Python: Data Visualization Notes

Klein
carlj.klein@gmail.com

1 Learning Objectives

What good is a data analysis if the answers or findings can't be conveyed properly? Or perhaps even more detrimental, what good is a dataset if the proper questions or hypotheses can't be formed in the first place? How can we properly tell the story the data is providing? Enter in data visualization! Both explaining the data and exploring the data can be significantly helped through the process of data visualization.

Data visualization is two-pronged:

- Exploratory Analysis: Helping to understand the data prior to analysis. Searching for relationships and insights.
- Explanatory Analysis: Helping to present analysis findings, helping to tell a story with the data. After insights were found.

And it's all a part of the entire data analysis process. We can also simplify the data analysis process into 5 steps:

1. Extract
2. Clean: Exploratory
3. Explore: Exploratory
4. Analyze: Exploratory OR Explanatory
5. Share: Explanatory

In this course, we'll be using the matplotlib, seaborn, and Pandas libraries to assist in the data analysis process.

Concepts

- Design of Visualization
- Exploration of Data

  - Univariate Exploration of Data
  - Bivariate Exploration of Data
  - Multivariate Exploration of Data

- Explanatory Visualizations
- Visualization Case Study

2 Design of Visualization

To begin our discussion of visualization, we need to cover some basic vocabulary and distinctions.
Data can be broken into two main categories, each of which can be broken down further:

- Qualitative / Categorical:
  - Nominal: No order
  - Ordinal: Intrinsic Order
- Quantitative / Numerical:
  - Interval: Absolute differences are meaningful (addition and subtraction follows logic)
  - Ratio: relative differences are meaningful (multiplication and division follows logic)

It should be noted that the quantitative data type can be also be broken down into discrete and continuous variables.

What about those 3-dimensions charts or fun backgrounds that we used to add to our science experiment plots as kids? That used to add some fun to our projects, right? While fun for the youth, it turns out there is an empirical rule when figuring out how much the additional "junk" either adds or detracts from conveying the data.

- Data-Ink Ratio = data-ink / total ink used to print the graphic. The higher the Data-Ink Ratio, the better conveyed data is.

Can visualizations be purposefully misleading, even when using the data appropriately? Absolutely!

A great example of this is trying to over-inflate the difference or change between data points during different time periods. Say a presenter is trying to make a claim there was a very large change from one year to the next. We'll say in year

1 the y-value was 100, and in year 2 the y-value was 105. The presenter changes the window of the graph to display from a y-value of 99 to a y-value of 106, and the x-values are only the two years. Obviously, this is going to look like a massive change! In reality, had the reporter shown the data at a true scale, the visual shows in actuality that the change isn't so tremendous.

This concept also has an empirical rule:

- Lie Factor = size of effect shown in graphic / size of effect shown in data

  = (change in visual / visual start) / (change in data / data start)

  As was said in the example, this can be used to purposefully distort data. In fact, a Lie Factor 1 suggests a misleading visual, and even greater than that suggest an even greater disparity from the truth.

Away from the empirical side of visuals, and more into the logical, we come across the common mistake of using too many colors! Colors can be useful when separating categories, however, they it's very easy to cause redundancy with them. Here are some tips when using color:

- Get it right in black and white (and shades of grey)
- Use less intense colors such as natural or pastel, and higher grey colors. The eye can actually concentrate longer under these conditions.
- Color facilitates communication. Use color to separate the data into groups of interest, not just to color a visual.
- Design for Color Blindness. Stay away from red / green pallets, and use blue / orange pallets.

Don't want to overdo it on the color schemes? Don't forget about other visual queues such as shape and size.

Some tips on shape, size & other tools:

- Use different types of encodings, rather than using color (square / dot vs. colors to separate groups of interest).
- Color and shape are good for categorical variables.
- Size of marker can assist in adding additional quantitative data.

3 Exploration of Data

We have a dataset on a topic or concept that has been deemed worthy for inspection! Surely, there are some insights to be gained from it. We load up the data, and then we hit a wall... Which columns are important? What questions can be answered? We can find the general statistics of the set, so what?

This section will help with the initial process of data exploration, helping to find what is actually useful and should be examined further in the data. We'll start with single variables from the data and move into visually pairing multiple data points at once.

We'll be using a few different Python libraries in this section:

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

3.1 Univariate Exploration of Data

The first tools added into our data exploration toolkit will be for univariate data, or examining the data a single variable at a time. Even though we're going to be using several different libraries, we'll still try to keep track of new and essential commands and terminology:

Commands:

- df.head: Retrieves the first few rows of data from a Pandas DataFrame. Able to specify how many rows
- sb.countplot: Seaborn's command for generating a bar chart.
- df[column].value_counts().index: Will return an immutable sequence sorted number of categorical entries (highest to lowest).
- df.melt: Will unpivot a DataFrame from wide to long format (i.e. we can "melt" two entries together).
- plt.hist: Matplotlib's command for generating a histogram.
- sb.distplot: Seaborn's command for generating a histogram. Default command also includes a density curve estimation.

Terminology:

- bar chart: Useful to show counts across categories.

- relative frequency: Shows proportion of each category in population.
- pie chart: Use these to show how whole of data is broken down into parts, useful when plotting a small number of slices (usually no more than four).
- histogram: Useful to examine quantitative data.

We'll be using a Pokemon dataset for our examples. Let's get an idea of what our data looks like:

```
pokemon = pd.read_csv("pokemon.csv")
pokemon.shape
# pokemon.shape = (807, 14)
pokemon.head(10)
```

Pokemon.Head(10)

	id	species	generation_id	height	weight	base_experience	type_1
0	1	bulbasaur	1	0.7	6.9	64	grass
1	2	ivysaur	1	1.0	13.0	142	grass
2	3	venusaur	1	2.0	100.0	236	grass
3	4	charmander	1	0.6	8.5	62	fire
4	5	charmeleon	1	1.1	19.0	142	fire
5	6	charizard	1	1.7	90.5	240	fire
6	7	squirtle	1	0.5	9.0	63	water
7	8	wartortle	1	1.0	22.5	142	water
8	9	blastoise	1	1.6	85.5	239	water
9	10	caterpie	1	0.3	2.9	39	bug

	type_2	hp	attack	defense	speed	special-attack	special-defense
0	poison	45	49	49	45	65	65
1	poison	60	62	63	60	80	80
2	poison	80	82	83	80	100	100
3	NaN	39	52	43	65	60	50
4	NaN	58	64	58	80	80	65
5	flying	78	84	78	100	109	85
6	NaN	44	48	65	43	50	64
7	NaN	59	63	80	58	65	80
8	NaN	79	83	100	78	85	105
9	NaN	45	30	35	45	20	20

