CS410: Principles and Techniques of Data Science

Module 6: Wrangling Dataframes

https://drive.google.com/drive/folders/17SUxUV7w-Kx1_jvlh4mM7oK2ikriv-ga?usp=sharing

Introduction

- We often need to perform preparatory work on our data before we can begin our analysis.
- Depending on the data source, we often have different expectations for quality.
    - Some datasets require extensive wrangling to get them into an analyzable form, and other datasets arrive clean and we can quickly launch into modeling.

Introduction

Data from a scientific experiment - typically clean and well-documented.

Data from government surveys - often come with very detailed codebooks describing how the data are collected and formatted, and these datasets are typically ready for exploration.

Administrative data - can be clean, but often need to extensively check quality.

Informally collected data - data scraped from the Web; can be quite messy.

Eg: texts, tweets, blogs, etc. usually require formatting and cleaning to get them ready for analysis.

Quality Checks

We consider quality from four points:

- Scope: Do the data match your understanding of the population?
- Measurements and Values: Are the values reasonable?
- Relationships: Are related features in agreement?
- Analysis: Which features can be used?

Quality based on Scope

With San Francisco restaurant inspections, all restaurant zip codes should start with 941.

But, we can see that there are several zip codes that don't:

```
bus['postal_code'].value_counts().tail(10)
```

```
94080        1
94621        1
94544        1
            ..
64110        1
94545        1
941033148    1
Name: postal_code, Length: 10, dtype: int64
```

This lets us spot problems in the data.

Quality of Measurements and Recorded Values

Check the quality of measurements by considering what might be a reasonable value for a feature.

Based on common knowledge of ranges: a restaurant inspection score must be between 0 and 100;

Months run between 1 and 12.

We can use documentation to tells us the expected values for a feature. For example, the type of emergency room visit in the DAWN has been coded as 1, 2, …, 8

CASETYPE **TYPE OF VISIT**

Location: 1214-1214 (width: 1; decimal: 0)

Variable Type: numeric

Value	*Label*	*Unweighted Frequency*	*%*	*Valid %*
1	SUICICDE ATTEMPT:(1)	9033	3.9 %	3.9%
2	SEEKING DETOX:(2)	14841	6.5 %	6.5%
3	ALCOHOL ONLY (AGE < 21):(3)	7421	3.2 %	3.2%
4	ADVERSE REACTION:(4)	88096	38.4 %	38.4%
5	OVERMEDICATION:(5)	18146	7.9 %	7.9%
6	MALICIOUS POISONING:(6)	793	0.3 %	0.3%
7	ACCIDENTAL INGESTION:(7)	3253	1.4 %	1.4%
8	OTHER:(8)	87628	38.2 %	38.2%

Based upon 229211 valid cases out of 229211 total cases.

Ensure that the data type matches our expectations. Eg. we expect a price to be a number, whether or not it's stored as integer, floating point, or string.

Quality of Relationships

In addition to checking individual values in a column, we also want to cross-check values between features.

Eg: according to the documentation for the DAWN study, alcohol consumption is only considered a type of visit for patients under 21. Alcohol is coded as a 3, and ages under 21 are coded as 1, 2, 3, and 4.

```
display_df(pd.crosstab(dawn['age'], dawn['type']), rows=12)
```

type	1	2	3	4	5	6	7	8
age								
-8	2	2	0	21	5	1	1	36
1	0	6	20	6231	313	4	2101	69
2	8	2	15	1774	119	4	119	61
3	914	121	2433	2595	1183	48	76	4563
4	817	796	4953	3111	1021	95	44	6188
5	983	1650	0	4404	1399	170	48	9614
6	1068	1965	0	5697	1697	140	62	11408
7	957	1748	0	5262	1527	100	60	10296
8	1847	3411	0	10221	2845	113	115	18366
9	1616	3770	0	12404	3407	75	150	18381
10	616	1207	0	12291	2412	31	169	7109
11	205	163	0	24085	2218	12	308	1537

The cross tabulation shows that all of the alcohol cases are for patients under 21

The data values are as expected.

Quality for Analysis

Even when data pass your quality checks, problems can arise with its usefulness.

Type of restaurant inspection in San Francisco can be either routine or from a complaint. Since only one of the 14,000+ inspections was from a complaint, we lose little if we drop that single inspection.

```
pd.value_counts(insp['type'])
```

```
routine      14221
complaint        1
Name: type, dtype: int64
```

How to find bad values and features?

