Exercise: Introduction to Seaborn

Summary

This exercise uses a range of visualizations to explore data from the California Solar Initiative (CSI) program. The program ran from 2007 to 2019 and provided incentive payments for installing solar panels on California buildings. Homeowners and firms carrying out solar projects would apply to the program for the payments, and the exercise examines data from those applications.

Input Data

The input data is **ca_csi_2020_pkl.zip**, a zipped pickle file that will need to be downloaded from the course Google Drive folder. It contains one record for every CSI application, and it was last updated in early 2020. The full database contains 124 variables but for this exercise it has been trimmed down to thirteen: `"incentive"`, the dollar value of the incentive payments provided on the project; `"total_cost"`, the total cost of the project; `"nameplate"`, the project's rated DC power output in kW; `"app_status"`, the status of the application; `"sector"`, the host sector where the project was built (residential, commercial, etc.); `"county"`; `"state"`; `"zip"`; `"completed"`, the date the final incentive payment was sent; `"third_party"`, a string indicating whether the owner of the solar array is not the host customer; `"inst_status"`, the status of the system; `"type"`, a variable indicating whether the array tracks the sun or is fixed; and `"year"`, the year from the `"completed"` field. **firstlines.csv** provides a few lines from the file.

Deliverables

The deliverable for this assignment is a script, **figures.py**, that generates a range of figures using pandas and seaborn.

Instructions

1. Import `pandas` and `matplotlib.pyplot`, and import `seaborn` as `sns`.
2. Set the default DPI for plots to 300. Seaborn is built on Matplotlib so this will set the resolution of Seaborn plots as well.
3. Set the default style for Seaborn by calling `sns.set_theme()` with the argument `style="white"`.
4. Create dataframe `pv` by using `pd.read_pickle()` to read `"ca_csi_2020_pkl.zip"`.
5. Print an informative message and then call the `.info()` method on `pv`. The `.info()` method prints the name of each column, the column's count of non-null values, and the column's datatype.
6. Set up a list called `catvars` that includes the strings `'app_status'`, `'sector'`, `'state'`, `'inst_status'`, and `'type'`. All of these are categorical variables.
7. Produce a quick overview of the values taken on by the categorical variables. Use variable `var` to loop over `catvars` and within the loop do the following:
    1. Print `var`.
    2. Print the result of calling `.value_counts()` on the `var` column of `pv`.
    3. Set `fig` equal to the result of calling `sns.catplot()` with arguments `y=var`, `data=pv`, and `kind="count"`. For future reference, note that `catplot()` produces Figure-level plots (see the notes section for more discussion).

    4. That's it for the loop. It is not necessary to save this batch of plots: they're just temporary figures showing one approach for getting a sense of what's in an unfamiliar database.

