\_price` along axis 1

# remember, pd.concat will align on index

pca\_df = pd.concat([pca\_df, state\_avg\_price], axis=1)

pca\_df.head()

|  | **PC1** | **PC2** | **AdultWeekend** |
| --- | --- | --- | --- |
| **state** |  |  |  |
| **Alaska** | -1.336533 | -0.182208 | 57.333333 |
| **Arizona** | -1.839049 | -0.387959 | 83.500000 |
| **California** | 3.537857 | -1.282509 | 81.416667 |
| **Colorado** | 4.402210 | -0.898855 | 90.714286 |
| **Connecticut** | -0.988027 | 1.020218 | 56.800000 |

You saw some range in average ticket price histogram above, but it may be hard to pick out differences if you're thinking of using the value for point size. You'll add another column where you seperate these prices into quartiles; that might show something.



pca\_df['Quartile'] = pd.qcut(pca\_df.AdultWeekend, q=4, precision=1)

pca\_df.head()

|  | **PC1** | **PC2** | **AdultWeekend** | **Quartile** |
| --- | --- | --- | --- | --- |
| **state** |  |  |  |  |
| **Alaska** | -1.336533 | -0.182208 | 57.333333 | (53.1, 60.4] |
| **Arizona** | -1.839049 | -0.387959 | 83.500000 | (78.4, 93.0] |
| **California** | 3.537857 | -1.282509 | 81.416667 | (78.4, 93.0] |
| **Colorado** | 4.402210 | -0.898855 | 90.714286 | (78.4, 93.0] |
| **Connecticut** | -0.988027 | 1.020218 | 56.800000 | (53.1, 60.4] |



# Note that Quartile is a new data type: category

# This will affect how we handle it later on

pca\_df.dtypes

PC1 float64

PC2 float64

AdultWeekend float64

Quartile category

dtype: object

This looks great. But, let's have a healthy paranoia about it. You've just created a whole new DataFrame by combining information. Do we have any missing values? It's a narrow DataFrame, only four columns, so you'll just print out any rows that have any null values, expecting an empty DataFrame.



pca\_df[pca\_df.isnull().any(axis=1)]

|  | **PC1** | **PC2** | **AdultWeekend** | **Quartile** |
| --- | --- | --- | --- | --- |
| **state** |  |  |  |  |
| **Rhode Island** | -1.843646 | 0.761339 | NaN | NaN |

Ah, Rhode Island. How has this happened? Recall you created the original ski resort state summary dataset in the previous step before removing resorts with missing prices. This made sense because you wanted to capture all the other available information. However, Rhode Island only had one resort and its price was missing. You have two choices here. If you're interested in looking for any pattern with price, drop this row. But you are also generally interested in any clusters or trends, then you'd like to see Rhode Island even if the ticket price is unknown. So, replace these missing values to make it easier to handle/display them.

Because Quartile is a category type, there's an extra step here. Add the category (the string 'NA') that you're going to use as a replacement.



pca\_df['AdultWeekend'].fillna(pca\_df.AdultWeekend.mean(), inplace=True)

pca\_df['Quartile'] = pca\_df['Quartile'].cat.add\_categories('NA')

pca\_df['Quartile'].fillna('NA', inplace=True)

pca\_df.loc['Rhode Island']

PC1 -1.843646

PC2 0.761339

AdultWeekend 64.124388

Quartile NA

Name: Rhode Island, dtype: object

Note, in the above Quartile has the string value 'NA' that you inserted. This is different to numpy's NaN type.

You now have enough information to recreate the scatterplot, now adding marker size for ticket price and colour for the discrete quartile.

Notice in the code below how you're iterating over each quartile and plotting the points in the same quartile group as one. This gives a list of quartiles for an informative legend with points coloured by quartile and sized by ticket price (higher prices are represented by larger point markers).



x = pca\_df.PC1

y = pca\_df.PC2

price = pca\_df.AdultWeekend

quartiles = pca\_df.Quartile

state = pca\_df.index

pc\_var = 100 \* state\_pca.explained\_variance\_ratio\_.cumsum()[1]

fig, ax = plt.subplots(figsize=(10,8))

for q in quartiles.cat.categories:

im = quartiles == q

ax.scatter(x=x[im], y=y[im], s=price[im], label=q)

ax.set\_xlabel('First component')

ax.set\_ylabel('Second component')

plt.legend()

ax.set\_title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained')

for s, x, y in zip(state, x, y):

plt.annotate(s, (x, y))

Chart, scatter chart

Description automatically generated

Now, you see the same distribution of states as before, but with additional information about the average price. There isn't an obvious pattern. The red points representing the upper quartile of price can be seen to the left, the right, and up top. There's also a spread of the other quartiles as well. In this representation of the ski summaries for each state, which accounts for some 77% of the variance, you simply do not seeing a pattern with price.

