GitHub Repositories I Can Access

ConnorPestell
xiyue-liang918
Lestiiee
wyy12358
PeisongHao
Funkseko
qsigma
Fuji8ara
JakobMichaelGollreiter
jaredchew0322
xichen1017
dfannjiang
HFossdal
CancanJin-1
zyj1151
WertherZhang
KishC
lunamk
Wentio
Siqiqiang
XiaotongZhan
Tianrui-Y
HarshithaCal
Jun-tsune
brianfdo
jasminekhamuani1906
amashiana
Yirong-Huo
alexzhao-bk
tk-violin
xsgxlz
JHJORE
MMSeguin2003
kathylyf
songsheng01
johnssu714
ibrahim-anagaa
QichengZhu
crescent-rains
Wertwe1
calv2n
yukiasa118
Jaeyeonnn
Bru3e
erztristan
Larshle
GongCao
Rawanalytics
erikIsChamp
OziAnyanwu
Ruiwen-github
jjwang02
ConnorPestell8
sputnick95
XuanlinMao
JANGWON1210
camccaffrey

If your username is in **bold**, your repo is public and needs to be changed to private.

STAT 214 Spring 2025 Week 2

Zach Rewolinski

Heavily Borrowed from Anthony Ozerov's STAT 215A Materials.

Any Course Questions?

Announcements

lab1 has been released in the stat-214-gsi repo. **Due Friday Feb 21 at 11:59pm**

- submit in your stat-214 repo

My OH: **Evans 444**

- Monday 12:30pm-2:00pm
- Other GSI OH at stat214.berkeley.edu

Lab 1

How do I do lab1???

Read through:

`stat-214-gsi/lab1/instructions/lab1-instructions.pdf`

And make sure you are also familiar with:

`stat-214-gsi/disc/week1/lab-instructions.pdf`

Post on Ed or come to OH if anything doesn’t make sense!

stat-214-gsi/lab1/documents/Kuppermann_2009_The-Lancet_000.pdf

Identification of children at very low risk of clinically-important brain injuries after head trauma: a prospective cohort study

*Nathan Kuppermann, James F Holmes, Peter S Dayan, John D Hoyle, Jr, Shireen M Atabaki, Richard Holubkov, Frances M Nadel, David Monroe, Rachel M Stanley, Dominic A Borgialli, Mohamed K Badawy, Jeff E Schunk, Kimberly S Quayle, Prashant Mahajan, Richard Lichenstein, Kathleen A Lillis, Michael G Tunik, Elizabeth S Jacobs, James M Callahan, Marc H Gorelick, Todd F Glass, Lois K Lee, Michael C Bachman, Arthur Cooper, Elizabeth C Powell, Michael J Gerardi, Kraig A Melville, J Paul Muizelaar, David H Wisner, Sally Jo Zuspan, J Michael Dean, Sandra L Wootton-Gorges, for the Pediatric Emergency Care Applied Research Network (PECARN)**

What you should probably do by Tuesday

- Read/skim the paper
- Look through the data documentation
- Figure out how to load the data into Python
- Start working on data cleaning

Start soon (ASAP)! This is not the type of project that can be done with 24hrs and some coffee. Get started!

Next Friday will be about EDA and data visualization!

Data Cleaning

Data cleaning in general

What does “clean” data look like?

Data cleaning in general

What does “clean” data look like?

- No invalid/inconsistent values. This is based on domain knowledge.
  - What do you do with rows containing invalid values? Remove the whole row? Impute the values? Change the values to missing?
- Missing values are appropriately encoded.
  - e.g. `np.nan` instead of -1.
  - It is okay to keep missing values in your clean data as long as your downstream models appropriately address them.
- Nice data format. Ideally a tabular format, but this isn’t possible with all datasets.
- Consistent and clear naming scheme for columns and categorical values
- Appropriate datatypes are used
  - (e.g. categorical data shouldn’t be stored as float, booleans should be bool, etc.)
- Every observational unit should only appear once in the data

See https://vdsbook.com/04-data_cleaning#sec-examine-clean

Data cleaning in general

Data cleaning requires a lot of **judgment calls**!

Good practice: make a data cleaning function, e.g. `clean_data`, which takes in your data, a few parameters, and returns cleaned data. The parameters should control your judgment calls.