- Check summary statistics, distributions, and value counts.
- A table of counts of unique values in a feature can uncover unexpected rare occurrence.
- Computing the number of records that do not pass the quality check can quickly reveal the size of the problem.
- Examine the whole record for those records with problematic values. At times, an entire record is garbled when, for example, a comma is misplaced in a CSV formatted file.

What to do with your discoveries?

- Leave the data as is: Not every unusual aspect of the data needs to be fixed. You might find that the problem is relatively minor and most likely will not impact your analysis so you can leave the data as is.
- Modify values: You might want to replace corrupted values with `NaN`. You might have figured out what went wrong and correct the value.
- Remove features or drop records: If you find yourself modifying many values for a feature then you might consider eliminating the feature from the dataset.
- In general, we do not want to drop a large number of observations from a dataset without good reason.

Missing data

- Sometimes you want to replace corrupted data values with `NaN`, and hence treat them as missing
- At other times, data might arrive missing:
  - When someone refuses to participate in the study
- What to do with missing data is an important topic and there is a lot of research on this problem

Inputting Missing Values

Imputation: Substituting a reasonable value for missing ones to create a “clean” data frame.

Some common approaches for imputing values are **deductive**, **mean**, and **hot-deck** imputation.

Deductive imputation

Deductive imputation: fill in a value through logical relations.

Eg: below are rows in the business data frame for San Francisco restaurant inspections. Their zip codes are erroneously marked as “Ca” and latitude and longitude are missing. We can look up the address on the USPS Website to get the correct zip code and we can use Google Maps to find the latitude and longitude of the restaurant to fill in these missing values.

```
bus[bus['postal_code'] == "Ca"]
```

	business_id	name	address	city	...	postal_code	latitude	longitude	pl
5480	88139	TACOLICIOUS	2250 CHESTNUT ST	San Francisco	...	Ca	NaN	NaN	+

1 rows × 9 columns

Mean imputation

Mean imputation uses an average value from rows in the dataset that have values.

As a simple example, if a dataset on test scores has a missing value, mean imputation could fill in the missing value using the overall mean test score.

	Age	Gender	Fitness_Score
0	20	M	NaN
1	25	F	7.0
2	30	M	NaN
3	35	M	7.0
4	36	F	6.0
5	42	F	5.0
6	49	M	6.0
7	50	F	4.0
8	55	M	4.0
9	60	F	5.0
10	66	M	4.0
11	70	F	NaN
12	75	M	3.0
13	78	F	NaN

	Age	Gender	Fitness_Score
0	20	M	5.1
1	25	F	7.0
2	30	M	5.1
3	35	M	7.0
4	36	F	6.0
5	42	F	5.0
6	49	M	6.0
7	50	F	4.0
8	55	M	4.0
9	60	F	5.0
10	66	M	4.0
11	70	F	5.1
12	75	M	3.0
13	78	F	5.1

Hot-deck imputation

Uses a chance process to select a value at random from rows that have values.

As a simple example, hot-deck imputation could fill in missing test scores by randomly choosing another test score in the dataset.

A potential problem with hot-deck imputation is that the strength of a relationship might decline because we have added randomness to the values.

Transformations

- At times a feature is not in a form best suited for analysis and so we need to transform it.
- Many reasons a feature might need a transformation:
  - The value codings might not be useful for analysis,
  - We may want to combine two features into one using an arithmetic expression, or
  - We might want to pull information out of a feature to create a new feature.

We describe these three basic kinds of transformations: type conversions, mathematical transformations, and extractions.

Type conversion

This kind of transformation occurs when we convert the data from one format to another to make the data more useful for analysis.

We might convert information stored as a string to another format.

For example, we would want to convert prices reported as strings to numeric (like changing the string `"$2.17"` to the number 2.17) so that we can compute summary statistics on them.

Mathematical transformation

One kind of mathematical transformation is when we change the units of a measurement from, say, pounds to kilograms.

We might want to create new features from an arithmetic operations on others. For example, we can combine heights and weights to create body mass index by calculating weight / height^2

Extraction

Sometimes we will want to create a feature that contains partial information taken from another feature.

For example, the inspection violations consists of a string with a description of the violation, and we may be interested in only whether the violation is related to “vermin”. We can create a new feature that is `True` if the violation contains the word “vermin” in its text description and `False` otherwise.

Transforming Timestamps

A **timestamp** is a data value that records a specific date and time.

Timestamps come in many different formats. Example: a timestamp could be recorded like `Jan 1 2020 2pm` or `2021-01-31 14:00:00` or `2017 Mar 03 05:12:41.211 PDT`.

