

Data 100

Principles and Techniques of Data Science

Michael Pham

Sp24

CONTENTS

Contents	2
1 Introduction to Data Science	5
1.1 Lecture 1 – 01/16/24	5
1.1.1 Course Overview	5
Why Data Science Matters	5
What is Data Science?	6
1.1.2 Course Outline	6
Prerequisites	6
Topics (Tentative)	6
Course Components	7
Grading	7
1.1.3 The Data Science Lifecycle	8
1.2 Lecture 2 – 01/18/24	9
1.2.1 Tabular Data	9
1.2.2 DataFrames and Series	9
The Series Object	9
DataFrames	10
1.2.3 Data Extraction	12
.head and .tail	12
Label-based Extraction (.loc)	12
Integer-based Extraction (.iloc)	13
.loc versus .iloc	14
Context-dependent Extraction ([])	14
2 Pandas, and more Pandas!	15
2.1 Lecture 3 – 01/23/24	15

2.1.1	Conditional Selection	15
2.1.2	Adding, Removing, and Modifying Columns	16
	Adding and Modifying Columns	16
	Removing Columns	17
2.1.3	Useful Utility Functions	17
	Custom Sorting	18
2.2	Lecture 4 – 01/25/24	18
2.2.1	Grouping	18
	Group Basics	19
	Aggregation Functions	19
	groupby.size() versus groupby.count()	22
	Filtering by Groups	23
	More on Groups	24
2.2.2	Pivot Tables	24
2.2.3	Join Tables	26
3	Data Wrangling and EDA	28
3.1	Lecture 5 – 01/30/24	28
3.1.1	Structure	28
3.1.2	Granularity	29
3.1.3	More on Structure	29
	Primary and Foreign Keys	29
4	Models, Regularization, and Probability	30
4.1	Lecture 16 – 03/12/24	30
4.1.1	Cross-Validation	30
4.1.2	K-Fold Cross-Validation	32
	Validation Folds	33
4.1.3	Hyperparameters	33
	Hyperparameter Tuning	33
4.1.4	Regularization	34
	Constraining Model Parameters	34
	L1 Regularization (LASSO)	34
	L2 Regularization (Ridge Regression)	35
4.2	Lecture 17 – 03/14/24	35
4.2.1	Probability Review	35
4.2.2	Random Variables	36
	Bernoulli Random Variable	37
4.2.3	Expectation and Variance	38

4.2.4	Sums of Random Variables	39
	Properties of Expectation	39
	Properties of Variance	40
	Equal, Independently Distributed, and i.i.d.	40
5	SQL	42
5.1	Lecture 20 – 04/02/2024	42
5.1.1	DBMS versus Raw Files	42
5.1.2	What is SQL?	43
5.1.3	Database Systems	43
5.1.4	Running SQL in Python	43
	Data Types	44
	Constraints	44
	Basic Queries	45

WEEK 1

INTRODUCTION TO DATA SCIENCE

*The purpose of computing is insight,
not numbers.*

— R. Hamming

1.1 Lecture 1 – 01/16/24

The course website is located at: <https://ds100.org/sp24/>.

1.1.1 Course Overview

Why Data Science Matters

Data is used everywhere, from science to sports to medicine. Claims using data also comes up often within discussions (especially about important issues).

Furthermore, Data Science enhances critical thinking. The world is complicated, and decisions are hard. This field fundamentally facilitates decision-making by quantitatively balancing trade-offs.

In order to quantify things reliably, we have to:

- Find relevant data;
- Recognize the limitations of said data;
- Ask the right questions;
- Make reasonable assumptions;
- Conduct appropriate analysis; and
- Synthesize and explain our insights.

At each step of this process, we must apply critical thinking and consider how our decisions can affect others.

What is Data Science?

Definition 1.1 (Data Science). Data Science is the application of data-centric, computational, and inferential thinking to:

- Understand the world (science), and
- Solve problems (engineering).

We note that good data analysis is **not**:

- Simple applications of a statistics recipe.
- Simple application of software.

There are many tools out there for data science, but they are ultimately just tools; we are the ones doing the important thinking.

1.1.2 Course Outline

Prerequisites

The official prerequisites are:

- **DATA 8**;
- **CS 61A**, DATA C88C, or ENGIN 7; and
- EE 16A, **MATH 54**, or STAT 89A.

The bolded course names are the ones that I have already taken.

Topics (Tentative)

The tentative list of topics that will be covered in this course is:

Tentative Topics

- | | |
|------------------------------------|--|
| • Pandas and NumPy | • Model design and loss formulation |
| • Relational Databases and SQL | • Linear Regression |
| • Exploratory Data Analysis | • Feature Engineering |
| • Regular Expressions | • Regularization, Bias-Variance Tradeoff, and Cross-Validation |
| • Visualization | • Gradient Descent |
| – matplotlib | • Data Science in the Physical World |
| – Seaborn | • Logistic Regression |
| – plotly | • Clustering |
| • Sampling | • PCA |
| • Probability and random variables | |

Course Components

With respect to lectures and assignments, the course is structured as follows:

Course Components				
Mo	Tu	We	Th	Fr
	Live Lecture		Live Lecture	
	Discussion	Discussion		
	Office Hours	Office Hours	Office Hours	Office Hours
			Homework N-1 due	Homework N released
	Lab N-1 due			Lab N released

For lectures, note that there attendance is mandatory; participation will be graded on a 0/1 basis:

- Synchronous Participation: complete at least one participation poll question during the live lecture timeslot (11:00am-12:30pm, Tuesdays and Thursdays). As long as you submit a response to at least one poll question in this timeframe, you will receive synchronous attendance credit.
- Asynchronous Participation: complete all participation poll questions from the link provided on the course website within one week of the corresponding lecture.
- In both cases, participation is graded on completion, not correctness.

Also, if we submit all participation polls over the semester, there will be a 0.5% bonus points applied to the final overall grade.

Grading

The grading scheme for this class is as follows:

Grading Scheme	
Category	
Homeworks	25%
Projects	10%
Labs	5%
Discussions	-
Lecture Participation	5%
Midterm Exam	22.5%
Final Exam	32.5%

Important:



- Midterm: Thursday, March 7, 7-9 PM PST.
- Final: Thursday, May 9, 8-11 AM PST.

1.1.3 The Data Science Lifecycle

The data science lifecycle is a high-level description of the data science workflow. Note in the diagram below that there are two distinct entry points.

The Data Science Lifecycle goes as follows:

1. Question/Problem Formulation

- What do we want to know?
- What problems are we trying to solve?
- What hypotheses do we want to test?
- What are our metrics for success?

2. Data Acquisition and Cleaning

- What data do we have and what data do we need?
- How will we sample more data?
- Is our data representative of the population we want to study?

3. Exploratory Data Analysis and Visualization

- How is our data organized, and what does it contain?
- Do we already have the relevant data?
- What are the biases, anomalies, or other issues with the data?
- How do we transform the data to enable effective analysis?

4. Prediction and Inference

- What does the data say about the world?
- Does it answer our questions or accurately solve the problem?
- How robust are our conclusions and can we trust the predictions?

1.2 Lecture 2 – 01/18/24

1.2.1 Tabular Data

Tabular data simply refers to data that is in a table, where each row represents one observation and each column representing some characteristic, or feature, of the observation.

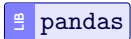
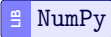
In this class, we will be using the Python  library.

Figure 1.1: pandas logo.



With Pandas, we can accomplish a lot of things. Below are a list of some things we will be learning to use Pandas for in this course:

- Arranging data in a tabular format.
- Extract useful information filtered by specific conditions.
- Operate on data to gain new insights.
- Apply  functions to our data.
- Perform vectorized computations to speed up our analysis.

1.2.2 DataFrames and Series

Definition 1.2 (DataFrames). In the language of pandas, we refer to a table as a DataFrame, which is a collection of named columns called Series.

Definition 1.3 (Series). A Series is a one-dimensional array-like object. It contains:

- A sequence of values of the same type.
- A sequence of data labels, called the index.

The Series Object

First, we look at some basic details about the Series object.

Code: Series Basics

```
1 import pandas as pd
2
3 # Creating a Series object
4 s = pd.Series(["Welcome", "to", "data 100"])
5 s.index # RangeIndex(start=0, stop=3, step=1)
6 s.values # array(['welcome', 'to', 'data 100'], dtype='object')
```

We see that the Series object has index and values variables. Note that we can in fact set custom indices, as shown below:

Code: Custom Indexing

```
1 # Creating Series with custom indexing
2 s = pd.Series([-1, 10, 2], index = ["a", "b", "c"])
3 s.index # Index(['a', 'b', 'c'], dtype='object')
4
5 # Changing Series' index
6 s.index = ["first", "second", "third"]
7 s.index # Index(['first', 'second', 'third'], dtype='object')
```

Finally, we can also select a single value or a set of values in a Series object using:

- A single label.
- A list of labels.
- A filtering condition.