Now that we've had a peak at the data, let's get into some visualization.

```
# create a bar chart for one of the columns (generation_id)
# this will show counts of each pokemon for each generation
sb.countplot(data = pokemon, x = 'generation_id')
# See Figure 1

# let's use a single neutral color to reduce redundancy and make
                                it easier on the eyes
base_color = sb.color_palette()[0]
sb.countplot(data = pokemon, x = 'generation_id', color =
                                base_color)
# See Figure 2
```

A simple, yet effective visual trick is adding some order to your plots via sorting.

```
# let's sort the data highest -> lowest
# we can use pandas series function: value_counts()
gen_order = pokemon['generation_id'].value_counts().index
sb.countplot(data = pokemon, x = 'generation_id', color =
                                base_color, order = gen_order)
# See Figure 3
```

Let's combine the previous coloring and sorting to look at a new variable. Additionally, since we know the x-axis labels are somewhat long. Two favorable ways

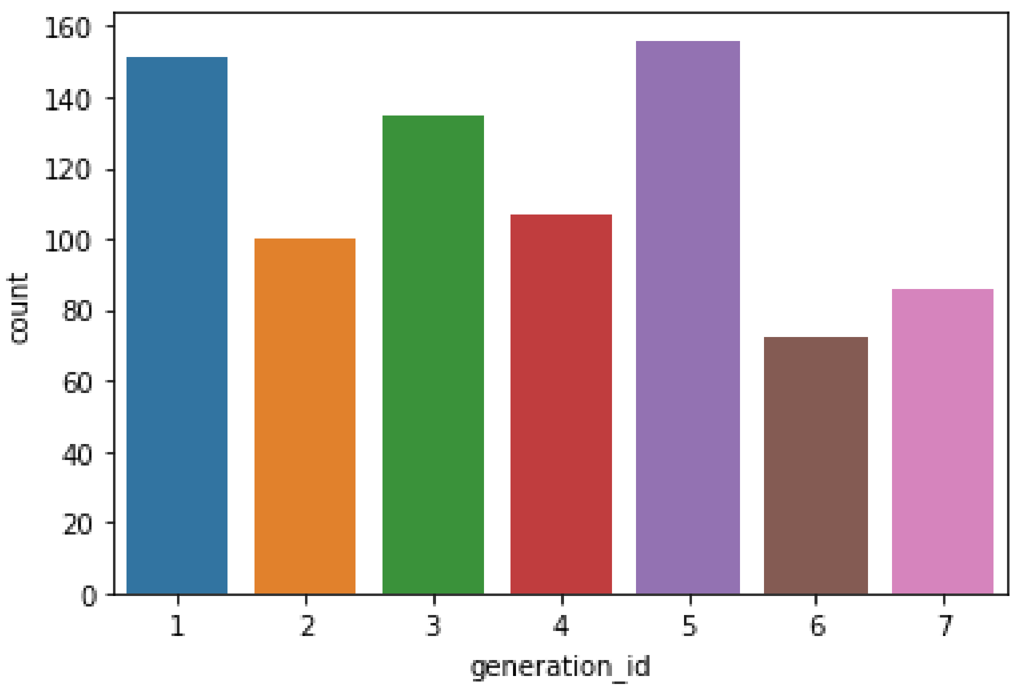

Figure 1: Generic bar chart.

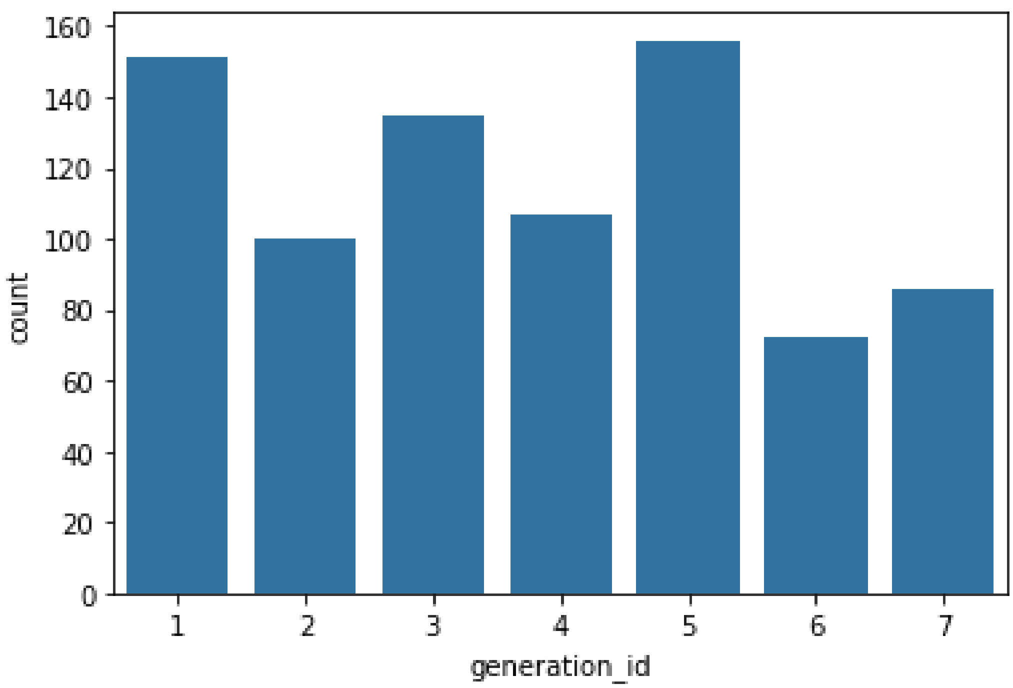

Figure 2: Bar chart with neutral coloring.

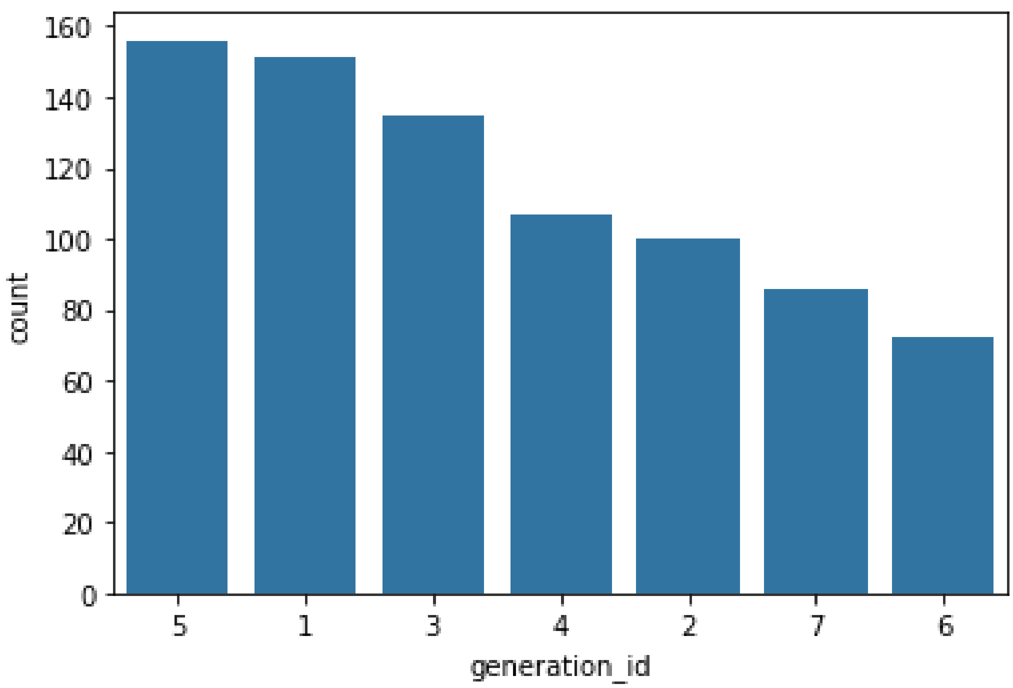

Figure 3: Bar chart with neutral coloring and ordering applied.