The above scatterplot was created using matplotlib. This is powerful, but took quite a bit of effort to set up. You have to iterate over the categories, plotting each separately, to get a colour legend. You can also tell that the points in the legend have different sizes as well as colours. As it happens, the size and the colour will be a 1:1 mapping here, so it happily works for us here. If we were using size and colour to display fundamentally different aesthetics, you'd have a lot more work to do. So matplotlib is powerful, but not ideally suited to when we want to visually explore multiple features as here (and intelligent use of colour, point size, and even shape can be incredibly useful for EDA).

Fortunately, there's another option: seaborn. You saw seaborn in action in the previous notebook, when you wanted to distinguish between weekend and weekday ticket prices in the boxplot. After melting the dataframe to have ticket price as a single column with the ticket type represented in a new column, you asked seaborn to create separate boxes for each type.



#Code task 11#

#Create a seaborn scatterplot by calling `sns.scatterplot`

#Specify the dataframe pca\_df as the source of the data,

#specify 'PC1' for x and 'PC2' for y,

#specify 'AdultWeekend' for the pointsize (scatterplot's `size` argument),

#specify 'Quartile' for `hue`

#specify pca\_df.Quartile.cat.categories for `hue\_order` - what happens with/without this?

x = pca\_df.PC1

y = pca\_df.PC2

state = pca\_df.index

plt.subplots(figsize=(12, 10))

# Note the argument below to make sure we get the colours in the ascending

# order we intuitively expect!

sns.scatterplot(x='PC1', y='PC2', size='AdultWeekend', hue='Quartile',

hue\_order=pca\_df.Quartile.cat.categories, data=pca\_df)

#and we can still annotate with the state labels

for s, x, y in zip(state, x, y):

plt.annotate(s, (x, y))

plt.title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained');

Chart

Description automatically generated

Seaborn does more! You should always care about your output. What if you want the ordering of the colours in the legend to align intuitively with the ordering of the quartiles? Add a hue\_order argument! Seaborn has thrown in a few nice other things:

* the aesthetics are separated in the legend
* it defaults to marker sizes that provide more contrast (smaller to larger)
* when starting with a DataFrame, you have less work to do to visualize patterns in the data

The last point is important. Less work means less chance of mixing up objects and jumping to erroneous conclusions. This also emphasizes the importance of getting data into a suitable DataFrame. In the previous notebook, you melted the data to make it longer, but with fewer columns, in order to get a single column of price with a new column representing a categorical feature you'd want to use. A **key skill** is being able to wrangle data into a form most suited to the particular use case.

Having gained a good visualization of the state summary data, you can discuss and follow up on your findings.

In the first two components, there is a spread of states across the first component. It looks like Vermont and New Hampshire might be off on their own a little in the second dimension, although they're really no more extreme than New York and Colorado are in the first dimension. But if you were curious, could you get an idea what it is that pushes Vermont and New Hampshire up?

The components\_ attribute of the fitted PCA object tell us how important (and in what direction) each feature contributes to each score (or coordinate on the plot). **NB we were sensible and scaled our original features (to zero mean and unit variance)**. You may not always be interested in interpreting the coefficients of the PCA transformation in this way, although it's more likely you will when using PCA for EDA as opposed to a preprocessing step as part of a machine learning pipeline. The attribute is actually a numpy ndarray, and so has been stripped of helpful index and column names. Fortunately, you thought ahead and saved these. This is how we were able to annotate the scatter plots above. It also means you can construct a DataFrame of components\_ with the feature names for context:



pd.DataFrame(state\_pca.components\_, columns=state\_summary\_columns)

|  | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.486079 | 0.318224 | 0.489997 | 0.488420 | 0.334398 | 0.187154 | 0.192250 |
| **1** | -0.085092 | -0.142204 | -0.045071 | -0.041939 | -0.351064 | 0.662458 | 0.637691 |
| **2** | -0.177937 | 0.714835 | 0.115200 | 0.005509 | -0.511255 | 0.220359 | -0.366207 |
| **3** | 0.056163 | -0.118347 | -0.162625 | -0.177072 | 0.438912 | 0.685417 | -0.512443 |
| **4** | -0.209186 | 0.573462 | -0.250521 | -0.388608 | 0.499801 | -0.065077 | 0.399461 |
| **5** | -0.818390 | -0.092319 | 0.238198 | 0.448118 | 0.246196 | 0.058911 | -0.009146 |
| **6** | -0.090273 | -0.127021 | 0.773728 | -0.613576 | 0.022185 | -0.007887 | -0.005631 |

For the row associated with the second component, are there any large values?