Code: Selection

```
1 # Creating a Series
2 s = pd.Series([4, -2, 0, 6], index = ["a", "b", "c", "d"])
3
4 # Selection
5 s["a"] # Single Label: [4]
6 s[["a", "c"]] # List of Labels: [4, 0]
7 s[s > 0] # Filtering Condition: [4, 6]
```

Note that for selection via filtering condition, we do the following:

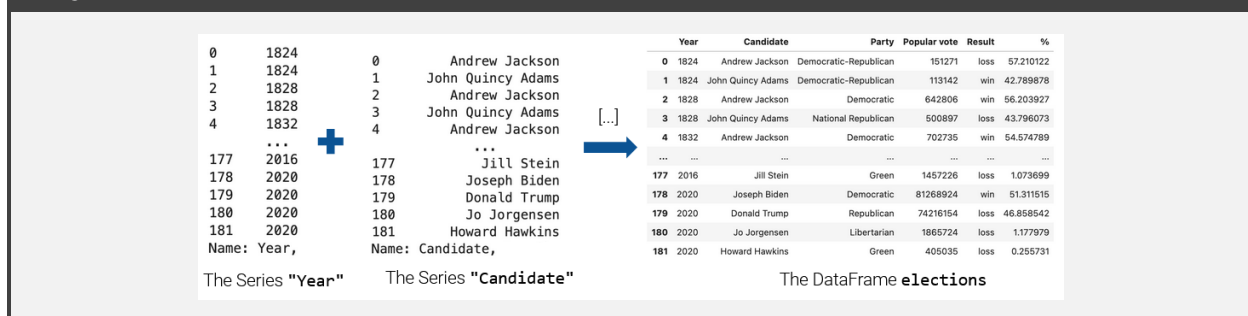
1. Apply a boolean condition to the Series that satisfy a particular condition.
2. Index into our Series using this boolean condition. pandas will select only the entries in the Series that satisfy the condition.

DataFrames

Typically, we imagine Series as columns within a DataFrame object.

One way to think of a DataFrame is a collection of Series object which shares the same index.

Figure 1.2: Relation between DataFrame and Series



The syntax for creating a DataFrame is `pandas.DataFrame(data, index, columns)`.

We have many approaches for creating a DataFrame. Below are some of the most common ones:

- From a CSV file.
- Using a list and column name(s).
- From a dictionary.
- From a Series.

Code: Creating DataFrame Objects

```

1 # From a CSV File
2 elections = pd.read_csv("data/elections.csv")
3
4 # We can also create a DataFrame with one of the columns as the index value
5 elections = pd.read_csv("data/elections.csv", index_col="Year")
6
7 # From a list and column name(s)
8 pd.DataFrame([1, 2, 3], columns=["Numbers"]) # One column
9 pd.DataFrame([[1, "one"], [2, "two"]], columns = ["Number", "Description"]) # Multiple columns
10
11 # From a dictionary
12 pd.DataFrame({"Fruit": ["Strawberry", "Orange"], "Price": [5.49, 3.99]}) # Specify columns of the
   ↪ DataFrame
13 pd.DataFrame([{"Fruit": "Strawberry", "Price": 5.49}, {"Fruit": "Orange", "Price": 3.99}]) # Specify the
   ↪ rows of the DataFrame
14
15 # From a Series
16 s_a = pd.Series(["a1", "a2", "a3"], index = ["r1", "r2", "r3"])
17 s_b = pd.Series(["b1", "b2", "b3"], index = ["r1", "r2", "r3"])
18
19 pd.DataFrame({"A-column": s_a, "B-column": s_b})

```

! Note that indices are not necessarily row numbers! We can set non-numeric values to be the index as well. Furthermore, they aren't necessarily unique either; we can have duplicate values in the index.

For example, suppose we ran the following code:

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

Then, we will get the following DataFrame:

Figure 1.3: DataFrame with non-numeric, non-unique indexing.

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

After creating a DataFrame, we can then change its index using `df.set_index(column_name)`. And if we happen to change our mind, we can simply do `df.reset_index()`.

However, while indexes are not necessarily unique, column names in pandas are almost always unique. Having two columns share a name is also just bad practice in general.

We can also extract basic, useful information about a DataFrame, such as its index, columns, and shape.

Code: Getting Basic DataFrame Information

```

1 # Getting the row labels
2 elections.index
3
4 # Getting the column labels
5 elections.columns
6
7 # Getting the shape of the DataFrame, given in the form (row, col)
8 elections.shape

```

1.2.3 Data Extraction

When dealing with data, we want to be able to extract information that we want to work with specifically. Common ways we may want our data is as follows:

- Grab the first or last *n* rows in the DataFrame.
- Grab data with a certain label.
- Grab data at a certain position.

.head and .tail

The simplest scenarios: We want to extract the first or last *n* rows from the DataFrame.

- `df.head(n)` will return the first *n* rows of the DataFrame `df`.
- `df.tail(n)` will return the last *n* rows.

Label-based Extraction (.loc)

Something more complex is extracting data with specific columns or index labels. One way to do this is with `.loc`, which has the following format:

```
df.loc[row_labels, column_labels]
```

The `.loc` accessor lets us specify the labels of rows/columns we want to extract. These are the bolded values in a DataFrame.

Arguments to `.loc` can be:

- A list.
- A slice.
- A single value.



Note that for `.loc`, the slice is, unlike normal Python syntax, **inclusive** of the right side.

Code: Extraction with .loc

```

1 # Using a list
2 elections.loc[[87, 25, 179], ["Year", "Candidate", "Result"]]

```

```

3
4      # Using a slice
5      elections.loc[[87, 25, 179], "Popular vote": "%"]
6      elections.loc[:, ["Year", "Candidate", "Result"]] # Gets all rows
7      elections.loc[[87, 25, 179], :] # Gets all columns
8
9      # Using a single value.
10     elections.loc[[87, 25, 179], "Popular vote"] # Note that this returns the Popular vote series,
11     ↪ with only the select indices
12     elections.loc[0, "Candidate"] # This returns the string value at index 0 from the Candidate
13     ↪ column

```

Figure 1.4: .loc using a list.

Select the rows with labels 87, 25, and 179.

	Year	Candidate	Result
87	1932	Herbert Hoover	loss
25	1860	John C. Breckinridge	loss
179	2020	Donald Trump	loss

Select the columns with labels "Year", "Candidate", and "Result".

Figure 1.5: .loc using a slice.

Select the rows with labels 87, 25, and 179.

	Popular vote	Result	%
87	15761254	loss	39.830594
25	848019	loss	18.138998
179	74216154	loss	46.858542

Select all columns starting from "Popular vote" until "%".

Integer-based Extraction (.iloc)

A different scenario: We want to extract data according to its position.

To do this, we use `.iloc`, which allows us to specify the integers of rows and columns we wish to extract. It has the following format:

```
df.iloc[row_integers, column_integers]
```

Arguments to `.iloc` can be:

- A list.
- A slice.
- A single value.

! Note that while the slice was **inclusive** of the right side for `.loc`, it is **exclusive** for `.iloc`...! Very confusing!!

The syntax for `.iloc` is identical to `.loc`.

.loc versus .iloc

While `.loc` is both safer (i.e. if the order of the data gets shuffled around, our code will still work) and more readable (people actually know what columns we want), `.iloc` can still be useful!

Example 1.4 (Using `.iloc`). If we have a `DataFrame` of movie earnings sorted by earnings, we can use `.iloc` to get the median earnings for a given year by indexing into the middle.

Context-dependent Extraction (`[]`)

Finally, we have the possibly most-complicated way of extracting data: context-dependent extraction.

`[]` only takes one argument, which may be:

- A slice of row integers.
- A list of column labels.
- A single column label.

Code: Extraction with `[]`

```
1 # Using a slice of row integers
2 elections[3:7]
3
4 # Using a list of column labels
5 elections[["Year", "Candidate", "Result"]]
6
7 # Using a single column label
8 elections["Candidate"]
```

Using `[]` is a lot more concise than either `.loc` or `.iloc`, hence it's often used more in practice than the other extraction methods.

WEEK 2

PANDAS, AND MORE PANDAS!

*You're as soft as Po,
the Kung Fu Panda!*

— Z. Sherwin

2.1 Lecture 3 – 01/23/24

2.1.1 Conditional Selection

We can extract rows that satisfy a given condition. Note that `.loc` and `[]` also accept boolean arrays as input; using this, only rows that correspond to `True` will be extracted.

For example, consider the following code:

Code: Conditional Selection

```
1 # Getting first ten rows
2 babynames_first_10_rows = babynames.loc[:9, :]
3
4 # Selection with []
5 babynames_first_10_rows[[True, False, True, False, True, False, True, False, True, False]]
6
7 # Selection with .loc
8 babynames_first_10_rows.loc[[True, False, True, False, True, False, True, False, True, False], :]
```

In both cases, the following DataFrame will be returned:

Figure 2.1: DataFrame with filtered rows.

	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

Using this, we can then extend it to more powerful conditional selections; for example, we can select only

rows that are female, or were born before 2000 in the DataFrame (or both!):

```
babynames[(babynames["Sex"] == "F") | (babynames["Year"] < 2000)]
```

This example above also shows us how we can use bitwise operators with our conditionals.