Timestamps are also very useful for analysis, since they let us answer questions like:
“What times of day do we have the most website traffic?”.

The `insp` dataframe records when restaurant inspections happened. We see that the `date` column looks like a timestamp.

`insp`

	business_id	score	date	type
0	19	94	20160513	routine
1	19	94	20171211	routine
2	24	98	20171101	routine
...	...	...	...	...
14219	94142	100	20171220	routine
14220	94189	96	20171130	routine
14221	94231	85	20171214	routine

14222 rows × 4 columns

Transforming Timestamps

By default, `pandas` reads in the `date` column as an integer. This storage type makes it hard to answer some useful questions about the data.

```
insp['date']
```

```
0        20160513
1        20171211
2        20171101
           ...
14219    20171220
14220    20171130
14221    20171214
Name: date, Length: 14222, dtype: int64
```

We want to know: do inspections happen more often on the weekends? Or the weekdays? To answer this question, we want to convert the `date` column to the `pandas` Timestamp storage type.

The `date` values appear to come in the format: `YYYYMMDD`, where `YYYY`, `MM`, and `DD` correspond to the year, month, and day.

The `pd.to_datetime()` method can parse the date strings into objects, and we can pass in the format of the dates as a date format string.

Transforming Timestamps

```
# This is the Python representation of the YYYYMMDD format
date_format = '%Y%m%d'
# Converts the date column to datetime objects
insp_dates = pd.to_datetime(insp['date'], format=date_format)
```

```
0       2016-05-13
1       2017-12-11
2       2017-11-01
           ...
14219   2017-12-20
14220   2017-11-30
14221   2017-12-14
Name: date, Length: 14222, dtype: datetime64[ns]
```

We can see that the `insp_dates` now has a `dtype` of `datetime64[ns]`, which means that the values were successfully converted into `pd.Timestamp` objects.

Transforming Timestamps

`datetime64[ns]` means that `pandas` uses 64 bits of memory for each value, and that each datetime is accurate to the nanosecond (or ns, for short).

```
0       2016-05-13
1       2017-12-11
2       2017-11-01
           ...
14219   2017-12-20
14220   2017-11-30
14221   2017-12-14
Name: date, Length: 14222, dtype: datetime64[ns]
```

```
insp_dates = pd.to_datetime(insp['date'], format=date_format)
```

The `pd.to_datetime()` method tries to automatically infer the timestamp format if we don't pass in the `format=` argument. In many cases `pandas` will parse the timestamps properly. However, sometimes the parsing doesn't output the correct timestamps, so we must explicitly specify the format.

Transforming Timestamps

`pandas` has special methods that hold timestamps using the `.dt` accessor.

We can easily pull out the year for each timestamp:

```
insp_dates.dt.year
```

```
0        2016
1        2017
2        2017
         ...
14219    2017
14220    2017
14221    2017
Name: date, Length: 14222, dtype: int64
```

Transforming Timestamps

The `pandas` has the complete details on the `.dt` accessor. By looking at the documentation, we see that the `.dt.day_of_week` attribute gets the day of week for each timestamp (Monday=0, Tuesday=1, …, Sunday=6).

```
insp_dates.dt.dayofweek
```

```
0        4
1        0
2        2
        ..
14219    2
14220    3
14221    3
Name: date, Length: 14222, dtype: int64
```

Transforming Timestamps

Assigning new columns to the `insp` dataframe containing both the parsed timestamps and the day of week for each timestamp.

```
insp = insp.assign(timestamp=insp_dates, dow=insp_dates.dt.dayofweek)
```

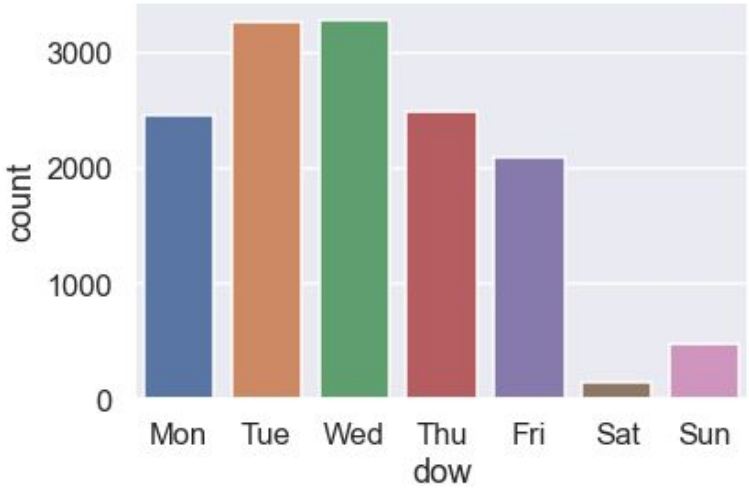
	business_id	score	date	type	timestamp	dow
0	19	94	20160513	routine	2016-05-13	4
1	19	94	20171211	routine	2017-12-11	0
2	24	98	20171101	routine	2017-11-01	2
...	...	...	...	...	...	...
14219	94142	100	20171220	routine	2017-12-20	2
14220	94189	96	20171130	routine	2017-11-30	3
14221	94231	85	20171214	routine	2017-12-14	3

