Principles and Techniques of Data Science

Data 100

Kanu Grover Bella Crouch

Table of contents

Welcome

About the Course Notes

This text was developed for the Spring 2023 Edition of the UC Berkeley course Data 100: Principles and Techniques of Data Science.

As this project is in development during the Spring 2023 semester, the course notes may be in flux. We appreciate your understanding. If you spot any errors or would like to suggest any changes, please email us. **Email**: data100.instructors@berkeley.edu

1 Introduction

> **Note**
>
> - Understand the stages of the data science lifecycle.

Data science is an interdisciplinary field with a variety of applications. The field is rapidly evolving; many of the key technical underpinnings in modern-day data science have been popularized during the early 21st century.

A true mastery of data science requires a deep theoretical understanding and strong grasp of domain expertise. This course will help you build on the former – specifically, the foundation of your technical knowledge. To do so, we've organized concepts in Data 100 around the **data science lifecycle**: an iterative process that encompasses the various statistical and computational building blocks of data science.

1.1 Data Science Lifecycle

The data science lifecycle is a high-level overview of the data science workflow. It's a cycle of stages that a data scientist should explore as they conduct a thorough analysis of a data-driven problem.

There are many variations of the key ideas present in the data science lifecycle. In Data 100, we visualize the stages of the lifecycle using a flow diagram. Notice how there are two entry points.

1.1.1 Ask a Question

Whether by curiosity or necessity, data scientists will constantly ask questions. For example, in the business world, data scientists may be interested in predicting the profit generated by a certain investment. In the field of medicine, they may ask whether some patients are more likely than others to benefit from a treatment.

Posing questions is one of the primary ways the data science lifecycle begins. It helps to fully define the question. Here are some things you should ask yourself before framing a question.

- What do we want to know?
  - A question that is too ambiguous may lead to confusion.
- What problems are we trying to solve?
  - The goal of asking a question should be clear in order to justify your efforts to stakeholders.
- What are the hypotheses we want to test?
  - This gives a clear perspective from which to analyze final results.
- What are the metrics for our success?
  - This gives a clear point to know when to finish the project.

1.1.2 Obtain Data

The second entry point to the lifecycle is by obtaining data. A careful analysis of any problem requires the use of data. Data may be readily available to us, or we may have to embark on a process to collect it. When doing so, its crucial to ask the following:

- What data do we have and what data do we need?
  - Define the units of the data (people, cities, points in time, etc.) and what features to measure.
- How will we sample more data?
  - Scrape the web, collect manually, etc.
- Is our data representative of the population we want to study?
  - If our data is not representative of our population of interest, then we can come to incorrect conclusions.

Key procedures: *data acquisition*, *data cleaning*

1.1.3 Understand the Data

Raw data itself is not inherently useful. It's impossible to discern all the patterns and relationships between variables without carefully investigating them. Therefore, translating pure data to actionable insights is a key job of a data scientist. For example, we may choose to ask:

- How is our data organized and what does it contain?

  - Knowing what the data says about the world helps us better understand the world.

- Do we have relevant data?
  - If the data we have collected is not useful to the question at hand, then we must collected more data.

- What are the biases, anomalies, or other issues with the data?
  - These can lead to many false conclusions if ignored, so data scientists must always be aware of these issues.

- How do we transform the data to enable effective analysis?
  - Data is not always easy to interpret at first glance, so a data scientist should reveal these hidden insights.

Key procedures: *exploratory data analysis*, *data visualization*.

1.1.4 Understand the World

After observing the patterns in our data, we can begin answering our question. This may require that we predict a quantity (machine learning), or measure the effect of some treatment (inference).

From here, we may choose to report our results, or possibly conduct more analysis. We may not be satisfied by our findings, or our initial exploration may have brought up new questions that require a new data.

- What does the data say about the world?
  - Given our models, the data will lead us to certain conclusions about the real world.

- Does it answer our questions or accurately solve the problem?
  - If our model and data can not accomplish our goals, then we must reform our question, model, or both.

- How robust are our conclusions and can we trust the predictions?
  - Inaccurate models can lead to untrue conclusions.

Key procedures: *model creation*, *prediction*, *inference*.

1.2 Conclusion

The data science lifecycle is meant to be a set of general guidelines rather than a hard list of requirements. In our journey exploring the lifecycle, we'll cover both the underlying theory and technologies used in data science, and we hope you'll build an appreciation for the field.

With that, let's begin by introducing one of the most important tools in exploratory data analysis: `pandas`.

2 Pandas I

> **Note**
>
> - Build familiarity with basic `pandas` syntax
> - Learn the methods of selecting and filtering data from a DataFrame.
> - Understand the differences between DataFrames and Series

Data scientists work with data stored in a variety of formats. The primary focus of this class is in understanding tabular data – one of the most widely used formats in data science. This note introduces DataFrames, which are among the most popular representations of tabular data. We'll also introduce `pandas`, the standard Python package for manipulating data in DataFrames.

2.1 Introduction to Exploratory Data Analysis

Imagine you collected, or have been given a box of data. What do you do next?

The first step is to clean your data. **Data cleaning** often corrects issues in the structure and formatting of data, including missing values and unit conversions.

Data scientists have coined the term **exploratory data analysis (EDA)** to describe the process of transforming raw data to insightful observations. EDA is an *open-ended* analysis of transforming, visualizing, and summarizing patterns in data. In order to conduct EDA, we first need to familiarize ourselves with `pandas` – an important programming tool.

2.2 Introduction to Pandas

`pandas` is a data analysis library to make data cleaning and analysis fast and convenient in Python.

The `pandas` library adopts many coding idioms from `NumPy`. The biggest difference is that `pandas` is designed for working with tabular data, one of the most common data formats (and the focus of Data 100).

Before writing any code, we must import `pandas` into our Python environment.

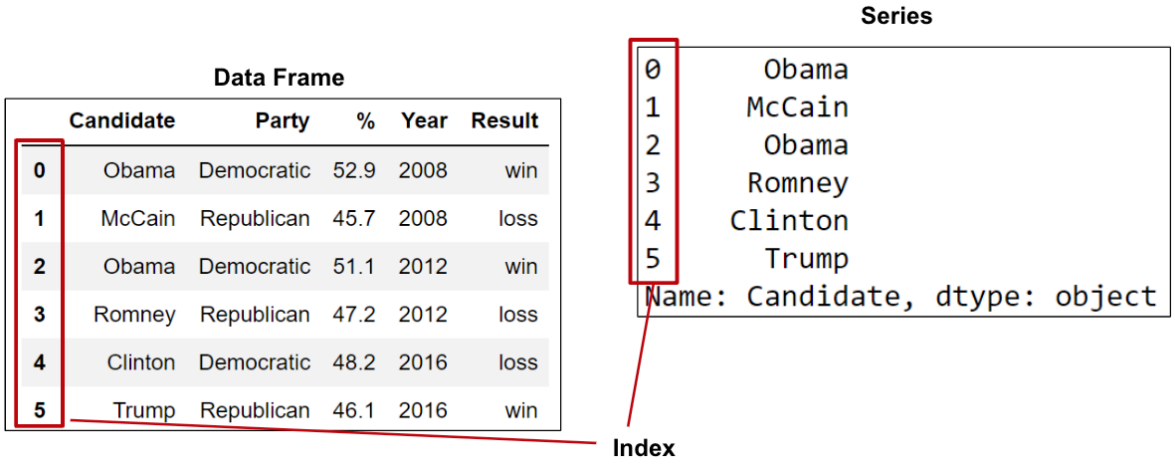
```
# `pd` is the conventional alias for Pandas, as `np` is for NumPy
import pandas as pd
```

2.3 Series, DataFrames, and Indices

There are three fundamental data structures in `pandas`:

1. **Series**: 1D labeled array data; best thought of as columnar data
2. **DataFrame**: 2D tabular data with rows and columns
3. **Index**: A sequence of row/column labels

DataFrames, Series, and Indices can be represented visually in the following diagram.



	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

```
0      Obama
1     McCain
2      Obama
3     Romney
4    Clinton
5      Trump
Name: Candidate, dtype: object
```

Notice how the **DataFrame** is a two dimensional object – it contains both rows and columns. The **Series** above is a singular column of this DataFrame, namely the `Candidate` column. Both contain an **Index**, or a shared list of row labels (the integers from 0 to 5, inclusive).

2.3.1 Series

A Series represents a column of a DataFrame; more generally, it can be any 1-dimensional array-like object containing values of the same type with associated data labels, called its index.

```
import pandas as pd

s = pd.Series([-1, 10, 2])
```

```
print(s)
```

```
0    -1
1    10
2     2
dtype: int64
```

```
s.array # Data contained within the Series
```

```
<PandasArray>
[-1, 10, 2]
Length: 3, dtype: int64
```

```
s.index # The Index of the Series
```

```
RangeIndex(start=0, stop=3, step=1)
```

By default, row indices in `pandas` are a sequential list of integers beginning from 0. Optionally, a list of desired indices can be passed to the `index` argument.

```
s = pd.Series([-1, 10, 2], index = ["a", "b", "c"])
print(s)
```

```
a    -1
b    10
c     2
dtype: int64
```

Indices can also be changed after initialization.

```
s.index = ["first", "second", "third"]
print(s)
```

```
first     -1
second    10
third      2
dtype: int64
```

2.3.1.1 Selection in Series

Similar to an array, we can select a single value or a set of values from a Series. There are 3 primary methods of selecting data.

1. A single index label
2. A list of index labels
3. A filtering condition

Let's define the following Series `ser`.

```
ser = pd.Series([4, -2, 0, 6], index = ["a", "b", "c", "d"])
print(ser)
```

```
a    4
b   -2
c    0
d    6
dtype: int64
```

2.3.1.1.1 A Single Index Label

```
print(ser["a"]) # Notice how the return value is a single array element
```

```
4
```

2.3.1.1.2 A List of Index Labels

```
ser[["a", "c"]] # Notice how the return value is another Series
```

	0
a	4
c	0

2.3.1.1.3 A Filtering Condition

Perhaps the most interesting (and useful) method of selecting data from a Series is with a filtering condition.

We first must apply a vectorized boolean operation to our Series that encodes the filter condition.

```
ser > 0 # Filter condition: select all elements greater than 0
```

	0
a	True
b	False
c	False
d	True

Upon "indexing" in our Series with this condition, `pandas` selects only the rows with `True` values.

```
ser[ser > 0]
```

	0
a	4
d	6

2.3.2 DataFrames

In Data 8, you encountered the `Table` class of the `datascience` library, which represented tabular data. In Data 100, we'll be using the `DataFrame` class of the `pandas` library.

Here is an example of a DataFrame that contains election data.

```
import pandas as pd

elections = pd.read_csv("data/elections.csv")
elections
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789
5	1832	Henry Clay	National Republican	484205	loss	37.603628
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
7	1836	Hugh Lawson White	Whig	146109	loss	10.005985
8	1836	Martin Van Buren	Democratic	763291	win	52.272472
9	1836	William Henry Harrison	Whig	550816	loss	37.721543
10	1840	Martin Van Buren	Democratic	1128854	loss	46.948787
11	1840	William Henry Harrison	Whig	1275583	win	53.051213
12	1844	Henry Clay	Whig	1300004	loss	49.250523
13	1844	James Polk	Democratic	1339570	win	50.749477
14	1848	Lewis Cass	Democratic	1223460	loss	42.552229
15	1848	Martin Van Buren	Free Soil	291501	loss	10.138474
16	1848	Zachary Taylor	Whig	1360235	win	47.309296
17	1852	Franklin Pierce	Democratic	1605943	win	51.013168
18	1852	John P. Hale	Free Soil	155210	loss	4.930283
19	1852	Winfield Scott	Whig	1386942	loss	44.056548
20	1856	James Buchanan	Democratic	1835140	win	45.306080
21	1856	John C. Frémont	Republican	1342345	loss	33.139919
22	1856	Millard Fillmore	American	873053	loss	21.554001
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
27	1864	Abraham Lincoln	National Union	2211317	win	54.951512
28	1864	George B. McClellan	Democratic	1812807	loss	45.048488
29	1868	Horatio Seymour	Democratic	2708744	loss	47.334695
30	1868	Ulysses Grant	Republican	3013790	win	52.665305
31	1872	Horace Greeley	Liberal Republican	2834761	loss	44.071406
32	1872	Ulysses Grant	Republican	3597439	win	55.928594
33	1876	Rutherford Hayes	Republican	4034142	win	48.471624
34	1876	Samuel J. Tilden	Democratic	4288546	loss	51.528376
35	1880	James B. Weaver	Greenback	308649	loss	3.352344
36	1880	James Garfield	Republican	4453337	win	48.369234
37	1880	Winfield Scott Hancock	Democratic	4444976	loss	48.278422
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
39	1884	Grover Cleveland	Democratic	4914482	win	48.884933
40	1884	James G. Blaine	Republican	4856905	loss	48.312208
41	1884	John St. John	Prohibition	147482	loss	1.467021
42	1888	Alson Streeter	Union Labor	146602	loss	1.288861
43	1888	Benjamin Harrison	Republican	5443633	win	47.858041
44	1888	Clinton B. Fisk	Prohibition	249819	loss	2.196299
45	1888	Grover Cleveland	Democratic	5534488	loss	48.656799
46	1892	Benjamin Harrison	Republican	5176108	loss	42.984101
47	1892	Grover Cleveland	Democratic	5553898	win	46.121393
48	1892	James B. Weaver	Populist	1041028	loss	8.645038
49	1892	John Bidwell	Prohibition	270879	loss	2.249468
50	1896	John M. Palmer	National Democratic	134645	loss	0.969566
51	1896	Joshua Levering	Prohibition	131312	loss	0.945565

Let's dissect the code above.

1. We first import the `pandas` library into our Python environment, using the alias `pd`. `import pandas as pd`
2. There are a number of ways to read data into a DataFrame. In Data 100, our data are typically stored in a CSV (comma-seperated values) file format. We can import a CSV file into a DataFrame by passing the data path as an argument to the following `pandas` function. `pd.read_csv("elections.csv")`

This code stores our DataFrame object in the `elections` variable. Upon inspection, our `elections` DataFrame has 182 rows and 6 columns (`Year`, `Candidate`, `Party`, `Popular Vote`, `Result`, `%`). Each row represents a single record – in our example, a presedential candidate from some particular year. Each column represents a single attribute, or feature of the record.

In the example above, we constructed a DataFrame object using data from a CSV file. As we'll explore in the next section, we can create a DataFrame with data of our own.

2.3.2.1 Creating a DataFrame

There are many ways to create a DataFrame. Here, we will cover the most popular approaches.

1. Using a list and column names
2. From a dictionary
3. From a Series

2.3.2.1.1 Using a List and Column Names

Consider the following examples.

```
df_list = pd.DataFrame([1, 2, 3], columns=["Numbers"])
df_list
```

	Numbers
0	1
1	2
2	3

The first code cell creates a DataFrame with a single column `Numbers`, while the second creates a DataFrame with an additional column `Description`. Notice how a 2D list of values is required to initialize the second DataFrame – each nested list represents a single row of data.

```
df_list = pd.DataFrame([[1, "one"], [2, "two"]], columns = ["Number", "Description"])
df_list
```

	Number	Description
0	1	one
1	2	two

2.3.2.1.2 From a Dictionary

A second (and more common) way to create a DataFrame is with a dictionary. The dictionary keys represent the column names, and the dictionary values represent the column values.

```
df_dict = pd.DataFrame({"Fruit": ["Strawberry", "Orange"], "Price": [5.49, 3.99]})
df_dict
```

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

2.3.2.1.3 From a Series

Earlier, we explained how a Series was synonymous to a column in a DataFrame. It follows then, that a DataFrame is equivalent to a collection of Series, which all share the same index.

In fact, we can initialize a DataFrame by merging two or more Series.

```
# Notice how our indices, or row labels, are the same

s_a = pd.Series(["a1", "a2", "a3"], index = ["r1", "r2", "r3"])
s_b = pd.Series(["b1", "b2", "b3"], index = ["r1", "r2", "r3"])

pd.DataFrame({"A-column": s_a, "B-column": s_b})
```

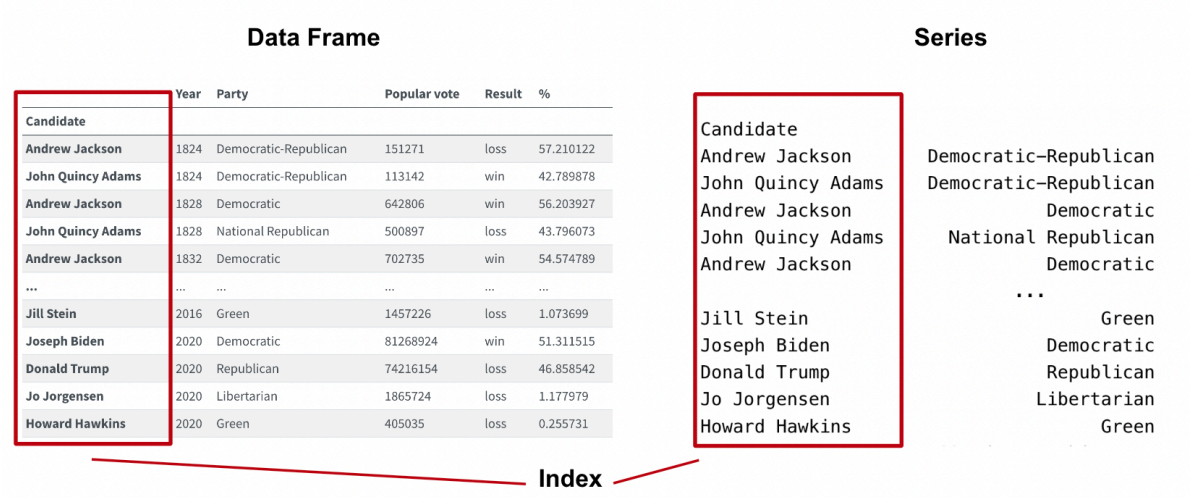
	A-column	B-column
r1	a1	b1
r2	a2	b2
r3	a3	b3

2.3.3 Indices

The major takeaway: we can think of a **DataFrame** as a collection of **Series** that all share the same **Index**.

On a more technical note, an Index doesn't have to be an integer, nor does it have to be unique. For example, we can set the index of the `elections` Dataframe to be the name of presedential candidates. Selecting a new Series from this modified DataFrame yields the following.

```
# This sets the index to the "Candidate" column
elections.set_index("Candidate", inplace=True)
```



Candidate	Year	Party	Popular vote	Result	%
Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
Andrew Jackson	1828	Democratic	642806	win	56.203927
John Quincy Adams	1828	National Republican	500897	loss	43.796073
Andrew Jackson	1832	Democratic	702735	win	54.574789
...	...	...	...	...	...
Jill Stein	2016	Green	1457226	loss	1.073699
Joseph Biden	2020	Democratic	81268924	win	51.311515
Donald Trump	2020	Republican	74216154	loss	46.858542
Jo Jorgensen	2020	Libertarian	1865724	loss	1.177979
Howard Hawkins	2020	Green	405035	loss	0.255731

Candidate	
Andrew Jackson	Democratic-Republican
John Quincy Adams	Democratic-Republican
Andrew Jackson	Democratic
John Quincy Adams	National Republican
Andrew Jackson	Democratic
...	...
Jill Stein	Green
Joseph Biden	Democratic
Donald Trump	Republican
Jo Jorgensen	Libertarian
Howard Hawkins	Green

To retrieve the indices of a DataFrame, simply use the `.index` attribute of the DataFrame class.

```
elections.index
```

```
Index(['Andrew Jackson', 'John Quincy Adams', 'Andrew Jackson',
       'John Quincy Adams', 'Andrew Jackson', 'Henry Clay', 'William Wirt',
       'Hugh Lawson White', 'Martin Van Buren', 'William Henry Harrison',
       ...
       'Darrell Castle', 'Donald Trump', 'Evan McMullin', 'Gary Johnson',
       'Hillary Clinton', 'Jill Stein', 'Joseph Biden', 'Donald Trump',
       'Jo Jorgensen', 'Howard Hawkins'],
      dtype='object', name='Candidate', length=182)
```

```
# This resets the index to be the default list of integers
elections.reset_index(inplace=True)
```

2.4 Slicing in DataFrames

Now that we've learned how to create DataFrames, let's dive deeper into their capabilities.

The API (application programming interface) for the DataFrame class is enormous. In this section, we'll discuss several methods of the DataFrame API that allow us to extract subsets of data.

The simplest way to manipulate a DataFrame is to extract a subset of rows and columns, known as **slicing**. We will do so with three primary methods of the DataFrame class:

1. `.loc`
2. `.iloc`
3. `[]`

2.4.1 Indexing with .loc

The `.loc` operator selects rows and columns in a DataFrame by their row and column label(s), respectively. The **row labels** (commonly referred to as the **indices**) are the bold text on the far *left* of a DataFrame, while the **column labels** are the column names found at the *top* of a DataFrame.

To grab data with `.loc`, we must specify the row and column label(s) where the data exists. The row labels are the first argument to the `.loc` function; the column labels are the second. For example, we can select the the row labeled `0` and the column labeled `Candidate` from the `elections` DataFrame.

```
elections.loc[0, 'Candidate']
```

```
'Andrew Jackson'
```

To select *multiple* rows and columns, we can use Python slice notation. Here, we select both the first four rows and columns.

```
elections.loc[0:3, 'Year':'Popular vote']
```

	Year	Party	Popular vote
0	1824	Democratic-Republican	151271
1	1824	Democratic-Republican	113142
2	1828	Democratic	642806
3	1828	National Republican	500897

Suppose that instead, we wanted *every* column value for the first four rows in the `elections` DataFrame. The shorthand `:` is useful for this.

```
elections.loc[0:3, :]
```

	Candidate	Year	Party	Popular vote	Result	%
0	Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
1	John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
2	Andrew Jackson	1828	Democratic	642806	win	56.203927
3	John Quincy Adams	1828	National Republican	500897	loss	43.796073

There are a couple of things we should note. Unlike conventional Python, Pandas allows us to slice string values (in our example, the column labels). Secondly, slicing with `.loc` is *inclusive*. Notice how our resulting DataFrame includes every row and column between and including the slice labels we specified.

Equivalently, we can use a list to obtain multiple rows and columns in our `elections` DataFrame.

```
elections.loc[[0, 1, 2, 3], ['Year', 'Candidate', 'Party', 'Popular vote']]
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Lastly, we can interchange list and slicing notation.

```
elections.loc[[0, 1, 2, 3], :]
```

	Candidate	Year	Party	Popular vote	Result	%
0	Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
1	John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
2	Andrew Jackson	1828	Democratic	642806	win	56.203927
3	John Quincy Adams	1828	National Republican	500897	loss	43.796073

2.4.2 Indexing with .iloc

Slicing with `.iloc` works similarily to `.loc`, although `.iloc` uses the integer positions of rows and columns rather the labels. The arguments to the `.iloc` function also behave similarly - single values, lists, indices, and any combination of these are permitted.

Let's begin reproducing our results from above. We'll begin by selecting for the first presedential candidate in our `elections` DataFrame:

```
# elections.loc[0, "Candidate"] - Previous approach
elections.iloc[0, 1]
```

```
1824
```

Notice how the first argument to both `.loc` and `.iloc` are the same. This is because the row with a label of 0 is conveniently in the 0^{th} (or first) position of the `elections` DataFrame. Generally, this is true of any DataFrame where the row labels are incremented in ascending order from 0.

However, when we select the first four rows and columns using `.iloc`, we notice something.

```
# elections.loc[0:3, 'Year':'Popular vote'] - Previous approach
elections.iloc[0:4, 0:4]
```

	Candidate	Year	Party	Popular vote
0	Andrew Jackson	1824	Democratic-Republican	151271
1	John Quincy Adams	1824	Democratic-Republican	113142
2	Andrew Jackson	1828	Democratic	642806
3	John Quincy Adams	1828	National Republican	500897

Slicing is no longer inclusive in `.iloc` - it's *exclusive*. This is one of Pandas syntatical subtleties; you'll get used to with practice.

List behavior works just as expected.

```
#elections.loc[[0, 1, 2, 3], ['Year', 'Candidate', 'Party', 'Popular vote']] - Previous Ap
elections.iloc[[0, 1, 2, 3], [0, 1, 2, 3]]
```

	Candidate	Year	Party	Popular vote
0	Andrew Jackson	1824	Democratic-Republican	151271
1	John Quincy Adams	1824	Democratic-Republican	113142
2	Andrew Jackson	1828	Democratic	642806
3	John Quincy Adams	1828	National Republican	500897

This discussion begs the question: when should we use `.loc` vs `.iloc`? In most cases, `.loc` is generally safer to use. You can imagine `.iloc` may return incorrect values when applied to a dataset where the ordering of data can change.

2.4.3 Indexing with []

The `[]` selection operator is the most baffling of all, yet the commonly used. It only takes a single argument, which may be one of the following:

1. A slice of row numbers
2. A list of column labels
3. A single column label

That is, `[]` is *context dependent*. Let's see some examples.

2.4.3.1 A slice of row numbers

Say we wanted the first four rows of our `elections` DataFrame.

```
elections[0:4]
```

	Candidate	Year	Party	Popular vote	Result	%
0	Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
1	John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
2	Andrew Jackson	1828	Democratic	642806	win	56.203927
3	John Quincy Adams	1828	National Republican	500897	loss	43.796073

2.4.3.2 A list of column labels

Suppose we now want the first four columns.

```
elections[["Year", "Candidate", "Party", "Popular vote"]]
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897
4	1832	Andrew Jackson	Democratic	702735
5	1832	Henry Clay	National Republican	484205
6	1832	William Wirt	Anti-Masonic	100715
7	1836	Hugh Lawson White	Whig	146109
8	1836	Martin Van Buren	Democratic	763291
9	1836	William Henry Harrison	Whig	550816
10	1840	Martin Van Buren	Democratic	1128854
11	1840	William Henry Harrison	Whig	1275583
12	1844	Henry Clay	Whig	1300004
13	1844	James Polk	Democratic	1339570
14	1848	Lewis Cass	Democratic	1223460
15	1848	Martin Van Buren	Free Soil	291501
16	1848	Zachary Taylor	Whig	1360235
17	1852	Franklin Pierce	Democratic	1605943
18	1852	John P. Hale	Free Soil	155210
19	1852	Winfield Scott	Whig	1386942
20	1856	James Buchanan	Democratic	1835140
21	1856	John C. Frémont	Republican	1342345
22	1856	Millard Fillmore	American	873053
23	1860	Abraham Lincoln	Republican	1855993
24	1860	John Bell	Constitutional Union	590901
25	1860	John C. Breckinridge	Southern Democratic	848019
26	1860	Stephen A. Douglas	Northern Democratic	1380202
27	1864	Abraham Lincoln	National Union	2211317
28	1864	George B. McClellan	Democratic	1812807
29	1868	Horatio Seymour	Democratic	2708744
30	1868	Ulysses Grant	Republican	3013790
31	1872	Horace Greeley	Liberal Republican	2834761
32	1872	Ulysses Grant	Republican	3597439
33	1876	Rutherford Hayes	Republican	4034142
34	1876	Samuel J. Tilden	Democratic	4288546
35	1880	James B. Weaver	Greenback	308649
36	1880	James Garfield	Republican	4453337
37	1880	Winfield Scott Hancock	Democratic	4444976
38	1884	Benjamin Butler	Anti-Monopoly	134294
39	1884	Grover Cleveland	Democratic	4914482
40	1884	James G. Blaine	Republican	4856905
41	1884	John St. John	Prohibition	147482
42	1888	Alson Streeter	Union Labor	146602
43	1888	Benjamin Harrison	Republican	5443633
44	1888	Clinton B. Fisk	Prohibition	249819
45	1888	Grover Cleveland	Democratic	5534488
46	1892	Benjamin Harrison	Republican	5176108
47	1892	Grover Cleveland	Democratic	5553898
48	1892	James B. Weaver	Populist	1041028
49	1892	John Bidwell	Prohibition	270879
50	1896	John M. Palmer	National Democratic	134645
51	1896	Joshua Levering	Prohibition	131312

2.4.3.3 A single column label

Lastly, if we only want the `Candidate` column.

```
elections["Candidate"]
```

	Candidate
0	Andrew Jackson
1	John Quincy Adams
2	Andrew Jackson
3	John Quincy Adams
4	Andrew Jackson
5	Henry Clay
6	William Wirt
7	Hugh Lawson White
8	Martin Van Buren
9	William Henry Harrison
10	Martin Van Buren
11	William Henry Harrison
12	Henry Clay
13	James Polk
14	Lewis Cass
15	Martin Van Buren
16	Zachary Taylor
17	Franklin Pierce
18	John P. Hale
19	Winfield Scott
20	James Buchanan
21	John C. Frémont
22	Millard Fillmore
23	Abraham Lincoln
24	John Bell
25	John C. Breckinridge
26	Stephen A. Douglas
27	Abraham Lincoln
28	George B. McClellan
29	Horatio Seymour
30	Ulysses Grant
31	Horace Greeley
32	Ulysses Grant
33	Rutherford Hayes
34	Samuel J. Tilden
35	James B. Weaver
36	James Garfield
37	Winfield Scott Hancock
38	Benjamin Butler
39	Grover Cleveland
40	James G. Blaine
41	John St. John
42	Alson Streeter
43	Benjamin Harrison
44	Clinton B. Fisk
45	Grover Cleveland
46	Benjamin Harrison
47	Grover Cleveland
48	James B. Weaver
49	John Bidwell
50	John M. Palmer
51	Joshua Levering

The output looks like a Series! In this course, we'll become very comfortable with `[]`, especially for selecting columns. In practice, `[]` is much more common than `.loc`.

2.5 Parting Note

The `pandas` library is enormous and contains many useful functions. Here is a link to documentation.

The introductory `pandas` lectures will cover important data structures and methods you should be fluent in. However, we want you to get familiar with the real world programming practice of …Googling! Answers to your questions can be found in documentation, Stack Overflow, etc.

With that, let's move on to Pandas II.

3 Pandas II

> **Note**
>
> - Build familiarity with advanced `pandas` syntax
> - Extract data from a DataFrame using conditional selection
> - Recognize situations where aggregation is useful and identify the correct technique for performing an aggregation

Last time, we introduced the `pandas` library as a toolkit for processing data. We learned the DataFrame and Series data structures, familiarized ourselves with the basic syntax for manipulating tabular data, and began writing our first lines of `pandas` code.

In this lecture, we'll start to dive into some advanced `pandas` syntax. You may find it helpful to follow along with a notebook of your own as we walk through these new pieces of code.

We'll start by loading the `babynames` dataset.

```
import pandas as pd
import numpy as np
import urllib.request
import os.path
import zipfile

data_url = "https://www.ssa.gov/oact/babynames/state/namesbystate.zip"
local_filename = "babynamesbystate.zip"
if not os.path.exists(local_filename): # if the data exists don't download again
    with urllib.request.urlopen(data_url) as resp, open(local_filename, 'wb') as f:
        f.write(resp.read())

zf = zipfile.ZipFile(local_filename, 'r')

ca_name = 'CA.TXT'
field_names = ['State', 'Sex', 'Year', 'Name', 'Count']
with zf.open(ca_name) as fh:
    babynames = pd.read_csv(fh, header=None, names=field_names)
```

```
babynames.head()
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

3.1 Conditional Selection

Conditional selection allows us to select a subset of rows in a DataFrame if they follow some specified condition.

To understand how to use conditional selection, we must look at another possible input of the `.loc` and `[]` methods – a boolean array, which is simply an array where each element is either `True` or `False`. This boolean array must have a length equal to the number of rows in the DataFrame. It will return all rows in the position of a corresponding True value in the array.

To see this in action, let's select all even-indexed rows in the first 10 rows of our DataFrame.

```
# Ask yourself: why is :9 is the correct slice to select the first 10 rows?
babynames_first_10_rows = babynames.loc[:9, :]

# Notice how we have exactly 10 elements in our boolean array argument
babynames_first_10_rows[[True, False, True, False, True, False, True, False, True, False]]
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
2	CA	F	1910	Dorothy	220
4	CA	F	1910	Frances	134
6	CA	F	1910	Evelyn	126
8	CA	F	1910	Virginia	101

We can perform a similar operation using `.loc`.

```
babynames_first_10_rows.loc[[True, False, True, False, True, False, True, False, True, Fal
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
2	CA	F	1910	Dorothy	220
4	CA	F	1910	Frances	134
6	CA	F	1910	Evelyn	126
8	CA	F	1910	Virginia	101

These techniques worked well in this example, but you can imagine how tedious it might be to list out `Trues` and `Falses` for every row in a larger DataFrame. To make things easier, we can instead provide a logical condition as an input to `.loc` or `[]` that returns a boolean array with the necessary length.

For example, to return all names associated with `F` sex:

```
# First, use a logical condition to generate a boolean array
logical_operator = (babynames["Sex"] == "F")

# Then, use this boolean array to filter the DataFrame
babynames[logical_operator].head()
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Here, `logical_operator` evaluates to a Series of boolean values with length 400762.

```
print("There are a total of {} values in 'logical_operator'".format(len(logical_operator))
```

```
There are a total of 400762 values in 'logical_operator'
```

Rows starting at row 0 and ending at row 235790 evaluate to `True` and are thus returned in the DataFrame.

```
print("The 0th item in this 'logical_operator' is: {}".format(logical_operator.iloc[0]))
print("The 235790th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2
print("The 235791th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2
```

```
The 0th item in this 'logical_operator' is: True
The 235790th item in this 'logical_operator' is: True
The 235791th item in this 'logical_operator' is: False
```

Passing a Series as an argument to `babynames[]` has the same affect as using a boolean array. In fact, the `[]` selection operator can take a boolean Series, array, and list as arguments. These three are used interchangeably thoughout the course.

We can also use `.loc` to achieve similar results.

```
babynames.loc[babynames["Sex"] == "F"].head()
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Boolean conditions can be combined using various operators that allow us to filter results by multiple conditions. Some examples include the `&` (and) operator and the `|` (or) operator.

Note: When combining multiple conditions with logical operators, be sure to surround each condition with a set of parenthesis `()`. If you forget, your code will throw an error.

For example, if we want to return data on all females born before the 21st century, we can write:

```
babynames[(babynames["Sex"] == "F") & (babynames["Year"] < 2000)].head()
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Boolean array selection is a useful tool, but can lead to overly verbose code for complex conditions. `Pandas` provide many alternatives:

```
(
    babynames[(babynames["Name"] == "Bella") |
```

```
                (babynames["Name"] == "Alex") |
                (babynames["Name"] == "Ani") |
                (babynames["Name"] == "Lisa")]
).head()
# Note: The parentheses surrounding the code make it possible to break the code on to mult
```

	State	Sex	Year	Name	Count
6289	CA	F	1923	Bella	5
7512	CA	F	1925	Bella	8
12368	CA	F	1932	Lisa	5
14741	CA	F	1936	Lisa	8
17084	CA	F	1939	Lisa	5

The `.isin` function can be used to filter dataframes. The method helps in selecting rows with having a particular (or multiple) value in a particular column.

```
names = ["Bella", "Alex", "Ani", "Lisa"]
babynames[babynames["Name"].isin(names)].head()
```

	State	Sex	Year	Name	Count
6289	CA	F	1923	Bella	5
7512	CA	F	1925	Bella	8
12368	CA	F	1932	Lisa	5
14741	CA	F	1936	Lisa	8
17084	CA	F	1939	Lisa	5

The function `str.startswith` can be used to define a filter based on string values in a `Series` object.

```
babynames[babynames["Name"].str.startswith("N")].head()
```

	State	Sex	Year	Name	Count
76	CA	F	1910	Norma	23
83	CA	F	1910	Nellie	20
127	CA	F	1910	Nina	11
198	CA	F	1910	Nora	6
310	CA	F	1911	Nellie	23

3.2 Handy Utility Functions

`pandas` contains an extensive library of functions that can help shorten the process of setting and getting information from its data structures. In the following section, we will give overviews of each of the main utility functions that will help us in Data 100.

- `Numpy` and built-in function support
- `.shape`
- `.size`
- `.describe()`
- `.sample()`
- `.value_counts()`
- `.unique()`
- `.sort_values()`

3.2.1 `Numpy`

```
bella_counts = babynames[babynames["Name"] == "Bella"]["Count"]
```

```
# Average number of babies named Bella each year
np.mean(bella_counts)
```

```
270.1860465116279
```

```
# Max number of babies named Bella born on a given year
max(bella_counts)
```

```
902
```

3.2.2 `.shape` and `.size`

`.shape` and `.size` are attributes of Series and DataFrames that measure the "amount" of data stored in the structure. Calling `.shape` returns a tuple containing the number of rows and columns present in the DataFrame or Series. `.size` is used to find the total number of elements in a structure, equivalent to the number of rows times the number of columns.

Many functions strictly require the dimensions of the arguments along certain axes to match. Calling these dimension-finding functions is much faster than counting all of the items by hand.

```
babynames.shape
```

```
(400762, 5)
```

```
babynames.size
```

```
2003810
```

3.2.3 `.describe()`

If many statistics are required from a DataFrame (minimum value, maximum value, mean value, etc.), then `.describe()` can be used to compute all of them at once.

```
babynames.describe()
```

	Year	Count
count	400762.000000	400762.000000
mean	1985.131287	79.953781
std	26.821004	295.414618
min	1910.000000	5.000000
25%	1968.000000	7.000000
50%	1991.000000	13.000000
75%	2007.000000	38.000000
max	2021.000000	8262.000000

A different set of statistics will be reported if `.describe()` is called on a Series.

```
babynames["Sex"].describe()
```

	Sex
count	400762
unique	2
top	F
freq	235791

3.2.4 `.sample()`

As we will see later in the semester, random processes are at the heart of many data science techniques (for example, train-test splits, bootstrapping, and cross-validation). `.sample()` lets us quickly select random entries (a row if called from a DataFrame, or a value if called from a Series).

```
babynames.sample()
```

	State	Sex	Year	Name	Count
308855	CA	M	1987	Katherine	5

```
babynames.sample(5).iloc[:, 2:]
```

	Year	Name	Count
388278	2017	Kamron	9
243628	1928	Gene	114
50863	1964	Sharleen	7
3598	1919	Mamie	22
53007	1965	Juan	5

```
babynames[babynames["Year"] == 2000].sample(4, replace = True).iloc[:, 2:]
```

	Year	Name	Count
150443	2000	Gretchen	18
150719	2000	Ahtziri	13
151703	2000	Cherry	7
151650	2000	Alaysha	7

3.2.5 `.value_counts()`

When we want to know the distribution of the items in a Series (for example, what items are most/least common), we use `.value-counts()` to get a breakdown of the unique *values* and their *counts*. In the example below, we can determine the name with the most years in which at least one person has taken that name by counting the number of times each name appears in the `"Name"` column of `babynames`.

```
babynames["Name"].value_counts().head()
```

	Name
Jean	221
Francis	219
Guadalupe	216
Jessie	215
Marion	213

3.2.6 `.unique()`

If we have a Series with many repeated values, then `.unique()` can be used to identify only the *unique* values. Here we can get a list of all the names in `babynames`.

Exercise: what function can we call on the Series below to get the number of unique names?

```
babynames["Name"].unique()
```

```
array(['Mary', 'Helen', 'Dorothy', ..., 'Zyire', 'Zylo', 'Zyrus'],
      dtype=object)
```

3.2.7 `.sort_values()`

Ordering a DataFrame can be useful for isolating extreme values. For example, the first 5 entries of a row sorted in descending order (that is, from highest to lowest) are the largest 5 values. `.sort_values` allows us to order a DataFrame or Series by a specified rule. For DataFrames, we must specify the column by which we want to compare the rows and the function will return such rows. We can choose to either receive the rows in `ascending` order (default) or `descending` order.

```
babynames.sort_values(by = "Count", ascending=False).head()
```

	State	Sex	Year	Name	Count
263272	CA	M	1956	Michael	8262
264297	CA	M	1957	Michael	8250
313644	CA	M	1990	Michael	8247
278109	CA	M	1969	Michael	8244
279405	CA	M	1970	Michael	8197

We do not need to explicitly specify the column used for sorting when calling `.value_counts()` on a Series. We can still specify the ordering paradigm – that is, whether values are sorted in ascending or descending order.

```
babynames["Name"].sort_values(ascending=True).head()
```

	Name
380256	Aadan
362255	Aadan
365374	Aadan
394460	Aadarsh
366561	Aaden

3.2.7.1 Sorting With a Custom Key

Using `.sort_values` can be useful in many situations, but it many not cover all use cases. This is because `pandas` automatically sorts values in order according to numeric value (for number data) or alphabetical order (for string data). The following code finds the top 5 most popular names in California in 2021.

```
# Sort names by count in year 2021
# Recall that `.head(5)` displays the first five rows in the DataFrame
babynames[babynames["Year"] == 2021].sort_values("Count", ascending=False).head()
```

	State	Sex	Year	Name	Count
397909	CA	M	2021	Noah	2591
397910	CA	M	2021	Liam	2469
232145	CA	F	2021	Olivia	2395
232146	CA	F	2021	Emma	2171
397911	CA	M	2021	Mateo	2108

This offers us a lot of functionality, but what if we need to sort by some other metric? For example, what if we wanted to find the longest names in the DataFrame?

We can do this by specifying the `key` parameter of `.sort_values`. The `key` parameter is assigned to a function of our choice. This function is then applied to each value in the specified column. `pandas` will, finally, sort the DataFrame by the values outputted by the function.

```
# Here, a lambda function is applied to find the length of each value, `x`, in the "Name"
babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False).head(5)
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

3.3 Adding and Removing Columns

To add a new column to a DataFrame, we use a syntax similar to that used when accessing an existing column. Specify the name of the new column by writing `dataframe["new_column"]`, then assign this to a Series or Array containing the values that will populate this column.

```
# Add a column named "name_lengths" that includes the length of each name
babynames["name_lengths"] = babynames["Name"].str.len()
babynames.head(5)
```

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	4
1	CA	F	1910	Helen	239	5
2	CA	F	1910	Dorothy	220	7
3	CA	F	1910	Margaret	163	8
4	CA	F	1910	Frances	134	7

```
# Sort by the temporary column
babynames = babynames.sort_values(by = "name_lengths", ascending=False)
babynames.head()
```

	State	Sex	Year	Name	Count	name_lengths
313143	CA	M	1989	Franciscojavier	6	15
333732	CA	M	1997	Ryanchristopher	5	15
330421	CA	M	1996	Franciscojavier	8	15
323615	CA	M	1993	Johnchristopher	5	15
310235	CA	M	1988	Franciscojavier	10	15

In the example above, we made use of an in-built function given to us by the `str` accessor for getting the length of names. Then we used `name_length` column to sort the dataframe. What if we had wanted to generate the values in our new column using a function of our own making?

We can do this using the Series `.map` method. `.map` takes in a function as input, and will apply this function to each value of a Series.

For example, say we wanted to find the number of occurrences of the sequence "dr" or "ea" in each name.

```
# First, define a function to count the number of times "dr" or "ea" appear in each name
def dr_ea_count(string):
    return string.count("dr") + string.count("ea")

# Then, use `map` to apply `dr_ea_count` to each name in the "Name" column
babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count)

# Sort the DataFrame by the new "dr_ea_count" column so we can see our handiwork
babynames.sort_values(by = "dr_ea_count", ascending = False).head(5)
```

	State	Sex	Year	Name	Count	name_lengths	dr_ea_count
101969	CA	F	1986	Deandrea	6	8	3
304390	CA	M	1985	Deandrea	6	8	3
131022	CA	F	1994	Leandrea	5	8	3
115950	CA	F	1990	Deandrea	5	8	3
108723	CA	F	1988	Deandrea	5	8	3

If we want to remove a column or row of a DataFrame, we can call the `.drop` method. Use the `axis` parameter to specify whether a column or row should be dropped. Unless otherwise specified, `pandas` will assume that we are dropping a row by default.

```
# Drop our "dr_ea_count" and "length" columns from the DataFrame
babynames = babynames.drop(["dr_ea_count", "name_lengths"], axis="columns")
babynames.head(5)
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

Notice that we reassigned `babynames` to the result of `babynames.drop(...)`. This is a subtle, but important point: `pandas` table operations **do not occur in-place**. Calling `dataframe.drop(...)` will output a *copy* of `dataframe` with the row/column of interest removed, without modifying the original `dataframe` table.

In other words, if we simply call:

```
# This creates a copy of `babynames` and removes the row with label 3...
babynames.drop(3, axis="rows")

# ...but the original `babynames` is unchanged!
# Notice that the row with label 3 is still present
babynames.head(5)
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

3.4 Aggregating Data with GroupBy

Up until this point, we have been working with individual rows of DataFrames. As data scientists, we often wish to investigate trends across a larger *subset* of our data. For example, we may want to compute some summary statistic (the mean, median, sum, etc.) for a group of rows in our DataFrame. To do this, we'll use `pandas` `GroupBy` objects.

Let's say we wanted to aggregate all rows in `babynames` for a given year.

```
babynames.groupby("Year")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fc2d30fb970>
```

What does this strange output mean? Calling `.groupby` has generated a `GroupBy` object. You can imagine this as a set of "mini" sub-DataFrames, where each subframe contains all of the rows from `babynames` that correspond to a particular year.

The diagram below shows a simplified view of `babynames` to help illustrate this idea.

We can't work with a `GroupBy` object directly – that is why you saw that strange output earlier, rather than a standard view of a DataFrame. To actually manipulate values within these "mini" DataFrames, we'll need to call an *aggregation method*. This is a method that tells `pandas` how to aggregate the values within the `GroupBy` object. Once the aggregation is applied, `pandas` will return a normal (now grouped) DataFrame.

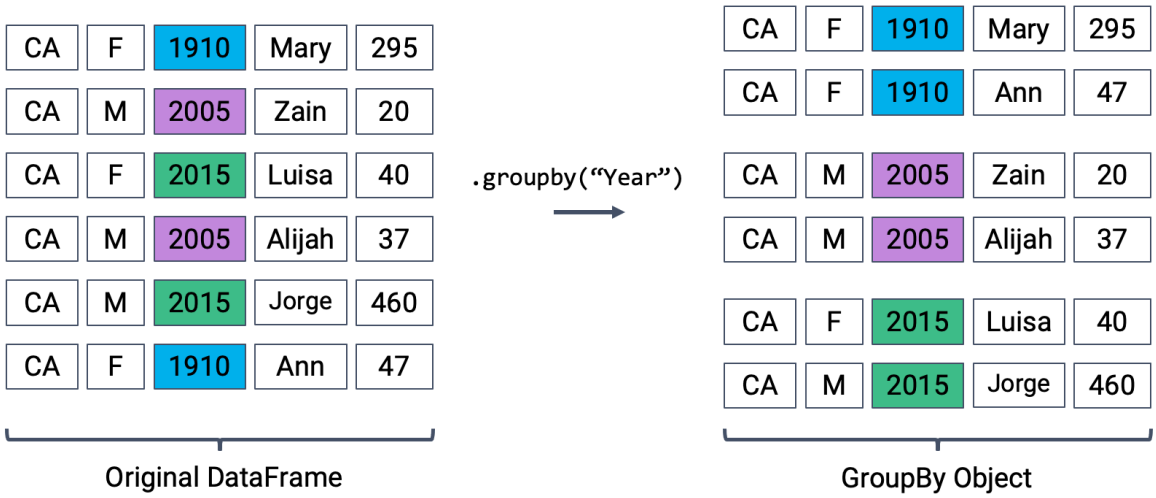


Figure 3.1: Creating a GroupBy object

The first aggregation method we'll consider is `.agg`. The `.agg` method takes in a function as its argument; this function is then applied to each column of a "mini" grouped DataFrame. We end up with a new DataFrame with one aggregated row per subframe. Let's see this in action by finding the `sum` of all counts for each year in `babynames` – this is equivalent to finding the number of babies born in each year.

```
babynames.groupby("Year").agg(sum).head(5)
```

Year	Count
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

We can relate this back to the diagram we used above. Remember that the diagram uses a simplified version of `babynames`, which is why we see smaller values for the summed counts.