Bitwise Operators

If p and q are boolean arrays or Series, then:

Symbol	Usage	Meaning
\sim	$\sim p$	Negation of p
$ $	$p q$	p or q
$\&$	$p \& q$	p and q
\wedge	$p \wedge q$	p xor q

While boolean array selection is useful, it can make our code overly verbose for more complicated conditions. Luckily, pandas offers many alternatives:

- `.isin`
- `.str.startswith`
- `.groupby.filter`

Code: Using `.isin`

```
1 names = ["Bella", "Alex", "Narges", "Lisa"]
2
3 # .isin
4 babynames[babynames["Name"].isin(names)] # Returns a Series whose entry is True if the corresponding
   ↳ name in babynames is found in the names list
5
6 # .str.startswith
7 babynames[babynames["Name"].str.startswith("N")] # Returns a Series whose entry is True if the
   ↳ corresponding name starts with N
```

2.1.2 Adding, Removing, and Modifying Columns

Adding and Modifying Columns

To add a column, we proceed as follows:

1. Use `[]` to reference the desired new column.
2. Assign this column to a Series or array of the appropriate length.

We can also modify a column's name or values. The latter's steps are similar to adding a column. For the former, we can use the `.rename()` function which takes in a dictionary mapping old column names to the new one.

Below, we show how to add, then modify a column:

Code: Adding and Modifying Columns

```

1 # Create a Series of the length of each name
2 babyname_lengths = babynames["Name"].str.len()
3
4 # Add a column named "name_lengths" that includes the length of each name
5 babynames["name_lengths"] = babyname_lengths
6
7 # Modify the "name_lengths" column to be one less than its original value
8 babynames["name_lengths"] = babynames["name_lengths"]-1
9
10 # Rename "name_lengths" to "Length"
11 babynames = babynames.rename(columns={"name_lengths": "Length"})

```

Removing Columns

Removing columns a bit more tricky. First, if we want to drop a column, we can use `.drop()`.

! Note that `.drop()` assumes that we're dropping a row by default; to drop columns, we have to use `axis = "columns"`...! Furthermore, we **must** reassign to the updated DataFrame after dropping a row/column. By default, pandas methods create a copy of the DataFrame, without changing the original DataFrame at all. To apply our changes, we must update our DataFrame to this new, modified copy.

Code: Removing Columns

```

1 # Does NOT modify babynames DataFrame
2 babynames.drop("Length", axis="columns")
3
4 # Does modify babynames (through reassignment)
5 babynames = babynames.drop("Length", axis="columns")

```

2.1.3 Useful Utility Functions

As stated previously, one of the libraries which we will use is NumPy, which has a lot of useful operations such as `np.mean()` and `np.max()`.

And with the pandas library, we have access to a wider array of useful functions. Some which will be covered in this lecture are:

- `.shape`: returns the shape of a DataFrame or Series in the form (rows, cols).
- `.size`: returns the total number of entries in a DataFrame or Series.
- `.describe()`: returns a "description" of a DataFrame or Series that lists summary statistics of the data.
 - For DataFrame: count, mean, std, min/max, quartiles.
 - For Series: count, unique, top, freq
- `.sample()`: samples a random selection of rows from a DataFrame.
 - By default, it is without replacement. Use `replace=True` for replacement.
 - Naturally, can be chained with other methods and operators.
- `.value_counts()`: counts the number of occurrences of each unique value in a Series.

- `.unique()` : returns an array of every unique value in a Series.
- `.sort_values()` : sort a DataFrame (or Series).
 - `Series.sort_values()` will automatically sort all values in the Series.
 - `DataFrame.sort_values(column_name)` must specify the name of the column to be used for sorting.

! Note that for `.sort_values()`, rows are sorted in ascending order.
 We can do `.sort_values(ascending=False)` to sort in descending order.

Custom Sorting

Suppose that we wanted to sort using some custom sorting system. To accomplish this task, there are three main ways of doing so:

1. Creating a temporary column and sort Based on the new column.
2. Sorting using the key argument.
3. Sorting using the map function.

Code: Custom Sorting

```

1 # Create a Series of the length of each name
2 babynames_lengths = babynames["Name"].str.len()
3
4 # Approach 1: Create a temporary column
5 babynames["name_lengths"] = babynames_lengths # Add a column named "name_lengths" that includes the
6   ↪ length of each name
7 babynames = babynames.sort_values(by="name_lengths", ascending=False)
8
9 # Approach 2: Sorting using the key Argument
10 babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False)
11
12 # Approach 3: Sorting using the map Function
13 def dr_ea_count(string):
14     return string.count('dr') + string.count('ea')
15
16 babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count) # Use map to apply dr_ea_count to each
17   ↪ name in the "Name" column
18 babynames = babynames.sort_values(by="dr_ea_count", ascending=False)

```

2.2 Lecture 4 – 01/25/24

2.2.1 Grouping

Our goal:

- Group together rows that fall under the same category.
 - For example, group together all rows from the same year.
- Perform an operation that aggregates across all rows in the category.

- For example, sum up the total number of babies born in that year.

Grouping is a powerful tool as we can perform large operations all at once, and summarize trends within our dataset.

Group Basics

To do this, we can use the `.groupby()` operation, which involves splitting a DataFrame up, applying a function, and then combining the result.

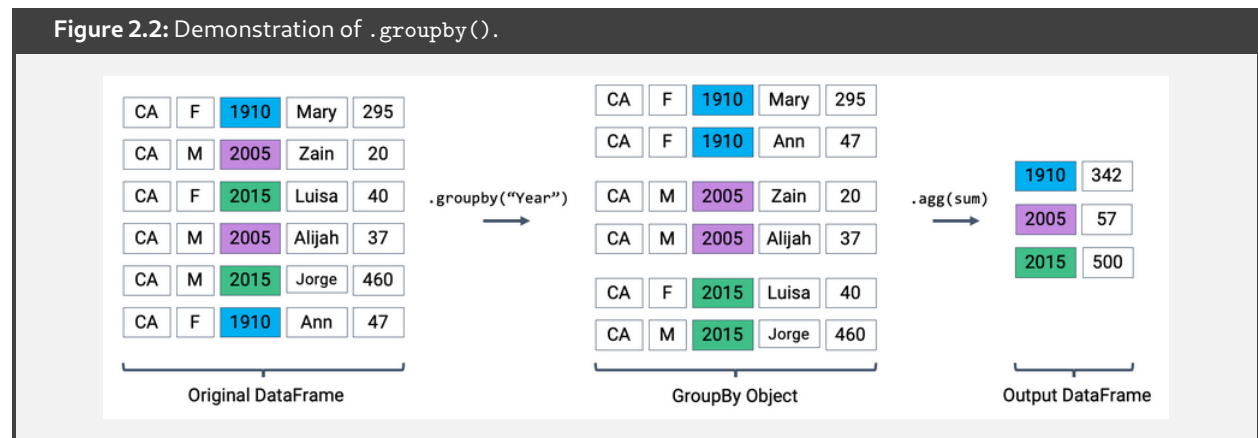
When we call `.groupby()` on a DataFrame, it generates DataFrameGroupBy objects; these are like sub-frames which contains all rows corresponding to the same group.

When using `.groupby()`, note that we can't directly work with the DataFrameGroupBy objects; instead, we use some aggregation function with `.agg()` to transform them back into a DataFrame.

The general syntax for the `.groupby()` function is:

```
dataframe.groupby(column_name).agg(aggregation_function).
```

For example, suppose we ran `babynames[["Year", "Count"]].groupby("Year").agg(sum)`, which returns the total number of babies born each year. The result is shown in the figure below:



Aggregation Functions

A number of things can go inside of the `.agg()` function:

Python Functions:

- `.agg(sum)`
- `.agg(max)`
- `.agg(min)`

NumPy Functions:

- `.agg(np.sum)`
- `.agg(np.max)`
- `.agg(np.min)`
- `.agg(np.mean)`

pandas Functions:

- `.agg("sum")`
- `.agg("max")`
- `.agg("min")`
- `.agg("mean")`
- `.agg("first")`
- `.agg("last")`

Now, with this in mind, we show some alternatives to finding the total number of babies born each year:

Code: Alternatives with .groupby()

```
1 # Approach 1: Using .agg(sum)
2 babynames[["Year", "Count"]].groupby("Year").agg(sum)
3
4 # Approach 2: Using .sum()
5 babynames.groupby("Year")[["Count"]].sum()
6
7 # Approach 3: Using .sum(numeric_only=True)
8 babynames.groupby("Year").sum(numeric_only=True)
```

Case Study: Name Popularity

Suppose that we wanted to find the female baby name which has fallen in popularity the most in California.

