I2SL_Exploratory_Data_Analysis

December 14, 2023

1 Exploratory Data Analysis

In this lesson, we look at statistical and graphical techniques that summarize their main characteristics of one, two variable, and multi-variable data sets. To find relationships amongst variables and to find the variables which are most interesting for a particular analysis task.

Rationale: Exploratory data analysis helps one understand the data, to form and change new theories, and decide which techniques are appropriate for analysis. After a model is finished, exploratory data analysis can look for patterns in these data that may have been missed by the original hypothesis tests. Successful exploratory analyses help the researcher modify theories and refine the analysis.

1.1 Topics

- Exploratory Data Analysis
- Types of Variables
- Descriptive Statistics
- Measures of Central Tendency
- Measures of spread
- Summary Statistics
- Univariate Data Analysis
- Box-Plots
- Bar charts
- Histograms
- Line plots
- Multivariate Data Analysis
- Aesthetic mappings
- Faceting
- Position Adjustments
- Scatter plots
  - Scatter Plot with No apparent relationship
  - Scatter Plot with Linear relationship
  - Scatter Plot with Regression Lines
  - Scatter Plot with Quadratic relationship
  - Scatter plot with Homoscedastic relationship
  - Scatter plot with Jittering
- QQplot

1.2 Exploratory Data Analysis

In statistics,exploratory data analysis (EDA) is an approach to analyzing data sets to summarize their main characteristics, often with visual methods concepts apply to statistics and to graphical methods. EDA is for seeing what the data can tell us before the formal modeling or hypothesis testing task.

- from Exploratory Data Analysis - Wikipedia)

Early forms of Exploratory Data Analysis such as the box plot are often attributed to John Tukey (1970s)

John Tukey

John Wilder Tukey (June 16, 1915 - July 26, 2000) was an American mathematician best known for development of the FFT algorithm and box plot.The Tukey range test, the Tukey lambda distribution, the Tukey test all bear his name.

- from John Tukey - Wikipedia)

Goals of Exploratory Data Analysis

- get a general sense of the data

- data-driven (model-free)
- visual (Humans are great pattern recognizers)
- test assumptions (e.g. normal distributions or skewed?)
- identify useful raw data & transforms (e.g. log(x))
- distributions (symmetric, normal, skewed)
- data quality problems
- outliers
- correlations and inter-relationships
- subsets of interest
- suggest functional relationships

1.3 Types of Variables

Continuous variables

Always numeric Integers - a whole number; a number that is not a fraction. e.g. -1,0,1,2,3,4, …

Floating point numbers (a rational number or a 'float') - a rational number is any number that can be expressed as the quotient or fraction p/q of two integers, p and q, with the denominator q not equal to zero. Since q may be equal to 1, every integer is a rational number. e.g. -1.3, 0.0, 3.14159265359, 2.71828, …

Continuous variables can be any number, positive or negative

Examples: age in years, weight, website 'hits' and other measurements

Categorical variables A categorical variable is a variable that can take on one of a limited, and usually fixed, number of possible values.

Types of categorical variables are ordinal, nominal and dichotomous (binary)

1.3.1 Nominal Variables

Nominal variable is a categorical variable without an intrinsic order

Examples of nominal variables: Where a person lives in the U.S. (Northeast, South, Midwest, etc.) Sex (male, female) Nationality (American, Mexican, French) Race/ethnicity (African American, Hispanic, White, Asian American)

Enum	*Nationality* ← nominal
1	American
2	Mexican
3	French
4	Brasilian

Ordinal Variables

Ordinal variables are categorical variable with some intrinsic order or numeric value

Examples of ordinal variables: Education (no high school degree, HS degree, some college, college degree) Agreement (strongly disagree, disagree, neutral, agree, strongly agree) Rating (excellent, good, fair, poor) Any other scale (e.g. On a scale of 1 to 5)

Rank	*Degree* ← ordinal
1	PhD
2	Master's
3	Bachelors
4	Associate's
5	High School

Dichotomous Variables

Dichotomous (or binary) (or boolean) variables – a categorical variable with only 2 levels of categories * yes or no * true or false * accept or reject * pass or fail

1.4 Descriptive Statistics

Descriptive statistics is the discipline of quantitatively describing the main features of a collection of data. Common descriptive measures used are measures of central tendency and measures of variability or dispersion or spread.

Measures of central tendency

Measures of central tendency are used to describe the most typical measure.

Mode: the value in a string of numbers that occurs most often Median: the value whose occurrence lies in the middle of a set of ordered values

Mean: sometimes referred to as the arithmetic mean. It is the average value characterizing a set of numbers

1.4.1 Mode

- the value that occurs most frequently
- used w/ nominal data
- there can be "ties"

1.4.2 Median

- score at the center of the distribution
- sort and take the middle value (or average or two middle values)
- determine w/ : $\frac{N+1}{2}$
- equal to 50th percentile

1.4.3 Mean (or arithmetic mean)

- $\bar{X} = \frac{\sum X}{N}$
- can be misleading when there are large outliers

1.4.4 Properties of the Measures of Center

1. adding or subtracting a constant does the same to the measure of center
2. multiplying or dividing by a constant does the same to the measure of center

Measures of Spread

Measures of variability aare used to reveal the typical difference between the values in a set of values

Measures of Spread
* Range, Quartile 2nd Quartile is the median * Quartiles: sort and divide in 4 parts. * Frequency distribution reveals the number (percent) of occurrences of each number or set of numbers * Range identifies the maximum and minimum values in a set of numbers * Standard deviation (or variance) indicates the degree of variation

1.5 Summary statistics

In descriptive statistics, summary statistics are used to summarize a set of observations, in order to communicate the largest amount of information as simply as possible. Statisticians commonly try to describe the observations in:

- a measure of location, or central tendency, such as the arithmetic mean
- a measure of statistical dispersion like the standard deviation
- a measure of the shape of the distribution like skewness or kurtosis
- if more than one variable is measured, a measure of statistical dependence such as a correlation coefficient

from Summary statistics - Wikipedia

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas.testing as tm
```

```
from scipy import stats
import seaborn as sns

# Make plots larger
plt.rcParams['figure.figsize'] = (15, 9)
```

[ ]:
```
tips = sns.load_dataset("tips")
tips.head()
```

[ ]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

[ ]:
```
tips.describe()
```

[ ]:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
std	8.902412	1.383638	0.951100
min	3.070000	1.000000	1.000000
25%	13.347500	2.000000	2.000000
50%	17.795000	2.900000	2.000000
75%	24.127500	3.562500	3.000000
max	50.810000	10.000000	6.000000

[ ]:
```
tips.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.3 KB
```

[ ]:
```
tips.shape
```

[ ]: (244, 7)

[ ]:
```
tips.time
```

[ ]:
```
0      Dinner
1      Dinner
2      Dinner
3      Dinner
4      Dinner
        …
239    Dinner
240    Dinner
241    Dinner
242    Dinner
243    Dinner
Name: time, Length: 244, dtype: category
Categories (2, object): ['Lunch', 'Dinner']
```

[ ]:
```
tips['time']
```

[ ]:
```
0      Dinner
1      Dinner
2      Dinner
3      Dinner
4      Dinner
        …
239    Dinner
240    Dinner
241    Dinner
242    Dinner
243    Dinner
Name: time, Length: 244, dtype: category
Categories (2, object): ['Lunch', 'Dinner']
```

[ ]:
```
tips['time'].value_counts()
```

[ ]:
```
Dinner    176
Lunch      68
Name: time, dtype: int64
```

[ ]:
```
tips[['time', 'tip']].head()
```

[ ]:

	time	tip
0	Dinner	1.01
1	Dinner	1.66
2	Dinner	3.50
3	Dinner	3.31
4	Dinner	3.61

[ ]:
```
%matplotlib inline
np.random.seed(sum(map(ord, "distributions")))
```

1.6 Histograms

A histogram is like a bar chart but with continuous variables. To group with divide a continuous variable into intervals called *bins* then count the number of cases within each bin.

- use bars to reflect counts
- intervals on the horizontal axis
- counts on the vertical axis

A histogram is a form of density estimation. That is, the construction of an estimate, based on observed data, of an unobservable underlying probability density function.

Histogram and density plots show the distribution of a single variable.

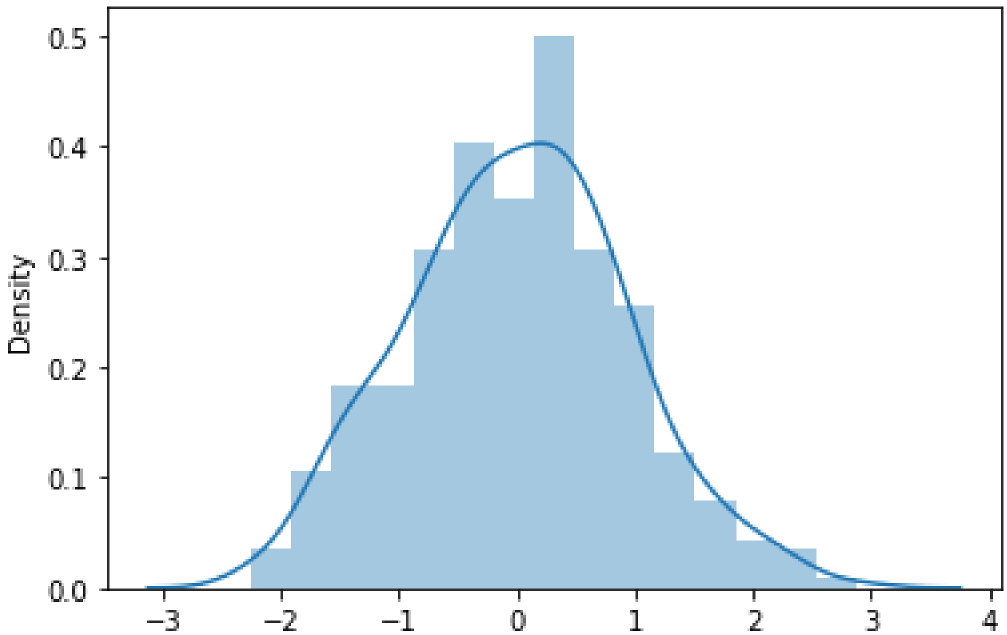
1.7 Density plots

You can use probability densities instead of (or with) frequencies for density estimation

[ ]:
```
x = np.random.normal(size=333)
sns.distplot(x)
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/distributions.py:2551:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

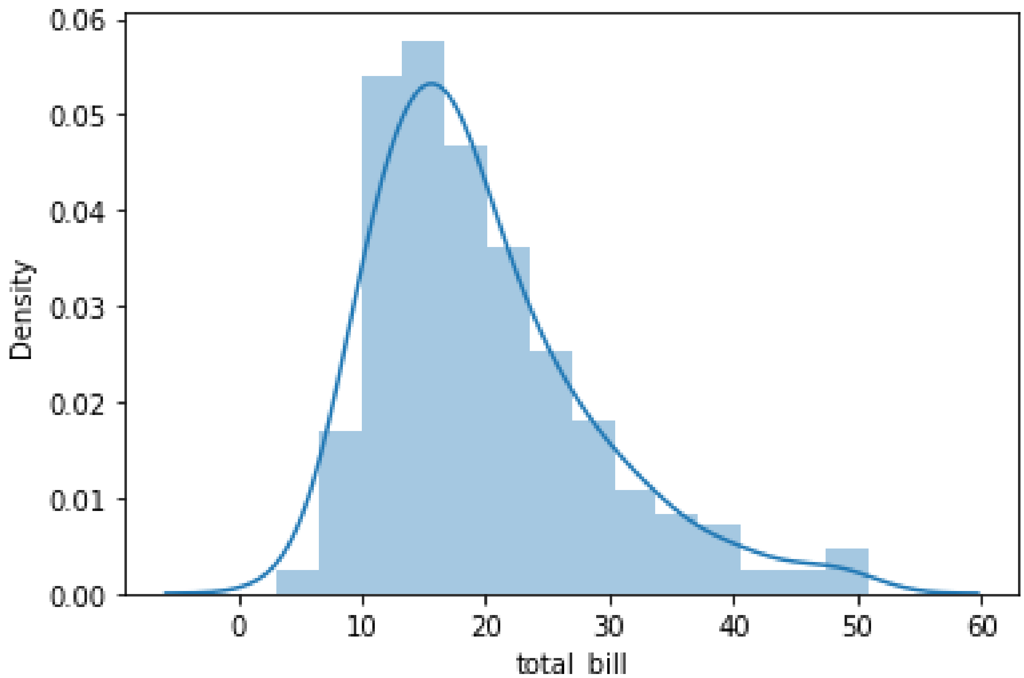
[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f35cd3f2710>
```



[ ]:
```
sns.distplot(tips['total_bill'])
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/distributions.py:2551:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd46b518>



[ ]:
```
titanic= sns.load_dataset('titanic')
titanic.describe()
```

[ ]:

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

[ ]:
```
titanic.head()
```

[ ]:

	survived	pclass	sex	age	…	deck	embark_town	alive	alone
0	0	3	male	22.0	…	NaN	Southampton	no	False
1	1	1	female	38.0	…	C	Cherbourg	yes	False
2	1	3	female	26.0	…	NaN	Southampton	yes	True
3	1	1	female	35.0	…	C	Southampton	yes	False
4	0	3	male	35.0	…	NaN	Southampton	no	True

[5 rows x 15 columns]

[ ]:
```
iris = sns.load_dataset('iris')
iris.describe()
```

[ ]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

[ ]:
```
iris.head()
```

[ ]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

1.7.1 Excercise

Plot the distributions for the data *'tips'* columns *tip* and *size*

1.8 Plotting with categorical data

1.9 Univariate Data Analysis

Univariate data analysis-explores each variable in a data set separately. This serves as a good method to check the quality of the data on a variable by variable basis. See Wikipedia Univariate analysis

1.10 Five point summary

Five point summary (min, Q1,Q2,Q3, max)

- the sample minimum (smallest observation)
- the lower quartile or first quartile
- the median (middle value)
- the upper quartile or third quartile
- the sample maximum (largest observation)

1.11 Box-Plot

A Box-Plot is a visual representation of a five point summary, with some additional information about outliers (1.5 times the lower and upper quartiles)

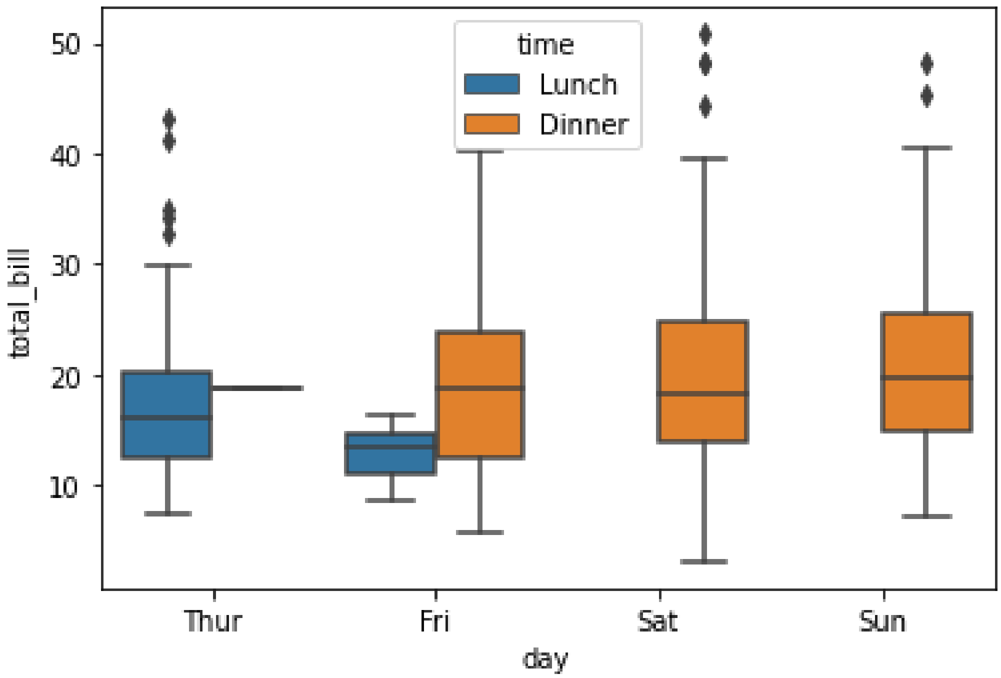
Box-Plot ($<$ 1.5 times Q1 outliers, expected min, Q1,Q2,Q3, expected max, outliers $>$ 1.5 times Q3)

1.11.1 Excercise

Interpret the categorical plots below

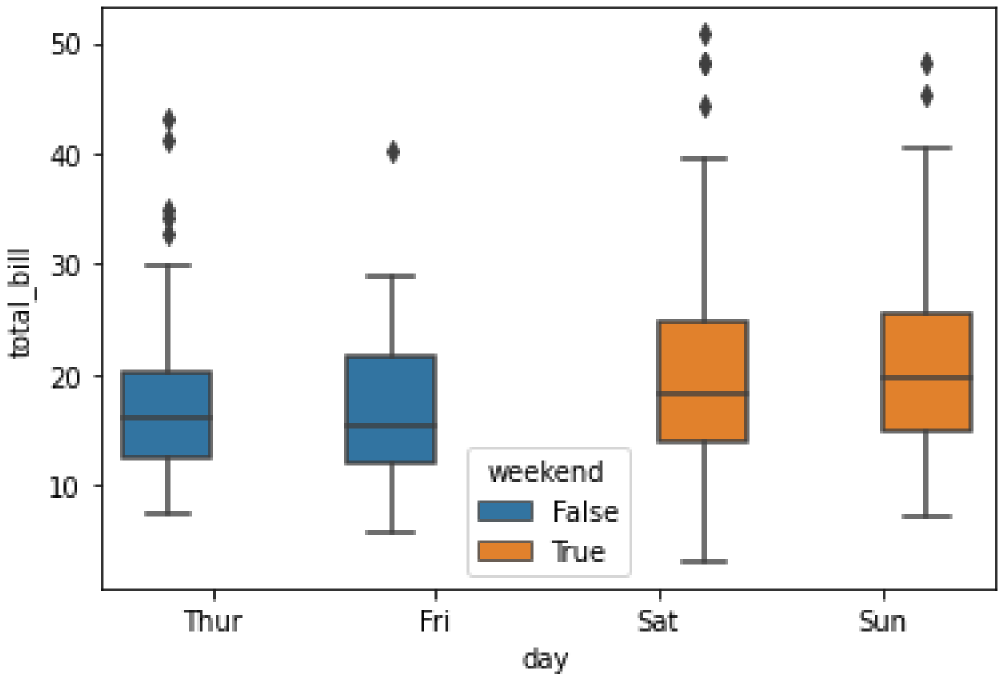
[ ]:
```
sns.boxplot(x="day", y="total_bill", hue="time", data=tips)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd28c198>



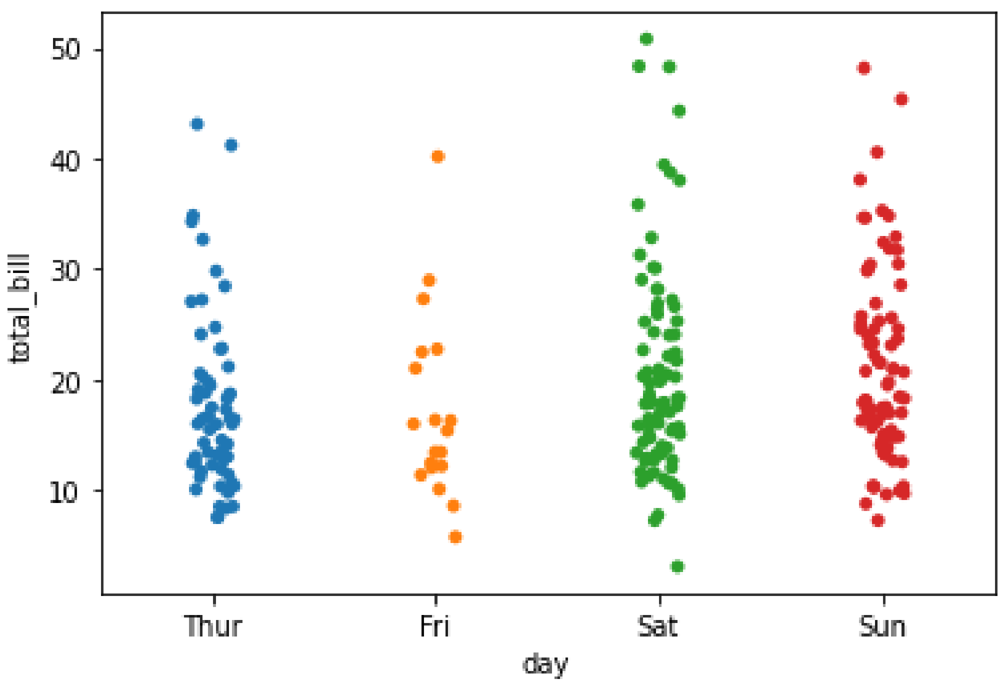
[ ]:
```
tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
sns.boxplot(x="day", y="total_bill", hue="weekend", data=tips)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd1e3c18>



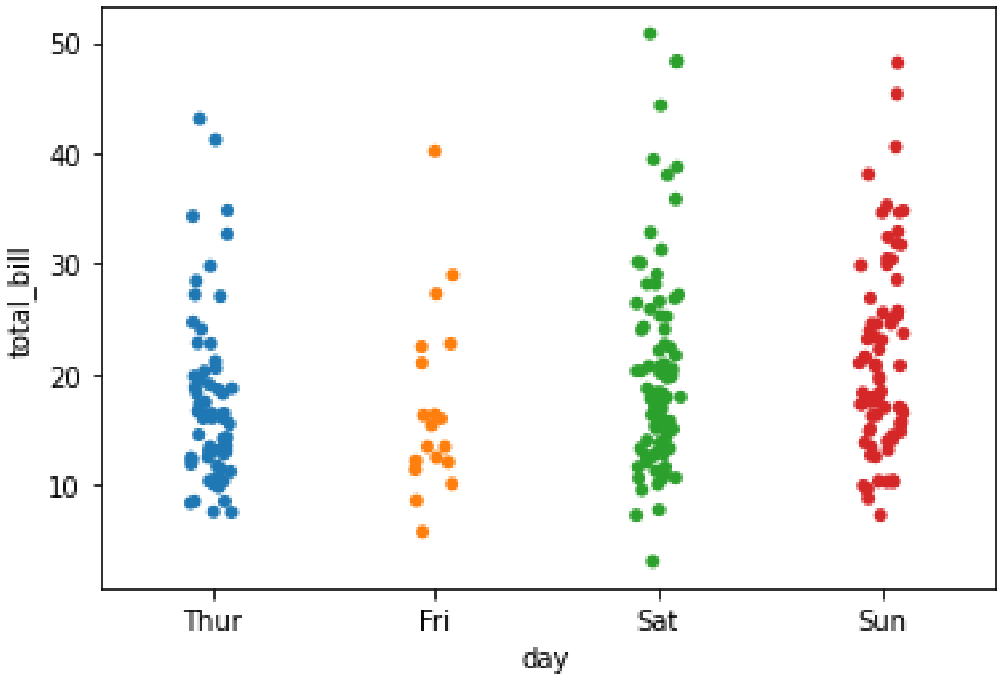
[ ]:
```
sns.stripplot(x="day", y="total_bill", data=tips)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd1837f0>



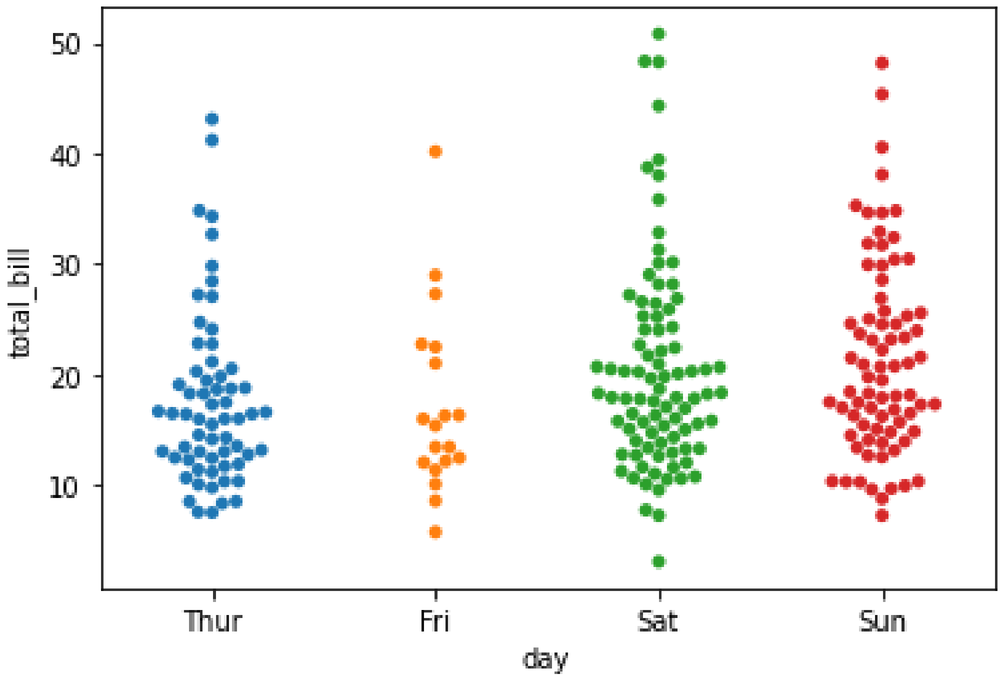
[ ]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=True)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd176b38>



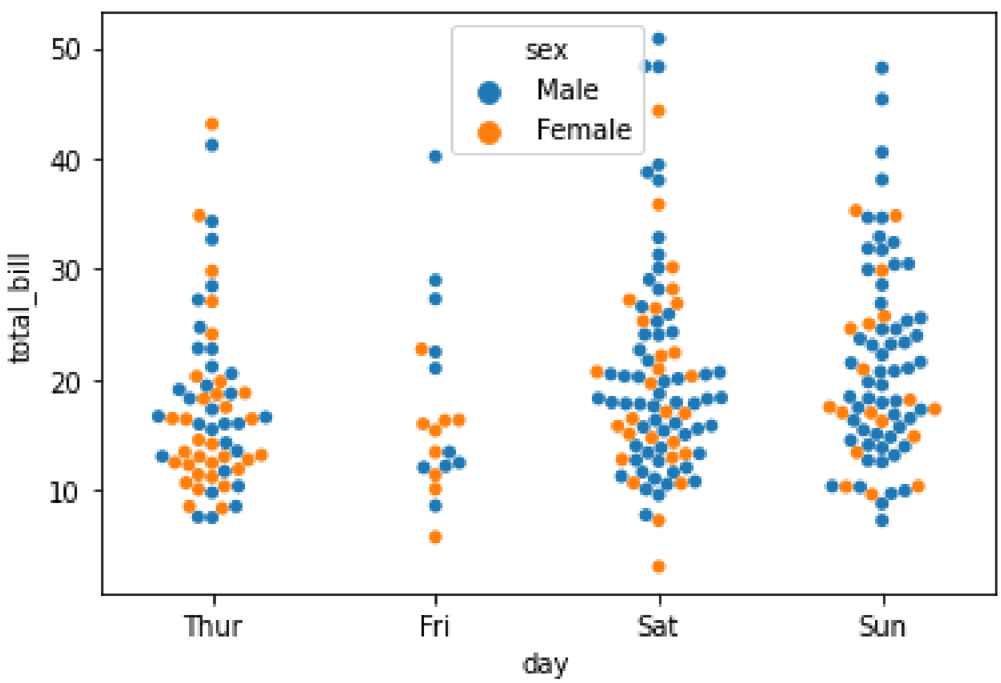
[ ]:
```
sns.swarmplot(x="day", y="total_bill", data=tips)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd0c9940>




```
[ ]: sns.swarmplot(x="day", y="total_bill", hue="sex", data=tips)
```

