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:

```
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

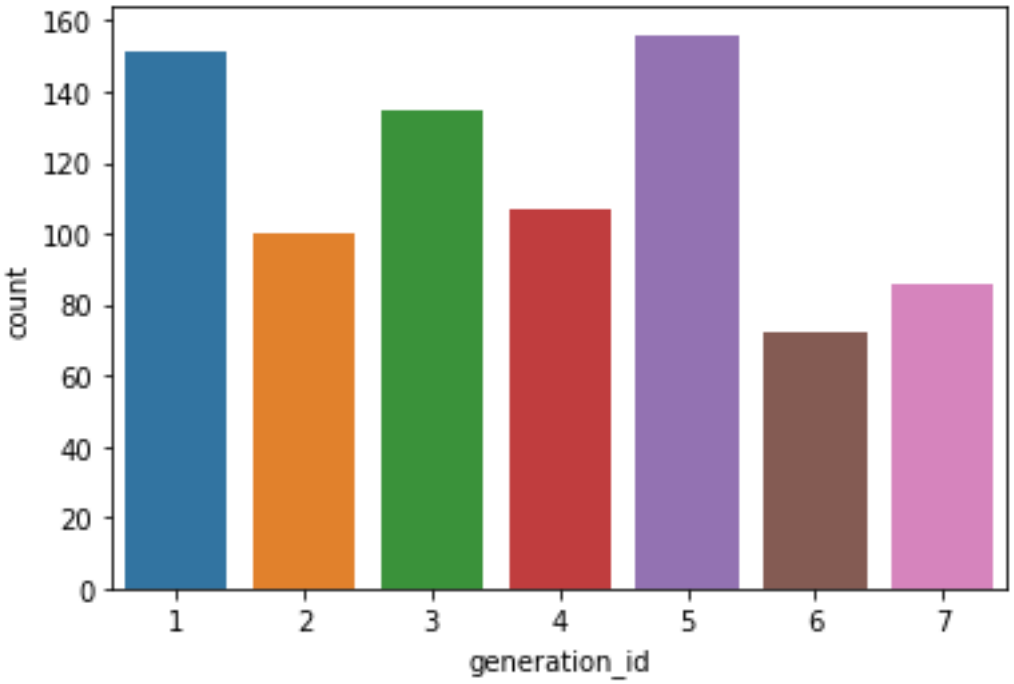

Figure 1: Generic bar chart.

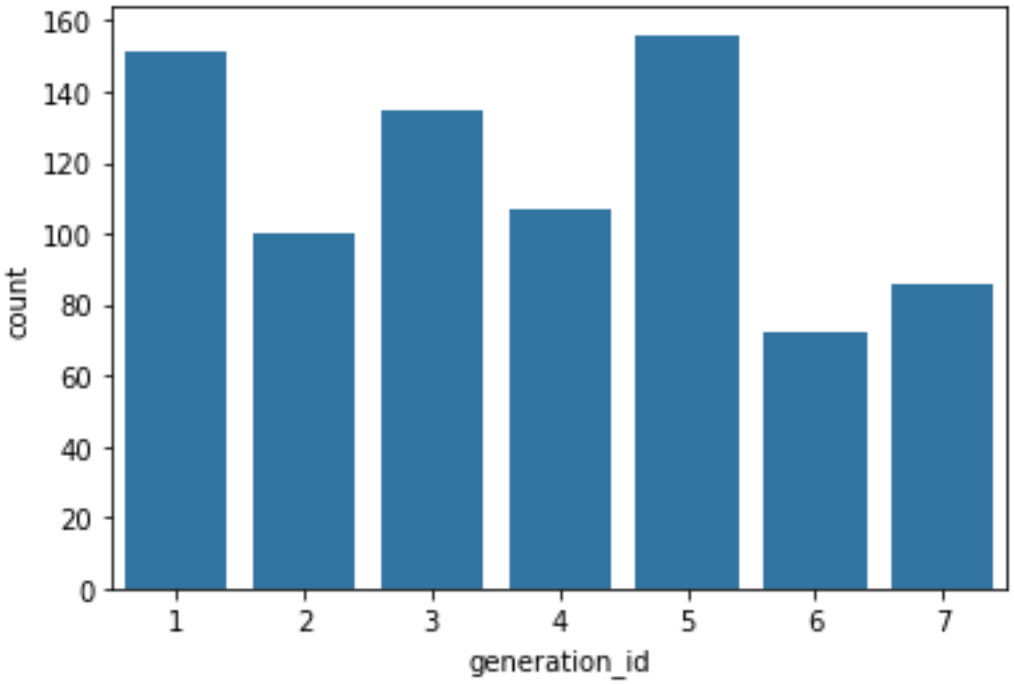

Figure 2: Bar chart with neutral coloring.

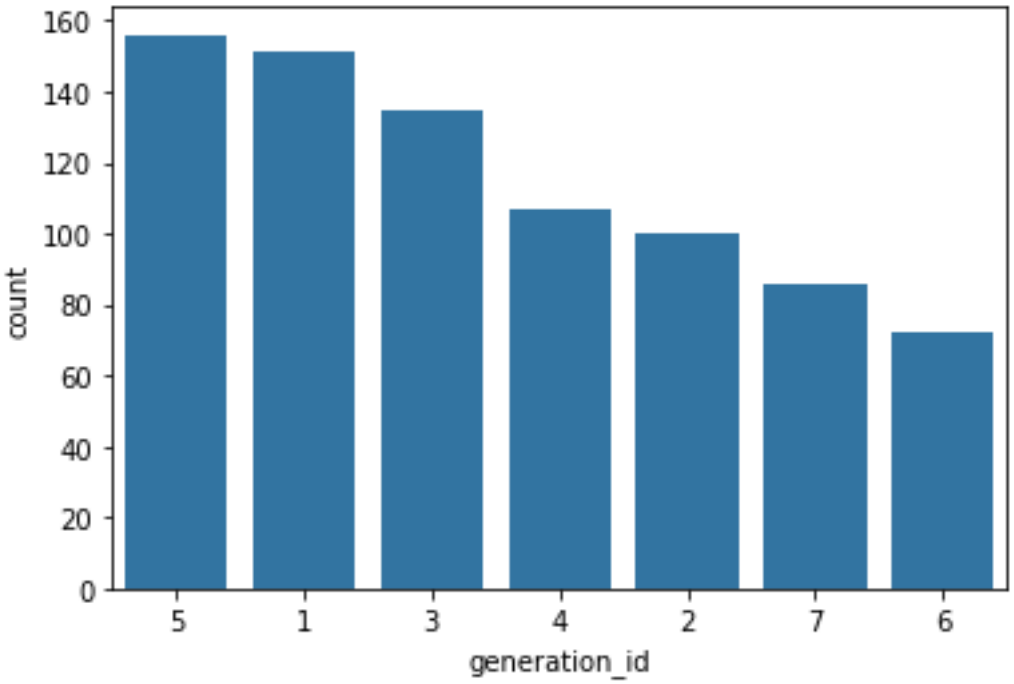

Figure 3: Bar chart with neutral coloring and ordering applied.

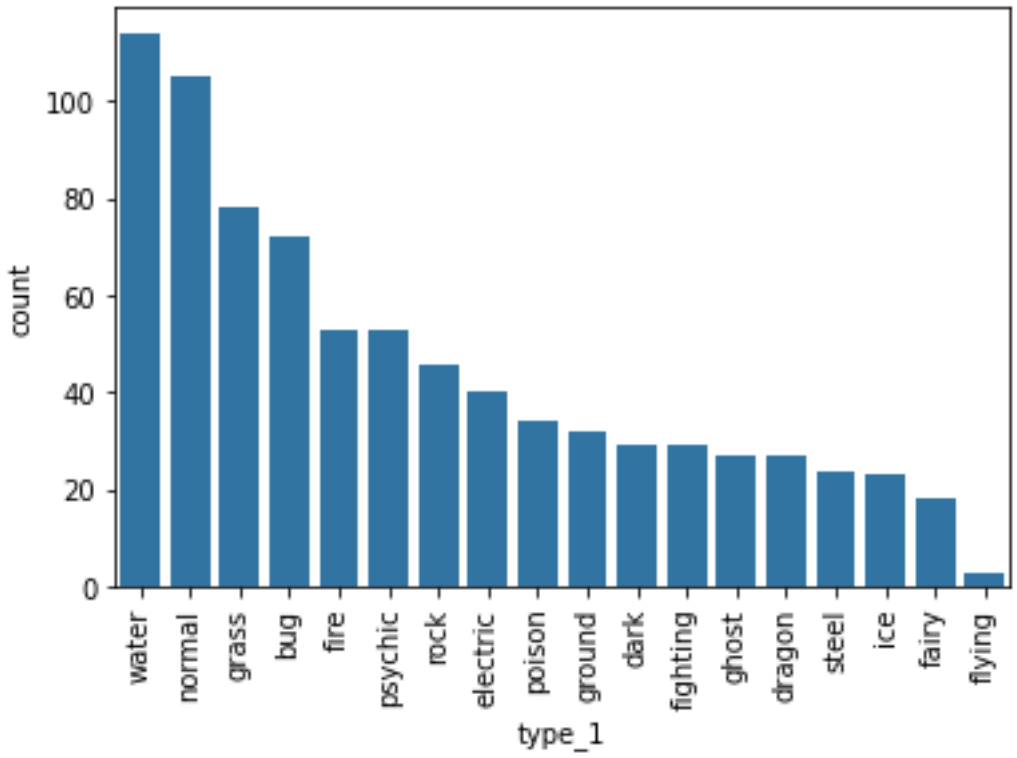

Figure 4: Standard bar chart format with x-axis labels rotated.

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'],
```

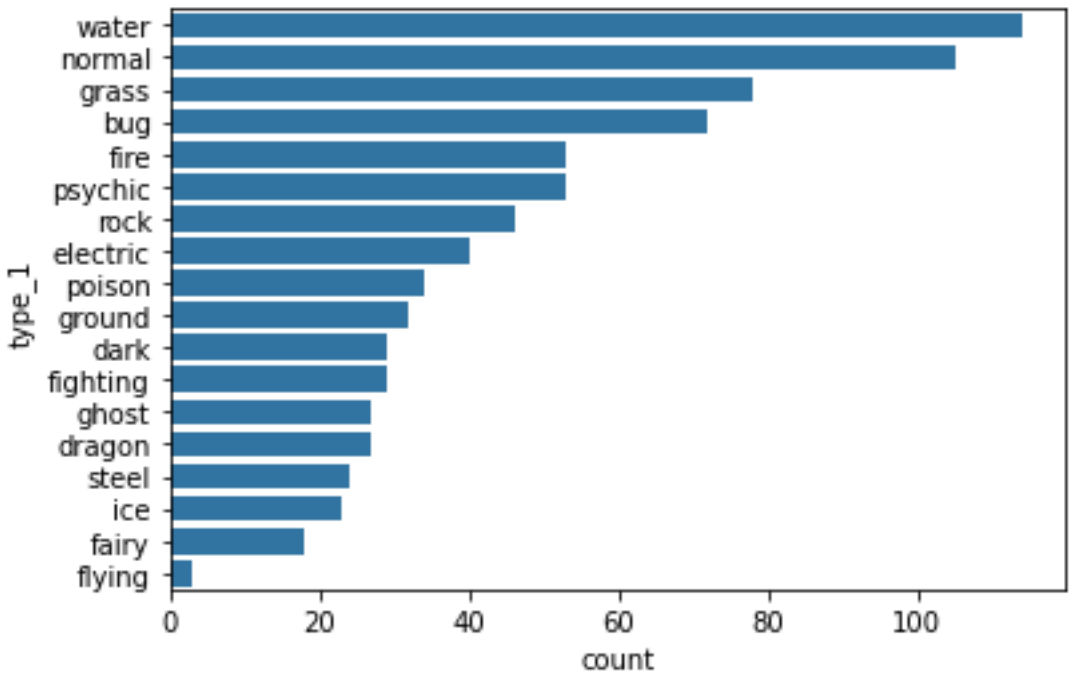

Figure 5: Standard bar chart format in vertical form.

```
        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

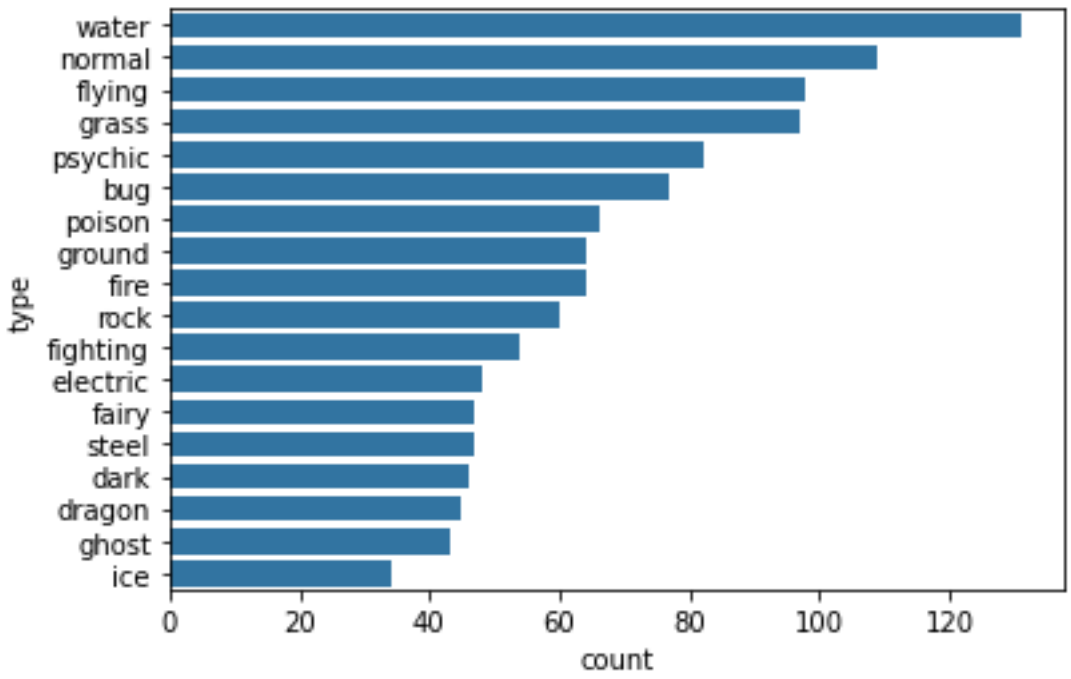

Figure 6: Bar chart of melted pokemon types.

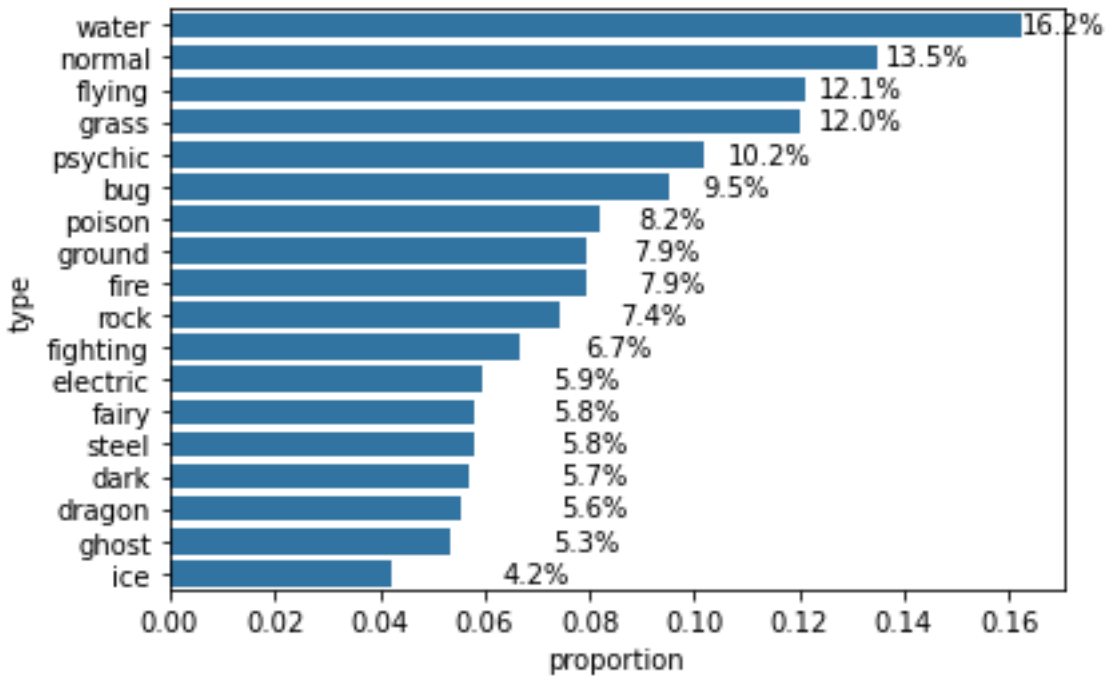

Figure 7: Bar chart of melted pokemon types shown in relative frequency.

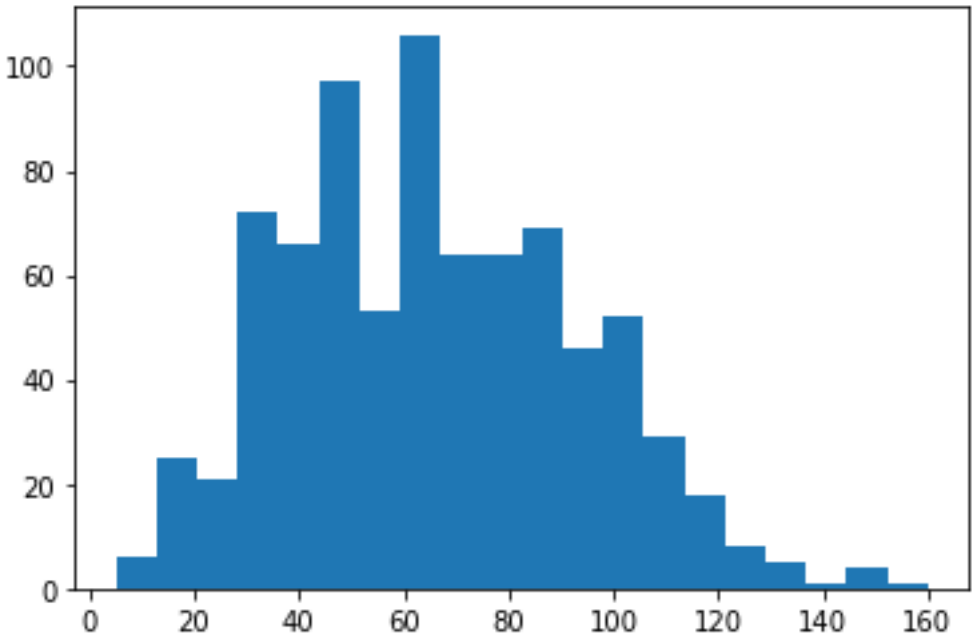

Figure 8: Basic histogram of Pokemon speeds.

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.

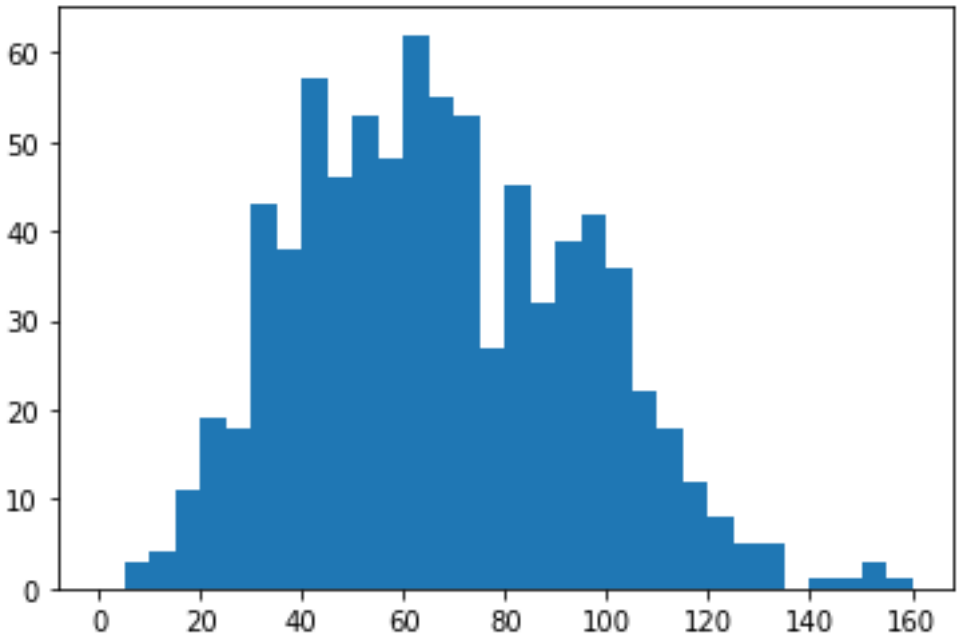

Figure 9: Histogram of Pokemon speeds with altered bins.