Calling `.agg` has condensed each subframe back into a single row. This gives us our final output: a DataFrame that is now indexed by `"Year"`, with a single row for each unique year in the original `babynames` DataFrame.

You may be wondering: where did the `"State"`, `"Sex"`, and `"Name"` columns go? Logically, it doesn't make sense to `sum` the string data in these columns (how would we add "Mary" + "Ann"?). Because of this, `pandas` will simply omit these columns when it performs the

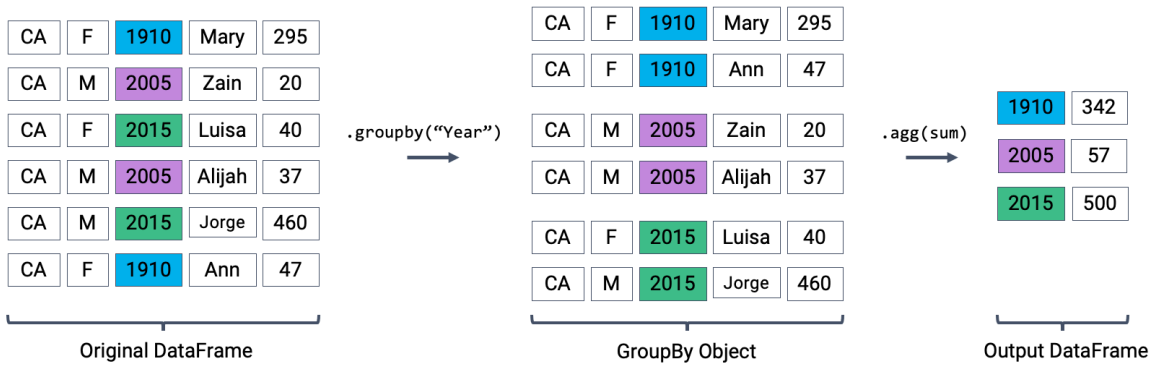


Figure 3.2: Performing an aggregation

aggregation on the DataFrame. Since this happens implicitly, without the user specifying that these columns should be ignored, it's easy to run into troubling situations where columns are removed without the programmer noticing. It is better coding practice to select *only* the columns we care about before performing the aggregation.

```
# Same result, but now we explicitly tell Pandas to only consider the "Count" column when
babynames.groupby("Year")[["Count"]].agg(sum).head(5)
```

Year	Count
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

3.4.1 Parting note

Manipulating `DataFrames` is a skill that is not mastered in just one day. Due to the flexibility of `pandas`, there are many different ways to get from a point A to a point B. We recommend trying multiple different ways to solve the same problem to gain even more practice and reach that point of mastery sooner.

Next, we will start digging deeper into the mechanics behind grouping data.

4 Pandas III

> **Note**
>
> - Perform advanced aggregation using `.groupby()`
> - Use the `pd.pivot_table` method to contruct a pivot table
> - Perform simple merges between DataFrames using `pd.merge()`

4.1 More on `agg()` Function

Last time, we introduced the concept of aggregating data – we familiarized ourselves with `GroupBy` objects and used them as tools to consolidate and summarize a DataFrame. In this lecture, we will explore some advanced `.groupby` methods to show just how powerful of a resource they can be for understanding our data. We will also introduce other techniques for data aggregation to provide flexibility in how we manipulate our tables.

4.2 `GroupBy()`, Continued

As we learned last lecture, a `groupby` operation involves some combination of **splitting a DataFrame into grouped subframes**, **applying a function**, and **combining the results**.

For some arbitrary DataFrame `df` below, the code `df.groupby("year").agg(sum)` does the following:

- Organizes all rows with the same year into a subframe for that year.
- Creates a new DataFrame with one row representing each subframe year.
- Combines all integer rows in each subframe using the `sum` function.

4.2.1 Aggregation with `lambda` Functions

Throughout this note, we'll work with the `elections` DataFrame.

```
import pandas as pd

elections = pd.read_csv("data/elections.csv")
elections.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789

What if we wish to aggregate our DataFrame using a non-standard function – for example, a function of our own design? We can do so by combining `.agg` with `lambda` expressions.

Let's first consider a puzzle to jog our memory. We will attempt to find the `Candidate` from each `Party` with the highest `%` of votes.

A naive approach may be to group by the `Party` column and aggregate by the maximum.

```
elections.groupby("Party").agg(max).head(10)
```

Party	Year	Candidate	Popular vote	Result	%
American	1976	Thomas J. Anderson	873053	loss	21.554001
American Independent	1976	Lester Maddox	9901118	loss	13.571218
Anti-Masonic	1832	William Wirt	100715	loss	7.821583
Anti-Monopoly	1884	Benjamin Butler	134294	loss	1.335838
Citizens	1980	Barry Commoner	233052	loss	0.270182
Communist	1932	William Z. Foster	103307	loss	0.261069
Constitution	2016	Michael Peroutka	203091	loss	0.152398
Constitutional Union	1860	John Bell	590901	loss	12.639283
Democratic	2020	Woodrow Wilson	81268924	win	61.344703
Democratic-Republican	1824	John Quincy Adams	151271	win	57.210122

This approach is clearly wrong – the DataFrame claims that Woodrow Wilson won the presidency in 2020.

Why is this happening? Here, the `max` aggregation function is taken over every column *independently*. Among Democrats, `max` is computing:

- The most recent `Year` a Democratic candidate ran for president (2020)
- The `Candidate` with the alphabetically "largest" name ("Woodrow Wilson")

- The `Result` with the alphabetically "largest" outcome ("win")

Instead, let's try a different approach. We will:

1. Sort the DataFrame so that rows are in descending order of `%`
2. Group by `Party` and select the first row of each groupby object

While it may seem unintuitive, sorting `elections` by descending order of `%` is extremely helpful. If we then group by `Party`, the first row of each groupby object will contain information about the `Candidate` with the highest voter `%`.

```
elections_sorted_by_percent = elections.sort_values("%", ascending=False)
elections_sorted_by_percent.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
114	1964	Lyndon Johnson	Democratic	43127041	win	61.344703
91	1936	Franklin Roosevelt	Democratic	27752648	win	60.978107
120	1972	Richard Nixon	Republican	47168710	win	60.907806
79	1920	Warren Harding	Republican	16144093	win	60.574501
133	1984	Ronald Reagan	Republican	54455472	win	59.023326

```
elections_sorted_by_percent.groupby("Party").agg(lambda x : x.iloc[0]).head(10)

# Equivalent to the below code
# elections_sorted_by_percent.groupby("Party").agg('first').head(10)
```

Party	Year	Candidate	Popular vote	Result	%
American	1856	Millard Fillmore	873053	loss	21.554001
American Independent	1968	George Wallace	9901118	loss	13.571218
Anti-Masonic	1832	William Wirt	100715	loss	7.821583
Anti-Monopoly	1884	Benjamin Butler	134294	loss	1.335838
Citizens	1980	Barry Commoner	233052	loss	0.270182
Communist	1932	William Z. Foster	103307	loss	0.261069
Constitution	2008	Chuck Baldwin	199750	loss	0.152398
Constitutional Union	1860	John Bell	590901	loss	12.639283
Democratic	1964	Lyndon Johnson	43127041	win	61.344703
Democratic-Republican	1824	Andrew Jackson	151271	loss	57.210122

Notice how our code correctly determines that Lyndon Johnson from the Democratic Party has the highest voter `%`.

More generally, `lambda` functions are used to design custom aggregation functions that aren't pre-defined by Python. The input parameter `x` to the `lambda` function is a `GroupBy` object. Therefore, it should make sense why `lambda x : x.iloc[0]` selects the first row in each groupby object.

In fact, there's a few different ways to approach this problem. Each approach has different tradeoffs in terms of readability, performance, memory consumption, complexity, etc. We've given a few examples below.

Note: Understanding these alternative solutions is not required. They are given to demonstrate the vast number of problem-solving approaches in `pandas`.

```
# Using the idxmax function
best_per_party = elections.loc[elections.groupby('Party')['%'].idxmax()]
best_per_party.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
22	1856	Millard Fillmore	American	873053	loss	21.554001
115	1968	George Wallace	American Independent	9901118	loss	13.571218
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
127	1980	Barry Commoner	Citizens	233052	loss	0.270182

```
# Using the .drop_duplicates function
best_per_party2 = elections.sort_values('%').drop_duplicates(['Party'], keep='last')
best_per_party2.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
148	1996	John Hagelin	Natural Law	113670	loss	0.118219
164	2008	Chuck Baldwin	Constitution	199750	loss	0.152398
110	1956	T. Coleman Andrews	States' Rights	107929	loss	0.174883
147	1996	Howard Phillips	Taxpayers	184656	loss	0.192045
136	1988	Lenora Fulani	New Alliance	217221	loss	0.237804

4.2.2 Other `GroupBy` Features

There are many aggregation methods we can use with `.agg`. Some useful options are:

- `.max`: creates a new DataFrame with the maximum value of each group
- `.mean`: creates a new DataFrame with the mean value of each group
- `.size`: creates a new Series with the number of entries in each group

In fact, these (and other) aggregation functions are so common that `pandas` allows for writing shorthand. Instead of explicitly stating the use of `.agg`, we can call the function directly on the `GroupBy` object.

For example, the following are equivalent:

- `elections.groupby("Candidate").agg(mean)`
- `elections.groupby("Candidate").mean()`

4.2.3 The `groupby.filter()` function

Another common use for `GroupBy` objects is to filter data by group.

`groupby.filter` takes an argument f, where f is a function that:

- Takes a `GroupBy` object as input
- Returns a single `True` or `False` for the entire subframe

`GroupBy` objects that correspond to `True` are returned in the final result, whereas those with a `False` value are not. Importantly, `groupby.filter` is different from `groupby.agg` in that the *entire* subframe is returned in the final DataFrame, not just a single row.

To illustrate how this happens, consider the following `.filter` function applied on some arbitrary data. Say we want to identify "tight" election years – that is, we want to find all rows that correspond to elections years where all candidates in that year won a similar portion of the total vote. Specifically, let's find all rows corresponding to a year where no candidate won more than 45% of the total vote.

An equivalent way of framing this goal is to say:

- Find the years where the maximum `%` in that year is less than 45%
- Return all DataFrame rows that correspond to these years

For each year, we need to find the maximum `%` among *all* rows for that year. If this maximum `%` is lower than 45%, we will tell `pandas` to keep all rows corresponding to that year.

```
elections.groupby("Year").filter(lambda sf: sf["%"].max() < 45).head(9)
```

	Year	Candidate	Party	Popular vote	Result	%
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
66	1912	Eugene V. Debs	Socialist	901551	loss	6.004354
67	1912	Eugene W. Chafin	Prohibition	208156	loss	1.386325
68	1912	Theodore Roosevelt	Progressive	4122721	loss	27.457433
69	1912	William Taft	Republican	3486242	loss	23.218466
70	1912	Woodrow Wilson	Democratic	6296284	win	41.933422

What's going on here? In this example, we've defined our filtering function, f, to be `lambda sf: sf["%"].max() < 45`. This filtering function will find the maximum `"%"` value among all entries in the grouped subframe, which we call `sf`. If the maximum value is less than 45, then the filter function will return `True` and all rows in that grouped subframe will appear in the final output DataFrame.

Examine the DataFrame above. Notice how, in this preview of the first 9 rows, all entries from the years 1860 and 1912 appear. This means that in 1860 and 1912, no candidate in that year won more than 45% of the total vote.

You may ask: how is the `groupby.filter` procedure different to the boolean filtering we've seen previously? Boolean filtering considers *individual* rows when applying a boolean condition. For example, the code `elections[elections["%"] < 45]` will check the `"%"` value of every single row in `elections`; if it is less than 45, then that row will be kept in the output. `groupby.filter`, in contrast, applies a boolean condition *across* all rows in a group. If not all rows in that group satisfy the condition specified by the filter, the entire group will be discarded in the output.

4.3 Aggregating Data with Pivot Tables

We know now that `.groupby` gives us the ability to group and aggregate data across our DataFrame. The examples above formed groups using just one column in the DataFrame. It's possible to group by multiple columns at once by passing in a list of columns names to `.groupby`.

Let's consider the `babynames` dataset from last lecture. In this problem, we will find the total number of baby names associated with each sex for each year. To do this, we'll group by *both* the `"Year"` and `"Sex"` columns.

```
import urllib.request
import os.path

# Download data from the web directly
data_url = "https://www.ssa.gov/oact/babynames/names.zip"
local_filename = "data/babynames.zip"
if not os.path.exists(local_filename): # if the data exists don't download again
    with urllib.request.urlopen(data_url) as resp, open(local_filename, 'wb') as f:
        f.write(resp.read())


# Load data without unzipping the file
import zipfile
babynames = []
with zipfile.ZipFile(local_filename, "r") as zf:
    data_files = [f for f in zf.filelist if f.filename[-3:] == "txt"]
    def extract_year_from_filename(fn):
        return int(fn[3:7])
    for f in data_files:
        year = extract_year_from_filename(f.filename)
        with zf.open(f) as fp:
            df = pd.read_csv(fp, names=["Name", "Sex", "Count"])
            df["Year"] = year
            babynames.append(df)
babynames = pd.concat(babynames)
```

```
babynames.head()
```

	Name	Sex	Count	Year
0	Mary	F	7065	1880
1	Anna	F	2604	1880
2	Emma	F	2003	1880
3	Elizabeth	F	1939	1880
4	Minnie	F	1746	1880

```
# Find the total number of baby names associated with each sex for each year in the data
babynames.groupby(["Year", "Sex"])[["Count"]].agg(sum).head(6)
```

Year	Sex	Count
1880	F	90994
	M	110490
1881	F	91953
	M	100737
1882	F	107847
	M	113686

Notice that both `"Year"` and `"Sex"` serve as the index of the DataFrame (they are both rendered in bold). We've created a *multindex* where two different index values, the year and sex, are used to uniquely identify each row.

This isn't the most intuitive way of representing this data – and, because multindexes have multiple dimensions in their index, they can often be difficult to use.

Another strategy to aggregate across two columns is to create a pivot table. You saw these back in Data 8. One set of values is used to create the index of the table; another set is used to define the column names. The values contained in each cell of the table correspond to the aggregated data for each index-column pair.

The best way to understand pivot tables is to see one in action. Let's return to our original goal of summing the total number of names associated with each combination of year and sex. We'll call the `pandas` `.pivot_table` method to create a new table.

```
# The `pivot_table` method is used to generate a Pandas pivot table
import numpy as np
babynames.pivot_table(index = "Year", columns = "Sex", values = "Count", aggfunc = np.sum)
```

Sex Year	F	M
1880	90994	110490
1881	91953	100737
1882	107847	113686
1883	112319	104625
1884	129019	114442

Looks a lot better! Now, our DataFrame is structured with clear index-column combinations. Each entry in the pivot table represents the summed count of names for a given combination of `"Year"` and `"Sex"`.

Let's take a closer look at the code implemented above.

- `index = "Year"` specifies the column name in the original DataFrame that should be used as the index of the pivot table
- `columns = "Sex"` specifies the column name in the original DataFrame that should be used to generate the columns of the pivot table
- `values = "Count"` indicates what values from the original DataFrame should be used to populate the entry for each index-column combination
- `aggfunc = np.sum` tells `pandas` what function to use when aggregating the data specified by `values`. Here, we are summing the name counts for each pair of `"Year"` and `"Sex"`

We can even include multiple values in the index or columns of our pivot tables.

```
babynames_pivot = babynames.pivot_table(
    index="Year",     # the rows (turned into index)
    columns="Sex",    # the column values
    values=["Count", "Name"],
    aggfunc=max,   # group operation
)
babynames_pivot.head(6)
```

	Count		Name	
Sex	F	M	F	M
Year				
1880	7065	9655	Zula	Zeke
1881	6919	8769	Zula	Zeb
1882	8148	9557	Zula	Zed
1883	8012	8894	Zula	Zeno
1884	9217	9388	Zula	Zollie
1885	9128	8756	Zula	Zollie

4.4 Joining Tables

When working on data science projects, we're unlikely to have absolutely all the data we want contained in a single DataFrame – a real-world data scientist needs to grapple with data coming from multiple sources. If we have access to multiple datasets with related information, we can join two or more tables into a single DataFrame.

To put this into practice, we'll revisit the `elections` dataset.

```
elections.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789

Say we want to understand the 2020 popularity of the names of each presidential candidate. To do this, we'll need the combined data of `babynames` *and* `elections`.

We'll start by creating a new column containing the first name of each presidential candidate. This will help us join each name in `elections` to the corresponding name data in `babynames`.

```
# This `str` operation splits each candidate's full name at each
# blank space, then takes just the candidiate's first name
elections["First Name"] = elections["Candidate"].str.split().str[0]
elections.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%	First Name
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878	John
2	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073	John
4	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew

```
# Here, we'll only consider `babynames` data from 2020
babynames_2020 = babynames[babynames["Year"]==2020]
babynames_2020.head()
```

	Name	Sex	Count	Year
0	Olivia	F	17641	2020
1	Emma	F	15656	2020
2	Ava	F	13160	2020
3	Charlotte	F	13065	2020
4	Sophia	F	13036	2020

Now, we're ready to join the two tables. `pd.merge` is the `pandas` method used to join DataFrames together. The `left` and `right` parameters are used to specify the DataFrames to be joined. The `left_on` and `right_on` parameters are assigned to the string names of the columns to be used when performing the join. These two `on` parameters tell `pandas` what values should act as pairing keys to determine which rows to merge across the DataFrames. We'll talk more about this idea of a pairing key next lecture.

```
merged = pd.merge(left = elections, right = babynames_2020, \
                  left_on = "First Name", right_on = "Name")
merged.head()
# Notice that pandas automatically specifies `Year_x` and `Year_y`
# when both merged DataFrames have the same column name to avoid confusion
```

	Year_x	Candidate	Party	Popular vote	Result	%	First Name	N
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	A
1	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	A
2	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	A
3	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	A
4	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew	A

5 Data Cleaning and EDA

> **Note**
>
> - Recognize common file formats
> - Categorize data by its variable type
> - Build awareness of issues with data faithfulness and develop targeted solutions

In the past few lectures, we've learned that `pandas` is a toolkit to restructure, modify, and explore a dataset. What we haven't yet touched on is *how* to make these data transformation decisions. When we receive a new set of data from the "real world," how do we know what processing we should do to convert this data into a usable form?

Data cleaning, also called **data wrangling**, is the process of transforming raw data to facilitate subsequent analysis. It is often used to address issues like:

- Unclear structure or formatting
- Missing or corrupted values
- Unit conversions
- …and so on

Exploratory Data Analysis (EDA) is the process of understanding a new dataset. It is an open-ended, informal analysis that involves familiarizing ourselves with the variables present in the data, discovering potential hypotheses, and identifying potential issues with the data. This last point can often motivate further data cleaning to address any problems with the dataset's format; because of this, EDA and data cleaning are often thought of as an "infinite loop," with each process driving the other.

In this lecture, we will consider the key properties of data to consider when performing data cleaning and EDA. In doing so, we'll develop a "checklist" of sorts for you to consider when approaching a new dataset. Throughout this process, we'll build a deeper understanding of this early (but very important!) stage of the data science lifecycle.

5.1 Structure

5.1.1 File Format

In the past two `pandas` lectures, we briefly touched on the idea of file format: the way data is encoded in a file for storage. Specifically, our `elections` and `babynames` datasets were stored and loaded as CSVs:

```
import pandas as pd
pd.read_csv("data/elections.csv").head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789

CSVs, which stand for **Comma-Separated Values**, are a common tabular data format. To better understand the properties of a CSV, let's take a look at the first few rows of the raw data file to see what it looks like before being loaded into a DataFrame.

```
Year,Candidate,Party,Popular vote,Result,%

1824,Andrew Jackson,Democratic-Republican,151271,loss,57.21012204

1824,John Quincy Adams,Democratic-Republican,113142,win,42.78987796

1828,Andrew Jackson,Democratic,642806,win,56.20392707
```

Each row, or **record**, in the data is delimited by a newline. Each column, or **field**, in the data is delimited by a comma (hence, comma-separated!).

Another common file type is the **TSV (Tab-Separated Values)**. In a TSV, records are still delimited by a newline, while fields are delimited by `\t` tab character. A TSV can be loaded into `pandas` using `pd.read_csv()` with the `delimiter` parameter: `pd.read_csv("file_name.tsv", delimiter="\t")`. A raw TSV file is shown below.

```
Year   Candidate   Party   Popular vote    Result  %

1824    Andrew Jackson  Democratic-Republican   151271  loss    57.21012204
```

```
1824    John Quincy Adams   Democratic-Republican   113142  win 42.78987796

1828    Andrew Jackson  Democratic  642806  win 56.20392707
```

JSON (JavaScript Object Notation) files behave similarly to Python dictionaries. They can be loaded into `pandas` using `pd.read_json`. A raw JSON is shown below.

```
[

 {

   "Year": 1824,

   "Candidate": "Andrew Jackson",

   "Party": "Democratic-Republican",

   "Popular vote": 151271,

   "Result": "loss",

   "%": 57.21012204

 },
```

5.1.2 Variable Types

After loading data into a file, it's a good idea to take the time to understand what pieces of information are encoded in the dataset. In particular, we want to identify what variable types are present in our data. Broadly speaking, we can categorize variables into one of two overarching types.

Quantitative variables describe some numeric quantity or amount. We can sub-divide quantitative data into:

- **Continuous quantitative variables**: numeric data that can be measured on a continuous scale to arbitrary precision. Continuous variables do not have a strict set of possible values – they can be recorded to any number of decimal places. For example, weights, GPA, or CO2 concentrations
- **Discrete quantitative variables**: numeric data that can only take on a finite set of possible values. For example, someone's age or number of siblings.

Qualitative variables, also known as **categorical variables**, describe data that isn't measuring some quantity or amount. The sub-categories of categorical data are:

- **Ordinal qualitative variables**: categories with ordered levels. Specifically, ordinal variables are those where the difference between levels has no consistent, quantifiable meaning. For example, a Yelp rating or set of income brackets.
- **Nominal qualitative variables**: categories with no specific order. For example, someone's political affiliation or Cal ID number.

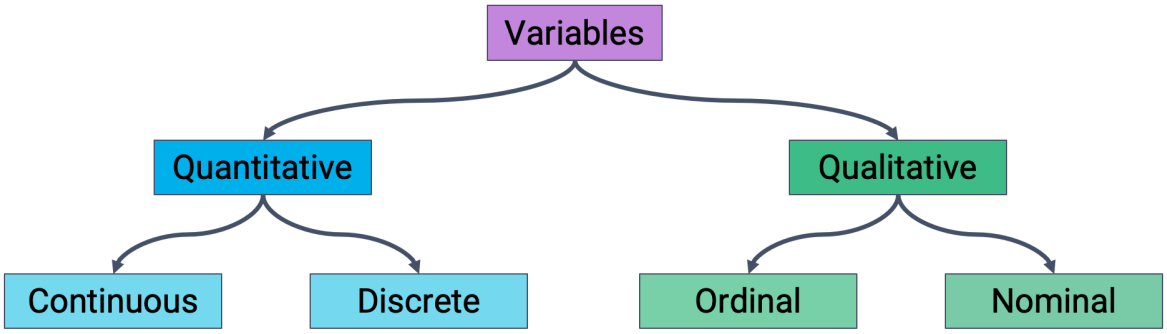


Figure 5.1: Classification of variable types

5.1.3 Primary and Foreign Keys

Last time, we introduced `.merge` as the `pandas` method for joining multiple DataFrames together. In our discussion of joins, we touched on the idea of using a "key" to determine what rows should be merged from each table. Let's take a moment to examine this idea more closely.

The **primary key** is the column or set of columns in a table that determine the values of the remaining columns. It can be thought of as the unique identifier for each individual row in the table. For example, a table of Data 100 students might use each student's Cal ID as the primary key.

	Cal ID	Name	Major
0	3034619471	Oski	Data Science
1	3035619472	Ollie	Computer Science
2	3025619473	Orrie	Data Science
3	3046789372	Ollie	Economics

The **foreign key** is the column or set of columns in a table that reference primary keys in other tables. Knowing a dataset's foreign keys can be useful when assigning the `left_on` and

`right_on` parameters of `.merge`. In the table of office hour tickets below, `"Cal ID"` is a foreign key referencing the previous table.

	OH Request	Cal ID	Question
0	1	3034619471	HW 2 Q1
1	2	3035619472	HW 2 Q3
2	3	3025619473	Lab 3 Q4
3	4	3035619472	HW 2 Q7

5.2 Granularity, Scope, and Temporality

After understanding the structure of the dataset, the next task is to determine what exactly the data represents. We'll do so by considering the data's granularity, scope, and temporality.

The **granularity** of a dataset is what a single row represents. You can also think of it as the level of detail included in the data. To determine the data's granularity, ask: what does each row in the dataset represent? Fine-grained data contains a high level of detail, with a single row representing a small individual unit. For example, each record may represent one person. Coarse-grained data is encoded such that a single row represents a large individual unit – for example, each record may represent a group of people.

The **scope** of a dataset is the subset of the population covered by the data. If we were investigating student performance in Data Science courses, a dataset with narrow scope might encompass all students enrolled in Data 100; a dataset with expansive scope might encompass all students in California.

The **temporality** of a dataset describes the time period over which the data was collected. To fully understand the temporality of the data, it may be necessary to standardize timezones or inspect recurring time-based trends in the data (Do patterns recur in 24-hour patterns? Over the course of a month? Seasonally?).

5.3 Faithfulness

At this stage in our data cleaning and EDA workflow, we've achieved quite a lot: we've identified how our data is structured, come to terms with what information it encodes, and gained insight as to how it was generated. Throughout this process, we should always recall the original intent of our work in Data Science – to use data to better understand and model the real world. To achieve this goal, we need to ensure that the data we use is faithful to reality; that is, that our data accurately captures the "real world."

Data used in research or industry is often "messy" – there may be errors or inaccuracies that impact the faithfulness of the dataset. Signs that data may not be faithful include:

- Unrealistic or "incorrect" values, such as negative counts, locations that don't exist, or dates set in the future
- Violations of obvious dependencies, like an age that does not match a birthday
- Clear signs that data was entered by hand, which can lead to spelling errors or fields that are incorrectly shifted
- Signs of data falsification, such as fake email addresses or repeated use of the same names
- Duplicated records or fields containing the same information

A common issue encountered with real-world datasets is that of missing data. One strategy to resolve this is to simply drop any records with missing values from the dataset. This does, however, introduce the risk of inducing biases – it is possible that the missing or corrupt records may be systemically related to some feature of interest in the data.

Another method to address missing data is to perform **imputation**: infer the missing values using other data available in the dataset. There is a wide variety of imputation techniques that can be implemented; some of the most common are listed below.

- Average imputation: replace missing values with the average value for that field
- Hot deck imputation: replace missing values with some random value
- Regression imputation: develop a model to predict missing values
- Multiple imputation: replace missing values with multiple random values

Regardless of the strategy used to deal with missing data, we should think carefully about *why* particular records or fields may be missing – this can help inform whether or not the absence of these values is signficant in some meaningful way.

6 EDA Demo: Tuberculosis in the United States

Now, let's follow this data-cleaning and EDA workflow to see what can we say about the presence of Tuberculosis in the United States!

We will examine the data included in the original CDC article published in 2021.

6.1 CSVs and Field Names

Suppose Table 1 was saved as a CSV file located in `data/cdc_tuberculosis.csv`.

We can then explore the CSV (which is a text file, and does not contain binary-encoded data) in many ways: 1. Using a text editor like emacs, vim, VSCode, etc. 2. Opening the CSV directly in DataHub (read-only), Excel, Google Sheets, etc. 3. The Python file object 4. pandas, using `pd.read_csv()`

1, 2. Let's start with the first two so we really solidify the idea of a CSV as **rectangular data (i.e., tabular data) stored as comma-separated values**.

3. Next, let's try using the Python file object. Let's check out the first three lines:

```
with open("data/cdc_tuberculosis.csv", "r") as f:
    i = 0
    for row in f:
        print(row)
        i += 1
        if i > 3:
            break
```

```
,No. of TB cases,,,TB incidence,,

U.S. jurisdiction,2019,2020,2021,2019,2020,2021

Total,"8,900","7,173","7,860",2.71,2.16,2.37
```

```
Alabama,87,72,92,1.77,1.43,1.83
```

Whoa, why are there blank lines interspaced between the lines of the CSV?

You may recall that all line breaks in text files are encoded as the special newline character `\n`. Python's `print()` prints each string (including the newline), and an additional newline on top of that.

If you're curious, we can use the `repr()` function to return the raw string with all special characters:

```
with open("data/cdc_tuberculosis.csv", "r") as f:
    i = 0
    for row in f:
        print(repr(row)) # print raw strings
        i += 1
        if i > 3:
            break
```

```
',No. of TB cases,,,TB incidence,,\n'
'U.S. jurisdiction,2019,2020,2021,2019,2020,2021\n'
'Total,"8,900","7,173","7,860",2.71,2.16,2.37\n'
'Alabama,87,72,92,1.77,1.43,1.83\n'
```

4. Finally, let's see the tried-and-true Data 100 approach: pandas.

```
tb_df = pd.read_csv("data/cdc_tuberculosis.csv")
tb_df.head()
```

	Unnamed: 0	No. of TB cases	Unnamed: 2	Unnamed: 3	TB incidence	Unnamed: 5	Unname
0	U.S. jurisdiction	2019	2020	2021	2019.00	2020.00	202
1	Total	8,900	7,173	7,860	2.71	2.16	
2	Alabama	87	72	92	1.77	1.43	
3	Alaska	58	58	58	7.91	7.92	
4	Arizona	183	136	129	2.51	1.89	

Wait, what's up with the "Unnamed" column names? And the first row, for that matter?

Congratulations – you're ready to wrangle your data. Because of how things are stored, we'll need to clean the data a bit to name our columns better.

A reasonable first step is to identify the row with the right header. The `pd.read_csv()` function (documentation) has the convenient `header` parameter:

```
tb_df = pd.read_csv("data/cdc_tuberculosis.csv", header=1) # row index
tb_df.head(5)
```

	U.S. jurisdiction	2019	2020	2021	2019.1	2020.1	2021.1
0	Total	8,900	7,173	7,860	2.71	2.16	2.37
1	Alabama	87	72	92	1.77	1.43	1.83
2	Alaska	58	58	58	7.91	7.92	7.92
3	Arizona	183	136	129	2.51	1.89	1.77
4	Arkansas	64	59	69	2.12	1.96	2.28

Wait…but now we can't differentiate betwen the "Number of TB cases" and "TB incidence" year columns. pandas has tried to make our lives easier by automatically adding ".1" to the latter columns, but this doesn't help us as humans understand the data.

We can do this manually with `df.rename()` (documentation):

```
rename_dict = {'2019': 'TB cases 2019',
               '2020': 'TB cases 2020',
               '2021': 'TB cases 2021',
               '2019.1': 'TB incidence 2019',
               '2020.1': 'TB incidence 2020',
               '2021.1': 'TB incidence 2021'}
tb_df = tb_df.rename(columns=rename_dict)
tb_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Total	8,900	7,173	7,860	2.71	2
1	Alabama	87	72	92	1.77	1
2	Alaska	58	58	58	7.91	7
3	Arizona	183	136	129	2.51	1
4	Arkansas	64	59	69	2.12	1

6.2 Record Granularity

You might already be wondering: What's up with that first record?

Row 0 is what we call a **rollup record**, or summary record. It's often useful when displaying tables to humans. The **granularity** of record 0 (Totals) vs the rest of the records (States) is different.

Okay, EDA step two. How was the rollup record aggregated?

Let's check if Total TB cases is the sum of all state TB cases. If we sum over all rows, we should get **2x** the total cases in each of our TB cases by year (why?).

```
tb_df.sum(axis=0)
```

	0
U.S. jurisdiction	TotalAlabamaAlaskaArizonaArkansasCaliforniaCol...
TB cases 2019	8,9008758183642,111666718245583029973261085237...
TB cases 2020	7,1737258136591,706525417194122219282169239376...
TB cases 2021	7,8609258129691,750585443194992281064255127494...
TB incidence 2019	109.94
TB incidence 2020	93.09
TB incidence 2021	102.94

Whoa, what's going on? Check out the column types:

```
tb_df.dtypes
```

	0
U.S. jurisdiction	object
TB cases 2019	object
TB cases 2020	object
TB cases 2021	object
TB incidence 2019	float64
TB incidence 2020	float64
TB incidence 2021	float64

Looks like those commas are causing all TB cases to be read as the `object` datatype, or **storage type** (close to the Python string datatype), so pandas is concatenating strings instead of adding integers.

Fortunately `read_csv` also has a `thousands` parameter (https://pandas.pydata.org/docs/reference/api/pandas.r

```
# improve readability: chaining method calls with outer parentheses/line breaks
tb_df = (
    pd.read_csv("data/cdc_tuberculosis.csv", header=1, thousands=',')
    .rename(columns=rename_dict)
)
tb_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Total	8900	7173	7860	2.71	2
1	Alabama	87	72	92	1.77	1
2	Alaska	58	58	58	7.91	7
3	Arizona	183	136	129	2.51	1
4	Arkansas	64	59	69	2.12	1

```
tb_df.sum()
```

	0
U.S. jurisdiction	TotalAlabamaAlaskaArizonaArkansasCaliforniaCol...
TB cases 2019	17800
TB cases 2020	14346
TB cases 2021	15720
TB incidence 2019	109.94
TB incidence 2020	93.09
TB incidence 2021	102.94

The Total TB cases look right. Phew!

Let's just look at the records with **state-level granularity**:

```
state_tb_df = tb_df[1:]
state_tb_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
1	Alabama	87	72	92	1.77	1
2	Alaska	58	58	58	7.91	7
3	Arizona	183	136	129	2.51	1
4	Arkansas	64	59	69	2.12	1
5	California	2111	1706	1750	5.35	4

6.3 Gather More Data: Census

U.S. Census population estimates source (2019), source (2020-2021).

Running the below cells cleans the data. There are a few new methods here: * `df.convert_dtypes()` (documentation) conveniently converts all float dtypes into ints and is out of scope for the class. * `df.drop_na()` (documentation) will be explained in more detail next time.

```
# 2010s census data
census_2010s_df = pd.read_csv("data/nst-est2019-01.csv", header=3, thousands=",")
census_2010s_df = (
    census_2010s_df
    .reset_index()
    .drop(columns=["index", "Census", "Estimates Base"])
    .rename(columns={"Unnamed: 0": "Geographic Area"})
    .convert_dtypes()                # "smart" converting of columns, use at your own ris
    .dropna()                        # we'll introduce this next time
)
census_2010s_df['Geographic Area'] = census_2010s_df['Geographic Area'].str.strip('.')

# with pd.option_context('display.min_rows', 30): # shows more rows
#     display(census_2010s_df)

census_2010s_df.head(5)
```

	Geographic Area	2010	2011	2012	2013	2014	2015	2016
0	United States	309321666	311556874	313830990	315993715	318301008	320635163	32294131
1	Northeast	55380134	55604223	55775216	55901806	56006011	56034684	5604233
2	Midwest	66974416	67157800	67336743	67560379	67745167	67860583	6798754
3	South	114866680	116006522	117241208	118364400	119624037	120997341	12235176
4	West	72100436	72788329	73477823	74167130	74925793	75742555	7655968

Occasionally, you will want to modify code that you have imported. To reimport those modifications you can either use the python importlib library:

```
from importlib import reload
reload(utils)
```

or use iPython magic which will intelligently import code when files change:

```
%load_ext autoreload
%autoreload 2
```

```
# census 2020s data
census_2020s_df = pd.read_csv("data/NST-EST2022-POP.csv", header=3, thousands=",")
census_2020s_df = (
    census_2020s_df
    .reset_index()
    .drop(columns=["index", "Unnamed: 1"])
```

```
    .rename(columns={"Unnamed: 0": "Geographic Area"})
    .convert_dtypes()                  # "smart" converting of columns, use at your own ris
    .dropna()                          # we'll introduce this next time
)
census_2020s_df['Geographic Area'] = census_2020s_df['Geographic Area'].str.strip('.')

census_2020s_df.head(5)
```

	Geographic Area	2020	2021	2022
0	United States	331511512	332031554	333287557
1	Northeast	57448898	57259257	57040406
2	Midwest	68961043	68836505	68787595
3	South	126450613	127346029	128716192
4	West	78650958	78589763	78743364

6.4 Joining Data on Primary Keys

Time to `merge`! Here we use the DataFrame method `df1.merge(right=df2, ...)` on DataFrame `df1` (documentation). Contrast this with the function `pd.merge(left=df1, right=df2, ...)` (documentation). Feel free to use either.

```
# merge TB dataframe with two US census dataframes
tb_census_df = (
    tb_df
    .merge(right=census_2010s_df,
           left_on="U.S. jurisdiction", right_on="Geographic Area")
    .merge(right=census_2020s_df,
           left_on="U.S. jurisdiction", right_on="Geographic Area")
)
tb_census_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

This is a little unwieldy. We could either drop the unneeded columns now, or just merge on smaller census DataFrames. Let's do the latter.

```
# try merging again, but cleaner this time
tb_census_df = (
    tb_df
    .merge(right=census_2010s_df[["Geographic Area", "2019"]],
           left_on="U.S. jurisdiction", right_on="Geographic Area")
    .drop(columns="Geographic Area")
    .merge(right=census_2020s_df[["Geographic Area", "2020", "2021"]],
           left_on="U.S. jurisdiction", right_on="Geographic Area")
    .drop(columns="Geographic Area")
)
tb_census_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

6.5 Reproducing Data: Compute Incidence

Let's recompute incidence to make sure we know where the original CDC numbers came from.

From the CDC report: TB incidence is computed as "Cases per 100,000 persons using mid-year population estimates from the U.S. Census Bureau."

If we define a group as 100,000 people, then we can compute the TB incidence for a given state population as

$$\text{TB incidence} = \frac{\text{TB cases in population}}{\text{groups in population}} = \frac{\text{TB cases in population}}{\text{population}/100000}$$

$$= \frac{\text{TB cases in population}}{\text{population}} \times 100000$$

Let's try this for 2019:

```
tb_census_df["recompute incidence 2019"] = tb_census_df["TB cases 2019"]/tb_census_df["201
tb_census_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

Awesome!!!

Let's use a for-loop and Python format strings to compute TB incidence for all years. Python f-strings are just used for the purposes of this demo, but they're handy to know when you explore data beyond this course (Python documentation).

```
# recompute incidence for all years
for year in [2019, 2020, 2021]:
    tb_census_df[f"recompute incidence {year}"] = tb_census_df[f"TB cases {year}"]/tb_cens
tb_census_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

These numbers look pretty close!!! There are a few errors in the hundredths place, particularly in 2021. It may be useful to further explore reasons behind this discrepancy.

```
tb_census_df.describe()
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB incid
count	51.000000	51.000000	51.000000	51.000000	51.000000	
mean	174.509804	140.647059	154.117647	2.102549	1.782941	
std	341.738752	271.055775	286.781007	1.498745	1.337414	
min	1.000000	0.000000	2.000000	0.170000	0.000000	
25%	25.500000	29.000000	23.000000	1.295000	1.210000	
50%	70.000000	67.000000	69.000000	1.800000	1.520000	
75%	180.500000	139.000000	150.000000	2.575000	1.990000	
max	2111.000000	1706.000000	1750.000000	7.910000	7.920000	

6.6 Bonus EDA: Reproducing the reported statistic

How do we reproduce that reported statistic in the original CDC report?

> Reported TB incidence (cases per 100,000 persons) increased **9.4%**, from **2.2** during 2020 to **2.4** during 2021 but was lower than incidence during 2019 (2.7). Increases occurred among both U.S.-born and non–U.S.-born persons.

This is TB incidence computed across the entire U.S. population! How do we reproduce this * We need to reproduce the "Total" TB incidences in our rolled record. * But our current `tb_census_df` only has 51 entries (50 states plus Washington, D.C.). There is no rolled record. * What happened…?

Let's get exploring!

Before we keep exploring, we'll set all indexes to more meaningful values, instead of just numbers that pertained to some row at some point. This will make our cleaning slightly easier.

```
tb_df = tb_df.set_index("U.S. jurisdiction")
tb_df.head(5)
```

U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
Total	8900	7173	7860	2.71	2.16
Alabama	87	72	92	1.77	1.43
Alaska	58	58	58	7.91	7.92
Arizona	183	136	129	2.51	1.89
Arkansas	64	59	69	2.12	1.96

```
census_2010s_df = census_2010s_df.set_index("Geographic Area")
census_2010s_df.head(5)
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	
United States	309321666	311556874	313830990	315993715	318301008	320635163	322941311	3
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	1
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	

```
census_2020s_df = census_2020s_df.set_index("Geographic Area")
census_2020s_df.head(5)
```

Geographic Area	2020	2021	2022
United States	331511512	332031554	333287557
Northeast	57448898	57259257	57040406
Midwest	68961043	68836505	68787595
South	126450613	127346029	128716192
West	78650958	78589763	78743364

It turns out that our merge above only kept state records, even though our original `tb_df` had the "Total" rolled record:

```
tb_df.head()
```

U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
Total	8900	7173	7860	2.71	2.16
Alabama	87	72	92	1.77	1.43
Alaska	58	58	58	7.91	7.92
Arizona	183	136	129	2.51	1.89
Arkansas	64	59	69	2.12	1.96

Recall that merge by default does an **inner** merge by default, meaning that it only preserves keys that are present in **both** DataFrames.

The rolled records in our census dataframes have different `Geographic Area` fields, which was the key we merged on:

```
census_2010s_df.head(5)
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	
United States	309321666	311556874	313830990	315993715	318301008	320635163	322941311	3
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	1
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	

The Census DataFrame has several rolled records. The aggregate record we are looking for actually has the Geographic Area named "United States".

One straightforward way to get the right merge is to rename the value itself. Because we now have the Geographic Area index, we'll use `df.rename()` (documentation):

```
# rename rolled record for 2010s
census_2010s_df.rename(index={'United States':'Total'}, inplace=True)
census_2010s_df.head(5)
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016
Total	309321666	311556874	313830990	315993715	318301008	320635163	322941311
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681

```
# same, but for 2020s rename rolled record
census_2020s_df.rename(index={'United States':'Total'}, inplace=True)
census_2020s_df.head(5)
```

Geographic Area	2020	2021	2022
Total	331511512	332031554	333287557
Northeast	57448898	57259257	57040406
Midwest	68961043	68836505	68787595
South	126450613	127346029	128716192
West	78650958	78589763	78743364

Next let's rerun our merge. Note the different chaining, because we are now merging on indexes (`df.merge()` documentation).

```
tb_census_df = (
    tb_df
    .merge(right=census_2010s_df[["2019"]],
           left_index=True, right_index=True)
    .merge(right=census_2020s_df[["2020", "2021"]],
           left_index=True, right_index=True)
)
tb_census_df.head(5)
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB ir
Total	8900	7173	7860	2.71	2.16	
Alabama	87	72	92	1.77	1.43	
Alaska	58	58	58	7.91	7.92	
Arizona	183	136	129	2.51	1.89	
Arkansas	64	59	69	2.12	1.96	

Finally, let's recompute our incidences:

```
# recompute incidence for all years
for year in [2019, 2020, 2021]:
    tb_census_df[f"recompute incidence {year}"] = tb_census_df[f"TB cases {year}"]/tb_cens
tb_census_df.head(5)
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB ir
Total	8900	7173	7860	2.71	2.16	
Alabama	87	72	92	1.77	1.43	
Alaska	58	58	58	7.91	7.92	
Arizona	183	136	129	2.51	1.89	
Arkansas	64	59	69	2.12	1.96	

We reproduced the total U.S. incidences correctly!

We're almost there. Let's revisit the quote:

> Reported TB incidence (cases per 100,000 persons) increased **9.4%**, from **2.2** during 2020 to **2.4** during 2021 but was lower than incidence during 2019 (2.7). Increases occurred among both U.S.-born and non–U.S.-born persons.

Recall that percent change from A to B is computed as percent change $= \frac{B-A}{A} \times 100$.

```
incidence_2020 = tb_census_df.loc['Total', 'recompute incidence 2020']
incidence_2020
```

```
2.1637257652759883
```

```
incidence_2021 = tb_census_df.loc['Total', 'recompute incidence 2021']
incidence_2021
```

```
2.3672448914298068
```

```
difference = (incidence_2021 - incidence_2020)/incidence_2020 * 100
difference
```

```
9.405957511804143
```

7 Regular Expressions

> **Note**
>
> - Understand Python string manipulation, Pandas Series methods
> - Parse and create regex, with a reference table
> - Use vocabulary (closure, metacharater, groups, etc.) to describe regex metacharacters

7.1 Why Work with Text?

Last lecture, we learned of the difference between quantitative and qualitative variable types. The latter includes string data - the primary focus of today's lecture. In this note, we'll discuss the necessary tools to manipulate text: Python string manipulation and regular expressions.

There are two main reasons for working with text.

1. Canonicalization: Convert data that has multiple formats into a standard form.
    - By manipulating text, we can join tables with mismatched string labels
2. Extract information into a new feature.
    - For example, we can extract date and time features from text

7.2 Python String Methods

First, we'll introduce a few methods useful for string manipulation. The following table includes a number of string operations supported by Python and `pandas`. The Python functions operate on a single string, while their equivalent in `pandas` are **vectorized** - they operate on a Series of string data.

Operation	Python	Pandas (Series)
Transformation	• `s.lower(_)` • `s.upper(_)`	• `ser.str.lower(_)` • `ser.str.upper(_)`

Operation	Python	Pandas (Series)
Replacement + Deletion	• `s.replace(_)`	• `ser.str.replace(_)`
Split	• `s.split(_)`	• `ser.str.split(_)`
Substring	• `s[1:4]`	• `ser.str[1:4]`
Membership	• `'_' in s`	• `ser.str.contains(_)`
Length	• `len(s)`	• `ser.str.len()`

We'll discuss the differences between Python string functions and `pandas` Series methods in the following section on canonicalization.

7.2.1 Canonicalization

Assume we want to merge the given tables.

```
import pandas as pd

with open('data/county_and_state.csv') as f:
    county_and_state = pd.read_csv(f)

with open('data/county_and_population.csv') as f:
    county_and_pop = pd.read_csv(f)
```

```
display(county_and_state), display(county_and_pop);
```

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LS

	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044

Last time, we used a **primary key** and **foreign key** to join two tables. While neither of these keys exist in our DataFrames, the `County` columns look similar enough. Can we convert these columns into one standard, canonical form to merge the two tables?