It looks like resorts\_per\_100kcapita and resorts\_per\_100ksq\_mile might count for quite a lot, in a positive sense. Be aware that sign matters; a large negative coefficient multiplying a large negative feature will actually produce a large positive PCA score.



state\_summary[state\_summary.state.isin(['New Hampshire', 'Vermont'])].T

|  | **17** | **29** |
| --- | --- | --- |
| **state** | New Hampshire | Vermont |
| **resorts\_per\_state** | 16 | 15 |
| **state\_total\_skiable\_area\_ac** | 3427.0 | 7239.0 |
| **state\_total\_days\_open** | 1847.0 | 1777.0 |
| **state\_total\_terrain\_parks** | 43.0 | 50.0 |
| **state\_total\_nightskiing\_ac** | 376.0 | 50.0 |
| **resorts\_per\_100kcapita** | 1.176721 | 2.403889 |
| **resorts\_per\_100ksq\_mile** | 171.141299 | 155.990017 |



state\_summary\_scaled\_df[state\_summary.state.isin(['New Hampshire', 'Vermont'])].T

|  | **17** | **29** |
| --- | --- | --- |
| **resorts\_per\_state** | 0.839478 | 0.712833 |
| **state\_total\_skiable\_area\_ac** | -0.277128 | 0.104681 |
| **state\_total\_days\_open** | 1.118608 | 1.034363 |
| **state\_total\_terrain\_parks** | 0.921793 | 1.233725 |
| **state\_total\_nightskiing\_ac** | -0.245050 | -0.747570 |
| **resorts\_per\_100kcapita** | 1.711066 | 4.226572 |
| **resorts\_per\_100ksq\_mile** | 3.483281 | 3.112841 |

So, yes, both states have particularly large values of resorts\_per\_100ksq\_mile in absolute terms, and these put them more than 3 standard deviations from the mean. Vermont also has a notably large value for resorts\_per\_100kcapita. New York, then, does not seem to be a stand-out for density of ski resorts either in terms of state size or population count.

### 3.5.4 Conclusion On How To Handle State Label

You can offer some justification for treating all states equally, and work towards building a pricing model that considers all states together, without treating any one particularly specially. You haven't seen any clear grouping yet, but you have captured potentially relevant state data in features most likely to be relevant to your business use case. This answers a big question!

### 3.5.5 Ski Resort Numeric Data



​

After what may feel a detour, return to examining the ski resort data. It's worth noting, the previous EDA was valuable because it's given us some potentially useful features, as well as validating an approach for how to subsequently handle the state labels in your modeling.



ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76.0 | 36.0 | 13.0 | 55.0 | 65.0 |
| **TerrainParks** | 2.0 | 1.0 | 1.0 | 4.0 | 2.0 |
| **LongestRun\_mi** | 1.0 | 2.0 | 1.0 | 2.0 | 1.2 |
| **SkiableTerrain\_ac** | 1610.0 | 640.0 | 30.0 | 777.0 | 800.0 |
| **Snow Making\_ac** | 113.0 | 60.0 | 30.0 | 104.0 | 80.0 |
| **daysOpenLastYear** | 150.0 | 45.0 | 150.0 | 122.0 | 115.0 |
| **yearsOpen** | 60.0 | 44.0 | 36.0 | 81.0 | 49.0 |
| **averageSnowfall** | 669.0 | 350.0 | 69.0 | 260.0 | 250.0 |
| **AdultWeekend** | 85.0 | 53.0 | 34.0 | 89.0 | 78.0 |
| **projectedDaysOpen** | 150.0 | 90.0 | 152.0 | 122.0 | 104.0 |
| **NightSkiing\_ac** | 550.0 | NaN | 30.0 | NaN | 80.0 |

#### 3.5.5.1 Feature engineering

Having previously spent some time exploring the state summary data you derived, you now start to explore the resort-level data in more detail. This can help guide you on how (or whether) to use the state labels in the data. It's now time to merge the two datasets and engineer some intuitive features. For example, you can engineer a resort's share of the supply for a given state.