```
# 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 acceptible
                                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

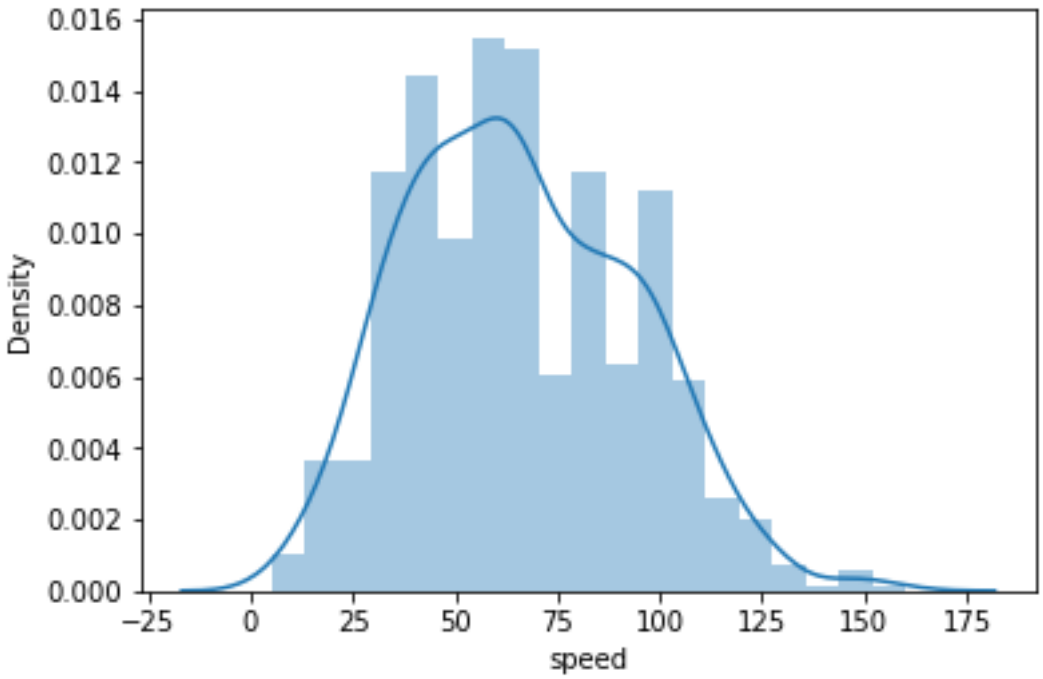

Figure 10: Seaborn's default histogram.

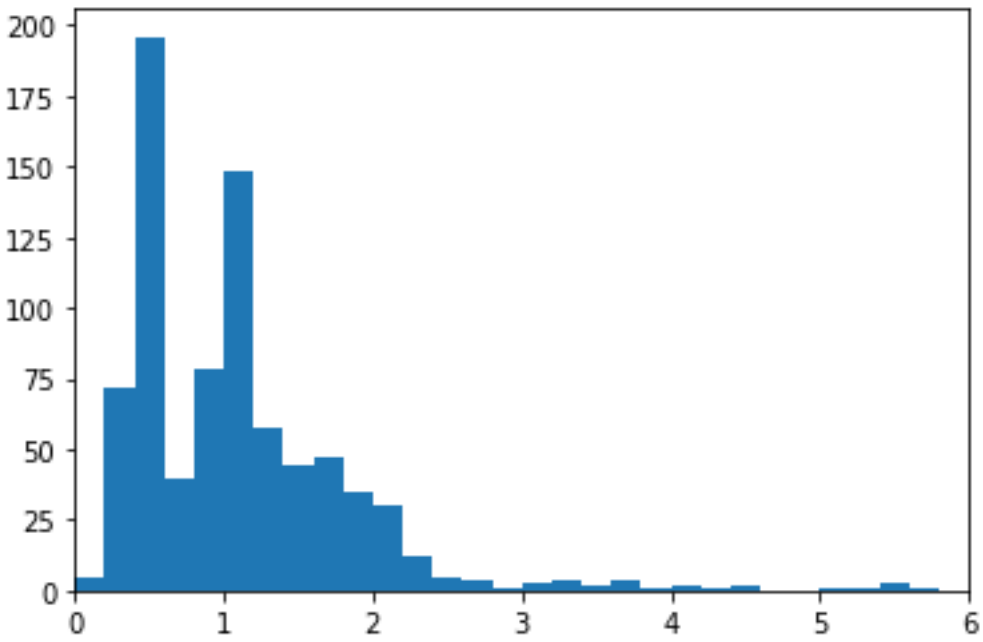

Figure 11: Histogram with altered x-axis limits.

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')
```

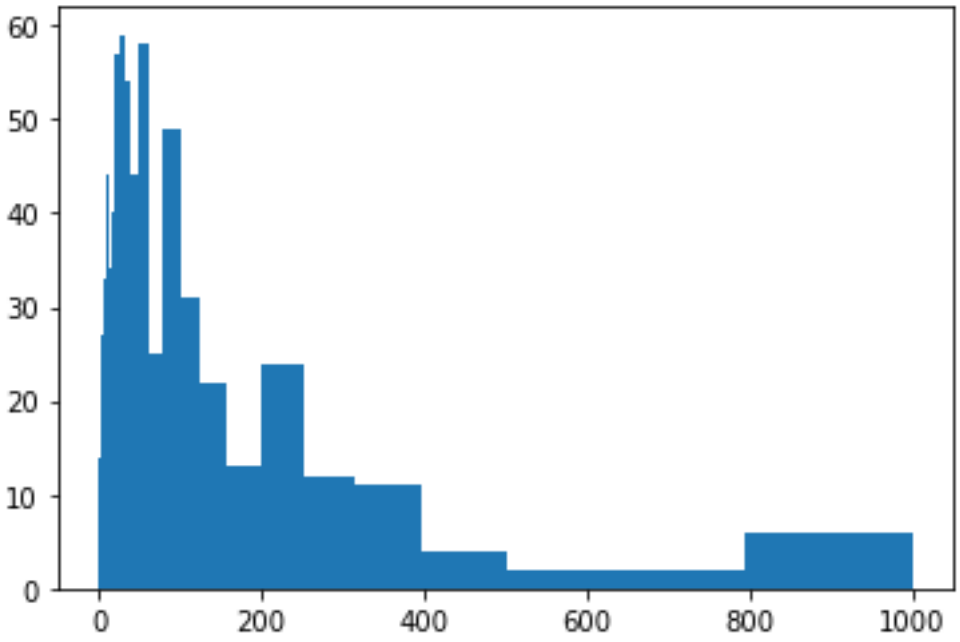

Figure 12: First try histogram of Pokemon's weight.

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

# See Figure 14
```

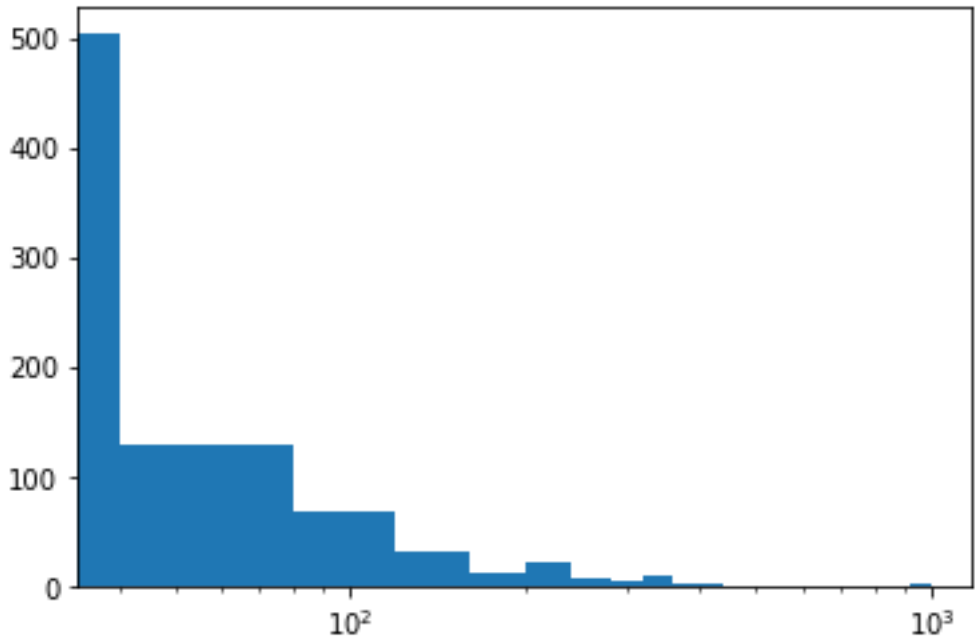

Figure 13: Histogram of Pokemon's weight with a log transformation applied to the x-scale.

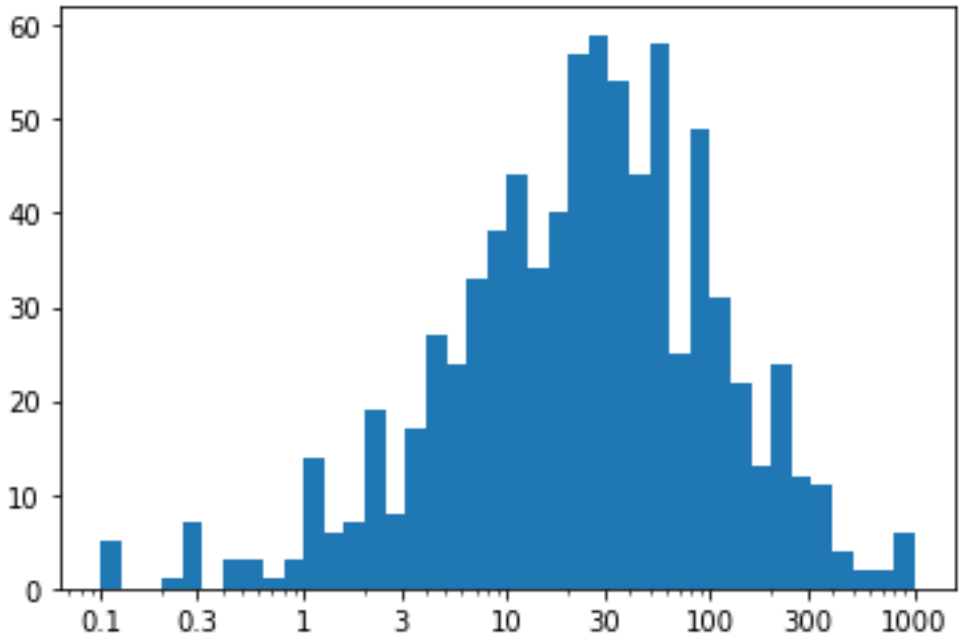

Figure 14: Histogram of Pokemon's weight with transformation and scaling applied.

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.
- plt.hist2d: Matplotlib's command for a heat map.
- sb.violinplot: Seaborn's command for a violin plot.
- sb.boxplot: Seaborn's command for a box plot.
- Creating Facets:
  - g = sb.FacetGrid: Seaborn's initial command for creating facets (creates object to be called in secondary command). Specify category to create facets around.
  - g.map: Seaborn's secondary command for creating facets. Specify type of plot and quantitative (usually) data to create facets around.
- sb.barplot: Seaborn's bar plot command (not to be confused with sb.countplot). For use when pairing categorical values with quantitative values.
- sb.pointplot: Seaborn's command for creating a line plot.

Terminology:

- heat maps: Use for discrete vs. discrete variables. Conceptualize as a top down 2-dimensional histogram. Can be an alternative to solving overplotting with a transparency method.
- scatter plots: Use for quantitative vs. quantitative variables.
- violin plots: Use for quantitative vs. qualitative variables. Plots a line from the minimum to maximum of the quantitative values, and features symmetrical smooth curves on both sides of the line which are wider at denser locations.
- box plots: Use for quantitative vs. qualitative variables. Plots a line from the minimum to maximum of the quantitative values, with a box drawn around the interquartile range [Q1:25th percentile, Q3; 75th percentile].
- clustered bar charts: Use for qualitative vs. quantitative variables. Compares variables by plotting aspects of categories side by side.

- faceting: creating a series of plots which share the same set of axes, but each subplot shows a subset of the data.
- line plots: A scatter plot where point estimates are marked for each of the categories with a given confidence interval. Also known as a time series plot.
- overplotting: When data or labels in a data visualization overlap, making it difficult to see individual points.
- sampling: Choose a subset of the data (best if randomized).
- transparency: By making the points partially transparent (increase the opacity), overplotting becomes evident by darker regions.
- jitter: By adding small randomly generated numbers to the values prior to plotting, overplotting becomes event by denser regions.

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

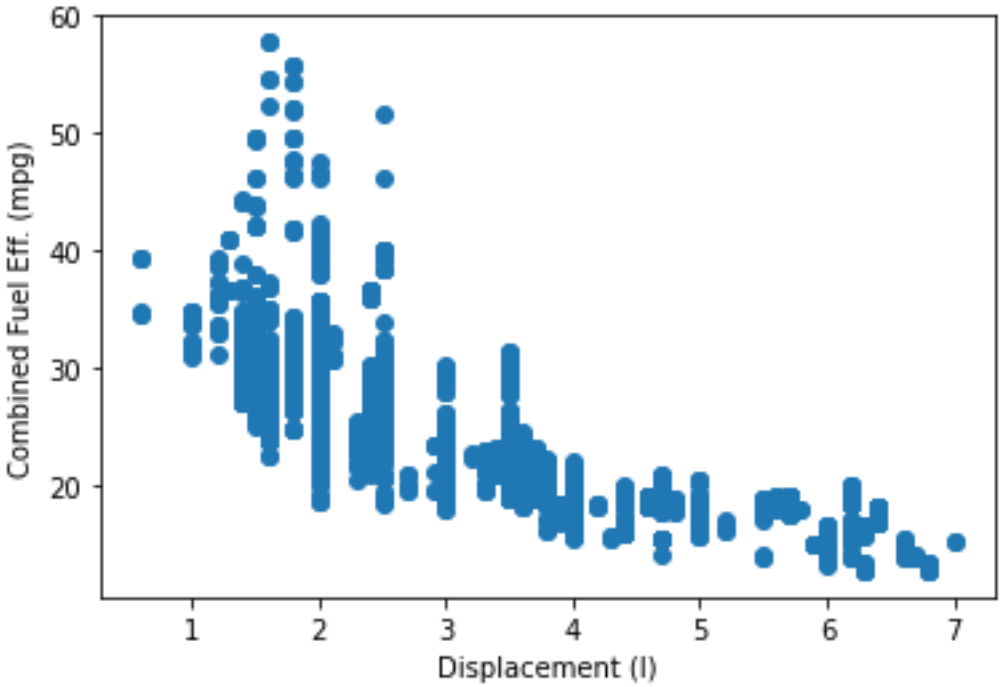

Figure 15: Basic Matplotlib Scatter Plot.