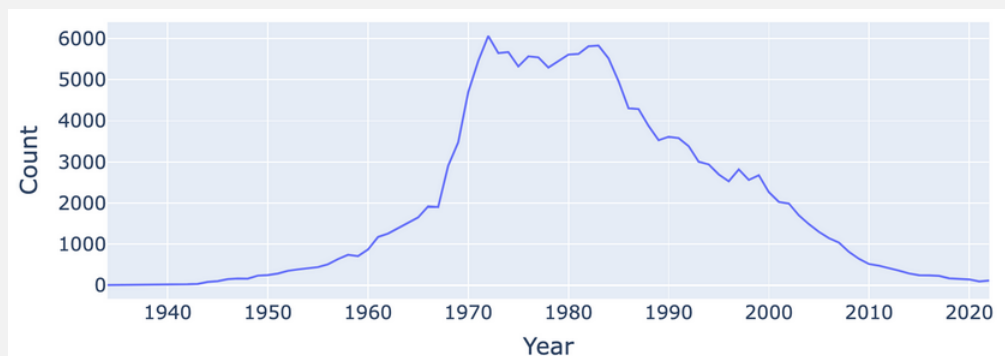
For example, the code below gives us the number of Jennifer's that are born in California per year.

Code: Jennifer's Popularity

```
1 f_babynames = babynames[babynames["Sex"] == "F"]
2 f_babynames = f_babynames.sort_values(["Year"])
3 jenn_counts_series = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"]
```

A line plot of the data is shown below:

Figure 2.3: Line plot of Jennifer's born in California per year.



Well, looking at the count, it certainly looks like the name Jennifer has decreased in popularity!

But, how do we actually *define* “fallen in popularity”...?

- To do this, let's create a metric: “Ratio to Peak” (RTP). This is the ratio of babies born with a given name in 2022 to the maximum number of babies born with that name in any year.

Going back to the Jennifer example, in 1972, we hit peak Jennifer. 6,065 Jennifers were born. In 2022, there were only 114 Jennifer's. So, calculating the RTP, we get:

$$114/6065 = 0.018796372629843364$$

So, we can write a function `ratio_to_peak()`, then use grouping to calculate the RTP of each name:

Code: Calculating the RTP

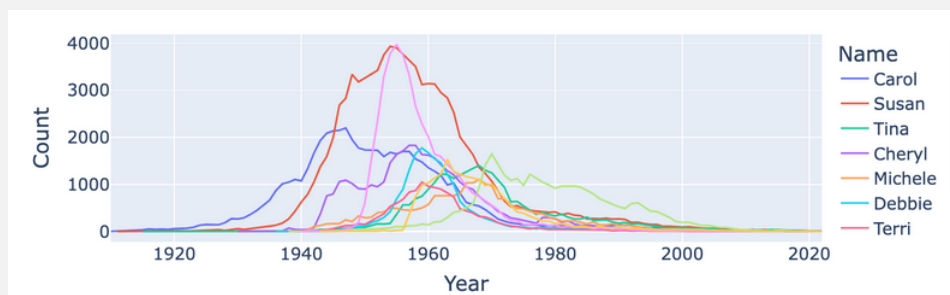
```

1 def ratio_to_peak(series):
2     return series.iloc[-1] / max(series)
3
4 # This WILL error!
5 f_babynames.groupby("Name").agg(ratio_to_peak) # Applies ratio_to_peak() onto all columns, even
↳ Str-type ones!!
6
7 # Calculates the RTP. Returns a table with a Year and Count column
8 rtp_table = f_babynames.groupby("Name")[["Year", "Count"]].agg(ratio_to_peak)
9
10 # Calculates the RTP. Returns a table with a Count column
11 rtp_table = f_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
12 rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"}) # Renames Count column to Count RTP
13
14 # Finds the top ten lowest RTP names
15 top10 = rtp_table.sort_values("Count RTP").head(10).index
16 # Plots the data
17 px.line(f_babynames[f_babynames["Name"].isin(top10)], x = "Year", y = "Count", color = "Name")

```

And after running the code above, we see that “Debra” has fallen in popularity the most out of all the names in California! For a visual representation of the ten names with the lowest RTP, we refer to the following graph:

Figure 2.4: Line graph of names which has fallen in popularity the most in California.



All in all, remember that the result of a groupby operation applied to a DataFrame is a DataFrameGroupBy object.

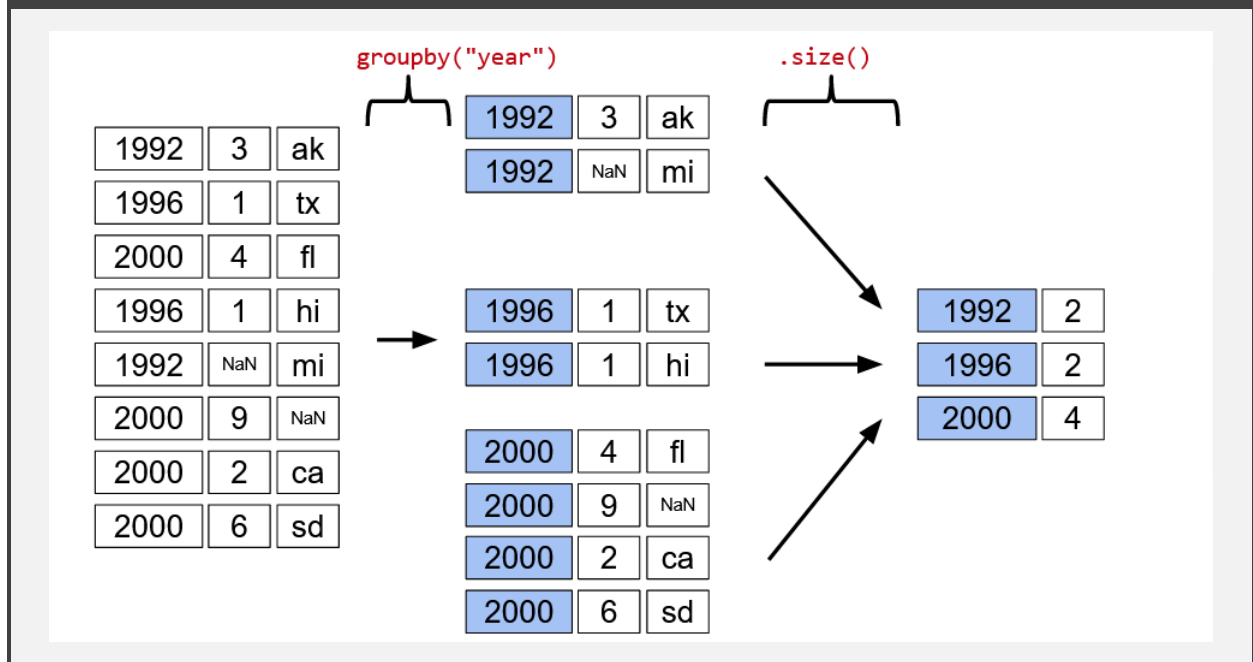
And once we’re given a DataFrameGroupBy object, we can use various functions to generate DataFrames (or Series). Note that `.agg()` is only one choice! Some other options are:

- `df.groupby(col).mean()`
- `df.groupby(col).first()`
- `df.groupby(col).filter()`
- `df.groupby(col).sum()`
- `df.groupby(col).last()`
- `df.groupby(col).min()`
- `df.groupby(col).size()`
- `df.groupby(col).max()`
- `df.groupby(col).count()`

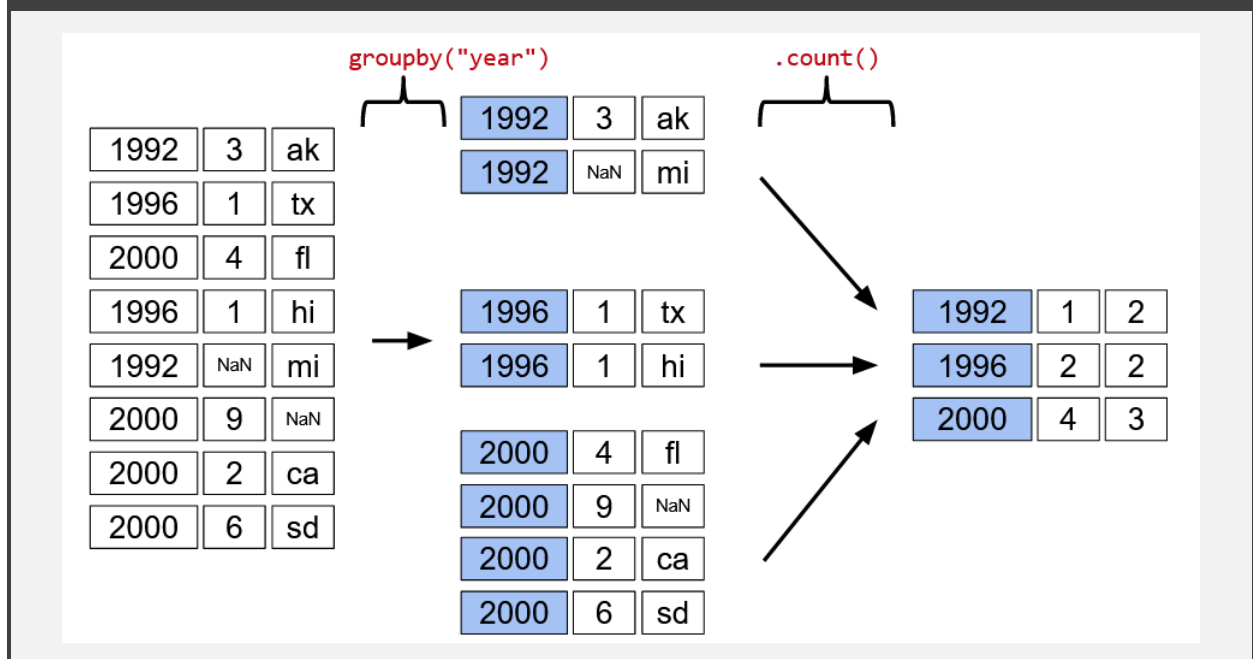
See <https://pandas.pydata.org/docs/reference/groupby.html> for a list of DataFrameGroupBy methods that we can use!

groupby.size() versus groupby.count()

One of the more subtle differences between some functions is that of `groupby().size()` and `groupby().count()`. `.size()` returns a Series object counting the number of rows in each group. This is similar to `.value_counts()`, though it does not sort the index based on the frequency of entries.