state\_summary.head()

|  | **state** | **resorts\_per\_state** | **state\_total\_skiable\_area\_ac** | **state\_total\_days\_open** | **state\_total\_terrain\_parks** | **state\_total\_nightskiing\_ac** | **resorts\_per\_100kcapita** | **resorts\_per\_100ksq\_mile** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | Alaska | 3 | 2280.0 | 345.0 | 4.0 | 580.0 | 0.410091 | 0.450867 |
| **1** | Arizona | 2 | 1577.0 | 237.0 | 6.0 | 80.0 | 0.027477 | 1.754540 |
| **2** | California | 21 | 25948.0 | 2738.0 | 81.0 | 587.0 | 0.053148 | 12.828736 |
| **3** | Colorado | 22 | 43682.0 | 3258.0 | 74.0 | 428.0 | 0.382028 | 21.134744 |
| **4** | Connecticut | 5 | 358.0 | 353.0 | 10.0 | 256.0 | 0.140242 | 90.203861 |



# DataFrame's merge method provides SQL-like joins

# here 'state' is a column (not an index)

ski\_data = ski\_data.merge(state\_summary, how='left', on='state')

ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76.0 | 36.0 | 13.0 | 55.0 | 65.0 |
| **TerrainParks** | 2.0 | 1.0 | 1.0 | 4.0 | 2.0 |
| **LongestRun\_mi** | 1.0 | 2.0 | 1.0 | 2.0 | 1.2 |
| **SkiableTerrain\_ac** | 1610.0 | 640.0 | 30.0 | 777.0 | 800.0 |
| **Snow Making\_ac** | 113.0 | 60.0 | 30.0 | 104.0 | 80.0 |
| **daysOpenLastYear** | 150.0 | 45.0 | 150.0 | 122.0 | 115.0 |
| **yearsOpen** | 60.0 | 44.0 | 36.0 | 81.0 | 49.0 |
| **averageSnowfall** | 669.0 | 350.0 | 69.0 | 260.0 | 250.0 |
| **AdultWeekend** | 85.0 | 53.0 | 34.0 | 89.0 | 78.0 |
| **projectedDaysOpen** | 150.0 | 90.0 | 152.0 | 122.0 | 104.0 |
| **NightSkiing\_ac** | 550.0 | NaN | 30.0 | NaN | 80.0 |
| **resorts\_per\_state** | 3 | 3 | 3 | 2 | 2 |
| **state\_total\_skiable\_area\_ac** | 2280.0 | 2280.0 | 2280.0 | 1577.0 | 1577.0 |
| **state\_total\_days\_open** | 345.0 | 345.0 | 345.0 | 237.0 | 237.0 |
| **state\_total\_terrain\_parks** | 4.0 | 4.0 | 4.0 | 6.0 | 6.0 |
| **state\_total\_nightskiing\_ac** | 580.0 | 580.0 | 580.0 | 80.0 | 80.0 |
| **resorts\_per\_100kcapita** | 0.410091 | 0.410091 | 0.410091 | 0.027477 | 0.027477 |
| **resorts\_per\_100ksq\_mile** | 0.450867 | 0.450867 | 0.450867 | 1.75454 | 1.75454 |

Having merged your state summary features into the ski resort data, add "state resort competition" features:

* ratio of resort skiable area to total state skiable area
* ratio of resort days open to total state days open
* ratio of resort terrain park count to total state terrain park count
* ratio of resort night skiing area to total state night skiing area

Once you've derived these features to put each resort within the context of its state,drop those state columns. Their main purpose was to understand what share of states' skiing "assets" is accounted for by each resort.



ski\_data['resort\_skiable\_area\_ac\_state\_ratio'] = ski\_data.SkiableTerrain\_ac / ski\_data.state\_total\_skiable\_area\_ac

ski\_data['resort\_days\_open\_state\_ratio'] = ski\_data.daysOpenLastYear / ski\_data.state\_total\_days\_open

ski\_data['resort\_terrain\_park\_state\_ratio'] = ski\_data.TerrainParks / ski\_data.state\_total\_terrain\_parks

ski\_data['resort\_night\_skiing\_state\_ratio'] = ski\_data.NightSkiing\_ac / ski\_data.state\_total\_nightskiing\_ac

​

ski\_data.drop(columns=['state\_total\_skiable\_area\_ac', 'state\_total\_days\_open',

'state\_total\_terrain\_parks', 'state\_total\_nightskiing\_ac'], inplace=True)

#### 3.5.5.2 Feature correlation heatmap

A great way to gain a high level view of relationships amongst the features.