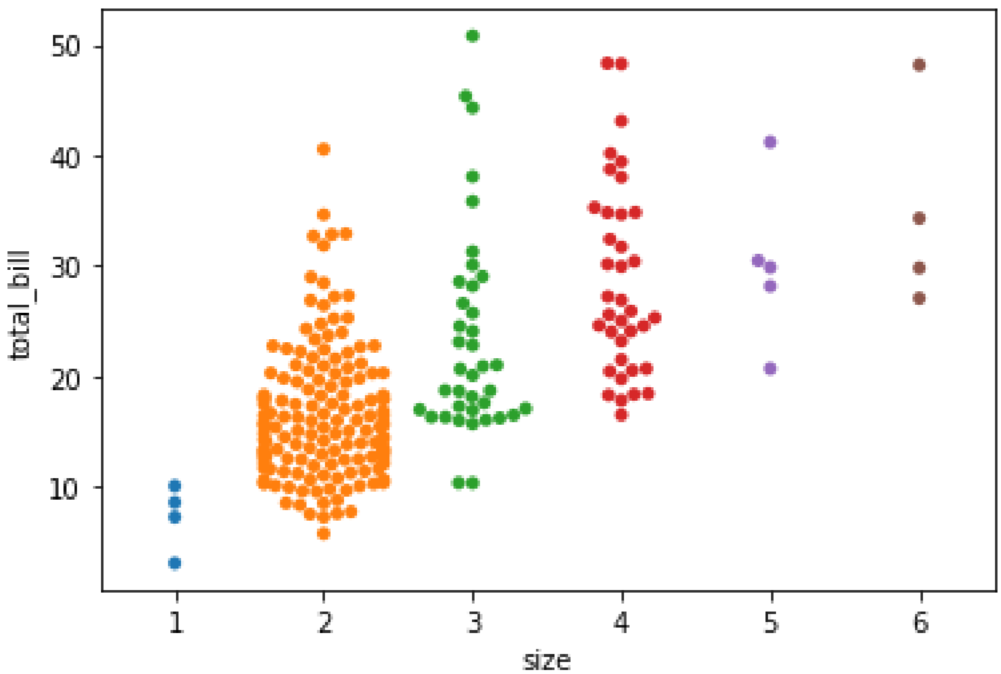
```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd00f358>
```



```
[ ]: sns.swarmplot(x="size", y="total_bill", data=tips)
```

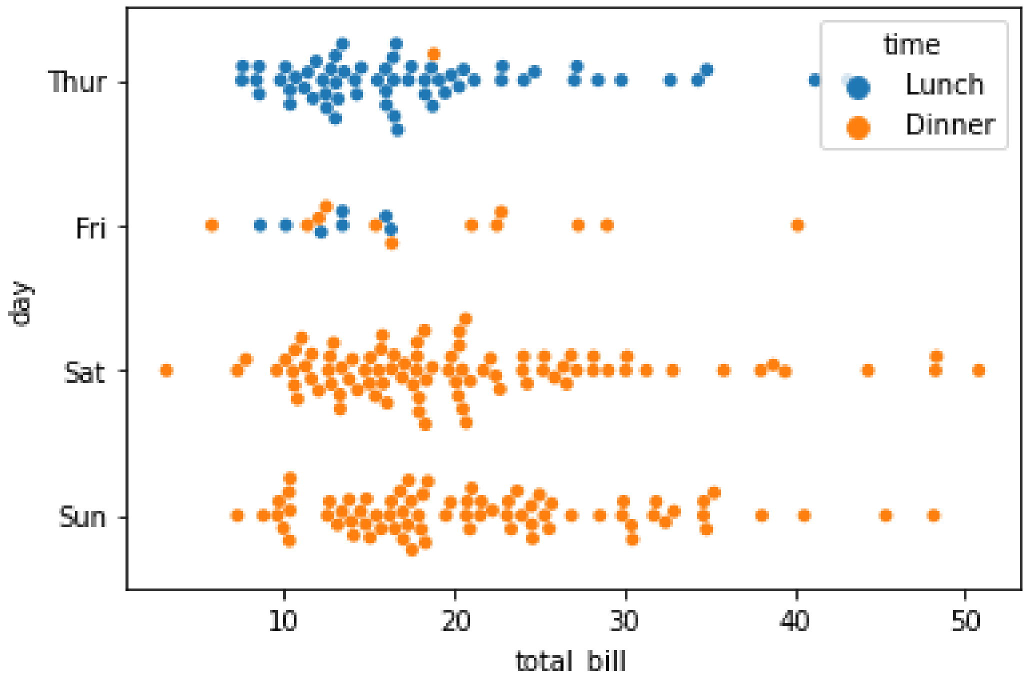
```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:1296: UserWarning:
26.9% of the points cannot be placed; you may want to decrease the size of the
markers or use stripplot.
  warnings.warn(msg, UserWarning)
```

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ccff4198>
```




```
[ ]: sns.swarmplot(x="total_bill", y="day", hue="time", data=tips)
```

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ccf53080>
```

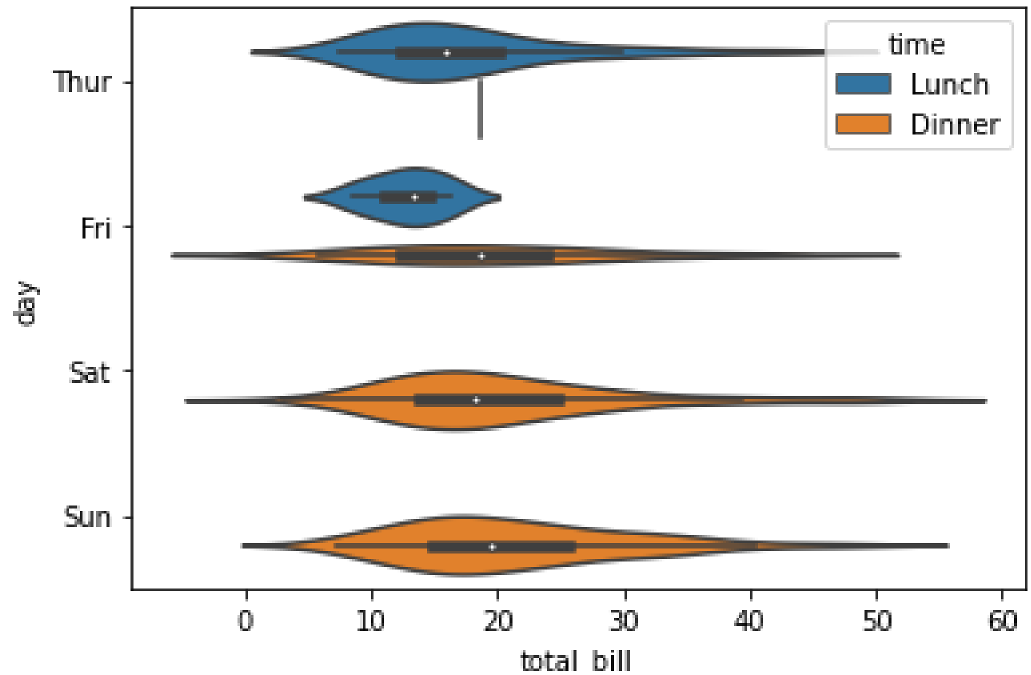


1.12 Violin plots

A violin plot is a method of plotting numeric data. It is a box plot with a rotated kernel density plot on each side. The violin plot is similar to box plots, except that they also show the probability density of the data at different values.

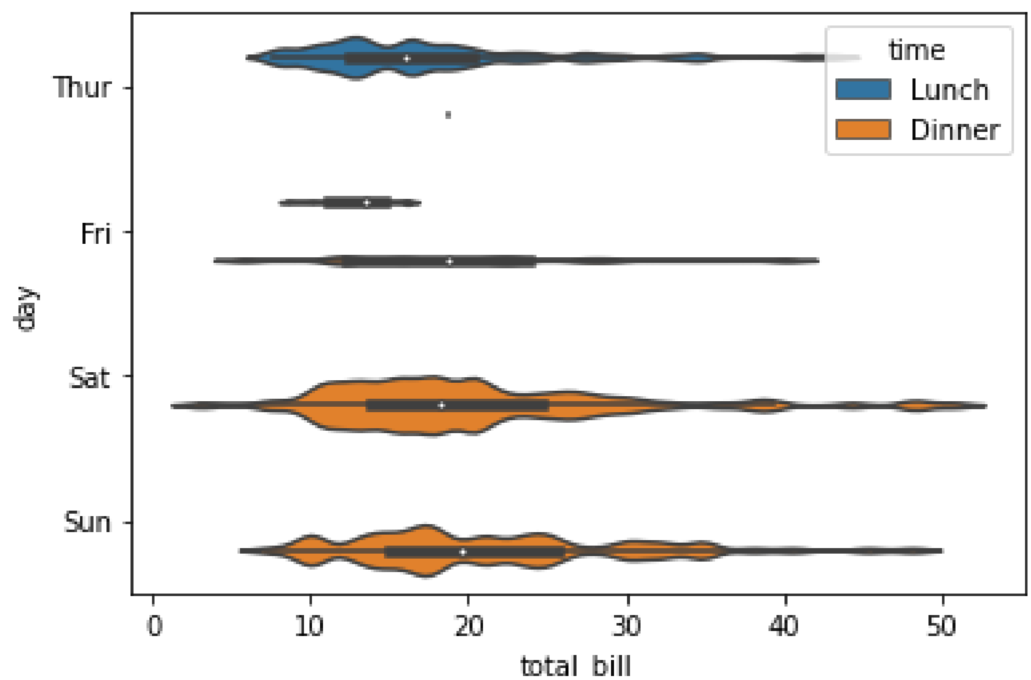
[ ]:
```
sns.violinplot(x="total_bill", y="day", hue="time", data=tips)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ccf42cc0>



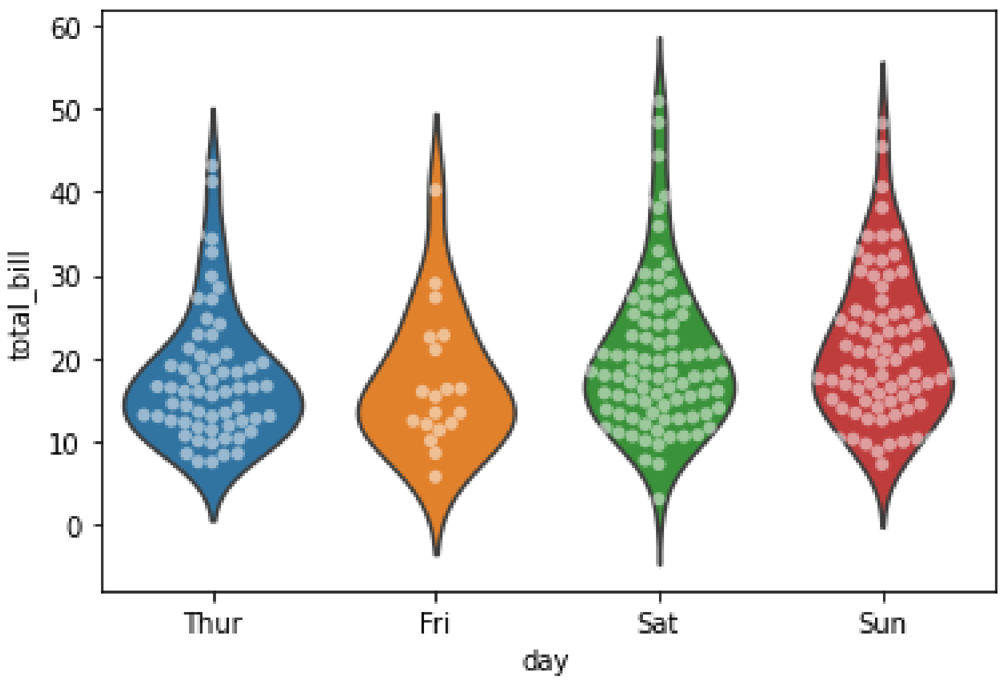
[ ]:
```python
sns.violinplot(x="total_bill", y="day", hue="time", data=tips,
               bw=.1, scale="count", scale_hue=False)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cd0b4080>



[ ]:
```
sns.violinplot(x="day", y="total_bill", data=tips, inner=None)
sns.swarmplot(x="day", y="total_bill", data=tips, color="w", alpha=.5)
```

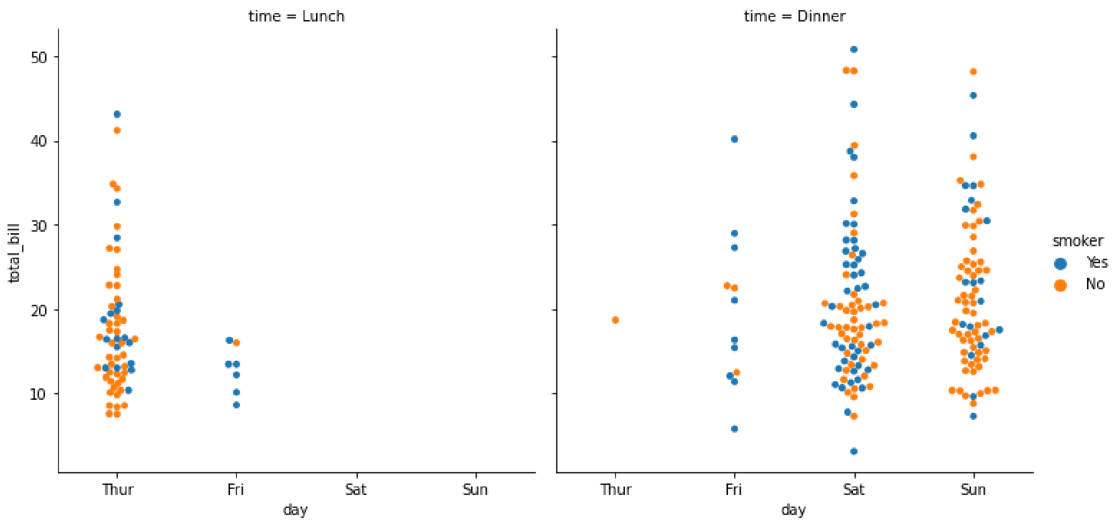
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ccdf2908>



[ ]:
```
sns.factorplot(x="day", y="total_bill", hue="smoker",
               col="time", data=tips, kind="swarm")
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:3704: UserWarning:
The `factorplot` function has been renamed to `catplot`. The original name will
be removed in a future release. Please update your code. Note that the default
`kind` in `factorplot` (`'point'`) has changed `'strip'` in `catplot`.
  warnings.warn(msg)
```

[ ]: <seaborn.axisgrid.FacetGrid at 0x7f35cd315f28>

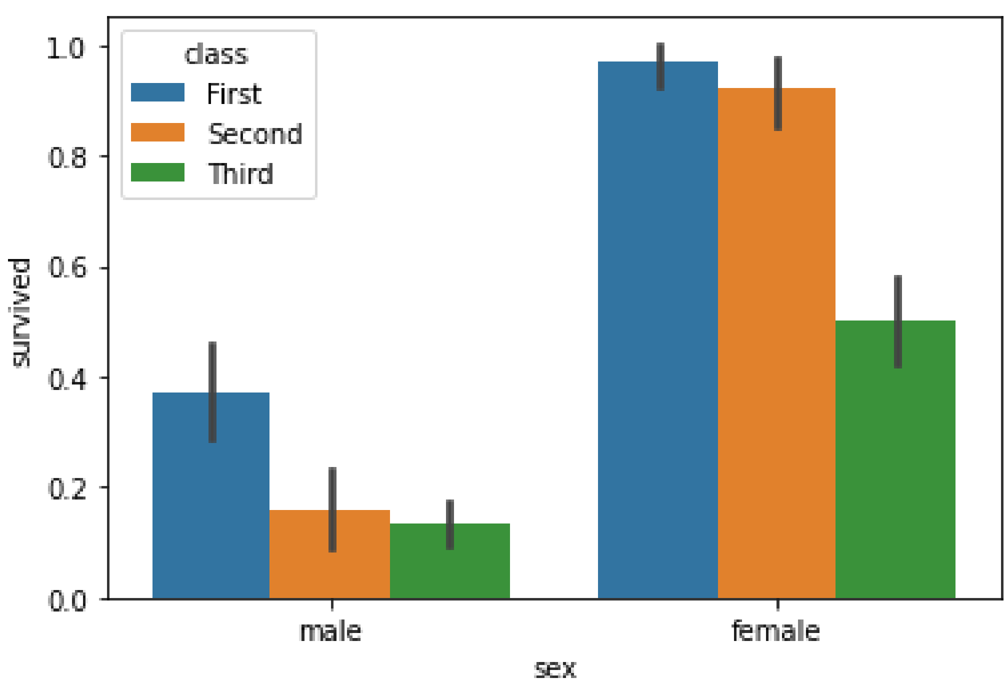


1.13 Bar charts

A bar chart or bar graph is a chart that presents Grouped data with rectangular bars with lengths proportional to the values that they represent. Bar chart - Wikipedia

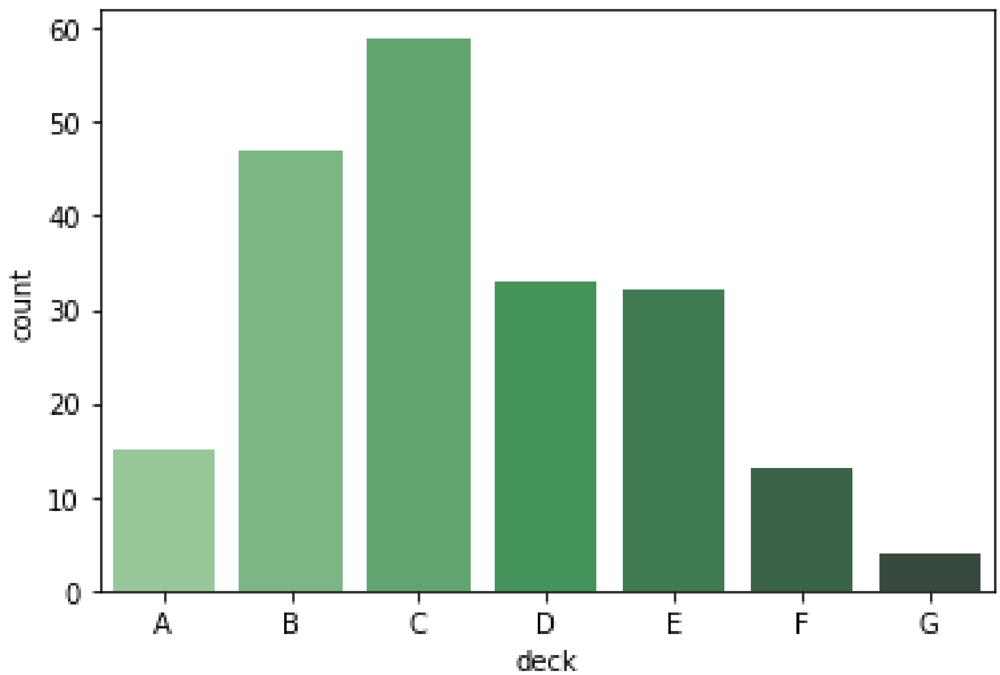
[ ]:
```
sns.barplot(x="sex", y="survived", hue="class", data=titanic)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ce9c5eb8>



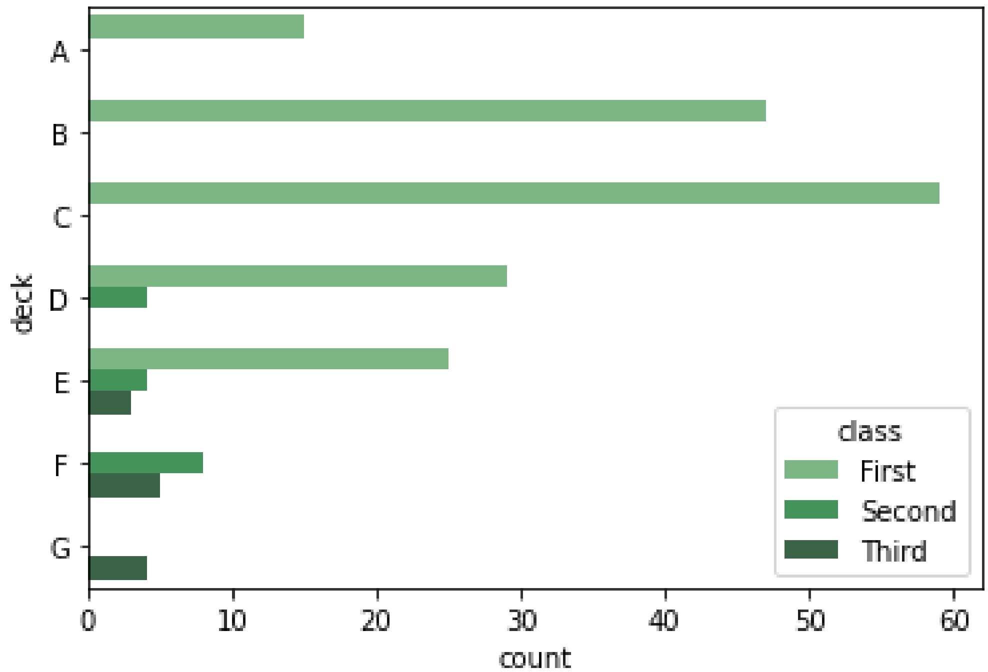
[ ]:
```
sns.countplot(x="deck", data=titanic, palette="Greens_d")
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ce987320>



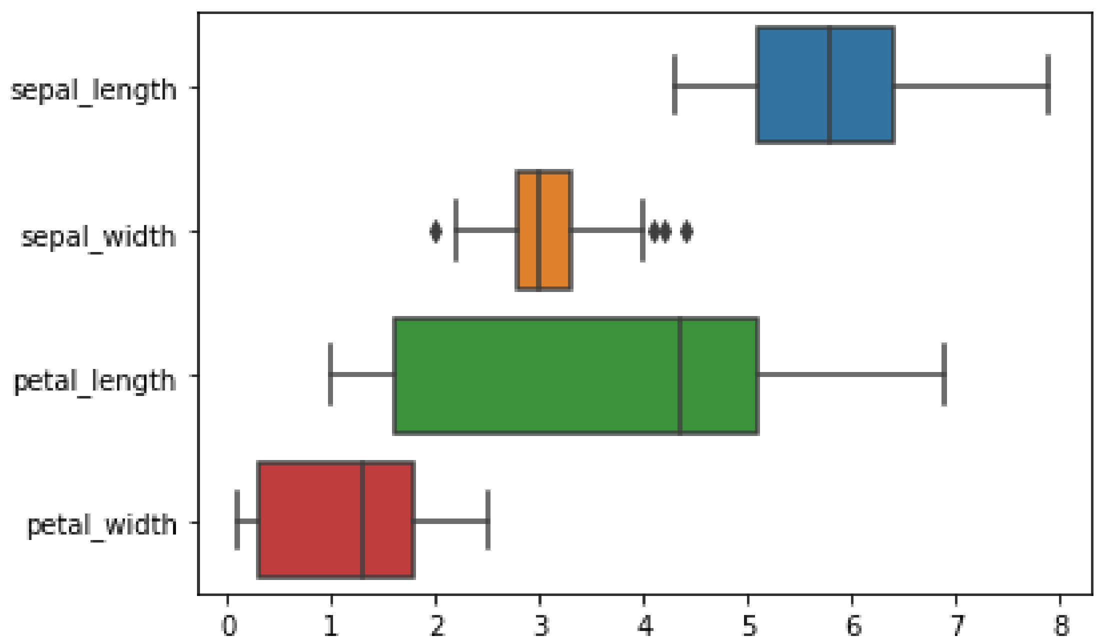
[ ]:
```
sns.countplot(y="deck", hue="class", data=titanic, palette="Greens_d")
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ce99c5f8>



[ ]:
```
sns.boxplot(data=iris, orient="h")
```

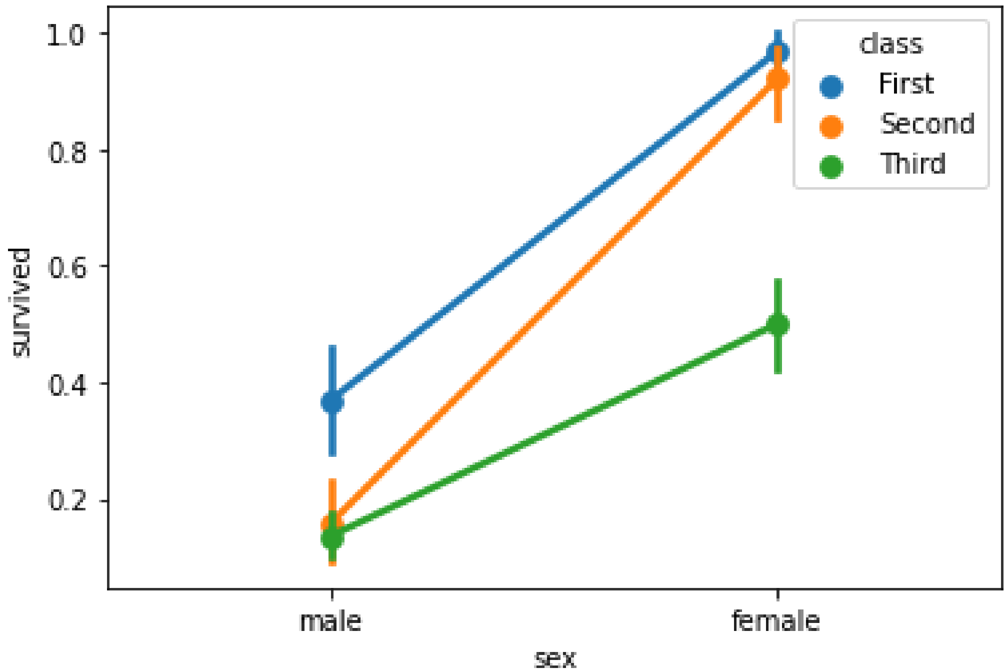
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ce99c518>



1.14 Point plots

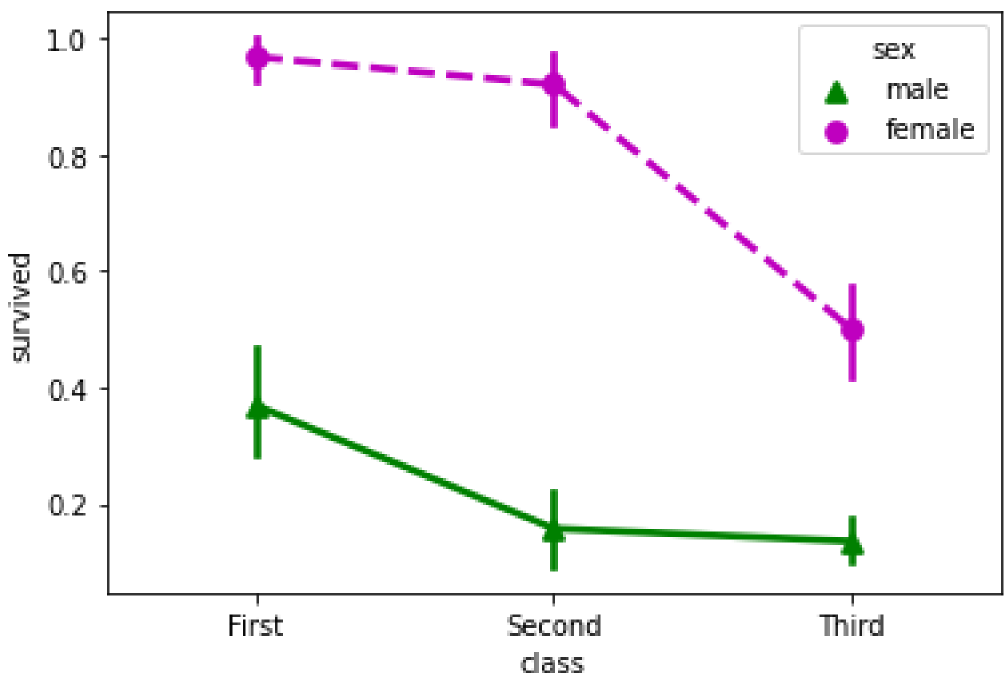
[ ]:
```
sns.pointplot(x="sex", y="survived", hue="class", data=titanic)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ccef6e80>