14222 rows × 6 columns

Transforming Timestamps

Now, we can see whether restaurant inspectors favor a certain day of the week by grouping on the `dow` column.

```
sns.countplot(x='dow', data=insp)
# set xticklabels to be the day of the week
 plt.gca().set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']);
```



So, restaurant owners can generally expect inspections to happen on a weekday rather than a weekend.

Using pipe() for transformations

In real data analyses, we typically apply many transformations on the data.

However, it is easy to introduce bugs when we repeatedly mutate a dataframe because Colab notebooks let us run cells in any order we want.

As good practice, we recommend putting transformation code into functions with helpful names and using the `.pipe()` method to chain transformations together.

Using pipe() for transformations

```
date_format = '%Y%m%d'

def parse_dates_and_years(df, column='date'):
    dates = pd.to_datetime(df[column], format=date_format)
    years = dates.dt.year
    return df.assign(timestamp=dates, year=years)
```

We can pipe the `insp` dataframe through this function using `.pipe()`:

```
insp = (pd.read_csv("data/inspections.csv")
        .pipe(parse_dates_and_years))
```

	business_id	score	date	type	timestamp	year
0	19	94	20160513	routine	2016-05-13	2016
1	19	94	20171211	routine	2017-12-11	2017
2	24	98	20171101	routine	2017-11-01	2017
...	...	...	...	...	...	...
14219	94142	100	20171220	routine	2017-12-20	2017
14220	94189	96	20171130	routine	2017-11-30	2017
14221	94231	85	20171214	routine	2017-12-14	2017

14222 rows × 6 columns

Using pipe() for transformations

We can chain as many `.pipe()` calls as we want.

```
def extract_day_of_week(df, col='timestamp'):
    return df.assign(dow=df[col].dt.day_of_week)
insp = (pd.read_csv("data/inspections.csv")
        .pipe(parse_dates_and_years)
        .pipe(extract_day_of_week))
```

	business_id	score	date	type	timestamp	year	dow
0	19	94	20160513	routine	2016-05-13	2016	4
1	19	94	20171211	routine	2017-12-11	2017	0
2	24	98	20171101	routine	2017-11-01	2017	2
...	...	...	...	...	...	...	...
14219	94142	100	20171220	routine	2017-12-20	2017	2
14220	94189	96	20171130	routine	2017-11-30	2017	3
14221	94231	85	20171214	routine	2017-12-14	2017	3

14222 rows × 7 columns

Advantages of pipe()

- When there are many transformations on a single dataframe, it's easier to see what transformations happen since we can simply read the function names.
- We can reuse transformation functions for different dataframes. For instance, the `viol` dataframe, which contains restaurant safety violations, also has a `date` column.

	business_id	date	description
0	19	20171211	Inadequate food safety knowledge or lack of ce...
1	19	20171211	Unapproved or unmaintained equipment or utensils
2	19	20160513	Unapproved or unmaintained equipment or utensi...
...	...	...	...
39039	94231	20171214	High risk vermin infestation [date violation...
39040	94231	20171214	Moderate risk food holding temperature [dat...
39041	94231	20171214	Wiping cloths not clean or properly stored or ...

39042 rows × 3 columns

Example: Wrangling CO2 Measurements

Dataset: co2_mm_mlo.txt

https://gml.noaa.gov/ccgg/trends/data.html

Colab:

https://drive.google.com/file/d/1agAOTJhnLAOrXsT9hQOnpv6imrldQA6P/view?usp=sharing

Example: Wrangling CO2 Measurements

- Carbon dioxide (CO2) is an important signal of global warming. CO2 is a gas that traps heat in the earth's atmosphere. Without it, earth would be impossibly cold, but increases in CO2 also drives global warming and threatens our planet's climate.
- CO2 concentrations have been monitored at Mauna Loa Observatory since 1958, and these data offer a crucial benchmark for understanding the threat of global warming.