If you want to check the stability of your results to data cleaning choices, you can easily **perturb** them by changing the parameters you pass to `clean_data`, and thus make many different cleaned versions with different judgment calls.

Useful tools for data cleaning

- Renaming columns:

```
df.rename(columns={'oldname1': 'newname1', 'oldname2': 'newname2', ...})
```

- Changing column datatype:

```
df['colname'] = df['colname'].astype('Int64')
df['colname'] = df['colname'].astype('str')
df['colname'] = df['colname'].astype(bool)
```

- Replacing all values in a column:

```
df['colname'] = df['colname'].replace({'old_value': 'new_value', ...})
```

- Replacing values in rows satisfying some condition:

```
df['colname'].loc[some_boolean_vec] = 'new_value'
```

- Checking that something in the data makes sense, and throwing an error if it doesn't. A sort of micro reality check on your data cleaning process:

```
boolean_to_check = all(df['some_column']<50)
assert boolean_to_check, 'some_column has values ≥ 50!'
```

NumPy

NumPy is a Python library for working with arrays of arbitrary dimension. Basically the foundation of scientific computing in Python.

Key point: NumPy code is clean and **extremely fast**

```
a = [0, 1, 1, 2, 3, 5]
b = []
for i in range(len(a)):
     b.append(sin(a[i]))
```



```
import numpy as np
a = np.array([0, 1, 1, 2, 3, 5])
b = np.sin(a)
```

If you are not familiar with NumPy, take a look at this introduction in the official documentation:

https://numpy.org/doc/stable/user/absolute_beginners.html

I'd recommend opening up a fresh Jupyter notebook and following along through the examples.

Pandas

Pandas Overview

- Tabular data (think csv with columns representing variables and observations as rows)
- Simple but powerful API for performing data manipulation
- Integrated with NumPy
- Native vectorized functions for efficient operations.
- pandas.Series: 1D vector data structure. Rows and columns.
- pandas.DataFrame: 2D tabular collection of Series.

Tabular Data

columns = attributes for those observations

Rows = observations

Player	Minutes	Points	Rebounds	Assists
A	41	20	6	5
B	30	29	7	6
C	22	7	7	2
D	26	3	3	9
E	20	19	8	0
F	9	6	14	14
G	14	22	8	3
I	22	36	0	9
J	34	8	1	3

Loading data

You can load data from a variety of formats into a DataFrame

```
import pandas as pd
df = pd.read_csv(...)
df = pd.read_txt(...)
df = pd.read_excel(...)
...
```

May need to play around with the parameters of these functions to get the DataFrame to look right!

Indexing and extracting data

	No	year	month	PM2.5	O3	TEMP	pressure	DEWP	wind	station
0	1	2013	3	4.0	69.0	-0.7	1023.0	-18.8	NNW	Guanyuan
1	2	2013	3	4.0	72.0	-1.1	1023.2	-18.2	N	Guanyuan
2	3	2013	3	3.0	69.0	-1.1	1023.5	-18.2	NNW	Guanyuan
3	4	2013	3	3.0	62.0	-1.4	1024.5	-19.4	NW	Guanyuan
4	5	2013	3	3.0	71.0	-2.0	1025.2	-19.5	N	Guanyuan

.loc: access by names

- df.loc[row_names, col_names]

.iloc: access by index number

- df.iloc[row_indices, col_indices]

Note: .loc is end-inclusive, .iloc is end-exclusive.

- df.loc[0:10] and df.iloc[0:10] (row-only indexing) will return different dfs.

Label indexing is better practice when data cleaning. Be careful with changing column names.

Indexing specific rows and columns by label

```
df.loc[[1, 2, 10, 11],['year', pressure, 'wind', 'station']]
```

Equivalently,

```
df.iloc[[1, 2, 10, 11], [1, 6, 8, 9]]
```

Result

	year	pressure	wind	station
1	2013	1023.2	N	Guanyuan
2	2013	1023.5	NNW	Guanyuan
10	2013	1028.2	NNW	Guanyuan
11	2013	1028.2	N	Guanyuan

Indexing and extracting data

Can also index using [] brackets.