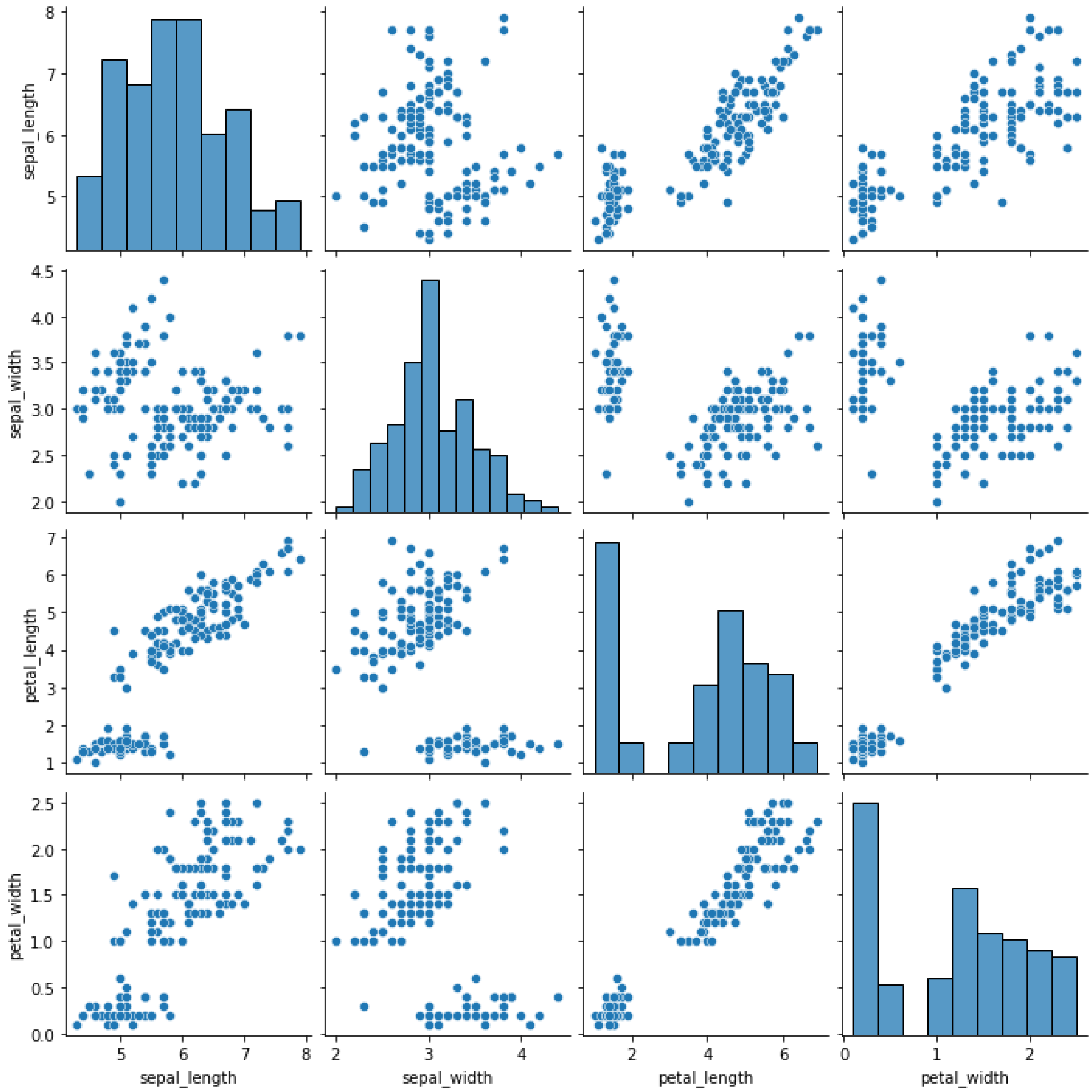
[ ]:
```
sns.pointplot(x="class", y="survived", hue="sex", data=titanic,
              palette={"male": "g", "female": "m"},
              markers=["^", "o"], linestyles=["-", "--"])
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35ccbefa90>



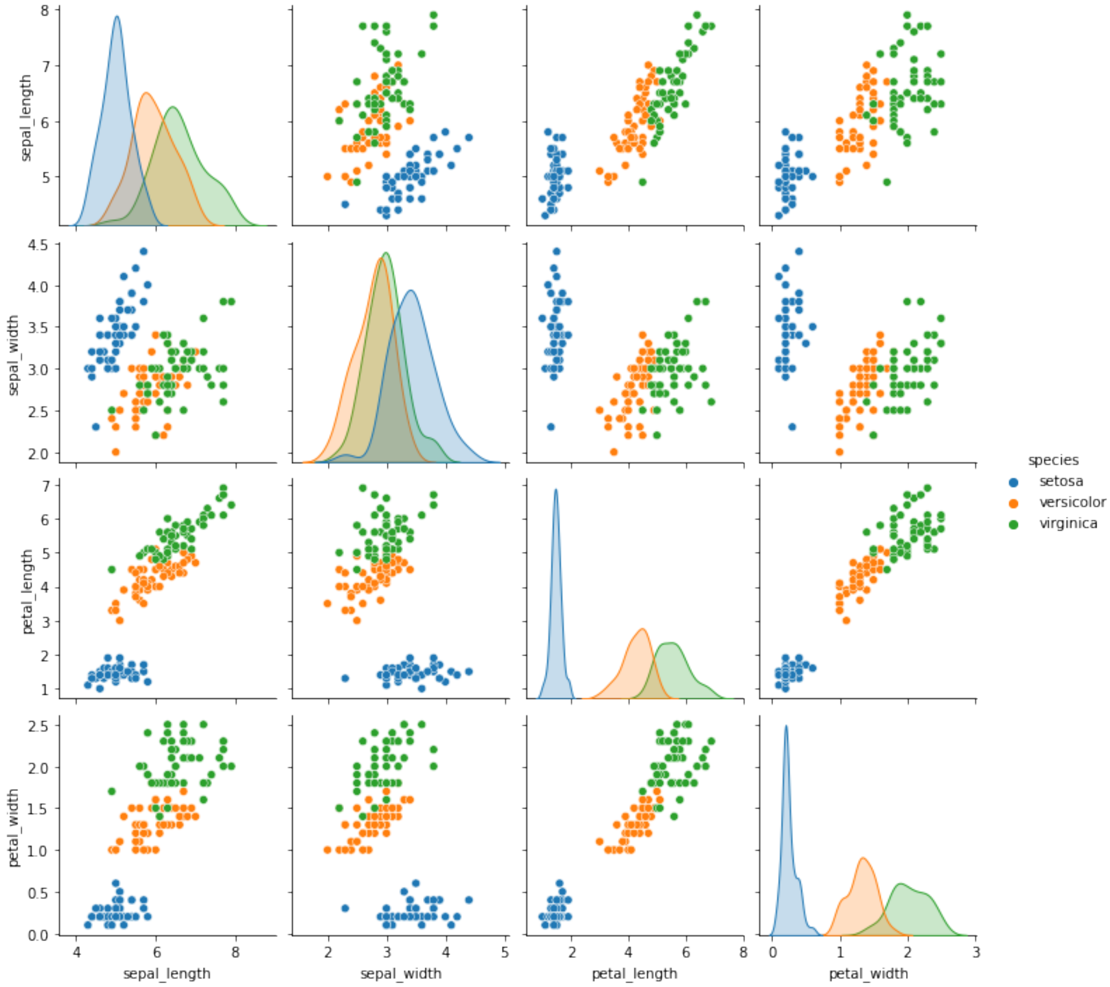

```
[ ]: sns.pairplot(iris)
```

```
[ ]: <seaborn.axisgrid.PairGrid at 0x7f35ccbe5390>
```




```
[ ]: sns.pairplot(iris, hue="species")
```

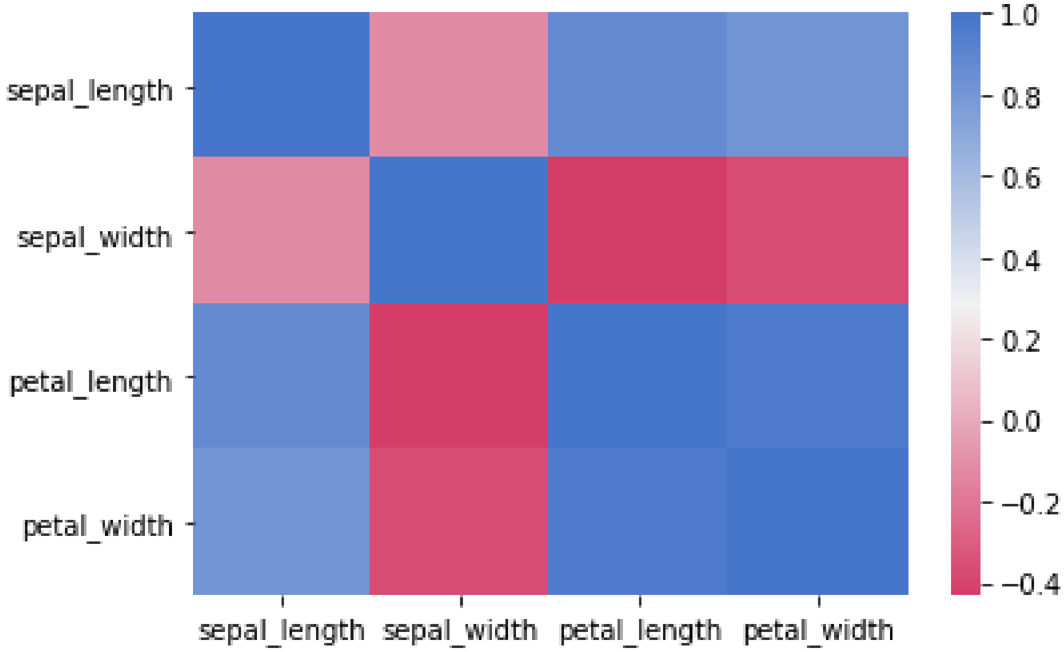
```
[ ]: <seaborn.axisgrid.PairGrid at 0x7f35cc640198>
```




```
[ ]: cmap = sns.diverging_palette(0, 255, sep=1, n=256, as_cmap=True)

     correlations = iris[['sepal_length', 'sepal_width', 'petal_length',␣
      ↪'petal_width']].corr()
     print (correlations)
     sns.heatmap(correlations, cmap=cmap)
```

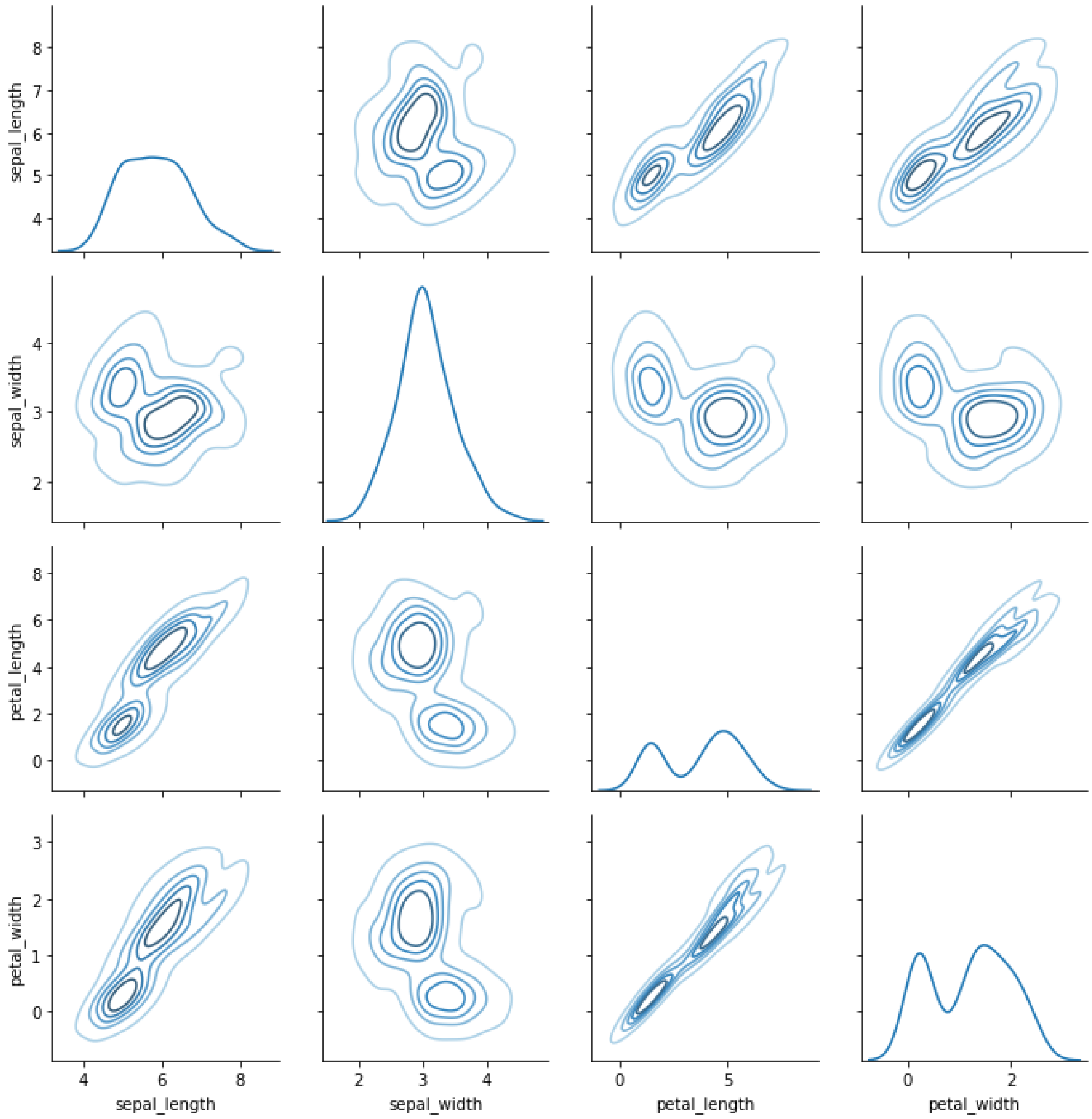
	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1.000000	-0.117570	0.871754	0.817941
sepal_width	-0.117570	1.000000	-0.428440	-0.366126
petal_length	0.871754	-0.428440	1.000000	0.962865
petal_width	0.817941	-0.366126	0.962865	1.000000

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cc515358>
```



[ ]:
```
g = sns.PairGrid(iris)
g.map_diag(sns.kdeplot)
g.map_offdiag(sns.kdeplot, cmap="Blues_d", n_levels=6)
```

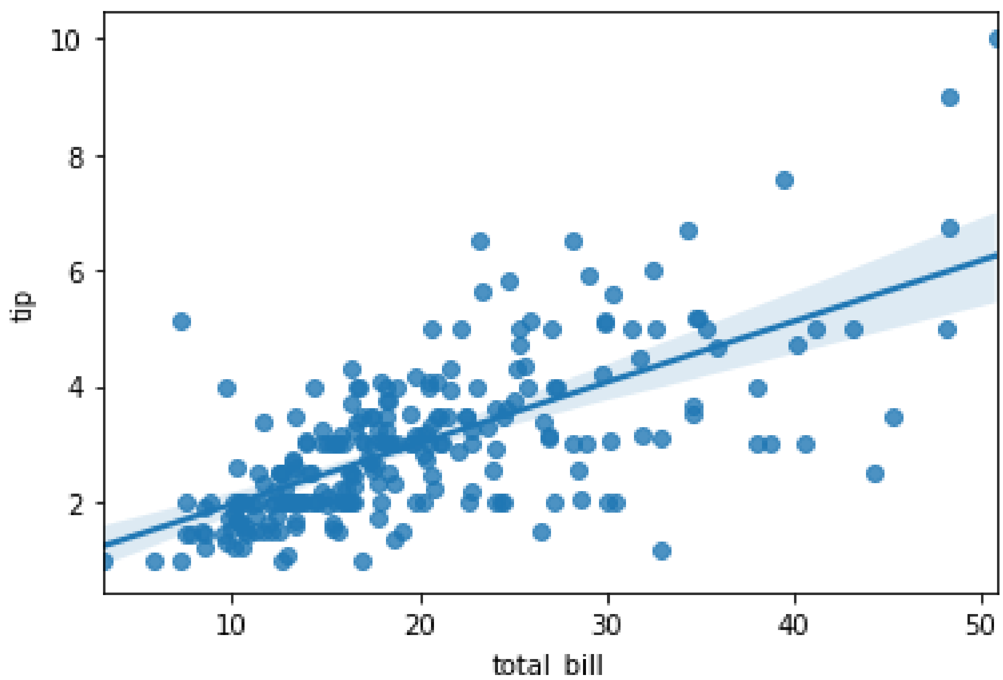
[ ]: <seaborn.axisgrid.PairGrid at 0x7f35cbe73b00>



1.15 Visualizing linear relationships

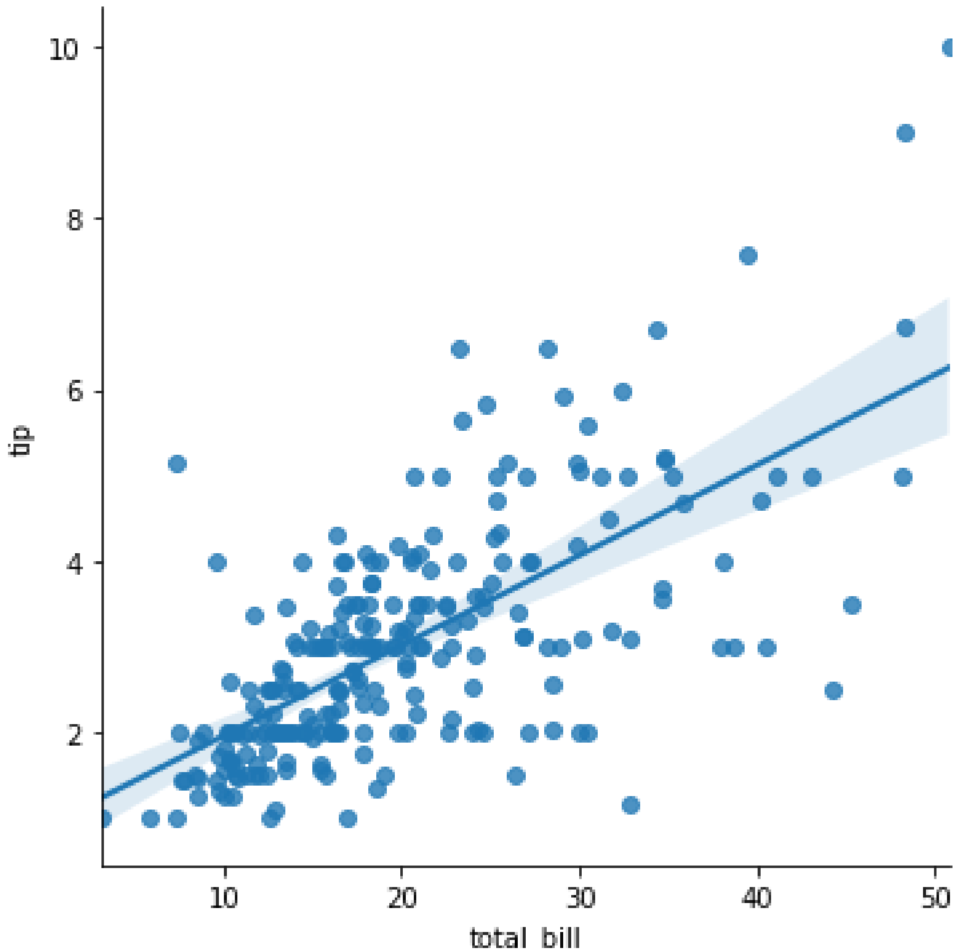
[ ]:
```
sns.regplot(x="total_bill", y="tip", data=tips)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cb714320>

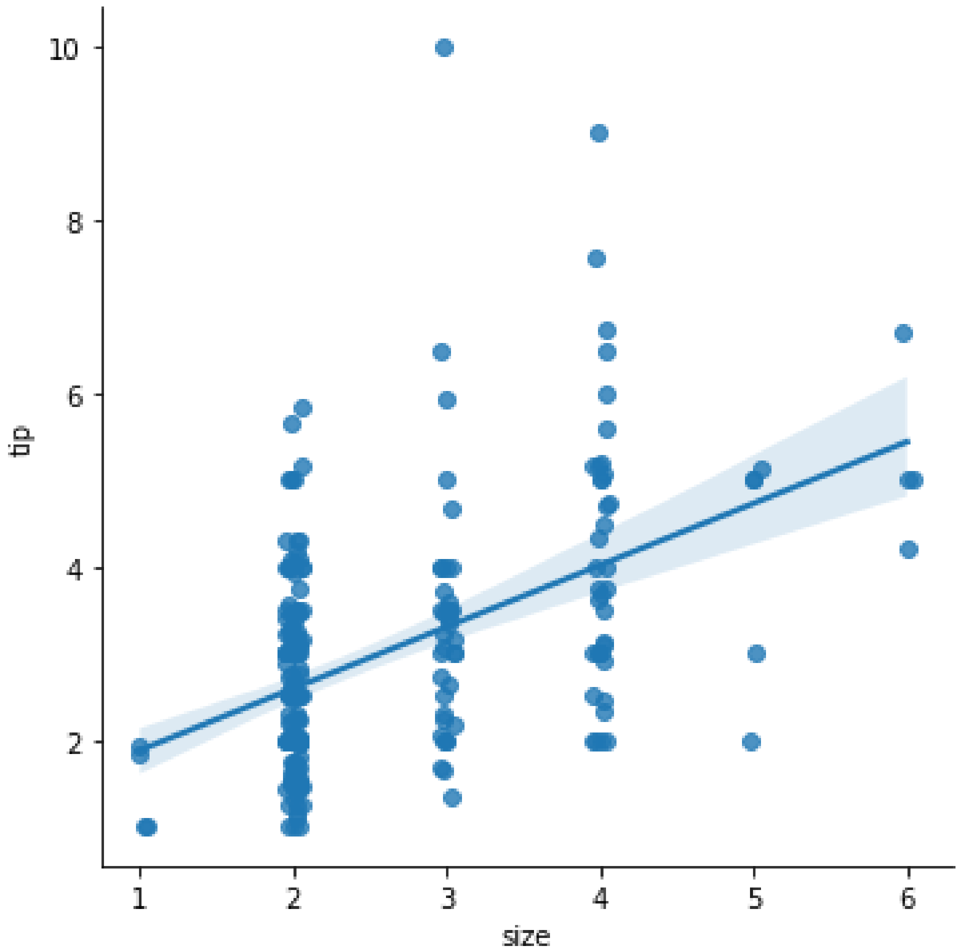


[ ]:
```
sns.lmplot(x="total_bill", y="tip", data=tips)
```

[ ]: <seaborn.axisgrid.FacetGrid at 0x7f35ccc6b898>

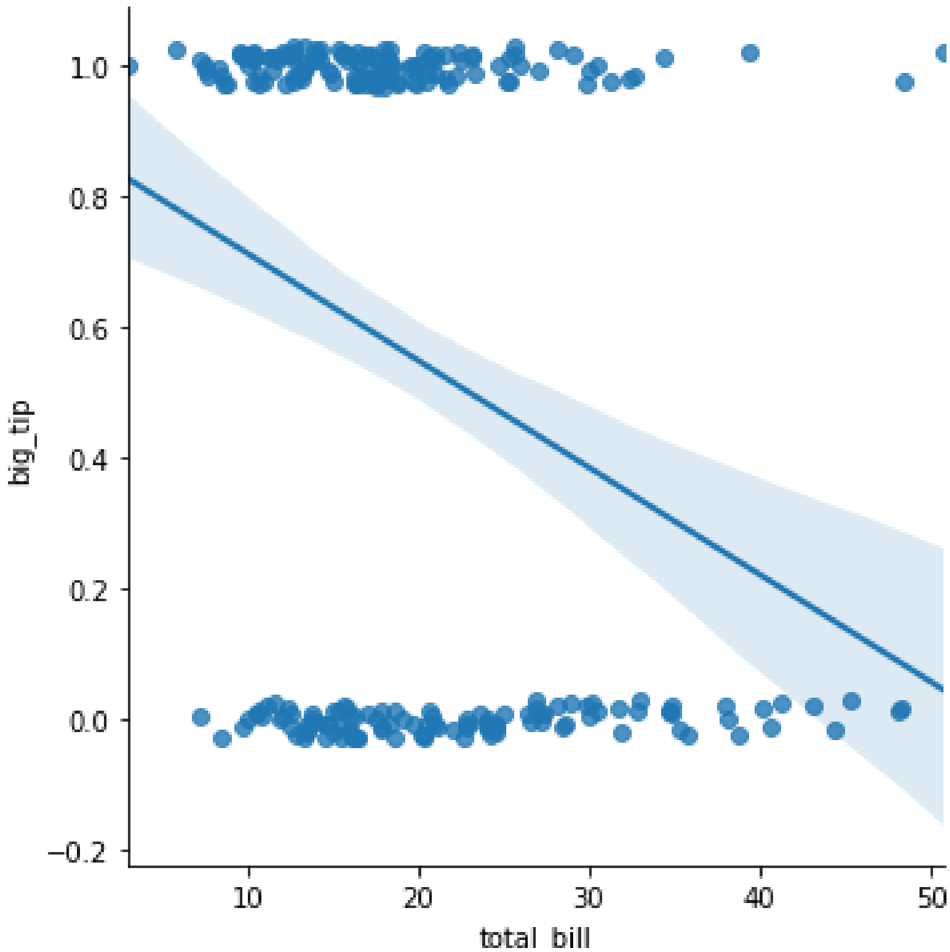


[ ]:
```
sns.lmplot(x="size", y="tip", data=tips, x_jitter=.05);
```



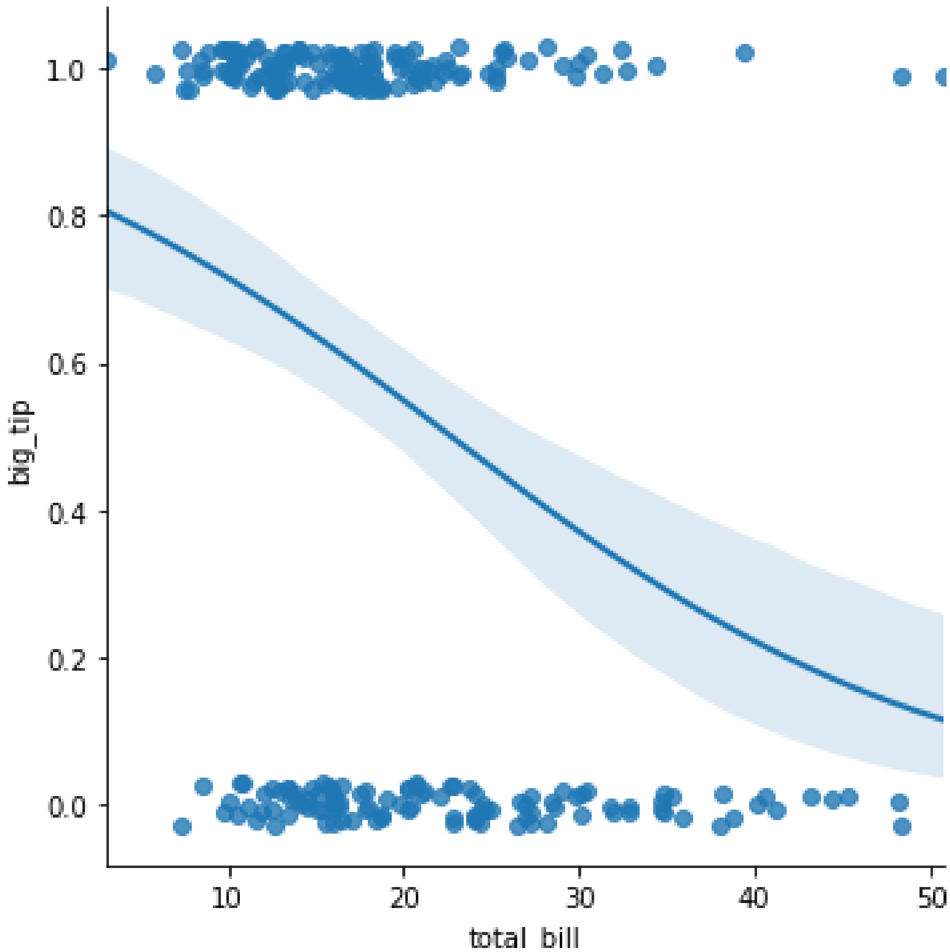