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 fuel efficiency, '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
```

The dataset we're using has almost 4,000 entries. It will likely be the case that future datasets will have many more elements. Even though we're treating the displacement variable (volume in the engine in Liters) as a quantitative variable, it could almost be expressed as a categorical variable, although it is technically a discrete quantitative variable.

```
fuel_econ['displ'].value_counts().size
# 46 different Liter options in this dataset
```

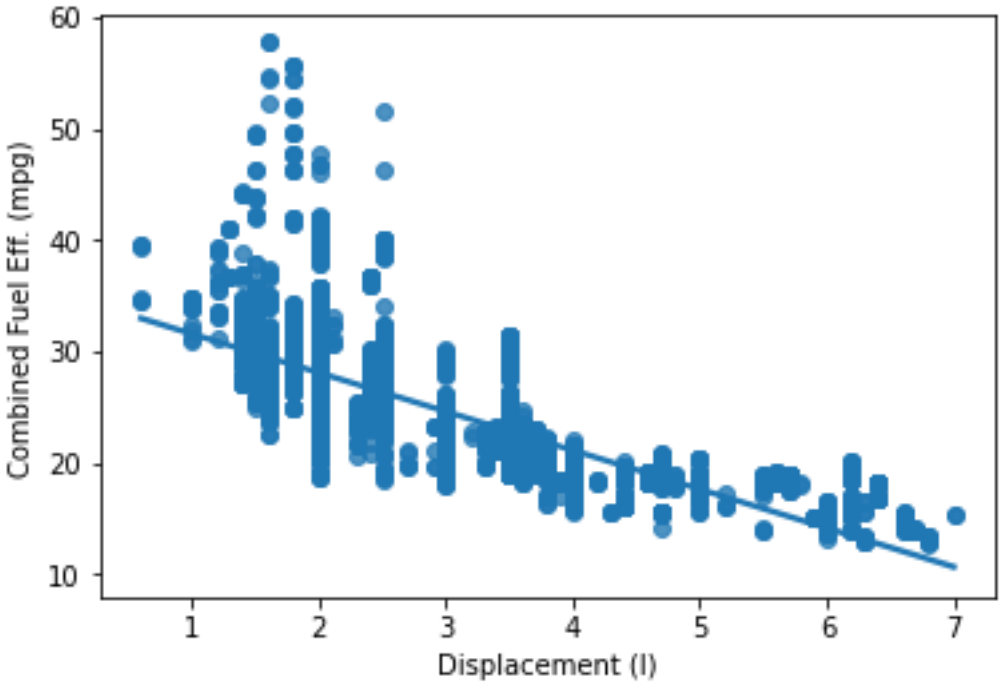

Figure 16: Basic Seaborn Scatter Plot.

```
fuel_econ['displ'].value_counts().head(10)
"""
2.0     907
3.0     515
3.5     245
1.6     243
3.6     208
1.8     189
2.5     169
2.4     142
1.4     136
4.4     130
"""
```

Although there will be different combinations when paired with the fuel efficiency variable, we've still stacked over 900 entries on x = 2, with much of the data points overlapping each other. This is a prime example of overplotting. How do we fix this?

There are a few main methods to fix overplotting:

- sampling: Choose a subset of the data (best if randomized).
- transparency: By making the points partially transparent (increase the opacity), overplotting becomes evident by the darker regions.
- jitter: By adding small randomly generated numbers to the values prior to plotting, overplotting becomes event by denser regions.
- heat map: think of this as a 2-dimensional histogram where denser regions take on different colors.

In this example, we'll use Seaborn's scatter plot capabilities to add jitter and transparency, where transparency level is denoted with alpha. In a slight change of plot, we're mapping year onto overall fuel efficiency. Year is clearly another true example of discrete data.

```
sb.regplot(data = fuel_econ, x = 'year', y = 'comb', x_jitter = 0
                                .3, scatter_kws = {'alpha' : 1/20
                                })
# See Figure 17
```

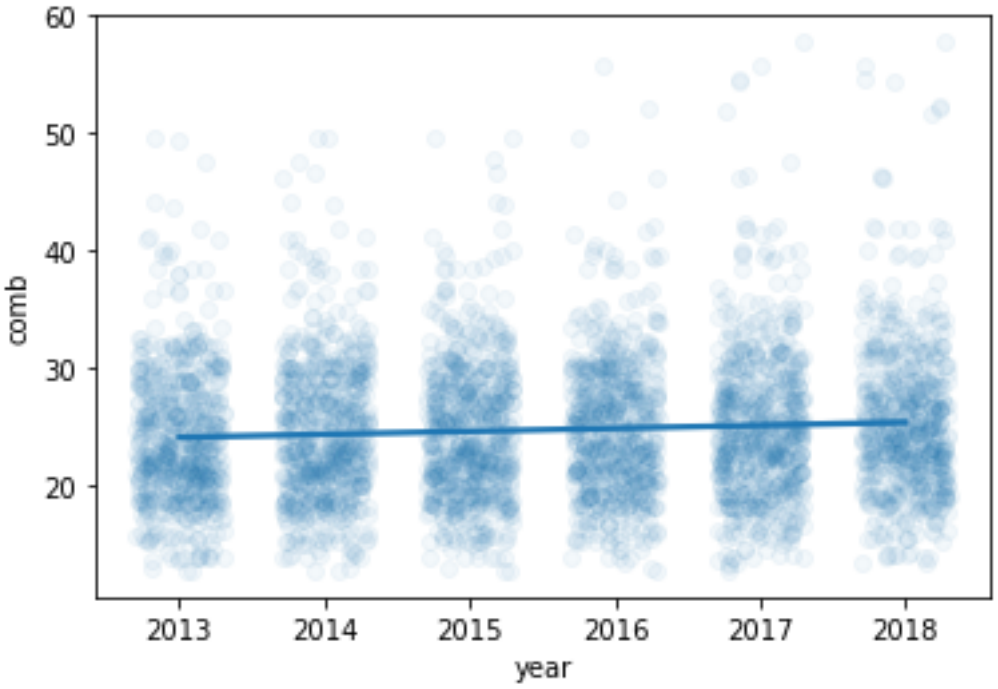

Figure 17: Seaborn Scatter Plot with jitter and transparency.

Thought we were done with histograms? Let's check them in 2-dimensions with heat maps. Instead of changing the settings on a scatter plot, we could use another type of plot called a heat map. Similar to histograms, we need to be thoughtful with bin selection. So, start with the descriptive statistics.

Statistics for Heat Map Guidance

	displ	comb
count	3929.000000	3929.000000
mean	2.950573	24.791339
std	1.305901	6.003246
min	0.600000	12.821700
25%	2.000000	20.658100
50%	2.500000	24.000000
75%	3.600000	28.227100
max	7.000000	57.782400

```
bins_x = np.arange(0.6, 7 + 0.3, 0.3)
bins_y = np.arange(12, 58 + 3, 3)
plt.hist2d(data = fuel_econ, x = 'displ', y = 'comb', cmin = 0.5,
                                  cmap = 'viridis_r', bins = [
                                 bins_x, bins_y])
plt.colorbar()
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 18
```

Not all inclusive, but to cover a major base, let's use a scatter plot for a continuous quantitative variable vs. a continuous quantitative variable.

```
sb.regplot(data = fuel_econ, x = 'city', y = 'highway',
scatter_kws = {'alpha' : 1/20})
# See Figure 19
```

Next up is pairing categorical data with quantitative data. There are two useful tools, which seem very similar, but can help tell different stories.

- violin plot
- box plot

One of the main differences between the two is that violin plots are wider where the data is more concentrated, which can be useful to see if data has more than one area of concentration (multimodal). Box plots shouldn't be down played. They are strong in placing emphasis on the interquartile range, and are known to be more simplistic and focused. Let's take a look!

```
# Violin Plots
sb.violinplot(data = fuel_econ, x = 'VClass', y = 'comb', color =
                                  base_color, inner = None)
plt.xticks(rotation = 15)
# See Figure 20
```

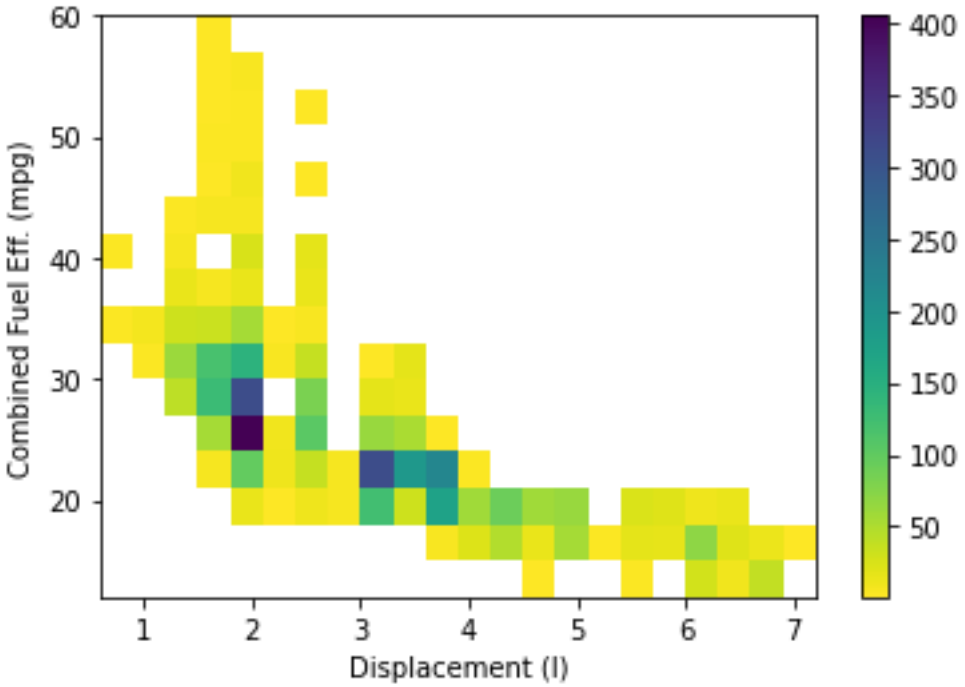

Figure 18: Basic Heat Map.

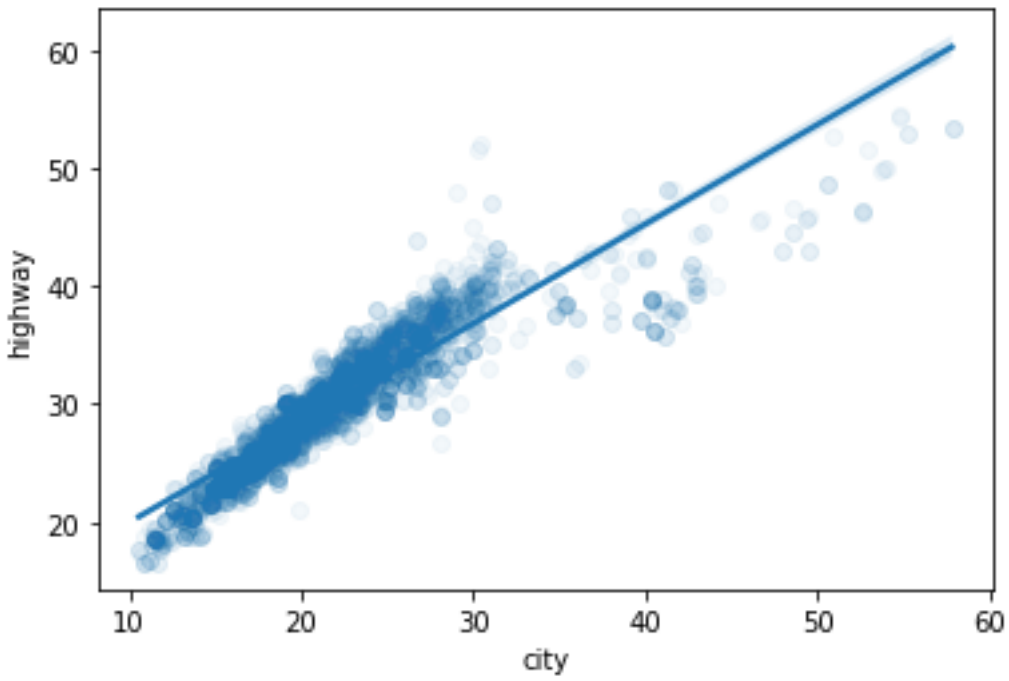

Figure 19: City vs. Highway Scatter Plot.

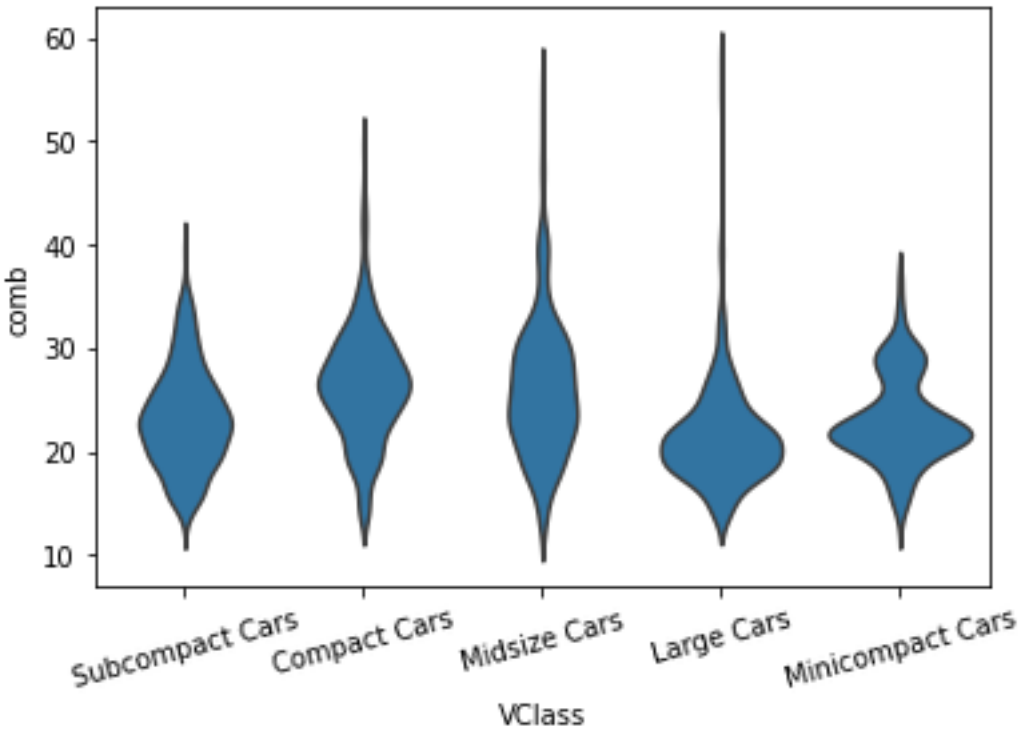

Figure 20: Basic Violin Plot.

```
# Box Plots (simplicity + focus, as compared to the violin plot)
sb.boxplot(data = fuel_econ, x = 'VClass', y = 'comb', color =
                                    base_color)
plt.xticks(rotation = 15)
# See Figure 21
```

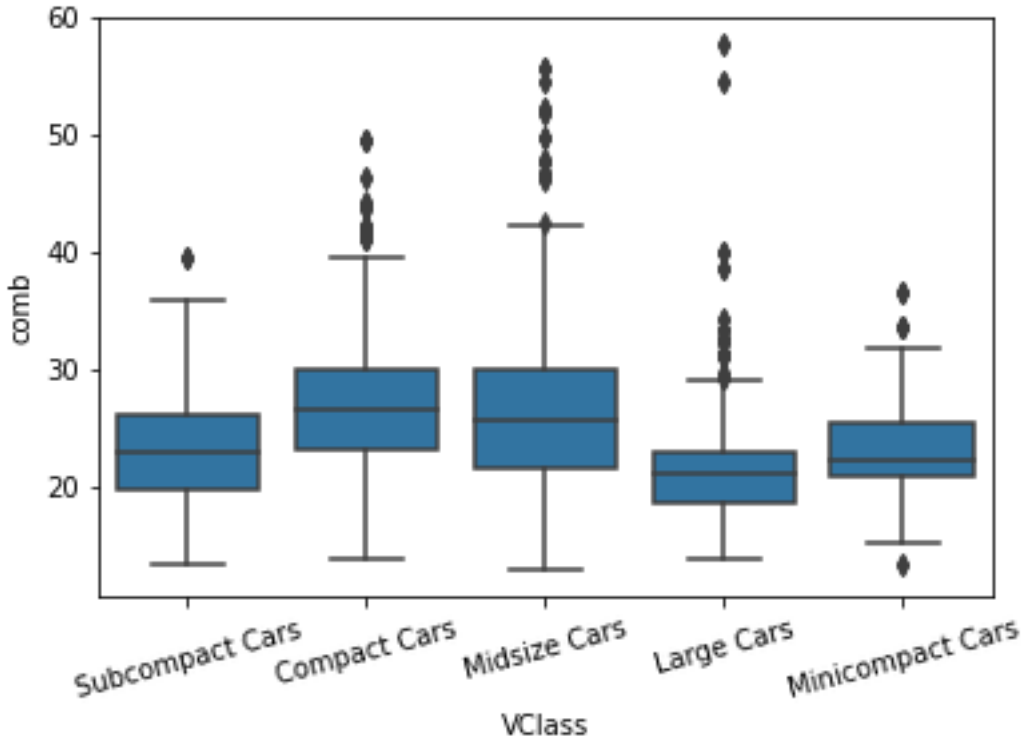

Figure 21: Basic Box Plot.

A quick aside:

It should absolutely be noted that with the prior, current and future plots illustrated, there are a plethora of options with each command. How should we handle the outliers? Can we transpose the IQR onto a violin plot? How can we focus on these specific data points? It's highy recommended to at least scan through at least some of the library and function documentation.

Probably the simplest, yet valuable, plot yet is a clustered bar chart. This takes our bar chart plot from the univariate exploration section, and simply adds more bars. These can be either different categories, or can be aspects of the same category.

For a quick example of a clustered bar chart, we'll continue on with our examination of Vehicle Class ('VClass'). As shown in the violin and box plots, there are 5 different types of vehicles in the dataset. However, we can break down the types of vehicles into subsets based on their transmission types (Automatic vs. Manual).

