Lab 2

After Lab 1, you should be able to import data, be able to determine a few quick facts about that data, and articulate the purpose of unit testing. Today we going to extend on our work from last time. Our goals are to:

1. Use GitHubActions for Continuous Integration
2. Use `pandas` to select parts of data, and
3. Use `matplotlib`, `pandas`, and `seaborn` to make images.

Before starting...

A few questions before we begin:

- Do you need to `pull` anything?
- Do you need to copy anything into a folder?
- Are you in the right `env`ironment?

These reminders will smaller and smaller as we get used to the flow of this course.

Continuous integration with GitHubActions

In Lab 1, we learned about unit testing. We were able to create and locally run our unit tests. Now we will shift to using GitHubActions to continuously test our code against the unit tests. Continually testing our code with each push is called **continuous integration** or CI.

For this part, I have created an assignment called `Lab02-GitHubActions`. Just as you did for Lab 0, please click on this link and create your own local repository. In this repo, you will find two files:

1. `.gitignore`, and
2. `requirements.txt`

Add files from Lab 1

Before continuing, add your two python files from lab 1 (not the jupyter notebooks) and the data file to this directory. Then commit your changes.

Note: You can have two terminals (or command prompts) open at a time. So you might choose to have one open for your `student-labs` and a second for the git-enabled `Lab02-GitHubActions` folder.

Creating the GitHubActions

On Moodle, you will find a video of this procedure. The next steps are the written out version of this video on Panapto. You are welcome to use just one or both as you navigate creating your continuous integration (CI) workflow.

Now we need to create a testing action on your repository on GitHub:

1. Visit your personal `Lab02-GitHubActions` repo on GitHub.
2. At the top of the repo, you will see the usual menu bar: *Code*, *Issues*, *Pull Requests*, *Actions*, etc. Click on **Actions**
3. GitHub is going to try guess which kind of action you would like. You are looking for the **"Python Application"** choice under **"Continuous integration."** When you find that one, click **Configure**
4. The next screen will be a YAML file called `python-app.yml` inside a newly created hidden directory called `.github/workflows/`. This `.yml` file tells GitHub Actions how we want these tests to run. We are going to make a few edits to this file. First delete all references to `flake8`. This happens first on line 26 and then lines 28-33.
5. Then click on **Start commit.** You can commit right to `main`. (In the video, I add a bit to the messages associated with the commit, but you are fine to leave that as is.) Then click **commit new file.**
6. Now you are back in the newly created hidden directory `.github/workflows/` and you should see your `.yml` file there.
7. Now click *Actions* again. You should see a header **All workflows** and under that your commit message with a **Yellow** dot next to it.

GitHub Actions uses color to tell you want is happening:

- **Yellow** means that GitHub Actions is thinking
- **Red** means that either the build or the tests have failed
- **Green** means that the build and the tests have worked!

Two notes:

- For GitHub Actions to "build" things correctly, you will need to have both `requirements.txt` and an associated `.yml`.
- Also as noted in Part 5 of McFee and Kell's Open MIR Tutorial, it is unlikely that your tests will pass locally, but fall on GitHubActions. When this happens it is usally due to a difference between your local machine's set-up and what you `require` (via `requirements.txt`) for the virtual environment.

For homework 1, you will need to submit green boxes showing that you have correctly linked your repo to GitHubActions and that your tests pass.

Note: Only the most recent tests need to pass. On your repo, in the grey bar under the **Green** **Code** box, you should see information about your commit. There will be a small icon to the left of that information.

- If you see a **Green** check, then your most recent tests are passing.
- If you see a **Red** X, then the most recent tests are not passing.

Resources consulted

Including continuous integration (CI) in this course was motivated by Part 5 of McFee and Kell's Open MIR Tutorial. Their tutorial used Travis, which was one of the most popular CI platforms at the time. This is not longer the case, which is why we are not using travis.

This text was adapted from the original (travis-based) Lab 2, which was adapted from Part 5

Preparing to code today

We need an extra package today: `seaborn`. If you run into issues (such as `not found` or other errors), please use `conda` to add this to your environment. (Also, double check that you are in your `csc294` environment! This one always gets me.)