#Code task 12#

#Show a seaborn heatmap of correlations in ski\_data

#Hint: call pandas' `corr()` method on `ski\_data` and pass that into `sns.heatmap`

plt.subplots(figsize=(12,10))

sns.heatmap(ski\_data.corr());

Chart

Description automatically generated

There is a lot to take away from this. First, summit and base elevation are quite highly correlated. This isn't a surprise. You can also see that you've introduced a lot of multicollinearity with your new ratio features; they are negatively correlated with the number of resorts in each state. This latter observation makes sense! If you increase the number of resorts in a state, the share of all the other state features will drop for each. An interesting observation in this region of the heatmap is that there is some positive correlation between the ratio of night skiing area with the number of resorts per capita. In other words, it seems that when resorts are more densely located with population, more night skiing is provided.

Turning your attention to your target feature, AdultWeekend ticket price, you see quite a few reasonable correlations. fastQuads stands out, along with Runs and Snow Making\_ac. The last one is interesting. Visitors would seem to value more guaranteed snow, which would cost in terms of snow making equipment, which would drive prices and costs up. Of the new features, resort\_night\_skiing\_state\_ratio seems the most correlated with ticket price. If this is true, then perhaps seizing a greater share of night skiing capacity is positive for the price a resort can charge.

As well as Runs, total\_chairs is quite well correlated with ticket price. This is plausible; the more runs you have, the more chairs you'd need to ferry people to them! Interestingly, they may count for more than the total skiable terrain area. For sure, the total skiable terrain area is not as useful as the area with snow making. People seem to put more value in guaranteed snow cover rather than more variable terrain area.

The vertical drop seems to be a selling point that raises ticket prices as well.

#### 3.5.5.3 Scatterplots of numeric features against ticket price

Correlations, particularly viewing them together as a heatmap, can be a great first pass at identifying patterns. But correlation can mask relationships between two variables. You'll now create a series of scatterplots to really dive into how ticket price varies with other numeric features.



# define useful function to create scatterplots of ticket prices against desired columns

def scatterplots(columns, ncol=None, figsize=(15, 8)):

if ncol is None:

ncol = len(columns)

nrow = int(np.ceil(len(columns) / ncol))

fig, axes = plt.subplots(nrow, ncol, figsize=figsize, squeeze=False)

fig.subplots\_adjust(wspace=0.5, hspace=0.6)

for i, col in enumerate(columns):

ax = axes.flatten()[i]

ax.scatter(x = col, y = 'AdultWeekend', data=ski\_data, alpha=0.5)

ax.set(xlabel=col, ylabel='Ticket price')

nsubplots = nrow \* ncol

for empty in range(i+1, nsubplots):

axes.flatten()[empty].set\_visible(False)



#Code task 13#

#Use a list comprehension to build a list of features from the columns of `ski\_data` that

#are \_not\_ any of 'Name', 'Region', 'state', or 'AdultWeekend'

features = [list for list in ski\_data.columns if list not in ['Name', 'Region', 'state','AdultWeekend']]



scatterplots(features, ncol=4, figsize=(15, 15))

Calendar

Description automatically generated with medium confidence

In the scatterplots you see what some of the high correlations were clearly picking up on. There's a strong positive correlation with vertical\_drop. fastQuads seems very useful. Runs and total\_chairs appear quite similar and also useful. resorts\_per\_100kcapita shows something interesting that you don't see from just a headline correlation figure. When the value is low, there is quite a variability in ticket price, although it's capable of going quite high. Ticket price may drop a little before then climbing upwards as the number of resorts per capita increases. Ticket price could climb with the number of resorts serving a population because it indicates a popular area for skiing with plenty of demand. The lower ticket price when fewer resorts serve a population may similarly be because it's a less popular state for skiing. The high price for some resorts when resorts are rare (relative to the population size) may indicate areas where a small number of resorts can benefit from a monopoly effect. It's not a clear picture, although we have some interesting signs.