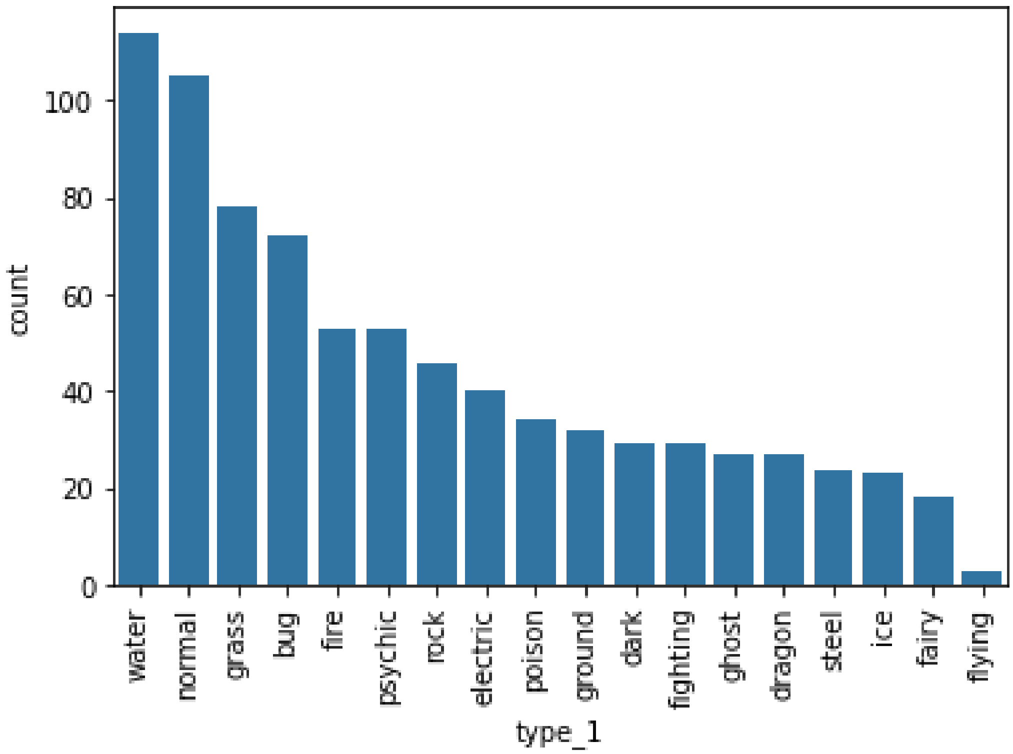

Figure 4: Standard bar chart format with x-axis labels rotated.

of dealing with this is to either rotate the x-axis labels, or create a horizontal bar chart.

```
type_order = pokemon['type_1'].value_counts().index
sb.countplot(data = pokemon, x = 'type_1', color = base_color,
                                order = type_order)
plt.xticks(rotation = 90)
# See Figure 4

# Change x to y to change from vertical to horizontal bar chart
sb.countplot(data = pokemon, y = 'type_1', color = base_color,
                                order = type_order)
# See Figure 5
```

The previous examples display actual counts in the bar charts, but it can also be useful to display data using relative frequency (proportion of each category in the population). This gives us an opportunity to introduce a few other concepts as well. Most importantly, we'll introduce the melt function, which we'll use to combine two categories (two distinct columns) into a single category.

```
pkmn_types = pokemon.melt(id_vars=['id', 'species'],
```