Figure 2.5: `groupby.size()` demo.

On the other hand, with `.count()`, it returns a DataFrame with the counts of non-missing values in each column.

Figure 2.6: `groupby.count()` demo.

Filtering by Groups

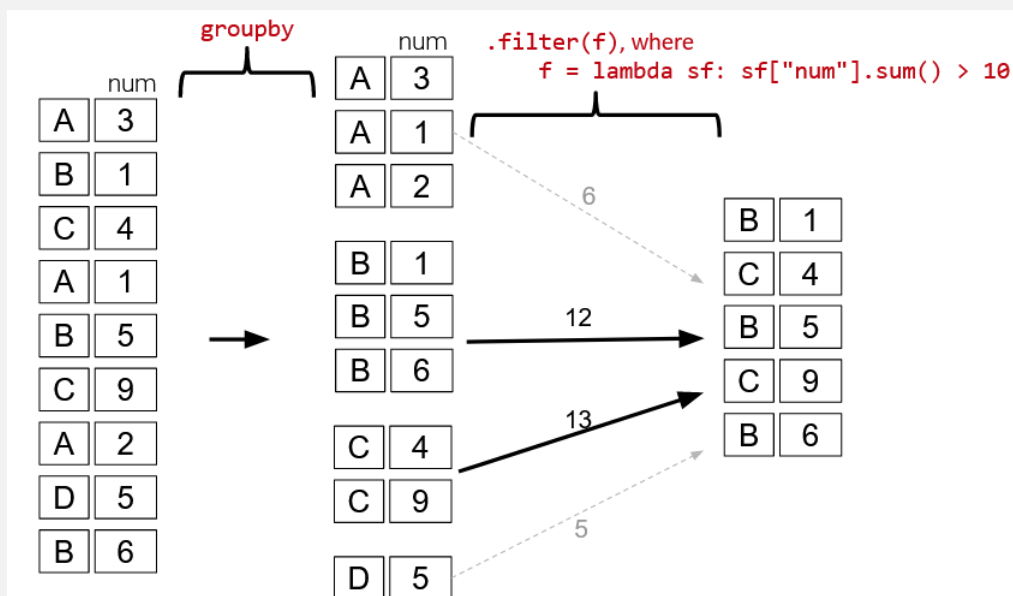
Another common use for groups is to filter data.

- `groupby.filter()` takes an argument `func`.
- `func` is a function that:
 - Takes a DataFrame as input.
 - Returns either `True` or `False`.
- `filter` applies `func` to each group/sub-DataFrame:
 - If `func` returns `True` for a group, then all rows belonging to the group are preserved.
 - If `func` returns `False` for a group, then all rows belonging to that group are filtered out.

Some caveats to filtering:

- Filtering is done per group, not per row. This is different from boolean filtering!
- Unlike `agg()`, the column we grouped on does NOT become the index!

Figure 2.7: Filtering by groups demo.



Case Study: Best Election

Suppose that we want to know the best election by each party, where the “best” election refers to the election with the highest percentage of votes.

- For example, Democrat’s best election was in 1964, with candidate Lyndon Johnson winning 61.3% of votes.

So... how do we do this?

A naive first attempt might be to do `elections.groupby("Party").max().head(10)`. However, we observe the following:

Figure 2.8: Error with naive attempt.

	Year	Candidate	Popular vote	Result	%
Party					
American	1976	Thomas J. Anderson	873053	lose	21.534001
American Independent	1976	Leiter Madison	990718	lose	13.271218
Anti-Monopoly	1832	William Hunt	100715	lose	7.521153
Anti-Monopoly	1884	Benjamin Butler	154204	lose	7.133355
Citizen	1980	Berry Commoner	233052	lose	0.270182
Communist	1932	William Z. Foster	103307	lose	0.581069
Constitutional Union	2016	Michael Pencucka	203091	lose	0.152388
Constitutional Union	1992	John Bell	590901	lose	12.892825
Democratic	2020	Woodrow Wilson	8128824	win	61.344703
Democratic-Republican	1824	John Quincy Adams	781275	win	57.819733

We see that we have that Woodrow Wilson won the presidency in 2020...!

Recall that each column is calculated independently! More concretely, the year's max is 2020, the name's max is Woodrow Wilson (due to alphabetical ordering), etc.

So, what do we do? Instead, we can do the following:

- First sort the DataFrame so that rows are in descending order of %.
- Then group by Party and take the first item of each sub-DataFrame.

The code is presented below, along with some alternate methods to solve the problem:

Code: Finding best election by each party.

```

1 # Attempt #2
2 elections_sorted_by_percent = elections.sort_values("%", ascending=False)
3 elections_sorted_by_percent.groupby("Party").first()
4
5 # Using lambda functions
6 elections_sorted_by_percent = elections.sort_values("%", ascending=False)
7 elections_sorted_by_percent.groupby("Party").agg(lambda x : x.iloc[0])
8
9 # Using idxmax
10 best_per_party = elections.loc[elections.groupby("Party")["%"].idxmax()]
11
12 # Using drop_duplicates function
13 best_per_party2 = elections.sort_values("%").drop_duplicates(["Party"], keep="last")

```

An important takeaway from this case study is that oftentimes, there are numerous approaches for a problem, with each one having its own benefit and drawbacks.

More on Groups

Finally, we note that we can look into DataFrameGroupBy objects with the following commands:

- `grouped_by_party.groups`
- `grouped_by_party.get_group("Socialist")`

2.2.2 Pivot Tables

Suppose we want to build a table showing the total number of babies born of each sex in each year. One way is to use `.groupby()` with both columns of interest as follows:

```
babynames.groupby(["Year", "Sex"])["Count"].agg(sum)
```


Figure 2.9: Using `.groupby()` on multiple columns.

		Count
Year	Sex	
1910	F	5950
	M	3213
1911	F	6602
	M	3381
1912	F	9804
	M	8142

! Note that the resulting DataFrame is multi-indexed in this case! That is, its index has multiple dimensions. This will be explored in a later lecture.

However, perhaps a more natural way of displaying this information is through the use of pivot tables.

Code: Basic pivot table.

```

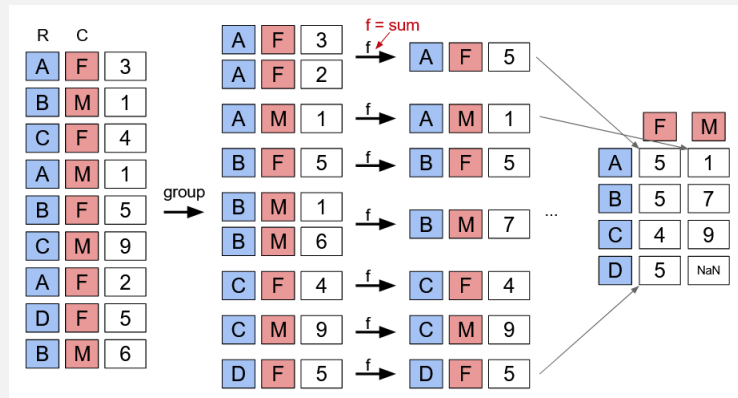
1 babynames_pivot = babynames.pivot_table(
2   index = "Year",      # rows (turned into index)
3   columns = "Sex",     # column values
4   values = ["Count"],  # field(s) to process in each group
5   aggfunc = np.sum,    # group operation
6 )

```

Figure 2.10: Basic pivot table.

Sex	F	M
Year		
1910	5950	3213
1911	6602	3381
1912	9804	8142
1913	11860	10234
1914	13815	13111
1915	18643	17192

Figure 2.11: Pivot table mechanics.



Note that we can also use pivot tables on multiple columns, as demonstrated below:

Code: Pivot table with multiple columns.

```
1 babynames_pivot = babynames.pivot_table(
2   index = "Year",      # rows (turned into index)
3   columns = "Sex",     # column values
4   values = ["Count", "Name"],
5   aggfunc = np.max,   # group operation
6 )
```

Figure 2.12: Pivot table with multiple columns.

Sex	Count		Name	
	F	M	F	M
Year				
1910	295	237	Yvonne	William
1911	390	214	Zelma	Willis
1912	534	501	Yvonne	Woodrow
1913	584	614	Zelma	Yoshio
1914	773	769	Zelma	Yoshio
1915	998	1033	Zita	Yukio

2.2.3 Join Tables

For the last topic of this lecture, we discuss joining two different tables together.