- `df[1:10]` selects the 2nd to 10th rows
  - Recall: Python is zero index
- `df[['year', 'pressure']]` selects two columns
  - Note: df[[‘year’]] will return a DataFrame, df[‘year’] will yield a Series

Filtering DataFrame

- E.g. filtering for data from a particular year:

```
df[df[“year”] == 2015]
```

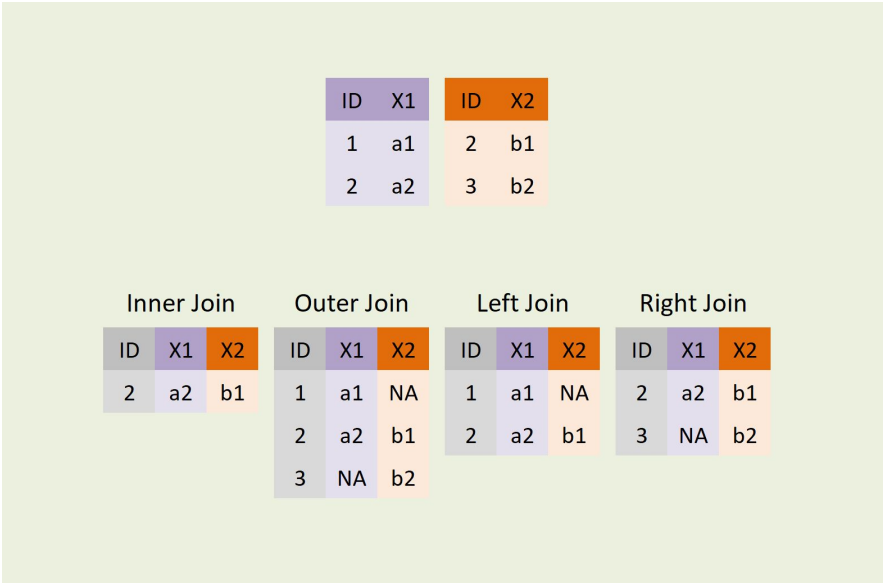
  - The inner expression produces a boolean array with is used to index df.
- Combining logical operators:

```
df[(df[“year”] == 2015) & (df[“pressure”] > 1010)]
```

  - & for ‘and’ operator
  - | for ‘or’
- Search for values from a list:

```
even_years = [“2014”, “2016”]
df[df[“year”].isin(even_years)]
```

Joins



Perform joins similar to SQL with merge:

```
pd.merge(left_df, right_df, how, left_on, right_on)
```

- *how* argument specifies with type of join.
- Supports left, right, inner, outer, cross joins
- left_on, right_on specifies with column(s) to join on from left and right dfs respectively.

Group by

```
df.groupby(['year', 'month'])[['temp', 'pressure']].agg('mean').head(15)
```

		temp	pressure
year	month		
2013	3	6.053629	1012.547446
	4	12.260694	1008.296944
	5	21.374194	1003.162231
	6	23.386111	1000.165556
	7	26.877419	996.033065
	8	26.744355	999.316801
	9	20.077778	1009.355139
	10	12.809812	1016.268952
	11	5.530972	1016.972083
	12	-0.161290	1019.950000
2014	1	-0.138306	1020.051344
	2	-0.097321	1022.388690
	3	9.920430	1013.727554
	4	16.891528	1010.459444
	5	21.176075	1001.558065

Group rows of dataframe with same values in one or more columns then apply an aggregation function

df.groupby(col_name).agg(aggregation_func)

If grouping by multiple columns, pass list to groupby()

Some possible arguments to .agg(): ‘max’, ‘min’, ‘count’, ‘mean’, ‘first’, ‘last’

Useful functions in Pandas

- .value_counts(): apply to a series or dataframe with a couple columns to get counts for each unique value.
- .sort_values(): sort dataframe across multiple columns.
- .cut(): bin continuous values into discrete bins
- .shift(): shifts rows of a column by desired # of steps. Useful for time series data (introducing lags)

Vectorization

Bad

```
squared_temp = []
for tmp in df['temp']:
    squared_temp.append(tmp**2)
pd.Series(squared_temp)
```

Good

```
def square_func(z):
    return z*z
df['temp'].apply(square_func)
```

Can do this with general functions

Built-in Pandas methods provide fast and simple data processing. These native functions are optimized for performance

Always avoid manually looping over entries.

NumPy/Pandas methods will be much cleaner and faster.

Performance differences will matter when we are working with large datasets.

See:

`stat-214-gsi/discussion/week2/pandas.ipynb`

for some pandas examples and practice, if you would like.