Lab 6 – Data preprocessing I

Submission: `lab06.ipynb`, pushed to your Git repo on `master`.
Points: 10
Due: Monday, February 4, 11:59pm

Objectives

- Experience the "joy" that is data munging. Munge, munge, munge!
- Start dealing with noisy, unclean, real-world data
- Work with times and dates in your data

Introduction

As you learned in class, data cleaning represents a large part of the work of the data scientist. You are going to download a real-world dataset, and do some preliminary cleaning, EDA, and reporting.

Preparing for your lab

Create a **`lab06.ipynb`** file. Create your header cell, then your first cell to set up your imports:

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

Create a new folder at the same level as your labs folder, called **`data`**. This folder will store the data that you are working with through the semester. I will try to have most data we work with available online. But sometimes (such as this exercise), the data will need to be downloaded from public repositories. Again, this is an essential skill! Most data are not already cleaned and tidied up for you to play with!

Work through this lab, and enter the answers to questions that are scattered throughout this lab

Local Climate Data

Go to the *The Pennsylvania State Climatologist*; a direct link to their data archive is: http://climate.psu.edu/data/ida . From this page, select *FAA Hourly* data. Select the entire hourly data for ALL of 2018. (i.e. select the date range of `2018-01-01` through `2018-12-31`.) You are going to investigate the weather observations for last year at Williamsport, PA, whose FAA code is `KIPT`. When prompted, select EVERY attribute to download (from Date/Time, Number of observations… etc… right through Max Wind Speed). Output file type should be a CSV file, and select **Yes** to include Metadata. (Technically, you could select No, but I want you to experience how some files will be often presented to you in practice.)

Download the data, and save the raw .csv file using the default file name provided by PSU. It'll be a long-ass filename. That's fine. Keep it. It's informative. You should always name your data files with long, self-descriptive names when possible. Then, you will need to use the following command to read in your data:

1) [P] Before you begin, print out your current working directory to understand where in your file system Python thinks your script is running from

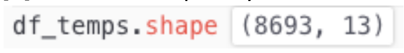
2) [P] Now, use pandas to read in your data file you downloaded above, which you should have placed in your **data** directory. Call the data frame `df_temps`. Read in the entire dataset.

   NOTE: BE SURE TO LOOK AT YOUR ACTUAL DATA BEFORE TRYING TO READ IN A RAW DATASET! JUST BECAUSE A DATASET HAS A .CSV EXTENSION DOES NOT MEAN THAT YOU CAN RELY ON EVERY ROW BEING A PROPERLY FORMATTED ROW! For instance, notice that the header row is scattered throughout your data! Notice that you have some extra columns at the end that are consistently empty! Noobs are tempted to manually edit the file to make it easy to read. NO. WRONG! BAD DATA SCIENTIST! NAUGHTY! Write your Python script to properly preprocess the file. Why? In practice, your data file may be huge. You may need to repeatedly grab fresh data, that will only have the same issues. Do you really want to repeat your manual editing silliness every time you have a fresh file? No! **It may take a bit more work up front, but ALWAYS strive to write code to preprocess every aspect of your data file!** It will always save you work later!

   You are done with this first step when you have a data frame with 8693 observations and 13 variables.

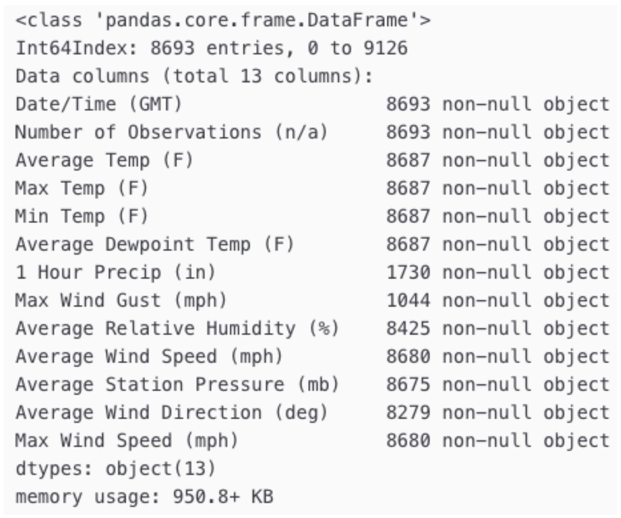
   Leave the default type of every variable as `object`.