In [ ]:
```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Selecting subsets with `pandas`

In Lab 1, we learned about data frame structure within `pandas` and for class today, you read about some of the functionalities of `pandas`. Today, we will practice various kinds of slicing and how we can create new dataframes from previous ones.

Let us start by importing the Forest Fires dataset from Lab 0 and reminding ourselves of what the first 5 rows look like. Add the `forestfire` file to the directory that you are using to work on Lab 2.

In [ ]:
```
ffire = pd.read_csv("forestfires.csv", sep=',')
ffire.head()
```

Here we have a small snapshot of our data. Before moving on, what do you think the **bold** means? (Make a few notes below)

In `pandas` we can select slices of our data either by row or by column using either the names or counters.

Selecting by Row

The two most common ways to access rows are by using `.loc` and `.iloc`, which stand

for *location* and *index location*, respectively.

In this data, the names of the row indices are the same as the counter. So we will find that we can access each row in *nearly* identical ways. Note that this is quite ***unusual***.

The general format for `loc` and `iloc` is row information first, followed by column selections (if desired). Try the below examples and note 1) how they are different and 2) what the arguments are:

In [ ]:
```
ffire.loc[2]
```

In [ ]:
```
ffire.loc[2,'month']
```

In [ ]:
```
ffire.loc[[0,2,5]]
```

In [ ]:
```
ffire.loc[0:5,'month']
```

In [ ]:
```
ffire.loc[0:5,['month', 'wind', 'day']]
```

In [ ]:
```
ffire.loc[[0,2,5],['month', 'wind', 'day']]
```

Note!!!! `.loc` does not slice in the usual manner for `python`. Notice that the last index is *included*. This is very unusual.

So far we have just used `.loc`. Repeat all the above examples using `.iloc`; do they all work?

In [ ]:
```
# Code Block for you!
```

The big difference between `.loc` and `.iloc` is that the first expects the index *names* for both the rows and the columns, while the second expects the index *numbers*. So to make `.iloc` work, we need to have the index number for each column. Try again below:

In [ ]:
```
# Code Block for you!
```

Selecting by Column

Extending the ideas from before, we can extend the ideas from above using `.loc` and `.iloc`, to get the full columns (ie. all the rows). How might you do this? Experiment with the two below blocks:

In [ ]:
```
ffire.loc[?????,['month', 'wind', 'day']]
```

In [ ]:
```
ffire.iloc[?????,[2,10,3]]
```

Another way to access the columns uses `[]` to do our slicing:

In [ ]:
```
ffire[['month', 'wind', 'day']]
```

Your turn

Experiment with selecting different parts of the forest fire dataset in the below code block.

Also, note that this is just the beginning of what we can do with `pandas`.

In [ ]:
```
# Code Block for you!
```

Getting started with `matplotlib`

For this course, most of our visualizations will be built using `matplotlib`. There is another language called MATLAB which can be used for scientific computing (of largely numerical data). MATLAB is optimized for matrix and vector operations, but unlike python, it is not freely available. Along with its powerful computational tools, MATLAB can create data-based images. As the transition away from MATLAB and towards python began, there was a need for all the functionality of MATLAB to exist within python. So the creation and popularity of `matplotlib` makes a lot of historical sense.

In this section, we will just begin to scratch the surface of what `matplotlib` can do. This section of the lab is based on Chapter 9 of *Python for Data Analysis* applied to the forest fires dataset.

Importing `matplotlib`

You might notice three new lines in our import block.

1. `%matplotlib inline` allows our plots to show up inline within our jupyter notebook
   Note - This **must** come before importing *matplotlib*
2. `import matplotlib.pyplot as plt` imports the plotting part of *matplotlib*
3. `import seaborn as sns` imports *seaborn* for our plots using *pandas*

For this first part, we will import our forest fire data as a *numpy* array:

In [ ]:
```
# Import forest fire data as a numpy array

ffire_np = np.genfromtxt("forestfires.csv",delimiter=",",skip_header=1)
```

First plot

In `jupyter` we have to do everything for our plots in one code block. So I will be adding comments within the next code block to explain each line.

In [ ]:
```
# Create the figure
# In MATLAB, as in `matplotlib`, plots exist within Figure objects.
fig = plt.figure()