Example 2.1. Suppose that we wanted to know the popularity of a presidential candidate's name in 2022. Using `.merge()`, we can merge the `babynames` and `elections` tables together.

Below, we provide the code for this:

Code: Joining `babynames` and `elections` tables.

```
1 # Getting babynames information in 2022.
2 babynames_2022 = babynames[babynames["Year"] == 2022]
3
```

```

4 # Creating First Name column for elections table.
5 elections["First Name"] = elections["Candidate"].str.split().str[0]
6
7 # Method 1
8 merged1 = pd.merge(left = elections, right = babynames_2022,
9 left_on = "First Name", right_on = "Name")
10
11 # Method 2
12 merged = elections.merge(right = babynames_2022,
13 left_on = "First Name", right_on = "Name")

```

Figure 2.13: Merging DataFrame together.

	Year_x	Candidate	Party	Popular vote	Result	%	First Name	State	Sex	Year_y	Name	Count
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	CA	M	2022	Andrew	741
1	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	CA	M	2022	Andrew	741
2	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew	CA	M	2022	Andrew	741
3	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878	John	CA	M	2022	John	490
4	1828	John Quincy Adams	National Republican	500897	loss	43.796073	John	CA	M	2022	John	490
...

WEEK 3

DATA WRANGLING AND EDA

*Exploratory data analysis is an attitude,
a state of flexibility,
a willingness to look for those things that
we believe are not there,
as well as those that we believe to be there.*

— J. Tukey

3.1 Lecture 5 – 01/30/24

In this week's lecture, we will be looking at Exploratory Data Analysis (EDA).

Key Properties in EDA

Below are the key properties which we consider in EDA:

- **Structure** – the “shape” of a data file.
- **Granularity** – how fine/coarse is each datum.
- **Scope** – how (in)complete is the data.
- **Temporality** – how is the data situated in time.
- **Faithfulness** – how well does the data capture “reality.”

3.1.1 Structure

Oftentimes, we prefer working with rectangular data. The reasons for this is that

1. Regular structures are easy manipulate and analyze
2. A big part of data cleaning is about transforming data to be more rectangular

Two types of rectangular data are Tables and Matrices.

Tables:

- Named columns with different types.
- Manipulated using data transformation languages (map, filter, group by, join, ...).

Matrices:

- Numeric data of the same type (float, int, etc.).
- Manipulated using linear algebra.

Remark 3.1. We note that Tables are referred to as `DataFrames` in R/Python, and as relations in SQL.

3.1.2 Granularity

Definition 3.2 (Granularity). The granularity refers to how coarse/fine each datum is; i.e. what each record represents.

When the granularity is fine, we see information on specific individuals/objects/etc., whereas coarse granularity is where we see information of some group.

We note that not all records may capture granularity at the same level; for example, sometimes data may include summaries ("rollups") as records.

Furthermore, if the data is coarse, we should consider how it was aggregated (sampling, averaging, maybe some combination of the two...?).

3.1.3 More on Structure**Primary and Foreign Keys**

idk

WEEK 4

MODELS, REGULARIZATION, AND PROBABILITY

Woohooooooooooooo!!!

— S. Rao

4.1 Lecture 16 – 03/12/24

Recall from the end of Lecture 14 that we had two lingering thoughts:

- The idea of “unseen” data – this is data which our model didn’t encounter during data.
- The idea of “model complexity” – this influences if it underfits or overfits.

Our goal for this lecture is to find this spot between underfitting and overfitting.

4.1.1 Cross-Validation

A complex model may not perform well on data which it hasn’t encountered during training. The figure below demonstrates this:

We can quantify the performance of the model on unseen data via **test sets**.

Definition 4.1 (Test Set). A **test set** is a portion of our data set that is set aside for testing purposes.

We don’t consider the test set when fitting/training the model.

Furthermore, the test set is only ever touched *once* to compute the performance (MSE/RMSE/etc.) of the model after all the fine-tuning is completed.

Our workflow for modeling is as follows: First, we perform a test-train split. We consider only the training set when designing the model. Then, we evaluate the test set.

Figure 4.1: New workflow for modeling.



Below, we demonstrate how we can actually perform a train-test split:

Code: Performing a train-test split.

```

1 from sklearn.model_selection import train_test_split
2
3 # We can specify the size of the testing data. The size shouldn't be too small nor too large.
4
5 # We can shuffle the data set before using it; this is to prevent there being any biases. The default is
  ↪ set to True.
6
7 # random_state sets the seed so every time we run the function, we get the same split of the data.
8
9 train_test_split(X, Y, test_size = 0.2, shuffle = True, random_state = 100)
  
```

And from this, we can then

Right now, we have two segments in our data set: training and testing.

But we need a third segment, a “validation set.”

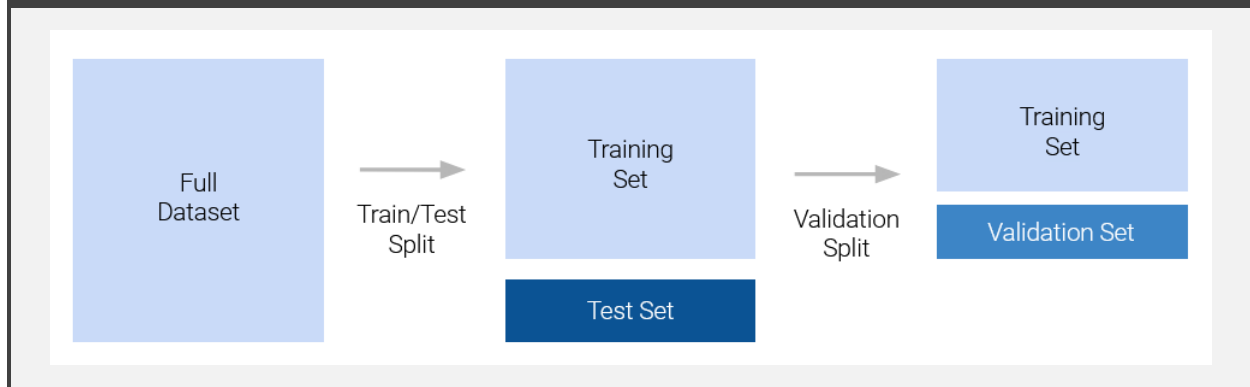
Consider the following: suppose we’ve ran our model, but then we are dissatisfied with the test set performance. In our current workflow, we’d be stuck; we can’t just go back to our model and adjust it, since we’d be using information from the test set to improve the model. Then, this test set would no longer be unseen data.

To remedy this, we can introduce a third segment: a “validation set.”

Definition 4.2 (Validation Set). A **validation set** is a portion of our training set that we set aside for assessing model performance while it is still being developed.

We train the model on the training set, and assess performance on the validation set. We then adjust the data and repeat.

After *all* model development is complete, we assess final performance on the test set.

Figure 4.2: Workflow for modeling with validation.**Code:** Validation sets.

```

1 # Split X_train further into X_train_mini and X_val.
2
3 X_train_mini, X_val, Y_train_mini, Y_val = train_test_split(X_train, Y_train, test_size=0.2,
4   ↪ random_state=100)

```

Furthermore, we can fit different models of varying complexity, compute their errors, and find the best complexity for our model:

Code: Finding the optimal model complexity.

```

1 from sklearn.pipeline import Pipeline
2 from sklearn.preprocessing import PolynomialFeatures
3
4 def fit_model_dataset(degree):
5     pipelined_model = Pipeline([
6         ('polynomial_transformation', PolynomialFeatures(degree)),
7         ('linear_regression', lm.LinearRegression())
8     ])
9
10    pipelined_model.fit(X_train_mini[["hp"]], Y_train_mini)
11    return mean_squared_error(Y_val, pipelined_model.predict(X_val[["hp"]]))
12
13    errors = [fit_model_dataset(degree) for degree in range(0, 18)]
14    MSEs_and_k = pd.DataFrame({"k": range(0, 18), "MSE": errors})
15
16    plt.figure(dpi=120)
17    plt.plot(range(0, 18), errors)
18    plt.xlabel("Model Complexity (degree of polynomial)")
19    plt.ylabel("Validation MSE")
20    plt.xticks(range(0, 18));

```

Below, we see that as complexity increases, the error decreases... until a certain point. But then, the validation error starts increasing after a certain point; our model is overfitting!

4.1.2 K-Fold Cross-Validation

Introducing a validation set gives us one extra chance to assess our model performance.

Specifically, we now understand how the model performs on *one* particular set of unseen data.

However, it's possible that we may have, by random chance, selected a set of validation points that was *not* representative of other unseen data that the model might encounter.

Ideally, we want to assess our model performance on several different validation sets before touching on

our test set.

Validation Folds

In our original validation split, we set aside $x\%$ of the training data to be used for validation.