7.2.1.1 Canonicalization with Python String Manipulation

The following function uses Python string manipulation to convert a single county name into canonical form. It does so by eliminating whitespace, punctuation, and unnecessary text.

```
def canonicalize_county(county_name):
    return (
        county_name
            .lower()
            .replace(' ', '')
            .replace('&', 'and')
            .replace('.', '')
            .replace('county', '')
            .replace('parish', '')
    )

canonicalize_county("St. John the Baptist")
```

```
'stjohnthebaptist'
```

We will use the `pandas map` function to apply the `canonicalize_county` function to every row in both DataFrames. In doing so, we'll create a new column in each called `clean_county_python` with the canonical form.

```
county_and_pop['clean_county_python'] = county_and_pop['County'].map(canonicalize_county)
county_and_state['clean_county_python'] = county_and_state['County'].map(canonicalize_coun
```

```
display(county_and_state), display(county_and_pop);
```

	County	State	clean_county_python
0	De Witt County	IL	dewitt
1	Lac qui Parle County	MN	lacquiparle
2	Lewis and Clark County	MT	lewisandclark
3	St John the Baptist Parish	LS	stjohnthebaptist

	County	Population	clean_county_python
0	DeWitt	16798	dewitt
1	Lac Qui Parle	8067	lacquiparle
2	Lewis & Clark	55716	lewisandclark
3	St. John the Baptist	43044	stjohnthebaptist

7.2.1.2 Canonicalization with Pandas Series Methods

Alternatively, we can use `pandas` Series methods to create this standardized column. To do so, we must call the `.str` attribute of our Series object prior to calling any methods, like `.lower` and `.replace`. Notice how these method names match their equivalent built-in Python string functions.

Chaining multiple Series methods in this manner eliminates the need to use the `map` function (as this code is vectorized).

```
def canonicalize_county_series(county_series):
    return (
        county_series
            .str.lower()
            .str.replace(' ', '')
            .str.replace('&', 'and')
            .str.replace('.', '')
            .str.replace('county', '')
            .str.replace('parish', '')
    )

county_and_pop['clean_county_pandas'] = canonicalize_county_series(county_and_pop['County'
county_and_state['clean_county_pandas'] = canonicalize_county_series(county_and_state['Cou
```

```
FutureWarning:

The default value of regex will change from True to False in a future version. In addition, s
```

```
display(county_and_pop), display(county_and_state);
```

	County	Population	clean_county_python	clean_county_pandas
0	DeWitt	16798	dewitt	dewitt
1	Lac Qui Parle	8067	lacquiparle	lacquiparle
2	Lewis & Clark	55716	lewisandclark	lewisandclark
3	St. John the Baptist	43044	stjohnthebaptist	stjohnthebaptist

	County	State	clean_county_python	clean_county_pandas
0	De Witt County	IL	dewitt	dewitt
1	Lac qui Parle County	MN	lacquiparle	lacquiparle
2	Lewis and Clark County	MT	lewisandclark	lewisandclark
3	St John the Baptist Parish	LS	stjohnthebaptist	stjohnthebaptist

7.2.2 Extraction

Extraction explores the idea of obtaining useful information from text data. This will be particularily important in model building, which we'll study in a few weeks.

Say we want to read some data from a `.txt` file.

```
with open('data/log.txt', 'r') as f:
    log_lines = f.readlines()
```

```
log_lines
```

```
['169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585
 '193.205.203.3 - - [2/Feb/2005:17:23:6 -0800] "GET /stat141/Notes/dim.html HTTP/1.0" 404 302
 '169.237.46.240 - "" [3/Feb/2006:10:18:37 -0800] "GET /stat141/homework/Solutions/hw1Sol.pdf
```

Suppose we want to extract the day, month, year, hour, minutes, seconds, and timezone. Unfortunately, these items are not in a fixed position from the beginning of the string, so slicing by some fixed offset won't work.

Instead, we can use some clever thinking. Notice how the relevant information is contained within a set of brackets, further seperated by `/` and `:`. We can hone in on this region of text, and split the data on these characters. Python's built-in `.split` function makes this easy.

```
first = log_lines[0] # Only considering the first row of data

pertinent = first.split("[")[1].split(']')[0]
day, month, rest = pertinent.split('/')
year, hour, minute, rest = rest.split(':')
seconds, time_zone = rest.split(' ')
day, month, year, hour, minute, seconds, time_zone
```

```
('26', 'Jan', '2014', '10', '47', '58', '-0800')
```

There are two problems with this code:

1. Python's built-in functions limit us to extract data one record at a time
    - This can be resolved using a map function or Pandas Series methods.
2. The code is quite verbose
    - This is a larger issue that is trickier to solve

In the next section, we'll introduce regular expressions - a tool that solves problem 2.

7.3 Regex Basics

A **regular expression ("regex")** is a sequence of characters that specifies a search pattern. They are written to extract specific information from text. Regular expressions are essentially part of a smaller programming language embedded in Python, made available through the `re` module. As such, they have a stand-alone syntax and methods for various capabilities.

Regular expressions are useful in many applications beyond data science. For example, Social Security Numbers (SSNs) are often validated with regular expresions.

```
r"[0-9]{3}-[0-9]{2}-[0-9]{4}" # Regular Expression Syntax

# 3 of any digit, then a dash,
# then 2 of any digit, then a dash,
# then 4 of any digit
```

```
'[0-9]{3}-[0-9]{2}-[0-9]{4}'
```

There are a ton of resources to learn and experiment with regular expressions. A few are provided below:

- Official Regex Guide
- Data 100 Reference Sheet
- Regex101.com
  - Be sure to check `Python` under the category on the left.

7.3.1 Basics Regex Syntax

There are four basic operations with regular expressions.

Operation	Order	Syntax Example	Matches	Doesn't Match
`Or: |`	4	AA|BAAB	AA BAAB	every other string
`Concatenation`	3	AABAAB	AABAAB	every other string
`Closure: *` (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA

Operation	Order	Syntax Example	Matches	Doesn't Match
`Group: ()` (parenthesis)	1	A(A|B)AAB (AB)*A	AAAAB ABAAB A ABABABABA	every other string AA ABBA

Notice how these metacharacter operations are ordered. Rather than being literal characters, these **metacharacters** manipulate adjacent characters. `()` takes precedence, followed by `*`, and finally `|`. This allows us to differentiate between very different regex commands like `AB*` and `(AB)*`. The former reads "`A` then zero or more copies of `B`", while the latter specifies "zero or more copies of `AB`".

7.3.1.1 Examples

Question 1: Give a regular expression that matches `moon`, `moooon`, etc. Your expression should match any even number of `o`s except zero (i.e. don't match `mn`).

Answer 1: `moo(oo)*n`

- Hardcoding `oo` before the capture group ensures that `mn` is not matched.
- A capture group of `(oo)*` ensures the number of `o`'s is even.

Question 2: Using only basic operations, formulate a regex that matches `muun`, `muuuun`, `moon`, `moooon`, etc. Your expression should match any even number of `u`s or `o`s except zero (i.e. don't match `mn`).

Answer 2: `m(uu(uu)*|oo(oo)*)n`

- The leading `m` and trailing `n` ensures that only strings beginning with `m` and ending with `n` are matched.
- Notice how the outer capture group surrounds the `|`.
  - Consider the regex `m(uu(uu)*)|(oo(oo)*)n`. This incorrectly matches `muu` and `oooon`.
    - Each OR clause is everything to the left and right of `|`. The incorrect solution matches only half of the string, and ignores either the beginning `m` or trailing `n`.
    - A set of paranthesis must surround `|`. That way, each OR clause is everything to the left and right of `|` **within** the group. This ensures both the beginning `m` *and* trailing `n` are matched.

7.4 Regex Expanded

Provided below are more complex regular expression functions.

Operation	Syntax Example	Matches	Doesn't Match
`Any Character: .` (except newline)	.U.U.U.	CUMULUS JUGULUM	SUCCUBUS TUMULTUOUS
`Character Class: []` (match one character in `[]`)	[A-Za-z][a-z]*	word Capitalized	camelCase 4illegal
`Repeated "a" Times: {a}`	j[aeiou]{3}hn	jaoehn jooohn	jhn jaeiouhn
`Repeated "from a to b" Times: {a, b}`	j[0u]{1,2}hn	john juohn	jhn jooohn
`At Least One: +`	jo+hn	john jooooooohn	jhn jjohn
`Zero or One: ?`	joh?n	jon john	any other string

A character class matches a single character in it's class. These characters can be hardcoded – in the case of `[aeiou]` – or shorthand can be specified to mean a range of characters. Examples include:

1. `[A-Z]`: Any capitalized letter
2. `[a-z]`: Any lowercase letter
3. `[0-9]`: Any single digit
4. `[A-Za-z]`: Any capitalized of lowercase letter
5. `[A-Za-z0-9]`: Any capitalized or lowercase letter or single digit

7.4.0.1 Examples

Let's analyze a few examples of complex regular expressions.

Matches	Does Not Match
1. `.*SPB.*` RASPBERRY SPBOO	SUBSPACE SUBSPECIES
2. `[0-9]{3}-[0-9]{2}-[0-9]{4}` 231-41-5121 573-57-1821	231415121 57-3571821

Matches	Does Not Match
3. `[a-z]+@([a-z]+\.)+(edu|com)` horse@pizza.com horse@pizza.food.com	frank_99@yahoo.com hug@cs

Explanations

1. `.*SPB.*` only matches strings that contain the substring `SPB`.
    - The `.*` metacharacter matches any amount of non-negative characters. Newlines do not count.

2. This regular expression matches 3 of any digit, then a dash, then 2 of any digit, then a dash, then 4 of any digit
    - You'll recognize this as the familiar Social Security Number regular expression
3. Matches any email with a `com` or `edu` domain, where all characters of the email are letters.
    - At least one `.` must preceed the domain name. Including a backslash `\` before any metacharacter (in this case, the `.`) tells regex to match that character exactly.

7.5 Convenient Regex

Here are a few more convenient regular expressions.

Operation	Syntax Example	Matches	Doesn't Match
`built in character class`	`\w+ \d+ \s+`	Fawef_03 231123 `whitespace`	this person 423 people `non-whitespace`
`character class negation: [^]` (everything except the given characters)	[^a-z]+.	PEPPERS3982 17211!↑å	porch CLAmS
`escape character: \` (match the literal next character)	cow\.com	cow.com	cowscom
`beginning of line: ^`	^ark	ark two ark o ark	dark
`end of line: $`	ark$	dark ark o ark	ark two
`lazy version of zero or more : *?`	5.*?5	5005 55	5005005

7.5.0.1 Examples

Let's revist our earlier problem of extracting date/time data from the given `.txt` files. Here is how the data looked.

```
log_lines[0]
```

```
'169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 '
```

Question: Give a regular expression that matches everything contained within and including the brackets - the day, month, year, hour, minutes, seconds, and timezone.

Answer: `\[.*\]`

- Notice how matching the literal `[` and `]` is necessary. Therefore, an escape character `\` is required before both `[` and `]` - otherwise these metacharacters will match character classes.
- We need to match a particular format between `[` and `]`. For this example, `.*` will suffice.

Alternative Solution: `\[\w+/\w+/\w+:\w+:\w+:\w+\s-\w+\]`

- This solution is much safer.
  - Imagine the data between `[` and `]` was garbage - `.*` will still match that.
  - The alternate solution will only match data that follows the correct format.

7.6 Regex in Python and Pandas (Regex Groups)

7.6.1 Canonicalization

7.6.1.1 Canonicalization with Regex

Earlier in this note, we examined the process of canonicalization using Python string manipulation and `pandas` Series methods. However, we mentioned this approach had a major flaw: our code was unnecessarily verbose. Equipped with our knowledge of regular expressions, let's fix this.

To do so, we need to understand a few functions in the `re` module. The first of these is the substitute function: `re.sub(pattern, rep1, text)`. It behaves similarily to Python's built-in `.replace` function, and returns text with all instances of `pattern` replaced by `rep1`.

The regular expression here removes text surrounded by `<>` (also known as HTML tags).

In order, the pattern matches … 1. a single `<` 2. any character that is not a `>` : div, td valign…, /td, /div 3. a single `>`

Any substring in `text` that fulfills all three conditions will be replaced by `''`.

```
import re

text = "<div><td valign='top'>Moo</td></div>"
pattern = r"<[^>]+>"
re.sub(pattern, '', text)
```

```
'Moo'
```

Notice the `r` preceeding the regular expression pattern; this specifies the regular expression is a raw string. Raw strings do not recognize escape sequences (ie the Python newline metacharacter `\n`). This makes them useful for regular expressions, which often contain literal `\` characters.

In other words, don't forget to tag your regex with a `r`.

7.6.1.2 Canonicalization with Pandas

We can also use regular expressions with Pandas Series methods. This gives us the benefit of operating on an entire column of data as opposed to a single value. The code is simple: `ser.str.replace(pattern, repl, regex=True)`.

Consider the following DataFrame `html_data` with a single column.

```
data = {"HTML": ["<div><td valign='top'>Moo</td></div>", \
                 "<a href='http://ds100.org'>Link</a>", \
                 "<b>Bold text</b>"]}
html_data = pd.DataFrame(data)
```

```
html_data
```

	HTML
0	<td valign='top'>Moo</td>
1	Link
2	Bold text

```
pattern = r"<[^>]+>"
html_data['HTML'].str.replace(pattern, '', regex=True)
```

	HTML
0	Moo
1	Link
2	Bold text

7.6.2 Extraction

7.6.2.1 Extraction with Regex

Just like with canonicalization, the `re` module provides capability to extract relevant text from a string: `re.findall(pattern, text)`. This function returns a list of all matches to `pattern`.

Using the familiar regular expression for Social Security Numbers:

```
text = "My social security number is 123-45-6789 bro, or maybe it's 321-45-6789."
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"
re.findall(pattern, text)
```

```
['123-45-6789', '321-45-6789']
```

7.6.2.2 Extraction with Pandas

Pandas similarily provides extraction functionality on a Series of data: `ser.str.findall(pattern)`

Consider the following DataFrame `ssn_data`.

```
data = {"SSN": ["987-65-4321", "forty", \
                "123-45-6789 bro or 321-45-6789",
               "999-99-9999"]}
ssn_data = pd.DataFrame(data)
```

```
ssn_data
```

	SSN
0	987-65-4321
1	forty
2	123-45-6789 bro or 321-45-6789
3	999-99-9999

```
ssn_data["SSN"].str.findall(pattern)
```

	SSN
0	[987-65-4321]
1	[]
2	[123-45-6789, 321-45-6789]
3	[999-99-9999]

This function returns a list for every row containing the pattern matches in a given string.

7.6.3 Regular Expression Capture Groups

Earlier we used parentheses `( )` to specify the highest order of operation in regular expressions. However, they have another meaning; paranthesis are often used to represent **capture groups**. Capture groups are essentially, a set of smaller regular expressions that match multiple substrings in text data.

Let's take a look at an example.

7.6.3.1 Example 1

```
text = "Observations: 03:04:53 - Horse awakens. \
        03:05:14 - Horse goes back to sleep."
```

Say we want to capture all occurences of time data (hour, minute, and second) as *seperate entities*.

```
pattern_1 = r"(\d\d):(\d\d):(\d\d)"
re.findall(pattern_1, text)
```

```
[('03', '04', '53'), ('03', '05', '14')]
```

Notice how the given pattern has 3 capture groups, each specified by the regular expression `(\d\d)`. We then use `re.findall` to return these capture groups, each as tuples containing 3 matches.

These regular expression capture groups can be different. We can use the `(\d{2})` shorthand to extract the same data.

```
pattern_2 = r"(\d\d):(\d\d):(\d{2})"
re.findall(pattern_2, text)
```

```
[('03', '04', '53'), ('03', '05', '14')]
```

7.6.3.2 Example 2

With the notion of capture groups, convince yourself how the following regular expression works.

```
first = log_lines[0]
first
```

```
'169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 '
```

```
pattern = r'\[(\d+)\/(\w+)\/(\d+):(\d+):(\d+):(\d+) (.+)\]'
day, month, year, hour, minute, second, time_zone = re.findall(pattern, first)[0]
print(day, month, year, hour, minute, second, time_zone)
```

```
26 Jan 2014 10 47 58 -0800
```

7.7 Limitations of Regular Expressions

Today, we explored the capabilities of regular expressions in data wrangling with text data. However, there are a few things to be wary of.

Writing regular expressions is like writing a program.

- Need to know the syntax well.
- Can be easier to write than to read.
- Can be difficult to debug.

Regular expressions are terrible at certain types of problems:

- For parsing a hierarchical structure, such as JSON, use the `json.load()` parser, not regex!
- Complex features (e.g. valid email address).
- Counting (same number of instances of a and b). (impossible)
- Complex properties (palindromes, balanced parentheses). (impossible)

8 Visualization I

> **Note**
>
> - Use `matplotlib`, `seaborn`, and `plotly` to create data visualization.
> - Analyze histogram and identify outliers, mode, and skewness.
> - Using `boxplot` and `violinplot` to compare two distributions.

In our journey of the data science lifecycle, we have begun to explore the vast world of exploratory data analysis. More recently, we learned how to pre-process data using various data manipulation techniques. As we work towards understanding our data, there is one key component missing in our arsenal - the ability to visualize and discern relationships in existing data.

These next two lectures will introduce you to various examples of data visualizations and their underlying theory. In doing so, we'll motivate their importace in real-world examples with the use of plotting libraries.

8.1 Visualizations in Data 8 and Data 100 (so far)

You've likely encountered several forms of data visualizations in your studies. You may remember two such examples from Data 8: line charts and histograms. Each of these served a unique purpose. For example, line charts displayed how numerical quantities changed over time, while histograms were useful in understanding a variable's distribution.

Line Chart

Histogram

8.2 Goals of Visualization

Visualizations are useful for a number of reasons. In Data 100, we consider two areas in particular:

1. To broaden your understanding of the data

- Key part in exploratory data analysis.
- Useful in investigating relationships between variables.

2. To communicate results/conclusions to others
    - Visualization theory is especially important here.

One of the most common applications of visualizations is in understanding a distribution of data.

8.3 An Overview of Distributions

A distribution describes the frequency of unique values in a variable. Distributions must satisfy two properties:

1. Each data point must belong to only one category.
2. The total frequency of all categories must sum to 100%. In other words, their total count should equal the number of values in consideration.

Not a Valid Distribution

Valid Distribution

Left Diagram: This is not a valid distribution since individuals can be associated to more than one category and the bar values demonstrate values in minutes and not probability

Right Diagram: This example satisfies the two properties of distributions, so it is a valid distribution.

8.4 Bar Plots

As we saw above, a **bar plot** is one of the most common ways of displaying the distribution of a **qualitative** (categorical) variable. The length of a bar plot encodes the frequency of a category; the width encodes no useful information.

Let's contextualize this in an example. We will use the familiar `births` dataset from Data 8 in our analysis.

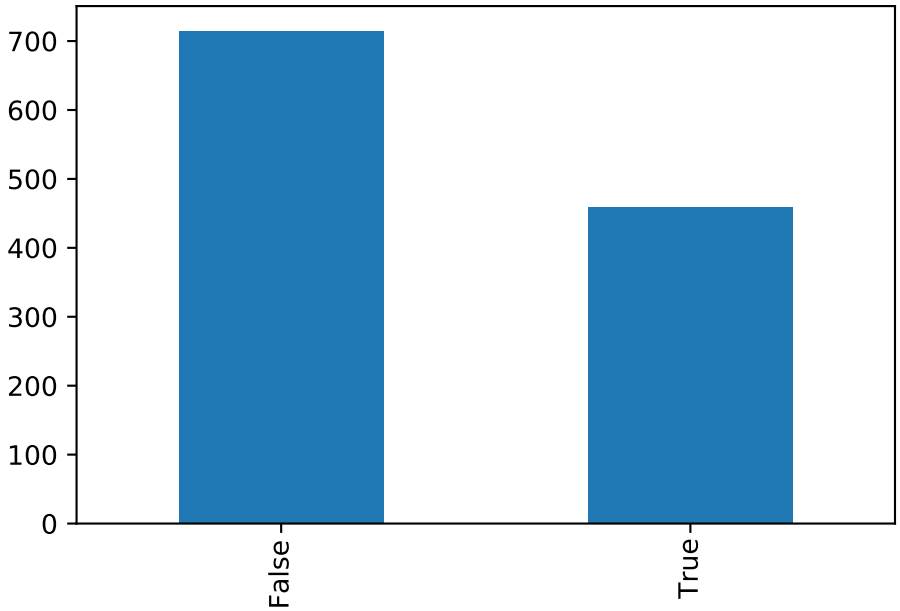
```
import pandas as pd

births = pd.read_csv("data/baby.csv")
births.head(5)
```

	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Mate
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
3	108	282	23	67	125	True
4	136	286	25	62	93	False

We can visualize the distribution of the `Maternal Smoker` column using a bar plot. There are a few ways to do this.

8.4.1 Plotting in Pandas

```
births['Maternal Smoker'].value_counts().plot(kind = 'bar');
```

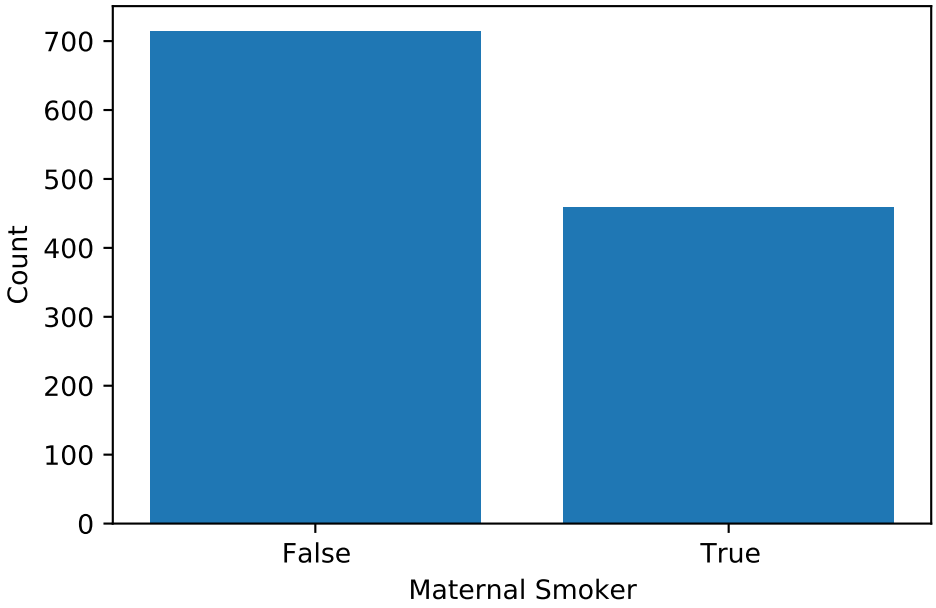


Recall that `.value_counts()` returns a `Series` with the total count of each unique value. We call `.plot(kind = 'bar')` on this result to visualize these counts as a bar plot.

Plotting methods in `pandas` are the least preferred and not supported in Data 100, as their functionality is limited. Instead, future examples will focus on other libaries built specifically for visualizing data. The most well-known library here is `matplotlib`.

8.4.2 Plotting in Matplotlib

```
import matplotlib.pyplot as plt

ms = births['Maternal Smoker'].value_counts()
plt.bar(ms.index.astype('string'), ms)
plt.xlabel('Maternal Smoker')
plt.ylabel('Count');
```

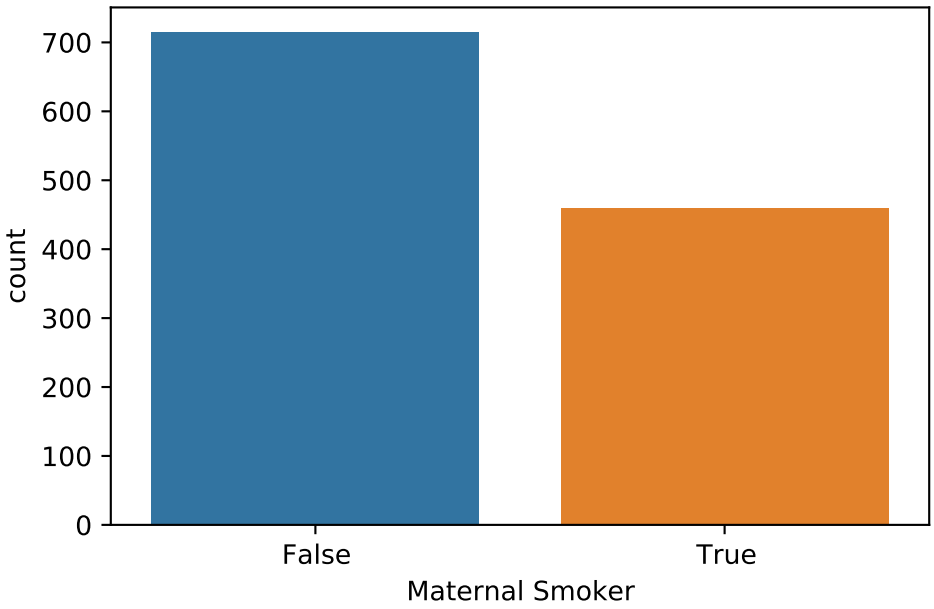


While more code is required to achieve the same result, `matplotlib` is often used over `pandas` for its ability to plot more complex visualizations, some of which are discussed shortly.

However, notice how we need to explicitly specify the type of the value for the x-axis to `string`. In absence of conversion, the x-axis will be a range of integers rather than the two categories, `True` and `False`. This is because `matplotlib` coerces `True` to a value of 1 and `False` to 0. Also, note how we needed to label the axes with `plt.xlabel` and `plt.ylabel` - `matplotlib` does not support automatic axis labeling. To get around these inconveniences, we can use a more effecient plotting library, `seaborn`.

8.4.3 Plotting in Seaborn

```
import seaborn as sns
sns.countplot(data = births, x = 'Maternal Smoker');
```



`seaborn.countplot` both counts and visualizes the number of unique values in a given column. This column is specified by the `x` argument to `sns.countplot`, while the `DataFrame` is specified by the `data` argument.

For the vast majority of visualizations, `seaborn` is far more concise and aesthetically pleasing than `matplotlib`. However, the color scheme of this particular bar plot is abritrary - it encodes no additional information about the categories themselves. This is not always true; color may signify meaningful detail in other visualizations. We'll explore this more in-depth during the next lecture.

8.4.4 Plotting in Plotly

`plotly` is one of the most versatile plottling libraries and widely used in industry. However, `plotly` has various dependencies that make it difficult to support in Data 100. Therfore, we have intentionally excluded the code to generate the plot above.

By now, you'll have noticed that each of these plotting libraries have a very different syntax. As with `pandas`, we'll teach you the important methods in `matplotlib` and `seaborn`, but you'll learn more through documentation.

1. Matplotlib Documentation
2. Seaborn Documentation

Example Questions:

- What colors should we use?
- How wide should the bars be?
- Should the legend exist?
- Should the bars and axes have dark borders?

To accomplish goal 2, here are some ways we can improve plot:

- Introducing different colors for each bar
- Including a legend
- Including a title
- Labeling the y-axis
- Using color-blind friendly palettes
- Re-orienting the labels
- Increase the font size

8.5 Histograms

Histograms are a natural extension to bar plots; they visualize the distribution of **quantitative** (numerical) data.

Revisiting our example with the `births` DataFrame, let's plot the distribution of the `Maternal Pregnancy Weight` column.

```
births.head(5)
```

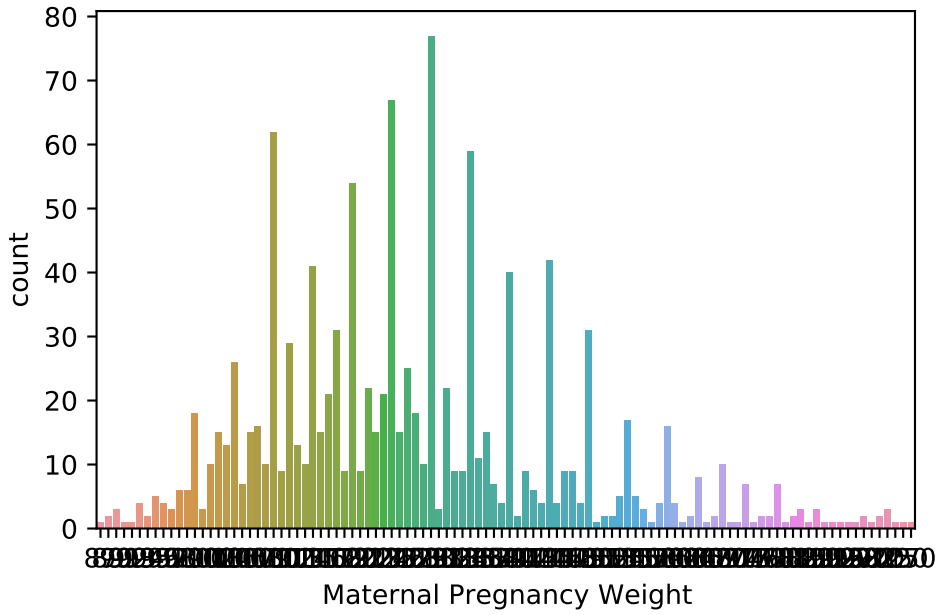
	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Mate
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
3	108	282	23	67	125	True
4	136	286	25	62	93	False

How should we define our categories for this variable? In the previous example, these were the unique values of the `Maternal Smoker` column: `True` and `False`. If we use similar logic

here, our categories are the different numerical weights contained in the `Maternal Pregnancy Weight` column.

Under this assumption, let's plot this distribution using the `seaborn.countplot` function.

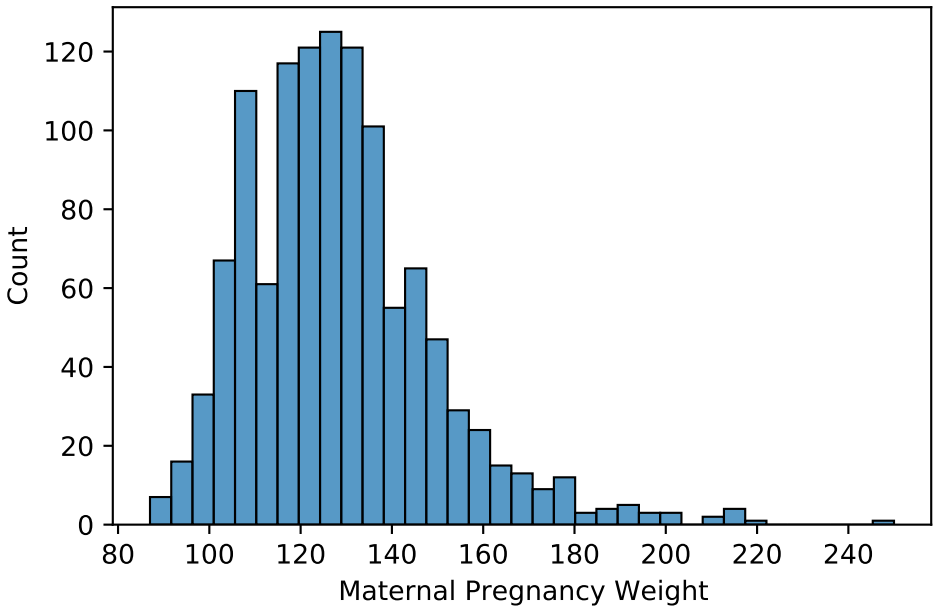
```
sns.countplot(data = births, x = 'Maternal Pregnancy Weight');
```



This histogram clearly suffers from **overplotting**. This is somewhat expected for `Maternal Pregnancy Weight` - it is a quantitative variable that takes on a wide range of values.

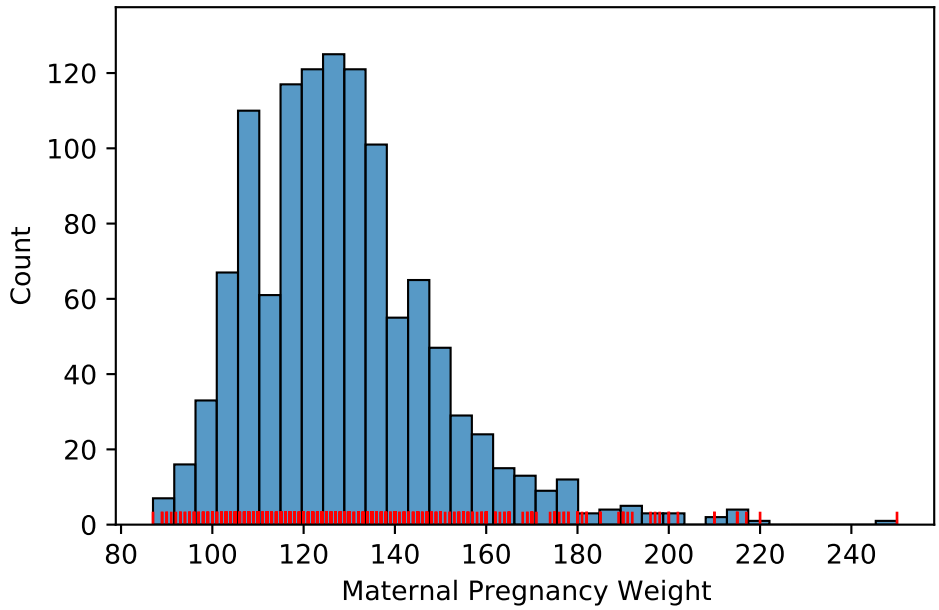
To combat this problem, statisticians use bins to categorize numerical data. Luckily, `seaborn` provides a helpful plotting function that automatically bins our data.

```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
```



This diagram is known as a histogram. While it looks more reasonable, notice how we lose fine-grain information on the distribution of data contained within each bin. We can introduce rug plots to minimize this information loss. An overlaid "rug plot" displays the within-bin distribution of our data, as denoted by the thickness of the colored line on the x-axis.

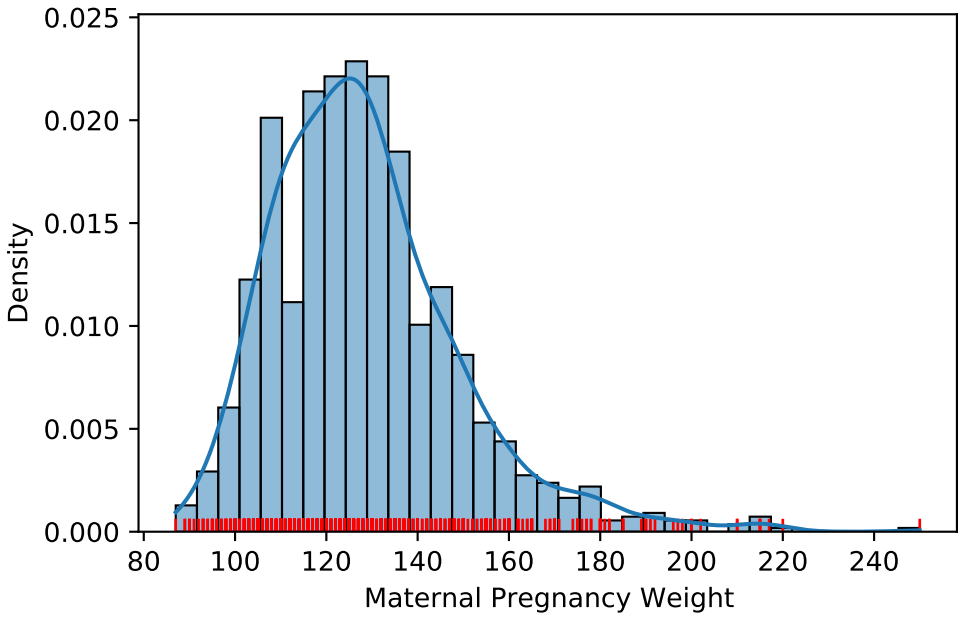
```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
sns.rugplot(data = births, x = 'Maternal Pregnancy Weight', color = 'red');
```



You may have seen histograms drawn differently - perhaps with an overlaid **density curve** and normalized y-axis. We can display both with a few tweaks to our code.

To visualize a density curve, we can set the the `kde = True` argument of the `sns.histplot`. Setting the argument `stat = 'density'` normalizes our histogram and displays densities, instead of counts, on the y-axis. You'll notice that the area under the density curve is 1.

```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight', kde = True,
             stat = 'density')
sns.rugplot(data = births, x = 'Maternal Pregnancy Weight', color = 'red');
```



8.6 Evaluating Histograms

Histograms allow us to assess a distribution by their shape. There are a few properties of histograms we can analyze:

1. Skewness and Tails
    - Skewed left vs skewed right
    - Left tail vs right tail
2. Outliers
    - Defined arbitrarily for now
3. Modes
    - Most commonly occuring data

8.6.1 Skewness and Tails

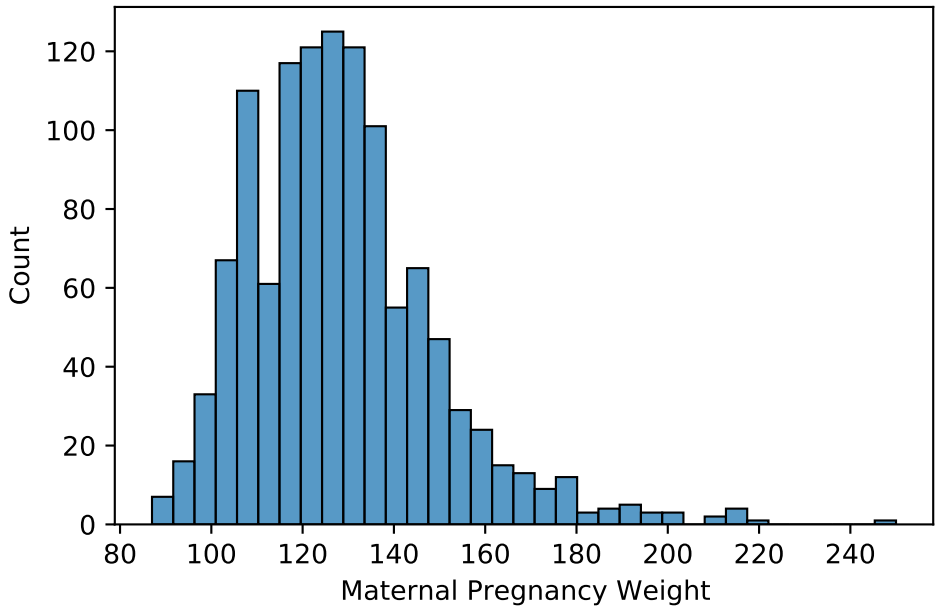
If a distribution has a long right tail (such as `Maternal Pregancy Weight`), it is **skewed right**. In a right-skewed distribution, the few large outliers "pull" the mean to the **right** of the median.

If a distribution has a long left tail, it is **skewed left**. In a left-skewed distribution, the few small outliers "pull" the mean to the **left** of the median.

In the case where a distribution has equal-sized right and left tails, it is **symmetric**. The mean is approximately **equal** to the median. Think of mean as the balancing point of the distribution

```
import numpy as np

sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
df_mean = np.mean(births['Maternal Pregnancy Weight'])
df_median = np.median(births['Maternal Pregnancy Weight'])
print("The mean is: {} and the median is {}".format(df_mean,df_median))
```

```
The mean is: 128.4787052810903 and the median is 125.0
```



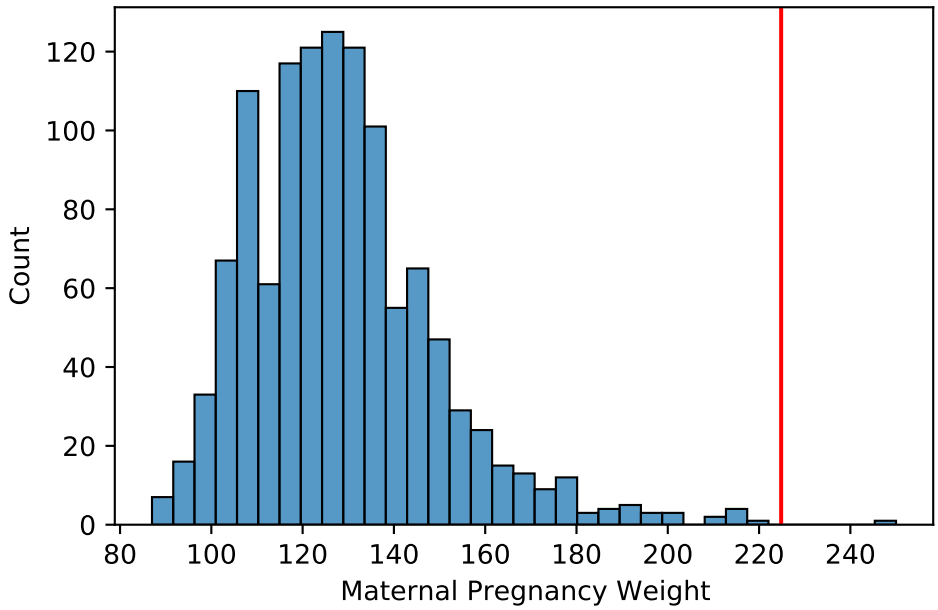
8.6.2 Outliers

Loosely speaking, an **outlier** is defined as a data point that lies an abnormally large distance away from other values. We'll define the statistical measure for this shortly.

Outliers disproportionately influce the mean because their magnitude is directly involved in computing the average. However, the median is largely unaffected - the magnitude of an outlier

is irrelevant; we only care that it is some non-zero distance away from the midpoint of the data.

```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
## Where do we draw the line of outlier?
plt.axvline(df_mean*1.75, color = 'red');
```

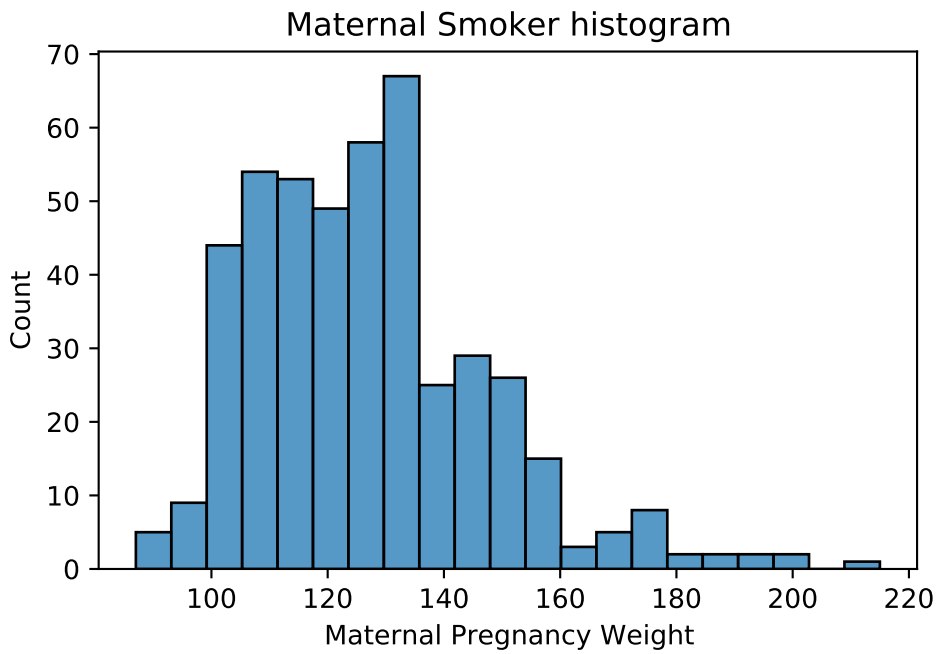


8.6.3 Modes

A **mode** of a distribution is a local or global maximum. A distribution with a single clear maximum is **unimodal**, distributions with two modes are **bimodal**, and those with 3 or more are **multimodal**. You need to distinguish between **modes** and *random noise*.

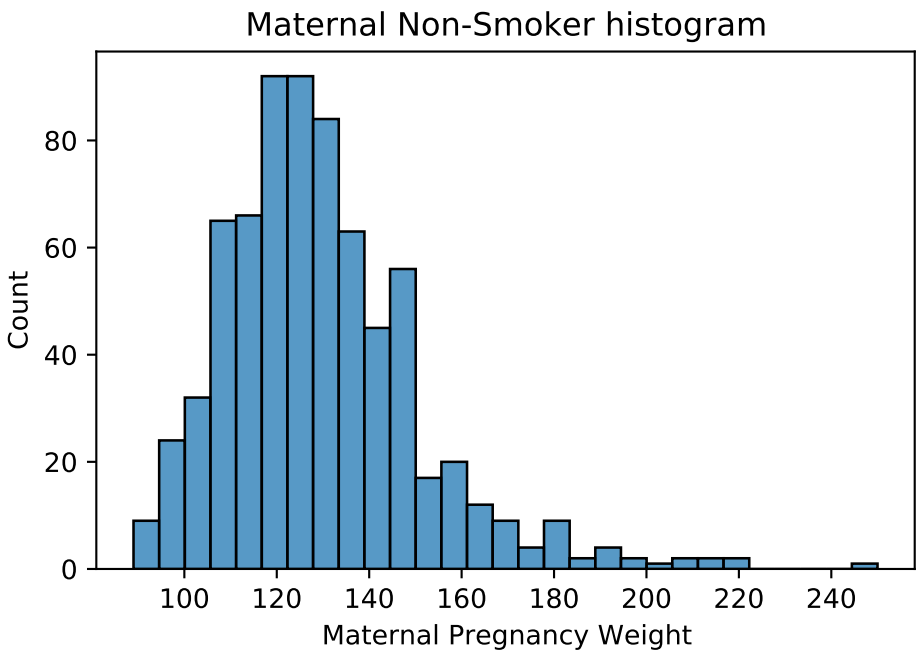
For example, the distribution of birth weights for maternal smokers is (weakly) multimodal.

```
births_maternal_smoker = births[births['Maternal Smoker'] == True]
sns.histplot(data = births_maternal_smoker, x = 'Maternal Pregnancy Weight')\
            .set(title = 'Maternal Smoker histogram');
```



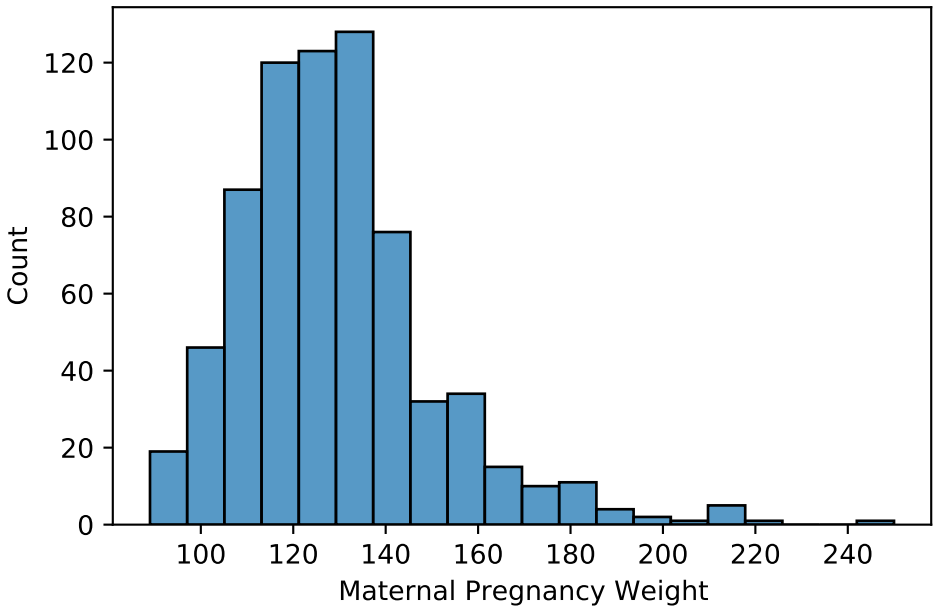
On the other hand, the distribution of birth weights for maternal non-smokers is weakly bimodal.

```
births_maternal_non_smoker = births[births['Maternal Smoker'] == False]
sns.histplot(data = births_maternal_non_smoker, x = 'Maternal Pregnancy Weight')\
            .set(title = 'Maternal Non-Smoker histogram');
```



However, changing the bins reveals that the data is not bi-modal.

```
sns.histplot(data = births_maternal_non_smoker, x = 'Maternal Pregnancy Weight',\
             bins = 20);
```



8.7 Density Curves

Instead of a discrete histogram, we can visualize what a continuous distribution corresponding to that same data could look like using a curve. - The smooth curve drawn on top of the histogram here is called a density curve.