```
[ ]: tips["big_tip"] = (tips.tip / tips.total_bill) > .15
     sns.lmplot(x="total_bill", y="big_tip", data=tips,
                y_jitter=.03)
```

```
[ ]: <seaborn.axisgrid.FacetGrid at 0x7f35cd1e3f60>
```



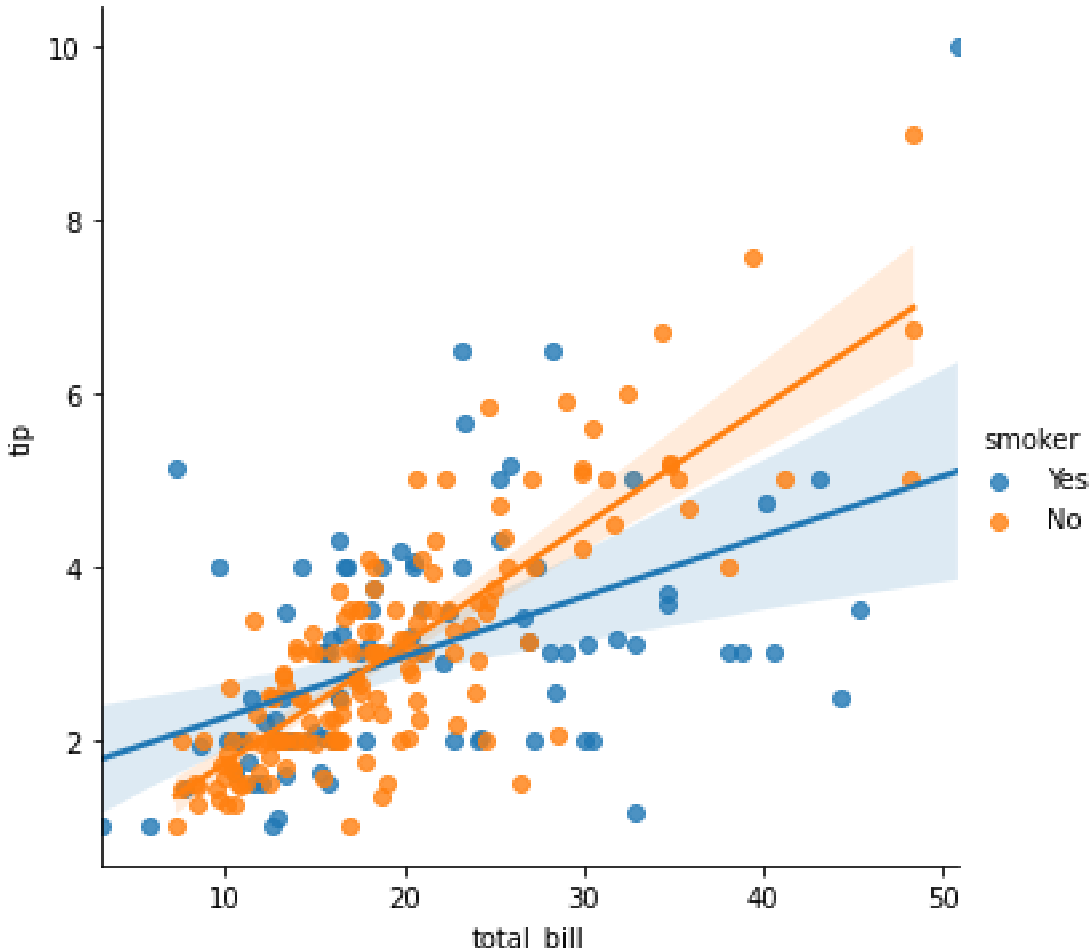
[ ]:
```
sns.lmplot(x="total_bill", y="big_tip", data=tips,
           logistic=True, y_jitter=.03)
```

[ ]: <seaborn.axisgrid.FacetGrid at 0x7f35cb665b00>



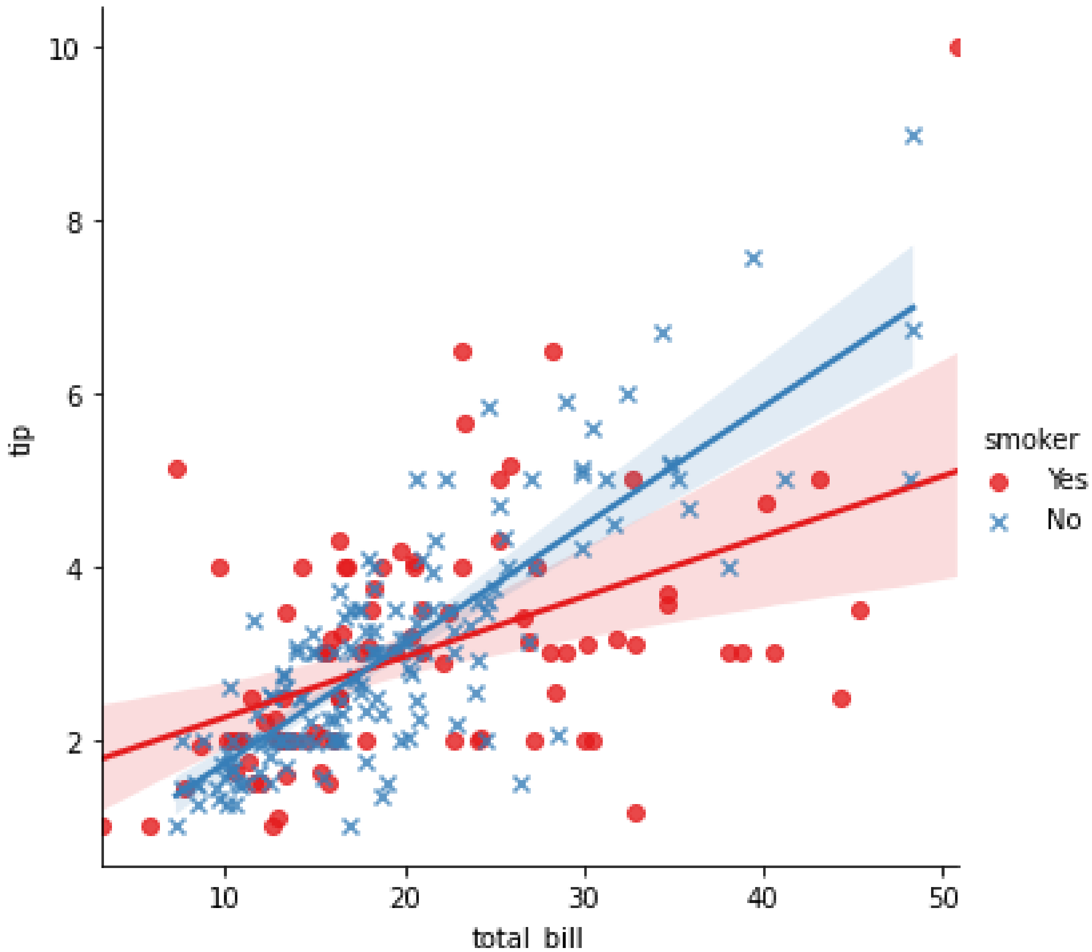
[ ]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips)
```

[ ]: <seaborn.axisgrid.FacetGrid at 0x7f35cb665710>



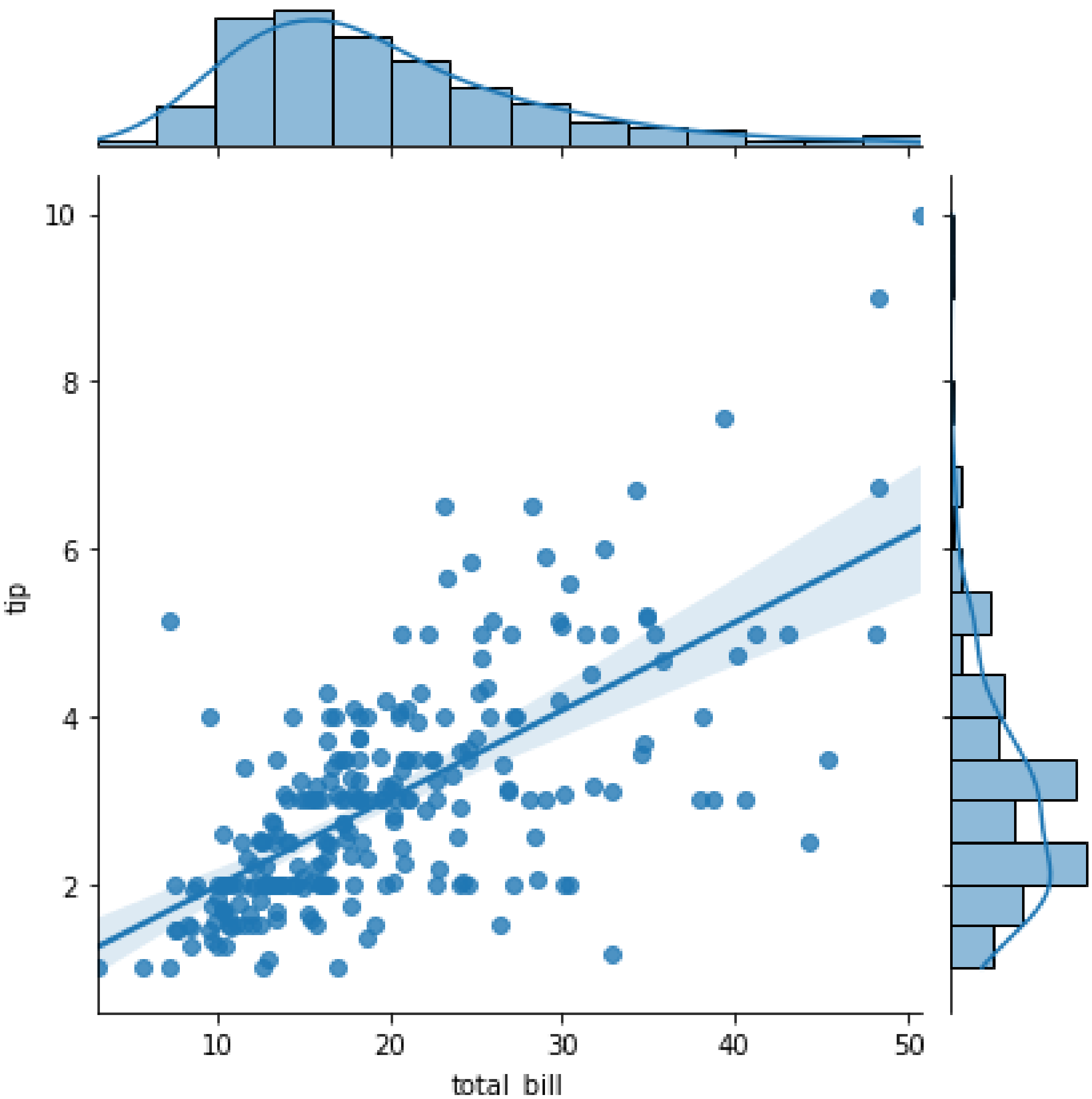

```
[ ]: sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips,
               markers=["o", "x"], palette="Set1")
```

```
[ ]: <seaborn.axisgrid.FacetGrid at 0x7f35cb4faf98>
```



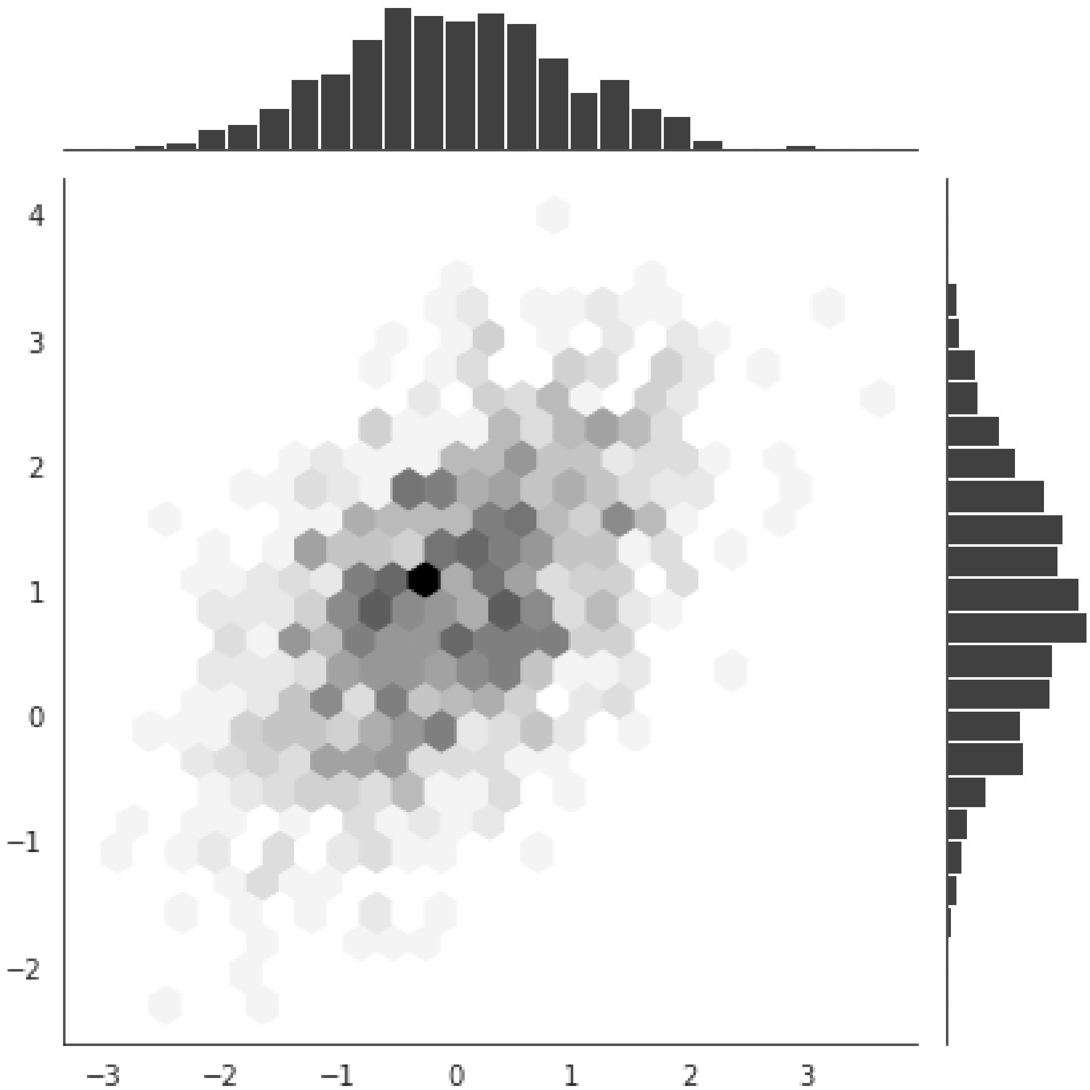

```
[ ]: sns.jointplot(x="total_bill", y="tip", data=tips, kind="reg")
```

```
[ ]: <seaborn.axisgrid.JointGrid at 0x7f35cb4e0128>
```



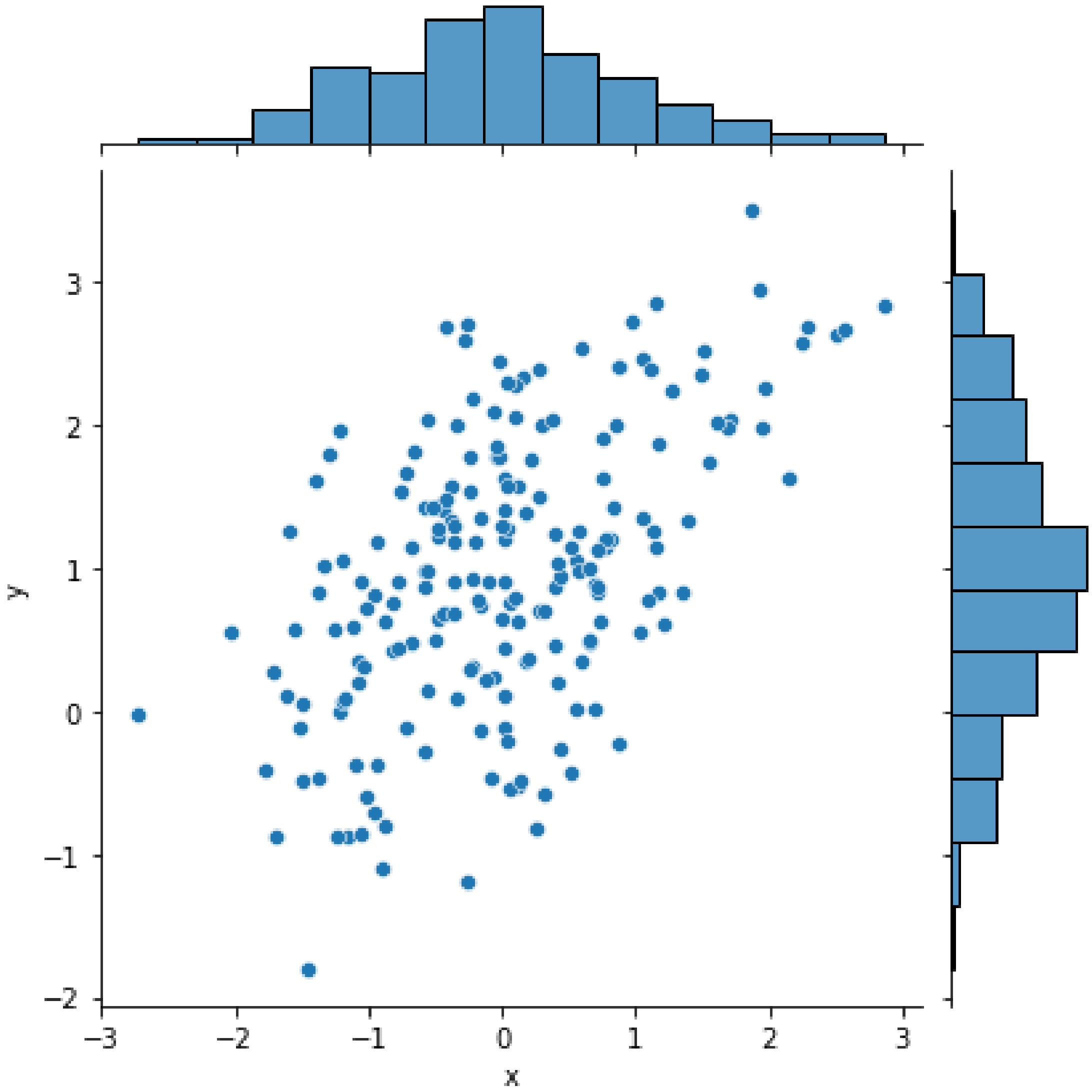
1.16 Even more plots

[ ]:
```
mean, cov = [0, 1], [(1, .5), (.5, 1)]
data = np.random.multivariate_normal(mean, cov, 200)
df = pd.DataFrame(data, columns=["x", "y"])

x, y = np.random.multivariate_normal(mean, cov, 1000).T
with sns.axes_style("white"):
    sns.jointplot(x=x, y=y, kind="hex", color="k")
```



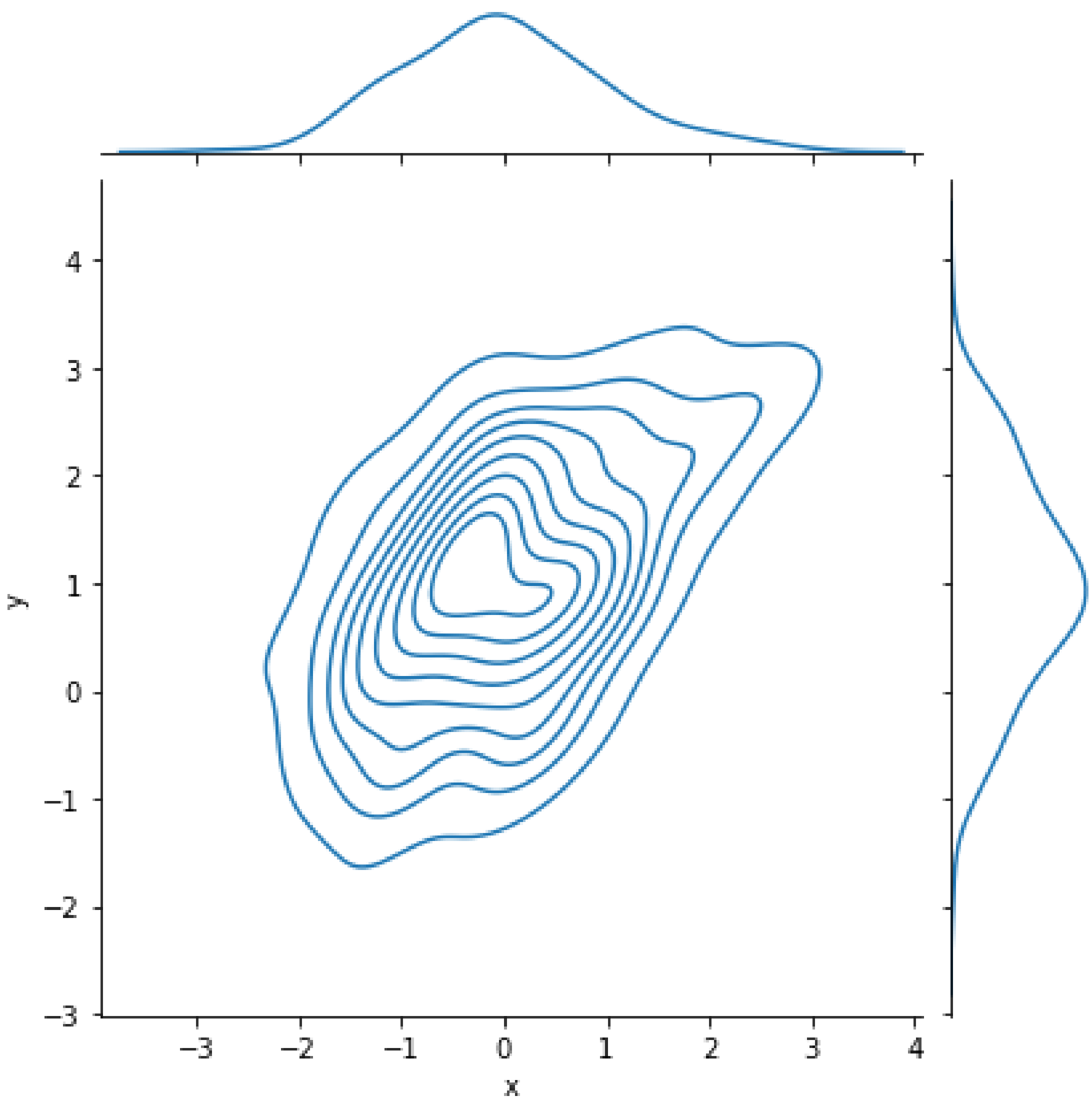
[ ]:
```
sns.jointplot(x="x", y="y", data=df)
```

[ ]: <seaborn.axisgrid.JointGrid at 0x7f35cd46b908>




```
[ ]: sns.jointplot(x="x", y="y", data=df, kind="kde")
```

```
[ ]: <seaborn.axisgrid.JointGrid at 0x7f35cb053a90>
```

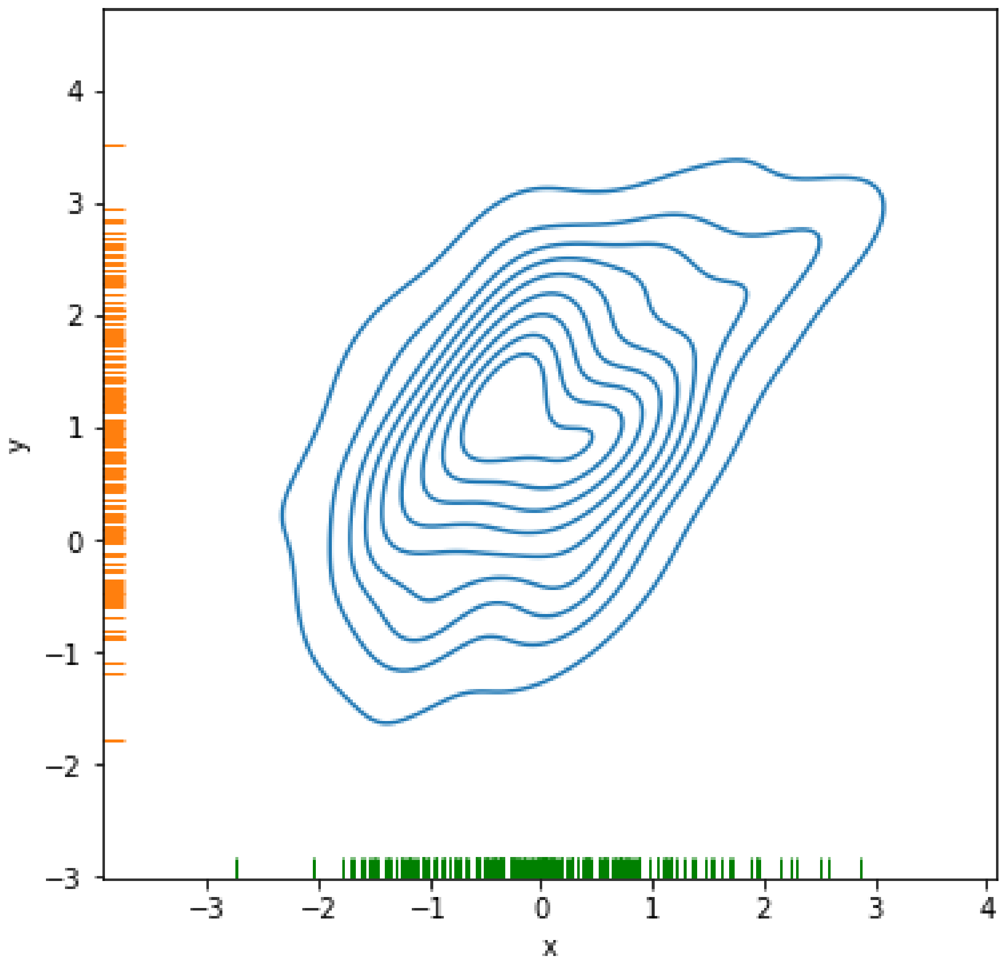



```
[ ]: f, ax = plt.subplots(figsize=(6, 6))
     sns.kdeplot(df.x, df.y, ax=ax)
     sns.rugplot(df.x, color="g", ax=ax)
     sns.rugplot(df.y, vertical=True, ax=ax)
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorators.py:43: FutureWarning:
Pass the following variable as a keyword arg: y. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  FutureWarning
/usr/local/lib/python3.6/dist-packages/seaborn/distributions.py:2064:
FutureWarning: Using `vertical=True` to control the orientation of the plot  is
deprecated. Instead, assign the data directly to `y`.
```

```
  warnings.warn(msg, FutureWarning)
```