8. Now filter the dataset down to residential projects that have been completed and are installed. Here we'll do it in three steps because that would be convenient if the process seemed to eliminate too many records and we needed to go back to see which comparison was filtering them out. First, set `res` equal to the result of calling `.query()` on `pv` with the argument `"sector == 'Residential'"`.
9. Next, set `res` equal to the result of calling `.query()` on `res` with the argument `"app_status == 'Completed'"`.
10. Finally, set `res` equal to the result of calling `.query()` on `res` again but this time with the argument `"inst_status == 'Installed'"`.‘
11. Print a message indicating the number of original records in `pv` and the number in `res` after filtering.
12. Convert the `"year"` column of `res` to an integer by using the `.astype(int)` method. (FAQ1)
13. Now we'll build a couple of additional figures showing project counts, and this time we'll save them. Start by using variable `var` to loop over a list consisting of the strings `"third_party"` and `"year"`. Within the loop do the following:
    1. Set `fig` equal to the result of calling `sns.catplot()` on `y=var`, `data=res`, and `kind="count"`.
    2. Save the figure using `f"res_{var}.png"` as the filename.
14. Now we'll look into the nameplate capacity and cost of the systems in more detail. First, set `n_last` equal to the length of `res`. Then drop any records that are missing data for those specific fields. (FAQ2)
15. Then, set `n_now` to the new length of `res` and use it and `n_last` to print an informative message indicating the number of records dropped.
16. Now create a figure with two panels in a row by setting `fig, (ax1,ax2)` to the result of calling `plt.subplots(1,2)`. The 1 and 2 in the call indicate the number of rows and columns of panels in the plot.
17. Next, call `.plot.hist()` on the `"nameplate"` column of `res` with the argument `ax=ax1` to put the histogram in the left panel. Just to be clear, this is a straight pandas call: we're not using seaborn yet.
18. Use the `.set_title()` method of `ax1` to set its title to `"Nameplate"`.
19. Now call `.plot.hist()` on the `"total_cost"` column of `res` with the argument `ax=ax2`. Then set its title to `"Cost"`.
20. Tighten the figure's layout using `.tight_layout()`. (FAQ3)
21. If all has gone well you should end up with a pretty meaningless pair of histograms: there will be a single bar at the left in each one. That's because there are a few projects in the dataset with much larger capacities or costs than normal for residential projects. Now we'll remove the projects above the 99th percentile in size or cost. Start by setting `kw99` equal to the result of calling `.quantile(0.99)` on the `"nameplate"` column of `res`. That will pick out the nameplate capacity at the 99th percentile. Then use a similar process to set `tc99` to 99th percentile of the `"total_cost"` column of `res`.
22. Print an informative message giving `kw99` and `tc99`.
23. Create a new dataframe called `trim` by calling the `.query()` method on `res` with the argument `f"nameplate <= {kw99} and total_cost <= {tc99}"`.

24. Print the new number of records in `trim` to make sure a reasonable number (between 1 and 2 percent) were removed.

25. Now repeat the steps used to construct the two-panel figure above but using dataframe `trim` instead of `res`. If all goes well you should see two much nicer histograms.

26. Save the figure as `"res_nameplate_cost.png"`.

27. Now we'll use seaborn to do some comparisons of projects with different values of the `"third_party"` variable. Use `var` to loop over a list consisting of the column names `"nameplate"` and `"total_cost"`. Within the loop do the following:

    1. Create a new figure by setting `fig, ax1` equal to the result of calling `plt.subplots()`. Please note: use the same statement for the rest of the semester whenever the instructions say to *create a new single-panel figure* and don't say explicitly how to do it. When doing so, include the `dpi=300` argument if the script doesn't set the default resolution using `plt.rcParams`.

    2. Call `sns.histplot()` using the following arguments: `data=trim`, `x=var`, hue=`"third_party"`, `kde=True`, and `ax=ax1`. The `hue` argument will cause superimposed histograms to be drawn for different values of `"third_party"`, and the `kde` argument will cause a kernel density estimate of the distribution to be added to the plot.

    3. Tighten the figure's layout and then save it using the name `f"res_{var}.png"`.

28. Another way to look at the distributions is to use boxen plots, which are a much-enhanced version of box plots. Create a new single-panel figure and then call `sns.boxenplot()` with the following arguments: `data=trim`, `x="third_party"`, `y="nameplate"`, and `ax=ax1`.

29. Then set the title for `ax1` to `"Nameplate Capacity"`, the label for the X axis to `"Third Party"`, and the label for the Y axis to `"kW"`. (FAQ4)

30. Tighten the figure's layout and then save it as `"res_boxen_all.png"`.

31. Yet another approach is to use violin plots, which show kernel density estimates. Create a new single-panel figure and then call `sns.violinplot()` with the following arguments: `data=trim`, `x="nameplate"`, `y="inst_status"`, `hue="third_party"`, `split=True`, and `ax=ax1`. Notice that the top and bottom distributions differ since the call requests a split violin plot. Also, if `"inst_status"` took on multiple values there would be a separate horizontal spine for each one.