In lecture 8, we will study how exactly to compute these density curves (using a technique is called Kernel Density Estimation).

If we plot `birth weights` of babies of *smoking mothers*, we get a histogram that appears bimodal.

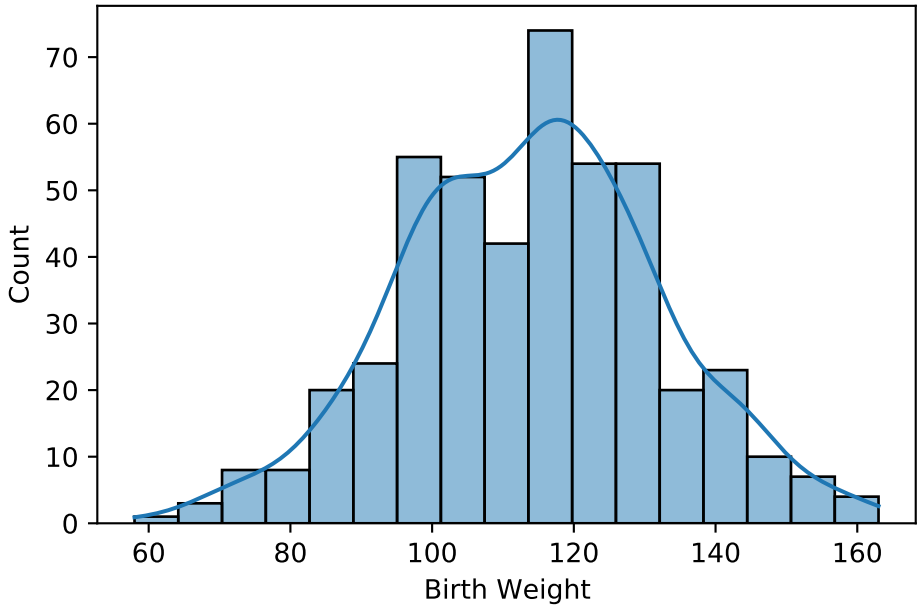
- Density curve reinforces belief in this bimodality.

However, if we plot `birth weights` of babies of *non-smoking mothers*, we get a histogram that appears unimodal.

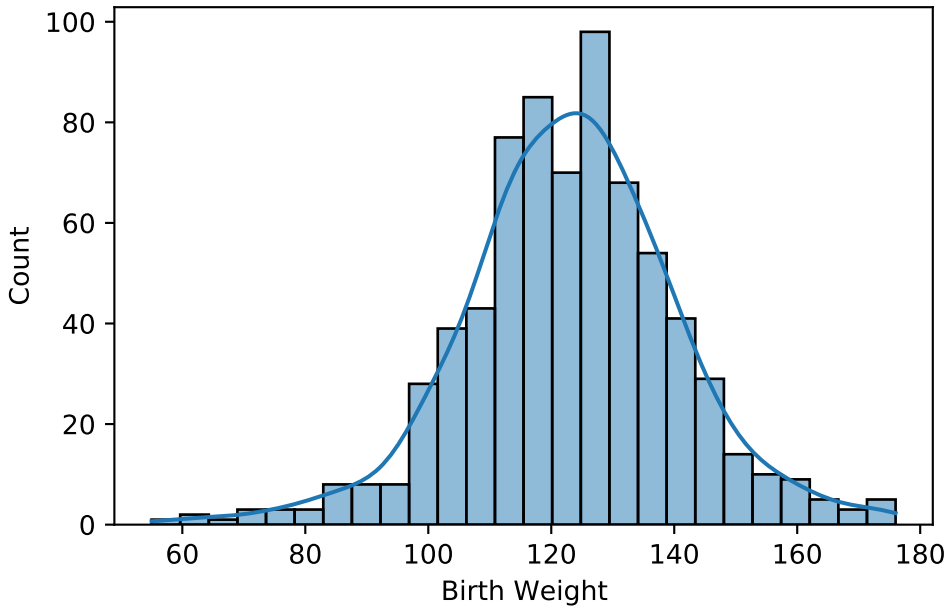
From a goal 1 perspective, this is EDA which tells us there may be something interesting here worth pursuing.

- Deeper analysis necessary!
- If we found something truly interesting, we'd have to cautiously write up an argument and create goal 2 level visualizations.

```
births_non_maternal_smoker = births[births['Maternal Smoker'] == False]
births_maternal_smoker = births[births['Maternal Smoker'] == True]

sns.histplot(data = births_maternal_smoker , x = 'Birth Weight',\
             kde = True);
```




```
sns.histplot(data = births_non_maternal_smoker , x = 'Birth Weight',\
             kde = True);
```



8.7.1 Histograms and Density

Rather than labeling by counts, we can instead plot the density, as shown below. Density gives us a measure that is invariant to the total number of observed units. The numerical values on the Y-axis for a sample of 100 units would be the same for when we observe a sample of 10000 units instead. We can still calculate the absolute number of observed units using density.

Example: There are 1174 observations total. - Total area of this bin should be: 120/1174 = ~10% - Density of this bin is therefore: 10% / (115 - 110) = 0.02

8.8 Box Plots and Violin Plots

8.8.1 Boxplots

Boxplots are an alternative to histograms that visualize numerical distributions. They are especially useful in graphicaly summarizing several characteristics of a distribution. These include:

1. Lower Quartile (1^{st} Quartile)
2. Median (2^{nd} Quartile)
3. Upper Quartile (3^{rd} Quartile)
4. Interquartile Range (IQR)

5. Whiskers
6. Outliers

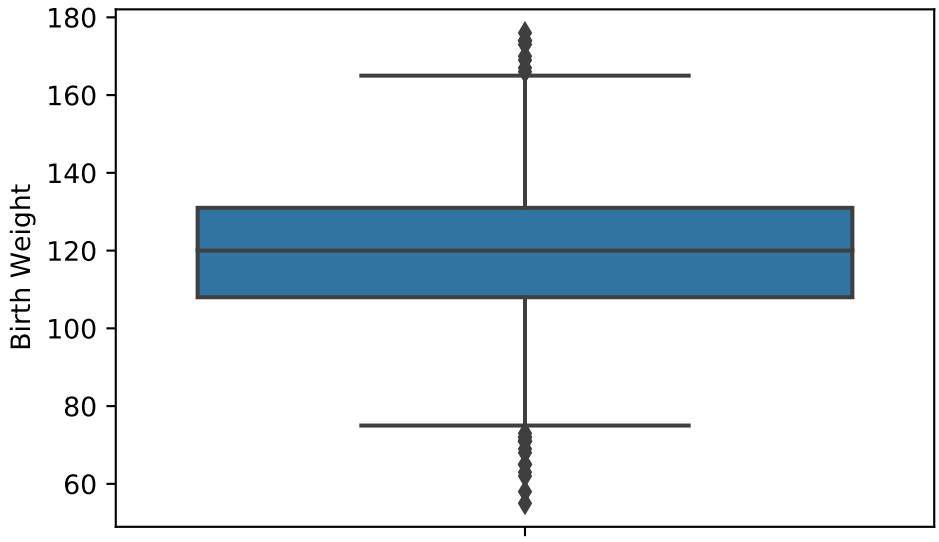
The **lower quartile**, **median**, and **uper quartile** are the 25^{th}, 50^{th}, and 75^{th} percentiles of data, respectively. The **interquartile range** measures the spread of the middle 50% of the distribution, calculated as the (3^{rd} Quartile $-$ 1^{st} Quartile).

The **whiskers** of a box-plot are the two points that lie at the [1^{st} Quartile $-$ (1.5$\times$ IQR)], and the [3^{rd} Quartile + (1.5$\times$ IQR)]. They are the lower and upper ranges of "normal" data (the points excluding outliers). Subsequently, the **outliers** are the data points that fall beyond the whiskers, or further than (1.5$\times$ IQR) from the extreme quartiles.

Let's visualize a box-plot of the `Birth Weight` column.

```
sns.boxplot(data = births, y = 'Birth Weight');

bweights = births['Birth Weight']
q1 = np.percentile(bweights, 25)
q2 = np.percentile(bweights, 50)
q3 = np.percentile(bweights, 75)
iqr = q3 - q1
whisk1 = q1 - (1.5 * iqr)
whisk2 = q3 + (1.5 * iqr)

print("The first quartile is {}".format(q1))
print("The second quartile is {}".format(q2))
print("The third quartile is {}".format(q3))
print("The interquartile range is {}".format(iqr))
print("The whiskers are {} and {}".format(whisk1, whisk2))
```

```
The first quartile is 108.0
The second quartile is 120.0
The third quartile is 131.0
The interquartile range is 23.0
The whiskers are 73.5 and 165.5
```

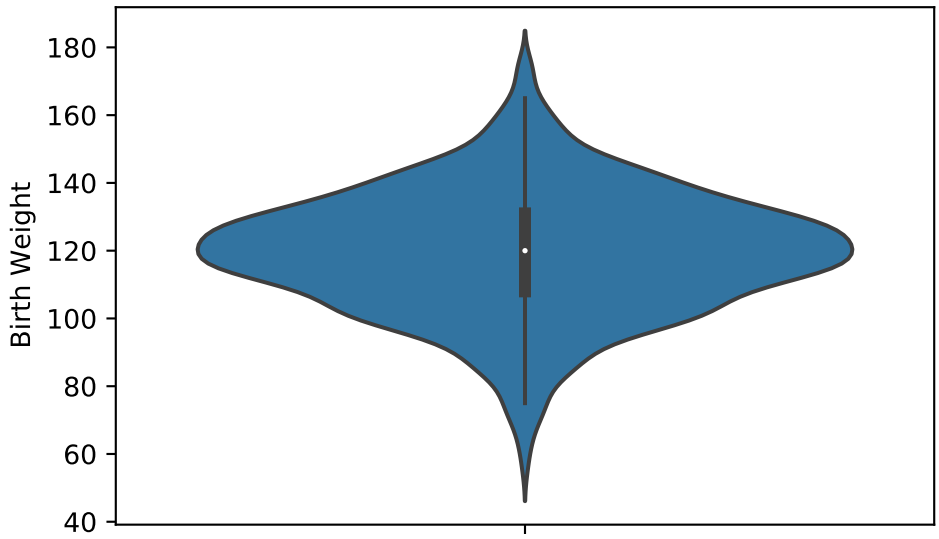


Here is a helpful visual that summarizes our discussion above.

8.8.2 Violin Plots

Another diagram that is useful in visualizing a variable's distribution is the violin plot. A **violin plot** supplements a box-plot with a smoothed density curve on either side of the plot. These density curves highlight the relative frequency of variable's possible values. If you look closely, you'll be able to discern the quartiles, whiskers, and other hallmark features of the box-plot.

```
sns.violinplot(data = births, y = 'Birth Weight');
```

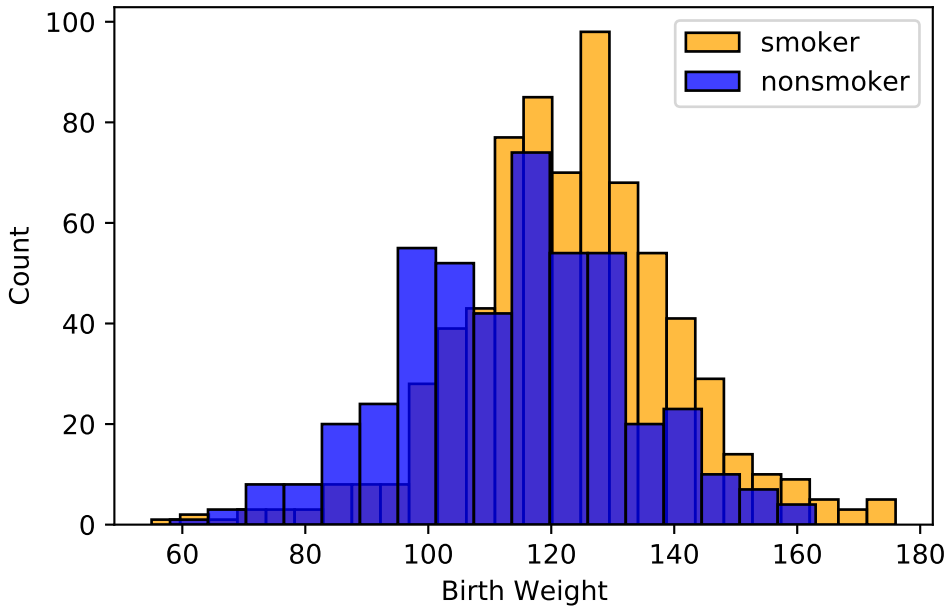


8.9 Comparing Quantitative Distributions

Earlier in our discussion of the mode, we visualized two histograms that described the distribution of birth weights for maternal smokers and non-smokers. However, comparing these histograms was difficult because they were displayed on seperate plots. Can we overlay the two to tell a more compelling story?

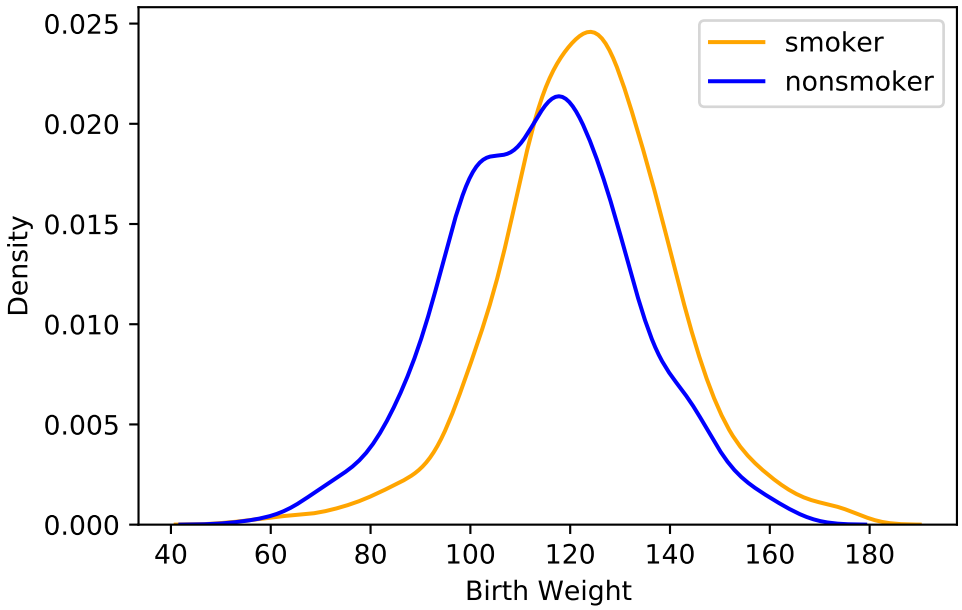
In `seaborn`, multiple calls to a plotting library in the same code cell will overlay the plots. For example:

```
births_maternal_smoker = births[births['Maternal Smoker'] == False]
births_non_maternal_smoker = births[births['Maternal Smoker'] == True]

sns.histplot(data = births_maternal_smoker, x = 'Birth Weight',
             color = 'orange', label = 'smoker')
sns.histplot(data = births_non_maternal_smoker, x = 'Birth Weight',
             color = 'blue', label = 'nonsmoker')
plt.legend();
```



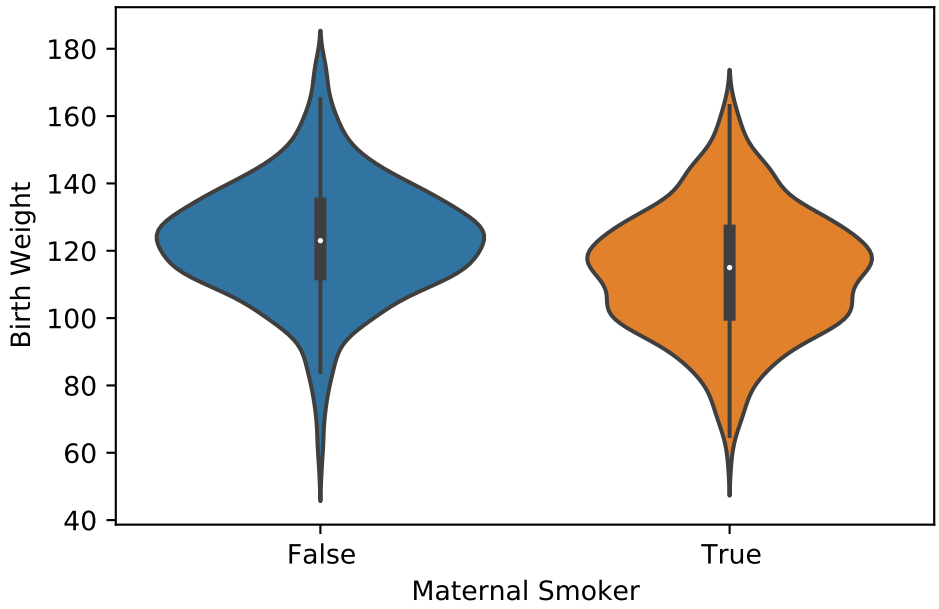
However, notice how this diagram suffers from overplotting. We can fix this with a call to `sns.kdeplot`. This will remove the bins and overlay the histogram with a density curve that better summarizes the distribution.

```
sns.kdeplot(data = births_maternal_smoker, x = 'Birth Weight', color = 'orange', label = '
sns.kdeplot(data = births_non_maternal_smoker, x = 'Birth Weight', color = 'blue', label =
plt.legend();
```

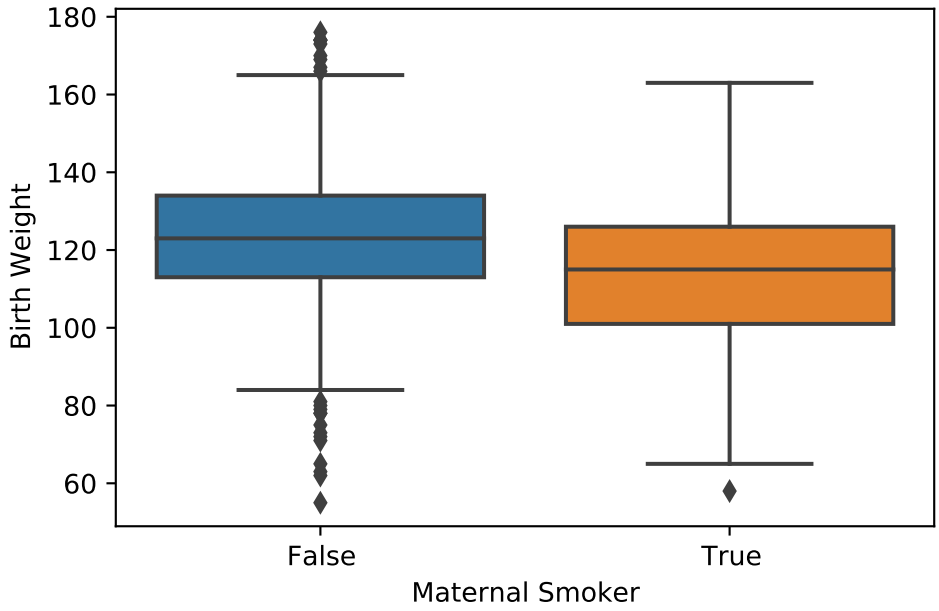


Unfortunately, we lose critical information in our distribution by removing small details. Therefore, we typically prefer to use box-plots and violin plots when comparing distributions. These are more concise and allow us to compare summary statistics across many distributions.

```
sns.violinplot(data = births, x = 'Maternal Smoker', y = 'Birth Weight');
```




```
sns.boxplot(data=births, x = 'Maternal Smoker', y = 'Birth Weight');
```



8.10 Ridge Plots

Ridge plots show many density curves offset from one another with minimal overlap. They are useful when the specific shape of each curve is important.

9 Visualization II

> **Note**
>
> - Use KDE for estimating density curve.
> - Using transformations to analyze the relationship between two variables.
> - Evalauting quality of a visualization based on visualization theory concepts.

9.1 Kernel Density Functions

9.1.1 KDE Mechanics

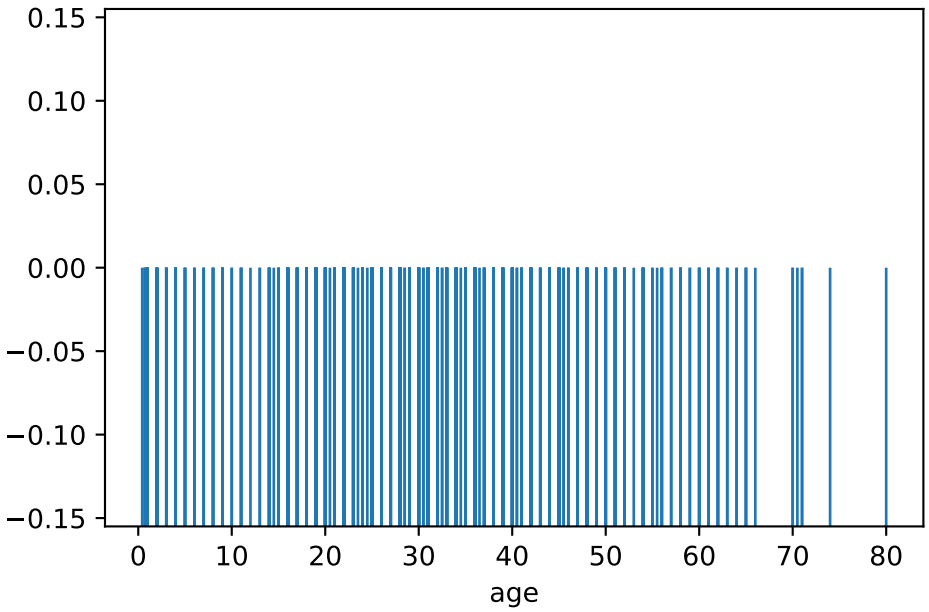
In the last lecture, we learned that density curves are smooth, continuous functions that represent a distribution of values. In this section, we'll learn how to construct density curves using Kernel Density Estimation (KDE).

9.1.1.1 Smoothing

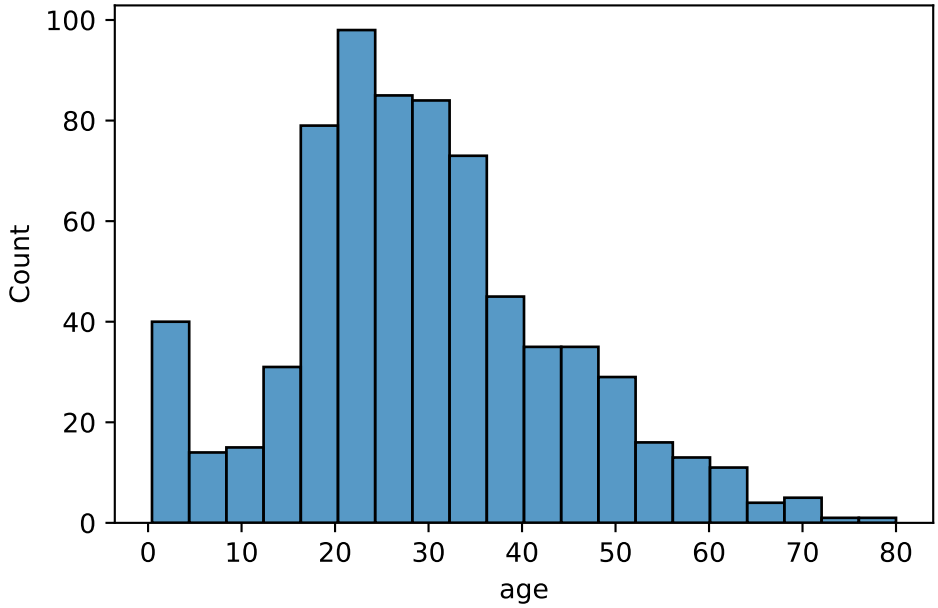
Kernel Density Estimation involves a technique called **smoothing** - a process applied to a distribution of values that allows us to analyze the more general structure of the dataset.

Many of the visualizations we learned during the last lecture are examples of smoothing. Histograms are smoothed versions of one-dimensional rug plots, and hex plots are smoother alternatives to two-dimensional scatter plots. They remove the detail from individual observations so we can visualize the larger properties of our distribution.

```
import seaborn as sns

titanic = sns.load_dataset('titanic')
sns.rugplot(titanic['age'],height = 0.5);
```




```
sns.histplot(titanic['age']);
```



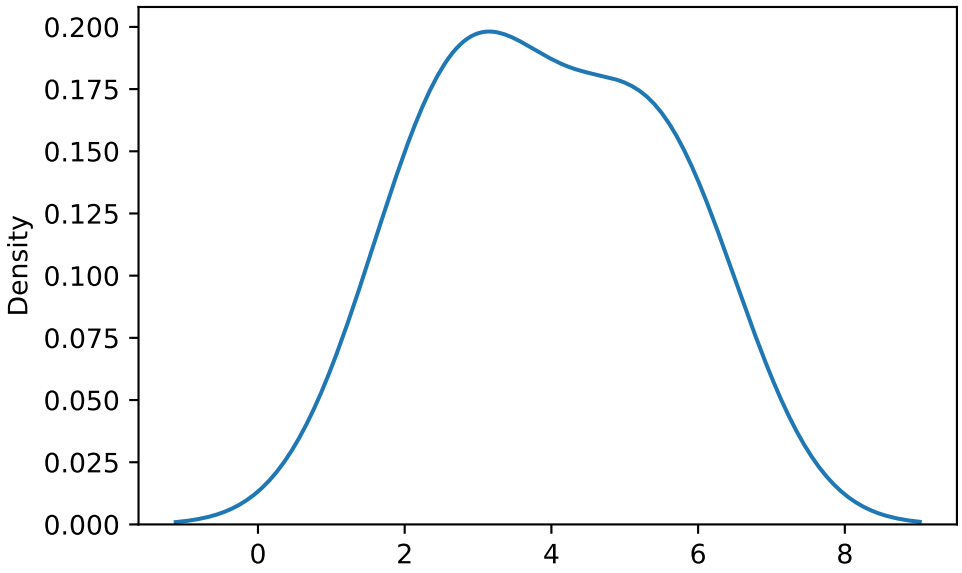
9.1.1.2 Kernel Density Estimation

Kernel Density Estimation is a smoothing technique that allows us to estimate a density curve (also known as a probability density function) from a set of observations. There are a few steps in this process:

1. Place a kernel at each data point
2. Normalize kernels to have total area of 1 (across all kernels)
3. Sum kernels together

Suppose we have 5 data points: $[2.2, 2.8, 3.7, 5.3, 5.7]$. We wish to recreate the following Kernel Density Estimate:

```
data = [2.2, 2.8, 3.7, 5.3, 5.7]
sns.kdeplot(data);
```



Let's walk through each step to construct this density curve.

9.1.1.2.1 Step 1 - Place a Kernel at Each Data Point

To begin generating a density curve, we need to choose a **kernel** and **bandwidth value** (α). What are these exactly? A **kernel** is a density curve itself, and the **bandwidth** (α) is a measure of the kernel's width. Recall that a valid density has an area of 1.

At each of our 5 points (depicted in the rug plot on the left), we've placed a Gaussian kernel with a bandwidth parameter of alpha = 1. We'll explore what these are in the next section.

Rugplot of Data

Kernelized Data

9.1.1.2.2 Step 2 - Normalize Kernels to Have Total Area of 1

Notice how these 5 kernels are density curves - meaning they each have an area of 1. In Step 3, we will be summing each these kernels, and we want the result to be a valid density that has an area of 1. Therefore, it makes sense to normalize our current set of kernels by multiplying each by $\frac{1}{5}$.

Kernelized Data

Normalized Kernels

9.1.1.2.3 Step 3 - Sum Kernels Together

Our kernel density estimate (KDE) is the sum of the normalized kernels along the x-axis. It is depicted below on the right.

Normalized Kernels

Kernel Density Estimate

9.1.2 Kernel Functions and Bandwidth

9.1.2.1 Kernels

A **kernel** (for our purposes) is a valid density function. This means it:

- Must be non-negative for all inputs.
- Must integrate to 1.

A general "KDE formula" function is given above.

1. $K_\alpha(x, xi)$ is the kernel centered on the observation `i`.
    - Each kernel individually has area 1.
    - x represents any number on the number line. It is the input to our function.
2. n is the number of observed data points that we have.
    - We multiply by $\frac{1}{n}$ so that the total area of the KDE is still 1.

3. Each $x_i \in \{x_1, x_2, \ldots, x_n\}$ represents an observed data point.
    - These are what we use to create our KDE by summing multiple shifted kernels centered at these points.

*α (alpha) is the bandwidth or smoothing parameter.

9.1.2.1.1 Gaussian Kernel

The most common kernel is the **Gaussian kernel**. The Gaussian kernel is equivalent to the Gaussian probability density function (the Normal distribution), centered at the observed value x_i with a standard deviation of α (this is known as the **bandwidth** parameter).

$K_a(x, x_i) = \frac{1}{\sqrt{2\pi\alpha^2}} e^{-\frac{(x-x_i)^2}{2\alpha^2}}$

```
import numpy as np
import matplotlib.pyplot as plt

def gaussian_kernel(alpha, x, z):
    return 1.0/np.sqrt(2. * np.pi * alpha**2) * np.exp(-(x - z) ** 2 / (2.0 * alpha**2))

xs = np.linspace(-5, 5, 200)
alpha = 1
kde_curve = [gaussian_kernel(alpha, x, 0) for x in xs]
plt.plot(xs, kde_curve);
```

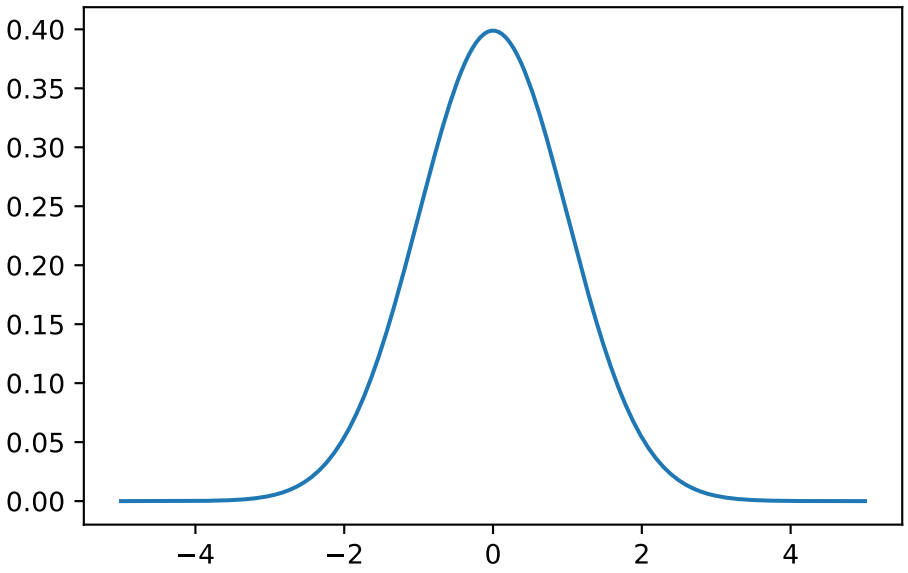


Figure 9.1: The Gaussian kernel centered at 0 with bandwidth $\alpha = 1$.

If you've taken a probability class, you'll recognize that the mean of this Gaussian kernel is x_i and the standard deviation is α. Increasing α - equivalently, the bandwidth - smoothens the density curve. Larger values of α are typically easier to understand; however, we begin to lose important distributional information.

Here is how adjusting α affects a distribution in some variable from an arbitrary dataset.

Gaussian Kernel, $\alpha = 0.1$

Gaussian Kernel, $\alpha = 1$

Gaussian Kernel, $\alpha = 2$

Gaussian Kernel, $\alpha = 10$

9.1.2.1.2 Boxcar Kernel

Another example of a kernel is the **Boxcar kernel**. The boxcar kernel assigns a uniform density to points within a "window" of the observation, and a density of 0 elsewhere. The equation below is a Boxcar kernel with the center at x_i and the bandwidth of α.

$$K_a(x, x_i) = \begin{cases} \frac{1}{\alpha}, & |x - x_i| \leq \frac{\alpha}{2} \\ 0, & \text{else} \end{cases}$$

```
def boxcar_kernel(alpha, x, z):
    return (((x-z)>=-alpha/2)&((x-z)<=alpha/2))/alpha

xs = np.linspace(-5, 5, 200)
alpha=1
kde_curve = [boxcar_kernel(alpha, x, 0) for x in xs]
plt.plot(xs, kde_curve);
```

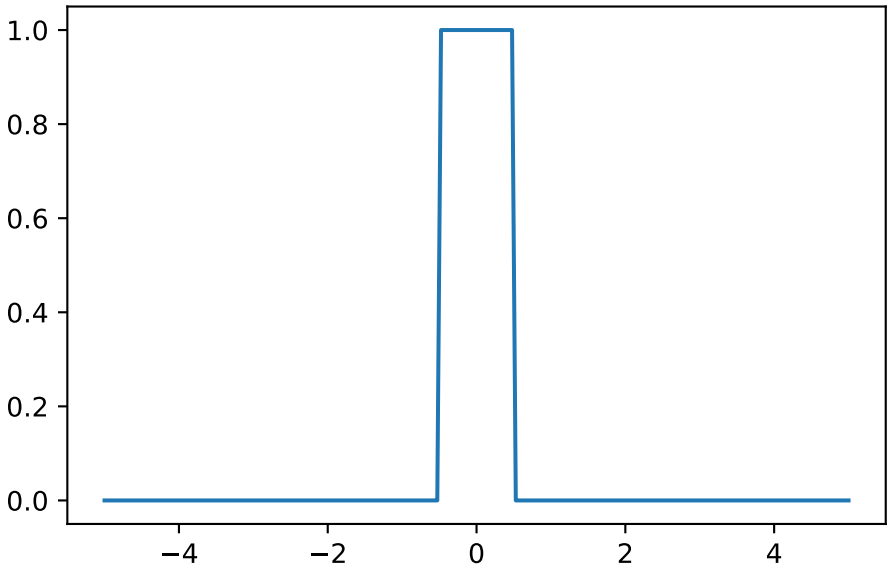


Figure 9.2: The Boxcar kernel centered at 0 with bandwidth $\alpha = 1$.

The diagram on the right is how the density curve for our 5 point dataset would have looked had we used the Boxcar kernel with bandwidth $\alpha = 1$.

9.1.3 Relationships Between Quantitative Variables

Up until now, we've discussed how to visualize single-variable distributions. Going beyond this, we want to understand the relationship between pairs of numerical variables.

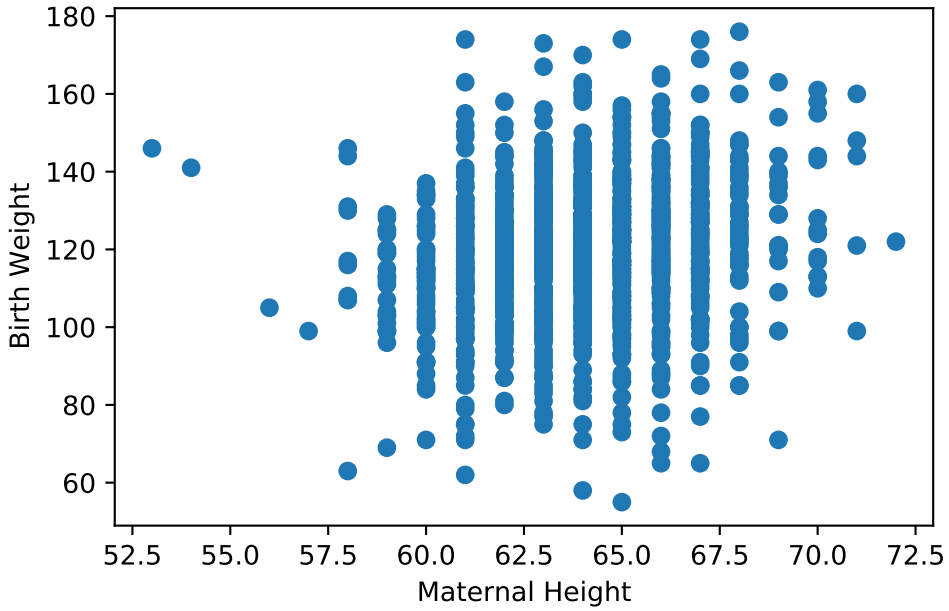
9.1.3.1 Scatter Plots

Scatter plots are one of the most useful tools in representing the relationship between two quantitative variables. They are particularly important in gauging the strength, or correla-

tion between variables. Knowledge of these relationships can then motivate decisions in our modeling process.

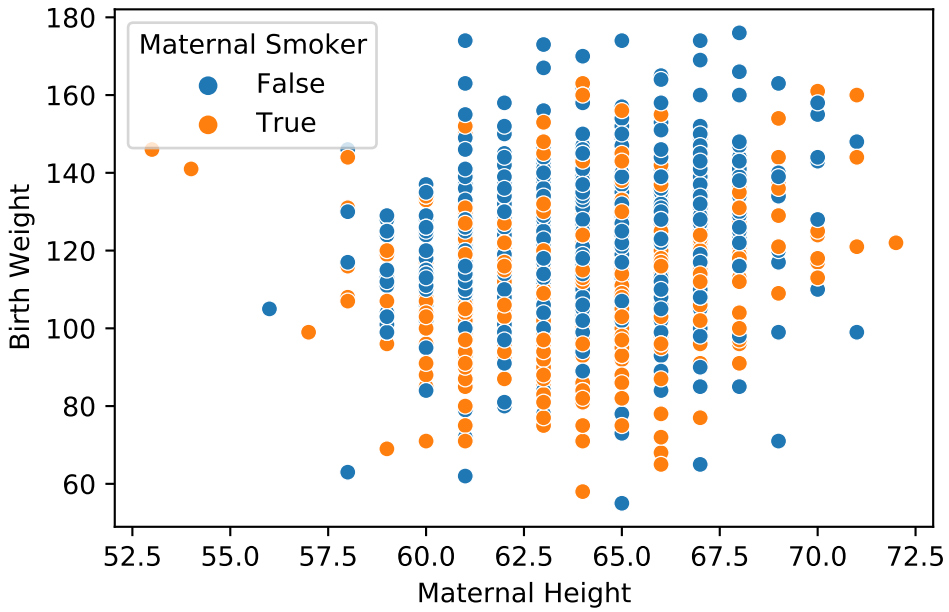
For example, let's plot a scatter plot comparing the `Maternal Height` and `Birth Weight` colums, using both `matplotlib` and `seaborn`.

```
import pandas as pd
births = pd.read_csv("data/baby.csv")
births.head(5)

# Matplotlib Example
plt.scatter(births['Maternal Height'], births['Birth Weight'])
plt.xlabel('Maternal Height')
plt.ylabel('Birth Weight');
```



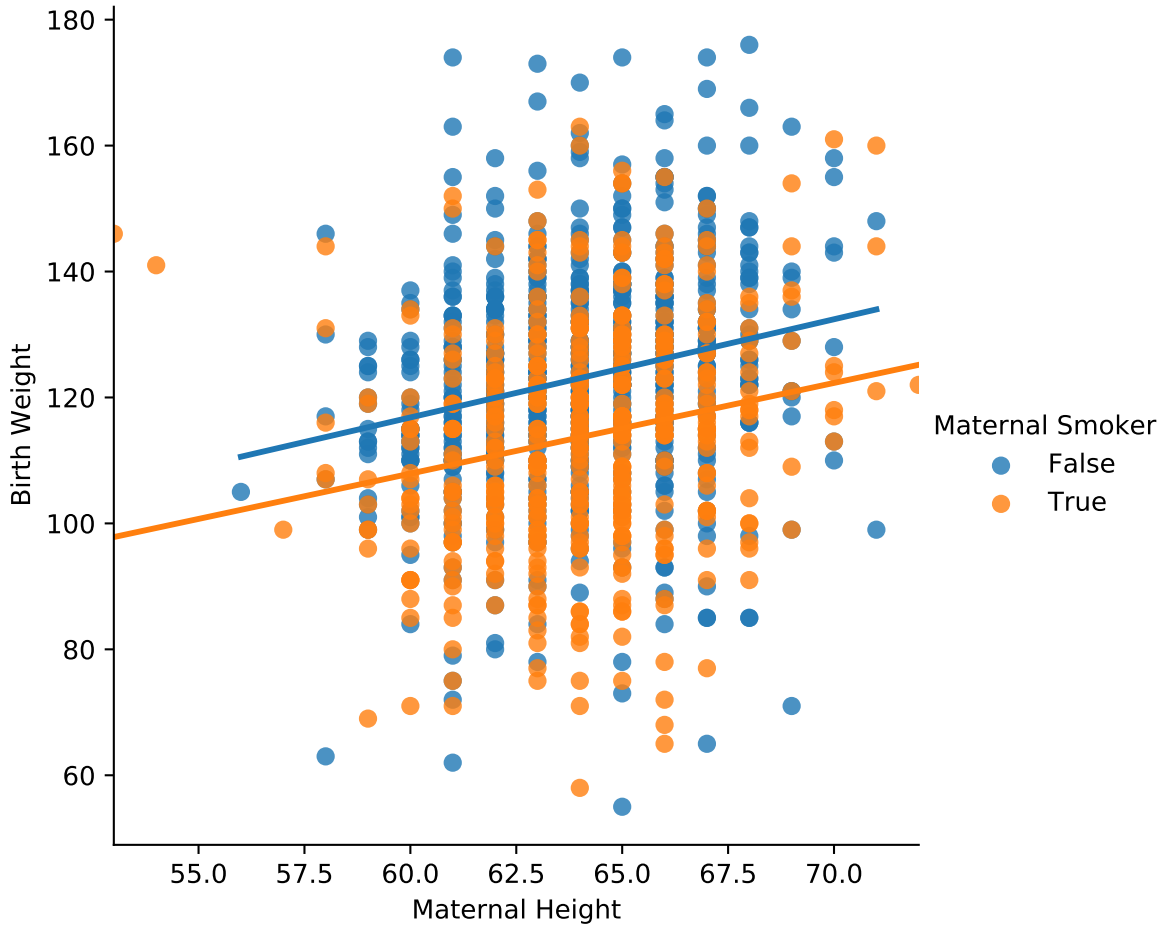

```
# Seaborn Example
sns.scatterplot(data = births, x = 'Maternal Height', y = 'Birth Weight',
                hue = 'Maternal Smoker')
```

```
<AxesSubplot:xlabel='Maternal Height', ylabel='Birth Weight'>
```



This is an example where color is used to add a third dimension to our plot. This is possible with the `hue` paramater in `seaborn`, which adds a categorical column encoding to an existing visualization. This way, we can look for relationships in `Maternal Height` and `Birth Weight` in both maternal smokers and non-smokers. If we wish to see the relationship's strength more clearly, we can use `sns.lmplot`.

```
sns.lmplot(data = births, x = 'Maternal Height', y = 'Birth Weight',
           hue = 'Maternal Smoker', ci = False);
```



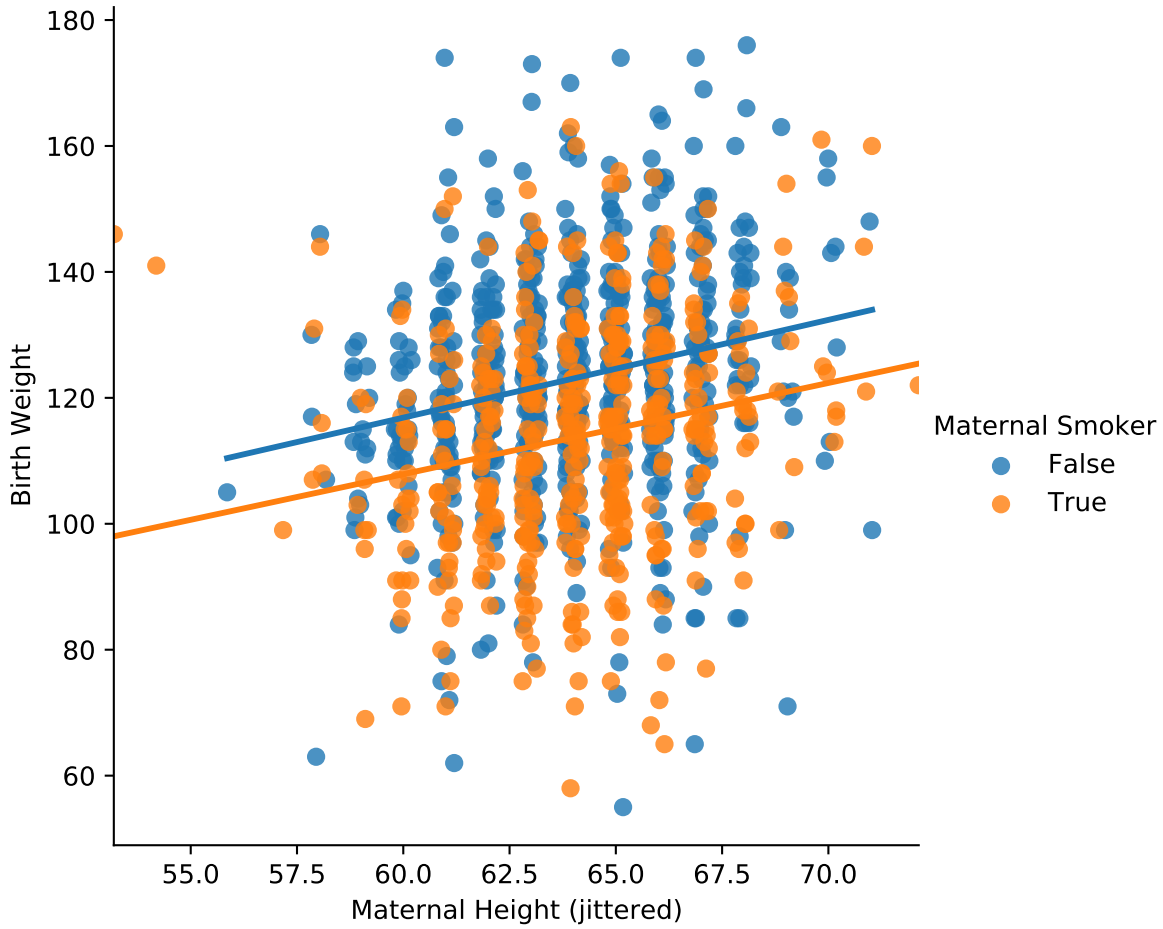
We can make out a weak, positive relationship in the mother's height and birth weight for both maternal smokers and non-smokers (the baseline is slightly lower in maternal smokers).

9.1.4 Overplotting

As you may have noticed, the scatterplots of `Maternal Height` vs. `Birth Weight` have many densely plotted areas. Many of the points are on top of one other! This makes it difficult to tell exactly how many babies are plotted in each the more densely populated regions of the graph. This can arise when the tools used for measuring data have low granularity, many different values are rounded to the same value, or if the ranges of the two variables differ greatly in scale.