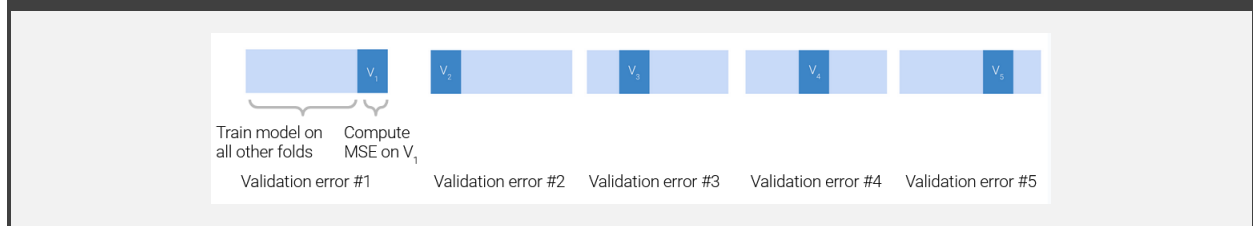
However, we could have selected any $x\%$ of the training data for validation – we could've used any of these portions once!

The algorithm for K-Fold Cross-Validation is as follows:

- We split our data into K folds.
- Pick one fold to be the validation fold.
- Train the model on data from every other fold.
- Compute the model's error on the validation fold and record it.
- Repeat this for each of the K folds.

Then, we define the **cross-validation error** to be the average error across all K validation folds.

Figure 4.3: K-Fold Cross-Validation Demo



4.1.3 Hyperparameters

Definition 4.3 (Hyperparameter). A hyperparameter are values in a model chosen before the model is fit to data.

Example 4.4. Examples of hyperparameters include:

- Degree of polynomial model.
- The value for K in K-Fold Cross-Validation.
- Gradient descent learning rate α .
- Regularization penalty λ .

Hyperparameter Tuning

To select a hyperparameter value via cross-validation:

- List out several different guesses for the best hyperparameter.
- For each guess, run cross-validation to compute the CV error for the choice of hyperparameter value.
- Select the hyperparameter value with the lowest CV error.

4.1.4 Regularization

Constraining Model Parameters

One of the questions we have to tackle is how we can control the complexity of our model.

One idea is to use each feature a little in our model.

$$\mathbb{Y} = \theta_0 + \theta_1\phi_1 + \theta_2\phi_2 \cdots + \theta_p\phi_p$$

If we restrict how large each parameter θ_i can be, we restrict how much each feature contributes to the model. When θ_i is close or equal to 0, the model decreases in complexity because ϕ_i barely impacts the prediction.

In regularization, we restrict complexity by putting a limit on the magnitude of the θ_i

Example 4.5. Suppose we specify that the sum of all absolute parameters can be no larger than some number Q :

$$\sum_{i=1}^p |\theta_i| \leq Q.$$

We've given the model a "budget" on how much weight to assign to each feature. Some features θ_i would need to be smaller in order for the sum to be less than Q .

Note here that the intercept θ_0 is typically excluded from our calculations.

We note that smaller values of Q will lead to less complex models, whereas larger values of Q will lead to more complex models.

L1 Regularization (LASSO)

Definition 4.6 (L1 Regularization). In L1 Regularization, we find thetas which minimize the objective function:

$$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \frac{1}{n} \sum_{i=1}^n (y_i - (\theta_0 + \theta_1\phi_{i,1} + \cdots + \theta_p\phi_{i,p}))^2,$$

where $\sum_{i=1}^p |\theta_i| \leq Q$.

Then, by the Lagrangian Duality, this is equivalent to finding thetas which minimize the augmented objective function:

$$\frac{1}{n} \sum_{i=1}^n (y_i - (\theta_0 + \theta_1\phi_{i,1} + \cdots + \theta_p\phi_{i,p}))^2 + \lambda \sum_{i=1}^p |\theta_i|$$

Definition 4.7 (Regularization Penalty Hyperparameter). λ is the regularization penalty hyperparameter. When λ is large, our objective function is penalized more for choosing larger thetas. So, model will adjust by reducing thetas and decreasing complexity.

We choose our λ via cross-validation.

Remark 4.8. L1 Regularization is also called LASSO: "Least Absolute Shrinkage and Selection Operator".

Code: L1 Regularization

```

1 import sklearn.linear_model as lm
2 lasso_model = lm.Lasso(alpha = 1) # alpha represents the hyperparameter lambda
3 lasso_model.fit(X_train, Y_train)
4 lasso_model.coef_

```

Note that the optimal parameters for a LASSO model tend to include a lot of zeroes.

In other words, LASSO effectively selects only a subset of the features.

However, there is an issue with this approach: features with larger values will naturally contribute more to the predicted \hat{y} for each observation. This means then that our LASSO model will need to spend more of its budget to allow smaller values to have much of an impact on each observation.

To remedy this, we can standardize the data; i.e. we replace everything with z-scores:

$$z_k = \frac{x_k - \mu_k}{\sigma_k}$$

L2 Regularization (Ridge Regression)

Alternatively, for our Q , we could have done:

$$\sum_{i=1}^p \theta_i^2 \leq Q.$$

Definition 4.9 (L2 Regularization). For L2 Regularization, we want to find thetas which minimize the following objective function:

$$\frac{1}{n} \sum_{i=1}^n (y_i - (\theta_0 + \theta_1 \phi_{i,1} + \dots + \theta_p \phi_{i,p}))^2 + \lambda \sum_{i=1}^p \theta_i^2$$

Code: L2 Regularization.

```

1 import sklearn.linear_model as lm
2 ridge_model = lm.Ridge(alpha = 1) # alpha represents the hyperparameter lambda
3 ridge_model.fit(X_train, Y_train)
4 ridge_model.coef_
5

```

One thing to note about Ridge Regression is that it has a closed-form solution:

$$\hat{\theta}_{\text{ridge}} = (\mathbb{X}^\top \mathbb{X} + n\lambda I)^{-1} \mathbb{X}^\top \mathbb{Y}.$$

! We note that there is a solution even if \mathbb{X} is not full-rank – this is an important reason for why we often prefer L2 Regularization.

4.2 Lecture 17 – 03/14/24**4.2.1 Probability Review**

Recall the following:

Definition 4.10 (Sample Mean). The **sample mean** is the mean of our random sample. We can calculate this using `np.mean(data)`.

Furthermore, an important theorem is the following:

Theorem 4.11 (Central Limit Theorem). If you draw a large random sample with replacement, then, regardless of the population distribution, the probability distribution of the sample mean:

- Is roughly normal.
- Is centered around the population mean.
- Has a standard deviation of:

$$s = \frac{\sigma}{\sqrt{n}}$$

where σ is the population standard deviation, and n is the sample size.

4.2.2 Random Variables

Definition 4.12 (Random Variable). A **random variable** is a function from the outcome of a random event to a number.

Example 4.13. For example, a coin can land either heads (H) or tails (T). We can define a random variable X as follows:

$$X = \begin{cases} 1 & \text{if coin lands heads} \\ 0 & \text{if coin lands tails} \end{cases}$$

We note that $\text{dom } X = \{H, T\}$ and $\text{ran } X = \{1, 0\}$.

Remark 4.14. Note that Random Variables are typically denoted with uppercase letters, whereas Regular Variables are typically denoted with lowercase letters!

For any random variable, we need to be able to specify two things:

- **Possible values:** the set of values the random variable can take
- **Probabilities:** the chance that the random variable will take each possible value.
 - Each probability should be a real-number between 0 and 1 (inclusive).
 - The total probability of all possible values should be 1.

Assuming that X is discrete, then it has a finite number of possible values. Then, we denote $P(X = x)$ to be the probability that X takes on x , and we have:

$$\sum_x P(X = x) = 1.$$

We often do this using a **probability distribution table**. Going back to our coin toss example, we have:

x	$P(X = x)$
0	1/2
1	1/2

Definition 4.15 (Distribution). A **distribution** fully defines a random variable. If you know the distribution of a random variable you can:

- Can compute properties of the random variables and derived variables.
- Can simulate the random variables.

We can use `np.random.choice` or `df.sample` or `scipy.stats.<dist>.rvs(...)`

Remark 4.16. We often don't know the (true) distribution and instead compute an empirical distribution:

Note that the total area of the bars/density curve will be equal to 1.

Bernoulli Random Variable

Definition 4.17 (Bernoulli Random Variable). A Bernoulli Random Variable is one that takes on two values: 0 or 1.

The Bernoulli distribution is parameterized by p , the probability that $P(X = 1)$.

Common Distributions

Below are some common distributions:

- Bernoulli (p)
 - $P(X = 1) = p$ and $P(X = 0) = 1 - p$.
 - Sometimes called the "indicator" random variable.
- Binomial (n, p)
 - Number of 1's in n independent Bernoulli trials.
- Categorical (p_1, \dots, p_k) of values
 - Probability of each value is $1 / (\text{number of possible values})$.
- Uniform on the open unit interval $(0, 1)$
 - Density is flat at 1 on $(0, 1)$ and 0 elsewhere.
- Normal (μ, σ^2)
 - Can be described with the following equation:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

4.2.3 Expectation and Variance

Definition 4.18 (Expectation). The **expectation** of a random variable X is the weighted average of the possible values of X , where the weights are the probabilities of the values.