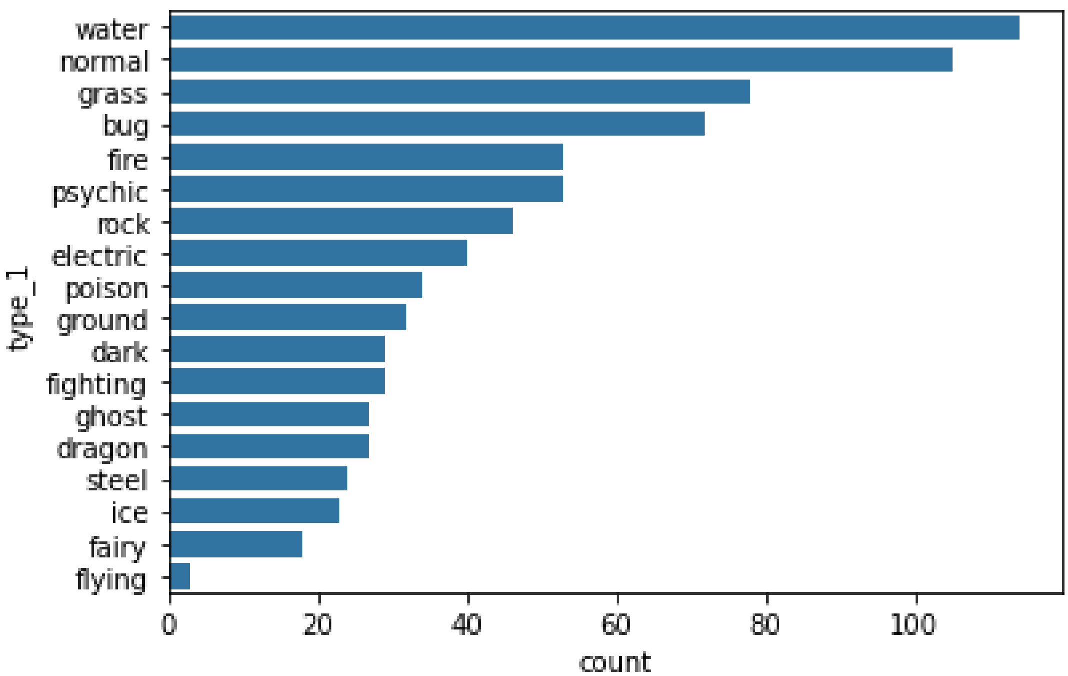

Figure 5: Standard bar chart format in vertical form.

```
        value_vars=['type_1', 'type_2'],
        var_name='type_level',
        value_name='type')
"""
Let's walk through how the melt function is being used here:
- id_vars: columns to utilize as identifier variables
- value_vars: columns to unpivot
- var_name: name of column to describe data being unpivoted
                                together
- value_name: name of column to match values from the unpivoted
                                labels

Note: the pokemon DataFrame has 807 rows (pokemon entries). This
                                use of melt puts two categories
                                together into a single category.
                                Thus, each pokemon now has two
                                entries each, making the
                                pkmn_types DataFrame have double
                                the rows as the original
                                DataFrame or 1614 rows.
"""

base_color = sb.color_palette()[0]
type_counts = pkmn_types['type'].value_counts()
type_order = type_counts.index
sb.countplot(data = pkmn_types, y = 'type', color = base_color,
                                order = type_order)
# See Figure 6

# now we'll change the tick counts
n_pokemon = pokemon.shape[0]
max_type_count = type_counts[0]
max_prop = max_type_count / n_pokemon
# note: max_prop = 0.16

tick_props = np.arange(0, max_prop, 0.02)
tick_names = ['{:0.2f}'.format(v) for v in tick_props]
"""
tick_props creates the numbers for an equally spaced axis, while
                                tick_names takes the numeric
                                version and returns a list of a
                                string values.
"""

# The culmination of the above work into a chart:
sb.countplot(data = pkmn_types, y = 'type', color = base_color,
                                order = type_order)
plt.xticks(tick_props * n_pokemon, tick_names)
plt.xlabel('proportion')
for j in range(type_counts.shape[0]):
count = type_counts[j]
pct_string = '{:0.1f}%'.format(100*count/n_pokemon)
plt.text(count + j, j, pct_string, va = 'center')
# See Figure 7
```

Whereas bar charts are a great initial step in exploring categorical data, histogram are useful in the exploration of quantitative data. Let's take a look at

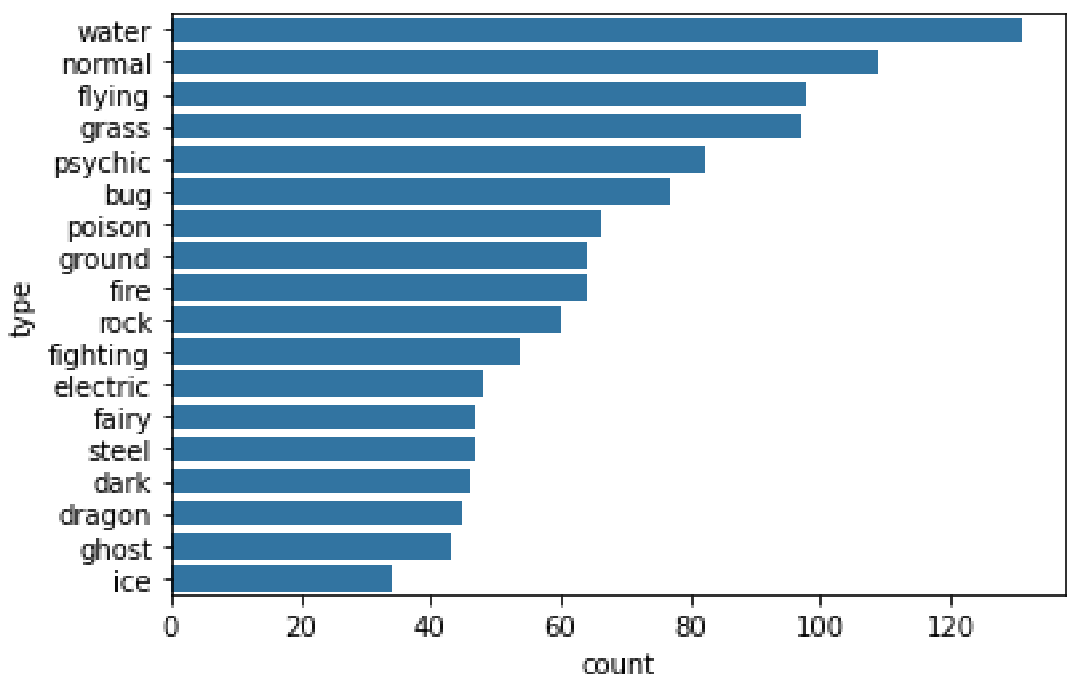

Figure 6: Bar chart of melted pokemon types.

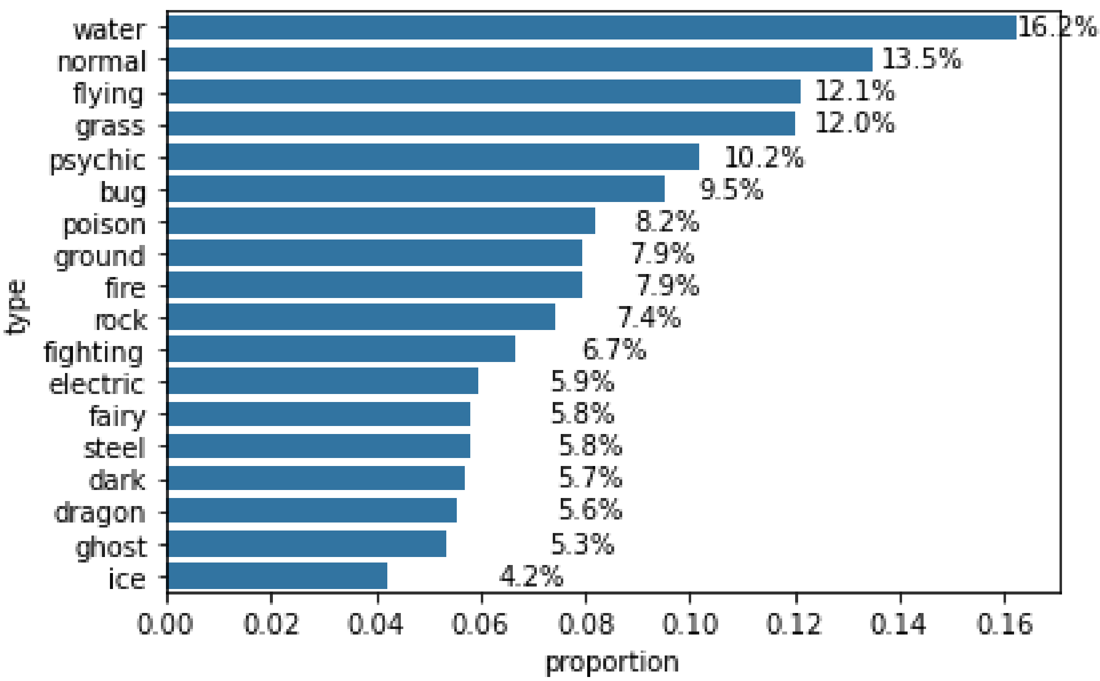

Figure 7: Bar chart of melted pokemon types shown in relative frequency.