3) [P] Show the shape of your dataframe. it should be:



```
df_temps.shape  (8693, 13)
```

4) [P] Show the result of `info()`. It should look like the following:



```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8693 entries, 0 to 9126
Data columns (total 13 columns):
Date/Time (GMT)                    8693 non-null object
Number of Observations (n/a)       8693 non-null object
Average Temp (F)                   8687 non-null object
Max Temp (F)                       8687 non-null object
Min Temp (F)                       8687 non-null object
Average Dewpoint Temp (F)          8687 non-null object
1 Hour Precip (in)                 1730 non-null object
Max Wind Gust (mph)                1044 non-null object
Average Relative Humidity (%)      8425 non-null object
Average Wind Speed (mph)           8680 non-null object
Average Station Pressure (mb)      8675 non-null object
Average Wind Direction (deg)       8279 non-null object
Max Wind Speed (mph)               8680 non-null object
dtypes: object(13)
memory usage: 950.8+ KB
```

5) This is a small dataset compared to many that data scientists work with in the field. However, it does give you a chance to understand how important it is to select the smallest, yet most accurate data type for every variable. This is particularly true with respect to your memory footprint. With enormous data involving millions of records, you often need to perform various paging exercises to load in chunks of data into memory, substantially slowing down the machine learning methods. In other words, the more data you can fit in memory, the better!

   [P] Read about the `memory_usage()` method of pandas data frames. Then, report the total memory in bytes for each variable of `df_temps`. Also, report the *total* memory required for the data frame. Set the parameter `drop=True`, to get the most accurate assessment of your total memory usage.

6) [P, M] Report the current index. You have 8693 observations, but the index still goes from 0 to 9126. Why?

7) [P] Reindex your data, and show that the new index is indeed reset. (There are many ways to do this. I suggest using `reset_index()`. There is no need to retain the original index, so `drop=True` is fine.)

8) [P] I cannot emphasize this enough – you will get the most out of your data when you take the time to set up the most accurate type for each variable. Currently, the type of every variable is `object`. However, notice that in your raw data file, EVERY variable is a number except the first variable, which is a date. Dates are COMMON in data, and it is important that you represent dates as actual date types! We'll deal with that shortly.

   Convert all numeric data to actual numeric data types. You'll need to look up how to do this. (HINT: `pd.to_numeric()` is your friend.) Leave the NaN fields alone! The fact that they are missing is IMPORTANT! And, leave the date/time variable in the first column alone.

   You should output the shape of your data, and show `info()` to show every variable is a floating point number

9) [P] Show the current total memory usage after converting your data types? There should be a substantial drop in your memory footprint! Report the percentage that your memory was reduced.

10) [P] Did you notice that `to_numeric()` has a parameter called `downcast`? Go back and read about this parameter. By default, most of the time your integer types will be converted to a 64-bit integer, and floating point types will use double precision numbers. You can do even better. Read about this parameter, and downcast your types accordingly. Report the percentage that your memory was reduced from the previous step, as well as from your first

11) At this point, you should have good data to start working with (with the exception of the date column. Verify it by outputting the results of `describe()`. Every variable should have its basic stats reported.

Data Transformation with Dates

It is very common to deal with dates in data. Unfortunately, few organizations around the world have agreed to one format for universally representing dates in data. And, then you have time zones to deal with. We'll discuss that later. Let's suppose we wanted to represent January 15, 2016, depending on your location in the world, the date might be stored in the data as:

- `01/27/2019`
- `01/27/19`
- `27/01/2019`
- `27.01.2019`
- `2019-Jan-27`
- `January 27, 2019`
- `27-Jan-2019`
- `20190127`

And, there are others. These are all acceptable formats. However, pandas will not know how to interpret this data other than being viewed as a general `object` type. It's up to YOU to make sure you convert your data to the most appropriate type.

Generally speaking, when your data consists of a series of observations recorded over time, we refer to these types of data as time series data. And, usually every observation will have a time or a date variable that identifies when the observation was recorded.

This page has just about everything you need to deal with dates with time series data. It has far more than you'll need. https://pandas.pydata.org/pandas-docs/version/0.23.4/timeseries.html . As with most of the API with core packages like pandas, there is a LOT to absorb, and at best, you'll just become familiar with how to find the answers you are after in their documentation!

This portion of the lab will help you learn how to confidently work with dates and times in data.

12) [M] What are the four primary classes in `pandas` for working with dates and times? What is each used for?

13) [M] What is the name of the pandas function that is used to convert string objects or other types to a `Timestamp` object?

