Lab 6 – Data preprocessing I

Submission: `lab06.ipynb`, pushed to your Git repo on `master`.
Points: 10
Due: Wednesday, February 5, 2pm

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

The Pennsylvania State Climatologist Database

Penn State has an excellent public database of weather observations collected from a wide range of stations scattered throughout the state. Some of them go back to the 1940s. For this lab, we're going to explore one of those datasets – Williamsport, PA.

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. You are going to investigate the weather observations Williamsport, PA, whose FAA code is `KIPT`. Select it.

On the next screen, enter the following:

- Start and End Dates: 2000-01-01 to 2019-12-31.
- 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

- Select **Yes** to include Metadata. (Metadata is info about my data. This usually contains valuable information, and you almost always want to retain this information.)

Download the data, and save the raw .csv file using the default file name provided by PSU. It'll be a long filename. That's fine. I usually always add the suffix "`_raw`" to indicate this is the raw data that I'm working with from my source. Never lose track of your original dataset.

You are not done until you have a data file in your **`data`** directory (which should be at the same level as your labs directory.) If you placed the .csv file in the correct place, then your path should be:

`"../data/faa_hourly-KIPT_20000101-20191231_raw.csv"`

Exercises

Before you begin, you should have a single .csv data file as named above. If not, then complete the above exercises to download the data from the PSU climate database.

1) [P] Use pandas to read in your CSV 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, however, peek at the dataset first. You'll notice 16 rows of metadata. Ignore the first 16 rows (HINT: Use the `skiprows=` option!)

   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! The inexperienced are tempted to manually edit the file to make it easy to read. NO. WRONG! BAD DATA SCIENTIST! Write your Python cleaning code to always work with raw, uncleaned data. 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!

2) Report the general structure of the data frame using `df_temps.info()`. You should notice that almost every variable was read in as a plan object data type. You have a lot of work to do!

   Your result should look as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 181914 entries, 0 to 181913
Data columns (total 14 columns):
 #   Column                        Non-Null Count   Dtype
---  ------                        --------------   -----
 0   Date/Time (GMT)               181914 non-null  object
 1   Number of Observations (n/a)  181914 non-null  object
 2   Average Temp (F)              180915 non-null  object
 3   Max Temp (F)                  180915 non-null  object
 4   Min Temp (F)                  180915 non-null  object
 5   Average Dewpoint Temp (F)     180809 non-null  object
 6   1 Hour Precip (in)            37596 non-null   object
 7   Max Wind Gust (mph)           32206 non-null   object
 8   Average Relative Humidity (%) 177416 non-null  object
 9   Average Wind Speed (mph)      181372 non-null  object
 10  Average Station Pressure (mb) 181636 non-null  object
 11  Average Wind Direction (deg)  149252 non-null  object
 12  Max Wind Speed (mph)          181372 non-null  object
 13  Unnamed: 13                   0 non-null       float64
dtypes: float64(1), object(13)
memory usage: 19.4+ MB
```

This is a pretty good dataset with lots of real problems! It gives 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!

3) [P] Read about the `memory_usage()` method of pandas data frames. Then, report the total memory in bytes for each variable of `df_temps`. Set the parameter `deep=True`, to get the most accurate assessment of your total memory usage. (NOTE – this could take a bit of time to return an answer.)

4) [P] Report the ***total*** memory required for the data frame in MB. (Just sum the previous answer.) You should get an answer showing over a hundred megabytes! Also, store the total as a variable called `original_memory`. We're going to compare memory after we're done.

5) [P] Remember those extra column header lines that appeared throughout the entire CSV file? You need to get rid of those. Write the code to eliminated those from `df_temps`. (This is tricky. Think about it… you are selecting the data that does NOT have `columns[0]` as the first value in the observation!)

   HINT – At this point, you should have 173252 observations.

6) [M-P] Examine the last entry of your index (i.e. `df_temps.index[-1]`). Then, show the number of observations in `df_temps`. These are unequal. Why?

7) When you permanently delete observations, it's usually a good idea to reset your index, especially if the index is nothing more than a unique number to each observations. 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) You have a rather annoying extra column that was read in in the last column position. (Look closely at the output of `describe()` above!) You should always confirm that it's garbage before deleting it. Write the single line of code that reports the *count* of valid values in the last column (HINT: `count()`)

9) Drop that last column from `df_temps`.

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. Let's continue cleaning this up.

10) [P] 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 except the date/time field in the first column, which should still be of type `object`.

11) [P] How much did our memory footprint improve? (Show the total memory usage using deep=True). Report the total memory usage in MB, and report the percentage improvement.

12) [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 far better. Downcast your types accordingly. Report your latest memory usage in MB.

13) At this point, with the exception of the date column, you should have good data to start working with. 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. Adding to the complexity are time zones that you must 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! Insanity! Can't we all just get along??? Well, apparently, no. At least when it comes to date (and times, and currencies, and etc.) The fact is that these are all acceptable formats for dates. Sometimes pandas can do a pretty good just detecting date fields. However, as we noticed in this case, it was not (mostly because of those extra lines of metadata at the top.) **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/stable/user_guide/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.

14) [M] There are four primary classes in `pandas` for working with dates and times? Consider the Scalar Class for each, and state what concept each is representing.

15) [M] For each above, state the primary creation method used to create each type of data

16) [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.

17) [P] Using `d1` and string formatting codes, print the string from `d1`:

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

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

19) [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.)

20) [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.

21) [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.

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

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

24) [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!)

25) [P] Create a `DatetimeIndex` object that represents every hour during the month of January, 2020. The first index should be midnight, January 1, 2020, and the last index should be January 31, 2020 at 11pm. Store the object as `dr`. (HINT – use the `pd.date_range()` method!)

OK, so that was a little practice with understanding how to work a bit with dates and times. They are objects, with lots of methods to help you access those timestamps in different ways.

Back to our weather data. 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 is usually the time. In other words, every observation should be indexed by a `Timestamp` object! You'll make that happen next…

26) [P] The first variable in our data is currently an `object`. But, notice the name and its units? It's a date/time in the GMT time zone! Convert the first column of data into an actual time stamp.

    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.** There are missing observations in these data, and your data will be massively flawed if you neglect this! 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.

    (HINT: Go back to your reference table. You are creating an array of timestamps. Which function? Either `to_datetime` or `date_range`. We already told you that `date_range` is wrong!)

27) [P] Confirm that your first column data type is now a timestamp by showing the output of `df_temps.info()`. (It should show that it is `datetime64`, to be exact) . Then, show the values of the first column of the first AND last row only. Your result should look like:

```
0         2000-01-01 00:00:00
173251    2019-12-31 23:00:00
```

28) Finally, let's move that first column to be the new index for your dataframe. Use the `set_index` method of of `df_temps` to be the first column of data, then use the `drop` method to eliminate the first column. It is

now your index, and thus there is no need to keep this information twice.

29) [P] Give one final report on the total memory usage, and also show the % memory reduction made compared to when you first loaded 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!

30) [P] This dataset has missing dates. But, how many? First, calculate how many observations SHOULD be there. Use the difference between the first and last index value to compute this.

31) [P] There are quite a lot! It's time to investigate. Create a data frame called `df_missing` that has an index of the time stamp of every missing date, with a simple variable called "missing" that has a value of 1 for every entry. (i.e. it should only contain the missing dates.) Report the number of rows in `df_missing`. It should match the number you computed previously.

32) [P] Let's get a sense of which years seem to be missing the most dates. How? Well, the easiest approach is probably to use the `resample()` method of data frames. Check out this section: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling . This method works phenomenally well for grouping and aggregating your data when you have a datetime index type!

We're going to resample our data by year, and perform a count aggregation all in one line:

Enter the following:
```
df_missing_by_year = df_missing.resample('Y').count()
```

There are many, many ways you can resample your data. You need to jump over to the options for dateoffset objects. The letter codes are specified there:

https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects

Show the result of `df_missing_by_year`

(HINT: The first five rows of your new data frame should look as follows:

`df_missing_by_year`

	missing
2000-12-31 00:00:00+00:00	792
2001-12-31 00:00:00+00:00	54
2002-12-31 00:00:00+00:00	30
2003-12-31 00:00:00+00:00	39
2004-12-31 00:00:00+00:00	72

33) [P] You can see that pretty much every year has missing data. Not uncommon. However, one year in particular is really bad. Which one? Write the code to eliminate that entire year from `df_temps`.

Congratulations! At this point, you performed your first real-world example of what you need to go through to complete basic preprocessing steps!

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.