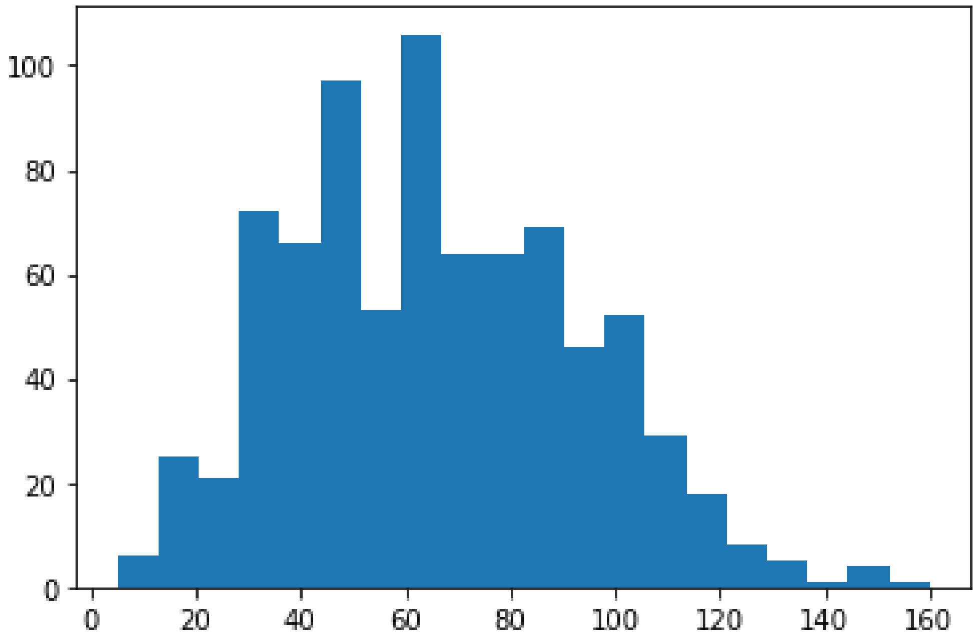

Figure 8: Basic histogram of Pokemon speeds.

our variables in the pokemon DataFrame which contain quantitative data.

```
# basic histogram exploring the speed variable
plt.hist(data = pokemon, x = 'speed', bins = 20)
# See Figure 8

# for historgrams, bin specification is paramount in
                                visualization
# this is an example
bins = np.arange(0, pokemon['speed'].max() + 5, 5)
# recall that arange will not include last value
plt.hist(data = pokemon, x = 'speed', bins = bins)
# See Figure 9

# Seaborn's histogram command has a default density curve
                                estimation
sb.distplot(pokemon['speed'])
# See Figure 10
```

Earlier in the course, we mentioned avoiding a high "lie factor". One case in which it may actually be beneficial to "zoom" in on data is when dealing with outliers, or when a large majority of the data is within certain axis limits.

For example, let's take a look at the height variable of Pokemon.

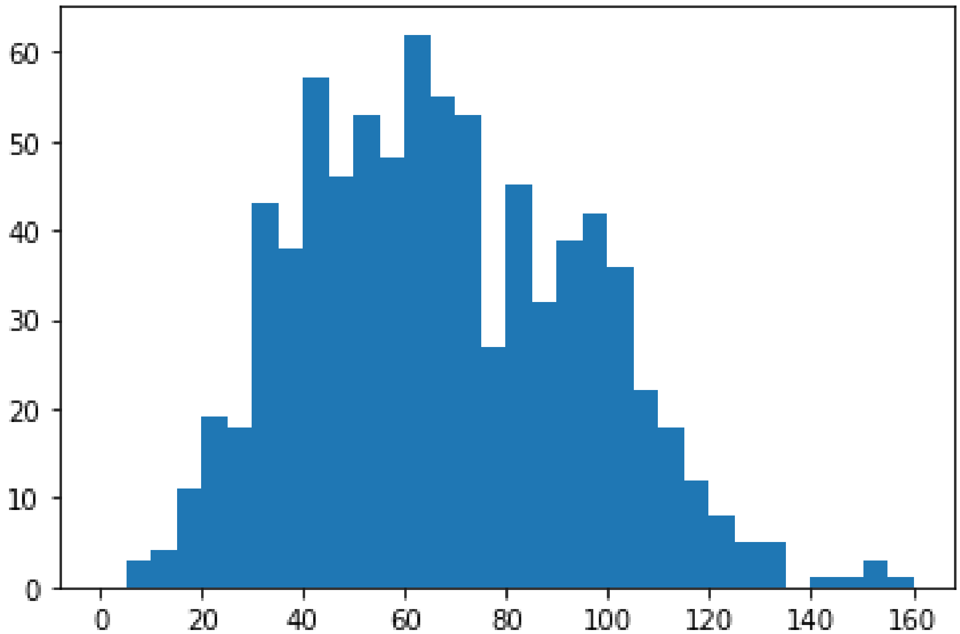

Figure 9: Histogram of Pokemon speeds with altered bins.

```
# even though this is a course on visuals, basic math and
                                descriptive statistics can still
                                be beneficial
pokemon['height'].describe()
"""
count    807.000000
mean       1.162454
std        1.081030
min        0.100000
25%        0.600000
50%        1.000000
75%        1.500000
max       14.500000

With the majority of our data between 0 and 1.5, it's acceptable
                                to change our x-axis limits. The
                                outlier of 14.5 would make our
                                histogram and bin choice less
                                effective from a visual sense.
"""
bins = np.arange(0, pokemon['height'].max() + 0.2, 0.2)
plt.hist(data = pokemon, x = 'height', bins = bins)
plt.xlim((0, 6))
# See Figure 11
```