14) [P] Create a `Timestamp` object from the string "`07/04/19`", which is a date representing July 4, 2019. Store the object as `d1` and show it.

15) [P] Using `d1` and string formatting capabilities, print the string

    `"Today's date is Thursday, July 4, 2019".`

16) [P] Create another `Timestamp` object representing Sept 7, 2019 at 3pm, called `d2`. Report it

17) [P] Subtract `d2 – d1`, and report the difference as the number of days and seconds between these two. Also report the difference as total seconds. (NOTE: The difference should be 65 days, 54000 seconds. Or 5670000 total seconds.)

18) [P] Create a new `Timestamp` object from the string "`2019-07-01 08:30pm`", but, localize the time stamp to represent the time in the **US Eastern Time Zone**. Store the result as `d3` and output it.

19) [P] Show time represented by `d3`, but converted to the **US / Pacific Time Zone**. The time reported should be three hours earlier than EST shown in the previous question.

20) [P] Create a `Timestamp` object representing right now, stored as `ts_now`. Report the result.

21) [P] Create a `Timedelta` object representing 1 hour, stored as `td_hour`. Report the result.

22) [P] Demonstrate how you can do basic mathematical operations by adding 6 hours to `ts_now` using `td_hour` and basic math operations. (i.e. No loops or further calculations necessary!)

23) [P] Create a `DatetimeIndex` object that represents every hour during the month of January, 2019. The first index should be midnight, January 1, 2019, and the last index should be January 31, 2019 at 11pm. Store the object as dr. (HINT – use the `pd.date_range()` method!)

Usually, the index to a dataframe represents the data you will use most often to access and select your data. In the case of a time series dataset, the index should be time. In other words, every observation should be indexed by a `Timestamp` object! You'll make that happen next…

24) [P] Now, deal with that first column of data. It's currently an `object`. Use it to form the **index** of `df_temps` to be a `DatetimeIndex` type. NOTE: You can NOT simply generate this column using your own date range object! You must generate it directly from the actual time/date stamp in the data! Why? **This is very important. Do NOT ever be fooled into thinking any real-world dataset you are dealing with is 100% complete.** If you simply try to use a date range between 1/1 – 12/31, with every hour, you are making an

incorrect assumption that every observation is present! WRONG!

25) [P] Confirm that your index is indeed matching your first column of data, then use the `drop` method to eliminate the first column of time / date data. It is now your index, and thus there is no need to keep this information twice.

26) [P] Give one final report on the % memory reduction made now, compared to when you first loaded in the data. Again, please take this seriously. This is a substantial amount of memory saved! Why? Because you took the time to properly process every column to have it represent its most accurate type, using the smallest type necessary. HUGE savings!

27) [P] As the previous question suggested, this dataset has missing observations! How may records are missing? Compute this by reporting the number of observations you expect to see with 24 observations over 365 days, and then report the number of actual observations. What is the difference? This is the number of observations that are potentially missing!

28) [P] Time to investigate. Write code to perform a sanity check on the occurrence of index entries for *every* hour of *every* day between Jan 1, 2018 at midnight, through December 31, 2018, at 23:00. Report the observations that are missing, and any observations that are duplicate. Report the total quantity of both.

Congratulations! At this point, you performed your first real-world example of what you need to go through to complete basic preprocessing steps! Are you done? Bwhahaha! You knew the answer to that already.

29) [P] The next step is to assess missing data in each variable. Use the `isna()` method on `df_temps` to report the total number of entries in each variable that have missing values.

30) [M] Which variables seem to have the most consistent, complete observations? Which are missing the most? Are they really "missing", or are they observations where an event did not occur? Discuss.

31) [P] Report the time stamps that have missing temperatures. Do you see a pattern? Do they happen on a particular day of the week? Or time?

32) [P] Create a new categorical variable in `df_temps` called "`Quarter`", an ordinal, that is "`Q1`" if the month is 1-3, "`Q2`" if the month is 4-6, "`Q3`" if the month is 7-9, and "`Q4`" if the month is 10-12.

33) [P] Draw a boxplot showing the distribution of the average temperature over the entire year.

34) [P] Draw a boxplot showing the distribution of the average temperature for each quarter.

35) [P] Plot the average temperature for the entire year. (NOTE: Some plots are very simply completed using matplotlib. You can still obtain the look and feel of seaborn by using `sns.set_style()` first, then `plt.plot()`)

Deliverables

Commit and push `lab06.ipynb`. Be sure you have every cell run, and output generated. Verify that your file is pushed properly on Gitlab.