32. Set the title to `"Nameplate Capacity"`, the X label to `"kW"`, the Y label to `""` (an empty string), tighten the layout, and then save the figure as `"res_violin.png"`.

33. Another option is to overlay the density estimates in a single figure. To do that, create a new single-panel figure and then call `sns.kdeplot()` with the following arguments: `data=trim`, `x="nameplate"`, `hue="third_party"`, `palette="crest"`, `fill=True`, and `ax=ax1`. The `palette` keyword picks out the color palette that will be used to distinguish between the curves, and `fill=True` says that the areas under the curves should be filled in. Then set the title to `"Nameplate Capacity"`, the X label to `"kW"`, tighten the layout, and save the figure as `"res_kde.png"`

34. Now we'll look in more detail at projects by year. First, trim off the last few years when the program was essentially over by setting `main` equal to the result of calling `.query()` on `trim` with the argument `"year <= 2016"`.

35. Create a new single-panel figure and then call `sns.boxenplot()` with the following arguments: `data=main`, `y="year"`, `x="nameplate"`, `orient="h"`, and `ax=ax1`. The `orient` keyword causes

the boxes to be drawn horizontally. It's worth noting that swapping the X and Y variables will *not* change the orientation of a boxen plot: you *must* use the `orient` keyword.

36. Set the title for `ax1` to `"Nameplate Capacity by Year"`, the label for the X axis to `"kW"`, and the label for the Y axis to `"Year"`.
37. Tighten the figure's layout and then save it as `"res_boxen_year.png"`.
38. Finally, we'll show the joint distribution of `"nameplate"` and `"total_cost"` using a hex plot. A hex plot is essentially an enhanced scatter plot for large datasets: it shows the density of points using colors that vary in intensity. The function we'll use produces a high-level seaborn JointGrid graphics object and does *not* require that `plt.subplots()` be called first. To create the plot, set `jg` equal to the result of calling `sns.jointplot()` with the following arguments: `data=trim`, `x="nameplate"`, `y="total_cost"`, and `kind="hex"`.
39. Set the labels of the X and Y axes by calling the `.set_axis_labels()` method of `jg` with the arguments `"Nameplate"` and `"Total Cost"`. Note that this differs from the way labels are set on individual Axes objects.
40. Now set the overall title by calling `jg.figure.suptitle()` with the argument `"Distribution of Systems by Cost and`
As you can probably tell, `jg.figure` provides access to the Matplotlib Figure object embedded in seaborn's JointGrid object.
41. Then call `jg.figure.tight_layout()` to tidy up the layout.
42. Finally, use the `.savefig()` method of `jg` with argument `"res_hexbin.png"` to save the figure.

Submitting

Once you're happy with everything and have committed all of the changes to your local repository, please push the changes to GitHub. At that point, you're done: you have submitted your answer.

Notes

- Constructing figures can be confusing because of several kinds of objects are involved. In Matplotlib itself, the workhorse objects are Axes and Figures. Seaborn adds additional complication because some of its functions return Axes, some return Figures, and some return higher-level objects (FacetGrid, JointGrid, and PairGrid objects). The Axes and Figure objects can be tweaked using standard Matplotlib calls but the three seaborn grid objects are more complex and have their own sets of methods.

FAQ

1. *How do I change the datatype of a column?*

   Use the `.astype()` method on the column and store the result back in the dataframe under the original name. For example, to change the datatype of `D["c"]` to `int`, use `D["c"] = D["c"].astype(int)`.

2. *How do I drop records that are missing data in specific columns?*

   To drop records from dataframe `D` that are missing data for any of the columns in list `L` use `D = D.dropna(subset=L)`

3. *How do I tighten the figure's layout?*

   To tighten the layout of figure `F` use `F.tight_layout()`. Keep in mind that this method applies to figures, not axes.

4. *How do I add titles and labels to a Matplotlib axes object?*

   To set the title of axes object `A` to `"words"` use `A.set_title("words")`. To set the X label to `"words"` use `A.set_xlabel("words")`. Set the Y label in an analogous way.