We can overcome this by introducing a small amount of uniform random noise to our data. This is called *jittering*. Let's see what happens when we introduce noise to the `Maternal Height`.

```
births["Maternal Height (jittered)"] = births["Maternal Height"] + np.random.uniform(-0.2,
sns.lmplot(data = births, x = 'Maternal Height (jittered)', y = 'Birth Weight',
           hue = 'Maternal Smoker', ci = False);
```



This plot more clearly shows that most of the data is clustered tightly around the point (62.5,120) and gradually becomes more loose further away from the center. It is much easier for us and others to see how the data is distributed. In conclusion, *jittering* helps us better understand our own data (Goal 1) and communicate results to others (Goal 2).

9.1.4.1 Hex Plots and Contour Plots

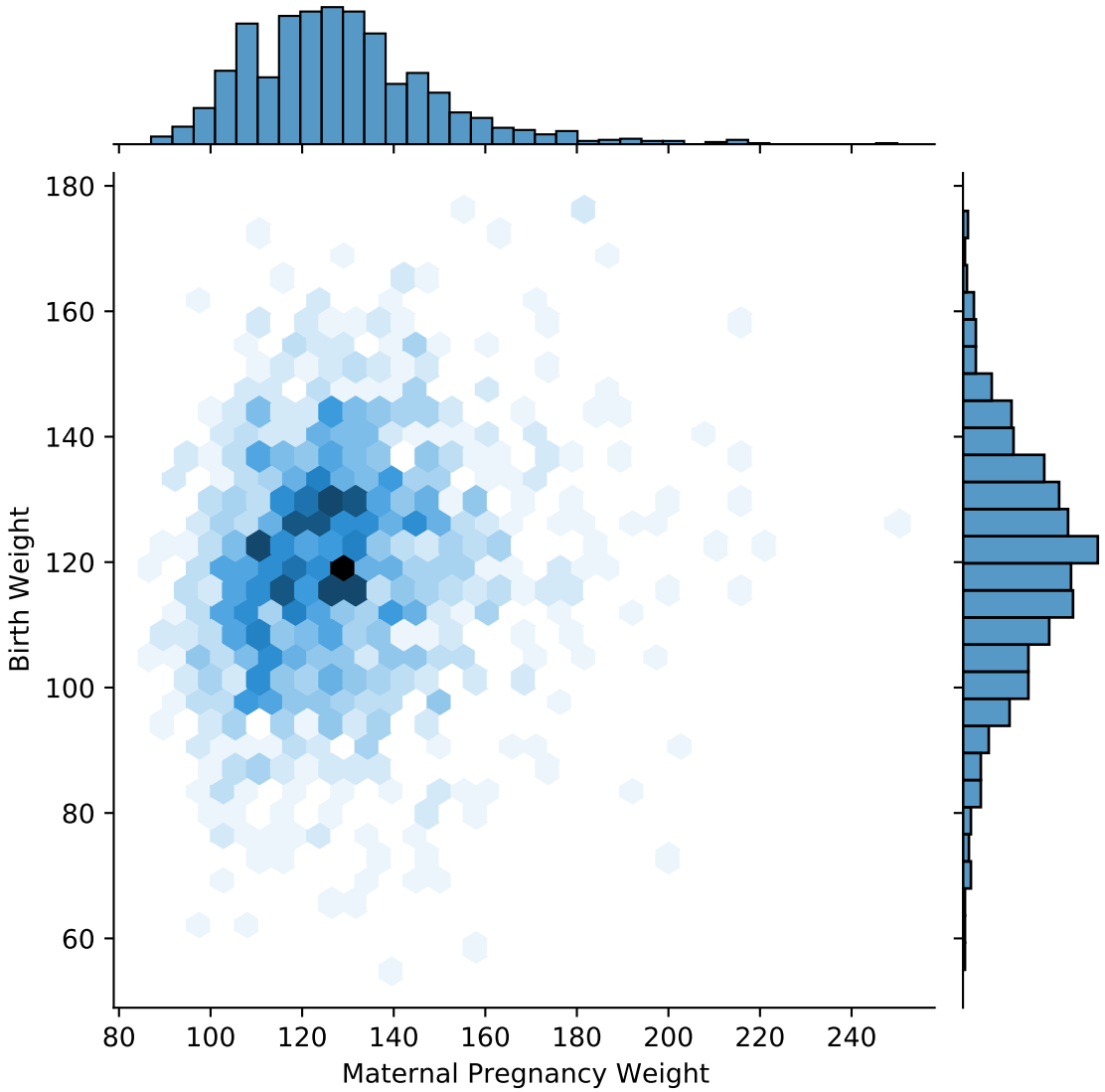
Unfortunately, our scatter plots above suffered from overplotting, which made them hard to interpret. And with a large number of points, jittering is unlikely to resolve the issue. Instead,

we can look to hex plots and contour plots.

Hex Plots can be thought of as a two dimensional histogram that shows the joint distribution between two variables. This is particularly useful working with very dense data.

```
sns.jointplot(data = births, x = 'Maternal Pregnancy Weight',
              y = 'Birth Weight', kind = 'hex')
```

```
<seaborn.axisgrid.JointGrid at 0x7fab89c4b310>
```

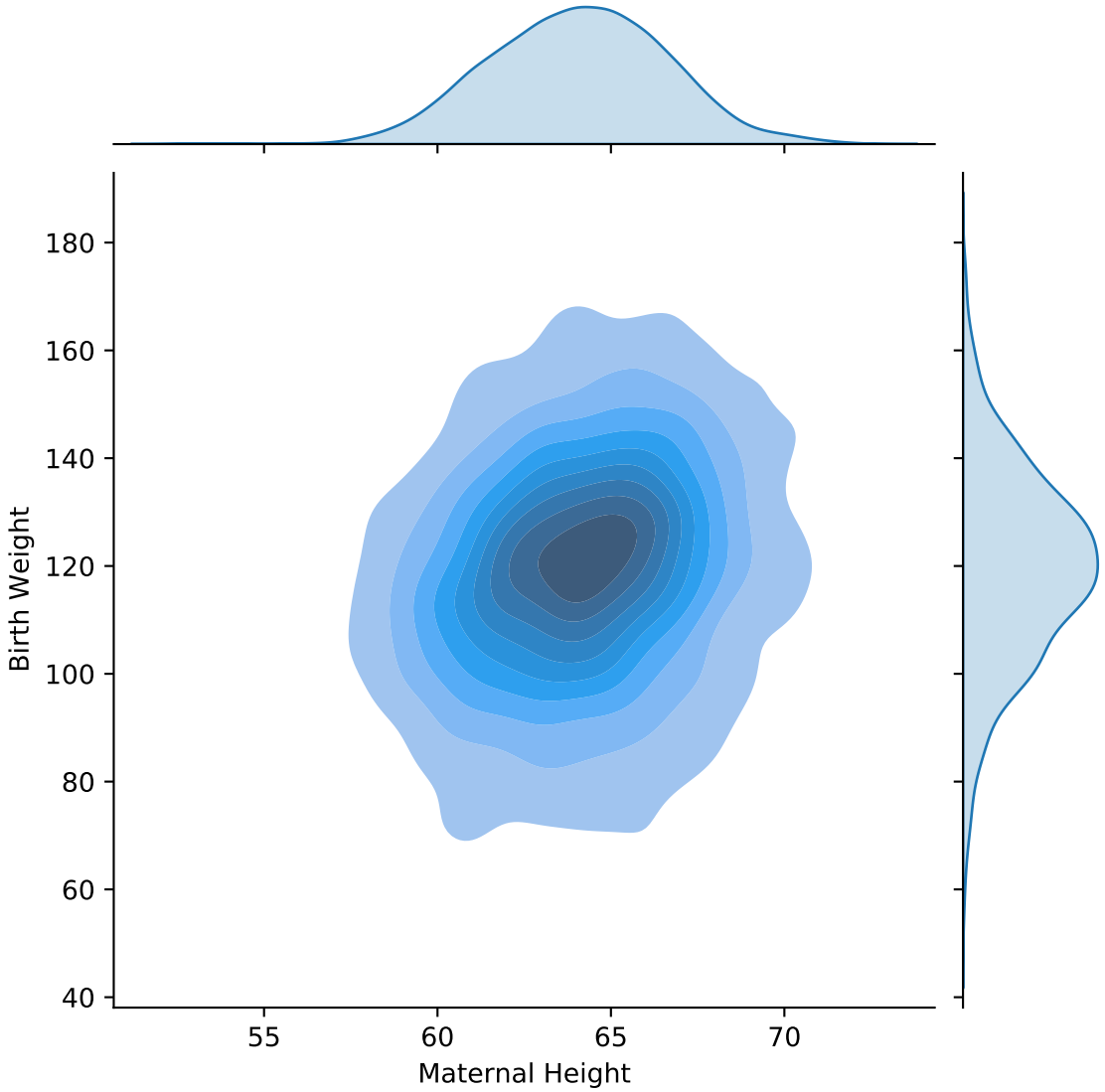


The axes are evidently binned into hexagons, which makes the linear relationship easier to decipher. Darker regions generally indicate a higher density of points.

On the other hand, **contour plots** are two dimensional versions of density curves with marginal distributions of each variable on the axes. We've used very similar code here to generate our contour plots, with the addition of the `kind = 'kde'` and `fill = True` arguments.

```
sns.jointplot(data = births, x = 'Maternal Height', y = 'Birth Weight',\
              kind = 'kde', fill = True)
```

```
<seaborn.axisgrid.JointGrid at 0x7fab89900d00>
```



9.2 Transformations

These last two lectures have covered visualizations in great depth. We looked at various forms of visualizations, plotting libraries, and high-level theory.

Much of this was done to uncover insights in data, which will prove necessary for the modeling process. A strong graphical correlation between two variables hinted an underlying relationship that has reason for further study. However, relying on visual relationships alone is limiting -

not all plots show association. The presence of outliers and other statistical anomalies make it hard to interpret data.

Transformations are the process of manipulating data to find significant relationships between variables. These are often found by applying mathematical functions to variables that "transform" their range of possible values and highlight some previously hidden associations between data.

9.2.0.1 Transforming a Distribution

When a distribution has a large dynamic range, it can be useful to take the logarithm of the data. For example, computing the logarithm of the ticket prices on the Titanic reduces skeweness and yields a distribution that is more "spread" across the x-axis. While it makes individual observations harder to interpret, the distribution is more favorable for subsequent analysis.

9.2.0.2 Linearizing a Relationship

Transformations are perhaps most useful to **linearize a relationship** between variables. If we find a transformation to make a scatter plot of two variables linear, we can "backtrack" to find the exact relationship between the variables. Linear relationships are particularly simple to interpret, and we'll be doing a lot of linear modeling in Data 100 - starting next week!

Say we want to understand the relationship between healthcare and life expectancy. Intuitively there should be a positive correlation, but upon plotting values from a dataset, we find a non-linear relationship that is somewhat hard to understand. However, applying a logarithmic transformation to both variables - healthcare and life expectancy - results in a scatter plot with a linear trend that we can interpret.

How can we find the relationship between the original variables? We know that taking a log of both axes gave us a linear relationship, so we can say (roughly) that

$$\log y = a \times \log x + b$$

Solving for y implies a **power** relationship in the original plot.

$$y = e^{a \times \log x + b}$$
$$y = Ce^{a \times \log x}$$
$$y = Cx^a$$

How did we know that taking the logarithm of both sides would result in a linear relationship? The **Tukey-Mosteller Bulge Diagram** is helpful here. We can use the direction of the buldge in our original data to find the appropriate transformations that will linearize the relationship. These transformations are found on axes that are nearest to the buldge. The buldge in our earlier example lay in Quadrant 2, so the transformations $\log x$, $\sqrt{x}$, y^2, or y^3 are possible contenders. It's important to note that this diagram is not perfect, and some transformations will work better than others. In our case, $\log x$ and $\log y$ (found in Quadrant 3) were the best.

9.2.0.3 Additional Remarks

Visualization requires a lot of thought! - There are many tools for visualizing distributions. - Distribution of a single variable: 1. rug plot 2. histogram 3. density plot 4. box plot 5. violin plot - Joint distribution of two quantitative variables: 1. scatter plot 2. hex plot 3. contour plot.

This class primarily uses `seaborn` and `matplotlib`, but `Pandas` also has basic built-in plotting methods. Many other visualization libraries exist, and `plotly` is one of them. - `plotly` creates very easily creates interactive plots. - `plotly` will occasionally appear in lecture code, labs, and assignments!

Next, we'll go deeper into the theory behind visualization.

9.3 Visualization Theory

This section marks a pivot to the second major topic of this lecture - visualization theory. We'll discuss the abstract nature of visualizations and analyze how they convey information.

Remember, we had two goals for visualizing data. This section is particularly important in:

1. Helping us understand the data and results
2. Communicating our results and conclusions with others

9.3.1 Information Channels

Visualizations are able to convey information through various encodings. In the remainder of this lecture, we'll look at the use of color, scale, and depth, to name a few.

9.3.1.1 Encodings in Rugplots

One detail that we may have overlooked in our earlier discussion of rugplots is the importance of encodings. Rugplots are effective visuals because they utilize line thickness to encode frequency. Consider the following diagram:

9.3.1.2 Multi-Dimensional Encodings

Encodings are also useful for representing multi-dimensional data. Notice how the following visual highlights four distinct "dimensions" of data:

- X-axis
- Y-axis
- Area
- Color

The human visual perception sytem is only capable of visualizing data in a three-dimensional plane, but as you've seen, we can encode many more channels of information.

9.3.2 Harnessing the Axes

9.3.2.1 Consider Scale of the Data

However, we should be careful to not misrepresent relationships in our data by manipulating the scale or axes. The visualization below improperly portrays two seemingly independent relationships on the same plot. The authors have clearly changed the scale of the y-axis to mislead their audience.

Notice how the downwards-facing line segment contains values in the millions, while the upwards-trending segment only contains values near three hundred thousand. These lines should not be intersecting.

When there is a large difference in the magnitude of the data, it's advised to analyze percentages instead of counts. The following diagrams correctly display the trends in cancer screening and abortion rates.

9.3.2.2 Reveal the Data

Great visualizations not only consider the scale of the data, but also utilize the axes in a way that best conveys information. For example, data scientists commonly set certain axes limits to highlight parts of the visualization they are most interested in.

The visualization on the right captures the trend in coronavirus cases during the month March in 2020. From only looking at the visualization on the left, a viewer may incorrectly believe that coronavirus began to skyrocket on March 4^{th}, 2020. However, the second illustration tells a different story - cases rose closer to March 21^{th}, 2020.

9.3.3 Harnessing Color

Color is another important feature in visualizations that does more than what meets the eye.

Last lecture, we used color to encode a categorical variable in our scatter plot. In this section, we will discuss uses of color in novel visualizations like colormaps and heatmaps.

5-8% of the world is red-green color blind, so we have to be very particular about our color scheme. We want to make these as accessible as possible. Choosing a set of colors which work together is evidently a challenging task!

9.3.3.1 Colormaps

Colormaps are mappings from pixel data to color values, and they're often used to highlight distinct parts of an image. Let's investigate a few properties of colormaps.

Jet Colormap

Viridis Colormap

The jet colormap is infamous for being misleading. While it seems more vibrant than viridis, the aggressive colors poorly encode numerical data. To understand why, let's analyze the following images.

The diagram on the left compares how a variety of colormaps represent pixel data that transitions from a high to low intensity. These include the jet colormap (row a) and grayscale (row b). Notice how the grayscale images do the best job in smoothly transitioning between pixel data. The jet colormap is the worst at this - the four images in row (a) look like a conglomeration of individual colors.

The difference is also evident in the images labeled (a) and (b) on the left side. The grayscale image is better at preserving finer detail in the vertical line strokes. Additionally, grayscale is preferred in x-ray scans for being more neutral. The intensity of dark red color in the jet colormap is frightening and indicates something is wrong.

Why is the jet colormap so much worse? The answer lies in how its color composition is percieved to the human eye.

Jet Colormap Perception

Viridis Colormap Perception

The jet colormap is largely misleading because it is not perceptually uniform. **Perceptually uniform colormaps** have the property that if the pixel data goes from 0.1 to 0.2, the perceptual change is the same as when the data goes from 0.8 to 0.9.

Notice how the said uniformity is present within the linear trend displayed in the viridis colormap. On the other hand, the jet colormap is largely non-linear - this is precisely why it's considered a worse colormap.

9.3.4 Harnessing Markings

In our earlier discussion of multi-dimensional encodings, we analyzed a scatter plot with four pseudo-dimensions: the two axes, area, and color. Were these appropriate to use? The following diagram analyzes how well the human eye can distinguish between these "markings".

There are a few key takeaways from this diagram

- Lengths are easy to discern. Don't use plots with jiggled baselines - keep everything axis-aligned.
- Avoid pie charts! Angle judgements are inaccurate.
- Areas and volumes are hard to distinguish (area charts, word clouds, etc)

9.3.5 Harnessing Conditioning

Conditioning is the process of comparing data that belong to seperate groups. We've seen this before in overlayed distributions, side-by-side box-plots, and scatter plots with categorical encodings. Here, we'll introduce terminology that formalizes these examples.

Consider an example where we want to analyze income earnings for male and females with varying levels of education. There are multiple ways to compare this data.

The barplot is an example of **juxtaposition**: placing multiple plots side by side, with the same scale. The scatter plot is an example of **superposition**: placing multiple density curves, scatter plots on top of each other.

Which is better depends on the problem at hand. Here, superposition makes the precise wage difference very clear from a quick glance. But many sophisticated plots convey information that favors the use of juxtaposition. Below is one example.

9.3.6 Harnessing Context

The last component to a great visualization is perhaps the most critical - the use of context. Adding informative titles, axis labels, and descriptive captions are all best practices that we've heard repeatedly in Data 8.

A publication-ready plot (and every Data 100 plot) needs:

- Informative title (takeaway, not description)
- Axis labels
- Reference lines, markers, etc
- Legends, if appropriate
- Captions that describe data

Captions should be:

- Comprehensive and self-contained
- Describe what has been graphed
- Draw attention to important features
- Describe conclusions drawn from graphs

10 Sampling

> **Note**
>
> - Understand how to appropriately collect data to help answer a question.

In Data Science, understanding characteristics of a population starts with having quality data to investigate. While it is often impossible to collect all the data describing a population, we can overcome this by properly sampling from the population. In this note, we will discuss appropriate techniques for sampling from populations.

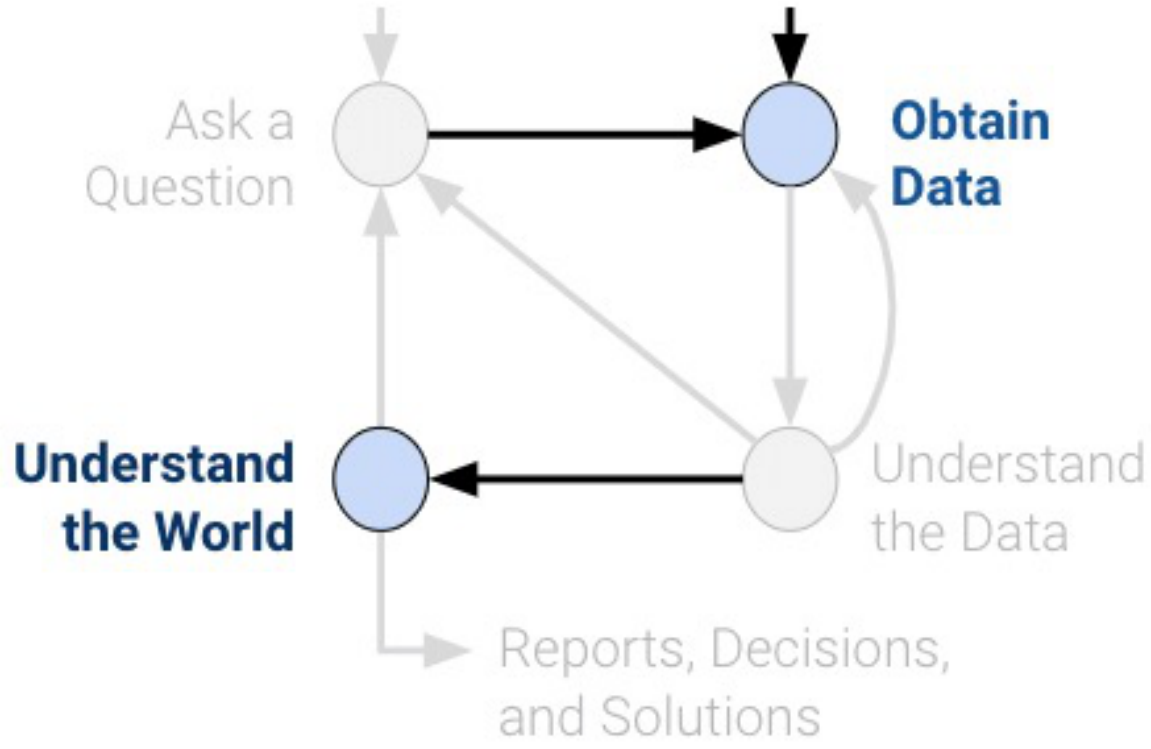


Figure 10.1: Lifecycle diagram

10.1 Censuses and Surveys

In general: a **census** is "an official count or survey of a population, typically recording various details of individuals."

- Example: The U.S. Decennial Census was held in April 2020, and it counts **every person** living in all 50 states, DC, and US territories. (Not just citizens.) Participation is required by law (it is mandated by the U.S. Constitution). Important uses include the allocation of Federal funds, congressional representation, and drawing congressional and state legislative districts. The census is composed of a **survey** mailed to different housing addresses in the United States.
- **Individuals** in a population are not always people. Other populations include: bacteria in your gut (sampled using DNA sequencing); trees of a certain species; small businesses receiving a microloan; or published results in an academic journal / field.

A **survey** is a set of questions. An example is workers sampling individuals and households. What is asked, and how it is asked, can affect how the respondent answers, or even whether the respondent answers in the first place.

While censuses are great, it is often difficult and expensive to survey everyone in a population. Thus, we usually survey a subset of the population instead.

A **sample** is often used to make inferences about the population. That being said, how the sample is drawn will affect the reliability of such inferences. Two common source of error in sampling are **chance error**, where random samples can vary from what is expected, in any direction; and **bias**, which is a a systematic error in one direction.

Because of how surveys and samples are drawn, it turns out that samples are usually—but not always—a subset of the population: * **Population**: The group that you want to learn something about. * **Sampling Frame**: The list from which the sample is drawn. For example, if sampling people, then the sampling frame is the set of all people that could possibly end up in your sample. * **Sample**: Who you actually end up sampling. The sample is therefore a subset of your *sampling frame*.

While ideally these three sets would be exactly the same, in practice they usually aren't. For example, there may be individuals in your sampling frame (and hence, your sample) that are not in your population. And generally, sample sizes are much smaller than population sizes.

10.2 Bias: A Case Study

The following case study is adapted from *Statistics* by Freedman, Pisani, and Purves, W.W. Norton NY, 1978.

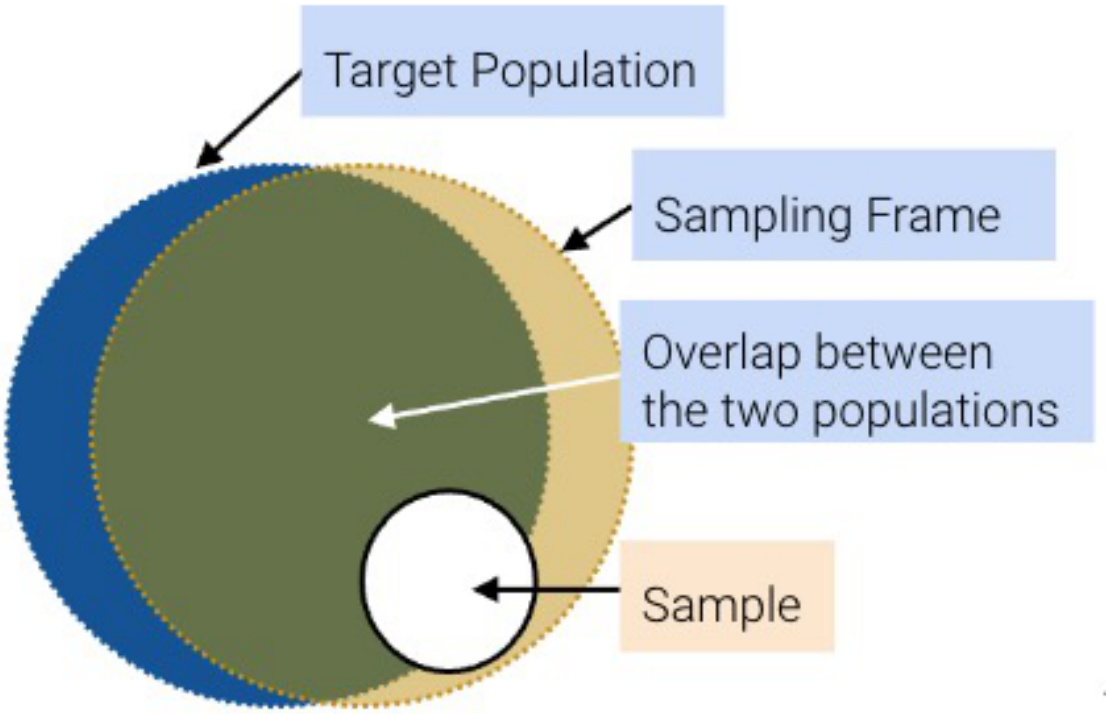


Figure 10.2: Sampling_Frames

In 1936, President Franklin D. Roosevelt (D) went up for re-election against Alf Landon (R) . As is usual, **polls** were conducted in the months leading up to the election to try and predict the outcome. The *Literary Digest* was a magazine that had successfully predicted the outcome of 5 general elections coming into 1936. In their polling for the 1936 election, they sent out their survey to 10 million individuals, who they found from phone books, lists of magazine subscribers, and lists of country club members. Of the roughly 2.4 million people who filled out the survey, only 43% reported they would vote for Roosevelt; thus the *Digest* predicted that Landon would win.

On election day, Roosevelt won in a landslide, winning 61% of the popular vote of about 45 million voters. How could the *Digest* have been so wrong with their polling?

It turns out that the *Literary Digest* sample was not representative of the population. Their sampling frame inherently skewed towards more affluent voters, who tended to vote Republican, and they completely overlooked the lion's share of voters who were still suffering through the Great Depression. Furthermore, they had a dismal response rate (about 24%); who knows how the other non-respondents would have polled? The *Digest* folded just 18 months after this disaster.

At the same time, George Gallup, a rising statistician, also made predictions about the 1936 elections. His estimate (56% Roosevelt) was much closer despite having a smaller sample size of "only" 50,000 (still more than necessary; more when we cover the Central Limit Theorem). Gallup also predicted the *Digest*'s prediction within 1%, with a sample size of only 3000 people. He did so by anticipating the *Digest*'s affluent sampling frame and subsampled those individuals. The **Gallup Poll** today is one of the leading polls for election results.

So what's the moral of the story? Samples, while convenient, are subject to chance error and **bias**. Election polling, in particular, can involve many sources of bias. To name a few: * **Selection bias** systematically excludes (or favors) particular groups. * **Response bias** occurs because people don't always respond truthfully. Survey designers pay special detail to the nature and wording of questions to avoid this type of bias. * **Non-response bias** occurs because people don't always respond to survey requests, which can skew responses. For example, the Gallup poll is conducted through landline phone calls, but many different populations in the U.S. do not pay for a landline, and still more do not always answer the phone. Surveyers address this bias by staying persistent and keeping surveys short.

10.3 Probability Samples

When sampling, it is essential to focus on the quality of the sample rather than the quantity of the sample. A huge sample size does not fix a bad sampling method. Our main goal is to gather a sample that is representative of the population it came from. The most common way to accomplish this is by randomly sampling from the population.

- A **convenience sample** is whatever you can get ahold of. Note that haphazard sampling is not necessarily random sampling; there are many potential sources of bias.
- In a **probability sample**, we know the chance any given set of individuals will be in the sample.
  - Probability samples allow us to estimate the bias and chance error, which helps us quantify uncertainty (more in a future lecture).
  - Note that this does not imply that all individuals in the population need have the same chance of being selected (see: stratified random samples).
  - Further note that the real world is usually more complicated. For example, we do not generally know the probability that a given bacterium is in a microbiome sample, or whether people will answer when Gallup calls landlines. That being said, we try to model probability sampling where possible if the sampling or measurement process is not fully under our control.

A few common random sampling schemes: * A **random sample with replacement** is a sample drawn **uniformly** at random **with** replacement. * Random doesn't always mean "uniformly at random," but in this specific context, it does. * Some individuals in the population might get picked more than once

- A **simple random sample (SRS)** is a sample drawn uniformly at random without replacement.
  - Every individual (and subset of individuals) has the same chance of being selected.
  - Every pair has the same chance as every other pair.
  - Every triple has the same chance as every other triple.
  - And so on.
- A **stratified random sample**, where random sampling is performed on strata (specific groups), and the groups together compose a sample.

10.3.1 Example: Stratified random sample

Suppose that we are trying to run a poll to predict the mayoral election in Bearkeley City (an imaginary city that neighbors Berkeley). Suppose we try a **stratified random sample** to select 100 voters as follows: 1. First, we take a simple random sample and obtain 50 voters that are above the median city income ("above-median-income"), i.e., in the upper 50-th percentile of income in the city. 2. We then take a simple random sample of the other 50 from voters that are below the median city income.

This is a **probability sample**: For any group of 100 people, if there are not exactly 50 "above-median-income" voters, then that group has zero probability of being chosen. For any other group (which has exactly 50 "above-median-income" voters), then the chance of it being chosen is 1/ # of such groups.

Note that even if we replace the group counts with 80/20 (80 "above-median-income" voters, 20 others), then it is still a probability sample, because we can compute the precise probability of each group being chosen. However, the sampling scheme (and thus the modeling of voter preferences) becomes biased towards voters with income above the median.

10.4 Approximating Simple Random Sampling

The following is a very common situation in data science: - We have an enormous population. - We can only afford to sample a relatively small number of individuals. If the population is huge compared to the sample, then random sampling with and without replacement are pretty much the same.

Example : Suppose there are 10,000 people in a population. Exactly 7,500 of them like Snack 1; the other 2,500 like Snack 2. What is the probability that in a random sample of 20, all people like Snack 1?

- Method 1: SRS (Random Sample Without Replacement): $\prod_{k=0}^{19} \frac{7500-k}{10000-k} \approx 0.003151$
- Method 2: Random Sample with Replacement: $(0.75)^{20} \approx 0.003171$

As seen here, when the population size is large, probabilities of sampling with replacement are much easier to compute and lead to a reasonable approximation.

10.4.1 Multinomial Probabilities

The approximation discussed above suggests the convenience of **multinomial probabilities**, which arise from sampling a categorical distribution at random **with replacement*.

Suppose that we have a bag of marbles with the following distribution: 60% are blue, 30% are green, and 10% are red. If we then proceed to draw 100 marbles from this bag, at random with replacement, then the resulting 100-size sample is modeled as a multinomial distribution using `np.random.multinomial`:

```
import numpy as np
np.random.multinomial(100, [0.60, 0.30, 0.10])
```

```
array([61, 27, 12])
```

This method allows us to generate, say, 10 samples of size 100 using the `size` parameter:

```
np.random.multinomial(100, [0.60, 0.30, 0.10], size=10)
```

```
array([[56, 34, 10],
       [64, 22, 14],
       [58, 36,  6],
       [60, 28, 12],
       [54, 36, 10],
       [64, 29,  7],
       [52, 38, 10],
       [70, 19, 11],
       [62, 32,  6],
       [57, 31, 12]])
```

10.5 Comparing Convenience Sample and SRS

Suppose that we are trying to run a poll to predict the mayoral election in Bearkeley City (an imaginary city that neighbors Berkeley). Suppose we took a sample to predict the election outcome by polling all retirees. Even if they answer truthfully, we have a **convenience sample**. How biased would this sample be in predicting the results? While we will not numerically quantify the bias, in this demo we'll visually show that because of the voter population distribution, any error in our prediction from a retiree sample cannot be simply due to chance:

First, let's grab a data set that has every single voter in the Bearkeley (again, this is a fake dataset) and how they **actually** voted in the election. For the purposes of this example, assume: * "high income" indicates a voter is above the median household income, which is $97,834 (actual Berkeley number). * There are only two mayoral candidates: one Democrat and one Republican. * Every registered voter votes in the election for the candidate under their registered party (Dem or Rep).

```
import pandas as pd
import numpy as np
bearkeley = pd.read_csv("data/bearkeley.csv")

# create a 1/0 int that indicates democratic vote
bearkeley['vote.dem'] = (bearkeley['vote'] == 'Dem').astype(int)
bearkeley.head()
```

	age	high_income	vote	vote.dem
0	35	False	Dem	1
1	42	True	Rep	0
2	55	False	Dem	1
3	77	True	Rep	0
4	31	False	Dem	1

```
bearkeley.shape
```

```
(1300000, 4)
```

```
actual_vote = np.mean(bearkeley["vote.dem"])
actual_vote
```

```
0.5302792307692308
```

This is the **actual outcome** of the election. Based on this result, the Democratic candidate would win. However, if we were to only consider retiree voters (a retired person is anyone age 65 and up):

```
convenience_sample = bearkeley[bearkeley['age'] >= 65]
np.mean(convenience_sample["vote.dem"])
```

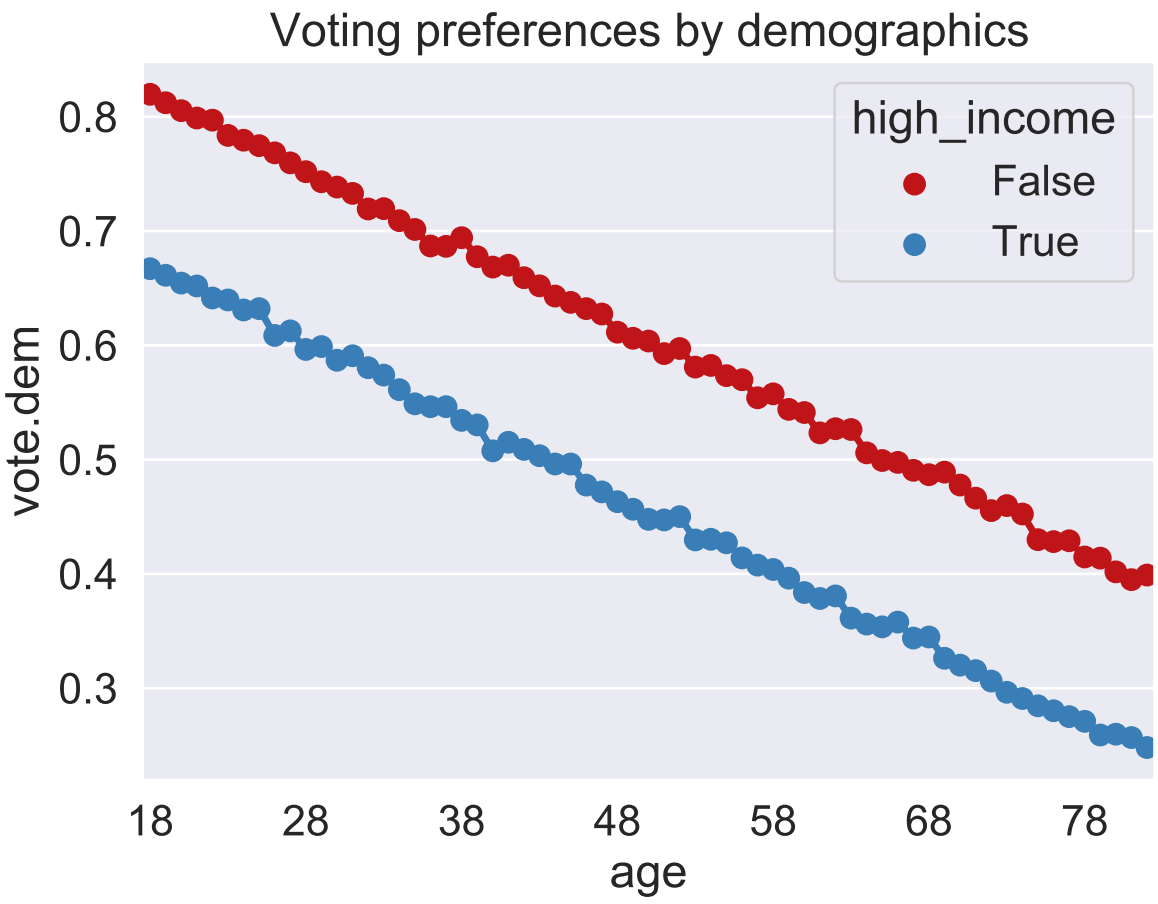
```
0.3744755089093924
```

Based on this result, we would have predicted that the Republican candidate would win! This error is not due to the sample being too small to yield accurate predictions, because there are 359,396 retirees (about 27% of the 1.3 million Bearkeley voters). Instead, there seems to be something larger happening. Let's visualize the voter preferences of the entire population to see how retirees trend:

Let us aggregate all voters by age and visualize the fraction of Democratic voters, split by income.

```
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

sns.set_theme(style='darkgrid', font_scale = 1.5,
```

```
            rc={'figure.figsize':(7,5)})

# aggregate all voters by age
votes_by_demo = bearkeley.groupby(["age","high_income"]).agg("mean").reset_index()

fig = plt.figure();
red_blue = ["#bf1518", "#397eb7"]
with sns.color_palette(sns.color_palette(red_blue)):
    ax = sns.pointplot(data=votes_by_demo, x = "age", y = "vote.dem", hue = "high_income")

ax.set_title("Voting preferences by demographics")
fig.canvas.draw()
new_ticks = [i.get_text() for i in ax.get_xticklabels()];
plt.xticks(range(0, len(new_ticks), 10), new_ticks[::10]);
```



From the plot above, we see that retirees in the imaginary city of Bearkeley tend to vote less Democrat, which skewed our predictions from our sample. We also note that high-income voters tend to vote less Democrat (and more Republican).

Let's compare our biased convenience sample to a simple random sample. Supposing we took an SRS the same size as our retiree sample, we see that we get a result very close to the actual vote:

```
## By default, replace = False
n = len(convenience_sample)
random_sample = bearkeley.sample(n, replace = False)

np.mean(random_sample["vote.dem"])
```

```
0.5309686251377311
```

This is very close to the actual vote!

We could even get pretty close with a *much smaller sample size*, say 800:

It turns out that we are pretty close, **much smaller sample size**, say, 800 (we'll learn how to choose this number when we introduce the Central Limit Theorem):

```
n = 800
random_sample = bearkeley.sample(n, replace = False)
np.mean(random_sample["vote.dem"])
```

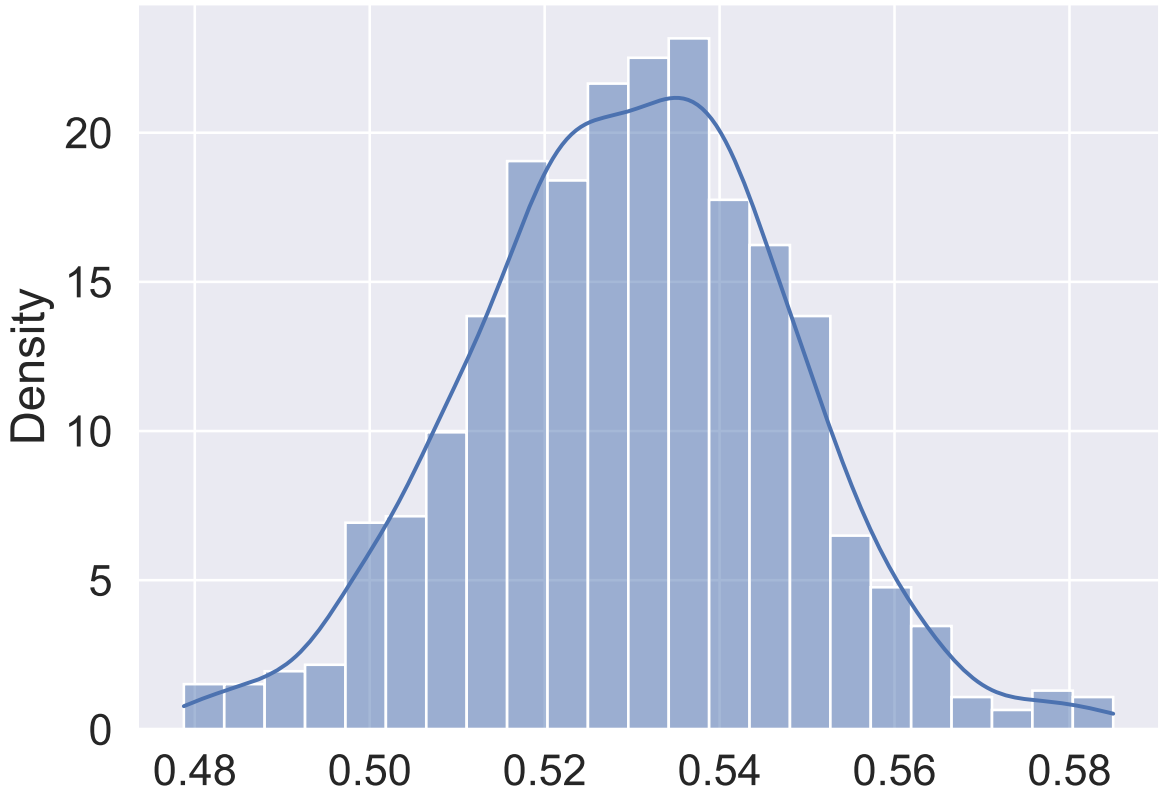
```
0.53
```

To visualize the chance error in an SRS, let's simulate 1000 samples of the 800-size Simple Random Sample:

```
poll_result = []
nrep = 1000   # number of simulations
n = 800       # size of our sample
for i in range(0,nrep):
    random_sample = bearkeley.sample(n, replace = False)
    poll_result.append(np.mean(random_sample["vote.dem"]))
sns.histplot(poll_result, stat='density', kde=True)

# What fraction of these simulated samples would have predicted Democrat?
poll_result = pd.Series(poll_result)
```

```
np.sum(poll_result >= 0.5)/1000
```

```
0.95
```



A few observations: First, the KDE looks roughly Gaussian. Second, supposing that we predicted a Democratic winner if 50% of our sample voted Democrat, then just about 4% of our simulated samples would have predicted the election result incorrectly. This visualization further justifies why our convenience sample had error that was not entirely just due to chance. We'll revisit this notion later in the course.

10.6 Summary

Understanding the sampling process is what lets us go from describing the data to understanding the world. Without knowing / assuming something about how the data were collected, there is no connection between the sample and the population. Ultimately, the dataset doesn't tell us about the world behind the data.

11 Introduction to Modeling

> **Note**
>
> - Understand what models are and how to carry out the four-step modeling process
> - Define the concept of loss and gain familiarity with L1 and L2 loss
> - Fit a model using minimization techniques

Up until this point in the semester, we've focused on analyzing datasets. We've looked into the early stages of the data science lifecycle, focusing on the programming tools, visualization techniques, and data cleaning methods needed for data analysis.

This lecture marks a shift in focus. We will move away from examining datasets to actually *using* our data to better understand the world. Specifically, the next sequence of lectures will explore predictive modeling: generating models to make some prediction about the world around us. In this lecture, we'll introduce the conceptual framework for setting up a modeling task. In the next few lectures, we'll put this framework into practice by implementing several kinds of models.

11.1 What is a Model?

A model is an **idealized representation** of a system. A system is a set of principles or procedures according to which something functions. We live in a world full of systems: the procedure of turning on a light happens according to a specific set of rules dictating the flow of electricity. The truth behind how any event occurs are usually complex, and many times the specifics are unknown. The workings of the world can be viewed is its own giant procedure. Models seek to simplify the world and distill them it into workable pieces.

Example: We model the fall of an object on Earth as subject to a constant acceleration of $9.81 \frac{m}{s^2}$ due to gravity.

- While this describes the behavior of our system, it is merely an approximation.
- It doesn't account for the effects of air resistance, local variations in gravity, etc.
- In practice, it's accurate enough to be useful!

11.1.1 Reasons for building models

Often times, (1) we care about creating models that are simple and interpretable, allowing us to understand what the relationships between our variables are. Other times, (2) we care more about making extremely accurate predictions, at the cost of having an uninterpretable model. These are sometimes called black-box models, and are common in fields like deep learning.

1. To understand complex phenomena occurring in the world we live in.
    - What factors play a role in the growth of COVID-19?
    - How do an object's velocity and acceleration impact how far it travels? (Physics: $d = d_0 + vt + \frac{1}{2}at^2$)
2. To make accurate predictions about unseen data.
    - Can we predict if an email is spam or not?
    - Can we generate a one-sentence summary of this 10-page long article?

11.1.2 Common Types of Models

In general, models can be split into two categories:

Note: These specific models are not in the scope of Data 100 and exist to serve as motivation.

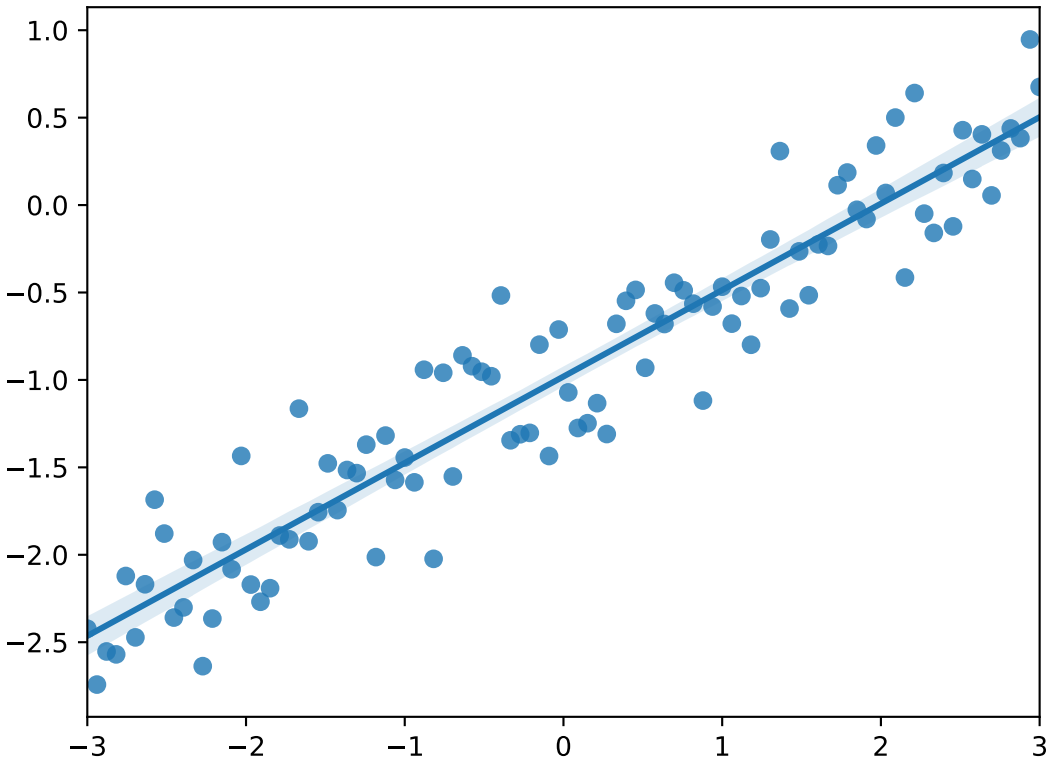
1. Deterministic physical (mechanistic) models: Laws that govern how the world works.
    - Kepler's Third Law of Planetary Motion (1619): The ratio of the square of an object's orbital period with the cube of the semi-major axis of its orbit is the same for all objects orbiting the same primary.
        - $T^2 \propto R^3$
    - Newton's Laws: motion and gravitation (1687): Newton's second law of motion models the relationship between the mass of an object and the force required to accelerate it.
        - $F = ma$
        - $F_g = G\frac{m_1 m_2}{r^2}$
2. Probabilistic models: models that attempt to understand how random processes evolve. These are more general and can be used describe many phenomena in the real world. These models commonly make simplifying assumption about the nature of the world.
    - Poisson Process models: Used to model random events that can happen with some probability at any point in time and are strictly increasing in count, such as the arrival of customers at a store.