```
# a bit of data cleaning: just need auto vs. manual from the
                                strings
fuel_econ['trans_type'] = fuel_econ['trans'].apply(lambda x: x.
                                split()[0])
sb.countplot(data = fuel_econ, x = 'VClass', hue = 'trans_type')
plt.xticks(rotation = 15)
# See Figure 22
```

Let's do another example of a clustered bar chart that puts together different aspects. In this example, we want to investigate the difference between the two main fuel types (regular and premium) by vehicle class.

```
fuel_econ_subset = fuel_econ.loc[fuel_econ['fuelType'].isin(['
                                Premium Gasoline', 'Regular
                                Gasoline'])]
sb.countplot(data = fuel_econ_subset, y = 'VClass', hue = '
                                fuelType')
# See Figure 23
```

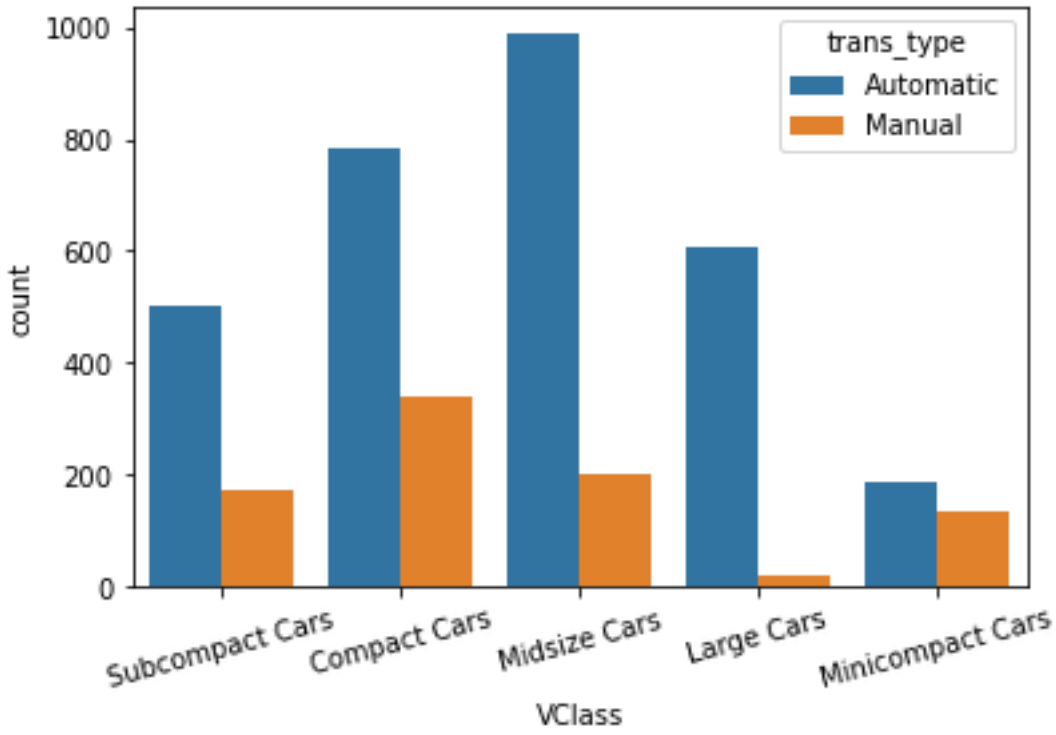

Figure 22: Vertical Clustered Bar Chart.

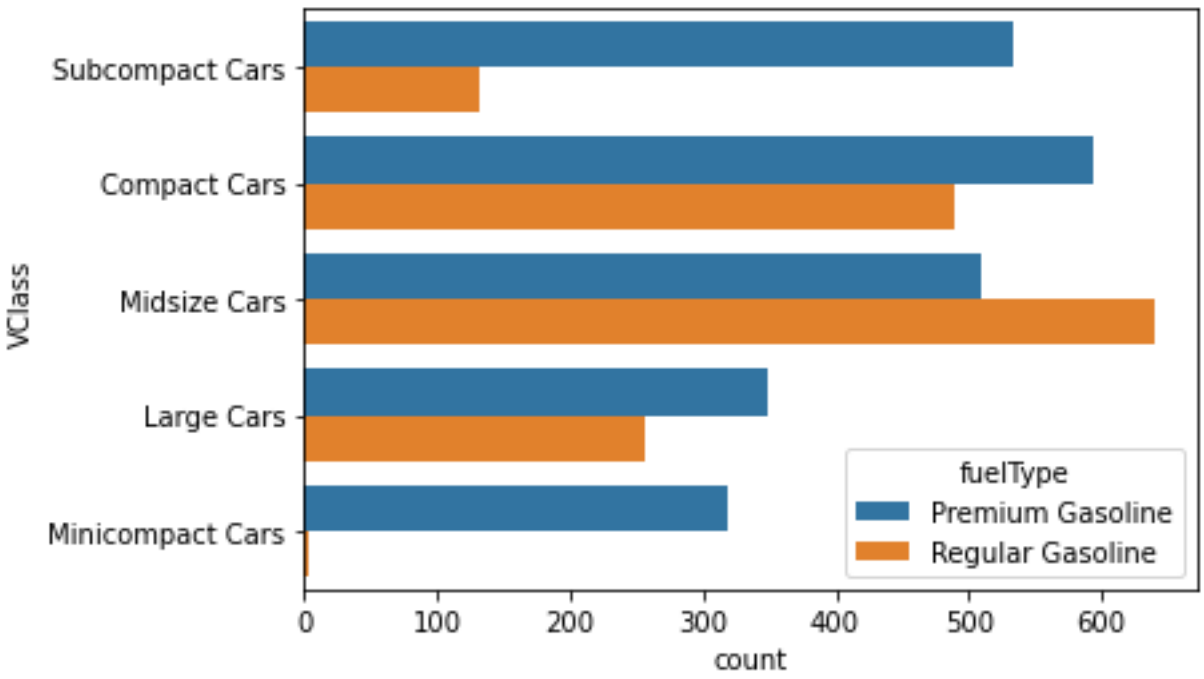

Figure 23: Vertical Clustered Bar Chart.

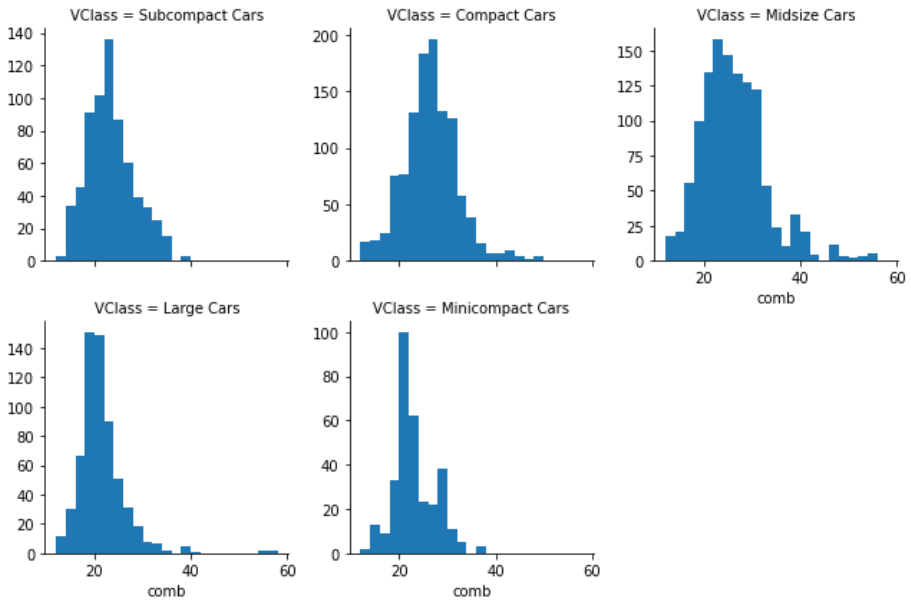

Figure 24: Vehicle Class Fuel Efficiency Facets.

With even more plotting techniques in our visualization toolbox, what happens when it seems sensible to test many different variations of a similar plot against the data, can we avoid creating a single plot at a time? Enter faceting, creating side by side series of plots containing subsets of the data.

For an example, let's create a plot for every type of vehicle paired with combined efficiency.

```
bins = np.arange(12, 58 + 2, 2)
g = sb.FacetGrid(data = fuel_econ, col = 'VClass', col_wrap = 3,
                                    sharey = False)
g.map(plt.hist, 'comb', bins = bins)
# See Figure 24
```

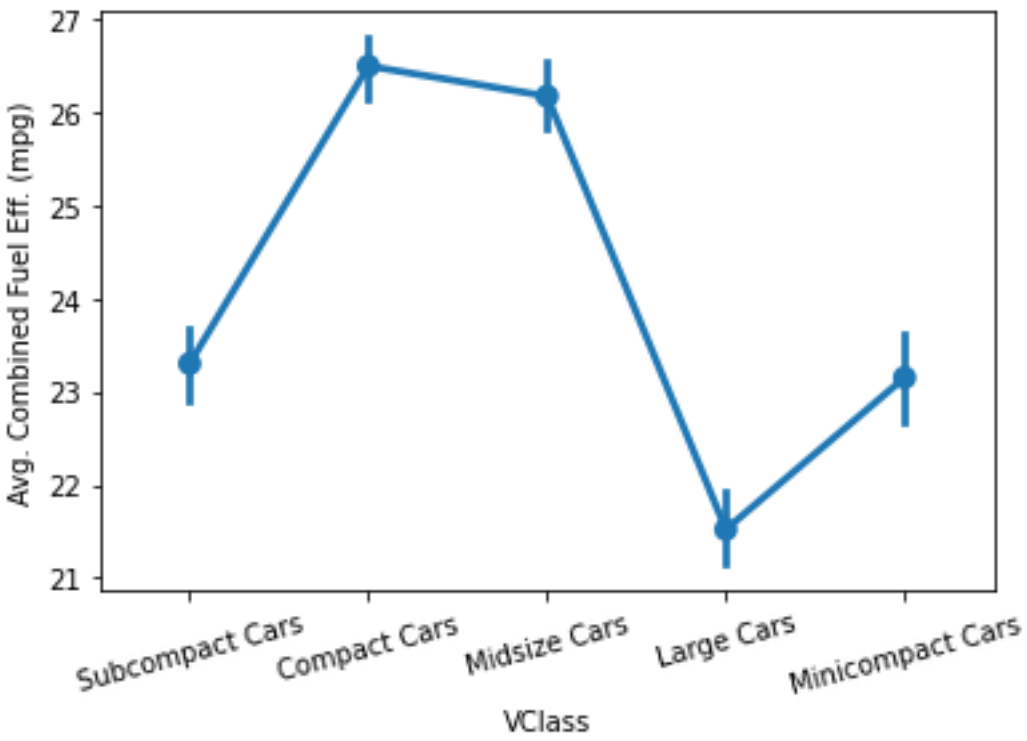

Figure 25: Line Plot.

One last plot type to show for this section is a line plot, which shows a point estimate along with a confidence interval while comparing categorical variables with quantitative data.

```
sb.pointplot(data = fuel_econ, x = 'VClass', y = 'comb')
plt.xticks(rotation = 15)
plt.ylabel('Avg. Combined Fuel Eff. (mpg)')
# See Figure 25
```

We'll end our main discussion of bivariate data exploration with a challenge. Our task is to plot the distribution of combined fuel mileage (comb), by manufacturer, for all manual transmission vehicles with at least eighty cars in dataset.

```
# get manufacters with more than 80 entries
threshold = 80
manu_count = fuel_econ['make'].value_counts()
manu_index = np.sum(manu_count > threshold)

manu_thresh = manu_count.index[:manu_index]
fuel_econ_subset = fuel_econ.loc[fuel_econ['make'].isin(
                                manu_thresh)]

sb.boxplot(data = fuel_econ_subset, x = 'make', y = 'comb', color
                                 = base_color)
plt.xticks(rotation = 90)
# See Figure 26

manu_means = fuel_econ_susetb.groupby('make').mean()
comb_order = manu_means.sort_values('comb', ascending = False).
                                index


# plotting
g = sb.FacetGrid(data = fuel_econ_sub, col = 'make', col_wrap = 6
                                , size = 2, col_order =
                                comb_order)

# try sb.distplot instead of plt.hist to see the plot in terms of
                                 density!
g.map(plt.hist, 'comb', bins = np.arange(12, fuel_econ_subset['
                                comb'].max()+2, 2))
g.set_titles('{col_name}')
# See Figure 27

# plot the mean fuel efficiency
sb.barplot(data = fuel_econ_sub, y = 'make', x = 'comb', color =
                                base_color, ci = 'sd', order =
                                comb_order)
# See Figure 28
```

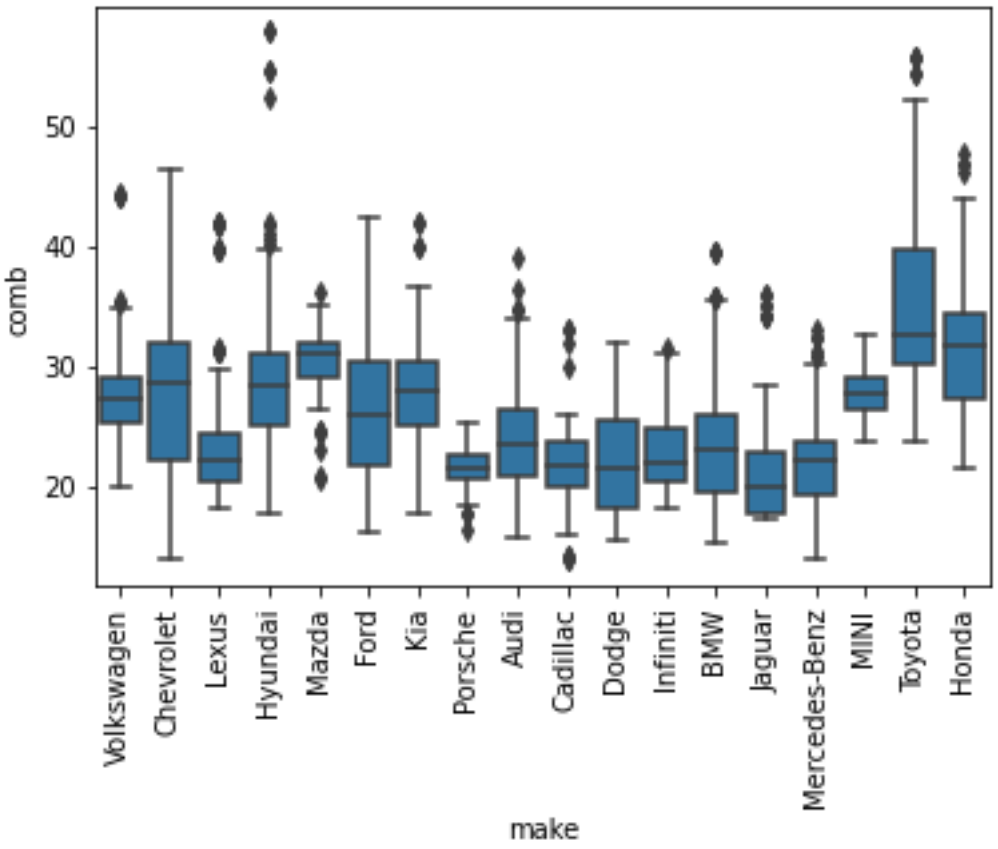

Figure 26: Final Challenge: Facets.

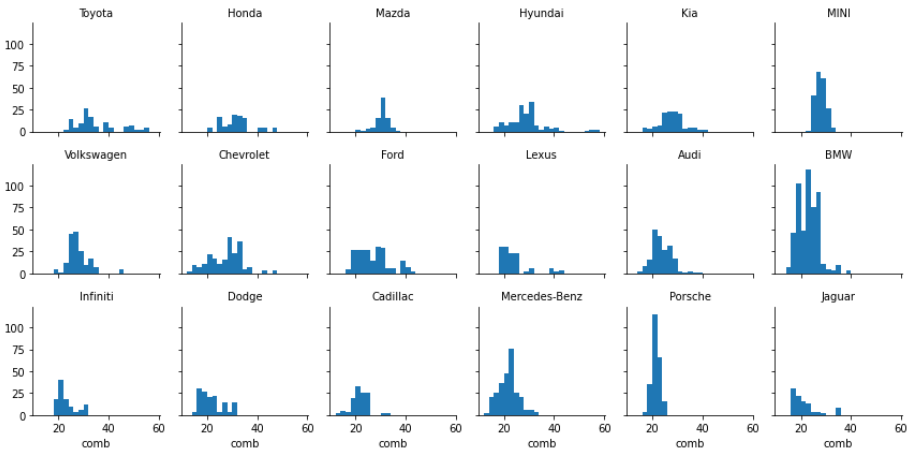

Figure 27: Final Challenge: Facets.

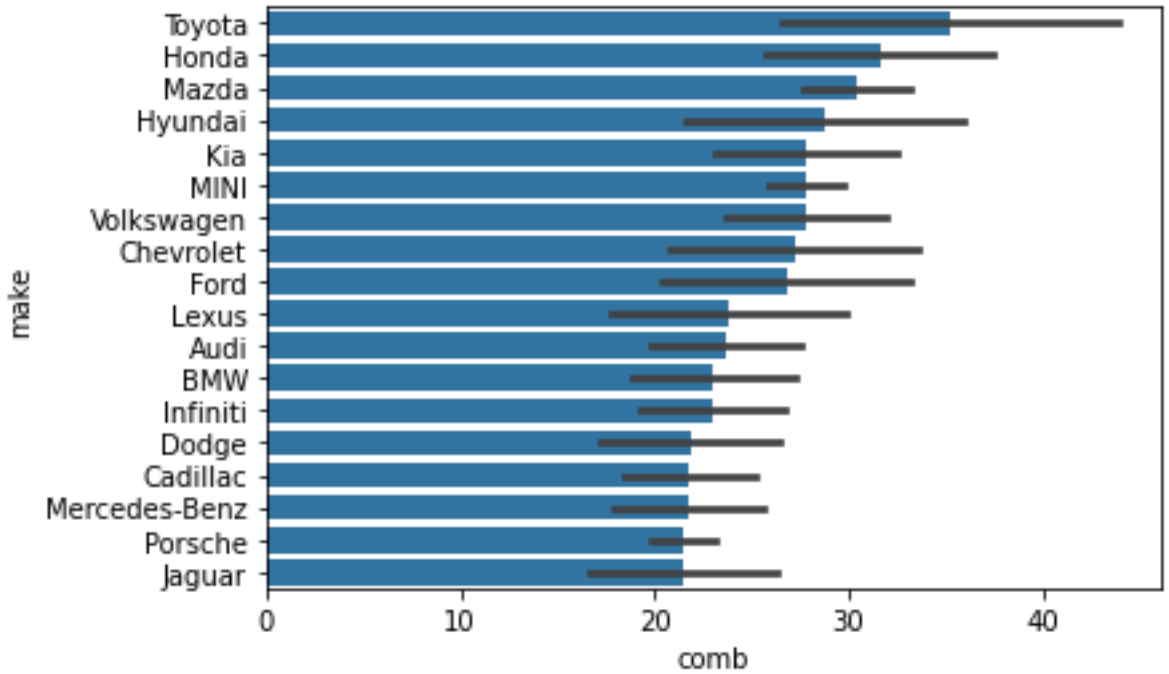

Figure 28: Final Challenge: Mean Fuel Efficiency.

3.3 Multivariate Exploration of Data

We'll be making use of both the pokemon and fuel_econ datasets while discussing exploration of multivariate data. When plotting three variables together, there are four major cases to consider:

- three numeric variables
- two numeric variables and one categorical variable
- one numberic variable and two categorical variables
- three categorical variables

Commands:

- np.random.seed: Use when you want "randomized" results to be reproducible.
- np.random.choice: If an ndarray is passed through, a random sampling is produced from its data. If an integer is passed through, a random sampling is generated as if it were np.arange.
- sb.PairGrid: Seaborn function for Plot Matrices (i.e. plot all pairwise relationships among data for specified variables).
- sb.PairGrid.map_offdiag: Will plot onto currently active matplotlib axes (i.e. will draw over the current diagonal of the plot matrices).

Terminology:

- Non-Positional Encoding (for Third Variables): Recall the main encodings of size, shape and color. If we have a plot which illustrates a relationship between two of the variables, we can use an encoding to illustrate another relationship.
- Plot Matrices: Plot pairwise relationships among data given specified variables.
- Feature Engineering: Creating a new variable by leveraging relationships between variables with mathematical operations.

We'll start with an example illustrating the effectiveness of non-positional encodings for a third variable.

An example of using color as the encoder:

```
"""
We'll start by taking a random sampling from the fuel_econ
                                dataset. Note that since this a
                                NumPy function, an ndarray must
                                be passed through. A Pandas
                                DataFrame will not work, which is
                                 why we must use two steps.
```

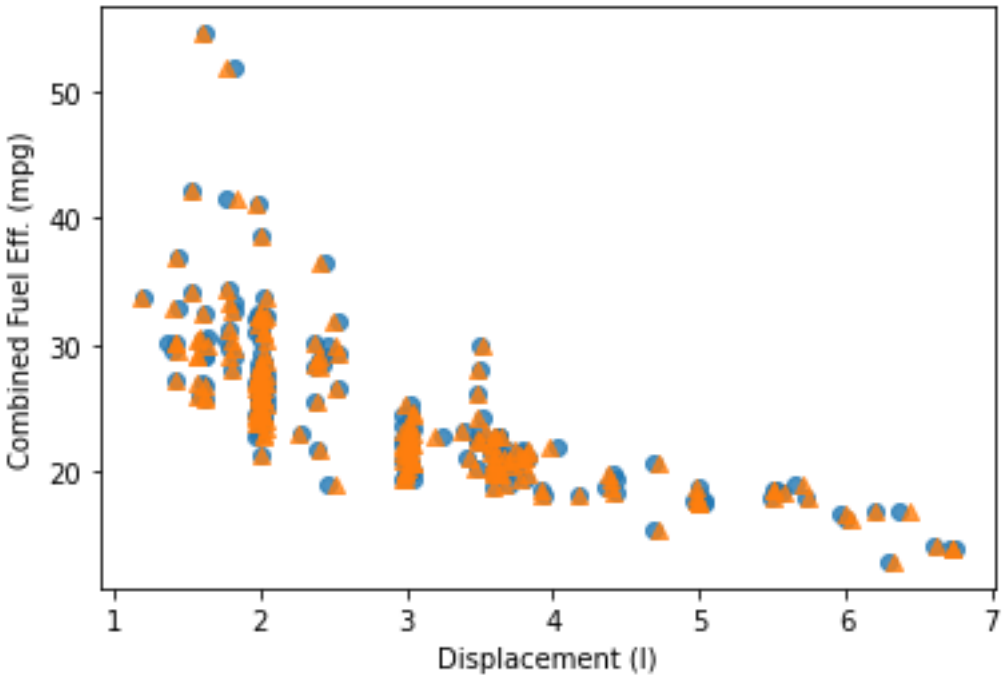

Figure 29: Non-Positional Encoding Example: Shape.

```
"""
np.random.seed(2018)
sample = np.random.choice(fuel_econ.shape[0], 200, replace =
                                    False)
fuel_econ_subset = fuel_econ.loc[sample]

ttype_markers = [['Automatic', 'o'], ['Manual', '^']]

for ttype, marker in ttype_markers:
  plot_data = fuel_econ_subset.loc[fuel_econ_subset['trans_type']
                                    == ttype]
  sb.regplot(data = fuel_econ_subset, x = 'displ', y = 'comb',
                                    x_jitter = 0.04, fit_reg = False,
                                     marker = marker)
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 29
```

An example of using size as an encoder:

```
sizes = [200, 350, 500]
legend_obj = []
for s in sizes:
  legend_obj.append(plt.scatter([],[],s = s/2, color = base_color
                                    ))
```

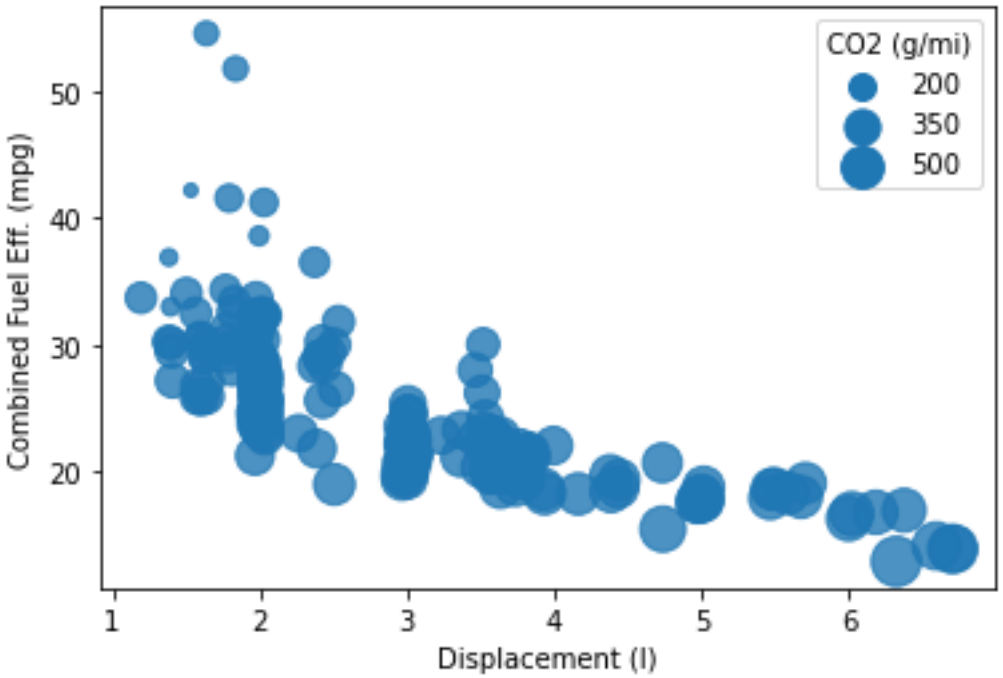

Figure 30: Non-Positional Encoding Example: Size.

```
plt.legend(legend_obj, sizes, title = 'CO2 (g/mi)')
sb.regplot(data = fuel_econ_subset, x = 'displ', y = 'comb',
                                    x_jitter = 0.04, fit_reg = False,
                                     scatter_kws = {'s' :
                                    fuel_econ_subset['co2']/2})
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 30
```

As we could've guessed, when it comes to encodings, color is more natural. Here are three different examples which use color encoding to represent a third variable.

```
"""
Note: FacetGrid is used differently in the first two examples.
                                Rather than create a series of
                                plots, its used to separate
                                categorical data from the third
                                variable.
"""

g = sb.FacetGrid(data = fuel_econ_subset, hue = 'trans_type',
                                hue_order = ['Automatic', 'Manual
                                '], size = 4, aspect = 1.5)
g.map(sb.regplot, 'displ', 'comb', x_jitter = 0.04, fit_reg =
                                False)
g.add_legend()
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 31

g = sb.FacetGrid(data = fuel_econ_subset, hue = 'VClass', size =
                                4, aspect = 1.5, palette = '
                                viridis_r')
g.map(sb.regplot, 'displ', 'comb', x_jitter = 0.04, fit_reg =
                                False)
g.add_legend()
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 32

plt.scatter(data = fuel_econ_subset, x = 'displ', y = 'comb', c =
                                 'co2', cmap = 'viridis_r')
plt.colorbar(label = 'CO2 (g/mi)')
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
# See Figure 33
```

Recall the pokemon dataset, and how we previously unpivoted the type_1 and type_2 data together? Let's resurrect that code and mix encodings with transformations to examine relationships between height, weight and type.

```
# create our single column of types
type_cols = ['type_1','type_2']
non_type_cols = pokemon.columns.difference(type_cols)
pkmn_types = pokemon.melt(id_vars = non_type_cols, value_vars =
                                type_cols, var_name = 'type_level
                                ', value_name = 'type').dropna()

# apply a log transformation to the data we're examining
pkmn_types['log_weight'] = np.log(pkmn_types['weight'])
pkmn_types['log_height'] = np.log(pkmn_types['height'])

# Let's specifically zoom in on fairy and dragon type pokemon
ptypes = ['fairy', 'dragon']
```

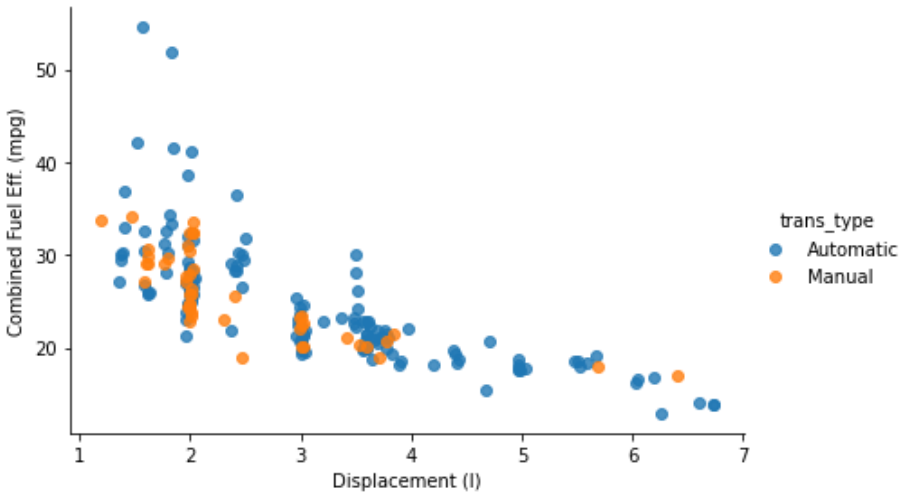

Figure 31: Non-Positional Encoding Example: Color #1.

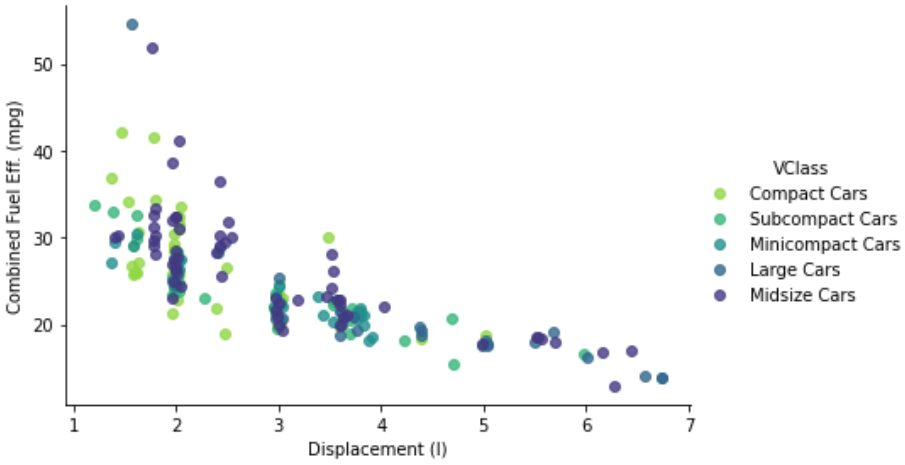

Figure 32: Non-Positional Encoding Example: Color #2.

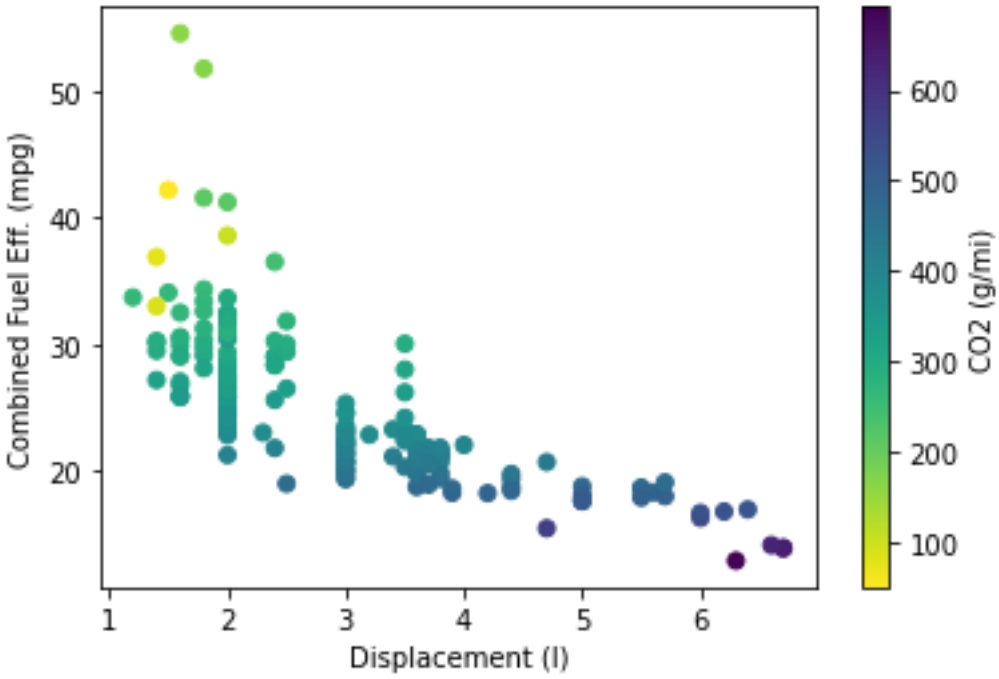

Figure 33: Non-Positional Encoding Example: Color #3.

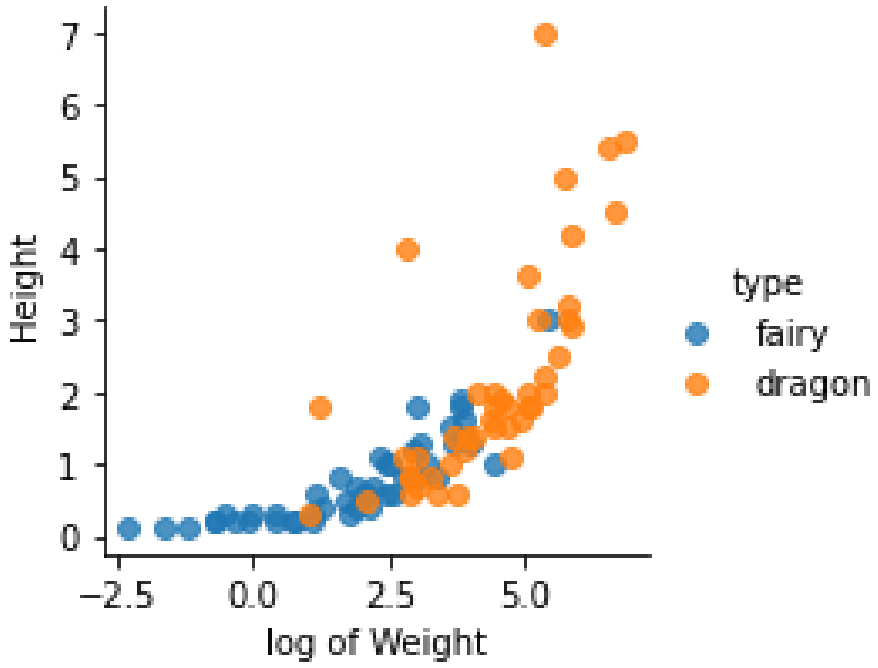

Figure 34: Pokemon Height vs. Weight by Category.