Another method when dealing with outliers or data points that have very large

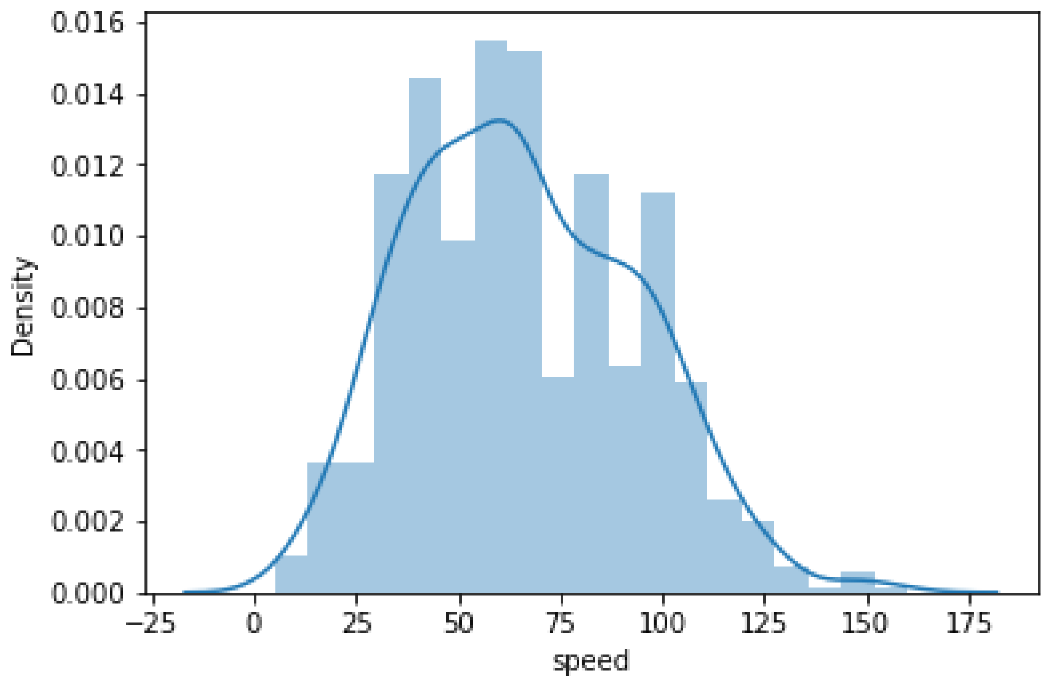

Figure 10: Seaborn's default histogram.

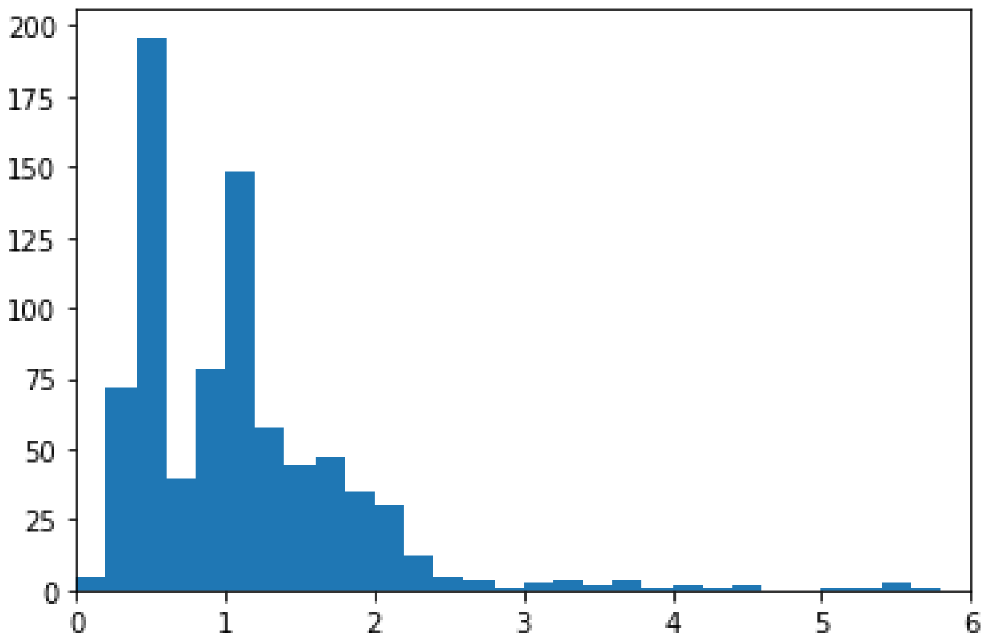

Figure 11: Histogram with altered x-axis limits.

differences between them is with scaling and transformations.

Let's take a look at the weight variable of Pokemon.

```
pokemon['weight'].describe()
"""
count     807.000000
mean       61.771128
std       111.519355
min         0.100000
25%         9.000000
50%        27.000000
75%        63.000000
max       999.900000

Obviously, there exists some outlier(s) here. We should think
                                about using transformations and
                                tick-mark manipulation to
                                visualize the data this time
                                around.
"""

# Let's see what we're dealing with without any scaling or
                                transformations
bins = np.arange(0, pokemon['weight'].max() + 40, 40)
plt.hist(data = pokemon, x = 'weight', bins = bins)
```

```
# See Figure 12
"""
Even with specifying the bins, the histrogram proves to not be
                                too insightful. Changing the bins
                                 alone likey won't solve this
                                issue.
"""

# Let's try applying a transformation of the x-scale, itself
bins = np.arange(0, pokemon['weight'].max() + 40, 40)
plt.hist(data = pokemon, x = 'weight', bins = bins)
plt.xscale('log')
# See Figure 13
"""
It could be argued this is a worse approach, but using a
                                logarithmic transformation could
                                steer us in the right direction.
"""

np.log10(pokemon['weight'].describe())
"""
count     2.906874
mean      1.790786
std       2.047350
min      -1.000000
25%       0.954243
50%       1.431364
75%       1.799341
max       2.999957

Applying a logarithmic transformation on the data, itself,
                                appears to shrinks the numbers
                                into a more manageable scale.
"""

# Get the ticks for bins between [0 - maximum weight]
bins = 10 ** np.arange(-1, 3 + 0.1, 0.1)

# Generate the x-ticks we want to apply
ticks = [0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000]
"""
Important: note here how we are using differently spaced tick
                                marks in the original scale.
"""

# Convert ticks into string values, to be displayed along the x-
                                axis
labels = ['{}'.format(v) for v in ticks]

# Plot the histogram
plt.hist(data=pokemon, x='weight', bins=bins);

# The argument in the xscale() represents the axis scale type to
                                apply.
# The possible values are: {"linear", "log", "symlog", "logit",
                                ...}
plt.xscale('log')
```

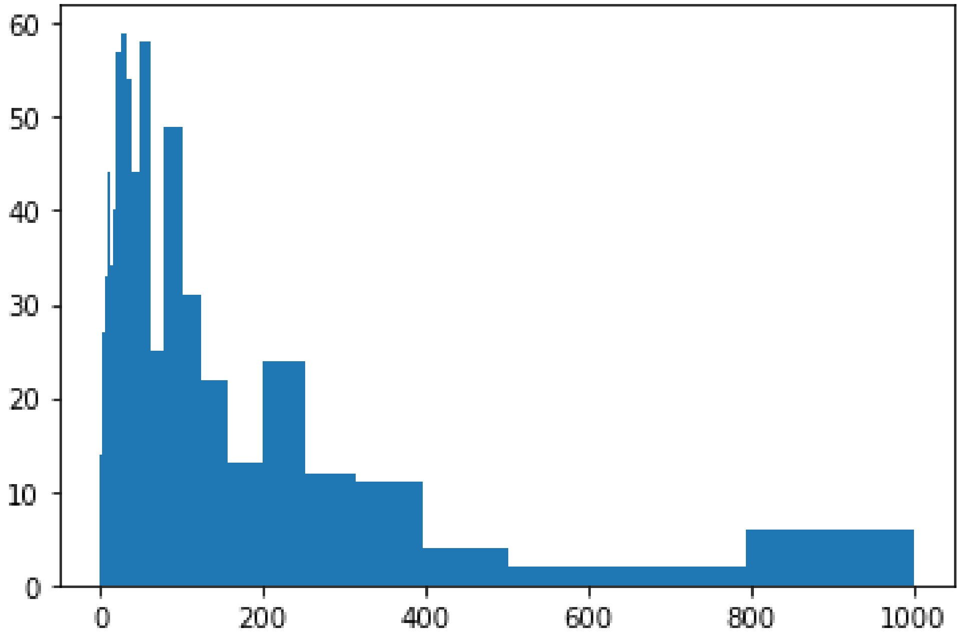

Figure 12: First try histogram of Pokemon's weight.

```
# Apply x-ticks
plt.xticks(ticks, labels)

# See Figure 14
```