11.2 Simple Linear Regression

The **regression line** is the unique straight line that minimizes the **mean squared error** of estimation among all straight lines. As with any straight line, it can be defined by a slope and a y-intercept:

- slope: $r \cdot \frac{\text{Standard Deviation of y}}{\text{Standard Deviation of x}}$
- y-intercept: average of y $-$ slope $\cdot$ average of x

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# Set random seed for consistency
np.random.seed(43)
plt.style.use('default')

#Generate random noise for plotting
x = np.linspace(-3, 3, 100)
y = x * 0.5 - 1 + np.random.randn(100) * 0.3

#plot regression line
sns.regplot(x=x,y=y);
```



11.2.1 Definitions

For a random variable x:

- Mean: $\bar{x}$
- Standard Deviation: σ_x
- Predicted value: $\hat{x}$

11.2.1.1 Standard Units

A random variable is represented in standard units if the following are true:

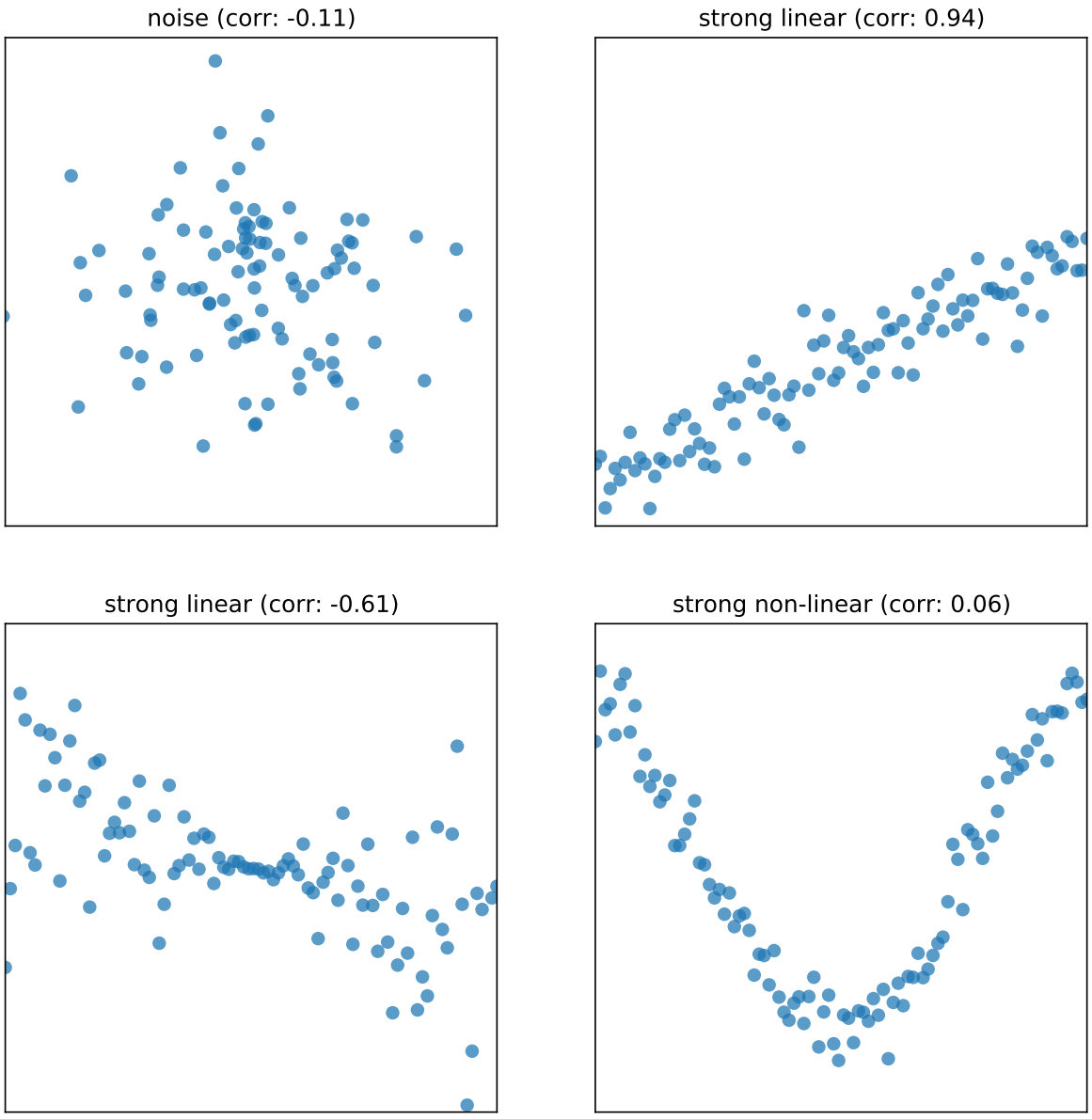
1. 0 in standard units is the mean ($\bar{x}$) in the original variable's units.
2. An increase of 1 standard unit is an increase of 1 standard deviation(σ_x) in the original variable's units

11.2.1.2 Correlation

The correlation (r) is the average of the product of x and y, both measured in *standard units*. Correlation measures the strength of a linear association between two variables.

1. $r = \frac{1}{n}\sum_1^n(\frac{x_i-\bar{x}}{\sigma_x})(\frac{y_i-\bar{y}}{\sigma_y})$
2. Correlations are between -1 and 1: $|r| < 1$

```
def plot_and_get_corr(ax, x, y, title):
    ax.set_xlim(-3, 3)
    ax.set_ylim(-3, 3)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.scatter(x, y, alpha = 0.73)
    r = np.corrcoef(x, y)[0, 1]
    ax.set_title(title + " (corr: {})".format(r.round(2)))
    return r

fig, axs = plt.subplots(2, 2, figsize = (10, 10))

# Just noise
x1, y1 = np.random.randn(2, 100)
corr1 = plot_and_get_corr(axs[0, 0], x1, y1, title = "noise")

# Strong linear
x2 = np.linspace(-3, 3, 100)
y2 = x2 * 0.5 - 1 + np.random.randn(100) * 0.3
corr2 = plot_and_get_corr(axs[0, 1], x2, y2, title = "strong linear")

# Unequal spread
x3 = np.linspace(-3, 3, 100)
y3 = - x3/3 + np.random.randn(100)*(x3)/2.5
corr3 = plot_and_get_corr(axs[1, 0], x3, y3, title = "strong linear")
extent = axs[1, 0].get_window_extent().transformed(fig.dpi_scale_trans.inverted())

# Strong non-linear
x4 = np.linspace(-3, 3, 100)
y4 = 2*np.sin(x3 - 1.5) + np.random.randn(100) * 0.3
corr4 = plot_and_get_corr(axs[1, 1], x4, y4, title = "strong non-linear")

plt.show()
```



11.2.2 Alternate Form

When the variables y and x are measured in *standard units*, the regression line for predicting y based on x has slope r and passes through the origin.

$$\hat{y}_{su} = r \cdot x_{su}$$

- In the original units, this becomes

$$\frac{\hat{y} - \bar{y}}{\sigma_y} = r \cdot \frac{x - \bar{x}}{\sigma_x}$$

11.2.3 Derivation

Starting from the top, we have our claimed form of the regression line and we want to show that its equivalent to the optimal linear regression line: $\hat{y} = \hat{a} + \hat{b}x$

Recall:

- $\hat{b}$: $r \cdot \frac{\text{Standard Deviation of y}}{\text{Standard Deviation of x}}$
- $\hat{a}$: average of y $-$ slope $\cdot$ average of x

Proof:

$$\frac{\hat{y} - \bar{y}}{\sigma_y} = r \cdot \frac{x - \bar{x}}{\sigma_x}$$

Multiply by σ_y and add $\bar{y}$ on both sides.

$$\hat{y} = \sigma_y \cdot r \cdot \frac{x - \bar{x}}{\sigma_x} + \bar{y}$$

Distribute coefficient $\sigma_y \cdot r$ to the $\frac{x - \bar{x}}{\sigma_x}$ term

$$\hat{y} = (\frac{r\sigma_y}{\sigma_x}) \cdot x + (\bar{y} - (\frac{r\sigma_y}{\sigma_x})\bar{x})$$

We now see that we have a line that matches our claim:

- slope: $r \cdot \frac{\text{SD of x}}{\text{SD of y}} = r \cdot \frac{\sigma_x}{\sigma_y}$
- intercept: $\bar{y} - \text{slope} \cdot x$

11.3 The Modeling Process

At a high level, a model is some way of representing a system. In Data 100, we'll treat a model as some mathematical rule we use to describe the relationship between variables.

What variables are we modeling? Typically, we use a subset of the variables in our sample of collected data to model another variable in this data. To put this more formally, say we have the following dataset $\mathbb{D}$:

$$\mathbb{D} = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$$

Each pair of values (x_i, y_i) represents a datapoint. In a modeling setting, we call these **observations**. y_i is the dependent variable we are trying to model, also called an **output** or **response**. x_i is the independent variable inputted into the model to make predictions, also known as a **feature**.

Our goal in modeling is to use the observed data $\mathbb{D}$ to predict the output variable y_i. We denote each prediction as $\hat{y}_i$ (read: "y hat sub i").

How do we generate these predictions? Some examples of models we'll encounter in the next few lectures are given below:

$$\hat{y}_i = \theta$$
$$\hat{y}_i = \theta_0 + \theta_1 x_i$$

The examples above are known as **parametric models**. They relate the collected data, x_i, to the prediction we make, $\hat{y}_i$. A few parameters (θ, θ_0, θ_1) are used to describe the relationship between x_i and $\hat{y}_i$.

Notice that we don't immediately know the values of these parameters. While the features, x_i, are taken from our observed data, we need to decide what values to give θ, θ_0, and θ_1 ourselves. This is the heart of parametric modeling: *what parameter values should we choose so our model makes the best possible predictions?*

To choose our model parameters, we'll work through the **modeling process**.

1. Choose a model: how should we represent the world?
2. Choose a loss function: how do we quantify prediction error?
3. Fit the model: how do we choose the best parameters of our model given our data?
4. Evaluate model performance: how do we evaluate whether this process gave rise to a good model?

11.4 Choosing a Model

Our first step is choosing a model: defining the mathematical rule that describes the relationship between the features, x_i, and predictions $\hat{y}_i$.

In Data 8, you learned about the **Simple Linear Regression (SLR) model**. You learned that the model takes the form:

$$\hat{y}_i = a + bx_i$$

In Data 100, we'll use slightly different notation: we will replace a with θ_0 and b with θ_1. This will allow us to use the same notation when we explore more complex models later on in the course.

$$\hat{y}_i = \theta_0 + \theta_1 x_i$$

The parameters of the SLR model are θ_0, also called the intercept term, and θ_1, also called the slope term. To create an effective model, we want to choose values for θ_0 and θ_1 that most accurately predict the output variable. The "best" fitting model parameters are given the special names $\hat{\theta}_0$ and $\hat{\theta}_1$ – they are the specific parameter values that allow our model to generate the best possible predictions.

In Data 8, you learned that the best SLR model parameters are:

$$\hat{\theta}_0 = \bar{y} - \hat{\theta}_1 \bar{x} \qquad\qquad \hat{\theta}_1 = r\frac{\sigma_y}{\sigma_x}$$

A quick reminder on notation:

- $\bar{y}$ and $\bar{x}$ indicate the mean value of y and x, respectively
- σ_y and σ_x indicate the standard deviations of y and x
- r is the correlation coefficient, defined as the average of the product of x and y measured in standard units: $\frac{1}{n}\sum_{i=1}^{n}(\frac{x_i-\bar{x}}{\sigma_x})(\frac{y_i-\bar{y}}{\sigma_y})$

In Data 100, we want to understand *how* to derive these best model coefficients. To do so, we'll introduce the concept of a loss function.

11.5 Choosing a Loss Function

We've talked about the idea of creating the "best" possible predictions. This begs the question: how do we decide how "good" or "bad" our model's predictions are?

A **loss function** characterizes the cost, error, or fit resulting from a particular choice of model or model parameters. This function, $L(y, \hat{y})$, quantifies how "far off" a single prediction by our model is from a true, observed value in our collected data.

The choice of loss function for a particular model depends on the modeling task at hand. Regardless of the specific function used, a loss function should follow two basic principles:

- If the prediction $\hat{y}_i$ is *close* to the actual value y_i, loss should be low
- If the prediction $\hat{y}_i$ is *far* from the actual value y_i, loss should be high

Two common choices of loss function are squared loss and absolute loss.

Squared loss, also known as **L2 loss**, computes loss as the square of the difference between the observed y_i and predicted $\hat{y}_i$:

$$L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$

Absolute loss, also known as **L1 loss**, computes loss as the absolute difference between the observed y_i and predicted $\hat{y}_i$:

$$L(y_i, \hat{y}_i) = |y_i - \hat{y}_i|$$

L1 and L2 loss give us a tool for quantifying our model's performance on a single datapoint. This is a good start, but ideally we want to understand how our model performs across our *entire* dataset. A natural way to do this is to compute the average loss across all datapoints in the dataset. This is known as the **cost function**, $\hat{R}(\theta)$:

$$\hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i)$$

The cost function has many names in statistics literature. You may also encounter the terms:

- Empirical risk (this is why we give the cost function the name R)
- Error function
- Average loss

We can substitute our L1 and L2 loss into the cost function definition. The **Mean Squared Error (MSE)** is the average squared loss across a dataset:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The **Mean Absolute Error (MAE)** is the average absolute loss across a dataset:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

11.6 Fitting the Model

Now that we've established the concept of a loss function, we can return to our original goal of choosing model parameters. Specifically, we want to choose the best set of model parameters that will minimize the model's cost on our dataset. This process is called fitting the model.

We know from calculus that a function is minimized when (1) its first derivative is equal to zero and (2) its second derivative is positive. We often call the function being minimized the **objective function** (our objective is to find its minimum).

To find the optimal model parameter, we:

1. Take the derivative of the cost function with respect to that parameter
2. Set the derivative equal to 0
3. Solve for the parameter

We repeat this process for each parameter present in the model. For now, we'll disregard the second derivative condition.

To help us make sense of this process, let's put it into action by deriving the optimal model parameters for simple linear regression using the mean squared error as our cost function. Remember: although the notation may look tricky, all we are doing is following the three steps above!

Step 1: take the derivative of the cost function with respect to each model parameter. We substitute the SLR model, $\hat{y}_i = \theta_0 + \theta_1 x_i$, into the definition of MSE above and differentiate with respect to θ_0 and θ_1.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)^2$$

$$\frac{\partial}{\partial \theta_0}\text{MSE} = \frac{-2}{n}\sum_{i=1}^{n} y_i - \theta_0 - \theta_1 x_i$$

$$\frac{\partial}{\partial \theta_1}\text{MSE} = \frac{-2}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)x_i$$

Step 2: set the derivatives equal to 0. After simplifying terms, this produces two **estimating equations**. The best set of model parameters (θ_0, θ_1) *must* satisfy these two optimality conditions.

$$0 = \frac{-2}{n}\sum_{i=1}^{n} y_i - \theta_0 - \theta_1 x_i \iff \frac{1}{n}\sum_{i=1}^{n} y_i - \hat{y}_i = 0$$

$$0 = \frac{-2}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)x_i \iff \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)x_i = 0$$

Step 3: solve the estimating equations to compute estimates for $\hat{\theta}_0$ and $\hat{\theta}_1$.

Taking the first equation gives the estimate of $\hat{\theta}_0$:

$$\frac{1}{n}\sum_{i=1}^{n} y_i - \hat{\theta}_0 - \hat{\theta}_1 x_i = 0 \left(\frac{1}{n}\sum_{i=1}^{n} y_i\right) - \hat{\theta}_0 - \hat{\theta}_1 \left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) = 0\hat{\theta}_0 = \bar{y} - \hat{\theta}_1 \bar{x}$$

With a bit more maneuvering, the second equation gives the estimate of $\hat{\theta}_1$. Start by multiplying the first estimating equation by $\bar{x}$, then subtracting the result from the second estimating equation.

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)x_i - \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)\bar{x} = 0 \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)(x_i - \bar{x}) = 0$$

Next, plug in $\hat{y}_i = \hat{\theta}_0 + \hat{\theta}_1 x_i = \bar{y} + \hat{\theta}_1(x_i - \bar{x})$:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y} - \hat{\theta}_1(x - \bar{x}))(x_i - \bar{x}) = 0 \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})(x_i - \bar{x}) = \hat{\theta}_1 \times \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

By using the definition of correlation $\left(r = \frac{1}{n}\sum_{i=1}^{n}(\frac{x_i - \bar{x}}{\sigma_x})(\frac{y_i - \bar{y}}{\sigma_y})\right)$ and standard deviation $\left(\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}\right)$, we can conclude:

$$r\sigma_x\sigma_y = \hat{\theta}_1 \times \sigma_x^2$$

$$\hat{\theta}_1 = r\frac{\sigma_y}{\sigma_x}$$

Just as was given in Data 8!

Remember, this derivation found the optimal model parameters for SLR when using the MSE cost function. If we had used a different model or different loss function, we likely would have found different values for the best model parameters. However, regardless of the model and loss used, we can *always* follow these three steps to fit the model.

11.7 Evaluating Performance

At this point, we've:

- Defined our model
- Defined our loss function
- Fit the model to identify the best model parameters

Now, what are some ways to determine if our model was a good fit to our data? We will delve into this more in the next chapter, but there are three main ways for evaluating a model.

1. Statistics:

    - Plot original data
    - Compute column means
    - Compute standard deviations
    - If we want to fit a linear model, compute correlation (r)

2. Performance metrics:

    - Root Mean Square Error (RMSE). It is the square root of MSE, which is the average loss that we've been minimizing to determine optimal model parameters.
    - RMSE is in the same units as y.
    - A lower RMSE indicates more "accurate" predictions (lower "average loss" across data)

3. Visualization:

    - Look at a residual plot of $e_i = y_i - \hat{y}_i$ to visualize the difference between actual and predicted y values.

12 Constant Model, Loss, and Transformations

> **Note**
>
> - Derive the optimal model parameters for the constant model under MSE and MAE cost functions
> - Evaluate the differences between MSE and MAE risk
> - Understand the need for linearization of variables and apply the Tukey-Mosteller bulge diagram for transformations

Last time, we introduced the modeling process. We set up a framework to predict target variables as functions of our features, following a set workflow:

1. Choose a model
2. Choose a loss function
3. Fit the model
4. Evaluate model performance

To illustrate this process, we derived the optimal model parameters under simple linear regression with mean squared error as the cost function. In this lecture, we'll continue familiarizing ourselves with the modeling process by finding the best model parameters under a new model. We'll also test out two different loss functions to understand how our choice of loss influences model design. Later on, we'll consider what happens when a linear model isn't the best choice to capture trends in our data – and what solutions there are to create better models.

12.1 Constant Model + MSE

In today's lecture, our focus will be on the **constant model**. The constant model is slightly different from the simple linear regression model we've explored previously. Rather than generate predictions from an inputted feature variable, the constant model *predicts the same constant number every time.* We call this constant θ.

$$\hat{y}_i = \theta$$

θ is the parameter of the constant model, just as θ_0 and θ_1 were the parameters in SLR. Our task now is to determine what value of θ represents the optimal model – in other words, what number should we guess each time to have the lowest possible average loss on our data?

Consider the case where L2 (squared) loss is used as the loss function and mean squared error is used as the cost function. At this stage, we're well into the modeling process:

1. Choose a model: constant model
2. Choose a loss function: L2 loss
3. Fit the model
4. Evaluate model performance

In Homework 5, you will fit the constant model under MSE cost to find that the best choice of θ is the **mean of the observed y values**. In other words, $\hat{\theta} = \bar{y}$.

Let's take a moment to interpret this result. Our optimal model parameter is the value of the parameter that minimizes the cost function. This minimum value of the cost function can be expressed:

$$R(\hat{\theta}) = \min_{\theta} R(\theta)$$

To restate the above in plain English: we are looking at the value of the cost function when it takes the best parameter as input. This optimal model parameter, $\hat{\theta}$, is the value of θ that minimizes the cost R.

For modeling purposes, we care less about the minimum value of cost, $R(\hat{\theta})$, and more about the *value of* θ that results in this lowest average loss. In other words, we concern ourselves with finding the best parameter value such that:

$$\hat{\theta} = \arg\min_{\theta} R(\theta)$$

That is, we want to find the **arg**ument θ that **min**imizes the cost function.

12.2 Constant Model + MAE

We see now that changing the model used for prediction leads to a wildly different result for the optimal model parameter. What happens if we instead change the loss function used in model evaluation?

This time, we will consider the constant model with L1 (absolute loss) as the loss function. This means that the average loss will be expressed as the mean absolute error.

1. Choose a model: constant model

2. Choose a loss function: L1 loss
3. Fit the model
4. Evaluate model performance

To fit the model and find the optimal parameter value $\hat{\theta}$, follow the usual process of differentiating the cost function with respect to θ, setting the derivative equal to zero, and solving for θ. Writing this out in longhand:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \theta| \frac{d}{d\theta}R(\theta) = \frac{d}{d\theta}\left(\frac{1}{n}\sum_{i=1}^{n}|y_i - \theta|\right)\frac{d}{d\theta}R(\theta) = \frac{1}{n}\sum_{i=1}^{n}\frac{d}{d\theta}|y_i - \theta|$$

Here, we seem to have run into a problem: the derivative of an absolute value is undefined when the argument is 0 (i.e. when $y_i = \theta$). For now, we'll ignore this issue. It turns out that disregarding this case doesn't influence our final result.

To perform the derivative, consider two cases. When θ is *less than* y_i, the term $y_i - \theta$ will be positive and the absolute value has no impact. When θ is *greater than* y_i, the term $y_i - \theta$ will be negative. Applying the absolute value will convert this to a positive value, which we can express by saying $-(y_i - \theta) = \theta - y_i$.

$$|y_i - \theta| = \begin{cases} y_i - \theta & \text{if: } \theta < y_i \\ \theta - y_i & \text{if: } \theta > y_i \end{cases}$$

Taking derivatives:

$$\frac{d}{d\theta}|y_i - \theta| = \begin{cases} \frac{d}{d\theta}(y_i - \theta) = -1 & \text{if: } \theta < y_i \\ \frac{d}{d\theta}(\theta - y_i) = 1 & \text{if: } \theta > y_i \end{cases}$$

This means that we obtain a different value for the derivative for datapoints where $\theta < y_i$ and where $\theta > y_i$. We can summarize this by saying:

$$\frac{d}{d\theta}R(\theta) = \frac{1}{n}\sum_{i=1}^{n}\frac{d}{d\theta}|y_i - \theta| = \frac{1}{n}\left[\sum_{\theta<y_i}(-1) + \sum_{\theta>y_i}(+1)\right]$$

To finish finding the best value of θ, set this derivative equal to zero and solve for θ. You'll do this in Homework 5 to show that $\hat{\theta} = \text{median}(y)$.

12.3 Comparing Loss Functions

Now, we've tried our hand at fitting a model under both MSE and MAE cost functions. How do the two results compare?

Let's consider a dataset where each entry represents the number of drinks sold at a bubble tea store each day. We'll fit a constant model to predict the number of drinks that will be sold tomorrow.

```
import numpy as np
drinks = np.array([20, 21, 22, 29, 33])
drinks
```

```
array([20, 21, 22, 29, 33])
```

From our derivations above, we know that the optimal model parameter under MSE cost is the mean of the dataset. Under MAE cost, the optimal parameter is the median of the dataset.

```
np.mean(drinks), np.median(drinks)
```

```
(25.0, 22.0)
```

If we plot each empirical risk function across several possible values of θ, we find that each $\hat{\theta}$ does indeed correspond to the lowest value of error:

Notice that the MSE above is a **smooth** function – it is differentiable at all points, making it easy to minimize using numerical methods. The MAE, in contrast, is not differentiable at each of its "kinks." We'll explore how the smoothness of the cost function can impact our ability to apply numerical optimization in a few weeks.

How do outliers affect each cost function? Imagine we replace the largest value in the dataset with 1000. The mean of the data increases substantially, while the median is nearly unaffected.

```
drinks_with_outlier = np.append(drinks, 1000)
display(drinks_with_outlier)
np.mean(drinks_with_outlier), np.median(drinks_with_outlier)
```

```
array([  20,   21,   22,   29,   33, 1000])
```

```
(187.5, 25.5)
```

This means that under the MSE, the optimal model parameter $\hat{\theta}$ is strongly affected by the presence of outliers. Under the MAE, the optimal parameter is not as influenced by outlying data. We can generalize this by saying that the MSE is **sensitive** to outliers, while the MAE is **robust** to outliers.

Let's try another experiment. This time, we'll add an additional, non-outlying datapoint to the data.

```
drinks_with_additional_observation = np.append(drinks, 35)
drinks_with_additional_observation
```

```
array([20, 21, 22, 29, 33, 35])
```

When we again visualize the cost functions, we find that the MAE now plots a horizontal line between 22 and 29. This means that there are *infinitely* many optimal values for the model parameter: any value $\hat{\theta} \in [22, 29]$ will minimize the MAE. In contrast, the MSE still has a single best value for $\hat{\theta}$. In other words, the MSE has a **unique** solution for $\hat{\theta}$; the MAE is not guaranteed to have a single unique solution.

12.4 Evaluating Models

This leaves us with one final question – how "good" are the predictions made by this "best" fitted model?

One way we might want to evaluate our model's performance is by computing summary statistics. If the mean and standard deviation of our predictions are close to those of the original observed y_is, we might be inclined to say that our model has done well. A large magnitude for the correlation coefficient between the feature and response variables might also support this conclusion. However, we should be cautious with this approach. To see why, we'll consider a classic dataset called **Anscombe's quartet.**

It turns out that the four sets of points shown here all have identical means, standard deviations, and correlation coefficients. However, it only makes sense to model the first of these four sets of data using SLR! It is important to visualize your data *before* starting to model to confirm that your choice of model makes sense for the data.

Another way of evaluating model performance is by using performance metrics. A common choice of metric is the **Root Mean Squared Error**, or RMSE. The RMSE is simply the square root of MSE. Taking the square root converts the value back into the original, non-squared units of y_i, which is useful for understanding the model's performance. A low RMSE

indicates more "accurate" predictions – that there is lower average loss across the dataset.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

We may also wish to visualize the model's **residuals**, defined as the difference between the observed and predicted y_i value ($e_i = y_i - \hat{y}_i$). This gives a high-level view of how "off" each prediction is from the true observed value. Recall that you explored this concept in Data 8: a good regression fit should display no clear pattern in its plot of residuals. The residual plots for Anscombe's quartet are displayed below. Note how only the first plot shows no clear pattern to the magnitude of residuals. This is an indication that SLR is not the best choice of model for the remaining three sets of points.

12.5 Linear Transformations

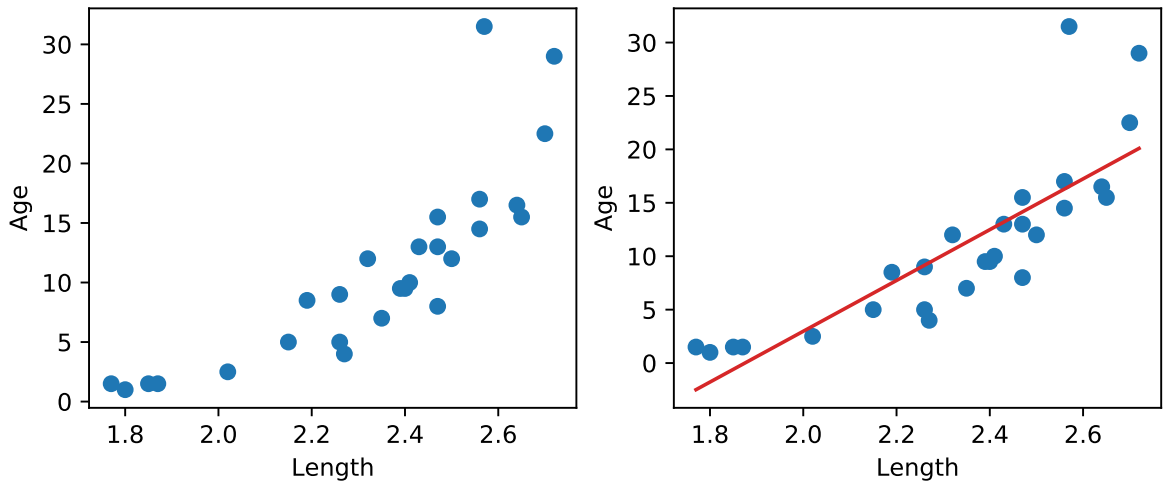
At this point, we have an effective method of fitting models to predict linear relationships. Given a feature variable and target, we can apply our four-step process to find the optimal model parameters.

A key word above is *linear*. When we computed parameter estimates earlier, we assumed that x_i and y_i shared roughly a linear relationship.

Data in the real world isn't always so straightforward. Consider the dataset below, which contains information about the ages and lengths of dugongs.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

dugong = pd.read_csv("data/dugongs.txt", delimiter="\t").sort_values("Length")
x, y = dugong["Length"], dugong["Age"]

# `corrcoef` computes the correlation coefficient between two variables
# `std` finds the standard deviation
r = np.corrcoef(x, y)[0, 1]
theta_1 = r*np.std(y)/np.std(x)
theta_0 = np.mean(y) - theta_1*np.mean(x)

fig, ax = plt.subplots(1, 2, dpi=200, figsize=(8, 3))
ax[0].scatter(x, y)
ax[0].set_xlabel("Length")
ax[0].set_ylabel("Age")
```

```
ax[1].scatter(x, y)
ax[1].plot(x, theta_0 + theta_1*x, "tab:red")
ax[1].set_xlabel("Length")
ax[1].set_ylabel("Age");
```



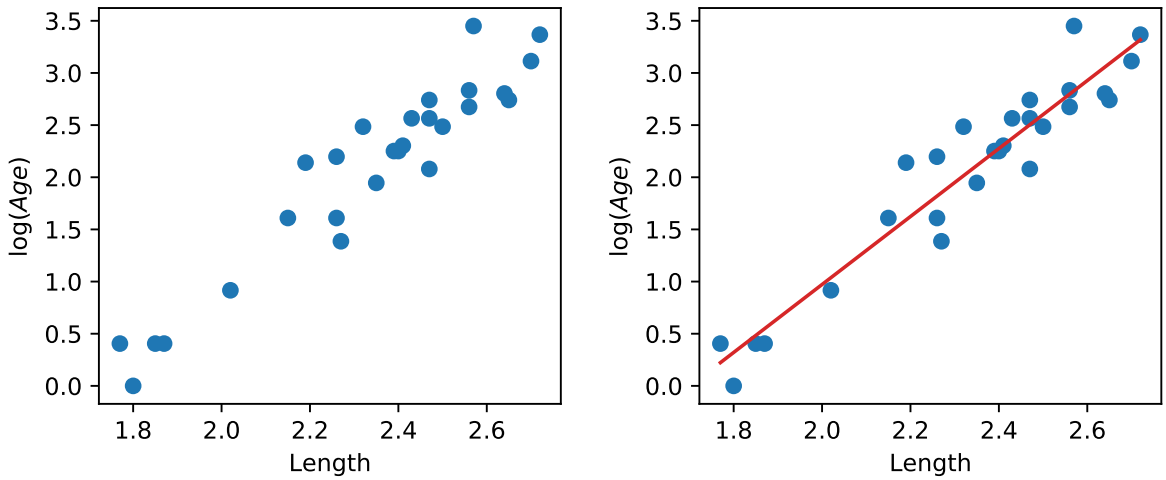
Looking at the plot on the left, we see that there is a slight curvature to the data points. Plotting the SLR curve on the right results in a poor fit.

For SLR to perform well, we'd like there to be a rough linear trend relating `"Age"` and `"Length"`. What is making the raw data deviate from a linear relationship? Notice that the data points with `"Length"` greater than 2.6 have disproportionately high values of `"Age"` relative to the rest of the data. If we could manipulate these data points to have lower `"Age"` values, we'd "shift" these points downwards and reduce the curvature in the data. Applying a logarithmic transformation to y_i (that is, taking log(`"Age"`)) would achieve just that.

An important word on log: in Data 100 (and most upper-division STEM courses), log denotes the natural logarithm with base e. The base-10 logarithm, where relevant, is indicated by $\log_{10}$.

```
z = np.log(y)

r = np.corrcoef(x, z)[0, 1]
theta_1 = r*np.std(z)/np.std(x)
theta_0 = np.mean(z) - theta_1*np.mean(x)

fig, ax = plt.subplots(1, 2, dpi=200, figsize=(8, 3))
```

```
ax[0].scatter(x, z)
ax[0].set_xlabel("Length")
ax[0].set_ylabel(r"$\log{(Age)}$")

ax[1].scatter(x, z)
ax[1].plot(x, theta_0 + theta_1*x, "tab:red")
ax[1].set_xlabel("Length")
ax[1].set_ylabel(r"$\log{(Age)}$")

plt.subplots_adjust(wspace=0.3);
```



Our SLR fit looks a lot better! We now have a new target variable: the SLR model is now trying to predict the *log* of `"Age"`, rather than the untransformed `"Age"`. In other words, we are applying the transformation $z_i = \log(y_i)$. The SLR model becomes:

$$\log(\hat{y_i}) = \theta_0 + \theta_1 x_i$$
$$\hat{z}_i = \theta_0 + \theta_1 x_i$$

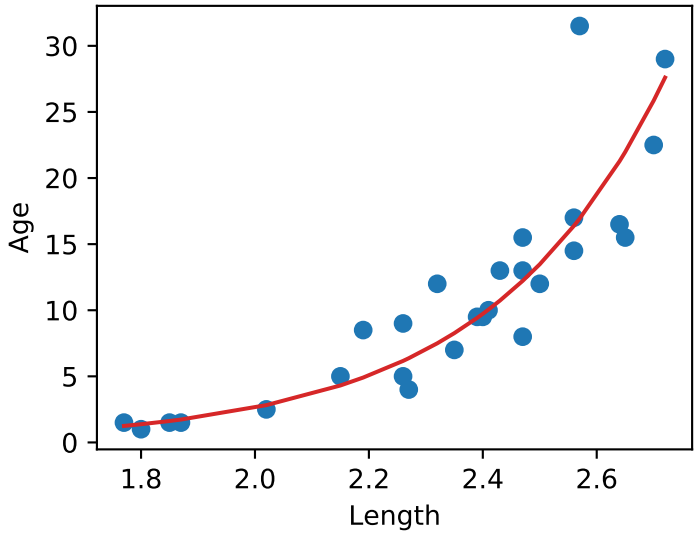
It turns out that this linearized relationship can help us understand the underlying relationship between x_i and y_i. If we rearrange the relationship above, we find:

$$\log(y_i) = \theta_0 + \theta_1 x_i y_i = e^{\theta_0 + \theta_1 x_i} y_i = (e^{\theta_0}) e^{\theta_1 x_i} y_i = C e^{k x_i}$$

For some constants C and k.

y_i is an *exponential* function of x_i. Applying an exponential fit to the untransformed variables corroborates this finding.

```
plt.figure(dpi=120, figsize=(4, 3))

plt.scatter(x, y)
plt.plot(x, np.exp(theta_0)*np.exp(theta_1*x), "tab:red")
plt.xlabel("Length")
plt.ylabel("Age");
```



You may wonder: why did we choose to apply a log transformation specifically? Why not some other function to linearize the data?

Practically, many other mathematical operations that modify the relative scales of `"Age"` and `"Length"` could have worked here. The **Tukey-Mosteller Bulge Diagram** is a useful tool for summarizing what transformations can linearize the relationship between two variables. To determine what transformations might be appropriate, trace the shape of the "bulge" made by your data. Find the quadrant of the diagram that matches this bulge. The transformations shown on the vertical and horizontal axes of this quadrant can help improve the fit between the variables.

13 Ordinary Least Squares

> **Note**
>
> - Define linearity with respect to a vector of parameters θ
> - Understand the use of matrix notation to express multiple linear regression
> - Interpret ordinary least squares as the minimization of the norm of the residual vector
> - Compute performance metrics for multiple linear regression

We've now spent a number of lectures exploring how to build effective models – we introduced the SLR and constant models, selected cost functions to suit our modeling task, and applied transformations to improve the linear fit.

Throughout all of this, we considered models of one feature ($\hat{y}_i = \theta_0 + \theta_1 x_i$) or zero features ($\hat{y}_i = \theta$). As data scientists, we usually have access to datasets containing *many* features. To make the best models we can, it will be beneficial to consider all of the variables available to us as inputs to a model, rather than just one. In today's lecture, we'll introduce **multiple Linear regression** as a framework to incorporate multiple features into a model. We will also learn how to accelerate the modeling process – specifically, we'll see how linear algebra offers us a powerful set of tools for understanding model performance.

13.1 Linearity

An expression is **linear in** θ (a set of parameters) if it is a linear combination of the elements of the set. Checking if an expression can separate into a matrix product of two terms: a **vector of** θ s, and a matrix/vector **not involving** θ.

Example: $\theta = [\theta_1, \theta_2, ...\theta_p]$

1. Linear in theta: $\hat{y} = \theta_0 + 2\theta_1 + 3\theta_2$

$$\hat{y} = \begin{bmatrix}1 & 2 & 3\end{bmatrix} \begin{bmatrix}\theta_0 \\ \theta_1 \\ \theta_2\end{bmatrix}$$

2. Not linear in theta: $\hat{y} = \theta_0\theta_1 + 2\theta_1^2 + 3log(\theta_2)$

13.2 Multiple Linear Regression

Multiple Linear regression is an extension of simple linear regression that adds additional features into the model. Say we collect information on several variables when making an observation. For example, we may record the age, height, and weekly hours of sleep for a student in Data 100. This single observation now contains data for multiple features. To accommodate for the fact that we now consider several feature variables, we'll adjust our notation slightly. Each observation can now be thought of as a row vector with an entry for each of p features.

The multiple Linear regression model takes the form:

$$\hat{y}_i = \theta_0 \ + \ \theta_1 x_{i1} \ + \ \theta_2 x_{i2} \ + \ ... \ + \ \theta_p x_{ip}$$

Our ith prediction, $\hat{y}_i$, is a linear combination of the parameters, θ_i. Because we are now dealing with many parameter values, we'll collect them all into a **parameter vector** with dimensions $(p+1) \times 1$ to keep things tidy.

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix}$$

We are now working with two vectors: a row vector representing the observed data, and a column vector containing the model parameters. The multiple Linear regression model given above is **equivalent to the dot (scalar) product of the observation vector and parameter vector**.

$$[1, \ x_{i1}, \ x_{i2}, \ x_{i3}, \ ..., \ x_{ip}]\theta = [1, \ x_{i1}, \ x_{i2}, \ x_{i3}, \ ..., \ x_{ip}] \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix} = \theta_0 \ + \ \theta_1 x_{i1} \ + \ \theta_2 x_{i2} \ + \ ... \ + \ \theta_p x_{ip}$$

Notice that we have inserted 1 as the first value in the observation vector. When the dot product is computed, this 1 will be multiplied with θ_0 to give the intercept of the regression model. We call this 1 entry the **intercept** or **bias** term.

13.3 Linear Algebra Approach

We now know how to generate a single prediction from multiple observed features. Data scientists usually work at scale – that is, they want to build models that can produce many predictions, all at once. The vector notation we introduced above gives us a hint on how we can expedite multiple Linear regression. We want to use the tools of linear algebra.

Let's think carefully about what we did to generate the single prediction above. To make a prediction from the first observation in the data, we took the scalar product of the parameter vector and first observation vector. To make a prediction from the *second* observation, we would repeat this process to find the scalar product of the parameter vector and the *second* observation vector. If we wanted to find the model predictions for each observation in the dataset, we'd repeat this process for all n observations in the data.

$$\begin{aligned}
\hat{y}_1 &= [1,\, x_{11},\, x_{12},\, x_{13},\, ...,\, x_{1p}]\theta \\
\hat{y}_2 &= [1,\, x_{21},\, x_{22},\, x_{23},\, ...,\, x_{2p}]\theta \\
&\vdots \\
\hat{y}_n &= [1,\, x_{n1},\, x_{n2},\, x_{n3},\, ...,\, x_{np}]\theta
\end{aligned}$$

Our observed data is represented by n row vectors, each with dimension $(p+1)$. We can collect them all into a single matrix, which we call $\mathbb{X}$.

The matrix $\mathbb{X}$ is known as the **design matrix**. It contains all observed data for each of our p features. It often (but not always) contains an additional column of all ones to represent the **intercept** or **bias column**.

To review what is happening in the design matrix: each row represents a single observation. For example, a student in Data 100. Each column represents a feature. For example, the ages of students in Data 100. This convention allows us to easily transfer our previous work in DataFrames over to this new linear algebra perspective.

The multiple Linear regression model can then be restated in terms of matrices:

$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

Here, $\hat{\mathbb{Y}}$ is the **prediction vector** with dimensions $(n \times 1)$. It contains the prediction made by the model for each of n input observations.

We now have a new approach to understanding models in terms of vectors and matrices. To accompany this new convention, we should update our understanding of cost functions and model fitting.

Recall our definition of MSE:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

At its heart, the MSE is a measure of *distance* – it gives an indication of how "far away" the predictions are from the true values, on average.

When working with vectors, this idea of "distance" is represented by the **norm**. More precisely, the distance between two vectors $\vec{a}$ and $\vec{b}$ can be expressed as:

$$||\vec{a} - \vec{b}||_2 = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2} = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

The double bars are mathematical notation for the norm. The subscript 2 indicates that we are computing the L2, or squared norm.

Looks pretty familiar! We can rewrite the MSE to express it as a squared L2 norm in terms of the prediction vector, $\hat{\mathbb{Y}}$, and true target vector, $\mathbb{Y}$:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}||\mathbb{Y} - \hat{\mathbb{Y}}||_2^2$$

Here, the superscript 2 outside of the norm double bars means that we are *squaring* the norm. If we plug in our linear model $\hat{\mathbb{Y}} = \mathbb{X}\theta$, we find the MSE cost function in vector notation:

$$R(\theta) = \frac{1}{n}||\mathbb{Y} - \mathbb{X}\theta||_2^2$$

Under the linear algebra perspective, our new task is to fit the optimal parameter vector θ such that the cost function is minimized. Equivalently, we wish to minimize the norm

$$||\mathbb{Y} - \mathbb{X}\theta||_2 = ||\mathbb{Y} - \hat{\mathbb{Y}}||_2$$

.

We can restate this goal in two ways:

- Minimize the *distance* between the vector of true values, $\mathbb{Y}$, and the vector of predicted values, $\hat{\mathbb{Y}}$
- Minimize the *length* of the **residual vector**, defined as:

$$e = \mathbb{Y} - \hat{\mathbb{Y}} = \begin{bmatrix} y_1 - \hat{y}_1 \\ y_2 - \hat{y}_2 \\ \vdots \\ y_n - \hat{y}_n \end{bmatrix}$$

13.4 Geometric Perspective

To derive the best parameter vector to meet this goal, we can turn to the geometric properties of our modeling set-up.

Up until now, we've mostly thought of our model as a scalar product between horizontally-stacked observations and the parameter vector. We can also think of $\hat{\mathbb{Y}}$ as a **linear combination of feature vectors**, scaled by the parameters. We use the notation $\mathbb{X}_{:,i}$ to denote the ith column of the design matrix. You can think of this as following the same convention as used when calling `.iloc` and `.loc`. ":" means that we are taking all entries in the ith column.

$$\hat{\mathbb{Y}} = \theta_0 \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} + \theta_1 \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{n1} \end{bmatrix} + ... + \theta_p \begin{bmatrix} x_{1p} \\ x_{2p} \\ \vdots \\ x_{np} \end{bmatrix} = \theta_0 \mathbb{X}_{:,\,1} + \theta_1 \mathbb{X}_{:,\,2} + ... + \theta_p \mathbb{X}_{:,\,p+1}$$

This new approach is useful because it allows us to take advantage of the properties of linear combinations.

Recall that the **span** or **column space** of a matrix is the set of all possible linear combinations of the matrix's columns. In other words, the span represents every point in space that could possibly be reached by adding and scaling some combination of the matrix columns.

Because the prediction vector, $\hat{\mathbb{Y}} = \mathbb{X}\theta$, is a linear combination of the columns of $\mathbb{X}$, we know that the **predictions are contained in the span of** $\mathbb{X}$. That is, we know that $\hat{\mathbb{Y}} \in \text{Span}(\mathbb{X})$.

The diagram below is a simplified view of Span($\mathbb{X}$), assuminh that each column of $\mathbb{X}$ has length n. Notice that the columns of $\mathbb{X}$ define a subspace of $\mathbb{R}^n$, where each point in the subspace can be reached by a linear combination of $\mathbb{X}$'s columns. The prediction vector $\hat{\mathbb{Y}}$ lies somewhere in this subspace.

Examining this diagram, we find a problem. The vector of true values, $\mathbb{Y}$, could theoretically lie *anywhere* in $\mathbb{R}^n$ space – its exact location depends on the data we collect out in the real world. However, our multiple Linear regression model can only make predictions in the subspace of $\mathbb{R}^n$ spanned by $\mathbb{X}$. Remember the model fitting goal we established in the previous section: we want to generate predictions such that the distance between the vector of true values, $\mathbb{Y}$, and the vector of predicted values, $\hat{\mathbb{Y}}$, is minimized. This means that **we want $\hat{\mathbb{Y}}$ to be the vector in Span($\mathbb{X}$) that is closest to $\mathbb{Y}$**.

Another way of rephrasing this goal is to say that we wish to minimize the length of the residual vector e, as measured by its L_2 norm.

The vector in Span($\mathbb{X}$) that is closest to $\mathbb{Y}$ is always the **orthogonal projection** of $\mathbb{Y}$ onto Span($\mathbb{X}$). Thus, we should choose the parameter vector θ that makes the **residual vector orthogonal to any vector in Span($\mathbb{X}$)**. You can visualize this as the vector created by dropping a perpendicular line from $\mathbb{Y}$ onto the span of $\mathbb{X}$.

How does this help us identify the optimal parameter vector, $\hat{\theta}$? Recall that two vectors are orthogonal if their dot product is zero. A vector $\vec{v}$ is orthogonal to the span of a matrix M if $\vec{v}$ is orthogonal to each column in M. Put together, a vector $\vec{v}$ is orthogonal to Span(M) if:

$$M^T \vec{v} = \vec{0}$$

Because our goal is to find $\hat{\theta}$ such that the residual vector $e = \mathbb{Y} - \mathbb{X}\theta$ is orthogonal to Span($\mathbb{X}$), we can write:

$$\begin{aligned}
\mathbb{X}^T e &= \vec{0} \\
\mathbb{X}^T(\mathbb{Y} - \mathbb{X}\hat{\theta}) &= \vec{0} \\
\mathbb{X}^T\mathbb{Y} - \mathbb{X}^T\mathbb{X}\hat{\theta} &= \vec{0} \\
\mathbb{X}^T\mathbb{X}\hat{\theta} &= \mathbb{X}^T\mathbb{Y}
\end{aligned}$$

This last line is known as the **normal equation**. Any vector θ that minimizes MSE on a dataset must satisfy this equation.