Example: Wrangling CO2 Measurements

Let's begin by checking the formatting, encoding, and size of the source before we load it into a data frame.

```
from pathlib import Path
import os
import chardet

co2_file = Path('data') / 'co2_mm_mlo.txt'
print(f'File size: {os.path.getsize(co2_file) / 1024:0.2f} KiB')

encoding = chardet.detect(co2_file.read_bytes())['encoding']
print(f'File encoding: {encoding}')
```

```
File size: 49.93 KiB
File encoding: ascii
```

Example: Wrangling CO2 Measurements

We found that the file is plain text with ASCII encoding and about 50 KiB.

Since the file is not particularly large, we shouldn't have any difficulty loading it into a data frame.

Let's look at the first few lines in the file.

```
lines = co2_file.read_text().split('\n')
```

```
len(lines)
```

811

```
['# ----------------------------------------------------------------------',
 '# USE OF NOAA ESRL DATA',
 '# ',
 '# These data are made freely available to the public and the',
 '# scientific community in the belief that their wide dissemination',
 '# will lead to greater understanding and new scientific insights.']
```

Example: Wrangling CO2 Measurements

We see that the file begins with information about the data source.

We should read this documentation before starting our analysis.

Let's find where the actual data values are located.

```
lines[69:79]
```

```
['#',
 '#            decimal     average   interpolated    trend    #days',
 '#              date                             (season corr)',
 '1958   3    1958.208      315.71      315.71      314.62     -1',
 '1958   4    1958.292      317.45      317.45      315.29     -1',
 '1958   5    1958.375      317.50      317.50      314.71     -1',
 '1958   6    1958.458      -99.99      317.10      314.85     -1',
 '1958   7    1958.542      315.86      315.86      314.98     -1',
 '1958   8    1958.625      314.93      314.93      315.94     -1',
 '1958   9    1958.708      313.20      313.20      315.91     -1']
```

Example: Wrangling CO2 Measurements

We have found the data begins on the 73rd line of the file. We also spot some relevant characteristics:

- The values are separated by white space, possibly tabs.
- The data line up down the rows. For example, the month appears in 7th to 8th position of each line.
- The 71st and 72nd lines in the file contain column headings split over two lines.

We can use `read_csv` to read the data into a Pandas dataframe, and we provide arguments to specify that the separators are white space, there is no header (we will set our own column names), and to skip the first 72 rows of the file.

```
co2 = pd.read_csv('data/co2_mm_mlo.txt', header=None, skiprows=72,
                  sep='\s+',
                  names=['Yr', 'Mo', 'DecDate', 'Avg', 'Int', 'Trend', 'days'])
```

	Yr	Mo	DecDate	Avg	Int	Trend	days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1
...	...	...	...	...	...	...	...
735	2019	6	2019.46	413.92	413.92	411.58	27
736	2019	7	2019.54	411.77	411.77	411.43	23
737	2019	8	2019.62	409.95	409.95	411.84	29

738 rows × 7 columns

Quality Checks

We can check a few rows of the dataframe to see if we can spot issues in the data.

```
co2.head()
```

	Yr	Mo	DecDate	Avg	Int	Trend	days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1
3	1958	6	1958.46	-99.99	317.10	314.85	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1

```
co2.tail()
```

	Yr	Mo	DecDate	Avg	Int	Trend	days
733	2019	4	2019.29	413.32	413.32	410.49	26
734	2019	5	2019.38	414.66	414.66	411.20	28
735	2019	6	2019.46	413.92	413.92	411.58	27
736	2019	7	2019.54	411.77	411.77	411.43	23
737	2019	8	2019.62	409.95	409.95	411.84	29

Some data have unusual values like -1 and -99.99. When we read the information at the top of the file, we find that -99.99 denotes a missing monthly average and -1 signifies a missing value for the number of days the equipment was in operation that month.

Quality Checks

First, we consider the shape of the data. How many rows should we have?

From looking at the head and tail of the data frame, the data appear to be in chronological order, beginning with March 1958 and ending with August 2019.

This means we should have $12 \times (2019 - 1957) - 2 - 4 = 738$ records.

```
co2.shape
```

```
(738, 7)
```

Our calculations match the number of rows.

Quality Checks

Let's check the quality of the feature, `Mo`. We expect the values to range from 1 to 12, and each month should have 2019-1957 = 62 or 61 instances (since the recordings begin in March of the first year and end in August of the most recent year).