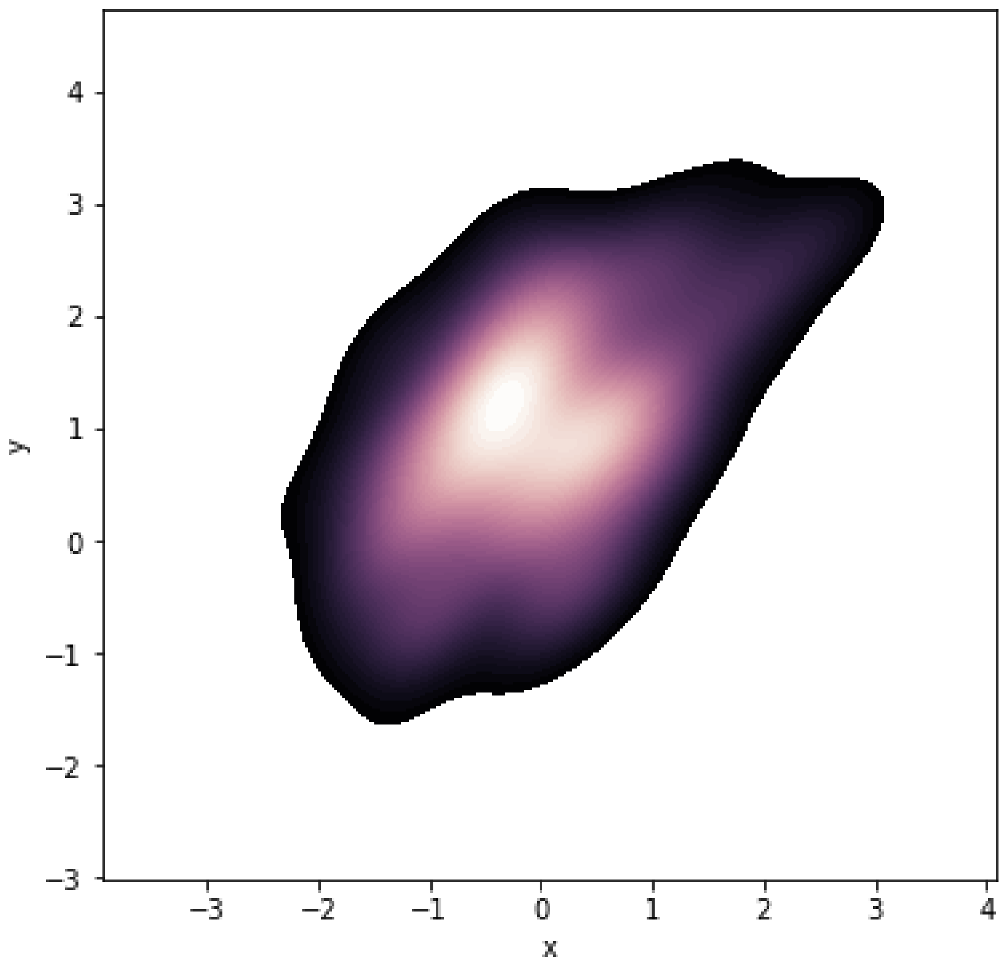
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35caf17518>



[ ]:
```
f, ax = plt.subplots(figsize=(6, 6))
cmap = sns.cubehelix_palette(as_cmap=True, dark=0, light=1, reverse=True)
sns.kdeplot(df.x, df.y, cmap=cmap, n_levels=60, shade=True)
```

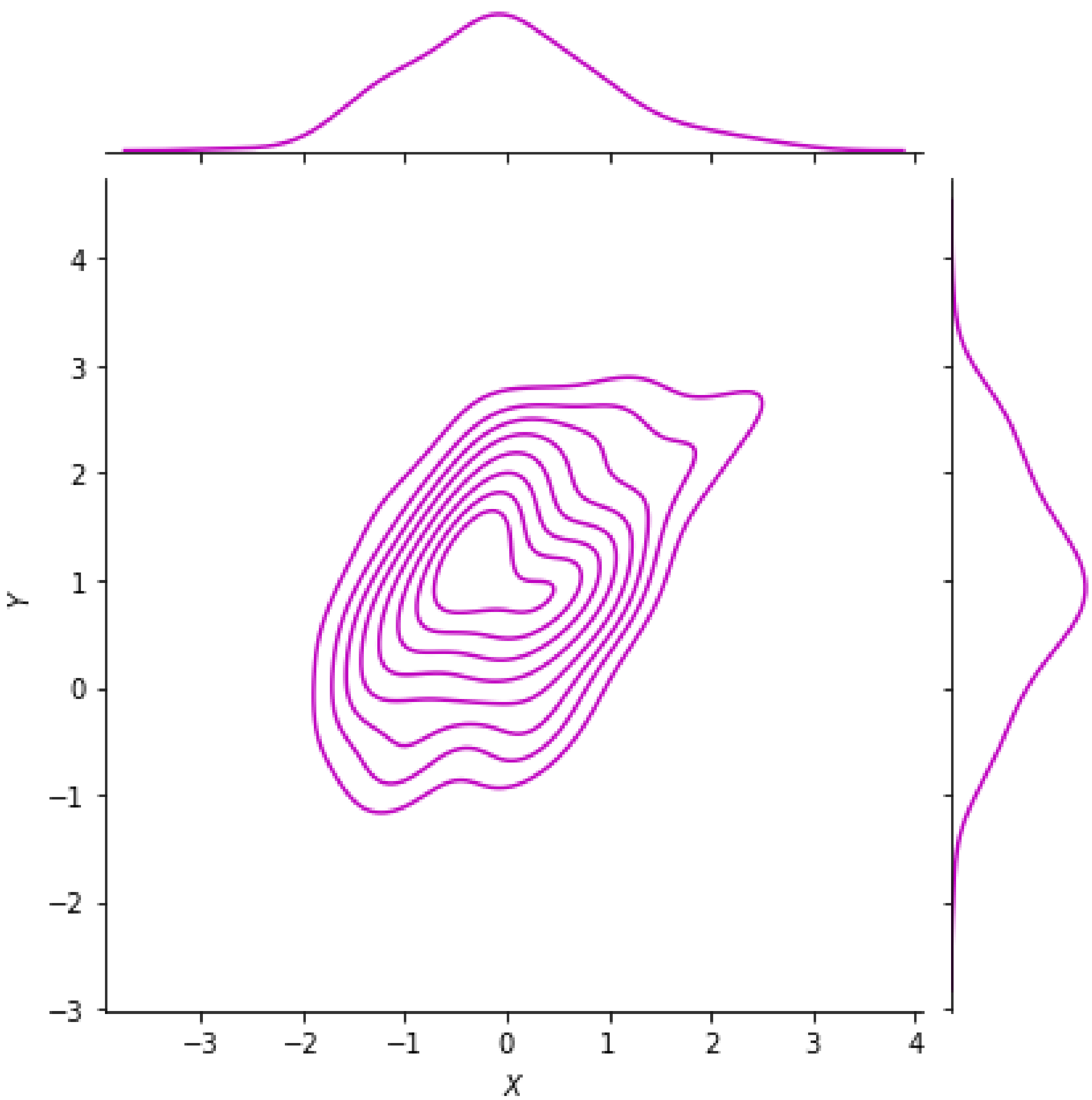
```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorators.py:43: FutureWarning:
Pass the following variable as a keyword arg: y. From version 0.12, the only
valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cae4a828>



[ ]:
```
g = sns.jointplot(x="x", y="y", data=df, kind="kde", color="m")
g.plot_joint(plt.scatter, c="w", s=30, linewidth=1, marker="+")
g.ax_joint.collections[0].set_alpha(0)
g.set_axis_labels("$X$", "$Y$")
```

[ ]: <seaborn.axisgrid.JointGrid at 0x7f35cad49080>



1.17 Time Series plots

Matplotlib and Seaborn can make some nice plots associated to time series data. For example, we can make plots of running. The following data contains the monthly price of the ETF VTI (a stock market index fund) over time.

See *Vanguard Total Stock Market Index Fund ETF Shares (VTI)* https://finance.yahoo.com/quote/VTI/history?ltr=1

[ ]:
```
vti = pd.read_csv("https://raw.githubusercontent.com/nikbearbrown/Google_Colab/
↪master/data/VTI.csv", sep=',')
vti.head()
```

[ ]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	9/20/19	153.000000	153.229996	151.600006	152.039993	149.790756	2106100
1	9/23/19	151.710007	152.460007	151.570007	152.119995	149.869568	2051800
2	9/24/19	152.600006	152.809998	150.199997	150.710007	148.480453	4234600
3	9/25/19	150.729996	151.910004	149.919998	151.660004	149.416382	6144100
4	9/26/19	151.660004	151.660004	150.470001	151.199997	148.963196	1723000

[ ]:
```
vti.describe()
```

[ ]:

	Open	High	Low	Close	Adj Close	Volume
count	252.000000	252.000000	252.000000	252.000000	252.000000	2.520000e+02
mean	156.291269	157.374286	154.998968	156.293334	155.154465	4.432734e+06
std	13.181936	12.534731	13.664153	13.065140	13.173404	3.458767e+06
min	113.650002	114.900002	109.489998	111.910004	110.852257	1.171000e+06
25%	150.282498	151.192497	148.917500	150.405002	148.460746	2.359950e+06
50%	158.370002	158.980004	157.425003	158.375000	157.013801	3.276750e+06
75%	165.382500	165.950000	164.324997	165.152504	164.517685	4.693275e+06
max	180.000000	181.669998	179.009995	181.240005	181.240005	2.228330e+07

[ ]:
```
import datetime

x = datetime.datetime.now()

print(x.strftime("%m/%d/%y"))
```

09/30/20

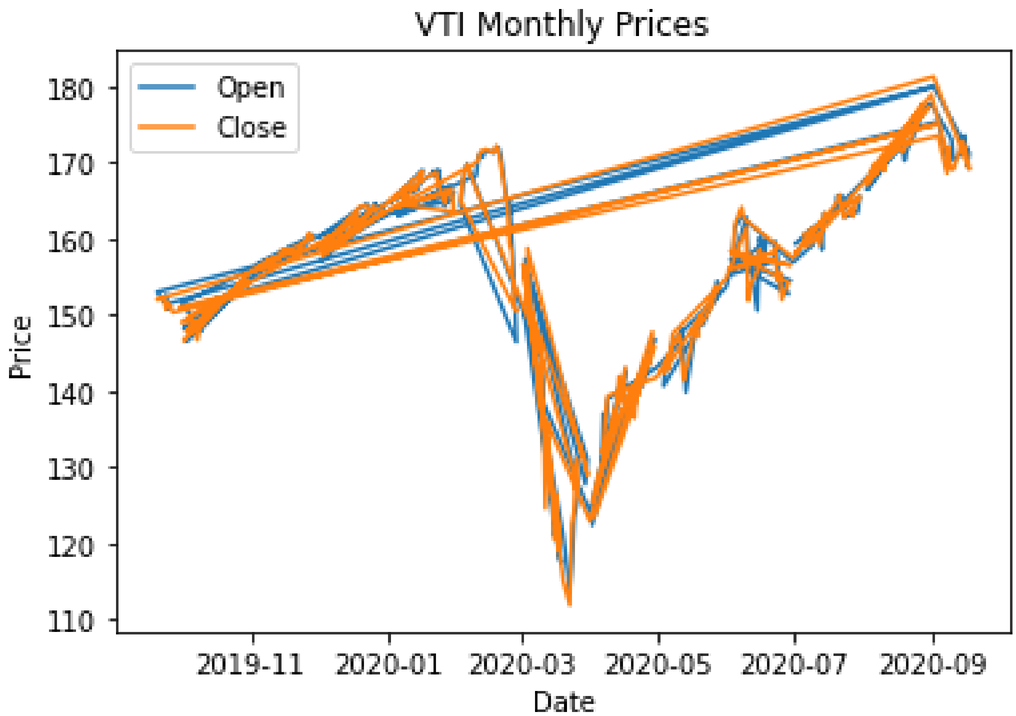
[ ]:
```
vti.sort_values(by="Date", inplace=True)
vti["Date"] = pd.to_datetime(vti["Date"], format='%m/%d/%y')
vti.head()
```

[ ]:

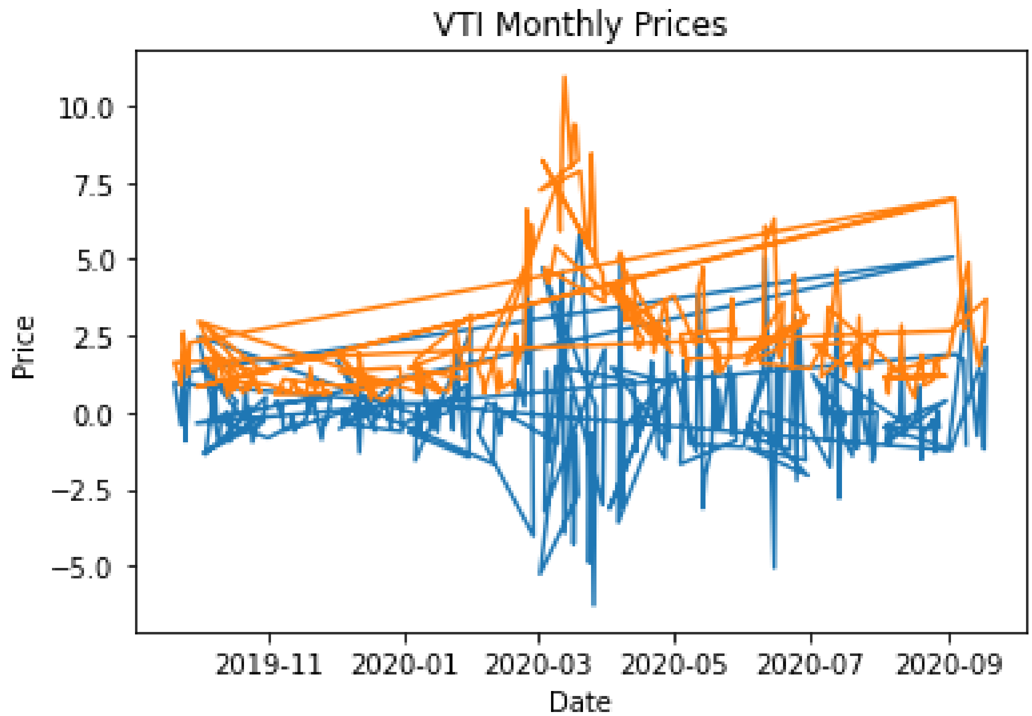
	Date	Open	High	…	Close	Adj Close	Volume
77	2020-01-10	166.259995	166.300003	…	165.460007	163.896118	4023000
78	2020-01-13	166.000000	166.630005	…	166.589996	165.015427	3997100
79	2020-01-14	166.550003	167.119995	…	166.500000	164.926285	3062000
80	2020-01-15	166.500000	167.399994	…	166.910004	165.332413	2479200
81	2020-01-16	167.710007	168.369995	…	168.339996	166.748886	2205600

[5 rows x 7 columns]

[ ]:
```
plt.plot(vti["Date"], vti["Open"], label="Open")
plt.plot(vti["Date"], vti["Close"], label="Close")
plt.xlabel("Date")
plt.ylabel("Price")
plt.title("VTI Monthly Prices")
plt.legend()
plt.show()
```




```
[ ]: plt.plot(vti["Date"], vti["Open"] - vti["Close"], label="Close-Open")
     plt.plot(vti["Date"], vti["High"] - vti["Low"], label="High-Low")
     plt.xlabel("Date")
     plt.ylabel("Price")
     plt.title("VTI Monthly Prices")
     plt.show()
```



1.18 Line plots

Line and path plots are typically used for time series data. Line plots join the points from left to right, while path plots join them in the order that they appear in the dataset (a line plot is just a path plot of the data sorted by x value).

To show this we'll use the ggplot2 economics dataset, which contains economic time-series data on the US measured over the last 40 years.

[ ]:

```
from scipy import stats
import random
data = []
for i in range(50):
    m = random.randint(5 + i, 15 + i)
    s = random.randint(3, 9)
    dist = stats.norm(m, s)
    draws = dist.rvs(33)
    data.append([np.mean(draws), np.std(draws)])
rd = pd.DataFrame(data, columns=["Mean", "Std"])
rd.head()
```

[ ]:

	Mean	Std
0	13.591512	6.024070
1	14.249740	8.415486

```
2   8.584941  8.723806
3  17.439352  7.266613
4  17.486449  7.338714
```

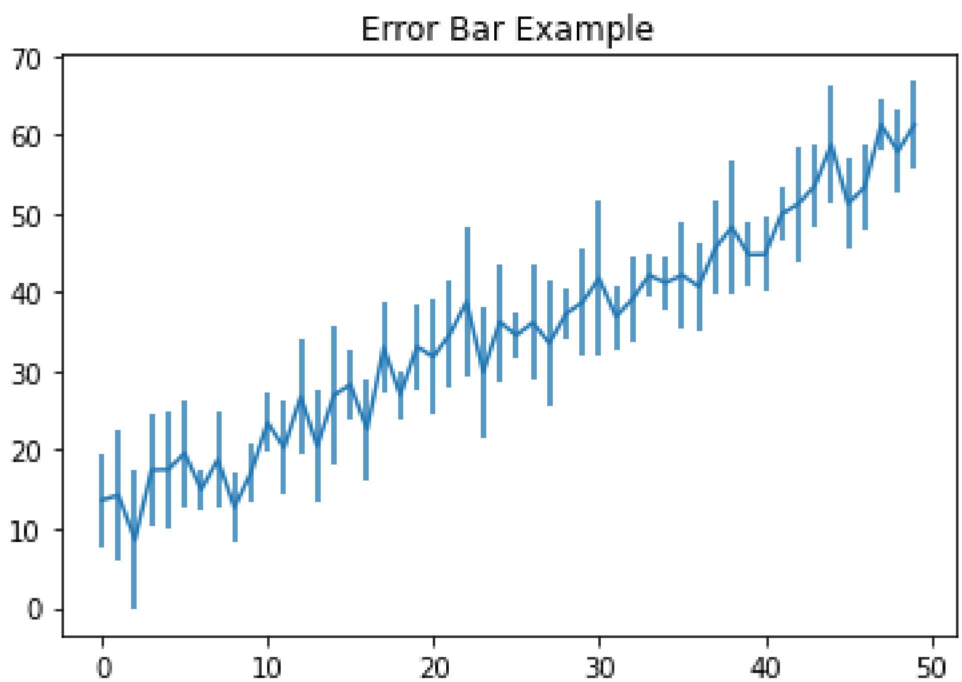
[ ]:
```python
rd.describe()
```

[ ]:

	Mean	Std
count	50.000000	50.000000
mean	34.656346	5.909662
std	13.968257	1.909855
min	8.584941	2.540109
25%	22.811757	4.336668
50%	35.392122	5.919476
75%	44.092999	7.265347
max	61.234352	9.925563

[ ]:
```python
plt.errorbar(range(len(rd)), rd["Mean"], yerr=rd["Std"])
plt.title("Error Bar Example")
```

[ ]: Text(0.5, 1.0, 'Error Bar Example')



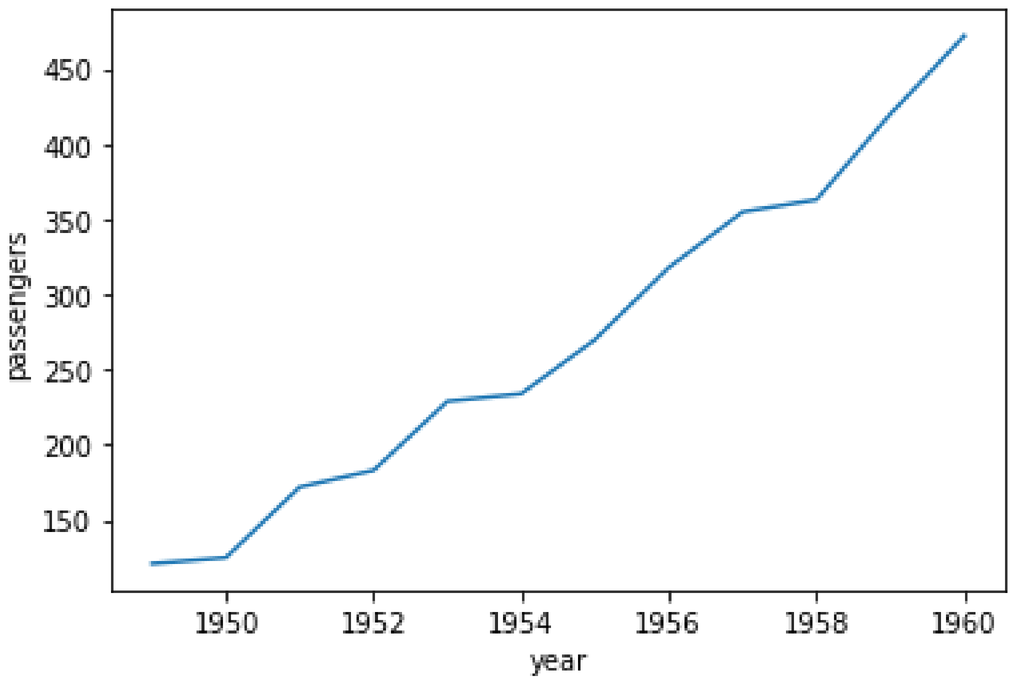
[ ]:
```python
flights = sns.load_dataset("flights")
flights.head()
```

```
[ ]:    year month  passengers
     0  1949   Jan         112
     1  1949   Feb         118
     2  1949   Mar         132
     3  1949   Apr         129
     4  1949   May         121
```

To draw a line plot using long-form data, assign the x and y variables:

```
[ ]: may_flights = flights.query("month == 'May'")
     sns.lineplot(data=may_flights, x="year", y="passengers")
```

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cabd75c0>
```



Passing the entire wide-form dataset to data plots a separate line for each column:

Pivot the dataframe to a wide-form representation:

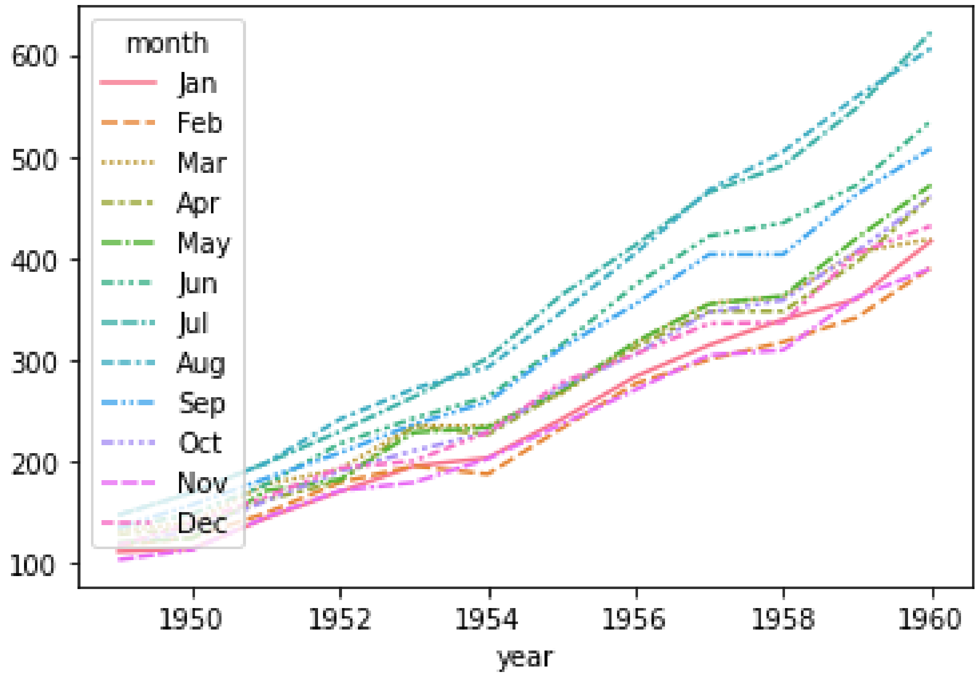
```
[ ]: flights_wide = flights.pivot("year", "month", "passengers")
     flights_wide.head()
```

```
[ ]: month  Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
     year
     1949   112  118  132  129  121  135  148  148  136  119  104  118
     1950   115  126  141  135  125  149  170  170  158  133  114  140
     1951   145  150  178  163  172  178  199  199  184  162  146  166
```

1952	171	180	193	181	183	218	230	242	209	191	172	194
1953	196	196	236	235	229	243	264	272	237	211	180	201

[ ]:
```
sns.lineplot(data=flights_wide)
```

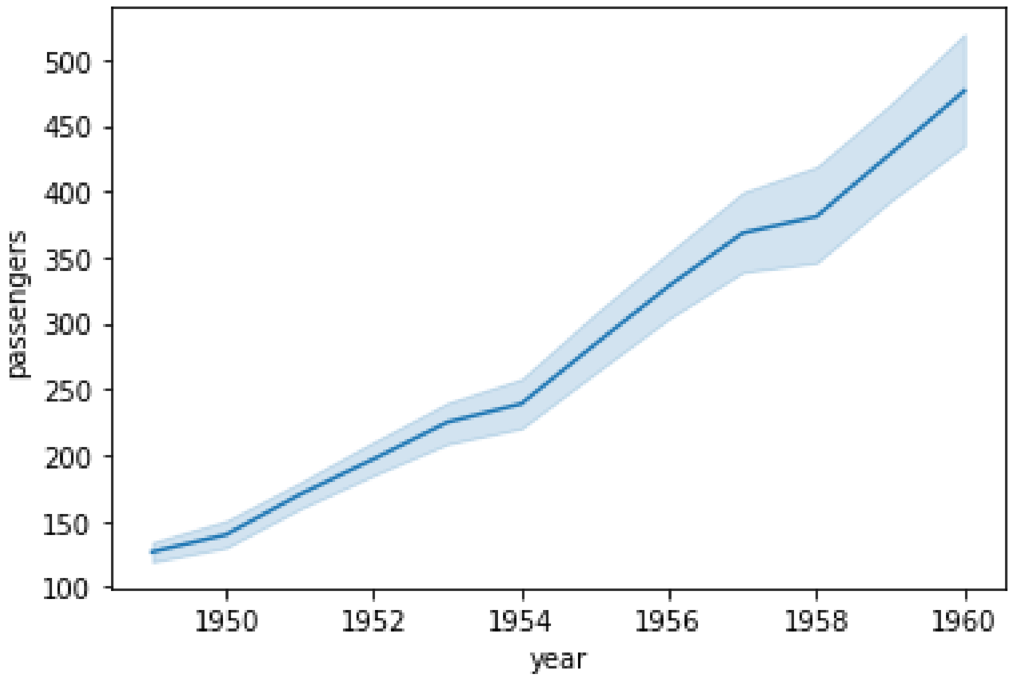
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cb1d9978>



Passing the entire dataset in long-form mode will aggregate over repeated values (each year) to show the mean and 95% confidence interval:

[ ]:
```
sns.lineplot(data=flights, x="year", y="passengers")
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cb975828>



Load another dataset with a numeric grouping variable:

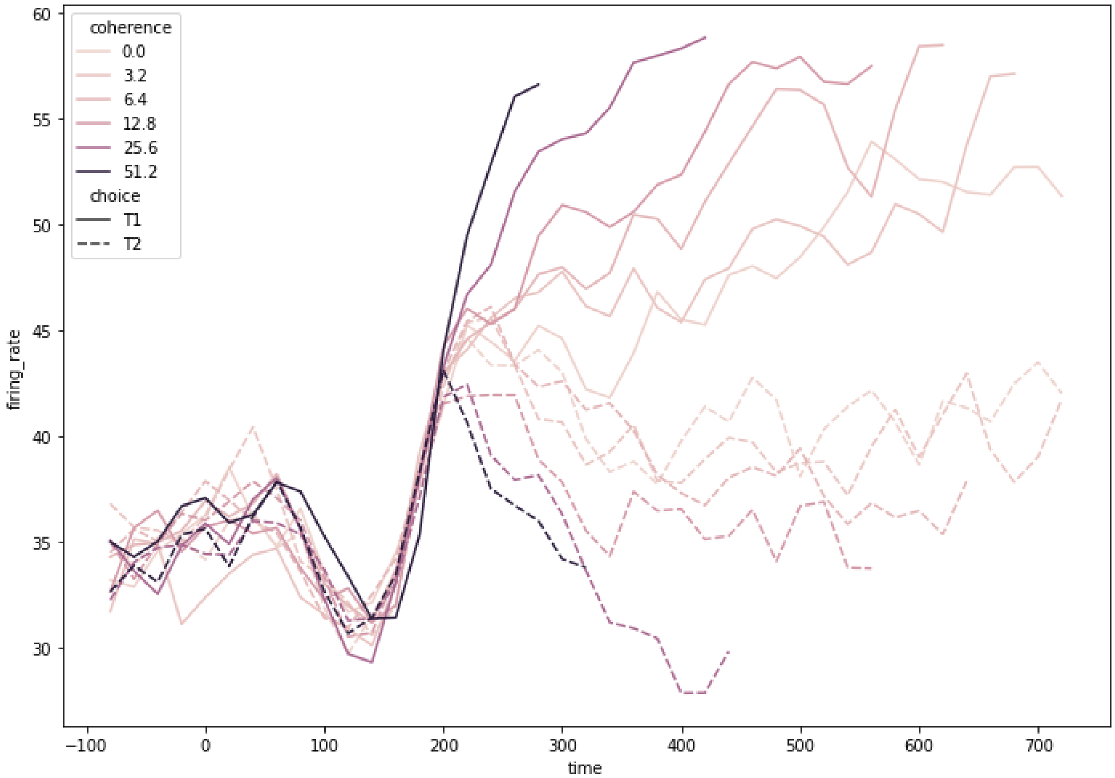
[ ]:
```
dots = sns.load_dataset("dots").query("align == 'dots'")
dots.head()
```

[ ]:

	align	choice	time	coherence	firing_rate
0	dots	T1	-80	0.0	33.189967
1	dots	T1	-80	3.2	31.691726
2	dots	T1	-80	6.4	34.279840
3	dots	T1	-80	12.8	32.631874
4	dots	T1	-80	25.6	35.060487

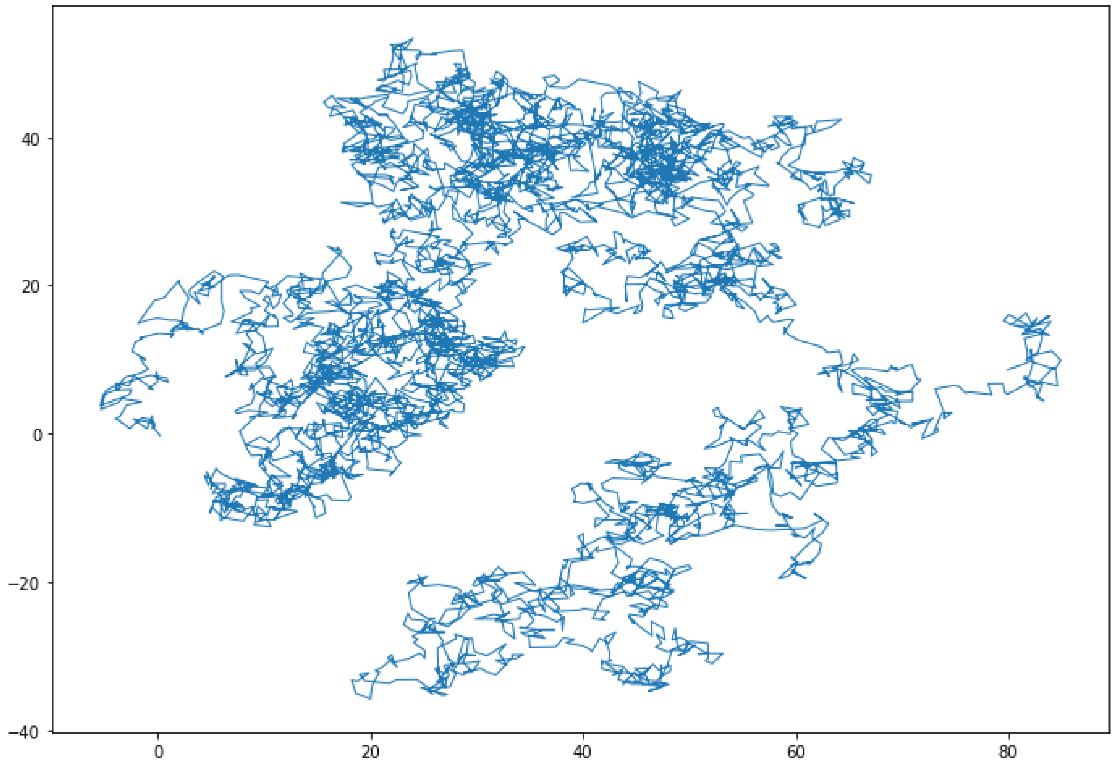
[ ]:
```
# code snippet to increase the fig size
fig, ax = plt.subplots()
# the size of A4 paper
fig.set_size_inches(11.7, 8.27)
sns.lineplot(
    data=dots, x="time", y="firing_rate", hue="coherence", style="choice", ax=ax
)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35cb294630>



[ ]:
```python
# code snippet to increase the fig size
fig, ax = plt.subplots()
# the size of A4 paper
fig.set_size_inches(11.7, 8.27)
x, y = np.random.normal(size=(2, 5000)).cumsum(axis=1)
sns.lineplot(x=x, y=y, sort=False, lw=1, ax = ax)
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f35cab46cf8>
```




```
[ ]: gammas = sns.load_dataset("gammas")
     gammas.head()
```

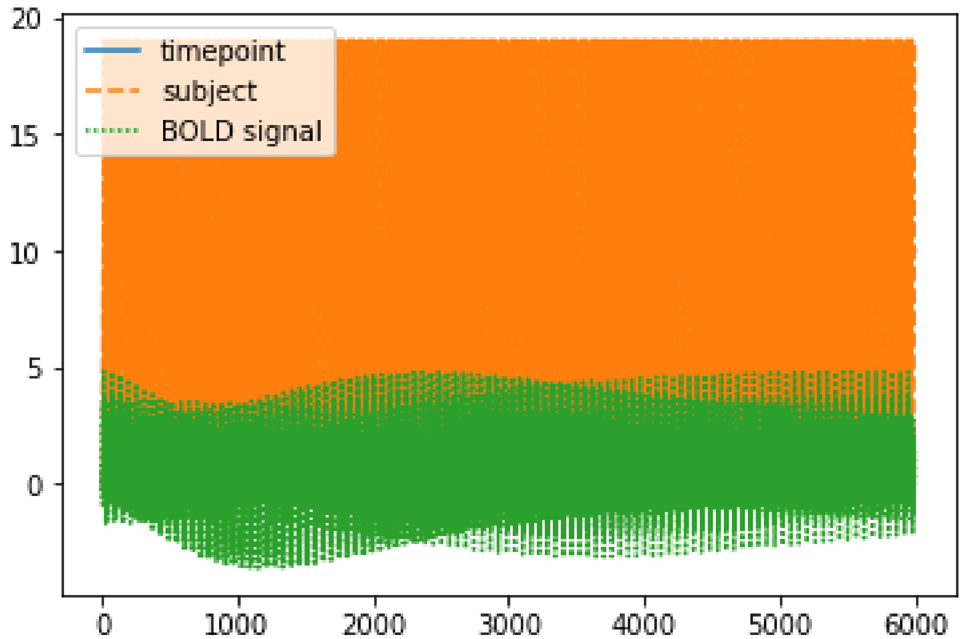
```
[ ]:    timepoint  ROI  subject  BOLD signal
     0        0.0  IPS        0     0.513433
     1        0.0  IPS        1    -0.414368
     2        0.0  IPS        2     0.214695
     3        0.0  IPS        3     0.814809
     4        0.0  IPS        4    -0.894992
```

```
[ ]: gammas.describe()
```

```
[ ]:          timepoint      subject  BOLD signal
     count  6000.000000  6000.000000  6000.000000
     mean      5.000000     9.500000     0.814837
     std       2.916008     5.766762     1.774536
     min       0.000000     0.000000    -3.611603
     25%       2.500000     4.750000    -0.481188
     50%       5.000000     9.500000     0.928425
     75%       7.500000    14.250000     2.169299
     max      10.000000    19.000000     4.829915
```

```
[ ]: sns.lineplot(data=gammas)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f35caaf29b0>



1.19 Data cleaning checklist

- Save original data
- Identify missing data
- Identify placeholder data (e.g. 0's for NA's)
- Identify outliers
- Check for overall plausibility and errors (e.g., typos, unreasonable ranges)
- Identify highly correlated variables
- Identify variables with (nearly) no variance
- Identify variables with strange names or values
- Check variable classes (eg. Characters vs factors)
- Remove/transform some variables (maybe your model does not like categorial variables)
- Rename some variables or values (if not all data is useful)
- Check some overall pattern (statistical/ numerical summaries)
- Possibly center/scale variables

1.20 Exploratory Data Analysis checklist

- Suggest hypotheses about the causes of observed phenomena
- Assess assumptions on which statistical inference will be based
- Support the selection of appropriate statistical tools and techniques
- Provide a basis for further data collection through surveys or experiments

Five methods that are must have:

- Five number summaries (mean/median, min, max, q1, q3)
- Histograms
- Line charts
- Box and whisker plots
- Pairwise scatterplots (scatterplot matrices)
- What values do you see?
- What distributions do you see?
- What relationships do you see?
- What relationships do you think might benefit the prediction problem?
- Answer the following questions for the data in each column:
  - How is the data distributed?
  - Test distribution assumptions (e.G. Normal distributions or skewed?)
  - What are the summary statistics?
  - Are there anomalies/outliers?
- Identify useful raw data & transforms (e.g. log(x))
- Identify data quality problems
- Identify outliers
- Identify subsets of interest
- Suggest functional relationships

Last update September 5, 2017

I2SL_NYC_Property_Sales

December 14, 2023

1 NYC Property Sales

A year's worth of properties sold on the NYC real estate market.

From https://www.kaggle.com/new-york-city/nyc-property-sales

Inspiration

What can you discover about New York City real estate by looking at a year's worth of raw transaction records? Can you spot trends in the market, or build a model that predicts sale value in the future?

Step 1 Look at and understand the data

We may have many questions about these data but the first step in statistical learning is exploratory data analysis (EDA).

1.1 Dataset Description

Properties sold in New York City over a 12-month period from September 2016 to September 2017.

1.1.1 Context

This dataset is a record of every building or building unit (apartment, etc.) sold in the New York City property market over a 12-month period.

1.1.2 Content

This dataset contains the location, address, type, sale price, and sale date of building units sold. A reference on the trickier fields:

BOROUGH: A digit code for the borough the property is located in; in order these are Manhattan (1), Bronx (2), Brooklyn (3), Queens (4), and Staten Island (5).

BLOCK; LOT: The combination of borough, block, and lot forms a unique key for property in New York City. Commonly called a BBL. BUILDING CLASS AT PRESENT and BUILDING CLASS AT TIME OF SALE: The type of building at various points in time.

[ ]:
```
# import libraries
'''
It is good to stick to convention unless there is a good reason to break it
a statement like
```

```
import pandas as pandybear
will cause issues
'''
import pandas as pd
import pandas.util.testing as tm
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import matplotlib.cbook
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:9: FutureWarning:
pandas.util.testing is deprecated. Use the functions in the public API at
pandas.testing instead.
  if __name__ == '__main__':
```

To use H2O.ai we need to

1. Install Java
2. Install H2O
3. Import H2O

[ ]:
```
! apt-get install default-jre
!java -version
```

```
Reading package lists… Done
Building dependency tree
Reading state information… Done
default-jre is already the newest version (2:1.11-68ubuntu1~18.04.1).
default-jre set to manually installed.
The following package was automatically installed and is no longer required:
  libnvidia-common-440
Use 'apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 39 not upgraded.
openjdk version "11.0.8" 2020-07-14
OpenJDK Runtime Environment (build 11.0.8+10-post-Ubuntu-0ubuntu118.04.1)
OpenJDK 64-Bit Server VM (build 11.0.8+10-post-Ubuntu-0ubuntu118.04.1, mixed
mode, sharing)
```

[ ]:
```
! pip install h2o
```

```
Collecting h2o
  Downloading https://files.pythonhosted.org/packages/b7/83/53eb19ffd83e99
ccd77bd1ee9f87b2a663f75f5cb725cdac3eaa004de197/h2o-3.30.1.2.tar.gz (129.4MB)
     |                 | 129.4MB 94kB/s
Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-
packages (from h2o) (2.23.0)
Requirement already satisfied: tabulate in /usr/local/lib/python3.6/dist-
packages (from h2o) (0.8.7)
Requirement already satisfied: future in /usr/local/lib/python3.6/dist-packages
```

```
(from h2o) (0.16.0)
Collecting colorama>=0.3.8
  Downloading https://files.pythonhosted.org/packages/c9/dc/45cdef1b4d119eb96316
b3117e6d5708a08029992b2fee2c143c7a0a5cc5/colorama-0.4.3-py2.py3-none-any.whl
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-
packages (from requests->h2o) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.6/dist-packages (from requests->h2o) (2020.6.20)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.6/dist-packages (from requests->h2o) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.6/dist-packages (from requests->h2o) (3.0.4)
Building wheels for collected packages: h2o
  Building wheel for h2o (setup.py) … done
  Created wheel for h2o: filename=h2o-3.30.1.2-py2.py3-none-any.whl
size=129446949
sha256=577d293d9772be2bfe817c5bd29bce115182357b6f9a124d54a6994c7f68a79e
  Stored in directory: /root/.cache/pip/wheels/c6/be/83/a33a3c1c97fce1d136222bf4
ed6d79da405ef6103f5b434c1e
Successfully built h2o
Installing collected packages: colorama, h2o
Successfully installed colorama-0.4.3 h2o-3.30.1.2
```

[ ]:
```python
# Disable the limit of columns and rows
pd.options.display.max_columns = None
pd.options.display.max_rows = None
df = pd.read_csv("https://raw.githubusercontent.com/nikbearbrown/Google_Colab/
↪master/data/nyc-rolling-sales.csv")
df.head()
```

[ ]:
```
   Unnamed: 0  BOROUGH   NEIGHBORHOOD  \
0           4        1  ALPHABET CITY
1           5        1  ALPHABET CITY
2           6        1  ALPHABET CITY
3           7        1  ALPHABET CITY
4           8        1  ALPHABET CITY

                      BUILDING CLASS CATEGORY TAX CLASS AT PRESENT  BLOCK  \
0 07 RENTALS - WALKUP APARTMENTS                                2A    392
1 07 RENTALS - WALKUP APARTMENTS                                 2    399
2 07 RENTALS - WALKUP APARTMENTS                                 2    399
3 07 RENTALS - WALKUP APARTMENTS                                2B    402
4 07 RENTALS - WALKUP APARTMENTS                                2A    404

   LOT EASE-MENT BUILDING CLASS AT PRESENT                 ADDRESS  \
0    6                                  C2            153 AVENUE B
1   26                                  C7   234 EAST 4TH   STREET
```

```
2   39                                   C7    197 EAST 3RD   STREET
3   21                                   C4      154 EAST 7TH STREET
4   55                                   C2   301 EAST 10TH   STREET

  APARTMENT NUMBER  ZIP CODE  RESIDENTIAL UNITS  COMMERCIAL UNITS  \
0                      10009                  5                 0
1                      10009                 28                 3
2                      10009                 16                 1
3                      10009                 10                 0
4                      10009                  6                 0

   TOTAL UNITS LAND SQUARE FEET GROSS SQUARE FEET  YEAR BUILT  \
0            5             1633              6440        1900
1           31             4616             18690        1900
2           17             2212              7803        1900
3           10             2272              6794        1913
4            6             2369              4615        1900

   TAX CLASS AT TIME OF SALE BUILDING CLASS AT TIME OF SALE SALE PRICE  \
0                          2                              C2    6625000
1                          2                              C7        -
2                          2                              C7        -
3                          2                              C4    3936272
4                          2                              C2    8000000

            SALE DATE
0  2017-07-19 00:00:00
1  2016-12-14 00:00:00
2  2016-12-09 00:00:00
3  2016-09-23 00:00:00
4  2016-11-17 00:00:00
```

1.2 Build a data dictionary

Many of these fields are fairly obvious from their names. ADDRESS, ZIP CODE, etc. Others are less so. BUILDING CLASS AT PRESENT - what does that mean? TAX CLASS AT PRESENT C2 - what does that mean?

Note that the sale price (which is probably in US dollars) is based on a sale date. If one were trying to build a statistical model to predict price this could be an issue. An apartment last sold in 1900 would reflect a very undervalued price.

Class Discussion

How would you address the undervalued price issue?

[ ]:
```
# Descriptive statistics
df.describe()
```

[ ]:
```
       Unnamed: 0       BOROUGH         BLOCK           LOT      ZIP CODE  \
count  84548.000000  84548.000000  84548.000000  84548.000000  84548.000000
mean   10344.359878      2.998758   4237.218976    376.224015  10731.991614
std     7151.779436      1.289790   3568.263407    658.136814   1290.879147
min        4.000000      1.000000      1.000000      1.000000      0.000000
25%     4231.000000      2.000000   1322.750000     22.000000  10305.000000
50%     8942.000000      3.000000   3311.000000     50.000000  11209.000000
75%    15987.250000      4.000000   6281.000000   1001.000000  11357.000000
max    26739.000000      5.000000  16322.000000   9106.000000  11694.000000

       RESIDENTIAL UNITS  COMMERCIAL UNITS   TOTAL UNITS    YEAR BUILT  \
count       84548.000000      84548.000000  84548.000000  84548.000000
mean            2.025264          0.193559      2.249184   1789.322976
std            16.721037          8.713183     18.972584    537.344993
min             0.000000          0.000000      0.000000      0.000000
25%             0.000000          0.000000      1.000000   1920.000000
50%             1.000000          0.000000      1.000000   1940.000000
75%             2.000000          0.000000      2.000000   1965.000000
max          1844.000000       2261.000000   2261.000000   2017.000000

       TAX CLASS AT TIME OF SALE
count               84548.000000
mean                    1.657485
std                     0.819341
min                     1.000000
25%                     1.000000
50%                     2.000000
75%                     2.000000
max                     4.000000
```

1.3 Descriptive statistics

Do the descriptive statistics make sense? What does it mean to have zero residential units? Does it mean that the property has commercial units?

The average year that houses were built is 537. Are houses in NYC really that old? How can we correct that?

Class Discussion

Find something odd about the descriptive statistics. Is it an issue for analysis. How would you deal with it?

[ ]:
```python
# What does this tell you? Why isn't it df.shape()?
print(df.shape)
```

```
(84548, 22)
```

[ ]:
```
# Find missing values. What does this tell you?
missing_values_count = df.isnull().sum()
missing_values_count
```

[ ]:
```
Unnamed: 0                        0
BOROUGH                           0
NEIGHBORHOOD                      0
BUILDING CLASS CATEGORY           0
TAX CLASS AT PRESENT              0
BLOCK                             0
LOT                               0
EASE-MENT                         0
BUILDING CLASS AT PRESENT         0
ADDRESS                           0
APARTMENT NUMBER                  0
ZIP CODE                          0
RESIDENTIAL UNITS                 0
COMMERCIAL UNITS                  0
TOTAL UNITS                       0
LAND SQUARE FEET                  0
GROSS SQUARE FEET                 0
YEAR BUILT                        0
TAX CLASS AT TIME OF SALE         0
BUILDING CLASS AT TIME OF SALE    0
SALE PRICE                        0
SALE DATE                         0
dtype: int64
```

[ ]:
```
# Find zero values. What does this tell you? Does this make sense?
zero_values_count = df.isin([0]).sum()
zero_values_count
```

[ ]:
```
Unnamed: 0                            0
BOROUGH                               0
NEIGHBORHOOD                          0
BUILDING CLASS CATEGORY               0
TAX CLASS AT PRESENT                  0
BLOCK                                 0
LOT                                   0
EASE-MENT                             0
BUILDING CLASS AT PRESENT             0
ADDRESS                               0
APARTMENT NUMBER                      0
ZIP CODE                            982
RESIDENTIAL UNITS                 24783
COMMERCIAL UNITS                  79429
TOTAL UNITS                       19762
```

```
LAND SQUARE FEET                      0
GROSS SQUARE FEET                     0
YEAR BUILT                         6970
TAX CLASS AT TIME OF SALE             0
BUILDING CLASS AT TIME OF SALE        0
SALE PRICE                            0
SALE DATE                             0
dtype: int64
```

[ ]:
```
# What does this do? Explain in your notes.
for column in df.columns:
    if df[column].dtype != 'int64':
        df[column].str.strip()
```

1.4 Zero values

Class Discussion

Are all of these zero values appropriate? Is it an issue for analysis? How would you deal with it?

[ ]:
```
# Is there a difference between these two python statements? Is one preferable?
df_tmp1=df
df_tmp2=df.copy()
```

[ ]:
```
# What do these statements do? Why would one do them?
df_tmp2[['SALE PRICE']]=df_tmp2[['SALE PRICE']].replace(0, df_tmp2[['SALE␣
↪PRICE']].median())
df_tmp2[['YEAR BUILT']]=df_tmp2[['YEAR BUILT']].replace(0, np.nan)
```

1.5 Changing data types

Class Discussion

Would you change any data types? Which ones and why?

[ ]:
```
# What does this do? Explain in your notes.
df['TAX CLASS AT TIME OF SALE'] = df['TAX CLASS AT TIME OF SALE'].
↪astype('category')
df['SALE PRICE'] = pd.to_numeric(df['SALE PRICE'], errors='coerce')
df['TOTAL UNITS'] = pd.to_numeric(df['TOTAL UNITS'], errors='coerce')
df['BOROUGH'] = df['BOROUGH'].astype('category')
df['NEIGHBORHOOD'] = df['NEIGHBORHOOD'].astype('category')
```

[ ]:
```
# Look at the SALE PRICE columns
missing_values_count = df.isnull().sum()
missing_values_count
```

[ ]:
```
Unnamed: 0                            0
BOROUGH                               0
```

```
NEIGHBORHOOD                          0
BUILDING CLASS CATEGORY               0
TAX CLASS AT PRESENT                  0
BLOCK                                 0
LOT                                   0
EASE-MENT                             0
BUILDING CLASS AT PRESENT             0
ADDRESS                               0
APARTMENT NUMBER                      0
ZIP CODE                              0
RESIDENTIAL UNITS                     0
COMMERCIAL UNITS                      0
TOTAL UNITS                           0
LAND SQUARE FEET                      0
GROSS SQUARE FEET                     0
YEAR BUILT                            0
TAX CLASS AT TIME OF SALE             0
BUILDING CLASS AT TIME OF SALE        0
SALE PRICE                        14561
SALE DATE                             0
dtype: int64
```

[ ]:
```
df['SALE PRICE'].head()
```

[ ]:
```
0    6625000.0
1          NaN
2          NaN
3    3936272.0
4    8000000.0
Name: SALE PRICE, dtype: float64
```

[ ]:
```
# What does this do? Explain in your notes.
df[['SALE PRICE']]=df[['SALE PRICE']].replace(np.nan, df[['SALE PRICE']].
 ↪median())
```

[ ]:
```
df['SALE PRICE'].head()
```

[ ]:
```
0    6625000.0
1     530000.0
2     530000.0
3    3936272.0
4    8000000.0
Name: SALE PRICE, dtype: float64
```

[ ]:
```
df['SALE PRICE'].describe()
```

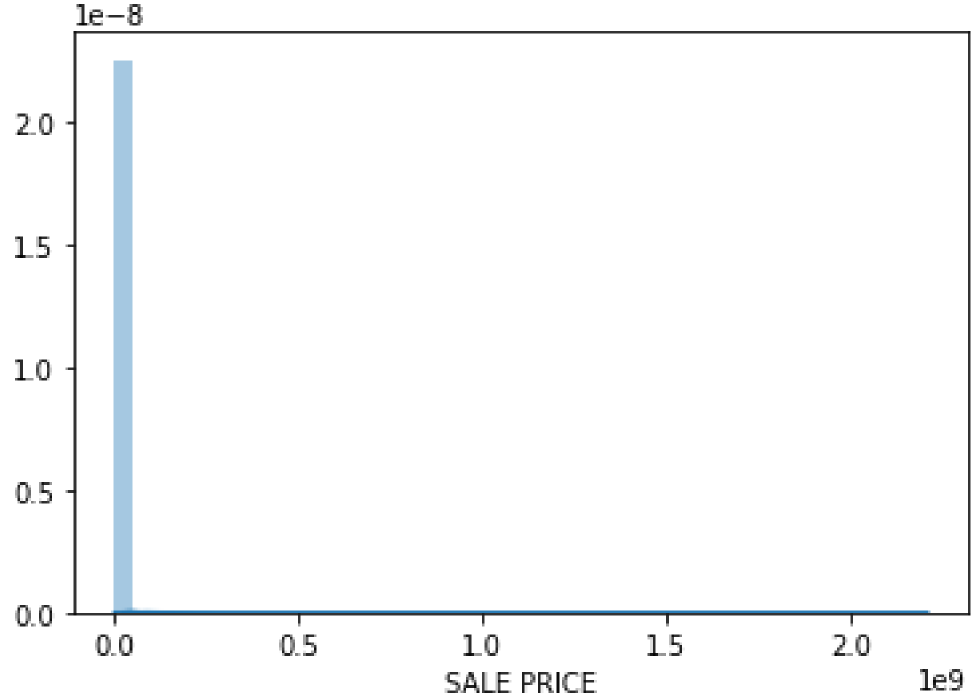
[ ]:
```
count    8.454800e+04
mean     1.147900e+06
std      1.038058e+07
min      0.000000e+00
25%      3.000000e+05
50%      5.300000e+05
75%      8.300000e+05
max      2.210000e+09
Name: SALE PRICE, dtype: float64
```

[ ]:
```python
# Find zero values. What does this tell you? Does this make sense?
zero_values_count = df.isin([0]).sum()
zero_values_count
```

[ ]:
```
Unnamed: 0                            0
BOROUGH                               0
NEIGHBORHOOD                          0
BUILDING CLASS CATEGORY               0
TAX CLASS AT PRESENT                  0
BLOCK                                 0
LOT                                   0
EASE-MENT                             0
BUILDING CLASS AT PRESENT             0
ADDRESS                               0
APARTMENT NUMBER                      0
ZIP CODE                            982
RESIDENTIAL UNITS                 24783
COMMERCIAL UNITS                  79429
TOTAL UNITS                       19762
LAND SQUARE FEET                      0
GROSS SQUARE FEET                     0
YEAR BUILT                         6970
TAX CLASS AT TIME OF SALE             0
BUILDING CLASS AT TIME OF SALE        0
SALE PRICE                        10228
SALE DATE                             0
dtype: int64
```

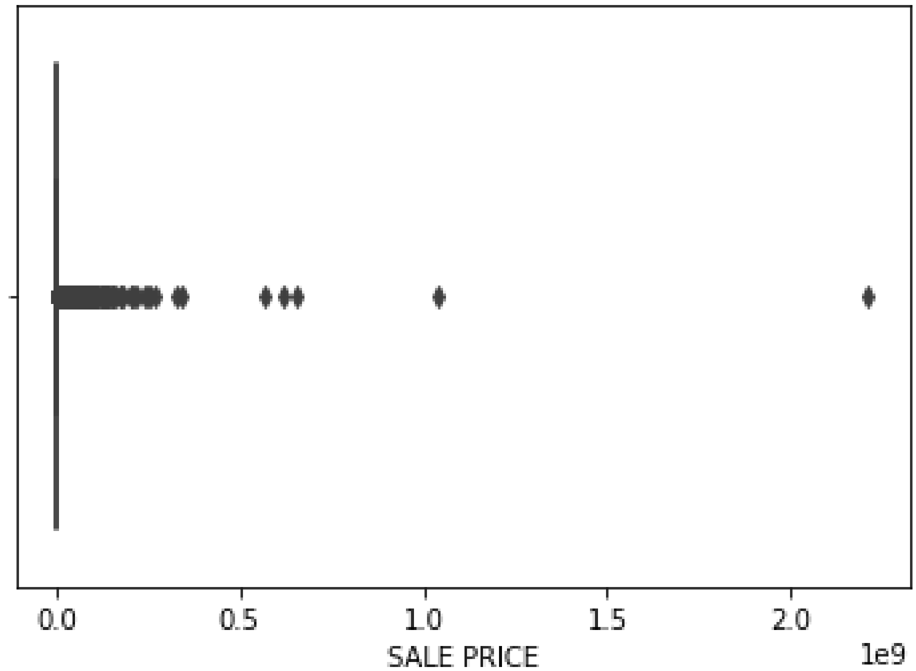
[ ]:
```python
# What does this tell you?
sns.distplot(df['SALE PRICE'])
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fce1c564748>
```




```
[ ]: # What does this tell you?
     sns.boxplot(x='SALE PRICE', data=df)
```

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fce1c48d860>
```



1.6 Where did all of the SALE PRICE zeros come from?

Class Discussion

Where did all of the SALE PRICE zeros come from?