```
g = sb.FacetGrid(data = pkmn_types, hue = 'type', hue_order =
                                    ptypes)
g.map(sb.regplot, 'log_weight', 'height', fit_reg = False)
g.add_legend()
plt.xlabel('log of Weight')
plt.ylabel('Height')
# See Figure 34
```

With bivariate data, when we used faceting to create a series of plots, we used bar charts or histograms while changing a specification of the second variable. With three variables, we could use scatter plots comparing two variables with a third variable encoded. This is known as faceting in two directions.

Let's go back to our fuel economy dataset and compare vehicle class, transmission type and combined fuel efficiency.

```
g = sb.FacetGrid(data = fuel_econ, col = 'VClass', row = '
                                    trans_type')
g.map(plt.scatter, 'displ', 'comb')
# See Figure 35
```

Continuing the trend, here are some more adaptations of bivariate plots extended to multivariate plots.

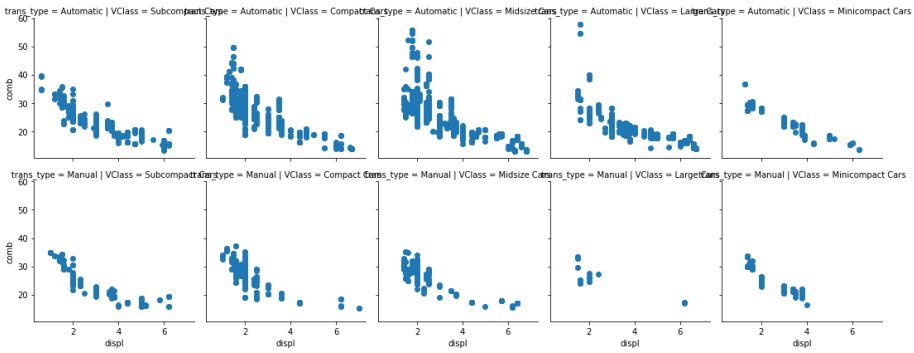

Figure 35: Basic Faceting in Two Directions.

```
sb.pointplot(data = fuel_econ, x = 'VClass', y = 'comb', hue = '
                                trans_type', ci = 'sd',
                                linestyles = '', dodge = True)
plt.xticks(rotation = 15)
plt.ylabel('Avg. Combined Efficiency (mpg)')
# See Figure 36

sb.barplot(data = fuel_econ, x = 'VClass', y = 'comb', hue = '
                                trans_type', ci = 'sd')
plt.xticks(rotation = 15)
plt.ylabel('Avg. Combined Efficiency (mpg)')
# See Figure 37

sb.boxplot(data = fuel_econ, x = 'VClass', y = 'comb', hue = '
                                trans_type')
plt.xticks(rotation = 15)
plt.ylabel('Avg. Combined Efficiency (mpg)')
# See Figure 38

# not illustrated, but feasible, is using a heatmap to illustrate
                                 a third variable
```

And a few more challenge examples:

```
# Plot the city vs highway fuel efficiencies for each vehicle
                                class
g = sb.FacetGrid(data = fuel_econ, col = 'VClass')
g.map(plt.scatter, 'city', 'highway')
# See Figure 39

# Plot the relationship between engine size (displ), class, and
                                fuel type (fuelType). Use premium
                                 vs. regular
sb.barplot(data = fuel_econ, x = 'VClass', y = 'displ', hue = '
                                fuelType', hue_order = ['Premium
                                Gasoline', 'Regular Gasoline'],
                                ci = 'sd')
plt.xticks(rotation = 15)
plt.ylabel('Engine Size (l)')
```

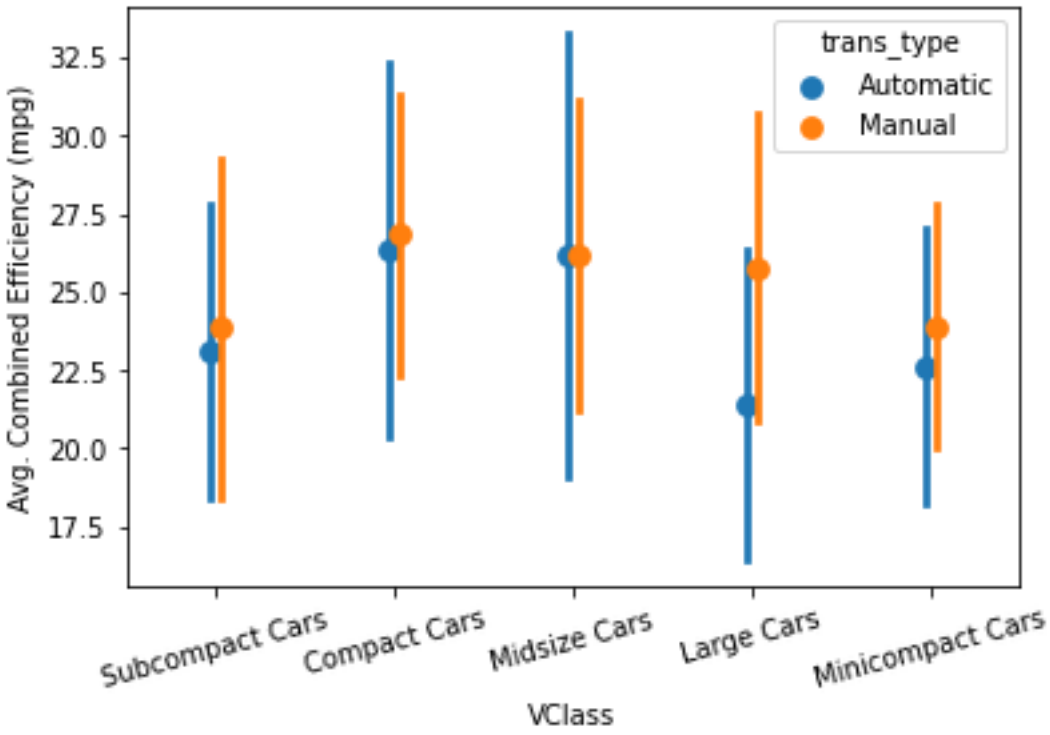

Figure 36: Pointplot for Multivariate.

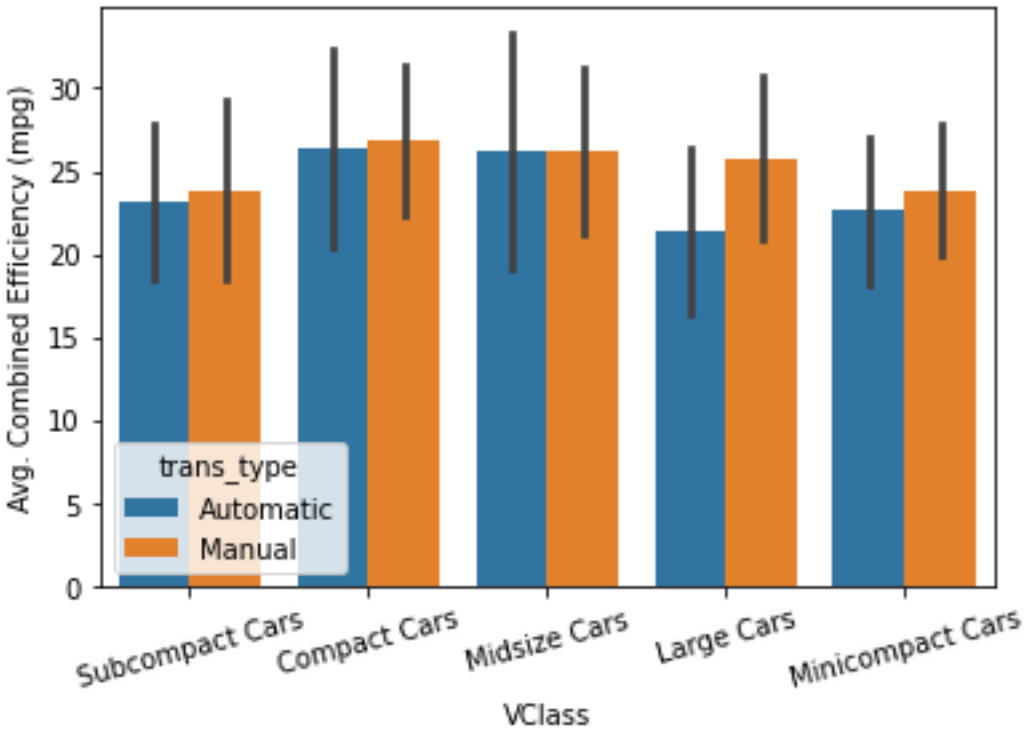

Figure 37: Bar Plot for Multivariate.

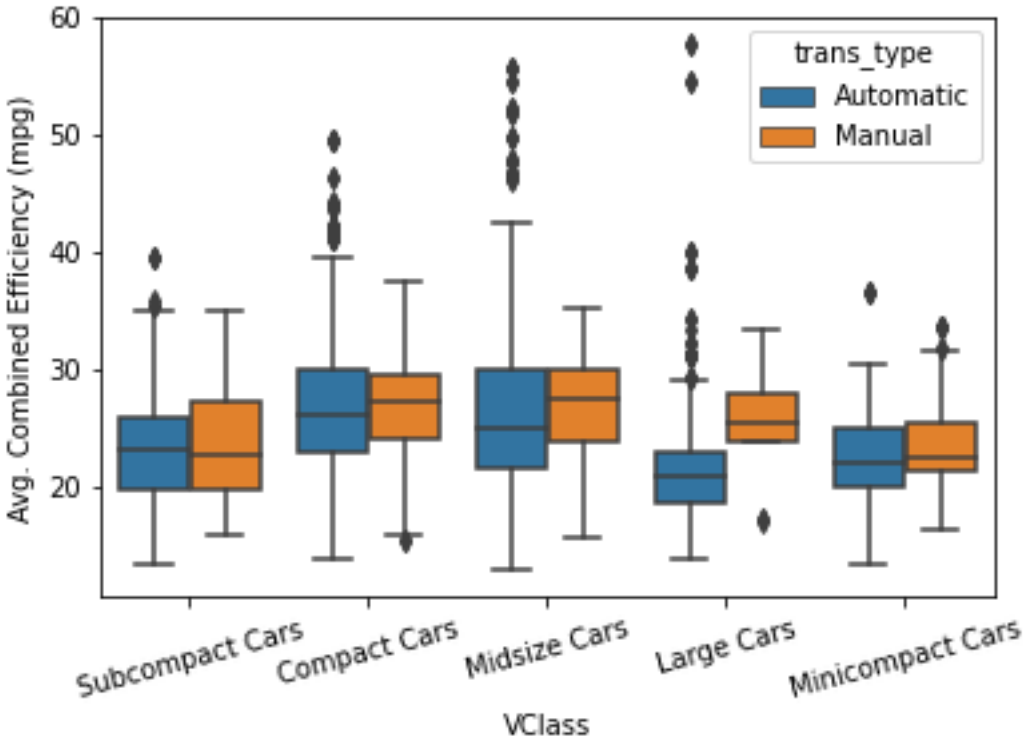

Figure 38: Box Plot for Multivariate.

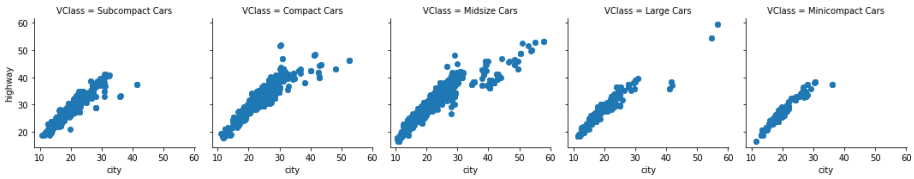

Figure 39: Vehicle Class vs. Combined Fuel Efficiency (by Transmission Type).

```
# See Figure 40
```

Possibly even more powerful than a few different combinations would be pairwise combinations between specified variables. This is the concept of plot matrices.

```
# let's look at the pairwise relationships between the 6 combat
                                    types
pkmn_stats = ['hp', 'attack', 'defense', 'speed', 'special-attack
                                    ', 'special-defense']
g = sb.PairGrid(data = pokemon, vars = pkmn_stats)
g.map(plt.scatter)
# See Figure 41

# if needed, we even impose another plot over the existing plots
                                    on the diagonal
g = g.map_offdiag(plt.scatter)
g.map_diag(plt.hist)
# See Figure 42
```

If we run a slight variaton of the code above, we'll actual just get the histogram along the diagonal, instead of imposed.

```
econ_vars = ['displ', 'co2', 'city', 'highway', 'comb']
g = sb.PairGrid(data = fuel_econ, vars = econ_vars)

"""
If we don't run:
g.map(plt.scatter)
Then, just the histograms will be drawn on the diagonals.
"""