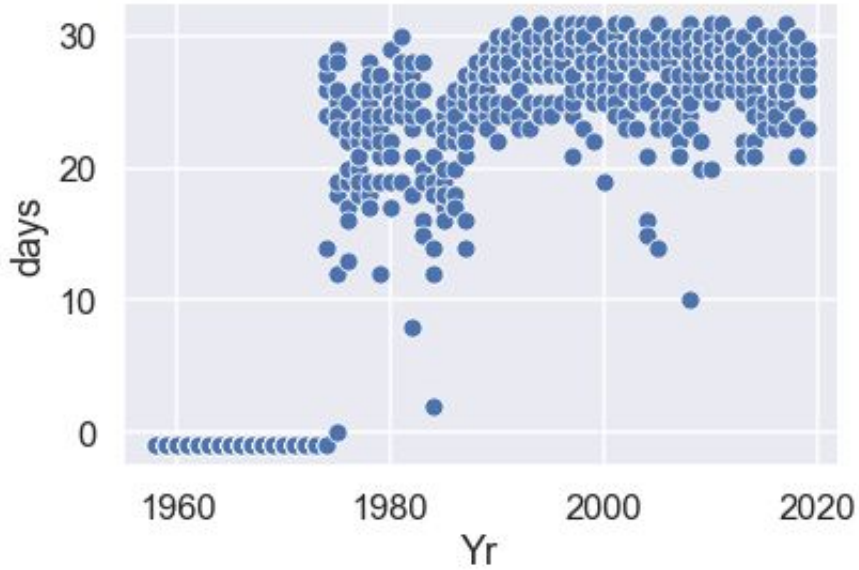
```
co2["Mo"].value_counts().reindex(range(1,13))
```

```
1     61
2     61
3     62
      ..
10    61
11    61
12    61
Name: Mo, Length: 12, dtype: int64
```

As expected Jan, Feb, Sep, Oct, Nov, and Dec have 61 occurrences and the rest 62.

Quality Checks

```
sns.scatterplot(x="Yr", y="days", data=co2)
```



All of the missing data are in the early years of operation.

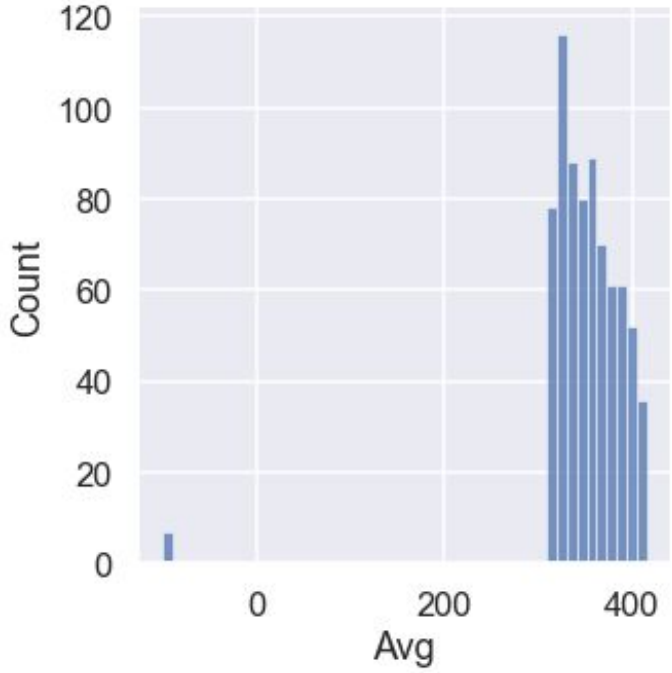
If we are concerned about the impact on the CO2 averages of the missing values for `days`, then a simple solution would be to drop the earliest recordings.

Quality Checks

Let's look at the -99.99 values in `Avg` and the overall quality of the CO2 measurements.

```
sns.displot(co2['Avg']);
```

The non-missing values are in the 300-400 range, which is as expected based on our external research on CO2 levels. We also see that there are only a few missing values.



Quality Checks

Since there aren't many missing values, we can examine all of them:

```
co2[co2["Avg"] < 0]
```

	Yr	Mo	DecDate	Avg	Int	Trend	days
3	1958	6	1958.46	-99.99	317.10	314.85	-1
7	1958	10	1958.79	-99.99	312.66	315.61	-1
71	1964	2	1964.12	-99.99	320.07	319.61	-1
72	1964	3	1964.21	-99.99	320.73	319.55	-1
73	1964	4	1964.29	-99.99	321.77	319.48	-1
213	1975	12	1975.96	-99.99	330.59	331.60	0
313	1984	4	1984.29	-99.99	346.84	344.27	2

What do we do with the -99.99s that we have discovered?