Finally, think of some further features that may be useful in that they relate to how easily a resort can transport people around. You have the numbers of various chairs, and the number of runs, but you don't have the ratio of chairs to runs. It seems logical that this ratio would inform you how easily, and so quickly, people could get to their next ski slope! Create these features now.



ski\_data['total\_chairs\_runs\_ratio'] = ski\_data.total\_chairs / ski\_data.Runs

ski\_data['total\_chairs\_skiable\_ratio'] = ski\_data.total\_chairs / ski\_data.SkiableTerrain\_ac

ski\_data['fastQuads\_runs\_ratio'] = ski\_data.fastQuads / ski\_data.Runs

ski\_data['fastQuads\_skiable\_ratio'] = ski\_data.fastQuads / ski\_data.SkiableTerrain\_ac



scatterplots(['total\_chairs\_runs\_ratio', 'total\_chairs\_skiable\_ratio',

'fastQuads\_runs\_ratio', 'fastQuads\_skiable\_ratio'], ncol=2)

Chart, scatter chart

Description automatically generated

At first these relationships are quite counterintuitive. It seems that the more chairs a resort has to move people around, relative to the number of runs, ticket price rapidly plummets and stays low. What we may be seeing here is an exclusive vs. mass market resort effect; if you don't have so many chairs, you can charge more for your tickets, although with fewer chairs you're inevitably going to be able to serve fewer visitors. Your price per visitor is high but your number of visitors may be low. Something very useful that's missing from the data is the number of visitors per year.

It also appears that having no fast quads may limit the ticket price, but if your resort covers a wide area then getting a small number of fast quads may be beneficial to ticket price.

## 3.6 Summary

**Q: 1** Write a summary of the exploratory data analysis above. What numerical or categorical features were in the data? Was there any pattern suggested of a relationship between state and ticket price? What did this lead us to decide regarding which features to use in subsequent modeling? What aspects of the data (e.g. relationships between features) should you remain wary of when you come to perform feature selection for modeling? Two key points that must be addressed are the choice of target feature for your modelling and how, if at all, you're going to handle the states labels in the data.

**A: 1** Your answer here



ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76.0 | 36.0 | 13.0 | 55.0 | 65.0 |
| **TerrainParks** | 2.0 | 1.0 | 1.0 | 4.0 | 2.0 |
| **LongestRun\_mi** | 1.0 | 2.0 | 1.0 | 2.0 | 1.2 |
| **SkiableTerrain\_ac** | 1610.0 | 640.0 | 30.0 | 777.0 | 800.0 |
| **Snow Making\_ac** | 113.0 | 60.0 | 30.0 | 104.0 | 80.0 |
| **daysOpenLastYear** | 150.0 | 45.0 | 150.0 | 122.0 | 115.0 |
| **yearsOpen** | 60.0 | 44.0 | 36.0 | 81.0 | 49.0 |
| **averageSnowfall** | 669.0 | 350.0 | 69.0 | 260.0 | 250.0 |
| **AdultWeekend** | 85.0 | 53.0 | 34.0 | 89.0 | 78.0 |
| **projectedDaysOpen** | 150.0 | 90.0 | 152.0 | 122.0 | 104.0 |
| **NightSkiing\_ac** | 550.0 | NaN | 30.0 | NaN | 80.0 |
| **resorts\_per\_state** | 3 | 3 | 3 | 2 | 2 |
| **resorts\_per\_100kcapita** | 0.410091 | 0.410091 | 0.410091 | 0.027477 | 0.027477 |
| **resorts\_per\_100ksq\_mile** | 0.450867 | 0.450867 | 0.450867 | 1.75454 | 1.75454 |
| **resort\_skiable\_area\_ac\_state\_ratio** | 0.70614 | 0.280702 | 0.013158 | 0.492708 | 0.507292 |
| **resort\_days\_open\_state\_ratio** | 0.434783 | 0.130435 | 0.434783 | 0.514768 | 0.485232 |
| **resort\_terrain\_park\_state\_ratio** | 0.5 | 0.25 | 0.25 | 0.666667 | 0.333333 |
| **resort\_night\_skiing\_state\_ratio** | 0.948276 | NaN | 0.051724 | NaN | 1.0 |
| **total\_chairs\_runs\_ratio** | 0.092105 | 0.111111 | 0.230769 | 0.145455 | 0.107692 |
| **total\_chairs\_skiable\_ratio** | 0.004348 | 0.00625 | 0.1 | 0.010296 | 0.00875 |
| **fastQuads\_runs\_ratio** | 0.026316 | 0.0 | 0.0 | 0.0 | 0.015385 |
| **fastQuads\_skiable\_ratio** | 0.001242 | 0.0 | 0.0 | 0.0 | 0.00125 |



# Save the data

​

datapath = '../data'

save\_file(ski\_data, 'ski\_data\_step3\_features.csv', datapath)