The following are two equivalent ways to apply the weights:

- One sample at a time:

$$E[X] = \sum_{\text{all samples } s} X(s)P(s)$$

- One possible value at a time:

$$E[X] = \sum_{\text{all possible } x} xP(X = x)$$

We note that the second way is more common, as we typically are given the distribution, not all possible samples!

The expectation of a random variable is a number, *not* a random variable!

- It is a generalization of the average (same units as the random variable).
- It is the center of gravity of the probability distribution histogram.
- If we simulate the variable many times, it is the long run average of the simulated values.

Definition 4.19 (Variance). **Variance** is the expected squared deviation from the expectation of X .

We define $\text{Var}(X)$ to be as follows:

$$\text{Var}(X) = E[(X - E[X])^2]$$

Remark 4.20. The units of the variance are the square of the units of X . To get back to the original units, we use the standard deviation of X :

$$\text{SD}(X) = \sqrt{\text{Var}(X)}$$

Variance and Standard Deviation return number, *not* random variables!

- Describes the variability of a random variable.
- Variance is the expected squared error between the random variable and its expected value.
- Could be viewed as the chance error between a sample X and the population mean.

Remark 4.21. By Chebyshev's Inequality, no matter what the shape of the distribution of X is, the vast majority of the probability lies in the interval "expectation plus or minus a few SDs."

There's a more convenient form for variance however:

$$\text{Var}(X) = E[X^2] - (E[X])^2$$

To calculate $E[X^2]$, we can simply do:

$$E[X^2] = \sum_x x^2 P(X = x).$$

Example 4.22. Let X be the outcome of a single dice roll, where X is a random variable. Then, we have the following:

$$P(X = x) = \begin{cases} \frac{1}{6} & x \in \{1, 2, 3, 4, 5, 6\} \\ 0 & \text{otherwise} \end{cases}$$

Then, $E[X]$ will be equal to:

$$\begin{aligned} E[X] &= \frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) \\ &= \frac{7}{2} \end{aligned}$$

And $\text{Var}(X)$ will be:

$$\begin{aligned} \text{Var}(X) &= E[X^2] - (E[X])^2 \\ &= \frac{91}{6} - \left(\frac{7}{2}\right)^2 \\ &= \frac{35}{12} \end{aligned}$$

4.2.4 Sums of Random Variables

First, we note that a function of random variables will be a random variable as well!

Example 4.23. If X_1, \dots, X_n are random variables, then the following are random variables as well:

- $\max(X_1, \dots, X_n)$.
- $\frac{1}{n} \sum_{i=1}^n X_i$.
- X_n^2

Properties of Expectation

The following are important properties of expectations:

- Expectation is linear: $E[aX + b] = aE[X] + b$.
- Expectation is linear in sums of RVs: $E[X + Y] = E[X] + E[Y]$.
- If g is non-linear, then $E[g(X)] \neq g(E[X])$.

Below is the proof for the second property:

Proof. ... ■

Using these properties, we can in fact derive the alternate definition of variance as follows:

Proof. ... ■

Properties of Variance

The following are important properties of variance:

- Variance is non-linear: $\text{Var}(aX + b) = a^2 \text{Var}(X)$.
- Variance of the sums of RVs is affected by the independence of the RVs: $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y)$.

Definition 4.24 (Covariance). **Covariance** is the expected product of deviations from expectation:

$$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])].$$

Remark 4.25. We note that covariance is a generalization of variance. Furthermore,

$$\text{Cov}(X, X) = E[(X - E[X])^2] = \text{Var}(X)$$

We can interpret by finding the correlation:

$$r(X, Y) = E \left[\frac{X - E[X]}{\text{SD}(X)} \frac{Y - E[Y]}{\text{SD}(Y)} \right] = \frac{\text{Cov}(X, Y)}{\text{SD}(X) \text{SD}(Y)}.$$

Correlation (and therefore covariance) measures a linear relationship between X and Y .

- If X and Y are correlated, then knowing X tells you something about Y .
- “ X and Y are uncorrelated” is the same as “Correlation and covariance equal to 0”.
- Independent X, Y are uncorrelated, because knowing X tells you nothing about Y .
- The converse is not necessarily true: X, Y could be uncorrelated but not independent.

Equal, Independently Distributed, and i.i.d.

Definition 4.26 (Equal). X and Y are **equal** if:

- $X(s) = Y(s)$ for every sample s .
- We write $X = Y$.

Definition 4.27 (Identically Distributed). X and Y are **identically distributed** if:

- The distribution of X is the same as the distribution of Y .
- We say “ X and Y are equal in distribution.”
- If $X = Y$, then X and Y are identically distributed; but the converse is not true (e.g. $Y = 7 - X$, where X is a die).

Definition 4.28 (i.i.d.). X and Y are independent and identically distributed (i.i.d.) if:

- X and Y are identically distributed, and
- Knowing the outcome of X does not influence your belief of the outcome of Y , and vice versa (" X and Y are independent.")

5.1 Lecture 20 – 04/02/2024

Thus far in the class, we've worked with data which is stored in CSV files.

This is a reasonable workflow for data which is **moderately sized** and is **unchanging** (static).

- Size is < 10 GB (or a third of the RAM on your workstation).
- Weekly snapshot.

However, in the real world, we note that data is typically stored in a Database Management System.

So... what exactly *is* a database even?

Definition 5.1 (Database). A **database** is an organized collection of data.

Definition 5.2 (Database Management System). A **Database Management System** (DBMS) is a software system that stores, manages, and facilitates access to one or more databases.

Example 5.3. Common large-scale DBMS's used in Data Science are:

- Google BigQuery
- Amazon Redshift
- ...

5.1.1 DBMS versus Raw Files

Some of the advantages of a DBMS over raw files (such as CSV) are as follow:

- Data Storage:
 - Reliable storage to survive system crashes and disk failures.
 - Optimize to compute on data that does not fit in memory.

- Special data structures to improve performance (you can learn more about this in CS186).
- Data Management:
 - Configure how data is logically organized and who has access.
 - Can enforce guarantees on the data (e.g. non-negative person weight or age):
 - * Can be used to prevent data anomalies.
 - * Ensures safe concurrent operations on data (i.e. multiple users reading/writing simultaneously.)

5.1.2 What is SQL?

SQL is its own programming language, distinct from Python. However, it is often called from within other programming languages.

SQL is a special purpose programming language which is used specifically for communicating with database systems.

- It is **the** dominant language/technology for working with data!
- It's is a **language of tables**; all inputs *and* outputs are tables.

Remark 5.4. One of the reasons why SQL is so widely-used is due to how old and ubiquitous it is; it's hard for companies to just switch to something else.

5.1.3 Database Systems

In this class, we'll be looking at two database systems: SQLite and DuckDB.

Definition 5.5 (SQLite). SQLite is an easy-to-use library that allows us to directly manipulate a database file or an in-memory database using a simplified version of SQL.

It's commonly used to store data for small apps on mobile devices, and it's optimized for simplicity and speed of simple data tasks.

Definition 5.6 (DuckDB). DuckDB is an easy-to-use library that lets you directly manipulate a database file, collection of table-formatted files (such as CSV), or in-memory pandas dataframes using a more complete version of SQL.

It's increasingly popular for data analysis on large datasets, and is optimized for simplicity and speed of advanced data analysis tasks.

5.1.4 Running SQL in Python

In order to connect to a database, we proceed as follows:

Code: Running SQL

```
1 %load_ext sql # Using ipython cell magic so we can use SQL
2
3 %%sql duckdb:///data/basic_examples.db
```

```

4
5  %%sql
6 SELECT * FROM Database

```

Definition 5.7 (Relation). SQL tables are often called **relations**.

It has columns (which are also referred to as attributes, or fields), and rows (or records/tuple).

Every column in a relation has three properties:

- ColName,
- Type, and
- Constraint(s) on values.

Definition 5.8 (Schema). A **schema** describes the logical structure of a table. Whenever a new table is created, the creator must declare its schema.

Code: Example of creating a table

```

1 CREATE TABLE Dragon (
2     name TEXT PRIMARY KEY,
3     year INTEGER CHECK (year >= 2000),
4     cute INTEGER
5 )

```

Data Types

Some types in SQL include:

- INT: integers.
- FLOAT: floating point numbers.
- VARCHAR: strings of text (also called TEXT).
- BLOB: arbitrary data (e.g. songs, video files, etc.).
- DATETIME: date and time.

Constraints

Some examples of constraints are:

- CHECK: data must obey the given check constant.
- PRIMARY KEY: specifies that this key is used to uniquely identify rows in the table.
- NOT NULL: null data can't be inserted for this column.
- DEFAULT: provides a default value to use if the user doesn't specify on insertion.

Definition 5.9 (Primary Key). A **primary key** is the set of column(s) used to uniquely identify each record in the table. They're used to ensure data integrity, and to optimize data access.

Definition 5.10 (Foreign Key). A **foreign key** is a reference to a primary key in another table.

Basic Queries

First, we can select columns using the `SELECT` query that we want to look at. If we want to rename a selected column, we can use the `AS` keyword.