Do we drop those records?

Do we replace -99.99 with NaN?

Or substitute it with a likely value for the average CO2?

What do you think are the pros and cons of each possible action?

Addressing Missing Data

Let's examine each of these three options.

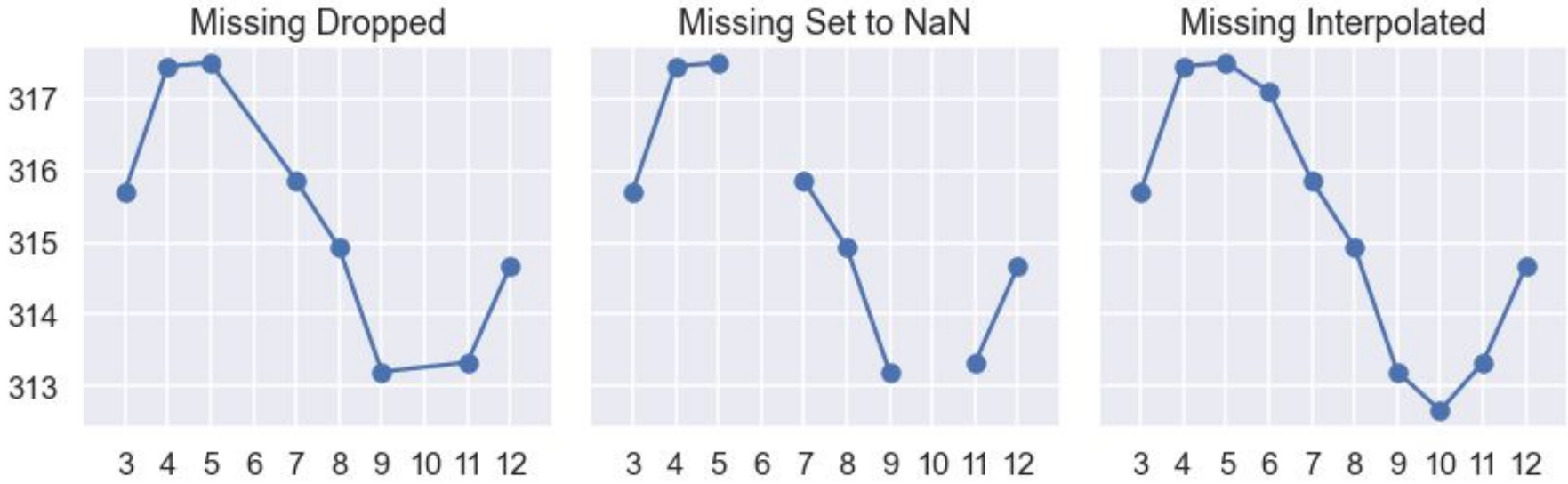
We make two versions of the data, one that drops records and the other that replaces -99.99 with NaN.

```
# Dropped missing values
co2_drop = co2[co2['Avg'] > 0]


# Replaced with NaN
 co2_NA = co2.replace(-99.99, np.NaN)
```

Addressing Missing Data

Time series plot for the three cases: drop the records with -99.99s (left plot); use NaN for missing value (middle plot); substitute an estimate for -99.99 (right plot).



Can you see what's happening in each of these plots?