g = g.map_offdiag(plt.scatter)
g.map_diag(plt.hist)
# See Figure 43
```

We'll draw the multivariate exploration to an end with one last concept, feature engineering. Let's work through a challenge problem to get an idea of how this can be used.

The output of the preceding task pointed out a potentially interesting relationship between co2 emissions and overall fuel efficiency. Engineer a new variable that depicts CO2 emissions as a function of gallons of gas (g / gal). (The 'co2'

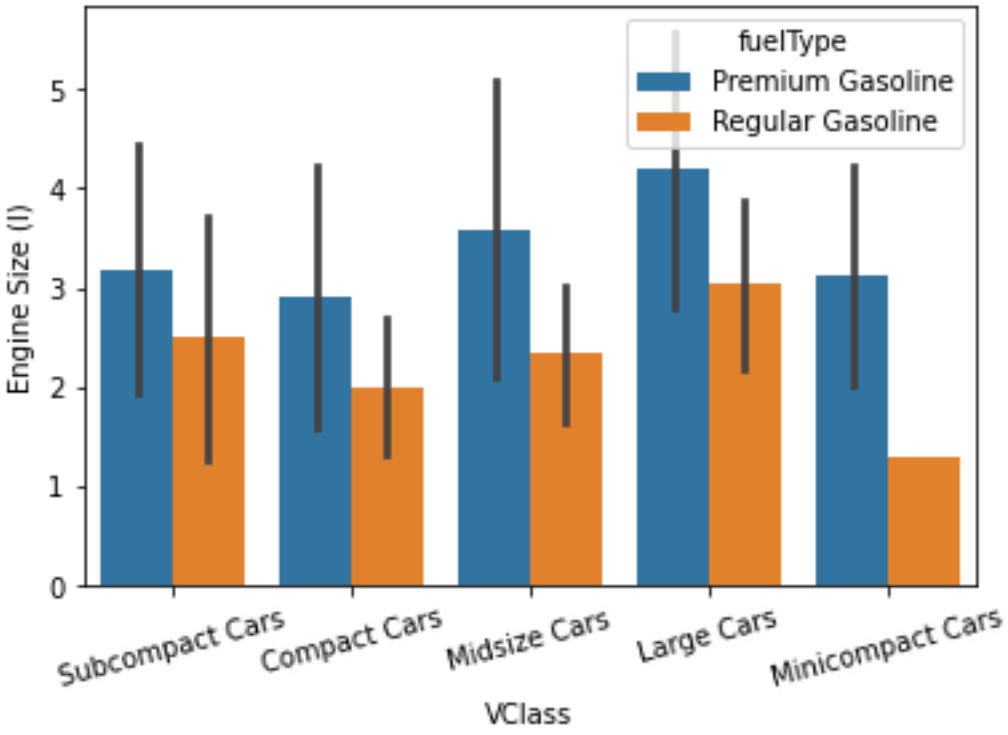

Figure 40: Vehicle Class vs. Engine Size (by Fuel Type).

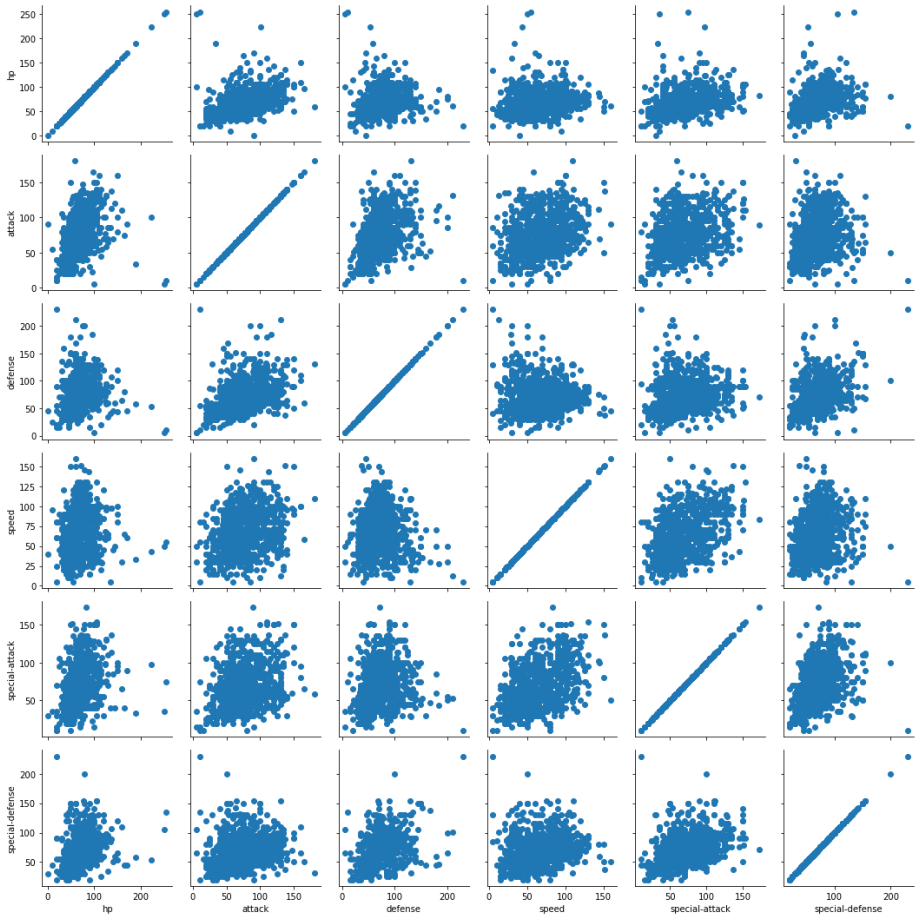

Figure 41: Pairwise Relationships.

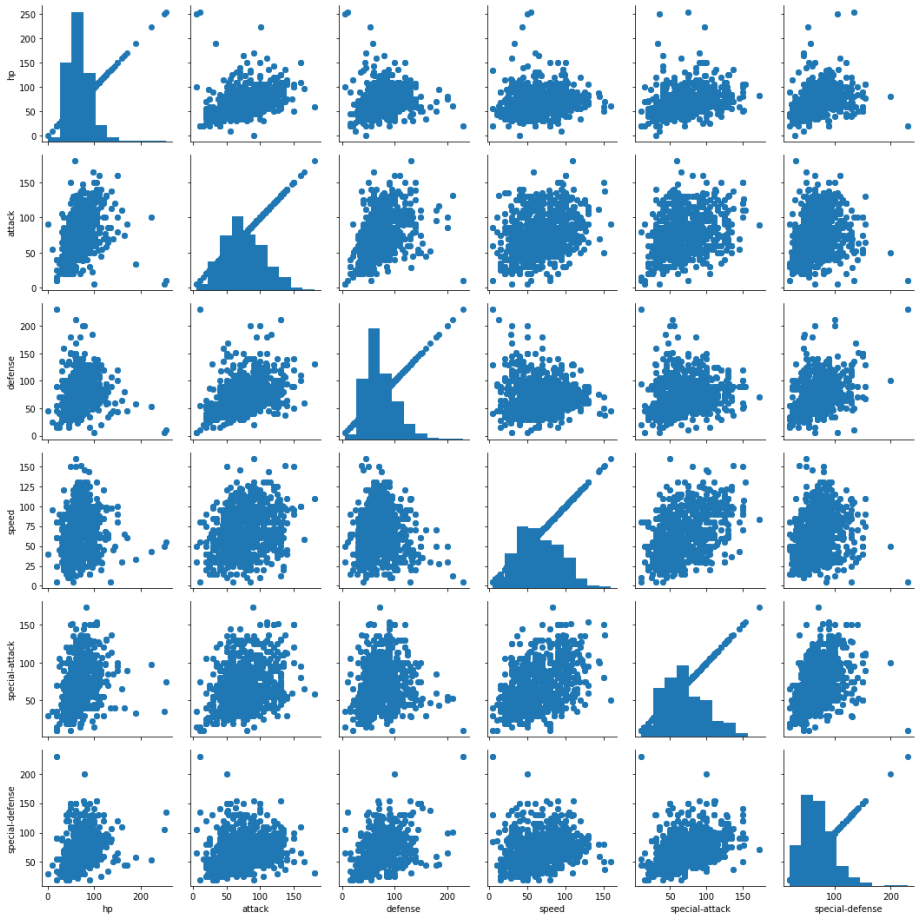

Figure 42: Histograms on the Diagonal.

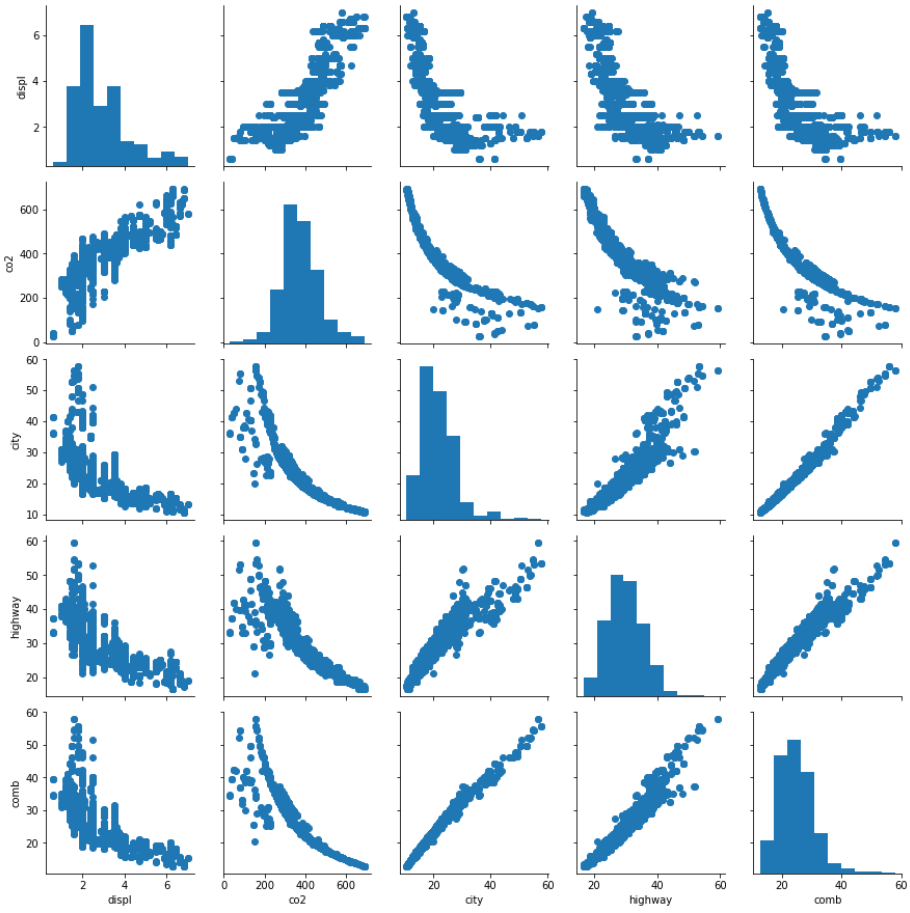

Figure 43: Histograms on the Diagonal.

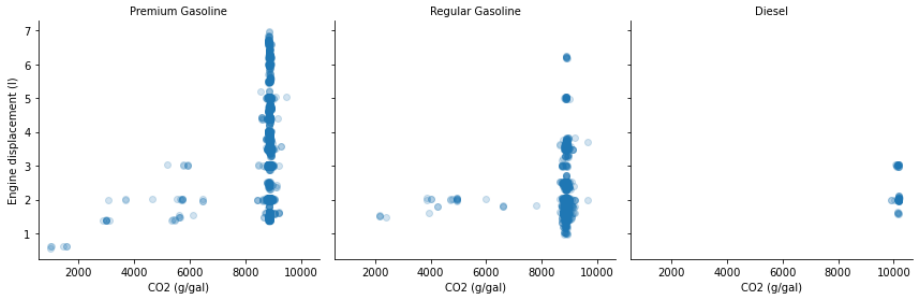

Figure 44: Feature Engineering.

variable is in units of g / mi, and the 'comb' variable is in units of mi / gal.) Then, plot this new emissions variable against engine size ('displ') and fuel type ('fuelType'). For this task, compare not just Premium Gasoline and Regular Gasoline, but also Diesel fuel.

```
fuel_econ['new_var'] = fuel_econ['co2'] * fuel_econ['comb']
fuel_econ_sub = fuel_econ.loc[fuel_econ['fuelType'].isin(['
                                Premium Gasoline', 'Regular
                                Gasoline', 'Diesel'])]

# plotting
g = sb.FacetGrid(data = fuel_econ_sub, col = 'fuelType', size = 4
                                , col_wrap = 3)
g.map(sb.regplot, 'new_var', 'displ', y_jitter = 0.04, fit_reg =
                                False, scatter_kws = {'alpha' : 1
                                /5})
g.set_ylabels('Engine displacement (l)')
g.set_xlabels('CO2 (g/gal)')
g.set_titles('{col_name}')
# See Figure 44
```

4 Explanatory Visualizations

We've examined the data using multiple methods, testing multiple ideas, and have formulated a story the data can tell. Now where do we go? Let's recall the entire data analysis process:

- Extract
- Clean
- Explore
- Analyze
- Share

We're in the endgame when it comes to the process, so here are some tips for telling a story and finalizing some visualizations.

Telling a Story:

1. Start with a Question
2. Use Repetition
3. Highlight the Answer
4. Call your Audience to Action

Polishing Plots:

- Appropriate Plot Type and Encodings
- Good Design Integrity
- Labeled Axes, Reasonable Tick Marks
- Descriptive Legend and Titles
- Accompanying Comments and Text

We'll use the pokemon dataset a final time to showcase a polished plot.

```
# We'll rerun the code to create variations for type and specify
                                fairy vs. dragon, with some
                                slight differences

type_cols = ['type_1','type_2']
non_type_cols = pokemon.columns.difference(type_cols)
pkmn_types = pokemon.melt(id_vars = non_type_cols, value_vars =
                                type_cols, var_name = 'type_level
                                ', value_name = 'type').dropna()
```

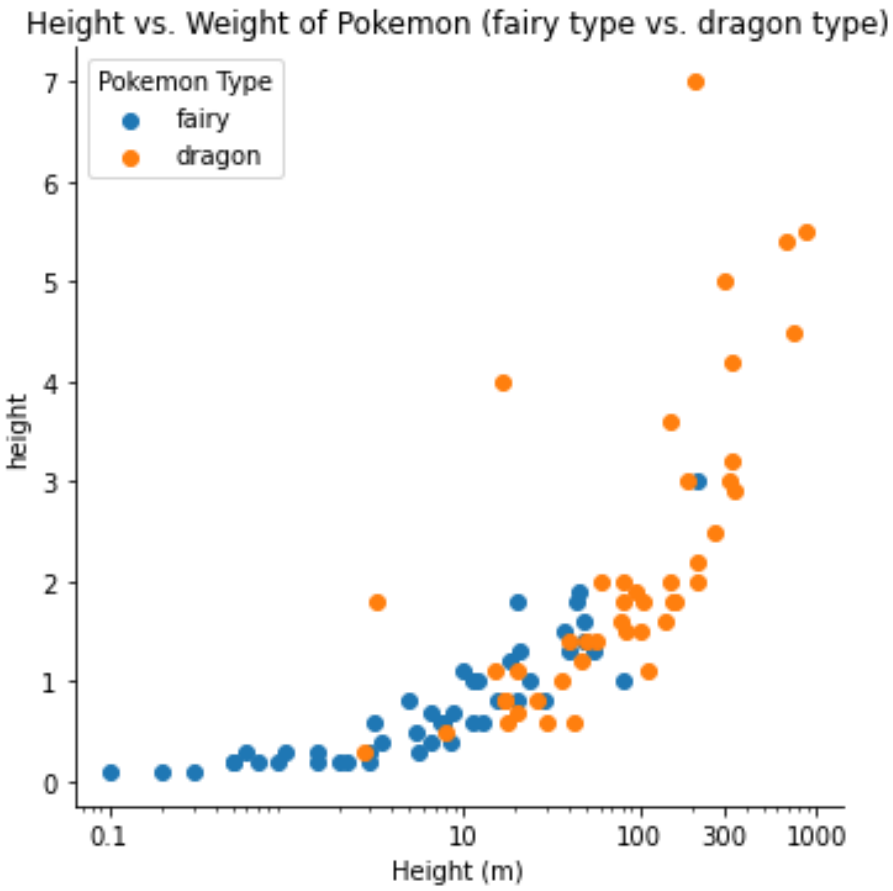

Figure 45: Polished Plot.

```
pokemon_sub = pkmn_types.loc[pkmn_types['type'].isin(['fairy','
                                dragon'])]

type_cols = ['type_1','type_2']
non_type_cols = pokemon.columns.difference(type_cols)
pkmn_types = pokemon.melt(id_vars = non_type_cols, value_vars =
                                type_cols,
var_name = 'type_level', value_name = 'type').dropna()

pokemon_sub = pkmn_types.loc[pkmn_types['type'].isin(['fairy','
                                dragon'])]
# See Figure 45
```

5 Visualization Case Study

We've built a solid toolkit for data visualization from the exploratory to the explanatory. We'll put all of this together in a final case study!

We're given a dataset containing entries on about 54,000 diamonds with 10 variables on each diamond observation.

- price
- carat
- cut
- color
- clarity
- x
- y
- z
- table
- depth

We haven't been given a specific task or goal with the provided dataset. So, let's just explore the variables and relationships.

```
diamonds = pd.read_csv("diamonds.csc")

# high-level overview of data shape and composition
diamonds.shape
diamonds.dtypes
diamonds.head(10)
diamonds.describe()
```

The exact shape of the diamonds dataset was (53940, 10).
diamonds.dtypes

	0
carat	float64
cut	object
color	object
clarity	object
depth	float64
table	float64
price	int64
x	float64
y	float64
z	float64

diamonds.head(10)

	0
carat	float64
cut	object
color	object
clarity	object
depth	float64
table	float64
price	int64
x	float64
y	float64
z	float64

diamonds.describe

	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	0.000000
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	2.910000
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	3.530000
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	4.040000
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	31.800000

After the initial look through the dataset, we'll begin with some univariate exploration:

```
# Let's start by looking at price... is distribution skewed /
                                symmetric? Unimodal or multimodal
                                ?
base_color = sb.color_palette()[0]
plt.hist(data = diamonds, x = 'price', bins = 150)
plt.xscale('log')
# See Figure 46

# Look at carat
plt.hist(data = diamonds, x = 'carat', bins = 50)
plt.xlim((0, 4))
# See Figure 47

# Look at cut, color, and clarity grades (categorical variables)

# cut
cut_order = diamonds['cut'].value_counts().index
sb.countplot(data = diamonds, x = 'cut', color = base_color,
                                order = cut_order)
# See Figure 48

# color
color_order = diamonds['color'].value_counts().index
sb.countplot(data = diamonds, x = 'color', color = base_color,
                                order = color_order)
# See Figure 49

# clarity
clarity_order = diamonds['clarity'].value_counts().index
sb.countplot(data = diamonds, x = 'clarity', color = base_color,
                                order = clarity_order)
# See Figure 50