1.7 Replacing values

Class Discussion

Why did SALE PRICE go from zero nan to 14561 nan after converting to numeric?

Would you replace any values? Which ones and why?

[ ]:
```python
# What does this do? Explain in your notes.
del df['EASE-MENT']
```

[ ]:
```python
df.head()
```

[ ]:
```
   Unnamed: 0 BOROUGH   NEIGHBORHOOD  \
0           4       1  ALPHABET CITY
1           5       1  ALPHABET CITY
2           6       1  ALPHABET CITY
3           7       1  ALPHABET CITY
4           8       1  ALPHABET CITY

                      BUILDING CLASS CATEGORY TAX CLASS AT PRESENT  BLOCK  \
0 07 RENTALS - WALKUP APARTMENTS                                2A    392
1 07 RENTALS - WALKUP APARTMENTS                                 2    399
2 07 RENTALS - WALKUP APARTMENTS                                 2    399
3 07 RENTALS - WALKUP APARTMENTS                                2B    402
4 07 RENTALS - WALKUP APARTMENTS                                2A    404

   LOT BUILDING CLASS AT PRESENT                 ADDRESS APARTMENT NUMBER  \
0    6                        C2            153 AVENUE B
1   26                        C7    234 EAST 4TH   STREET
2   39                        C7    197 EAST 3RD   STREET
3   21                        C4      154 EAST 7TH STREET
4   55                        C2   301 EAST 10TH   STREET

   ZIP CODE  RESIDENTIAL UNITS  COMMERCIAL UNITS  TOTAL UNITS  \
0     10009                  5                 0            5
1     10009                 28                 3           31
2     10009                 16                 1           17
3     10009                 10                 0           10
4     10009                  6                 0            6
```

```
  LAND SQUARE FEET GROSS SQUARE FEET  YEAR BUILT TAX CLASS AT TIME OF SALE  \
0             1633              6440        1900                         2
1             4616             18690        1900                         2
2             2212              7803        1900                         2
3             2272              6794        1913                         2
4             2369              4615        1900                         2

  BUILDING CLASS AT TIME OF SALE  SALE PRICE           SALE DATE
0                             C2   6625000.0  2017-07-19 00:00:00
1                             C7    530000.0  2016-12-14 00:00:00
2                             C7    530000.0  2016-12-09 00:00:00
3                             C4   3936272.0  2016-09-23 00:00:00
4                             C2   8000000.0  2016-11-17 00:00:00
```

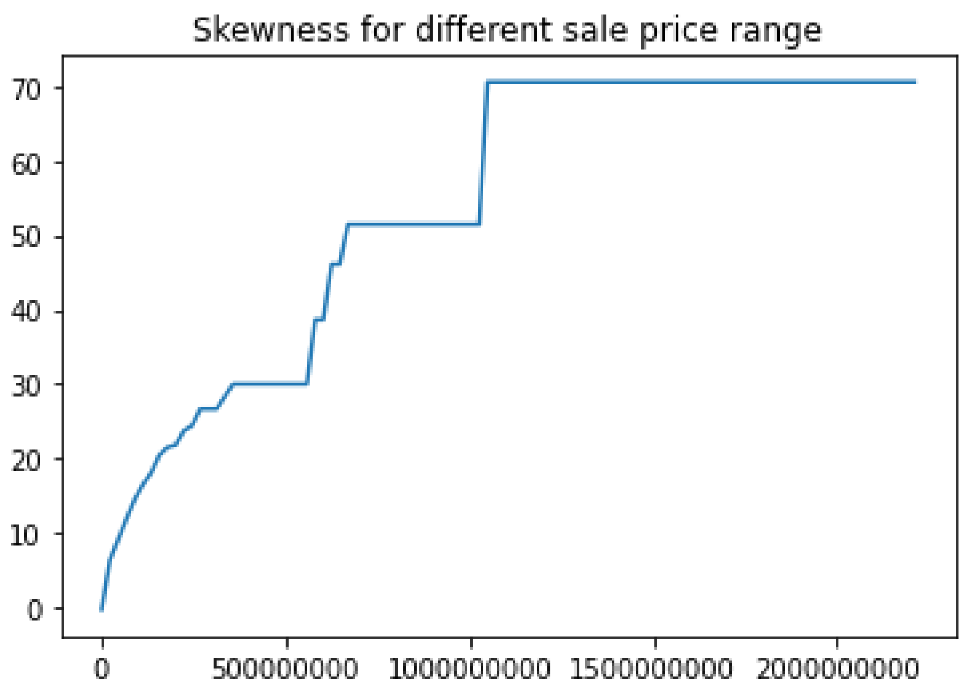
[ ]:
```python
# Data types
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 84548 entries, 0 to 84547
Data columns (total 21 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Unnamed: 0                      84548 non-null  int64
 1   BOROUGH                         84548 non-null  category
 2   NEIGHBORHOOD                    84548 non-null  category
 3   BUILDING CLASS CATEGORY         84548 non-null  object
 4   TAX CLASS AT PRESENT            84548 non-null  object
 5   BLOCK                           84548 non-null  int64
 6   LOT                             84548 non-null  int64
 7   BUILDING CLASS AT PRESENT       84548 non-null  object
 8   ADDRESS                         84548 non-null  object
 9   APARTMENT NUMBER                84548 non-null  object
 10  ZIP CODE                        84548 non-null  int64
 11  RESIDENTIAL UNITS               84548 non-null  int64
 12  COMMERCIAL UNITS                84548 non-null  int64
 13  TOTAL UNITS                     84548 non-null  int64
 14  LAND SQUARE FEET                84548 non-null  object
 15  GROSS SQUARE FEET               84548 non-null  object
 16  YEAR BUILT                      84548 non-null  int64
 17  TAX CLASS AT TIME OF SALE       84548 non-null  category
 18  BUILDING CLASS AT TIME OF SALE  84548 non-null  object
 19  SALE PRICE                      84548 non-null  float64
 20  SALE DATE                       84548 non-null  object
dtypes: category(3), float64(1), int64(8), object(9)
memory usage: 11.9+ MB
```

1.8 Skewness discussion

Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. The skewness value can be positive, zero, negative, or undefined.

[ ]:
```
# What does this tell you?
x = np.linspace(9.500000e+05,2.210000e+09, num=100)

y = [df[(df['SALE PRICE'] < x_range)]['SALE PRICE'].skew() for x_range in x]
sns.lineplot(x=x,y=y)
plt.ticklabel_format(style='plain', axis='x')
plt.title('Skewness for different sale price range')
plt.show()
```



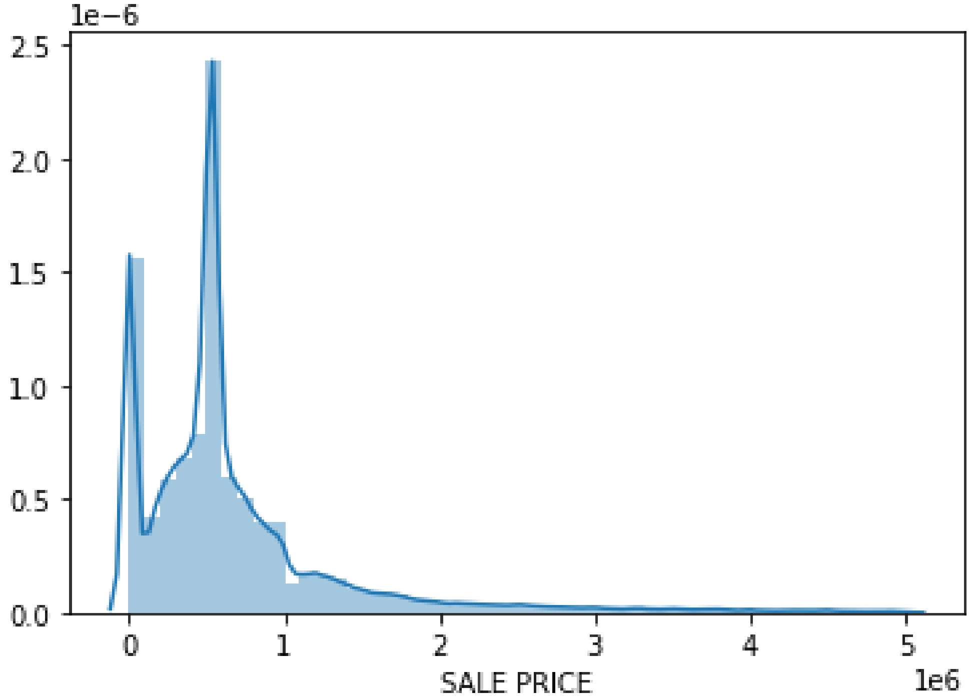
1.9 Skewness discussion

Class Discussion

What does the skewness graph tell you?

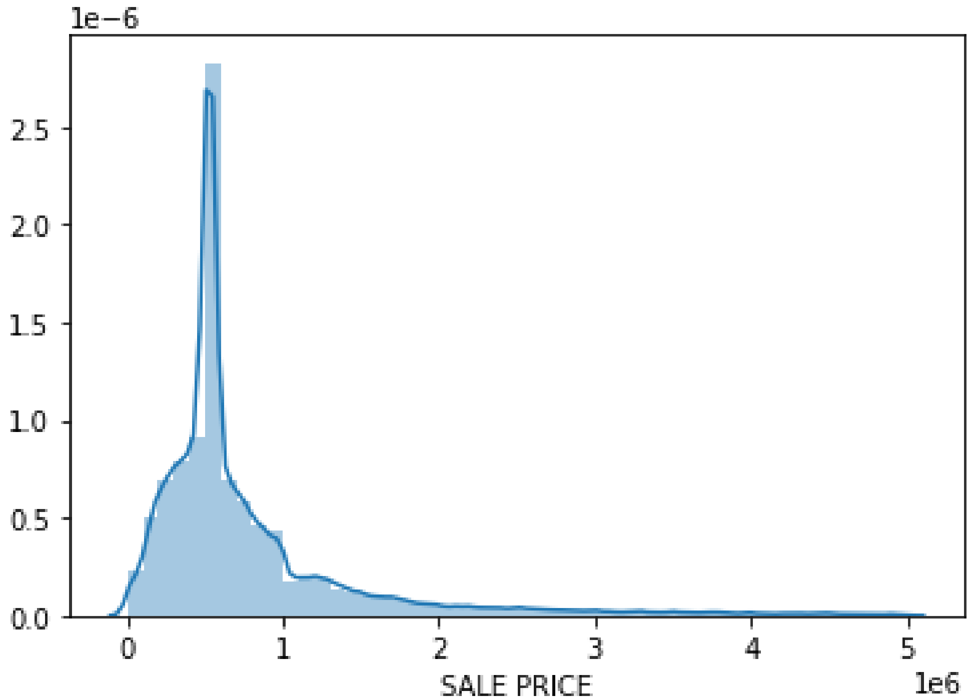
[ ]:
```
# What does this tell you?
sns.distplot(df[(df['SALE PRICE'] < 5e+06)]['SALE PRICE'])
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fce1c3eae10>



[ ]:
```
sns.distplot(df[(df['SALE PRICE'] > 1e+03) & (df['SALE PRICE'] < 5e+06)]['SALE␣
↪PRICE'])
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fce1bde7860>



1.10 Distribution

Class Discussion

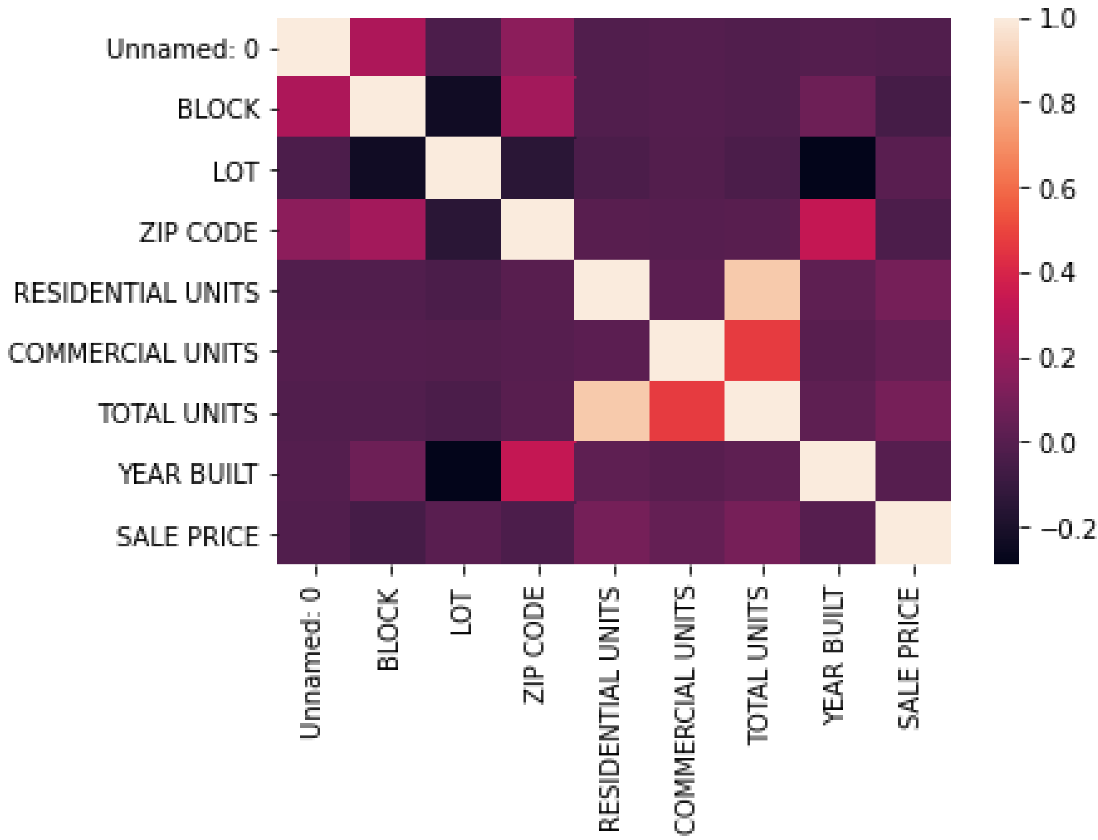
What does the distribution graph tell you?

1.11 Feature Selection

What features are related to the target (dependent variable) SALE PRICE

[ ]:
```
#Correlation between the features
corr = df.corr()
sns.heatmap(corr)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fce1bd82048>



[ ]:
```
#Let's sort the numeric correlation between sale price and other features
corr['SALE PRICE'].sort_values(ascending=False)
```

[ ]:
```
SALE PRICE           1.000000
TOTAL UNITS          0.103112
RESIDENTIAL UNITS    0.094278
COMMERCIAL UNITS     0.043670
LOT                  0.010957
YEAR BUILT          -0.002009
Unnamed: 0          -0.015184
ZIP CODE            -0.029975
BLOCK               -0.054124
Name: SALE PRICE, dtype: float64
```

1.12 Feature Selection Discussion

Class Discussion

Is there any significant correlation?

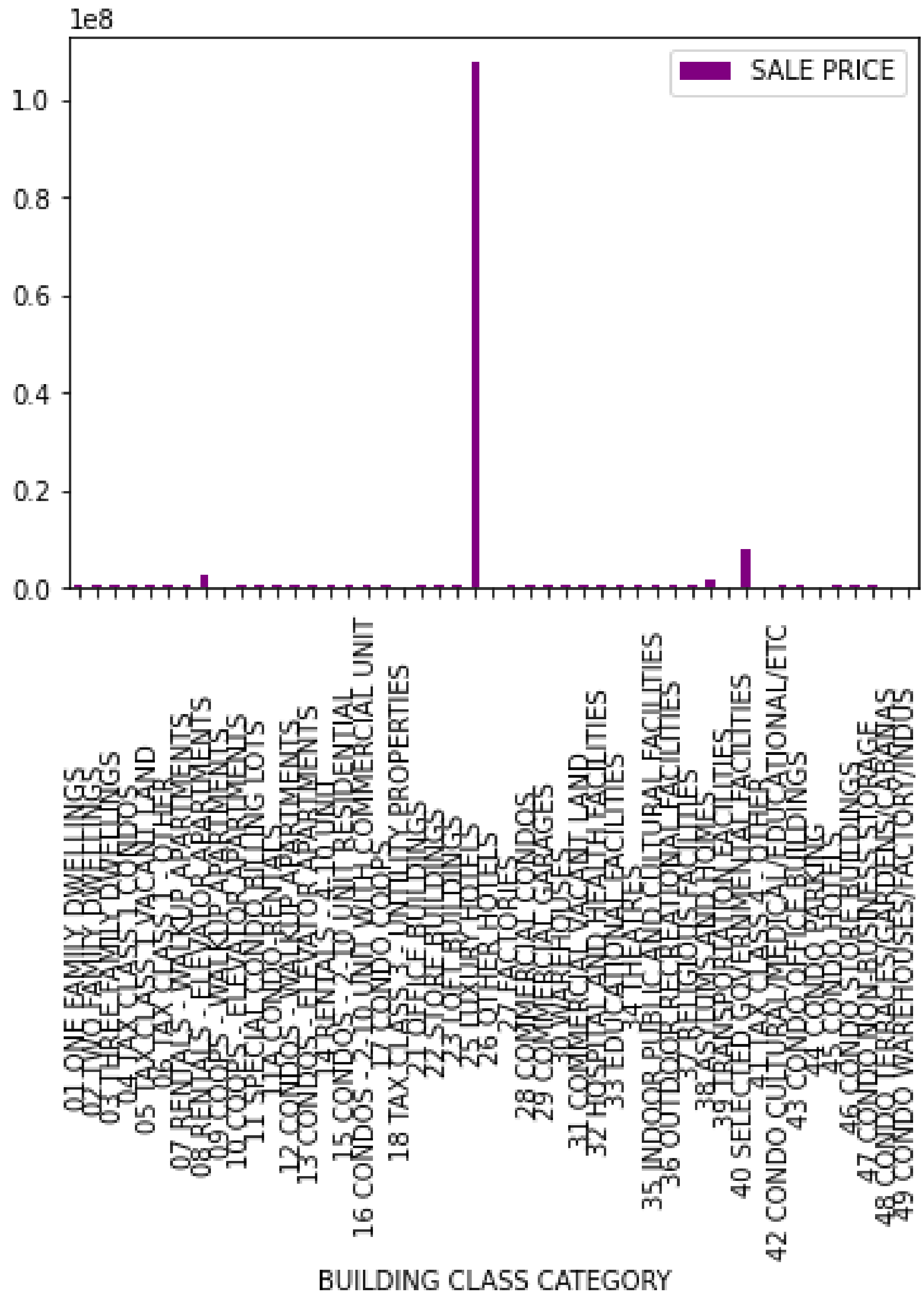
What about categorical variables? Can they be related to the target (dependent variable) SALE PRICE?

What other methods can be used for feature selection?

[ ]:
```
# What does this tell you?
print(df['BUILDING CLASS CATEGORY'].nunique())
pivot=df.pivot_table(index='BUILDING CLASS CATEGORY', values='SALE PRICE',␣
↪aggfunc=np.median)
pivot
pivot.plot(kind='bar', color='Purple')
```

```
47
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fce181d64a8>
```



1.13 A simple analysis

[ ]:
```python
print(df.columns)
```

```
Index(['Unnamed: 0', 'BOROUGH', 'NEIGHBORHOOD', 'BUILDING CLASS CATEGORY',
       'TAX CLASS AT PRESENT', 'BLOCK', 'LOT', 'BUILDING CLASS AT PRESENT',
```

```
       'ADDRESS', 'APARTMENT NUMBER', 'ZIP CODE', 'RESIDENTIAL UNITS',
       'COMMERCIAL UNITS', 'TOTAL UNITS', 'LAND SQUARE FEET',
       'GROSS SQUARE FEET', 'YEAR BUILT', 'TAX CLASS AT TIME OF SALE',
       'BUILDING CLASS AT TIME OF SALE', 'SALE PRICE', 'SALE DATE'],
      dtype='object')
```

[ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 84548 entries, 0 to 84547
Data columns (total 21 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Unnamed: 0                      84548 non-null  int64
 1   BOROUGH                         84548 non-null  category
 2   NEIGHBORHOOD                    84548 non-null  category
 3   BUILDING CLASS CATEGORY         84548 non-null  object
 4   TAX CLASS AT PRESENT            84548 non-null  object
 5   BLOCK                           84548 non-null  int64
 6   LOT                             84548 non-null  int64
 7   BUILDING CLASS AT PRESENT       84548 non-null  object
 8   ADDRESS                         84548 non-null  object
 9   APARTMENT NUMBER                84548 non-null  object
 10  ZIP CODE                        84548 non-null  int64
 11  RESIDENTIAL UNITS               84548 non-null  int64
 12  COMMERCIAL UNITS                84548 non-null  int64
 13  TOTAL UNITS                     84548 non-null  int64
 14  LAND SQUARE FEET                84548 non-null  object
 15  GROSS SQUARE FEET               84548 non-null  object
 16  YEAR BUILT                      84548 non-null  int64
 17  TAX CLASS AT TIME OF SALE       84548 non-null  category
 18  BUILDING CLASS AT TIME OF SALE  84548 non-null  object
 19  SALE PRICE                      84548 non-null  float64
 20  SALE DATE                       84548 non-null  object
dtypes: category(3), float64(1), int64(8), object(9)
memory usage: 11.9+ MB
```

[ ]:
```
# X and y is a common convention for the independent and dependent variables
y='SALE PRICE'
X=['BOROUGH', 'NEIGHBORHOOD', 'RESIDENTIAL UNITS',
       'COMMERCIAL UNITS', 'TOTAL UNITS', 'TAX CLASS AT TIME OF SALE','SALE␣
 ↪PRICE']
Xy=['BOROUGH', 'NEIGHBORHOOD', 'RESIDENTIAL UNITS',
       'COMMERCIAL UNITS', 'TOTAL UNITS', 'TAX CLASS AT TIME OF SALE','SALE␣
 ↪PRICE']
```

[ ]:
```
df=df[Xy]
df.head()
```

```
[ ]:   BOROUGH   NEIGHBORHOOD  RESIDENTIAL UNITS  COMMERCIAL UNITS  TOTAL UNITS  \
0        1  ALPHABET CITY                  5                 0            5
1        1  ALPHABET CITY                 28                 3           31
2        1  ALPHABET CITY                 16                 1           17
3        1  ALPHABET CITY                 10                 0           10
4        1  ALPHABET CITY                  6                 0            6

   TAX CLASS AT TIME OF SALE  SALE PRICE
0                          2   6625000.0
1                          2    530000.0
2                          2    530000.0
3                          2   3936272.0
4                          2   8000000.0
```

1.14 Statistcial learning with an ensemble of machine learning algorithms

Lessons from Kaggle – Ensemble ML and Feature Engineering

99.9% of high ranking Kaggle submissions shared two approaches. Stacking and feature engineering. In this notebook, we will use indivdual models and stacked models to predict lift. Stacking is a type of ensemble, creating a "super-model" by combining many complementary models.

We will generate thousands of individual models, select the best models and combine the best models into a "super-model" to predict lift.

Models and hyperparamter optimization

A model is an algorithm with a given set of hyperparamters. For example, a random forest estimator that uses 10 trees and one that uses 20 trees are two different models. Using a few algorithms and important tuning paramters (hyperparamters) we will try many combination and select rank the models on some metric like AUC, mean residual deviance, RSME as approriate for the analysis.

The machine learning algorithms

We will use the following algorithms as our base:

- Deep Learning (Neural Networks)
- Generalized Linear Model (GLM)
- Extreme Random Forest (XRT)
- Distributed Random Forest (DRF)
- Gradient Boosting Machine (GBM)
- XGBoost

Deep Learning (Neural Networks)

The are simple Multi-Layer Perceptrons (MLPs) as discussed in the first notebook.

Generalized Linear Model (GLM)

The generalized linear model (GLM) is a flexible generalization of ordinary linear regression that allows for response variables that have error distribution models other than a normal distribution. The GLM generalizes linear regression by allowing the linear model to be related to the response variable via a link function and by allowing the magnitude of the variance of each measurement to be a function of its predicted value.

In our case, we will assume that the the distribution of errors is normal and that the link function is the identity, which means the we will be performing simple linear regression. Linear regression predicts the response variable y assuming it has a linear relationship with predictor variable(s) x or $x_1, x_2, , , , x_n$.

$$y = \beta_0 + \beta_1 x + \varepsilon.$$

Distributed Random Forest (DRF)

A Distributed Random Forest (DRF) is a powerful low-bias classification and regression tool that can fit highly non-linear data. To prevent overfitting a DRF generates a forest of classification or regression trees, rather than a single classification or regression tree through a process called bagging. The variance of estimates can be adjusted by the number of trees used.

Extreme Random Forest (XRT)

Extreme random forests are nearly identical to standard random forests except that the splits, both attribute and cut-point, are chosen totally or partially at random. Bias/variance analysis has shown that XRTs work by decreasing variance while at the same time increasing bias. Once the randomization level is properly adjusted, the variance almost vanishes while bias only slightly increases with respect to standard trees.

Gradient Boosting Machine (GBM)

Gradient Boosting Machine (for Regression and Classification) is a forward learning ensemble method. The guiding heuristic is that good predictive results can be obtained through increasingly refined approximations. Boosting can create more accurate models than bagging but doesn't help to avoid overfitting as much as bagging does.

Unlike a DRF which uses bagging to prevent overfitting, a GBM uses boosting to sequentially refine a regression or classification tree. However, as each tree is built in parallel it allows for multi-threading (asynchronous) training large data sets.

As with all tree based methods it creates decision trees and is highly interpretable.

XGBoost

XGBoost is a supervised learning algorithm that implements a process called boosting to yield accurate models. Boosting refers to the ensemble learning technique of building many models sequentially, with each new model attempting to correct for the deficiencies in the previous model.

