Lab 1: Stats and Python Refresher

Learning Objectives

In this lab, we'll be reviewing some Python basics, but the emphasis will be on loading data, inspecting variables, coding missing data, and some basic data descriptives. You will need many of these basic skills for your main regression assignment. The skills covered today include how to:

- Load the data
- Inspect & clean data: detecting missing values, ensuring the data are the right "type" (integer vs. string vs. float)
- Calculate descriptive statistics for your data
- Conduct t-tests and proportions tests for the entire sample and for subsamples

Preparing your data is the most time-consuming part of any attempt to analyze data, but also the most important. Familiarizing yourself with the data, its attributes, and quirks, is fundamental to understanding what the data can tell you and what it cannot. Some experienced users try to skip the initial steps of becoming familiar with the data, only to regret it later when results are unexpected.

Data

For this first lab and for the next two labs, we'll be using a database on property values collected for properties in the Bogotá metropolitan area between 2001 and 2006. The data come from a web portal in which real estate agents in Bogotá list properties for sale and rent (www.metrocuadrado.com). We focus exclusively on all residential properties listed in the database: single-family (attached and detached) and units in multi-family apartments. The dataset includes information about the structure and the situation of each property. Structural characteristics of each property include the age, type of property, floor area, number of rooms, number of garages and number of bathrooms. By situation we mean the location of the property relative to surrounding places, local and regional. Recognizing the importance of neighborhood characteristics in influencing property values, the dataset also includes indicators of neighborhood-level socio economic status, density, and land uses.

This dataset has a fair number of observations (> 3,000) and is quite clean. Before we move on, what does it mean for data to be clean?

- No stray/erroneous responses
- No typos or stray punctuation
- Variables all have concise, distinct names
- Etc… Can you list any other criteria?

Before (and after!) loading your data, it is useful to be able to go between the different places to get information about your data. Take a look at the data dictionary: dictionaries can give you a broad idea of what the dataset includes (depending on size). It can also be really helpful to look at the underlying questions to get a better idea of what each variable represents, and to determine the cause of any discrepancies among responses.

Load your Data:

First, open the Lab 1 link from the Modules tab in bCourses. Next, navigate to the Files tab in bCourses, and download the "Data for Lab 1" folder. As discussed, we'll take a few minutes to open the .xlsx file to get familiar with the data dictionary.

Now let's go back to our Datahub page. Create a folder for your data (to keep up good organizational practices), and upload the .csv version of the dataset. Next, we can open our Lab 1 Notebook. It's blank except for code to import a few libraries!! We'll be writing this code together from scratch. Make sure to save your notebook as we go so you have a record of what we do today. An answer key with full code will also be posted to bCourses after lab.

To connect to our data, let's use the pandas "`.read_csv`" command on `'Data for Lab 1/Property Data.csv'`, and save this connection to a data frame variable. The "`.head`" function lets us view just the first five rows, or another number of your choosing - "`.head(#)`". We can't see all of our variables (there's a little '…' in the middle), but that's ok. It's often easier to create smaller sub-dataframes with only variables of interest, which is what we'll do here. However if you prefer, you can uncomment the "`display.max_`" code up in the libraries cell to see all columns or rows within your original data frame.

Now let's start looking at the different kinds of variables in this dataset! We don't have too many variables, so we can call "`df.dtypes`" to see all of our data. We can also check for missing values using the "`.isnull().sum()`" command.

Investigate a Continuous Numeric Variable

As discussed in class, quantitative data at a basic level is numbers. The number of walks taken in a week, yearly income, percent of the population who can wiggle their ears, etc…

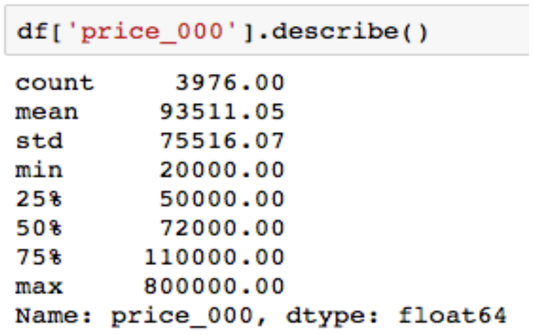
Numeric variables can be broken down into discrete and continuous variables:

- A discrete numeric variable consists of integers, (ie. number of children in a household).
    - Theoretically, statistical analysis could give you an answer of "2.5 children" to a question about how many children the average American household has, but we would read that as "between 2 and 3 children," and not literally 2 ½ children.
- A continuous numeric variable is one that can consist of numbers that are not integers (ie. annual income, which may be rounded to integers in the ACS, but could be analyzed to an infinite number of decimal places).

Here we will investigate property prices. What type of numeric variable is this?

Get summary statistics for a continuous variable

Since we're interested in the asking price of a property, we want to look at the variable "price_000". Using the ".describe()" command, let's see how the variable is stored in the file:



```
df['price_000'].describe()
```

```
count      3976.00
mean      93511.05
std       75516.07
min       20000.00
25%       50000.00
50%       72000.00
75%      110000.00
max      800000.00
Name: price_000, dtype: float64
```

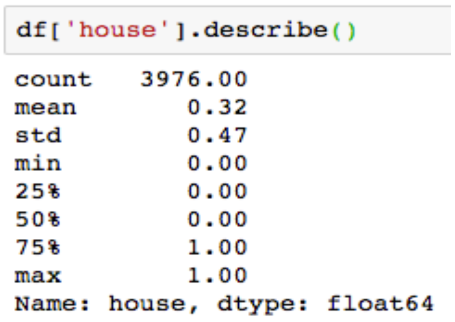
The Python output shows us that price_000 is stored as a "float" which means it's a numeric variable with decimals. This is different than the df.dtypes answer, where it showed as an integer ("int"). What's up with that? The describe function is calculating the mean by dividing the sum by the number of observations – and division always yields a float output (with decimals) in Python. If we want to see these statistics for all our variables, we can use "df.describe()".