If $\mathbb{X}^T\mathbb{X}$ is invertible, we can conclude:

$$\hat{\theta} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y}$$

This is called the **least squares estimate** of θ: it is the value of θ that minimizes the squared loss.

Note that the least squares estimate was derived under the assumption that $\mathbb{X}^T\mathbb{X}$ is *invertible*. This condition holds true when $\mathbb{X}^T\mathbb{X}$ is full column rank, which, in turn, happens when $\mathbb{X}$ is full column rank. We will explore the consequences of this fact in lab and homework.

13.5 Evaluating Model Performance

Our geometric view of multiple Linear regression has taken us far! We have identified the optimal set of parameter values to minimize MSE in a model of multiple features.

Now, we want to understand how well our fitted model performs. One measure of model performance is the **Root Mean Squared Error**, or RMSE. The RMSE is simply the square root of MSE. Taking the square root converts the value back into the original, non-squared units of y_i, which is useful for understanding the model's performance. A low RMSE indicates more "accurate" predictions – that there is lower average loss across the dataset.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

When working with SLR, we generated plots of the residuals against a single feature to understand the behavior of residuals. When working with several features in multiple Linear regression, it no longer makes sense to consider a single feature in our residual plots. Instead, multiple Linear regression is evaluated by making plots of the residuals against the predicted values. As was the case with SLR, a multiple Linear model performs well if its residual plot shows no patterns.

For SLR, we used the correlation coefficient to capture the association between the target variable and a single feature variable. In a multiple Linear setting, we will need a performance metric that can account for multiple features at once. **Multiple R^2**, also called the **coefficient of determination**, is the ratio of the variance of the predicted values $\hat{y}_i$ to the variance of the true values y_i. It can be interpreted as the *proportion* of variance in the observations that is explained by the model.

$$R^2 = \frac{\text{variance of } \hat{y}_i}{\text{variance of } y_i} = \frac{\sigma^2_{\hat{y}}}{\sigma^2_y}$$

As we add more features, our fitted values tend to become closer and closer to our actual values. Thus, $\mathbb{R}^2$ increases.

13.6 OLS Properties

1. When using the optimal parameter vector, our residuals $e = \mathbb{Y} - \hat{\mathbb{Y}}$ are orthogonal to $span(\mathbb{X})$

$$\mathbb{X}^T e = 0$$

Proof:
The optimal parameter vector, $\hat{\theta}$, solves the normal equations $\implies \hat{\theta} = \mathbb{X}^T\mathbb{X}^{-1}\mathbb{X}^T\mathbb{Y}$

$$\mathbb{X}^T e = \mathbb{X}^T(\mathbb{Y} - \hat{\mathbb{Y}})$$

$$\mathbb{X}^T(\mathbb{Y} - \mathbb{X}\hat{\theta}) = \mathbb{X}^T\mathbb{Y} - \mathbb{X}^T\mathbb{X}\hat{\theta}$$

Any matrix multiplied with its own inverse is the identity matrix $\mathbb{I}$

$$\mathbb{X}^T\mathbb{Y} - (\mathbb{X}^T\mathbb{X})(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y} = \mathbb{X}^T\mathbb{Y} - \mathbb{X}^T\mathbb{Y} = 0$$

2. For all linear models *with an intercept term*, the sum of residuals is zero.

$$\sum_{i}^{n} e_i = 0$$

Proof:
For all linear models *with an intercept term*, the average of the predicted y values is equal to the average of the true y values.

$$\bar{y} = \bar{\hat{y}}$$

Rewriting the sum of residuals as two separate sums,

$$\sum_{i}^{n} e_i = \sum_{i}^{n} y_i - \sum_{i}^{n} \hat{y}_i$$

Each respective sum is a multiple of the average of the sum.

$$\sum_{i}^{n} e_i = n\bar{y} - n\bar{y} = n(\bar{y} - \bar{y}) = 0$$

3. The Least Squares estimate $\hat{\theta}$ is unique if and only if $\mathbb{X}$ is full column rank.

Proof:
We know the solution to the normal equation $\mathbb{X}^T\mathbb{X}\hat{\theta} = \mathbb{Y}$ is the least square estimate that fulfills the prior equality.
$\hat{\theta}$ has a unique solution $\iff$ the square matrix $\mathbb{X}^T\mathbb{X}$ is invertible.
The rank of a square matrix is the maximum number of of linearly independent columns it contains. $\mathbb{X}^T\mathbb{X}$ has shape $(p+1) \times (p+1)$, and therefore has max rank p + 1.
$rank(\mathbb{X}^T\mathbb{X}) = rank(\mathbb{X})$ (proof out of scope).
Therefore $\mathbb{X}^T\mathbb{X}$ has rank p + 1 $\iff$ $\mathbb{X}$ has rank p + 1 $\iff$ $\mathbb{X}$ is full column rank.

14 Gradient Descent

> **Note**
>
> - Understand the standard workflow for fitting models in `sklearn`
> - Describe the conceptual basis for gradient descent
> - Compute the gradient descent update on a provided dataset

At this point, we've grown quite familiar with the modeling process. We've introduced the concept of loss, used it to fit several types of models, and, most recently, extended our analysis to multiple regression. Along the way, we've forged our way through the mathematics of deriving the optimal model parameters in all of its gory detail. It's time to make our lives a little easier – let's implement the modeling process in code!

In this lecture, we'll explore three techniques for model fitting:

1. Translating our derived formulas for regression to Python
2. Using the `sklearn` Python package
3. Applying gradient descent for numerical optimization

14.1 `sklearn`: Implementing Derived Formulas in Code

Throughout this lecture, we'll refer to the `penguins` dataset.

```
import pandas as pd
import seaborn as sns
import numpy as np

penguins = sns.load_dataset("penguins")
penguins = penguins[penguins["species"] == "Adelie"].dropna()
penguins.head(5)
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0	Male

Suppose our goal is to predict the value of the `'bill depth'` for a particular penguin given its `'flipper length'`.

```
# Define the design matrix, X...
X = penguins[["flipper_length_mm"]]

# ...as well as the target variable, y
y = penguins[["bill_depth_mm"]]
```

14.1.1 Simple Linear Regression (SLR)

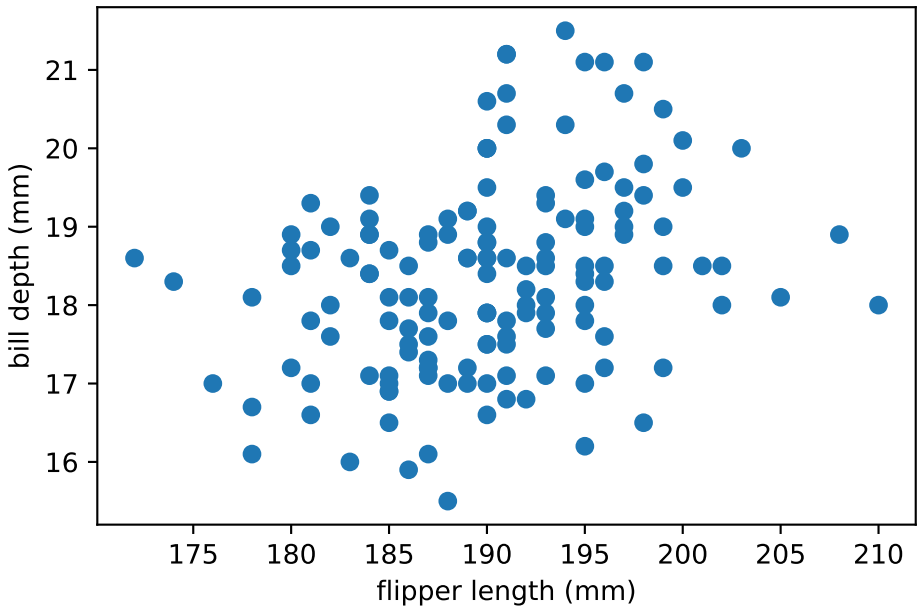
In the SLR framework we learned last week, this means we are saying our model for `bill depth`, y, is a linear function of `flipper length`, x:

$$\hat{y} = \theta_0 + \theta_1 x$$

Let's do some EDA first.

```
import matplotlib.pyplot as plt

plt.xlabel("flipper length (mm)")
plt.ylabel("bill depth (mm)")
plt.scatter(data = penguins, x = "flipper_length_mm", y = "bill_depth_mm")
```

```
<matplotlib.collections.PathCollection at 0x7ff147a143a0>
```



Based on our EDA, there is a linear relationship, though it is somewhat weak.

14.1.1.1 SLR w/ Derived Analytical Formulas

Let $\hat{\theta}_0$ and $\hat{\theta}_1$ be the choices that minimize the Mean Squared Error.

One approach to compute $\hat{\theta}_0$ and $\hat{\theta}_1$ is analytically, using the equations we derived in a previous lecture:

$$\hat{\theta}_0 = \bar{y} - \hat{\theta}_1 \bar{x}$$

$$\hat{\theta}_1 = r \frac{\sigma_y}{\sigma_x}$$

$$r = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i - x}{\sigma_x} \right) \left(\frac{y_i - y}{\sigma_y} \right)$$

Let's implement these using the base numpy library, which provides many important functions such as `.mean` and `.std` .

```
x = penguins["flipper_length_mm"]
y = penguins["bill_depth_mm"]

x_bar, sigma_x = np.mean(x), np.std(x)
y_bar, sigma_y = np.mean(y), np.std(y)
r = np.sum((x - x_bar) / sigma_x * (y - y_bar) / sigma_y) / len(x)

theta1_hat = r * sigma_y / sigma_x

theta0_hat = y_bar - theta1_hat * x_bar

print(f"bias parameter: {theta0_hat}, \nslope parameter: {theta1_hat}".format(theta0_hat,t
```

```
bias parameter: 7.297305899612297,
slope parameter: 0.058126223695067675
```

14.1.1.2 SLR Analytical Approach Performance

Let's first assess how "good" this model is using a performance metric. For this exercise, let's use the MSE. As a review:

Mean Squared Error: We can compute this explicitly by averaging the square of the residuals e_i:

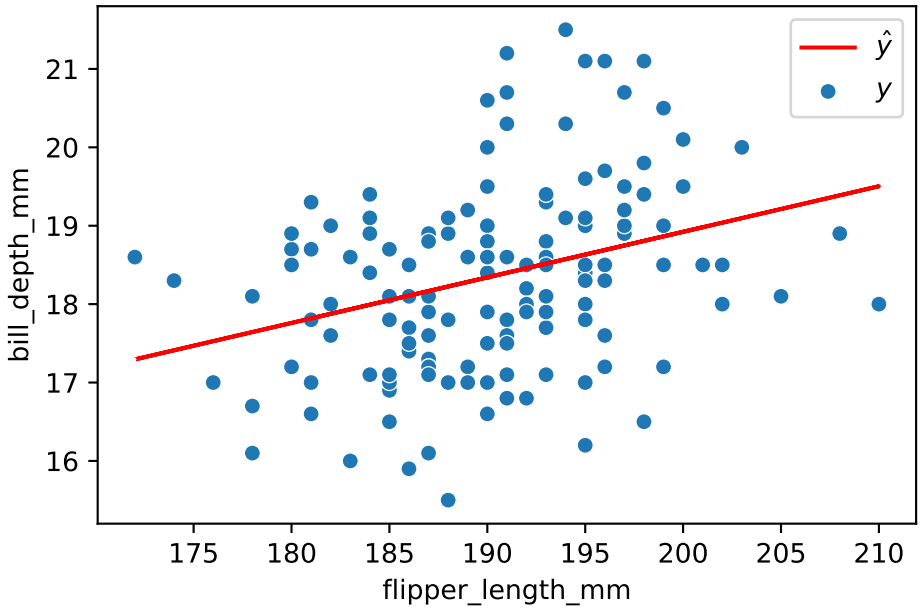
$$MSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(e_i)^2 = \frac{1}{n}$$

```
# using our estimated parameter values to create a column containing our
# SLR predictions and errors
penguins["analytical_preds_slr"] = theta0_hat + theta1_hat * penguins["flipper_length_mm"]

penguins["residual"] = penguins["bill_depth_mm"] - penguins["analytical_preds_slr"]

penguins

print("MSE: ", np.mean(penguins["residual"]**2))
```

```
MSE:  1.3338778799806363
```

Let's plot plot our results.

```
sns.scatterplot(data = penguins, x = "flipper_length_mm", y = "bill_depth_mm")
plt.plot(penguins["flipper_length_mm"], penguins["analytical_preds_slr"], 'r') # a line
plt.legend([r'$\hat{y}$', '$y$']);
```



14.1.1.3 SLR w/ `sklearn`

We've already saved a lot of time (and avoided tedious calculations) by translating our derived formulas into code. However, we still had to go through the process of writing out the linear algebra ourselves.

To make life *even easier*, we can turn to the `sklearn` Python library. `sklearn` is a robust library of machine learning tools used extensively in research and industry. It gives us a wide variety of in-built modeling frameworks and methods, so we'll keep returning to `sklearn` techniques as we progress through Data 100.

Regardless of the specific type of model being implemented, `sklearn` follows a standard set of steps for creating a model.

1. Create a model object. This generates a new instance of the model class. You can think of it as making a new copy of a standard "template" for a model. In pseudocode, this looks like: `my_model = ModelName()`

2. Fit the model to the `X` design matrix and `Y` target vector. This calculates the optimal model parameters "behind the scenes" without us explicitly working through the calculations ourselves. The fitted parameters are then stored within the model for use in future predictions: `my_model.fit(X, Y)`
3. Analyze the fitted parameters using `.coef_` or `.intercept_`, or use the fitted model to make predictions on the `X` input data using `.predict`.

   ```
   my_model.coef_

   my_model.intercept_

   my_model.predict(X)
   ```

Let's put this into action with our multiple regression task. First, initialize an instance of the `LinearRegression` class.

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
```

Next, fit the model instance to the design matrix `X` and target vector `Y` by calling `.fit`.

```
model.fit(X, y)
```

```
LinearRegression()
```

And, lastly, generate predictions for $\hat{Y}$ using the `.predict` method. Here are the first 5 penguins in our dataset. How close are our analytical solution's predictions to our `sklearn` model's predictions?

```
# Like before, show just the first 5 predictions. The output of predict is usually a np.ar
penguins["sklearn_preds_slr"] = model.predict(X)
sklearn_5 = penguins["sklearn_preds_slr"][:5].to_numpy()
print("Sklearn solution: ", sklearn_5)
analytical_5 = penguins["analytical_preds_slr"][:5].to_numpy()
print("Analytical solution: ", analytical_5)
```

```
Sklearn solution:  [17.81815239 18.10878351 18.63191952 18.51566707 18.3412884 ]
Analytical solution:  [17.81815239 18.10878351 18.63191952 18.51566707 18.3412884 ]
```

You can also use the model to predict what the `bill depth` of a hypothetical penguin with `flipper length` of 185mm would have.

```
# this produces a warning since we
# did not specify what X this refers
# to, but since we only
# have one input it is negligible

model.predict([[185]])
```

```
array([18.05065728])
```

We can also check if the fitted parameters,$\hat{\theta}_0, \hat{\theta}_1$, themselves are similar to our analytical solution. Note that since we can have at most 1 intercept in a SLR or OLS model, so we always get back a `scalar` value from `.intercept`. However, when OLS can have multiple coefficient values, so `.coef` returns an array.

```
theta0 = model.intercept_      # this a scalar
print("analytical bias term: ", theta0_hat)
print("sklearn bias term: ", theta0)


theta1 = model.coef_           # this an array
print("analytical coefficient terms: ", theta1_hat)
print("sklearn coefficient terms: ", theta1)
```

```
analytical bias term:  7.297305899612297
sklearn bias term:  7.297305899612306
analytical coefficient terms:  0.058126223695067675
sklearn coefficient terms:  [0.05812622]
```

14.1.1.4 SLR `sklearn` Performance

The `sklearn` package also provides a function that computes the MSE from a list of observations and predictions. This avoids us having to manually compute MSE by first computing residuals.

documentation

```
from sklearn.metrics import mean_squared_error
MSE_sklearn = mean_squared_error(penguins["bill_depth_mm"], penguins["sklearn_preds_slr"])
print("MSE: ", MSE_sklearn)
```

```
MSE:  1.3338778799806374
```

We've generated the exact same predictions and error as before, but without any need for manipulating matrices ourselves!

14.1.2 Multiple Linear Regression

In the previous lecture, we expressed multiple linear regression using matrix notation.

$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

14.1.2.1 OLS w/ Derived Analytical Formulas

We used a geometric approach to derive the following expression for the optimal model parameters under MSE error, also called Ordinary Least Squares (OLS):

$$\hat{\theta} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y}$$

That's a whole lot of matrix manipulation. How do we implement it in Python?

There are three operations we need to perform here: multiplying matrices, taking transposes, and finding inverses.

- To perform matrix multiplication, use the `@` operator
- To take a transpose, call the `.T` attribute of an array or DataFrame
- To compute an inverse, use `numpy`'s in-built method `np.linalg.inv`

```
X = penguins[["flipper_length_mm", "body_mass_g"]].copy()

X["bias"] = np.ones(len(X))
y = penguins["bill_depth_mm"]


theta_hat = np.linalg.inv(X.T @ X) @ X.T @ y
theta_hat
```

	0
0	0.009828
1	0.001477
2	11.002995

Note that since we added "bias" last, `theta_hat[2]` is our estimated value for the θ_0. To make predictions using our newly-fitted model coefficients, matrix-multiply `X` and `theta_hat`.

```
y_hat = X.to_numpy() @ theta_hat

# Show just the first 5 predictions to save space on the page
y_hat[:5]
penguins["analytical_preds_ols"] = y_hat
```

Note, this technique doesn't work if our X is **not invertible**.

14.1.2.2 OLS w/ `sklearn`

We can actually compute the optimal parameters very easily using sklearn, using the exact code that we wrote earlier. Note: sklearn does NOT use the normal equations. Instead it uses gradient descent, a technique we will learn about soon, which can minimize ANY function, not just the MSE.

```
# creating our design matrix.
# Note:
#  - no bias term needed here bc of sklearn automatically includes one
#  - to remove the intercept term, set the fit_intercept = true in LinearRegression constr

X_2d = penguins[["flipper_length_mm", "body_mass_g"]]
y = penguins["bill_depth_mm"]
model_2d = LinearRegression() # note fit_intercept=True by default
model_2d.fit(X_2d, y)
```

```
LinearRegression()
```

Now we again have a model with which we can use to make predictions. For example, we can ask our model about a penguin's bill depth if they have 185-mm flipper length and 3750 g body mass.

```
penguins["sklearn_predictions_ols"] = model_2d.predict(X_2d)
model_2d.predict([[185, 3750]])
# since we have a 2d data matrix, we maintain the same
# row-column expectation for our inputs.
```

```
array([18.36187501])
```

Just like with SLR, we cam also extract the coefficient estimates using `.coef_` and `.intercept`. The reason why `.intercept` returns an array should now be more clear.

```
print(f"(sklearn) theta0: {model_2d.intercept_}")
print(f"(analytical) theta0 {theta_hat[2]}")
print(f"(sklearn) theta1: {model_2d.coef_[0]}")
print(f"(analytical) theta1 {theta_hat[0]}")
print(f"(sklearn) theta2: {model_2d.coef_[1]}")
print(f"(analytical) theta2 {theta_hat[1]}")
```

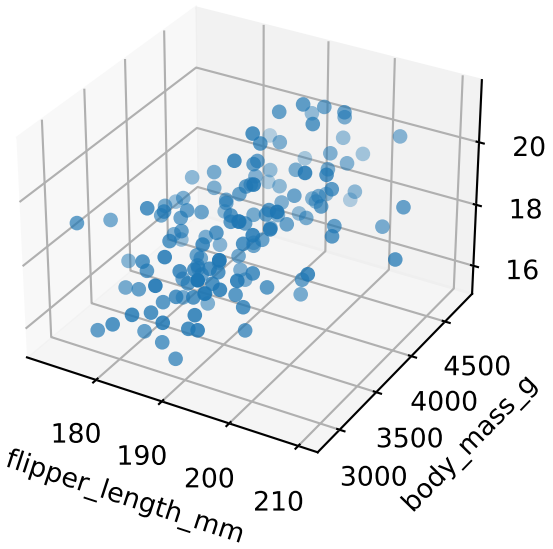
```
(sklearn) theta0: 11.002995277447067
(analytical) theta0 11.002995277445692
(sklearn) theta1: 0.009828486885248714
(analytical) theta1 0.009828486885249163
(sklearn) theta2: 0.0014774959083212898
(analytical) theta2 0.001477495908321363
```

14.1.2.3 Visualizing Our 2D Linear Model Predictions

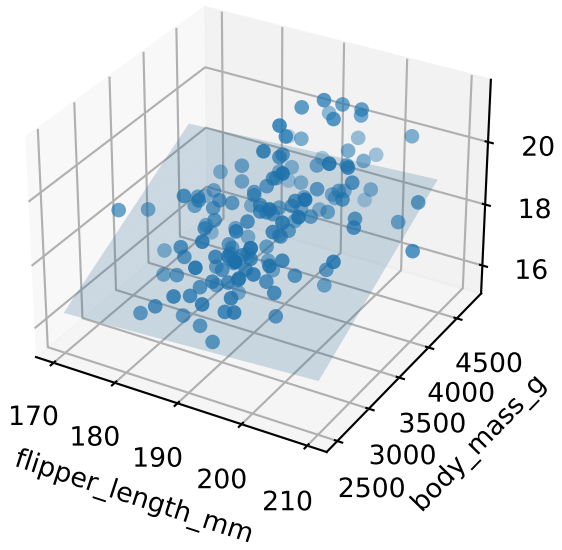
When we have two axis with which we can change an input, moving along this input plan creates a 2d plane with which we can get model outputs for. For example, for every single penguin with a `flipper length` , we also must specify a `body mass`. These two values in combination will help us predict the `bill depth`. Thus, we see that the predictions all lie in a 2d-plane. In higher dimensions, they all lie in a "hyperplane".

```
from mpl_toolkits.mplot3d import Axes3D  # noqa: F401 unused import
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(penguins["flipper_length_mm"], penguins["body_mass_g"], penguins["bill_depth_mm
plt.xlabel('flipper_length_mm')
plt.ylabel('body_mass_g')
```

```
Text(0.5, 0, 'body_mass_g')
```




```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(penguins["flipper_length_mm"], penguins["body_mass_g"], penguins["bill_depth_mm
xx, yy = np.meshgrid(range(170, 220, 10), range(2500, 4500, 100))
zz = ( 11.0029 + 0.00982 * xx + 0.001477 * yy) # thetas_using_sklearn
ax.plot_surface(xx, yy, zz, alpha=0.2)
plt.xlabel('flipper_length_mm')
plt.ylabel('body_mass_g')
plt.gcf().savefig("plane.png", dpi = 300, bbox_inches = "tight")
```



14.1.3 Loss Terminology

We use the word "loss" in two different (but very related) contexts in this course.

- In general, loss is the cost function that measures how far off model's prediction(s) is(are) from the actual value(s).
  - Per-datapoint loss is a cost function that measures the cost of y vs $\hat{y}$ for a particular datapoint.
  - Loss (without any adjectives) is generally a cost function measured across all datapoints. Often times, *empirical risk* is *average per-datapoint loss*.
- We prioritize using the latter term, because we don't particularly look at a given datapoint's loss when optimizing a model.
  - In other words, the dataset-level loss is the objective function that we'd like to minimize using gradient descent.
  - We achieve this minimization by using **per-datapoint** loss values.

14.2 Gradient Descent

At this point, we're fairly comfortable with fitting a regression model under MSE risk (indeed, we've done it three times now!). It's important to remember, however, that the results we've

found previously apply to one very specific case: the equations we used above are only relevant to a linear regression model using MSE as the cost function. In reality, we'll be working with a wide range of model types and objective functions, not all of which are as straightforward as the scenario we've discussed previously. This means that we need some more generalizable way of fitting a model to minimize loss.

To do this, we'll introduce the technique of **gradient descent**.

14.2.1 Minimizing a 1D Function

Let's shift our focus away from MSE to consider some new, arbitrary cost function. You can think of this function as outputting the empirical risk associated with some parameter `theta`.

14.2.1.1 The Naive Approach: Guess and Check

Above, we saw that the minimum is somewhere around 5.3ish. Let's see if we can figure out how to find the exact minimum algorithmically from scratch. One way very slow and terrible way would be manual guess-and-check.

```
def arbitrary(x):
    return (x**4 - 15*x**3 + 80*x**2 - 180*x + 144)/10

def simple_minimize(f, xs):
    # Takes in a function f and a set of values xs.
    # Calculates the value of the function f at all values x in xs
    # Takes the minimum value of f(x) and returns the corresponding value x
    y = [f(x) for x in xs]
    return xs[np.argmin(y)]

simple_minimize(arbitrary, np.linspace(1, 7, 20))
```

```
5.421052631578947
```

14.2.1.2 Scipy.optimize.minimize

One way to minimize this mathematical function is to use the `scipy.optimize.minimize` function. It takes a function and a starting guess and tries to find the minimum.

```
from scipy.optimize import minimize

# takes a function f and a starting point x0 and returns a readout
# with the optimal input value of x which minimizes f
minimize(arbitrary, x0 = 3.5)
```

```
     fun: -0.13827491292966557
 hess_inv: array([[0.73848255]])
     jac: array([6.48573041e-06])
 message: 'Optimization terminated successfully.'
    nfev: 20
     nit: 3
    njev: 10
  status: 0
 success: True
       x: array([2.39275266])
```

Our choice of start point can affect the outcome. For example if we start to the left, we get stuck in the local minimum on the left side.

```
minimize(arbitrary, x0 = 1)

#here we see the optimal value is different from before
```

```
     fun: -0.13827491294422317
 hess_inv: array([[0.74751575]])
     jac: array([-3.7997961e-07])
 message: 'Optimization terminated successfully.'
    nfev: 16
     nit: 7
    njev: 8
  status: 0
 success: True
       x: array([2.3927478])
```

`scipy.optimize.minimize` is great. It may also seem a bit magical. How could you write a function that can find the minimum of any mathematical function? There are a number of ways to do this, which we'll explore in today's lecture, eventually arriving at the important idea of **gradient descent**, which is the principle that `scipy.optimize.minimize` uses.

It turns out that under the hood, the `fit` method for `LinearRegression` models uses gradient descent. Gradient descent is also how much of machine learning works, including even advanced neural network models.

In Data 100, the gradient descent process will usually be invisible to us, hidden beneath an abstraction layer. However, to be good data scientists, it's important that we know the basic principles beyond the optimization functions that harness to find optimal parmaeters.

14.2.2 Digging into Gradient Descent

Looking at the function across this domain, it is clear that the function's minimum value occurs around $\theta = 5.3$. Let's pretend for a moment that we *couldn't* see the full view of the cost function. How would we guess the value of θ that minimizes the function?

It turns out that the first derivative of the function can give us a clue. In the plots below, the line indicates the value of the derivative of each value of θ. The derivative is negative where it is red and positive where it is green.

Say we make a guess for the minimizing value of θ. Remember that we read plots from left to right, and assume that our starting θ value is to the left of the optimal $\hat{\theta}$. If the guess "undershoots" the true minimizing value – our guess for θ is not quite at the value of the $\hat{\theta}$ that truly minimizes the function – the derivative will be **negative** in value. This means that if we increase θ (move further to the right), then we **can decrease** our loss function further. If this guess "overshoots" the true minimizing value, the derivative will be positive in value, implying the converse.

We can use this pattern to help formulate our next guess for the optimal $\hat{\theta}$. Consider the case where we've undershot θ by guessing too low of a value. We'll want our next guess to be greater in value than the previous guess – that is, we want to shift our guess to the right. You can think of this as following the slope "downhill" to the function's minimum value.

If we've overshot $\hat{\theta}$ by guessing too high of a value, we'll want our next guess to be lower in value – we want to shift our guess for $\hat{\theta}$ to the left.

14.3 Gradient Descent in 1 Dimension

These observations lead us to the **gradient descent update rule**:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L(\theta^{(t)})$$

Begin with our guess for $\hat{\theta}$ at timestep t. To find our guess for $\hat{\theta}$ at the next timestep, $t + 1$, subtract the objective function's derivative evaluted at $\theta^{(t)}$, $\frac{d}{d\theta} L(\theta^{(t)})$, scaled by a positive value α. We've replaced the generic function f with L to indicate that we are minimizing loss.

Supposing that any local minima is a global minimum (see **convexity** at the end of this page):

- If our guess $\theta^{(t)}$ is to the left of $\hat{\theta}$ (undershooting), the first derivative will be negative. Subtracting a negative number from $\theta^{(t)}$ will *increase* the value of the next guess, $\theta^{(t+1)}$ and move our loss function down. The guess will shift to the right.
- If our guess $\theta^{(t)}$ was too high (overshooting $\hat{\theta}$), the first derivative will be positive. Subtracting a positive number from $\theta^{(t)}$ will *decrease* the value of the next guess, $\theta^{(t+1)}$ and move our loss function down. The guess will shift to the left.

Put together, this captures the same behavior we reasoned through above. We repeatedly update our guess for the optimal θ until we've completed a set number of updates, or until each additional update iteration does not change the value of θ. In this second case, we say that gradient descent has **converged** on a solution.

The α term in the update rule is known as the **learning rate**. It is a positive value represents the size of each gradient descent update step – in other words, how "far" should we step to the left or right with each updated guess? A high value of α will lead to large differences in value between consecutive guesses for $\hat{\theta}$; a low value of α will result in smaller differences in value between consecutive guesses. This is the first example of a **hyperparameter**, a parameter that is hand picked by the data scientist that changes the model's behavior, in the course.

```
# define the derivative of the arbitrary function we want to minimize
def derivative_arbitrary(x):
    return (4*x**3 - 45*x**2 + 160*x - 180)/10

def gradient_descent(df, initial_guess, alpha, n):
    """Performs n steps of gradient descent on df using learning rate alpha starting
       from initial_guess. Returns a numpy array of all guesses over time."""
    guesses = [initial_guess]
    current_guess = initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)

    return np.array(guesses)

# calling our function gives us the path that gradient descent takes for 20 steps
# with a learning rate of 0.3 starting at theta = 4
trajectory = gradient_descent(derivative_arbitrary, 4, 0.3, 20)
trajectory
```

```
array([4.        , 4.12      , 4.26729664, 4.44272584, 4.64092624,
```

```
4.8461837 , 5.03211854, 5.17201478, 5.25648449, 5.29791149,
5.31542718, 5.3222606 , 5.32483298, 5.32578765, 5.32614004,
5.32626985, 5.32631764, 5.32633523, 5.3263417 , 5.32634408])
```

Above, we've simply run our algorithm a fixed number of times. More sophisticated implementations will stop based on a variety of different stopping criteria, e.g. error getting too small, error getting too large, etc. We will not discuss these in our course.

14.3.1 Application of 1D Gradient Descent

We've seen how to find the optimal parameters for a 1D linear model for the penguin dataset:

- Using the derived equations from Data 8.
- Using `sklearn`.
  - Uses gradient descent under the hood!

In real practice in this course, we'll usually use `sklearn`. But for now, let's see how we can do the gradient descent ourselves.

Let's consider a case where we have a linear model with no offset.

$$\hat{y} = \theta_1 x$$

We want to find the parameter θ_1 such that the L2 loss is minimized. In sklearn, this is easy. To avoid fitting an intercept, we set `fit_intercept` to false.

```
model = LinearRegression(fit_intercept = False)
df = sns.load_dataset("tips")
model.fit(df[["total_bill"]], df["tip"])
model.coef_ # the optimal tip percentage is 14.37%
```

```
array([0.1437319])
```

14.3.2 Creating an Explicit MSE Function

To employ gradient descent and do this ourselves, we need to define a function upon which we can use gradient descent. Suppose we select the L2 loss as our loss function. In this case, our goal will be to minimize the mean squared error.

Let's start by writing a function that computes the MSE for a given choice of θ_1 on our dataset.

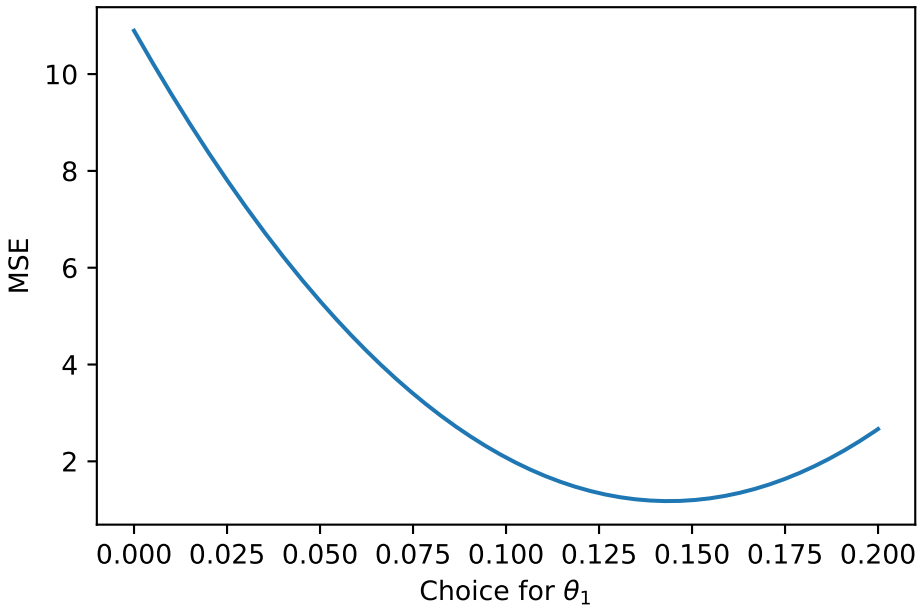
```
def mse_single_arg(theta1):
    """Returns the MSE on our data for the given theta1"""
    x = df["total_bill"]
    y_obs = df["tip"]
    y_hat = theta1 * x
    return np.mean((y_hat - y_obs) ** 2)

mse_single_arg(0.1437)
# The minimum loss value that we can achieve is 1.178 dollars on average away from the tru
```

```
1.1781165940051928
```

14.3.3 Plotting the MSE Function

Since we only have 1 parameter, we can simply cross reference our results with a simnple plot. We do not want to always do this since some functions can have thousands of inputs, making them difficult to plot. We can plot the MSE as a function of `theta1`. It turns out to look pretty smooth, and quite similar to a parabola.

```
theta1s = np.linspace(0, 0.2, 200)
x = df["total_bill"]
y_obs = df["tip"]

MSEs = [mse_single_arg(theta1) for theta1 in theta1s]

plt.plot(theta1s, MSEs)
plt.xlabel(r"Choice for $\theta_1$")
plt.ylabel(r"MSE");
```



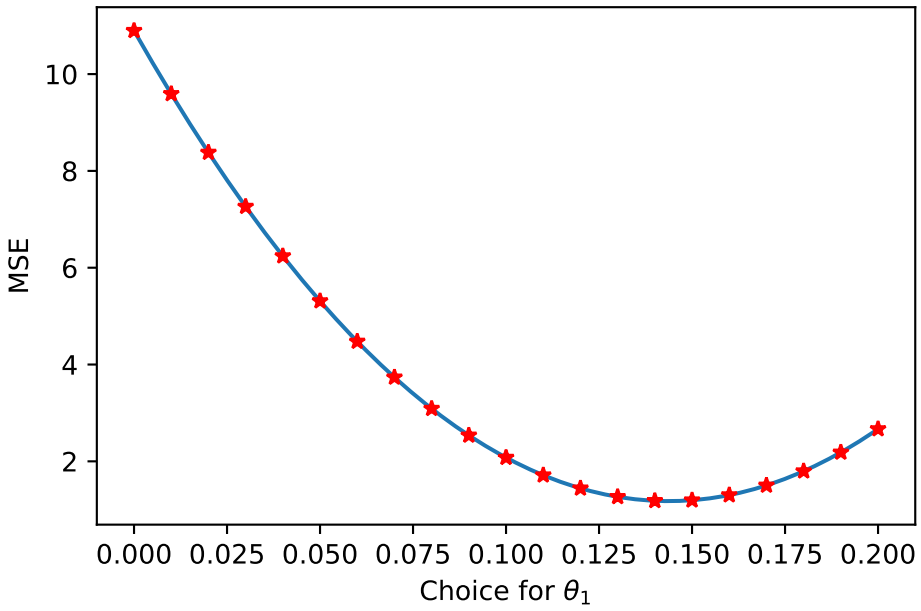
The minimum appears to be around $\theta_1 = 0.14$. We can once again check this naively.

```
simple_minimize(mse_single_arg, np.linspace(0, 0.2, 21))
```

```
0.14
```

As before, what we're doing is computing all the starred values below and then returning the θ_1 that goes with the minimum value.

```
theta1s = np.linspace(0, 0.2, 200)
sparse_theta1s = np.linspace(0, 0.2, 21)

loss = [mse_single_arg(theta1) for theta1 in theta1s]
sparse_loss = [mse_single_arg(theta1) for theta1 in sparse_theta1s]

plt.plot(theta1s, loss)
plt.plot(sparse_theta1s, sparse_loss, 'r*')
plt.xlabel(r"Choice for $\theta_1$")
plt.ylabel(r"MSE");
```



14.3.3.1 Using Scipy.Optimize.minimize

```
import scipy.optimize
from scipy.optimize import minimize
minimize(mse_single_arg, x0 = 0)
```

```
      fun: 1.1781161154513213
 hess_inv: array([[1]])
      jac: array([4.24683094e-06])
  message: 'Optimization terminated successfully.'
     nfev: 6
      nit: 1
     njev: 3
   status: 0
  success: True
        x: array([0.14373189])
```

14.3.3.2 Using Our Gradient Descent Function

Another approach is to use our 1D gradient descent algorithm from earlier. This is the exact same function as earlier. We can run it for 100 steps and see where it ultimately ends up.

```
def mse_loss_derivative_single_arg(theta_1):
    """Returns the derivative of the MSE on our data for the given theta1"""
    x = df["total_bill"]
    y_obs = df["tip"]
    y_hat = theta_1 * x

    return np.mean(2 * (y_hat - y_obs) * x)

gradient_descent(mse_loss_derivative_single_arg, 0.05, 0.0001, 100)[-5:]
```

```
array([0.14372404, 0.14372478, 0.14372545, 0.14372605, 0.1437266 ])
```

14.4 Multidimensional Gradient Descent

We're in good shape now: we've developed a technique to find the minimum value of a more complex objective function.

The function we worked with above was one-dimensional – we were only minimizing the function with respect to a single parameter, θ. However, as we've seen before, we often need to optimize a cost function with respect to several parameters (for example, when selecting the best model parameters for multiple linear regression). We'll need to extend our gradient descent rule to *multidimensional* objective functions.

Now suppose we improve our model so that we want to predict the tip from the total_bill plus a constant offset, in other words:

$$\text{tip} = \theta_0 + \theta_1 \text{bill}$$

To put this in more concrete terms what this means, let's return to the familiar case of simple linear regression with MSE loss.

$$\text{MSE}(\theta_0, \theta_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x)^2$$

Now, loss is expressed in terms of *two* parameters, θ_0 and θ_1. Rather than a one-dimensional loss function as we had above, we are now dealing with a two-dimensional **loss surface**.

```
# This code is for illustration purposes only
# It contains a lot of syntax you have not seen
import plotly.graph_objects as go
```

```
model = LinearRegression(fit_intercept = False)
df = sns.load_dataset("tips")
df["bias"] = 1
model.fit(df[["bias","total_bill"]], df["tip"])
model.coef_

uvalues = np.linspace(0, 2, 10)
vvalues = np.linspace(0, 0.2, 10)
(u,v) = np.meshgrid(uvalues, vvalues)
thetas = np.vstack((u.flatten(),v.flatten()))

X = df[["bias","total_bill"]].to_numpy()
Y = df["tip"].to_numpy()

def mse_loss_single_arg(theta):
    return mse_loss(theta, X, Y)

def mse_loss(theta, X, y_obs):
    y_hat = X @ theta
    return np.mean((y_hat - Y) ** 2)

MSE = np.array([mse_loss_single_arg(t) for t in thetas.T])

loss_surface = go.Surface(x=u, y=v, z=np.reshape(MSE, u.shape))

ind = np.argmin(MSE)
optimal_point = go.Scatter3d(name = "Optimal Point",
    x = [thetas.T[ind,0]], y = [thetas.T[ind,1]],
    z = [MSE[ind]],
    marker=dict(size=10, color="red"))

fig = go.Figure(data=[loss_surface, optimal_point])
fig.update_layout(scene = dict(
    xaxis_title = "theta0",
    yaxis_title = "theta1",
    zaxis_title = "MSE"))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Though our objective function looks a little different, we can use the same principles as we did earlier to locate the optimal model parameters. Notice how the minimum value of MSE, marked by the red dot in the plot above, occurs in the "valley" of the loss surface. Like before, we want our guesses for the best pair of (θ_0, θ_1) to move "downhill" towards this minimum point.

The difference now is that we need to update guesses for *both* θ_0 and θ_1 that minimize a loss function $L(\theta, \mathbb{X}, \mathbb{Y})$:

$$\theta_0^{(t+1)} = \theta_0^{(t)} - \alpha \frac{\partial L}{\partial \theta_0} \Big|_{\theta=\theta^{(t)}} \qquad \theta_1^{(t+1)} = \theta_1^{(t)} - \alpha \frac{\partial L}{\partial \theta_1} \Big|_{\theta=\theta^{(t)}}$$

We can tidy this statement up by using vector notation:

$$\begin{bmatrix} \theta_0^{(t+1)} \\ \theta_1^{(t+1)} \end{bmatrix} = \begin{bmatrix} \theta_0^{(t)} \\ \theta_1^{(t)} \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial \theta_0}|_{\theta=\theta^{(t)}} \\ \frac{\partial L}{\partial \theta_1}|_{\theta=\theta^{(t)}} \end{bmatrix}$$

To save ourselves from writing out long column vectors, we'll introduce some new notation. $\vec{\theta}^{(t)}$ is a column vector of guesses for each model parameter θ_i at timestep t. We call $\nabla_{\vec{\theta}} L$ the **gradient vector.** In plain English, it means "take the derivative of loss, $L(\theta, \mathbb{X}, \mathbb{Y})$, with respect to each model parameter in $\vec{\theta}$, and evaluate it at the given $\theta = \theta^{(t)}$."

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\theta^{(t)}, \mathbb{X}, \mathbb{Y})$$

14.4.1 Gradient Notation

Consider the 2D function:

$$f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$$

For a function of 2 variables, $f(\theta_0, \theta_1)$ we define the gradient as $\nabla_\theta f = \frac{\partial f}{\partial \theta_0}\vec{i} + \frac{\partial f}{\partial \theta_1}\vec{j}$, where $\vec{i}$ and $\vec{j}$ are the unit vectors in the θ_0 and θ_1 directions.

$$\frac{\partial f}{\partial \theta_0} = 16\theta_0 + 3\theta_1$$

$$\frac{\partial f}{\partial \theta_1} = 3\theta_0$$

$$\nabla_\theta f = (16\theta_0 + 3\theta_1)\vec{i} + (3\theta_0)\vec{j}$$

We can also write it in column vector notation.

$$\nabla_\theta f = \begin{bmatrix} \frac{\partial f}{\partial \theta_0} \\ \frac{\partial f}{\partial \theta_1} \\ \dots \end{bmatrix}$$

EX: $\nabla_\theta f = \begin{bmatrix} 16\theta_0 + 3\theta_1 \\ 3\theta_0 \end{bmatrix}$

You should read these gradients as:

- $\frac{\partial f}{\partial \theta_0}$: If I nudge the 1st model weight, what happens to loss?
- $\frac{\partial f}{\partial \theta_1}$:If I nudge the 2nd model weight, what happens to loss?
- etc.

14.4.2 Visualizing Gradient Descent

First, we need to be able to easily determine the gradient for any pair of values, θ_0, θ_1.

```
tips_with_bias = df.copy()
tips_with_bias["bias"] = 1
X = tips_with_bias[["bias", "total_bill"]]
X.head(5)

def mse_gradient(theta, X, y_obs):
    """Returns the gradient of the MSE on our data for the given theta"""
    x0 = X.iloc[:, 0]
    x1 = X.iloc[:, 1]
    dth0 = np.mean(-2 * (y_obs - theta[0]*x0 - theta[1]*x1) * x0)
    dth1 = np.mean(-2 * (y_obs - theta[0]*x0 - theta[1]*x1) * x1)
    return np.array([dth0, dth1])

def mse_gradient_single_arg(theta):
    """Returns the gradient of the MSE on our data for the given theta"""
    X = tips_with_bias[["bias", "total_bill"]]
    y_obs = tips_with_bias["tip"]
    return mse_gradient(theta, X, y_obs)

X = tips_with_bias[["bias", "total_bill"]]
y_obs = tips_with_bias["tip"]
ex1_mse = mse_gradient(np.array([0, 0]), X, y_obs)

print(f"Gradient for values theta0 = 0 and theta1 = 0 : {ex1_mse}")
```

```
Gradient for values theta0 = 0 and theta1 = 0 : [  -5.99655738 -135.22631803]
```

Using our previously previously defined `gradient_descent` function, we can see if our intuition extends to higher dimensions.