Addressing Missing Data

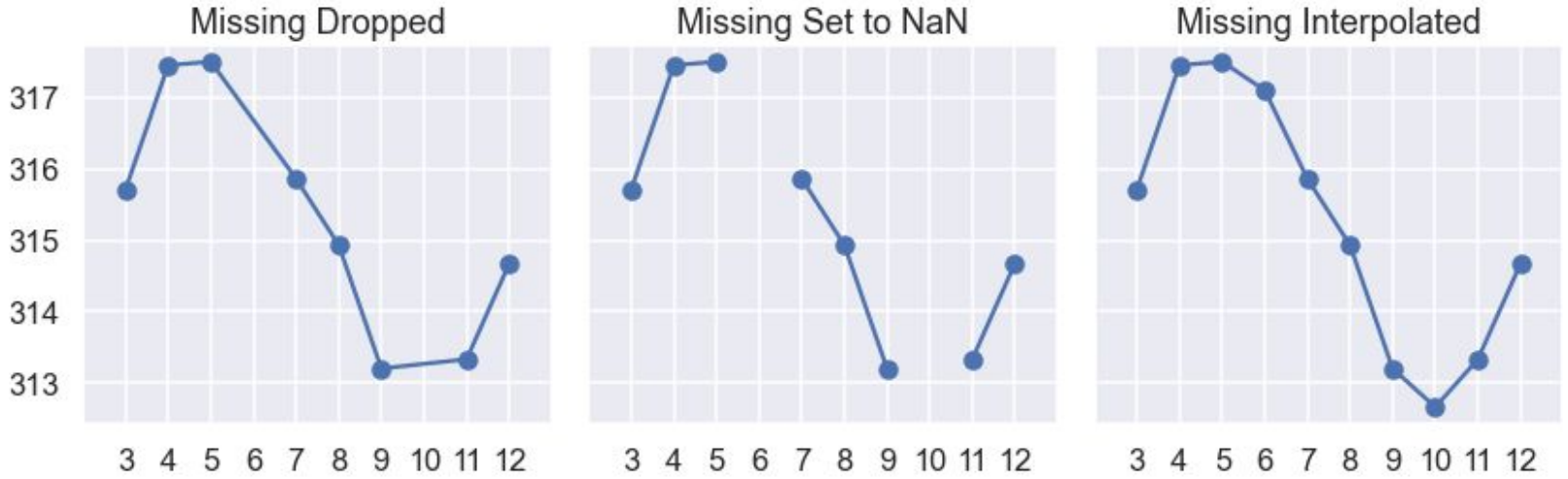
The leftmost plot connects dots over a two month time period, rather than one month.

In the middle, the line breaks where the data are missing.

On the right, we can see two monthly averages have been added.

In the big picture since there are only seven values missing from the 738 months, any of these options works.

However, there is some appeal to the right plot since the seasonal trends are more cleanly discernible.



Reshaping the Data Table

- We determined earlier that each row of the data table represents a month.
- The website has datasets for daily and hourly measurements too.

Why not always just use the data with the finest granularity available?

Reshaping the Data Table

Very fine-grained data can become very large. The Mauna Loa Observatory started recording CO2 levels in 1958. Imagine how many rows the data table would contain if they took measurements every single second!

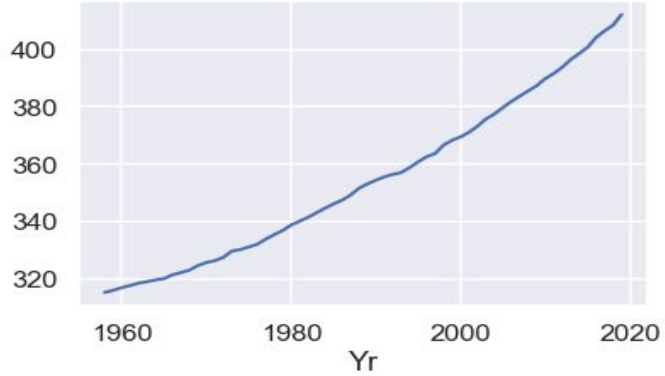
But more importantly, you want the **granularity of the data to match your research question**.

Suppose you want to see whether CO2 levels have risen over the past 50+ years, consistent with global warming predictions. You don't need a CO2 measurement every second. We might only need yearly data.

We can aggregate the data, changing its granularity to annual average measurements.

```
co2_NA.groupby('Yr')['Int'].mean().plot();
```

We see a rise by nearly 100 ppm of CO2 since Mauna Loa began recording in 1958.



Takeaways

- After reading the whitespace-separated, plain-text file into a data frame, we began to check its quality.
- We used the scope and context of the data to affirm that its shape matched the range of dates of collection.
- We confirmed the values and counts for the month were as expected.
- We considered three approaches to handling the missing data: drop records, work with NaN values, and impute values to have a full table.
- And, finally, we changed the granularity of the data frame by rolling it up from a monthly to an annual average.

Summary

Missing data and ways to handle missing data.

Few ways to impute missing data, including deductive, mean, and hot-deck imputation.

Decisions to keep or drop a record, to change a value, or to remove a feature, may seem small, but they are critical. One anomalous record can seriously impact your findings.

Whatever you decide, be sure to check the impact of dropping or changing features and records.

Be transparent and thorough in reporting any modifications you make to the data. It’s best to make these changes programmatically to reduce potential errors and enable others to confirm exactly what you have done by reviewing your code.

THANK YOU!