# Include general categorical variable comment
# Include distribution comment about each category
```

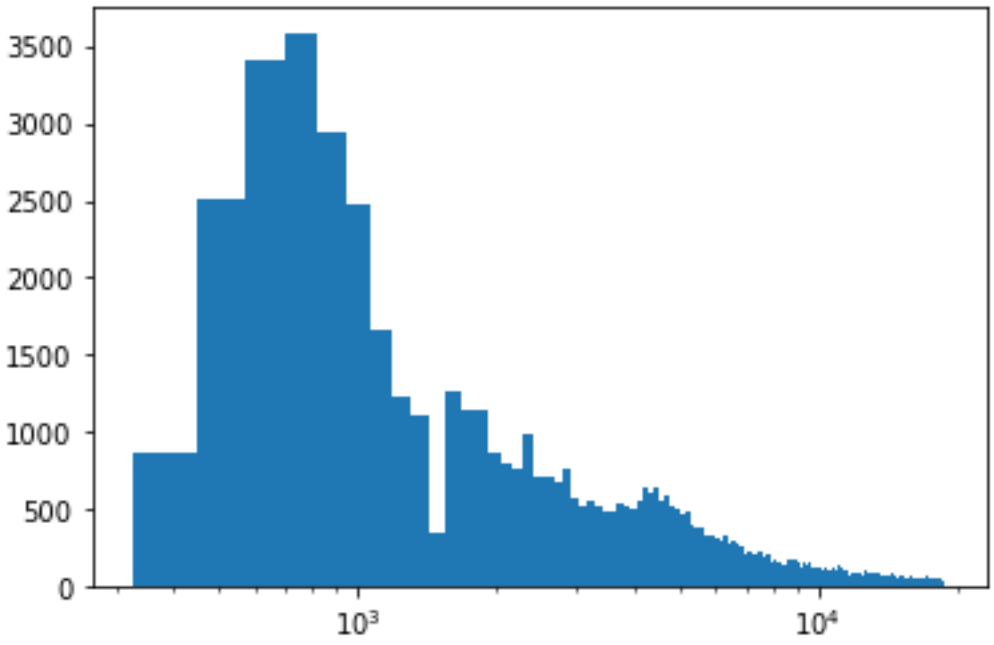

Figure 46: Price Histogram.

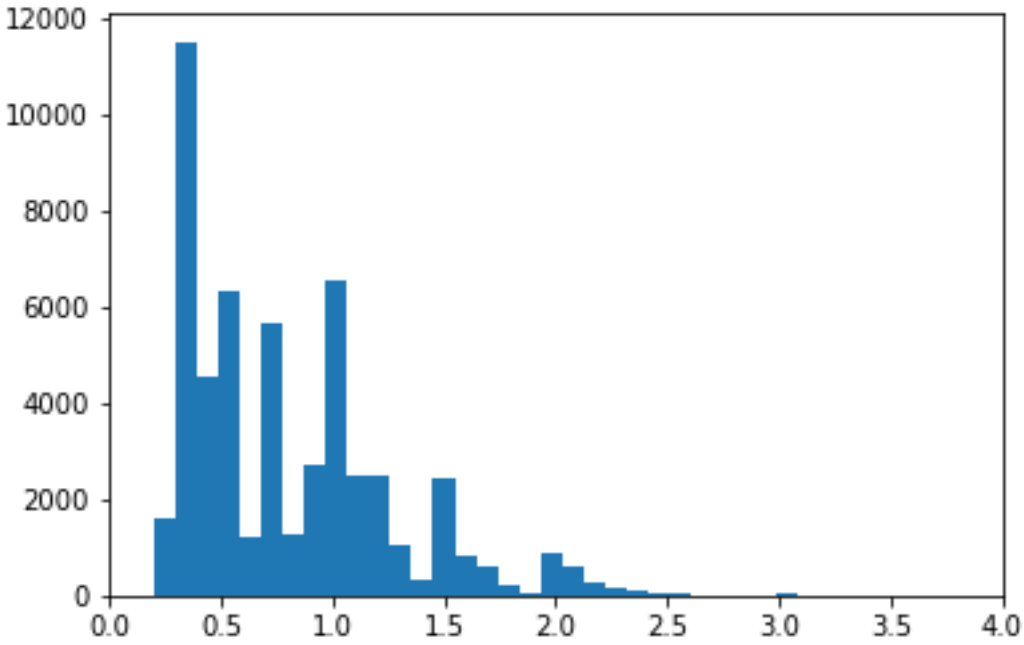

Figure 47: Carat Histogram.

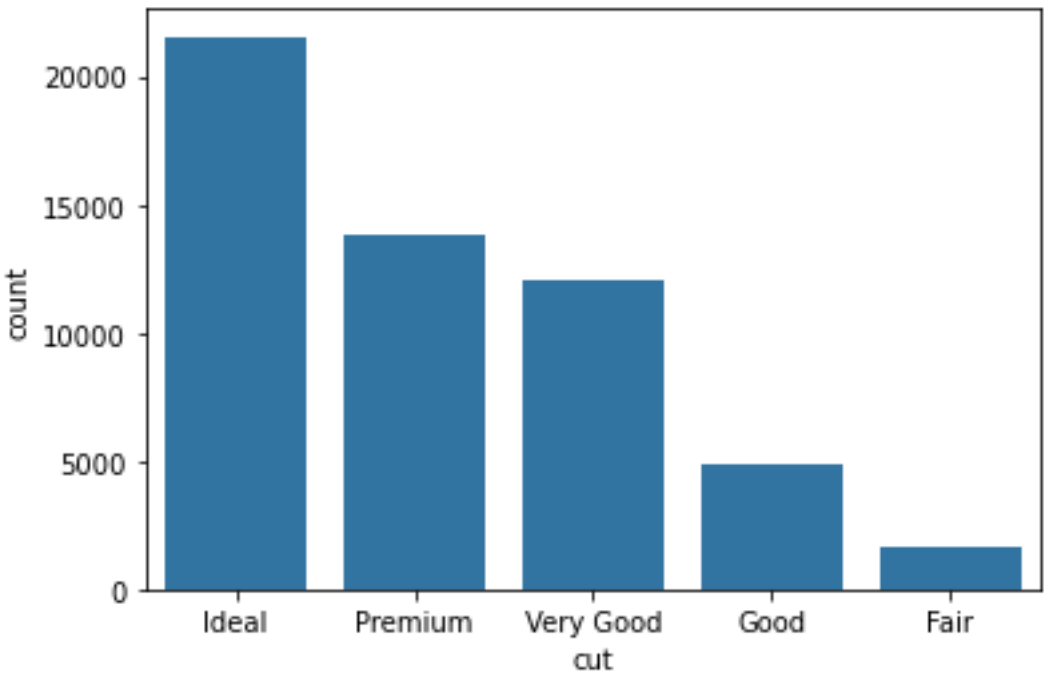

Figure 48: Cut Bar Plot.

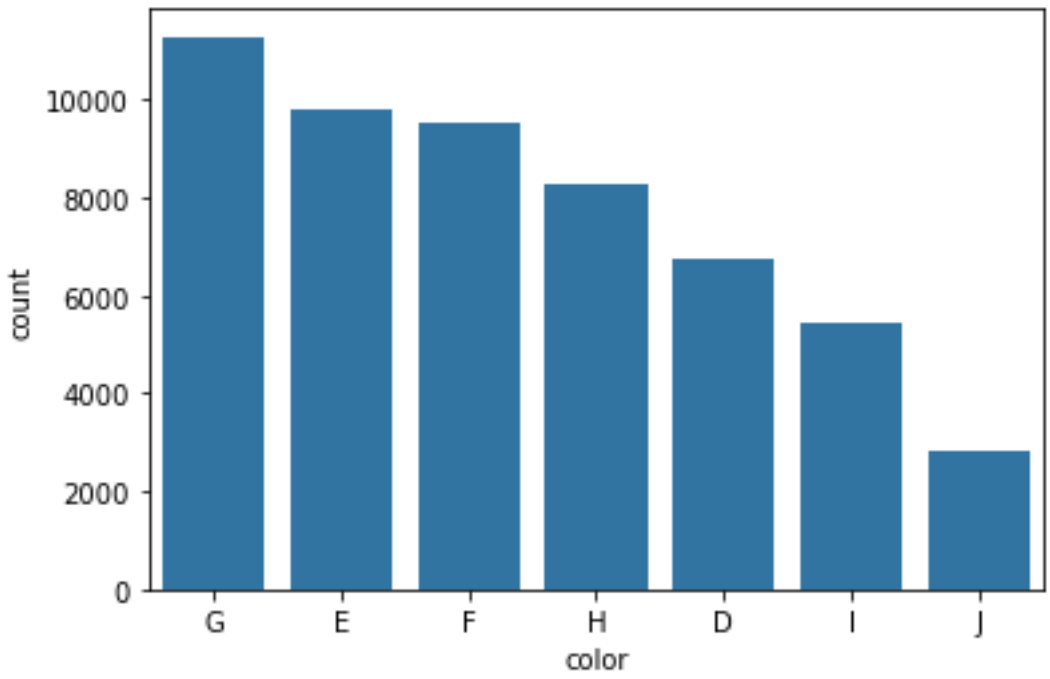

Figure 49: Color Bar Plot.

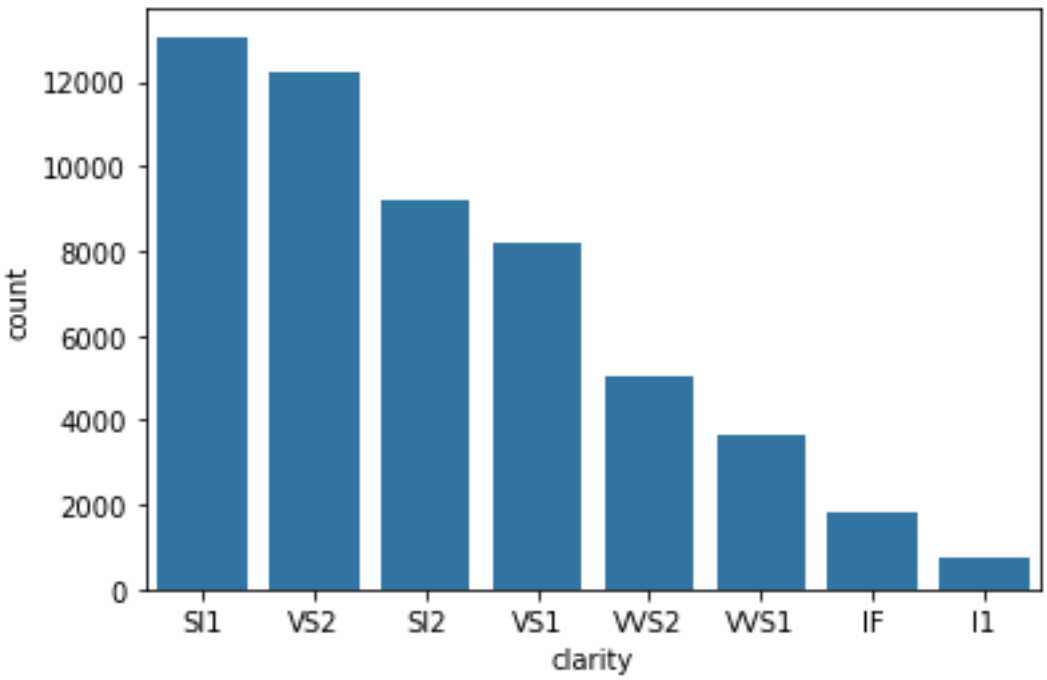

Figure 50: Clarity Bar Plot.

Next, we'll move into bivariate exploration.

```
# Step 2: Bivariate Exploration
# convert cut, color, and clarity into ordered categorical types
ordinal_var_dict = {'cut': ['Fair','Good','Very Good','Premium','
                                Ideal'],
  'color': ['J', 'I', 'H', 'G', 'F', 'E', 'D'],
  'clarity': ['I1', 'SI2', 'SI1', 'VS2', 'VS1', 'VVS2', 'VVS1', '
                                IF']}


for var in ordinal_var_dict:
  pd_ver = pd.__version__.split(".")
  if (int(pd_ver[0]) > 0) or (int(pd_ver[1]) >= 21): # v0.21 or
                                later
    ordered_var = pd.api.types.CategoricalDtype(ordered = True,
    categories = ordinal_var_dict[var])
    diamonds[var] = diamonds[var].astype(ordered_var)
  else: # pre-v0.21
    diamonds[var] = diamonds[var].astype('category', ordered =
                                True,
    categories = ordinal_var_dict[var])


# Compare Price vs. Carat
np.random.seed(0)
diamonds_sample = np.random.choice(diamonds.shape[0], 10000,
                                replace = False)
diamonds_subset = diamonds.loc[diamonds_sample]
plt.scatter(data = diamonds_subset, x = 'carat', y = 'price',
                                alpha = 1/3)
plt.yscale('log')
# See Figure 51

# Still Price vs. Carat, but we'll use some more transformations
diamonds['log_price'] = np.log(diamonds['price'])
diamonds['cube_carat'] = diamonds['carat'] ** (1/3)
plt.scatter(data = diamonds, x = 'cube_carat', y = 'log_price',
                                alpha = 1/10)
# See Figure 52

# Let's compare price and the categorical variables
"""
sb.boxplot(data = diamonds, x = 'cut', y = 'price', color =
                                base_color, order = cut_order)
sb.boxplot(data = diamonds, x = 'color', y = 'price', color =
                                base_color, order = color_order)
sb.boxplot(data = diamonds, x = 'clarity', y = 'price', color =
                                base_color, order = clarity_order
                                )

Which can all be done with the following commands:
"""

category_list = ['cut', 'color', 'clarity']
non_category_list = diamonds.columns.difference(category_list)
diamonds_c = diamonds.melt(id_vars = non_category_list,
                                value_vars = category_list,
```

```
                                var_name = 'main_category',
                                value_name = 'main_value')

sb.catplot(data = diamonds_c, y = 'price', x = 'main_value', col
                                = 'main_category', kind = 'box',
                                sharex= False, color = base_color
                                )
# See Figure 53

"""
Then, we'll check out the data after a transformation:

sb.boxplot(data = diamonds, x = 'cut', y = 'log_price', color =
                                base_color, order = cut_order)
sb.boxplot(data = diamonds, x = 'color', y = 'log_price', color =
                                 base_color, order = color_order)
sb.boxplot(data = diamonds, x = 'clarity', y = 'log_price', color
                                 = base_color, order =
                                clarity_order)

Which can all be done with the following commands:
"""


sb.catplot(data = diamonds_c, y = 'log_price', x = 'main_value',
                                col = 'main_category', kind = '
                                box', sharex= False, color =
                                base_color)
# See Figure 54

"""
With a violin plot, you can get more insight into what causes the
                                 trend in median prices to appear
                                 as it does. Faceted histograms
                                will also produce a similar
                                result, though unless the
                                faceting keeps the price axis
                                common across facets, the trend
                                will be harder to see. For each
                                ordinal variable, there are
                                multiple modes into which prices
                                appear to fall. Going across
                                increasing quality levels, you
                                should see that the modes rise in
                                 price - this should be the
                                expected effect of quality.
                                However, you should also see that
                                 more of the data will be located
                                 in the lower-priced modes - this
                                 explains the unintuitive result
                                noted in the previous comment.
                                This is clearest in the clarity
                                variable. Let's keep searching
                                the data to see if there's more
                                we can say about this pattern.
"""
```

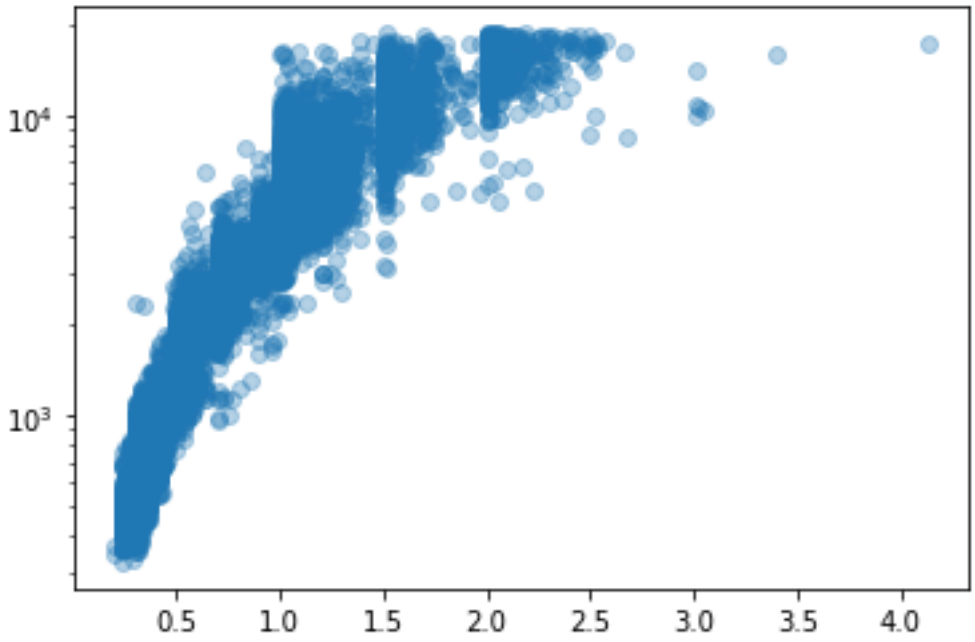

Figure 51: Price vs. Carat.

```
"""

Let's combine the boxplot images and the boxplot images using a
                                combination of the methods we've
                                seen:
"""
sb.violinplot(data = diamonds, x = 'cut', y = 'price', color =
                                base_color, inner = None)
sb.violinplot(data = diamonds, x = 'color', y = 'price', color =
                                base_color, inner = None)
sb.violinplot(data = diamonds, x = 'clarity', y = 'price', color
                                = base_color, inner = None)

sb.violinplot(data = diamonds, x = 'cut', y = 'carat', color =
                                base_color, inner = None)
sb.violinplot(data = diamonds, x = 'color', y = 'carat', color =
                                base_color, inner = None)
sb.violinplot(data = diamonds, x = 'clarity', y = 'carat', color
                                = base_color, inner = None)
"""
```

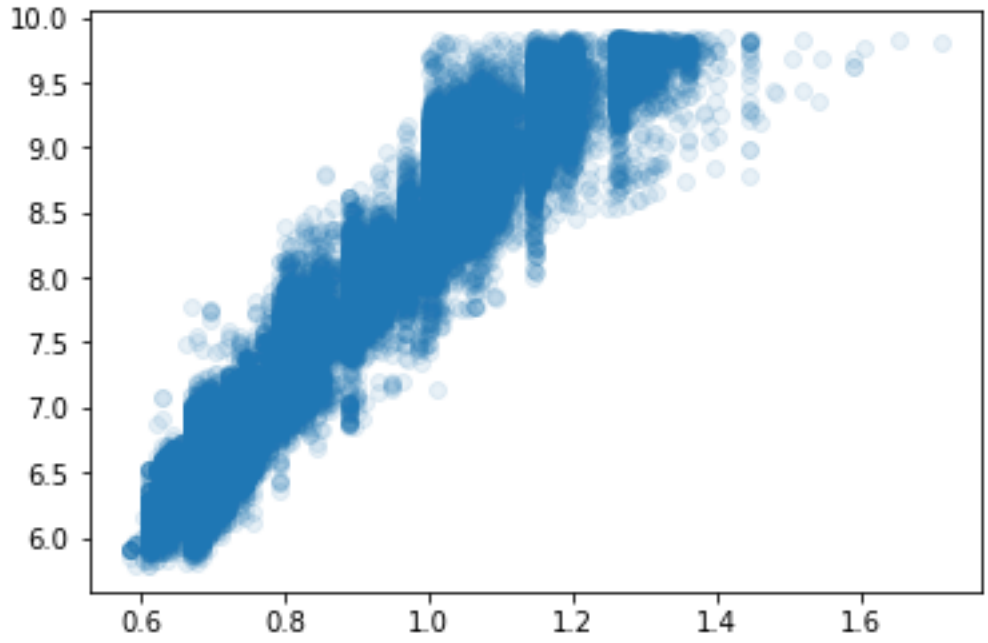

Figure 52: Price vs. Carat with Transformations.

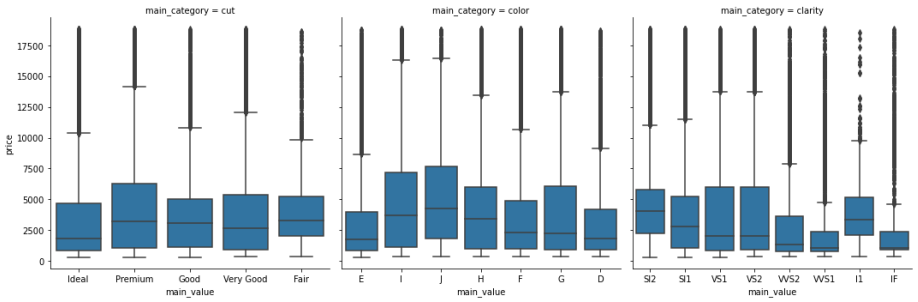

Figure 53: Box Plots Price vs. Category.

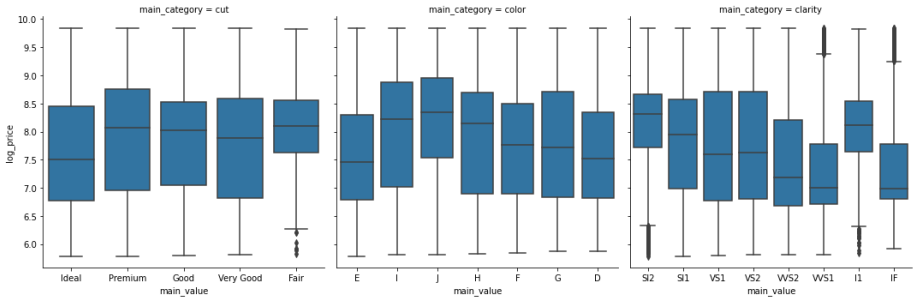

Figure 54: Box Plots Price vs. Category.