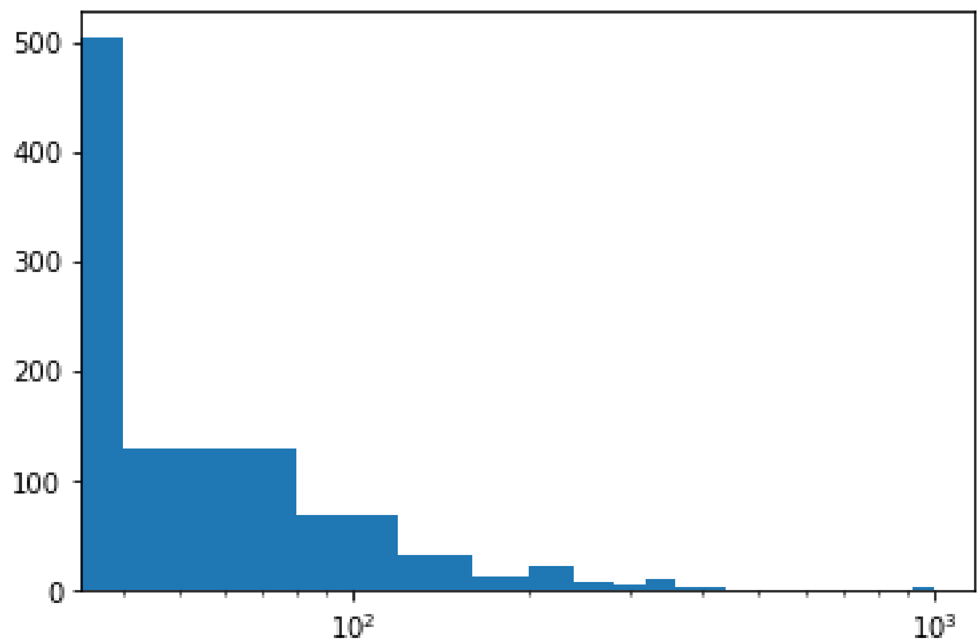

Figure 13: Histogram of Pokemon's weight with a log transformation applied to the x-scale.

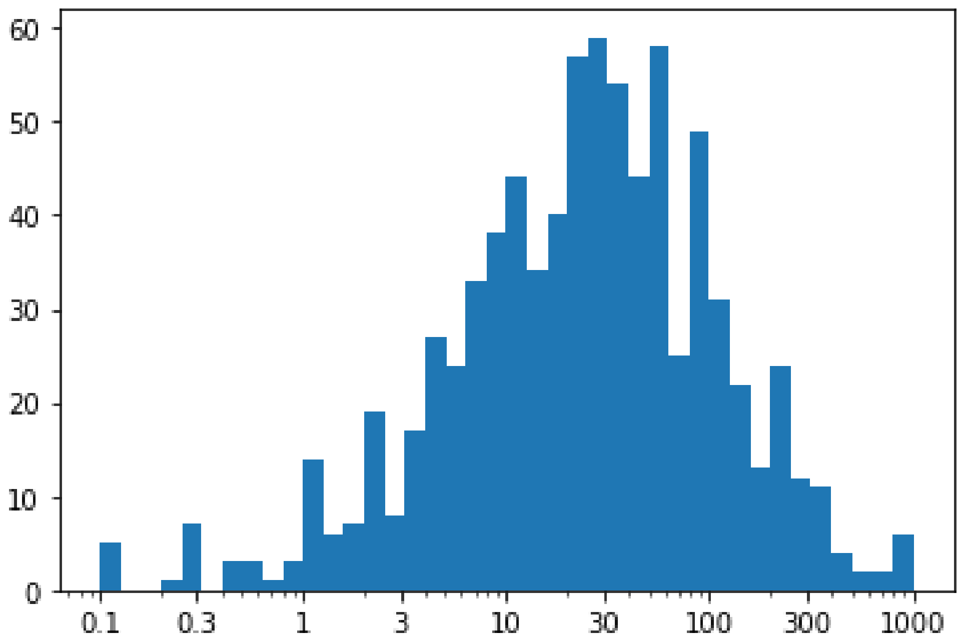

Figure 14: Histogram of Pokemon's weight with transformation and scaling applied.

3.2 Bivariate Exploration of Data

Having seen some basic ways to visually examine univariate data, we can move into exploring a data set by comparing two variables at a time with bivariate exploration of data.

Commands:

- plt.scatter: Matplotlib's command for a scatter plot.
- sb.regplot: Seaborn's command for a scatter plot. Much like sb's histogram default, their plot includes a line of best fit.

Terminology:

- heat maps:
- scatter plots: Use for quantitative vs. quantitative variables.
- violin plots: Use for quantitative vs. qualitative variables.
- box plots:
- clustered bar charts: Use for qualitative vs. qualitative variables.
- faceting:
- line plots:

Although univariate data seems simple by virtue of a single variable, due to practicality, most people are likely more familiar with the first example shown in this section, a scatter plot. This is the classic x vs. y, plot your data using coordinates plot. A very effective tool, and easy to implement when the data consists of a single dependent variable. However, it's not the only tool! Datasets of different complexity and data types require different approaches to understand their stories. Let's dive in with an example.

```
# first things first, let's check out our dataset
fuel_econ = pd.read_csv('fuel-econ.csv')
fuel_econ.shape
# (3929, 20)
fuel_econ.head(10)
```

fuel_econ.head(10)

	id	make	model	year	VClass
0	32204	Nissan	GT-R	2013	Subcompact Cars
1	32205	Volkswagen	CC	2013	Compact Cars
2	32206	Volkswagen	CC	2013	Compact Cars
3	32207	Volkswagen	CC 4motion	2013	Compact Cars
4	32208	Chevrolet	Malibu eAssist	2013	Midsize Cars
5	32209	Lexus	GS 350	2013	Midsize Cars
6	32210	Lexus	GS 350 AWD	2013	Midsize Cars
7	32214	Hyundai	Genesis Coupe	2013	Subcompact Cars
8	32215	Hyundai	Genesis Coupe	2013	Subcompact Cars
9	32216	Hyundai	Genesis Coupe	2013	Subcompact Cars