```
#print out the last 10 guesses our algorithm outputs to save space
guesses = gradient_descent(mse_gradient_single_arg, np.array([0, 0]), 0.001, 10000)[-10:]
```

14.5 Mini-Batch Gradient Decsent and Stochastic Gradient Descent

Formally, the algorithm we derived above is called **batch gradient descent.** For each iteration of the algorithm, the derivative of loss is computed across the entire batch of available data. While this update rule works well in theory, it is not practical in all circumstances. For large datasets (with perhaps billions of data points), finding the gradient across all the data is incredibly computationally taxing.

Mini-batch gradient descent tries to address this issue. In mini-batch descent, only a subset of the data is used to compute an estimate of the gradient. For example, we might consider only 10% of the total data at each gradient descent update step. At the next iteration, a different 10% of the data is sampled to perform the following update. Once the entire dataset has been used, the process is repeated. Each complete "pass" through the data is known as a **training epoch**. In practice, we choose the mini-batch size to be 32.

In the extreme case, we might choose a batch size of only 1 data point – that is, a single data point is used to estimate the gradient of loss with each update step. This is known as **stochastic gradient descent**.

Batch gradient descent is a deterministic technique – because the entire dataset is used at each update iteration, the algorithm will always advance towards the minimum of the loss surface. In contrast, both mini-batch and stochastic gradient descent involve an element of randomness. Since only a subset of the full data is used to update the guess for $\vec{\theta}$ at each iteration, there's a chance the algorithm will not progress towards the true minimum of loss with each update. Over the longer term, these stochastic techniques should still converge towards the optimal solution.

The diagrams below represent a "bird's eye view" of a loss surface from above. Notice that batch gradient descent takes a direct path towards the optimal $\hat{\theta}$. Stochastic gradient descent, in contrast, "hops around" on its path to the minimum point on the loss surface. This reflects the randomness of the sampling process at each update step.

14.6 Convexity

In our analysis above, we focused our attention on the global minimum of the loss function. You may be wondering: what about the local minimum just to the left?

If we had chosen a different starting guess for θ, or a different value for the learning rate α, we may have converged on the local minimum, rather than on the true optimum value of loss.

If the loss function is **convex**, gradient descent is guaranteed to find the global minimum of the objective function. Formally, a function f is convex if:

$$tf(a) + (1-t)f(b) \geq f(ta + (1-t)b)$$

To put this into words: if you drew a line between any two points on the curve, all values on the curve must be *on or below* the line. Importantly, any local minimum of a convex function is also its global minimum.

In summary, non-convex loss functions can cause problems with optimization. This means that our choice of loss function is an key factor in our modeling process. It turns out that MSE *is* convex, which is a major reason why it is such a popular choice of loss function.

15 Feature Engineering

> **Note**
>
> - Recognize the value of feature engineering as a tool to improve model performance
> - Implement polynominal feature generation and one hot encoding
> - Understand the interactions between model complexity, model variance, and training error

At this point in the course, we've equipped ourselves with some powerful techniques to build and optimize models. We've explored how to develop models of multiple variables, as well as how to fit these models to maximize their performance.

All of this was done with one major caveat: the regression models we've worked with so far are all **linear in the input variables**. We've assumed that our predictions should be some combination of linear variables. While this works well in some cases, the real world isn't always so straightforward. In today's lecture, we'll learn an important method to address this issue – and consider some new problems that can arise when we do so.

15.1 Feature Engineering

Feature Engineering is the process of *transforming* the raw features into *more informative features* that can be used in modeling or EDA tasks.

Feature engineering allows you to: Capture domain knowledge (e.g. periodicity or relationships between features). Express non-linear relationships using simple linear models. Encode non-numeric features to be used as inputs to models. Example: Using the country of origin of a car as an input to modeling its efficiency.

Why doesn't sklearn doesn't have SquareRegression /PolynomialRegression.

- We can translate these into linear models with features that are polynomials of x.
- Feature engineering saves `sklearn` a lot of redundancy in their library.
- Linear models have really nice properties.

15.2 Feature Functions

A feature function takes our original d dimensional input, $\mathbb{X}$, and transforms it into a d' dimensional input Φ.

For example, when we add the squared term of an existing column, we are effectively using kind of feature function, taking a $n \times 1$ matrix, $[hp]$, and turning it into an $n \times 2$ matrix $[hp, hp^2]$

As number of features grows, we can capture arbitrarily complex relationships.

Let's take a moment to dig further in to remind ourselves where this linearity comes from. Consider the following dataset on vehicles:

```
import seaborn as sns
vehicles = sns.load_dataset("mpg").rename(columns={"horsepower":"hp"}).dropna()
vehicles.head(5)
```

	mpg	cylinders	displacement	hp	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chevelle
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

Suppose we wish to develop a model to predict a vehicle's fuel efficiency (`"mpg"`) as a function of its horsepower (`"hp"`). Glancing at the plot below, we see that the relationship between `"mpg"` and `"hp"` is non-linear – an SLR fit doesn't capture the relationship between the two variables.

Recall our standard multiple linear regression model. In its current form, it is linear in terms of both θ_i and x:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x \ + \ ...$$

Just by eyeballing the `vehicle` data plotted above, it seems that a *quadratic* model might be more appropriate. In other words, a model of the form below would likely do a better job of capturing the non-linear relationship between the two variables:

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2$$

This looks fairly similar to our original multiple regression framework! Importantly, it is **still linear in** θ_i – the prediction $\hat{y}$ is a linear combination of the model parameters. This

means that we can use the same linear algebra methods as before to derive the optimal model parameters when fitting the model.

You may be wondering: how can this be a linear model if there is now a x^2 term? Although the model contains non-linear x terms, it is linear with respect to the *model parameters*, θ_i. Because our OLS derivation relied on assuming a linear model of θ_i, the method is still valid to fit this new model.

If we refit the model with `"hp"` squared as its own feature, we see that the model follows the data much more closely.

$$\hat{\text{mpg}} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp})^2$$

Looks much better! What we've done here is called **feature engineering**: the process of transforming the raw features of a dataset into more informative features for modeling. By squaring the `"hp"` feature, we were able to create a new feature that signficantly improved the quality of our model.

We perform feature engineering by defining a **feature function**. A feature function is some function applied to the original variables in the data to generate one or more new features. More formally, a feature function is said to take a d dimensional input and transform it to a p dimensional input. This results in a new, feature-engineered design matrix that we rename Φ.

$$\mathbb{X} \in \mathbb{R}^{n \times d} \longrightarrow \Phi \in \mathbb{R}^{n \times p}$$

In the `vehicles` example above, we applied a feature function to transform the original input with $d = 1$ features into an engineered design matrix with $p = 2$ features.

15.3 One Hot Encoding

Feature engineering opens up a whole new set of possibilities for designing better performing models. As you will see in lab and homework, feature engineering is one of the most important parts of the entire modeling process.

A particularly powerful use of feature engineering is to allow us to perform regression on non-numeric features. **One hot encoding** is a feature engineering technique that generates numeric features from categorical data, allowing us to use our usual methods to fit a regression model on the data.

To illustrate how this works, we'll refer back to the `tips` data from last lecture. Consider the `"day"` column of the dataset:

```
import numpy as np
np.random.seed(1337)
tips_df = sns.load_dataset("tips").sample(100)
tips_df[["day"]].head(5)
```

	day
54	Sun
46	Sun
86	Thur
199	Thur
106	Sat

At first glance, it doesn't seem possible to fit a regression model to this data – we can't directly perform any mathematical operations on the entry "Thur".

To resolve this, we instead create a new table with a feature for each unique value in the original `"day"` column. We then iterate through the `"day"` column. For each entry in `"day"` we fill the corresponding feature in the new table with 1. All other features are set to 0.

This can be implemented in code using `sklearn`'s `OneHotEncoder()` to generate the one hot encoding, then calling `pd.concat` to combine these new features with the original DataFrame.

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Perform the one hot encoding
oh_enc = OneHotEncoder()
oh_enc.fit(tips_df[['day']])
ohe_data = oh_enc.transform(tips_df[['day']]).toarray()


# Combine with original features
data_w_ohe = (tips_df
              .join(
                  pd.DataFrame(ohe_data, columns=oh_enc.get_feature_names_out(), index=tip
              )
data_w_ohe
```

	total_bill	tip	sex	smoker	day	time	size	day_Fri	day_Sat	day_Sun	day_Thu
54	25.56	4.34	Male	No	Sun	Dinner	4	0.0	0.0	1.0	0.0
46	22.23	5.00	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
86	13.03	2.00	Male	No	Thur	Lunch	2	0.0	0.0	0.0	1.0
199	13.51	2.00	Male	Yes	Thur	Lunch	2	0.0	0.0	0.0	1.0
106	20.49	4.06	Male	Yes	Sat	Dinner	2	0.0	1.0	0.0	0.0
87	18.28	4.00	Male	No	Thur	Lunch	2	0.0	0.0	0.0	1.0
110	14.00	3.00	Male	No	Sat	Dinner	2	0.0	1.0	0.0	0.0
166	20.76	2.24	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
237	32.83	1.17	Male	Yes	Sat	Dinner	2	0.0	1.0	0.0	0.0
103	22.42	3.48	Female	Yes	Sat	Dinner	2	0.0	1.0	0.0	0.0
185	20.69	5.00	Male	No	Sun	Dinner	5	0.0	0.0	1.0	0.0
69	15.01	2.09	Male	Yes	Sat	Dinner	2	0.0	1.0	0.0	0.0
163	13.81	2.00	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
157	25.00	3.75	Female	No	Sun	Dinner	4	0.0	0.0	1.0	0.0
187	30.46	2.00	Male	Yes	Sun	Dinner	5	0.0	0.0	1.0	0.0
67	3.07	1.00	Female	Yes	Sat	Dinner	1	0.0	1.0	0.0	0.0
211	25.89	5.16	Male	Yes	Sat	Dinner	4	0.0	1.0	0.0	0.0
238	35.83	4.67	Female	No	Sat	Dinner	3	0.0	1.0	0.0	0.0
126	8.52	1.48	Male	No	Thur	Lunch	2	0.0	0.0	0.0	1.0
68	20.23	2.01	Male	No	Sat	Dinner	2	0.0	1.0	0.0	0.0
121	13.42	1.68	Female	No	Thur	Lunch	2	0.0	0.0	0.0	1.0
56	38.01	3.00	Male	Yes	Sat	Dinner	4	0.0	1.0	0.0	0.0
119	24.08	2.92	Female	No	Thur	Lunch	4	0.0	0.0	0.0	1.0
180	34.65	3.68	Male	Yes	Sun	Dinner	4	0.0	0.0	1.0	0.0
74	14.73	2.20	Female	No	Sat	Dinner	2	0.0	1.0	0.0	0.0
85	34.83	5.17	Female	No	Thur	Lunch	4	0.0	0.0	0.0	1.0
101	15.38	3.00	Female	Yes	Fri	Dinner	2	1.0	0.0	0.0	0.0
53	9.94	1.56	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
159	16.49	2.00	Male	No	Sun	Dinner	4	0.0	0.0	1.0	0.0
41	17.46	2.54	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
178	9.60	4.00	Female	Yes	Sun	Dinner	2	0.0	0.0	1.0	0.0
11	35.26	5.00	Female	No	Sun	Dinner	4	0.0	0.0	1.0	0.0
35	24.06	3.60	Male	No	Sat	Dinner	3	0.0	1.0	0.0	0.0
175	32.90	3.11	Male	Yes	Sun	Dinner	2	0.0	0.0	1.0	0.0
173	31.85	3.18	Male	Yes	Sun	Dinner	2	0.0	0.0	1.0	0.0
25	17.81	2.34	Male	No	Sat	Dinner	4	0.0	1.0	0.0	0.0
206	26.59	3.41	Male	Yes	Sat	Dinner	3	0.0	1.0	0.0	0.0
98	21.01	3.00	Male	Yes	Fri	Dinner	2	1.0	0.0	0.0	0.0
89	21.16	3.00	Male	No	Thur	Lunch	2	0.0	0.0	0.0	1.0
36	16.31	2.00	Male	No	Sat	Dinner	3	0.0	1.0	0.0	0.0
94	22.75	3.25	Female	No	Fri	Dinner	2	1.0	0.0	0.0	0.0
63	18.29	3.76	Male	Yes	Sat	Dinner	4	0.0	1.0	0.0	0.0
28	21.70	4.30	Male	No	Sat	Dinner	2	0.0	1.0	0.0	0.0
24	19.82	3.18	Male	No	Sat	Dinner	2	0.0	1.0	0.0	0.0
124	12.48	2.52	Female	No	Thur	Lunch	2	0.0	0.0	0.0	1.0
168	10.59	1.61	Female	Yes	Sat	Dinner	2	0.0	1.0	0.0	0.0
107	25.21	4.29	Male	Yes	Sat	Dinner	2	0.0	1.0	0.0	0.0
116	29.93	5.07	Male	No	Sun	Dinner	4	0.0	0.0	1.0	0.0
160	21.50	3.50	Male	No	Sun	Dinner	4	0.0	0.0	1.0	0.0
43	9.68	1.32	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
42	13.94	3.06	Male	No	Sun	Dinner	2	0.0	0.0	1.0	0.0
30	9.55	1.45	Male	No	Sat	Dinner	2	0.0	1.0	0.0	0.0

Now, the "day" feature (or rather, the four new boolean features that represent day) can be used to fit a model.

15.4 Higher-order Polynomial Example

Let's return to where we started today: Creating higher-order polynomial features for the mpg dataset.

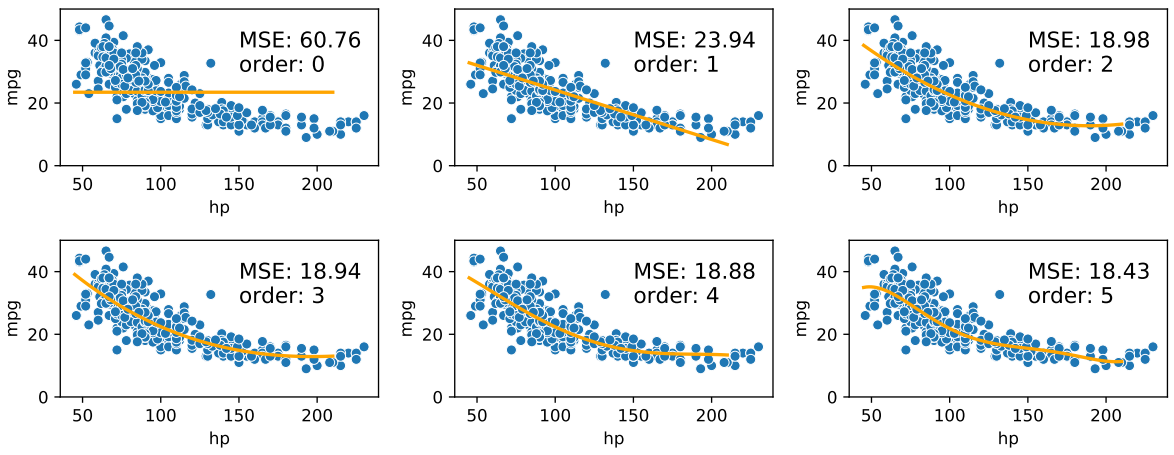
What happens if we add a feature corresponding to the horsepower, *cubed*? or to the fourth power? the fifth power?

- Will we get better results?
- What will the model look like?

Let's try it out. The below code plots polynomial models fit to the mpg dataset, from order 0 (the constant model) to order 5 (polynomial features through horsepower to the fifth power).

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt


vehicle_data = sns.load_dataset("mpg")
vehicle_data = vehicle_data.rename(columns = {"horsepower": "hp"})
vehicle_data = vehicle_data.dropna()


def get_MSE_for_degree_k_model(k):
    pipelined_model = Pipeline([
        ('poly_transform', PolynomialFeatures(degree = k)),
        ('regression', LinearRegression(fit_intercept = True))
    ])
    pipelined_model.fit(vehicle_data[["hp"]], vehicle_data["mpg"])
    return mean_squared_error(pipelined_model.predict(vehicle_data[["hp"]]), vehicle_data[

ks = np.array(range(0, 7))
MSEs = [get_MSE_for_degree_k_model(k) for k in ks]
MSEs_and_k = pd.DataFrame({"k": ks, "MSE": MSEs})
MSEs_and_k.set_index("k")
```

```
def plot_degree_k_model(k, MSEs_and_k, axs):
    pipelined_model = Pipeline([
        ('poly_transform', PolynomialFeatures(degree = k)),
        ('regression', LinearRegression(fit_intercept = True))
    ])
    pipelined_model.fit(vehicle_data[["hp"]], vehicle_data["mpg"])

    row = k // 3
    col = k % 3
    ax = axs[row, col]

    sns.scatterplot(data=vehicle_data, x='hp', y='mpg', ax=ax)

    x_range = np.linspace(45, 210, 100).reshape(-1, 1)
    ax.plot(x_range, pipelined_model.predict(pd.DataFrame(x_range, columns=['hp'])), c='or

    ax.set_ylim((0, 50))
    mse_str = f"MSE: {MSEs_and_k.loc[k, 'MSE']:.4}\norder: {k}"
    ax.text(150, 30, mse_str, dict(size=13))

fig = plt.figure(figsize=(10, 4))
axs = fig.subplots(nrows=2, ncols=3)

for k in range(6):
    plot_degree_k_model(k, MSEs_and_k, axs)
fig.tight_layout()
```



With constant and linear models, there seems to be a clear pattern in prediction error. With a quadratic model (order 2), the plot seems to match our data much more consistently across different `hp` values. For higher order polynomials, we observe a small improvement in MSE, but not much beyond 18.98. The MSE will continue to marginally decrease as we add more and more terms to our model. However, there ain't no free lunch. This decreasing of the MSE is coming at a major cost!

15.5 Variance and Training Error

We've seen now that feature engineering allows us to build all sorts of features to improve the performance of the model. In particular, we saw that designing a more complex feature (squaring `"hp"` in the `vehicles` data previously) substantially improved the model's ability to capture non-linear relationships. To take full advantage of this, we might be inclined to design increasingly complex features. Consider the following three models, each of different order (the maximum exponent power of each model):

- Model with order 1: $\widehat{\text{mpg}} = \theta_0 + \theta_1(\text{hp})$
- Model with order 2: $\widehat{\text{mpg}} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp})^2$
- Model with order 4: $\widehat{\text{mpg}} = \theta_0 + \theta_1(\text{hp}) + \theta_2(\text{hp})^2 + \theta_3(\text{hp})^3 + \theta_4(\text{hp})^4$

When we use our model to make predictions on the same data that was used to fit the model, we find that the MSE decreases with increasingly complex models. The **training error** is the model's error when generating predictions from the same data that was used for training purposes. We can conclude that the training error goes down as the complexity of the model increases.

This seems like good news – when working on the **training data**, we can improve model performance by designing increasingly complex models.

However, high model complexity comes with its own set of issues. When a model has many complicated features, it becomes increasingly sensitive to the data used to fit it. Even a small variation in the data points used to train the model may result in wildly different results for the fitted model. The plots below illustrate this idea. In each case, we've fit a model to two very similar sets of data (in fact, they only differ by two data points!). Notice that the model with order 2 appears roughly the same across the two sets of data; in contrast, the model with order 4 changes erratically across the two datasets.

The sensitivity of the model to the data used to train it is called the **model variance**. As we saw above, model variance tends to increase with model complexity.

We will further explore this tradeoff (and more precisely define model variance) in future lectures.

15.6 Overfitting

We can see that there is a clear "trade-off" that comes from the complexity of our model. As model complexity increases, the model's error on the training data decreases. At the same time, the model's variance tends to increase.

Why does this matter? To answer this question, let's take a moment to review our modeling workflow when making predictions on new data.

1. Sample a dataset of training data from the real world
2. Use this training data to fit a model
3. Apply this fitted model to generate predictions on unseen data

This first step – sampling training data – is important to remember in our analysis. As we saw above, a highly complex model may produce results that vary wildly across different samples of training data. If we happen to sample a set of training data that is a poor representation of the population we are trying to model, our model may perform poorly on any new set of data it has not seen before.

To see why, consider a model fit using the training data shown on the left. Because the model is so complex, it achieves zero error on the training set – it perfectly predicts each value in the training data! When we go to use this model to make predictions on a new sample of data, however, things aren't so good. The model now has enormous error on the unseen data.

The phenomenon above is called **overfitting**. The model effectively just memorized the training data it encountered when it was fitted, leaving it unable to handle new situations.

The takeaway here: we need to strike a balance in the complexity of our models. A model that is too simple won't be able to capture the key relationships between our variables of interest; a model that is too complex runs the risk of overfitting.

This begs the question: how do we control the complexity of a model? Stay tuned for the next lecture.

16 Cross Validation and Regularization

Consider the question: how do we control the complexity of our model? To answer this, we must know precisely when our model begins to overfit. The key to this lies in evaluating the model on unseen data using a process called Cross-Validation. A second point this note will address is how to combat overfitting – namely, through a technique known as regularization.

From the last lecture, we learned that *increasing* model complexity *decreased* our model's training error but increased variance. This makes intuitive sense; adding more features causes our model to better fit the given data, but generalize worse to new data. For this reason, a low training error is not representative of our model's underlying performance – this may be a side effect of overfitting.

Truly, the only way to know when our model overfits is by evaluating it on unseen data. Unfortunately, that means we need to wait for more data. This may be very expensive and time consuming.

How should we proceed? In this section, we will build up a viable solution to this problem.

16.0.1 The Holdout Method

The simplest approach to avoid overfitting is to keep some of our data secret from ourselves. This is known as the **holdout method**. We will train our models on *most* of the available data points – known as the **training data** . We'll then evaluate the models' performance on the unseen data points (called the **validation set**) to measure overfitting.

Imagine the following example where we wish to train a model on 35 data points. We choose the training set to be a random sample of 25 of these points, and the validation set to be the remaining 10 points. Using this data, we train 7 models, each with a higher polynomial degree than the last.

We get the following mean squared error (MSE) on the training data.

Using these same models, we compute the MSE on our 10 validation points and observe the following.

Notice how the training error monotonically *decreases* as polynomial degree *increases*. This is consistent with our knowledge. However, validation error first *decreases*, then *increases* (from $k = 3$). These higher degree models are performing worse on unseen data – this indicates they are overfitting. As such, the best choice of polynomial degree in this example is $\mathbf{k = 2}$.

More generally, we can represent this relationship with the following diagram.

Our goal is to train a model with complexity near the red line. Note that this relationship is a simplification of the real-world. But for purposes of Data 100, this is good enough.

16.0.1.1 Hyperparameters

In machine learning, a **hyperparameter** is a value that controls the learning process. In our example, we built seven models, each of which had a hyperparameter `k` that controlled the polynomial degree of the model.

To choose between hyperparameters, we use the validation set. This was evident in our example above; we found that `k` = 2 had the lowest validation error. However, this holdout method is a bit naive. Imagine our random sample of 10 validation points coincidentally favored a higher degree polynomial. This would have led us to incorrectly favor a more complex model. In other words, our small amount of validation data may be different from real-world data.

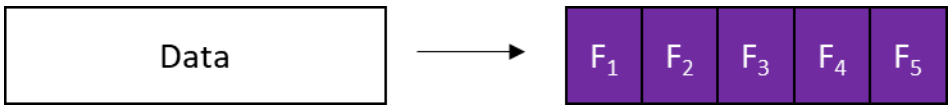
To minimize this possiblity, we need to evaluate our model on more data. Decreasing the size of the training set is not an option – doing so will worsen our model. How should we proceed?

16.0.2 K-Fold Cross Validation

In the holdout method, we train a model on *only* the training set, and assess the quality *only* on the validation set. On the other hand, **K-Fold Cross Validation** is a technique that determines the quality of a hyperparameter by evaluating a model-hyperparameter combination on `k` independent "folds" of data, which together make up the *entire* dataset. This is a more robust alternative to the holdout method. Let's break down each step.

Note: The `k` in k-fold cross validation is different from the `k` polynomial degree discussed in the earlier example.

In the k-fold cross-validation approach, we split our data into `k` equally sized groups (often called folds). In the example where `k` = 5:

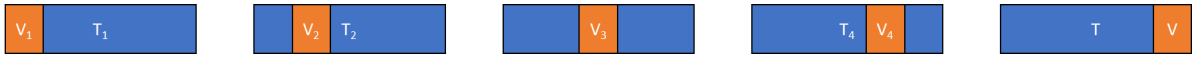


To determine the "quality" of a particular hyperparameter:

- Pick a fold, which we'll call the validation fold. Train a model on all other `k-1` folds. Compute an error on the validation fold.
- Repeat the step above for all `k` possible choices of validation fold, each time training a new model.

- Average the `k` validation fold errors. This will give a single error for the hyperparameter.

For `k` = 5, we have the following partitions. At each iteration, we train a model on the blue data and validate on the orange data. This gives us a total of 5 errors which we average to obtain a single representative error for the hyperparameter.

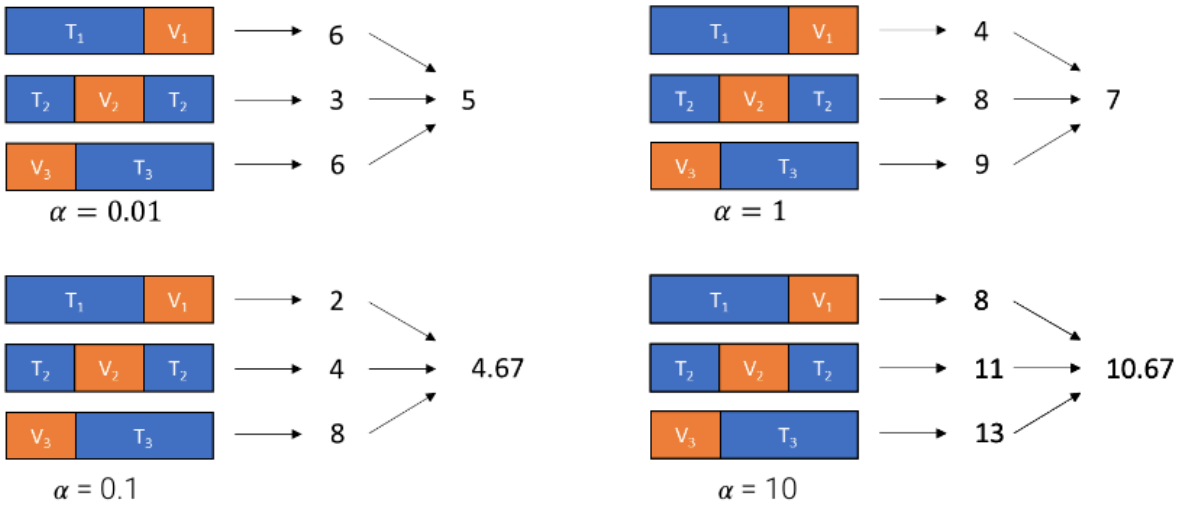


Note that the value of the hyperparameter is fixed during this process. By doing so, we can be confident in the hyperparameter's performance on the *entire* dataset. To compare multiple choices of a hyperparameter –say `m` choices of hyperparameter– we run k-fold cross validation `m` times. The smallest of the `m` resulting errors corresponds to the best hyperparameter value.

16.0.2.1 Hyperparameter Selection Example

K-fold cross validation can aid in choosing the best hyperparameter values in respect to our model and loss function.

Consider an example where we run k-fold cross validation with `k` = 3. We are implementing a model that depends on a hyperparameter α, and we are searching for an α that minimizes our loss function. We have narrowed down our hyperparameters such that $\alpha = [0.01, 0.1, 1, 10]$.



The losses of the model are shown per `k` fold of training data (arrows are pointing to the loss value) for each value of α. The average loss over the k-fold cross validation is displayed to the right of all the k-fold losses.

To determine the best α value, we must compare the average loss over the k-folds of training data. $\alpha = 0.01$ yields us an average loss of 5, $\alpha = 0.1$ yields us an average loss of 4.67, $\alpha = 1$ yields us an average loss of 7, and $\alpha = 10$ yields us an average loss of 10.67.

Thus, we would select $\alpha = 0.1$ as our hyperparameter value as it results in the lowest average loss over the k-folds of training data out of our possible α values.

16.0.2.2 Picking K

Typical choices of `k` are 5, 10, and N, where N is the number of data points.

`k` = N is known as "leave one out cross validation", and will typically give you the best results.

- In this approach, each validation set is only one point.
- Every point gets a chance to get used as the validation set.

However, `k` = N is very expensive and requires you to fit a large number of models.

16.0.3 Test Sets

Suppose we're researchers building a state of the art regression model. We choose a model with the lowest validation error and want to report this out to the world. Unfortunately, our validation error may be biased; that is, not representative of error on real-world data. This is because during our hyperparameter search, we were implicitly "learning" from our validation data by tuning our model to achieve better results. Before reporting our results, we should run our model on a special **test set** that we've never seen or used for any purpose whatsoever.

A test set can be something that we generate ourselves, or it can be a common dataset whose true values are unknown. In the case of the former, we can split our our available data into 3 partitions: the training set, testing set, and validation set. The exact amount of data allocated to each partition varies, but a common split is 80% train, 10% test, 10% validation.

Below is an implementation of extracting a training, testing and validation set using `sklearn.train_test_split` on a data matrix `X` and an array of observations `y`.

```
X_train_valid, X_test, y_train_valid, y_test = train_test_split(X, y, test_size=0.1)

X_train, X_val, y_train, y_val = train_test_split(X_train_valid, y_train_valid,
test_size=0.11)
```

As a recap:

- Training set used to pick parameters.
- Validation set used to pick hyperparameters (or pick between different models).
- Test set used to provide an unbiased error at the end.

Here is an idealized relationship between training error, test error, and validation error.

Notice how the test error behaves similarily to the validation error. Both come from data that is unseen during the model training process, so both are fairly good estimates of real-world data. Of the two, the test error is more unbiased for the reasons mentioned above.

As before, the optimal complexity level exists where validation error is minimized. Logically, we can't design a model that minimizes test error because we don't use the test set until final evaluation.

16.1 Regularization

Earlier, we found an optimal model complexity by choosing the hyperparameter that minimized validation error. This was the polynomial degree `k`= 2. Tweaking the "complexity" was simple; it was only a matter of adjusting the polynomial degree.

However, in most machine learning problems, complexity is defined differently. Today, we'll explore two different definitions of complexity: the *squared* and *absolute* magnitude of θ_i coefficients.

16.1.0.1 Constraining Gradient Descent

Before we discuss these definitions, let's first familiarize ourselves with the concept of **constrained gradient descent**. Imagine we have a two feature model with coeffiecient weights of θ_1 and θ_2. Below we've plotted a two dimensional contour plot of the OLS loss surface – darker areas indicate regions of lower loss. Gradient descent will find the optimal paramaters during training $(\theta_1, \theta_2) = \hat{\theta}_{NoReg.}$.

Suppose we arbitrarily decide that gradient descent can never land outside of the green ball.

Gradient descent finds a new solution at $\hat{\theta}_{Reg.}$. This is far from the global optimal solution $\hat{\theta}_{NoReg.}$ – however, it is the closest point that lives in the green ball. In other words, $\hat{\theta}_{Reg.}$ is the **constrained optimal solution**.

The size of this ball is completely arbitrary. Increasing its size allows for a constrained solution closer to the global optimum, and vice versa.

In fact, the size of this ball is inherently linked to model complexity. A smaller ball constrains $(\theta_1,\ \theta_2)$ more than a larger ball. This is synonymous with the behavior of a *less* complex model, which finds a solution farther from the optimum. A *larger* ball, on the other hand, is synonymous to a *more* complex model that can achieve a near optimal solution.

Consider the extreme case where the radius is infinitely small. The solution to every constrained modeling problem would lie on the origin, at $(\theta_1, \theta_2) = (0, 0)$. This is equivalent to

the constant model we studied – the least complex of all models. In the case where the radius is infinitely large, the optimal constrained solution exists at $\hat{\theta}_{NoReg.}$ itself! This is the solution obtained from OLS with no limitations on complexity.

The intercept coefficient is typically *not* constrained; θ_0 can be any value. This way, if all $\theta_i = 0$ except θ_0, the resulting model is a constant model (and θ_0 is the mean of all observations).

16.1.1 L2 Regularization

16.1.1.1 The Constrained Form

Regularization is the formal term that describes the process of limiting a model's complexity. This is done by constraining the solution of a cost function, much like how we constrained the set of permissible (θ_1, θ_2) above. **L2 Regularization**, commonly referred to as **Ridge Regression**, is the technique of constraining our model's parameters to lie within a *ball* around the origin. Formally, it is defined as

$$\min_{\theta} \frac{1}{n}||Y - X\theta||$$

such that $\sum_{j=1}^{d} \theta_j^2 \leq Q$

The mathematical definition of complexity in Ridge Regression is $\sum_{j=1}^{d} \theta_j^2 \leq Q$. This formulation of complexity limits the total squared magnitude of the coefficients to some constant Q. In two dimensional space, this is $\theta_1^2 + \theta_2^2 \leq Q$. You'll recognize this as the equation of a circle with axes θ_1, θ_2 and radius Q. In higher dimensions, this circle becomes a hypersphere and is conventionally referred to as the **L2 norm ball**. Decreasing Q shrinks the norm ball, and limits the complexity of the model (discussed in the previous section). Likewise, expanding the norm ball increases the allowable model complexity.

Without the constraint $\sum_{j=1}^{d} \theta_j^2 \leq Q$, the optimal solution is $\hat{\theta} = \hat{\theta}_{NoReg.}$. With an appropriate value of Q applied to the constraint, the solution $\hat{\theta} = \hat{\theta}_{Reg.}$ is sub-optimal on the training data but generalizes better to new data.

16.1.1.2 The Functional Form

Unfortunately, the function above requires some work. It's not easy to mathematically optimize over a constraint. Instead, in most machine learning text, you'll see a different formulation of Ridge Regression.

$$\min_{\theta} \frac{1}{n}||Y - X\theta|| + \alpha \sum_{j=1}^{d} \theta_j^2$$

These two equations are equivalent by Lagrangian Duality (not in scope).

Notice that we've replaced the constraint with a second term in our cost function. We're now minimizing a function with a regularization term that penalizes large coefficients. The α factor controls the degree of regularization. In fact, $\alpha \approx \frac{1}{Q}$. To understand why, let's consider these 2 extreme examples:

- Assume $\alpha \to \infty$. Then, $\alpha \sum_{j=1}^{d} \theta_j^2$ dominates the cost function. To minimize this term, we set $\theta_j = 0$ for all $j \geq 1$. This is a very constrained model that is mathematically equivalent to the constant model. Earlier, we explained the constant model also arises when the L2 norm ball radius $Q \to 0$.
- Assume $\alpha \to 0$. Then, $\alpha \sum_{j=1}^{d} \theta_j^2$ is infinitely small. Minimizing the cost function is equivalent to $\min_{\theta} \frac{1}{n}||Y - X\theta||$. This is just OLS, and the optimal solution is the global minimum $\hat{\theta} = \hat{\theta}_{NoReg.}$. We showed that the global optimum is achieved when the L2 norm ball radius $Q \to \infty$.

16.1.1.3 Closed Form Solution

An additional benefit to Ridge Regression is that it has a closed form solution.

$$\hat{\theta}_{ridge} = (X^T X + n\alpha I)^{-1} X^T Y$$

This solution exists even if there is linear dependence in the columns of the data matrix. We will not derive this result in Data 100, as it involves a fair bit of matrix calculus.

16.1.1.4 Implementation of Ridge Regression

Of course, `sklearn` has a built-in implementation of Ridge Regression. Simply import the `Ridge` class of the `sklearn.linear_model` library.

```
from sklearn.linear_model import Ridge
```

We will various Ridge Regression models on the familiar `vehicles` DataFrame from last lecture. This will help solidfy some of the theoretical concepts discussed earlier.

```
import pandas as pd
vehicles = pd.read_csv("data/vehicle_data.csv", index_col=0)
vehicles_mpg = pd.read_csv("data/vehicle_mpg.csv", index_col=0)
```

```
vehicles.head(5)
```

	cylinders	displacement	horsepower	weight	acceleration	cylinders^2	displacement^2	horsepower
0	8	307.0	130.0	3504	12.0	64	94249.0	1690
1	8	350.0	165.0	3693	11.5	64	122500.0	2722
2	8	318.0	150.0	3436	11.0	64	101124.0	2250
3	8	304.0	150.0	3433	12.0	64	92416.0	2250
4	8	302.0	140.0	3449	10.5	64	91204.0	1960

Here, we fit an extremeley regularized model without an intercept. Note the small coefficient values.

```
ridge_model_large_reg = Ridge(alpha = 10000)
ridge_model_large_reg.fit(vehicles, vehicles_mpg)
ridge_model_large_reg.coef_
```

```
array([[ 8.56292915e-04, -5.92399474e-02, -9.81013894e-02,
        -9.66985253e-03, -5.08353226e-03,  1.49576895e-02,
         1.04959034e-04,  1.14786826e-04,  9.07086742e-07,
        -4.60397349e-04]])
```

Note how Ridge Regression effectively **spreads a small weight across many features**.

When we apply very little regularization, our coefficients increase in size. Notice how they are identical to the coefficients retrieved from the `LinearRegression` model. This indicates the radius Q of the L2 norm ball is massive and encompasses the unregularized optimal solution. Once again, we see that α and Q are inversely related.

```
ridge_model_small_reg = Ridge(alpha = 10**-5)
ridge_model_small_reg.fit(vehicles, vehicles_mpg)
ridge_model_small_reg.coef_
```

```
array([[-8.06754383e-01, -6.32025048e-02, -2.92851012e-01,
        -3.41032156e-03, -1.43877512e+00,  1.25829303e-01,
         7.72841216e-05,  6.99398090e-04,  3.11031744e-07,
         3.16084838e-02]])
```

```
from sklearn.linear_model import LinearRegression
linear_model = LinearRegression()
linear_model.fit(vehicles, vehicles_mpg)
linear_model.coef_
```

```
array([[-8.06756280e-01, -6.32024872e-02, -2.92851021e-01,
        -3.41032211e-03, -1.43877559e+00,  1.25829450e-01,
         7.72840884e-05,  6.99398114e-04,  3.11031832e-07,
         3.16084973e-02]])
```

16.1.2 Scaling Data for Regularization

One issue with our approach is that our features are on vastly different scales. For example, `weight^2` is in the millions, while the number of `cylinders` are under 10. Intuitively, the coefficient value for `weight^2` must be very small to offset the large magnitude of the feature. On the other hand, the coefficient of the `cylinders` feature is likely quite large in comparison. We see these claims are true in the `LinearRegression` model above.

However, a problem arises in Ridge Regression. If we constrain our coefficients to a small region around the origin, we are unfairly restricting larger coefficients – like that of the `cylinders` feature. A smaller coefficient – that of the `weight^2` feature – likely lies within this region, so the value changes very little. Compare the coefficients of the regularized and unregularized `Ridge` models above, and you'll see this is true.

Therefore, it's imperative to standardize your data. We can do so using z-scores.

$$z_k = \frac{x_k - u_k}{\sigma_k}$$

You'll do this on lab 8 using the "StandardScaler" transformer. The resulting model coefficients will be all on the same scale.

16.1.3 L1 Regularization

L1 Regularization, commonly referred to as **Lasso Regression**, is an alternate regularization technique that limits the the *absolute* sum of θ_i coefficients.

16.1.3.1 The Constrained Form

$$\min_{\theta} \frac{1}{n}||Y - X\theta||$$

such that $\sum_{j=1}^{d} |\theta_j| \leq Q$

In two dimensions, our constraint equation is $|\theta_1| + |\theta_2| \leq Q$. This is the graph of a diamond centered on the origin with endpoints Q units away on each axis.

16.1.3.2 The Functional Form

A more convenient way to express Lasso Regression is as follows:

$$\min_{\theta} \frac{1}{n}||Y - X\theta|| + \alpha \sum_{j=1}^{d} |\theta_j|$$

As with Ridge Regression, the hyperparameter α has the same effect on Lasso Regression. That is, *increasing* α (equivalently, *decreasing* Q) *increases* the amount of regularization, and vice versa.

Unfortunately, Lasso Regression does not have a closed form solution – the cost function is not differentiable everywhere. Specifically, the sum $\sum_{j=1}^{d} |\theta_j|$ is problematic because it is composed of absolute value functions, each of which are non-differentiable at the origin.

So why use Lasso Regression? As we'll see shortly, it is great at implicit **feature selection**.

16.1.3.3 Implementation of Lasso Regression

Lasso Regression is great at reducing complexity by eliminating the least important features in a model. It does so by setting their respective feature weights to 0. See the following example.

```
from sklearn.linear_model import Lasso
lasso_model = Lasso(alpha = 1)

standardized_vehicles=(vehicles-vehicles.mean())/vehicles.std()
lasso_model.fit(standardized_vehicles, vehicles_mpg)
lasso_model.coef_
```

```
array([-0.14009981, -0.28452369, -1.14351999, -4.11329618,  0.        ,
       -0.        , -0.        , -0.        , -0.        ,  0.        ])
```

Notice how we standardized our data first. Lasso Regression then set the coefficients of our squared features to 0 – presumably, these are the least important predictors of `mpg`.

16.1.4 Summary of Regularization Methods

A summary of our regression models is shwon below:

Name	Model	Loss	Reg.	Objective	Solution
OLS	$\hat{\mathbb{Y}} = \mathbb{X}\theta$	Squared loss	None	$\frac{1}{n}\|\|\mathbb{Y} - \mathbb{X}\theta\|\|_2^2$	$\hat{\theta}_{\text{OLS}} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y}$
Ridge Regression	$\hat{\mathbb{Y}} = \mathbb{X}\theta$	Squared loss	L2	$\frac{1}{n}\|\|\mathbb{Y} - \mathbb{X}\theta\|\|_2^2 + \lambda\sum_{j=1}^{d}\theta_i^2$	$\hat{\theta}_{\text{ridge}} = (\mathbb{X}^T\mathbb{X} + n\lambda I)^{-1}\mathbb{X}^T\mathbb{Y}$
LASSO	$\hat{\mathbb{Y}} = \mathbb{X}\theta$	Squared loss	L1	$\frac{1}{n}\|\|\mathbb{Y} - \mathbb{X}\theta\|\|_2^2 + \lambda\sum_{j=1}^{d}\|\theta_i\|$	**No closed form**

Understanding the distinction between Ridge Regression and Lasso Regression is important. We've provided a helpful visual that summarizes the key differences.

This diagram displays the L1 and L2 constrained solution for various orientations of the OLS loss surface. Notice how the L1 (Lasso) solution almost always lies on some axis, or edge of the diamond. Graphically, this makes sense; the edges of the diamond are the farthest from the origin, and usually closest to the global optimum. When this happens, only one feature has a non-zero coefficient; this "feature selection" argument extends quite nicely to multiple features in higher dimensional space.

The L2 (Ridge) solution, however, typically has an optimal solution in some quadrant of the graph. Every point on the circumference of the L2 norm ball is equidistant from the origin, and thus similar in distance to the global optimum. As such, this technique of regularization is great at distributing a small weight across both features.

17 Probability I

> **Note**
>
> - Define a random variable in terms of its distribution
> - Compute the expectation and variance of a random variable
> - Gain familiarity with the Bernoulli and binomial random variables

In the past few lectures, we've examined the role of complexity in influencing model performance. We've considered model complexity in the context of a tradeoff between two competing factors: model variance and training error.

Thus far, our analysis has been mostly qualitative. We've acknowledged that our choice of model complexity needs to strike a balance between model variance and training error, but we haven't yet discussed *why* exactly this tradeoff exists.

To better understand the origin of this tradeoff, we will need to introduce the language of **random variables**. The next two lectures of probability will be a brief digression from our work on modeling so we can build up the concepts needed to understand this so-called **bias-variance tradeoff**. Our roadmap for the next few lectures will be:

1. Probability I: introduce random variables, considering the concepts of expectation, variance, and covariance
2. Probability II: re-express the ideas of model variance and training error in terms of random variables and use this new pespective to investigate our choice of model complexity

Let's get to it.

17.1 Random Variables and Distributions

Suppose we generate a set of random data, like a random sample from some population. A random variable is a numerical function of the randomness in the data. It is *random* from the randomness of the sample; it is *variable* because its exact value depends on how this random sample came out. We typically denote random variables with uppercase letters, such as X or Y.

To give a concrete example: say we draw a random sample s of size 3 from all students enrolled in Data 100. We might then define the random variable X to be the number of Data Science majors in this sample.

The **distribution** of a random variable X describes how the total probability of 100% is split over all possible values that X could take. If X is a **discrete random variable** with a finite number of possible values, define its distribution by stating the probability of X taking on some specific value, x, for all possible values of x.

The distribution of a discrete variable can also be represented using a histogram. If a variable is **continuous** – it can take on infinitely many values – we can illustrate its distribution using a density curve.

Often, we will work with multiple random variables at the same time. In our example above, we could have defined the random variable X as the number of Data Science majors in our sample of students, and the variable Y as the number of Statistics majors in the sample. For any two random variables X and Y:

- X and Y are **equal** if $X(s) = Y(s)$ for every sample s. Regardless of the exact sample drawn, X is always equal to Y.
- X and Y are **identically distributed** if the distribution of X is equal to the distribution of Y. That is, X and Y take on the same set of possible values, and each of these possible values is taken with the same probability. On any specific sample s, identically distributed variables do *not* necessarily share the same value.
- X and Y are **independent and identically distributed (IID)** if 1) the variables are identically distributed and 2) knowing the outcome of one variable does not influence our belief of the outcome of the other.

17.2 Expectation and Variance

Often, it is easier to describe a random variable using some numerical summary, rather than fully defining its distribution. These numerical summaries are numbers that characterize some properties of the random variable. Because they give a "summary" of how the variable tends to behave, they are *not* random – think of them as a static number that describes a certain property of the random variable. In Data 100, we will focus our attention on the expectation and variance of a random variable.

17.2.1 Expectation

The **expectation** of a random variable X is the weighted average of the values of X, where the weights are the probabilities of each value occurring. To compute the expectation, we find

each value x that the variable could possibly take, weight by the probability of the variable taking on each specific value, and sum across all possible values of x.

$$\mathbb{E}[X] = \sum_{\text{all possible } x} xP(X = x)$$

An important property in probability is the **linearity of expectation**. The expectation of the linear transformation $aX + b$, where a and b are constants, is:

$$\mathbb{E}[aX + b] = aE[\mathbb{X}] + b$$

Expectation is also linear in *sums* of random variables.

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

17.2.2 Variance

The **variance** of a random variable is a measure of its chance error. It is defined as the expected squared deviation from the expectation of X. Put more simply, variance asks: how far does X typically vary from its average value? What is the spread of X's distribution?

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$$

If we expand the square and use properties of expectation, we can re-express this statement as the **computational formula for variance**. This form is often more convenient to use when computing the variance of a variable by hand.

$$\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

How do we compute the expectation of X^2? Any function of a random variable is *also* a random variable – that means that by squaring X, we've created a new random variable. To compute $\mathbb{E}[X^2]$, we can simply apply our definition of expectation to the random variable X^2.

$$\mathbb{E}[X^2] = \sum_{\text{all possible } x} x^2 P(X^2 = x^2)$$

Unlike expectation, variance is *non-linear*. The variance of the linear transformation $aX + b$ is:

$$\text{Var}(aX + b) = a^2\text{Var}(X)$$

The full proof of this fact can be found using the definition of variance. As general intuition, consider that $aX + b$ scales the variable X by a factor of a, then shifts the distribution of X by b units.

- Shifting the distribution by b *does not* impact the *spread* of the distribution. Thus, $\text{Var}(aX + b) = \text{Var}(aX)$.
- Scaling the distribution by a *does* impact the spread of the distribution.

If we wish to understand the spread in the distribution of the *summed* random variables $X+Y$, we can manipulate the definition of variance to find:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

This last term is of special significance. We define the **covariance** of two random variables as the expected product of deviations from expectation. Put more simply, covariance is a generalization of variance to *two* random variables: $\text{Cov}(X, X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \text{Var}(X)$.

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])]$$

We can treat the covariance as a measure of association. Remember the definition of correlation given when we first established SLR?

$$r(X, Y) = \mathbb{E}\left[\left(\frac{X - \mathbb{E}[X]}{\text{SD}(X)}\right)\left(\frac{Y - \mathbb{E}[Y]}{\text{SD}(Y)}\right)\right] = \frac{\text{Cov}(X, Y)}{\text{SD}(X)\text{SD}(Y)}$$

It turns out we've been quietly using covariance for some time now! If X and Y are independent, then $\text{Cov}(X, Y) = 0$ and $r(X, Y) = 0$. Note, however, that the converse is not always true: X and Y could have $\text{Cov}(X, Y) = r(X, Y) = 0$ but not be independent. This means that the variance of a sum of independent random variables is the sum of their variances:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) \qquad \text{if } X, Y \text{ independent}$$

17.2.3 Standard Deviation

Notice that the units of variance are the *square* of the units of X. For example, if the random variable X was measured in meters, its variance would be measured in meters^2. The **standard deviation** of a random variable converts things back to the correct scale by taking the square root of variance.

$$\text{SD}(X) = \sqrt{\text{Var}(X)}$$

To find the standard deviation of a linear transformation $aX + b$, take the square root of the variance:

$$\text{SD}(aX + b) = \sqrt{\text{Var}(aX + b)} = \sqrt{a^2\text{Var}(X)} = |a|\text{SD}(X)$$