To better visualize the distribution of a continuous variable, we can utilize a histogram (which we'll cover in lab next week). You can also ask for specific information like the median and the interquartile range by using the .median() and iqr() functions respectively. The interquartile range is the difference between the value of the variable at the 75th percentile and at the 25th percentile. It's a measure of variation.

We're ok with property price being stored as a float, however if needed we could convert variable types with ".astype(int)" – or (str) or (float). This dataset today has no missing values (phew!).
More on Data Types: https://www.datacamp.com/community/tutorials/data-structures-python
More on Missing Data: https://www.geeksforgeeks.org/working-with-missing-data-in-pandas/

Investigate a Dummy Variable

A dummy or binary variable takes only two unique values. Typically the values are 0 or 1, but this is just a convention. As with the continuous variable, go ahead and look into the summary statistics for the dummy variable house (=1 if it's a single family house, 0 otherwise). How would you know that it is a dummy variable?

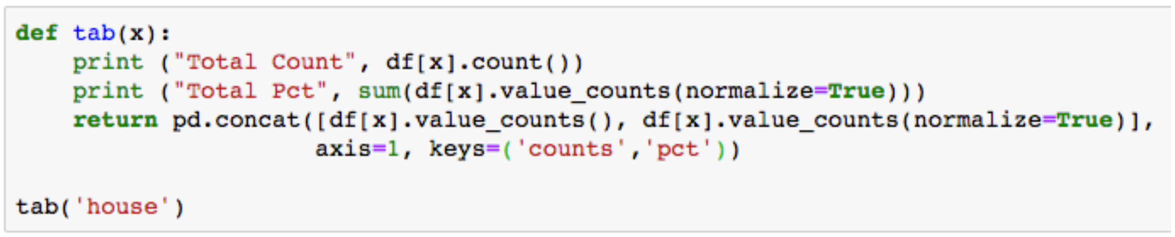


```
df['house'].describe()
```

```
count    3976.00
mean        0.32
std         0.47
min         0.00
25%         0.00
50%         0.00
75%         1.00
max         1.00
Name: house, dtype: float64
```

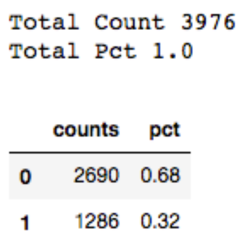
Unless you look at the codebook or the instrument used to collect the data, it wouldn't be immediately obvious from the output that this was a dummy variable. Using ".value_count()" can give you a better look, and can even show percentages if you specify "normalize=True" within the function parentheses.

We can also create a function to view the count and percentage outputs (and any other outputs we think are necessary) together.



```
def tab(x):
    print ("Total Count", df[x].count())
    print ("Total Pct", sum(df[x].value_counts(normalize=True)))
    return pd.concat([df[x].value_counts(), df[x].value_counts(normalize=True)],
                     axis=1, keys=('counts','pct'))

tab('house')
```

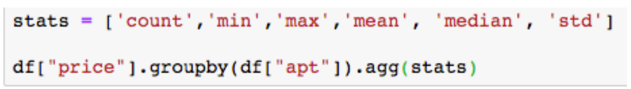


```
Total Count 3976
Total Pct 1.0
```

	counts	pct
0	2690	0.68
1	1286	0.32

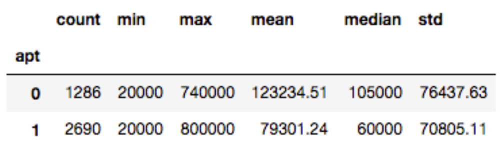
This provides a frequency count for all the distinct values of the variable house. In this case, there are two values: 0 or 1. Note: what do you think would happen if you ran the value counts command on the property price variable with "df['price_000'].value_counts"?

One last helpful tool to look into our data before we run statistical tests is the ".groupby()" command. We can group one variable (or more depending on how we define it) by another, and aggregate summary statistics using ".agg()".



```
stats = ['count','min','max','mean', 'median', 'std']

df["price"].groupby(df["apt"]).agg(stats)
```



	count	min	max	mean	median	std
apt						
0	1286	20000	740000	123234.51	105000	76437.63
1	2690	20000	800000	79301.24	60000	70805.11

Testing for Significant Differences

The code for statistical tests is straightforward, however it's important to take a step back and ask yourself why you're running these tests in the first place so you don't lose sight of the meaning behind your results. After running each test, take turns summarizing your results to a neighbor.

Comparing means

For example, what if someone is interested in testing whether the asking price of apartments is more or less than houses?

a) Test if apartments have different prices than houses (unpaired ttest of price by group)

Or if houses, rather than apartments, are more likely to have green spaces in their neighborhoods?

b) Test if houses are more likely to have more or less public green spaces in their neighborhood (pcn_green) than apartments

Compare means for a subset of the data

c) Test if, for apartments, prices vary for newer places (age_0_10 = 1) vs. older places

Comparing proportions

d) Test if houses (y/n) are more likely to be older (age_20_more) than not

Remember that code is amazing (amazing!) because you can load the raw data file and run your code again to get right back to work without ever changing the original data file. When we get to cleaning survey data, the goal will be to have a raw data file, a cleaned data file, and your (thoroughly) annotated python code so that someone unfamiliar with your data could follow your process and run your code without error.