	drive	trans	fuelType	cylinders	displ
0	All-Wheel Drive	Automatic (AM6)	Premium Gasoline	6	3.8
1	Front-Wheel Drive	Automatic (AM-S6)	Premium Gasoline	4	2.0
2	Front-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6
3	All-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6
4	Front-Wheel Drive	Automatic (S6)	Regular Gasoline	4	2.4
5	Rear-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.5
6	All-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.5
7	Rear-Wheel Drive	Automatic 8-spd	Premium Gasoline	4	2.0
8	Rear-Wheel Drive	Manual 6-spd	Premium Gasoline	4	2.0
9	Rear-Wheel Drive	Automatic 8-spd	Premium Gasoline	6	3.8

	pv2	pv4	city	UCity	highway
0	79	0	16.4596	20.2988	22.5568
1	94	0	21.8706	26.9770	31.0367
2	94	0	17.4935	21.2000	26.5716
3	94	0	16.9415	20.5000	25.2190
4	0	95	24.7726	31.9796	35.5340
5	0	99	19.4325	24.1499	28.2234
6	0	99	18.5752	23.5261	26.3573
7	89	0	17.4460	21.7946	26.6295
8	89	0	20.6741	26.2000	29.2741
9	89	0	16.4675	20.4839	24.5605

	UHighway	comb	co2	feScore	ghgScore
0	30.1798	18.7389	471	4	4
1	42.4936	25.2227	349	6	6
2	35.1000	20.6716	429	5	5
3	33.5000	19.8774	446	5	5
4	51.8816	28.6813	310	8	8
5	38.5000	22.6002	393	6	6
6	36.2109	21.4213	412	5	5
7	37.6731	20.6507	432	5	5
8	41.8000	23.8235	375	6	6
9	34.4972	19.3344	461	4	4

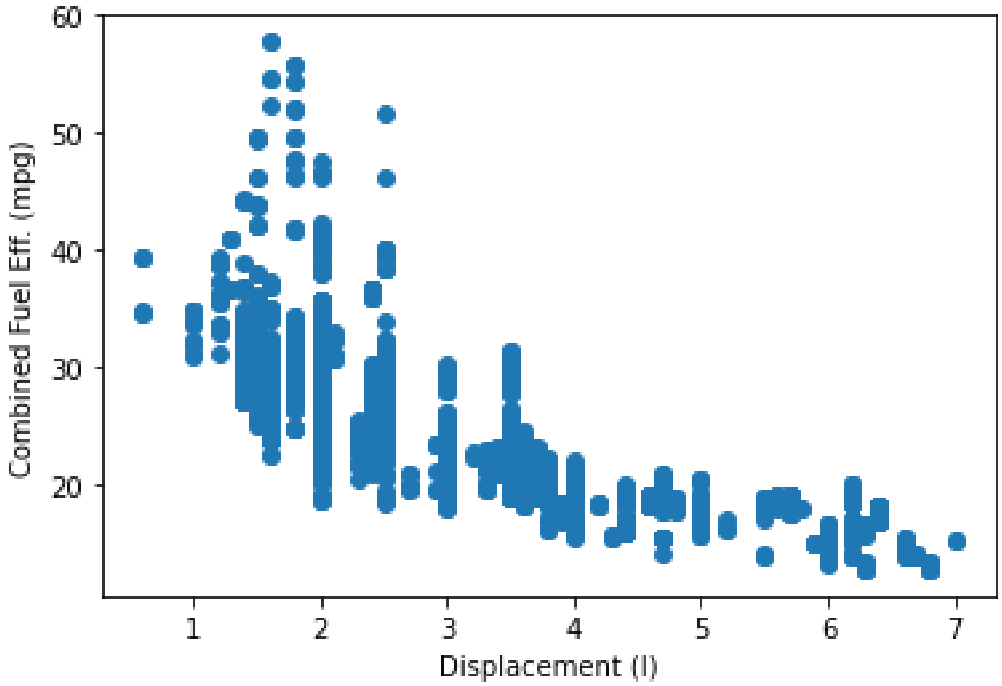

Figure 15: Basic Matplotlib Scatter Plot.

Next, we'll start off with our first bivariate plot, a scatterplot comparing size of the engine, 'disp (L)', to the combined city and highway miles per gallon, 'comb (mpg)'.

```
plt.scatter(data = fuel_econ, x = 'displ', y = 'comb')
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 15

# seaborn's version incudes a line
sb.regplot(data = fuel_econ, x = 'displ', y = 'comb', fit_reg =
                                True)
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 16
```

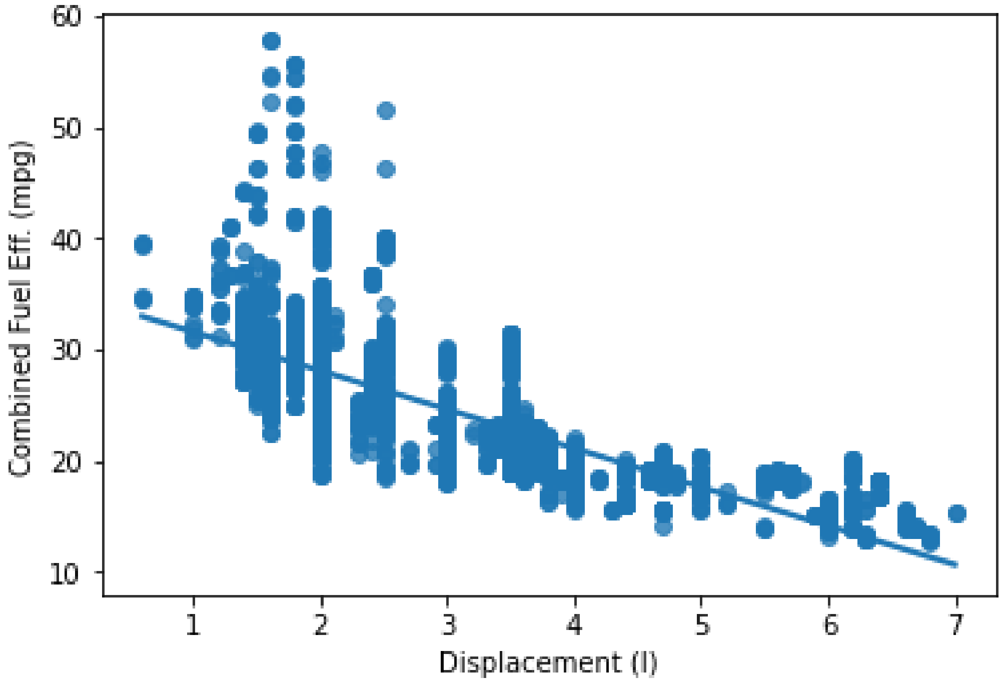

Figure 16: Basic Seaborn Scatter Plot.

3.3 Multivariate Exploration of Data

4 Explanatory Visualizations

5 Visualization Case Study