# Create each of the subplots in their locations
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)

# Plot the wind column in order by the rows
# Q1: Where is this plot placed?
# Q2: Would it stay there if you changed the order of the 'ax' lines?
plt.plot(ffire_np[:,10], "k--")

# Create a histogram of the wind information
_ = ax1.hist(ffire_np[:,10], bins=10, color="b", alpha = 0.3)

# Create a scatter plot of the X and Y columns
ax2.scatter(ffire_np[:,0], ffire_np[:,1])
```

Your turn!

Make a `matplotlib` plot of the forest fire dataset.

In [ ]:
```
# Code Block for you!
```

There is a lot that `matplotlib` can do, as seen in the matplotlib gallery. But like MATLAB and `numpy`, it is limited to largely numerical data. For example, notice that we had to count to where the 'wind' column was to access it.

If we want additional functionality, we are going to make use of some `pandas` functionality as well as the `seaborn` package.

Quick plots with `pandas`

We can make a few quick plots on our `pandas` data frames. Notice that these are built on top of `matplotlib`.

In [ ]:
```
ffire.hist('wind',bins = 10)
```

In [ ]:
```
ffire.plot('X','Y','scatter')
```

Your turn!

Make a `pandas` plot. Check out this helpfile for ideas.

In [ ]:
```
# Code Block for you!
```

More control with `seaborn`

The above plots are great first images, but we may want to generate more complex plots with layers. This is where we reach for `seaborn` which is built on top of `matplotlib` (similar to how `pandas` is built on `numpy`).

In this section, I am referencing the scatterplot and stripplot helpfiles for `seaborn`.

Scatterplot with colors in `seaborn`

We will build a series of plots, adding a new feature to each one. For each plot, note what was different in the code and that difference produced.

We begin with the basic `scatterplot`:

In [ ]:
```
sea1 = sns.scatterplot(x="wind", y="temp",data=ffire)
```

Now we will add color by month when that the fire occurred:

In [ ]:
```
sea2 = sns.scatterplot(x="wind", y="temp",
                       hue="month",data=ffire)
```

Let's change the style of the markers based on the day of the week:

In [ ]:
```
sea3 = sns.scatterplot(x="wind", y="temp",hue ="month",
                       style="day", data=ffire)
```

Let's vary the size of the data points based on the area that was burned:

In [ ]:
```
sea4 = sns.scatterplot(x="wind", y="temp", size="area", data=ffire)
```

Noticing that many of the points lie on top of each other, let's make a `stripplot` to view the data in another way:

In [ ]:
```
sea5 = sns.stripplot(x="day", y="temp", data=ffire, jitter=True)
```

Final Thoughts

To finish up this lab, create two posts on the **#lab02_submission** channel on slack sharing

1. a screenshot of all your tests passing on GitHubActions
2. a plot of the data that you imported in lab 1 that uses colors.

If your have questions from this lab, post them to #lab_questions with the same preamble (i.e. starting with **Lab2**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Resources consulted to build this lab:

1. *Python for Data Analysis*
2. Part 5 of McFee and Kell's Open MIR Tutorial
3. Wikipedia about hidden files
4. Ignoring Files and Directories in Git with .gitignore
5. Stackoverflow question about ipynb checkpoints: How to git ignore ipython notebook checkpoints anywhere in repository
6. Selecting Subsets of Data in Pandas: Part 1
7. Using matplotlib in jupyter notebooks — comparing methods and some tips [Python]
8. 04.00-Introduction-To-Matplotlib.ipynb
9. plot in pandas
10. seaborn scatterplot helpfile
11. seaborn stripplot helpfile
12. plotting in pandas helpfile