Both XGBoost and GBM follows the principle of gradient boosting. However, XGBoost has a more regularized model formalization to control overfitting. Boosting does not prevent overfitting the way bagging does, but typically gives better accuracy. XGBoost corrects for the deficiencies of boosting by ensembling regularized trees.

Like a GBM, each tree is built in parallel it allows for multi-threading (asynchronous) training large data sets.

As with all tree based methods it creates decision trees and is highly interpretable.

[ ]:
```
import h2o
from h2o.automl import H2OAutoML
```

[ ]:
```
# Start H2O server
h2o.init()
```

```
Checking whether there is an H2O instance running at http://localhost:54321
… not found.
Attempting to start a local H2O server…
  Java Version: openjdk version "11.0.8" 2020-07-14; OpenJDK Runtime Environment
(build 11.0.8+10-post-Ubuntu-0ubuntu118.04.1); OpenJDK 64-Bit Server VM (build
11.0.8+10-post-Ubuntu-0ubuntu118.04.1, mixed mode, sharing)
  Starting server from /usr/local/lib/python3.6/dist-
packages/h2o/backend/bin/h2o.jar
  Ice root: /tmp/tmp3le4pv11
  JVM stdout: /tmp/tmp3le4pv11/h2o_unknownUser_started_from_python.out
  JVM stderr: /tmp/tmp3le4pv11/h2o_unknownUser_started_from_python.err
  Server is running at http://127.0.0.1:54323
Connecting to H2O server at http://127.0.0.1:54323 … successful.

--------------------------
 ↪-----------------------------------------------------------------
H2O_cluster_uptime:         02 secs
H2O_cluster_timezone:       Etc/UTC
H2O_data_parsing_timezone:  UTC
H2O_cluster_version:        3.30.1.2
H2O_cluster_version_age:    6 days
H2O_cluster_name:           H2O_from_python_unknownUser_wu0xjk
H2O_cluster_total_nodes:    1
H2O_cluster_free_memory:    3.180 Gb
H2O_cluster_total_cores:    2
H2O_cluster_allowed_cores:  2
H2O_cluster_status:         accepting new members, healthy
H2O_connection_url:         http://127.0.0.1:54323
H2O_connection_proxy:       {"http": null, "https": null}
H2O_internal_security:      False
H2O_API_Extensions:         Amazon S3, XGBoost, Algos, AutoML, Core V3,
 ↪TargetEncoder, Core V4
Python_version:             3.6.9 final
--------------------------
 ↪-----------------------------------------------------------------
```

[ ]:
```
# conversion of pandas dataframe to h2o frame
hf = h2o.H2OFrame(df)
```

```
Parse progress: |                                        | 100%
```

[ ]:
```
hf.head()
```

[ ]:

[ ]:
```
# What does this do?
pct_rows=0.80
hf_train, hf_test = hf.split_frame([pct_rows])
print(hf_train.shape)
print(hf_test.shape)
```

```
(67584, 7)
(16964, 7)
```

1.15 Test-train split versus cross validation

Why would one want to take a test-train split? How does this relate to cross validation?

[ ]:
```
# Set up AutoML
aml = H2OAutoML(max_runtime_secs=333)
```

[ ]:
```
aml.train(x=X,y=y,training_frame=hf_train)
```

```
AutoML progress: |                                        | 100%
```

[ ]:
```
aml.leaderboard
```

[ ]:

Class Discussion

- What is the best metric to evaluate model performance?
- Which one would choose over the other - Mean Residual Deviance / RMSE/ MAE / RMSLE?

1.16 RSME comparison and understanding the leader board

The best models after running for a little under 4 minutes is around 0.005 about half of that of the 0.010 RSME that we got our simple MLP in notebook one and a quarter of the 0.017 RSME that we got with a simple MLP with the same independent variables.

When we run for a short time, under 10 minutes, our leaderboard will be biased towards tree-based methods as the deep learners take much more time to converge. It is rare to see deep learners in the top 500 models when we run for less than 5 minutes.

We should still plot the results but before we do that let's discuss a big advantage of these models, model interpretability.

[ ]:
```
# Get best GLM, GBM and XGBoost models
model_index=0
glm_index=0
glm_model=''
```

```
gbm_index=0
gbm_model=''
xgb_index=0
xgb_model=''
aml_leaderboard_df=aml.leaderboard.as_data_frame()
models_dict={}
for m in aml_leaderboard_df['model_id']:
  models_dict[m]=model_index
  if 'StackedEnsemble' not in m:
    break
  model_index=model_index+1

for m in aml_leaderboard_df['model_id']:
  if 'GLM' in m:
    models_dict[m]=glm_index
    break
  glm_index=glm_index+1

for m in aml_leaderboard_df['model_id']:
  if 'GBM' in m:
    models_dict[m]=gbm_index
    break
  gbm_index=gbm_index+1

for m in aml_leaderboard_df['model_id']:
  if 'XGB' in m:
    models_dict[m]=xgb_index
    break
  xgb_index=xgb_index+1
models_dict
```

[ ]:
```
{'GBM_grid__1_AutoML_20200910_041246_model_2': 0,
 'GLM_1_AutoML_20200910_041246': 17,
 'XGBoost_grid__1_AutoML_20200910_041246_model_2': 1}
```

1.17 Examine the Best Model

[ ]:
```
best_model = h2o.get_model(aml.leaderboard[model_index,'model_id'])
best_model.algo
```

[ ]:
```
'gbm'
```

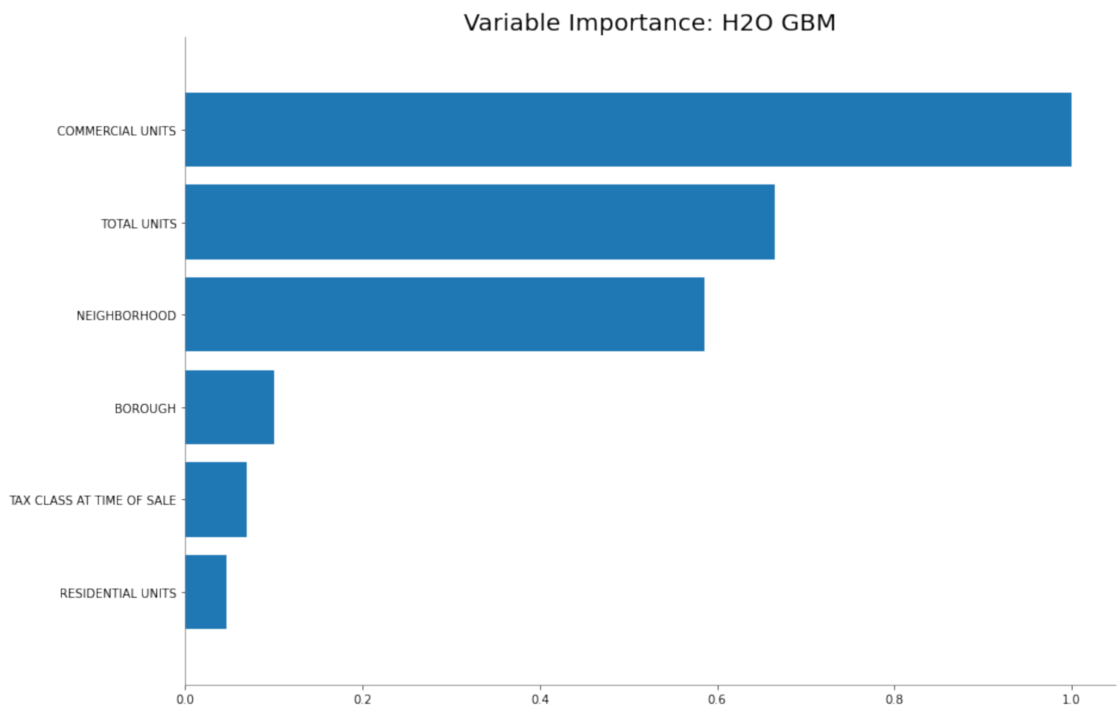
1.18 Variable importance plot

Variable importance plots in tree-based methods provides a list of the most significant variables in descending order by a measure of the information in each variable. Remember that tree calculates the information content of each variable. A variable importance plot is just a bar chart of each variables information content in decreasing order.

It can show actual information estimates or standardized plots like the one below. In a standardized plot the most important variable is always given a value of 1.0. The other variables scores represent their percentage of information relative to the most important variable.

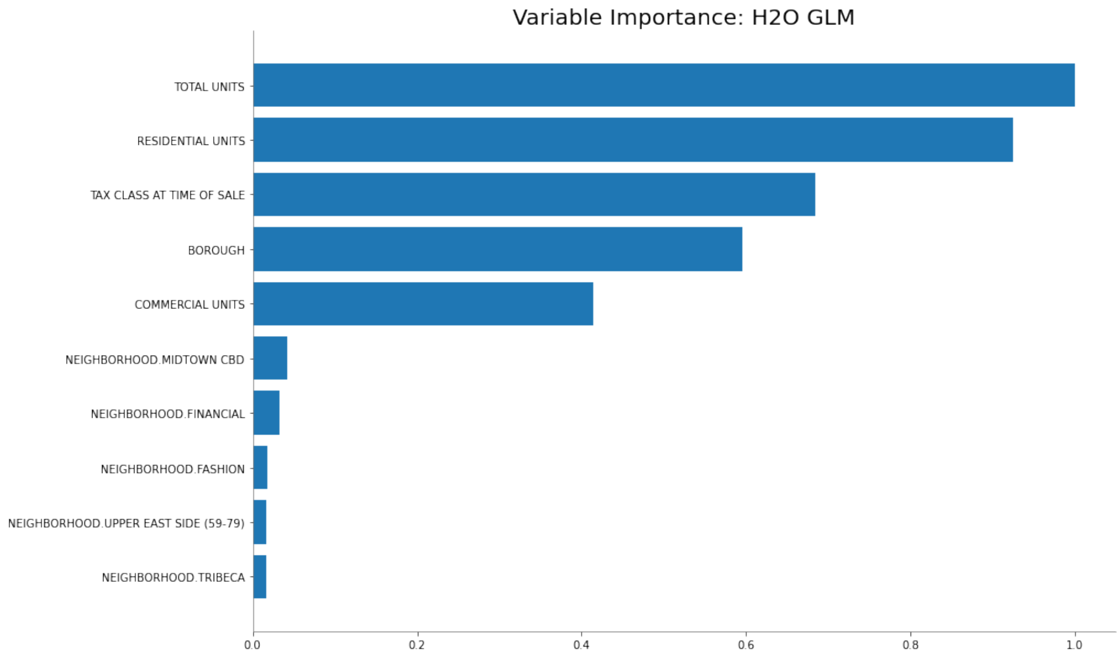
Notice that some varibales have almost no information content. Knowing this allows for feature selection by removing unimportant variables. This makes a model more effecient to run and helps prevent overfitting as the unimportant variables can fit noise and, as we saw in notebook one, make strange predictions.

[ ]:
```
if best_model.algo in ['gbm','drf','xrt','xgboost']:
  best_model.varimp_plot()
```



[ ]:
```
if glm_index is not 0:
  print(glm_index)
  glm_model=h2o.get_model(aml.leaderboard[glm_index,'model_id'])
  print(glm_model.algo)
  print(glm_model.varimp_plot())
```

```
17
glm
```




```
None
```

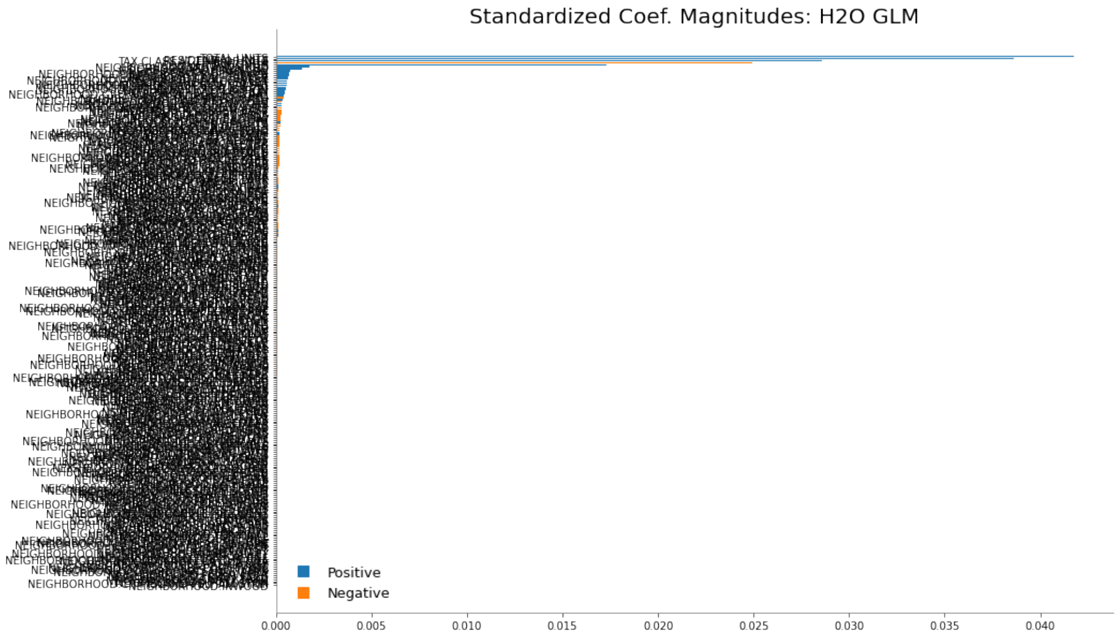
1.19 Variable importance plot discussion

Class Discussion

- Which variables are important according to this plot?
- Did correlation or other analysis agree that the same variables are important? If not, why not?
- What is the value along the X axis on the variable importance graph?
- How would you interpret the differentiation between the varaible importance results of GBM and GLM? How does it help you analyze it further?

[ ]:
```
if glm_index is not 0:
  print(glm_index)
  glm_model=h2o.get_model(aml.leaderboard[glm_index,'model_id'])
  print(glm_model.algo)
glm_model.std_coef_plot()
```

```
17
glm
```



1.20 GLM variable importance plot discussion

Class Discussion

- Which variables are important according to the GLM variable importance plot?
- Did correlation or other analysis agree that the same variables are important? If not, why not?
- Can the GLM variable importance be negative? If so, why?

- Did the GLM variable importance plot create more variables? If so, why?

[ ]:
```
print(best_model.rmse(train = True))
```

```
10555067.468180701
```

[ ]:
```
def model_performance_stats(perf):
    d={}
    try:
      d['mse']=perf.mse()
    except:
      pass
    try:
      d['rmse']=perf.rmse()
    except:
      pass
    try:
      d['null_degrees_of_freedom']=perf.null_degrees_of_freedom()
```

```
    except:
      pass
    try:
      d['residual_degrees_of_freedom']=perf.residual_degrees_of_freedom()
    except:
      pass
    try:
      d['residual_deviance']=perf.residual_deviance()
    except:
      pass
    try:
      d['null_deviance']=perf.null_deviance()
    except:
      pass
    try:
      d['aic']=perf.aic()
    except:
      pass
    try:
      d['logloss']=perf.logloss()
    except:
      pass
    try:
      d['auc']=perf.auc()
    except:
      pass
    try:
      d['gini']=perf.gini()
    except:
      pass
    return d
```

[ ]:
```
mod_perf=best_model.model_performance(hf_test)
stats_test={}
stats_test=model_performance_stats(mod_perf)
stats_test
```

[ ]:
```
{'mse': 33287452623597.574,
 'null_degrees_of_freedom': None,
 'null_deviance': None,
 'residual_degrees_of_freedom': None,
 'residual_deviance': None,
 'rmse': 5769527.9376737205}
```

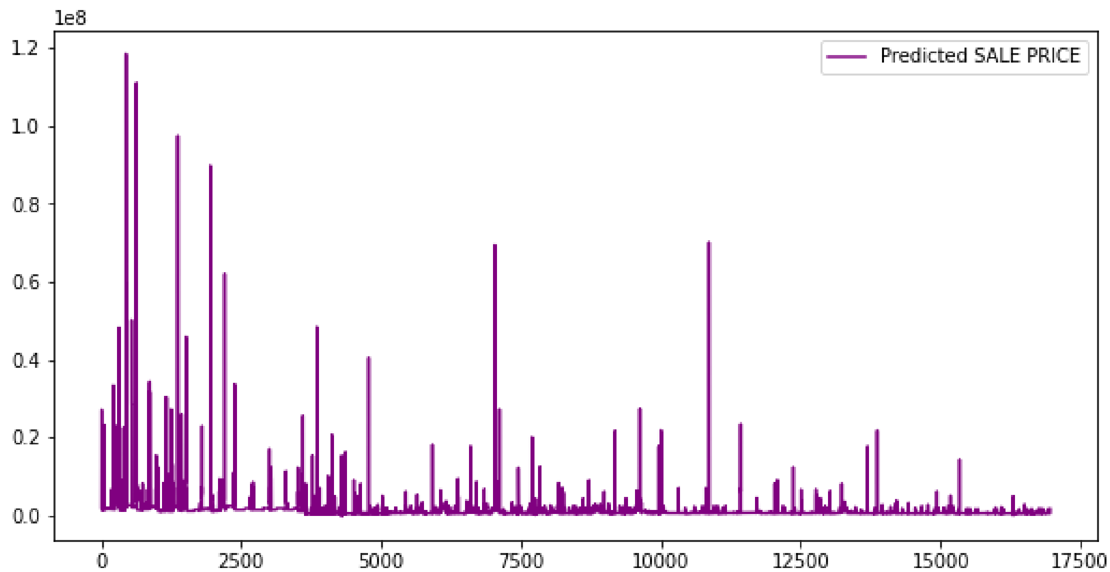
[ ]:
```
predictions = best_model.predict(hf_test)
```

```
gbm prediction progress: |                                 | 100%
```

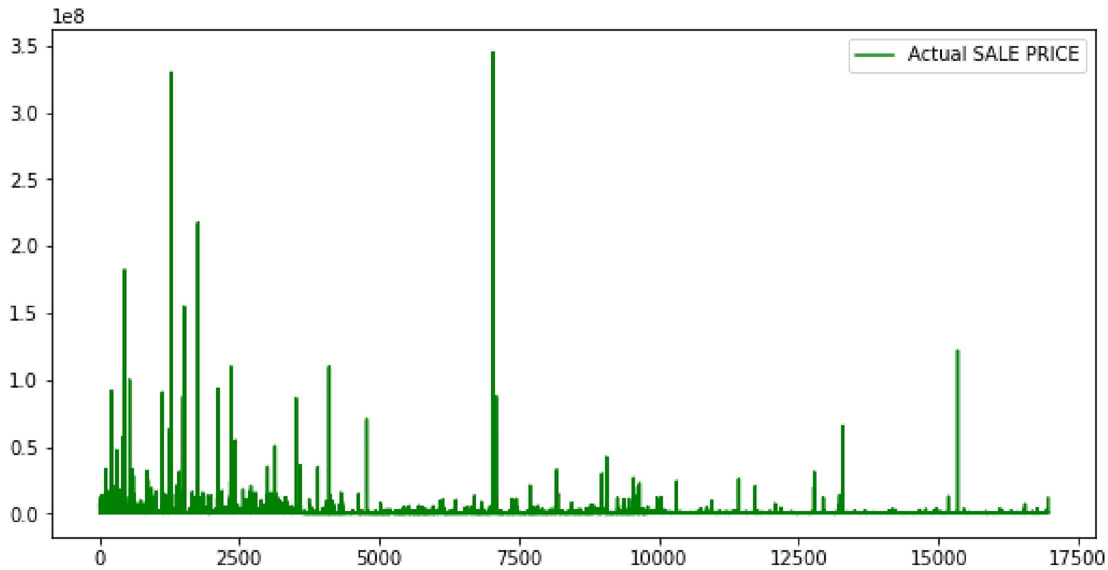
[ ]:
```
y_pred=h2o.as_list(predictions)
y_pred[0:5]
```

[ ]:
```
      predict
0  1.910229e+06
1  7.200150e+06
2  2.832275e+06
3  7.741397e+06
4  2.712342e+07
```

[ ]:
```
df_test=hf_test.as_data_frame()
df_train=hf_train.as_data_frame()
plt.figure(figsize=(10,5))
plt.plot(y_pred, color='purple', label='Predicted SALE PRICE')
plt.legend()
plt.show()
```



[ ]:
```
plt.figure(figsize=(10,5))
plt.plot(df_test['SALE PRICE'], color='g', label='Actual SALE PRICE')
plt.legend()
plt.show()
```



[ ]:
```
y_train_mean=df_train['SALE PRICE'].mean()
print(y_train_mean)
print(best_model.rmse(train = True))
```

```
1154264.1072147253
10555067.468180701
```

1.21 RSME discussion

Class Discussion

Did the model do well? One can think of the error as the mean (null model) plus/minus RSME or the mean (null model) plus/minus MAE. Is this good?

I2SL_Unsupervised_Learning

December 14, 2023

1 Unsupervised Learning

Unsupervised learning is a type of machine learning that looks for previously undetected patterns in a data set with no pre-existing labels and with a minimum of human supervision. In contrast to supervised learning that usually makes use of human-labeled data unsupervised learning, also known as self-organization allows for modeling of probability densities over inputs. It forms one of the three main categories of machine learning, along with supervised and reinforcement learning. Semi-supervised learning , a related variant, makes use of supervised and unsupervised techniques.

Two of the main methods used in unsupervised learning are principal component and cluster analysis . Cluster analysis is used in unsupervised learning to group, or segment, datasets with shared attributes in order to extrapolate algorithmic relationships. Cluster analysis is a branch of machine learning that groups the data that has not been labelled , classified or categorized. Instead of responding to feedback, cluster analysis identifies commonalities in the data and reacts based on the presence or absence of such commonalities in each new piece of data. This approach helps detect anomalous data points that do not fit into either group.

A central application of unsupervised learning is in the field of density estimation in statistics , though unsupervised learning encompasses many other domains involving summarizing and explaining data features. It could be contrasted with supervised learning by saying that whereas supervised learning intends to infer a conditional probability distribution $p_X(x \mid y)$ conditioned on the label y of input data; unsupervised learning intends to infer an a priori probability distribution $p_X(x)$.

Generative adversarial networks can also be used with supervised learning, though they can also be applied to unsupervised and reinforcement techniques.

1.0.1 Unsupervised vs Supervised Learning:

- Most of this course focuses on supervised learning methods such as regression and classification.
- In that setting we observe both a set of features X1;X2; : : : ;Xp for each object, as well as a response or outcome variable Y . The goal is then to predict Y using X1;X2; : : : ;Xp.
- Here we instead focus on unsupervised learning, we where observe only the features X1;X2; : : : ;Xp. We are not interested in prediction, because we do not have an associated response variable Y .

1.0.2 The Goals of Unsupervised Learning

- The goal is to discover interesting things about the measurements: is there an informative way to visualize the data? Can we discover subgroups among the variables or among the observations?
- We discuss two methods:
- principal components analysis, a tool used for data visualization or data pre-processing before supervised techniques are applied, and
- Clustering, a broad class of methods for discovering unknown subgroups in data.

1.0.3 The Challenge of Unsupervised Learning

- Unsupervised learning is more subjective than supervised learning, as there is no simple goal for the analysis, such as prediction of a response.
- But techniques for unsupervised learning are of growing importance in a number of fields:
  - subgroups of breast cancer patients grouped by their gene expression measurements,
  - groups of shoppers characterized by their browsing and purchase histories,
  - movies grouped by the ratings assigned by movie viewers.
- It is often easier to obtain unlabeled data | from a lab instrument or a computer | than labeled data, which can require human intervention.
- For example it is difficult to automatically assess the overall sentiment of a movie review: is it favorable or not?

1.0.4 Principal Components Analysis

- PCA produces a low-dimensional representation of a dataset. It finds a sequence of linear combinations of the variables that have maximal variance, and are mutually uncorrelated.
- Apart from producing derived variables for use in supervised learning problems, PCA also serves as a tool for data visualization.

1.0.5 Principal Components Analysis: details

The first principal component of a set of features $X_1, X_2, \ldots, X_p$ is the normalized linear combination of the features

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \ldots + \phi_{p1}X_p$$

that has the largest variance. By normalized, we mean that $\sum_{j=1}^{p} \phi_{j1}^2 = 1$ - We refer to the elements $\phi_{11}, \ldots, \phi_{p1}$ as the loadings of the first principal component; together, the loadings make up the principal component loading vector, $\phi_1 = \left(\phi_{11}\phi_{21} \ldots \phi_{p1}\right)^T$ - We constrain the loadings so that their sum of squares is equal to one, since otherwise setting these elements to be arbitrarily large in absolute value could result in an arbitrarily large variance.

1.0.6 Computation of Principal Components

- Suppose we have a $n \times p$ data set X. since we are only interested in variance, we assume that each of the variables in X has been centered to have mean zero (that is, the column means of X are zero).
- We then look for the linear combination of the sample feature values of the form

$$z_{i1} = \phi_{11}x_{i1} + \phi_{21}x_{i2} + \ldots + \phi_{p1}x_{ip}$$

for $i = 1, \ldots, n$ that has largest sample variance, subject to the constraint that $\sum_{j=1}^{p} \phi_{j1}^2 = 1$

- Since each of the x_{ij} has mean zero, then so does z_{i1} (for any values of ϕ_{j1}). Hence the sample variance of the z_{i1} can be written as $\frac{1}{n} \sum_{i=1}^{n} z_{i1}^2$
- Plugging in (1) the first principal component loading vector solves the optimization problem

$$\underset{\phi_{11},\ldots,\phi_{p1}}{\text{maximize}} \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{j1} x_{ij} \right)^2 \text{ subject to } \sum_{j=1}^{p} \phi_{j1}^2 = 1$$

- This problem can be solved via a singular-value decomposition of the matrix X, a standard technique in linear algebra.
- We refer to Z_1 as the first principal component, with realized values $z_{11}, \ldots, z_{n1}$

1.0.7 USArrests Data

- USAarrests data: For each of the fifty states in the United States, the data set contains the number of arrests per 100; 000 residents for each of three crimes: Assault, Murder, and Rape. We also record UrbanPop (the percent of the population in each state living in urban areas).
- The principal component score vectors have length n = 50, and the principal component loading vectors have length p = 4.
- PCA was performed after standardizing each variable to have mean zero and standard deviation one.

[ ]:
```
import pandas as pd
import pandas.util.testing as tm
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.model_selection import GridSearchCV

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline


from sklearn.decomposition import PCA
from sklearn.cluster import KMeans, AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram
from sklearn.metrics.pairwise import pairwise_distances

%matplotlib inline
plt.style.use('seaborn-white')
```

[ ]:
```
df = pd.read_csv('https://raw.githubusercontent.com/nikbearbrown/Google_Colab/
↪master/data/usarrests.csv', index_col=0).dropna()
df.head()
```

```
[ ]:             Murder  Assault  UrbanPop  Rape
     Alabama       13.2      236        58  21.2
     Alaska        10.0      263        48  44.5
     Arizona        8.1      294        80  31.0
     Arkansas       8.8      190        50  19.5
     California     9.0      276        91  40.6
```

```
[ ]: X = df.values
     scaled_pca = make_pipeline(StandardScaler(), PCA(n_components=2, whiten=False))
     pca = scaled_pca.named_steps['pca']
     pcaX = scaled_pca.fit_transform(X)
     pcaX = pcaX[:,:2]
```

```
[ ]: # the second pca component is inverted so the plot matches the book
     # the original features as a function of the principal components have been␣
      ↪scaled by 2 so they can be seen easier

     fig, ax = plt.subplots(figsize=(8,8))

     ax.scatter(pcaX[:, 0], pcaX[:, 1], s=0)
     ax.set_xlabel('1st PC')
     ax.set_ylabel('2nd PC')

     for i, txt in enumerate(df_heart.index):
         ax.annotate(txt, (pcaX[i, 0], -pcaX[i, 1]), horizontalalignment='center',␣
      ↪verticalalignment='center', color='b')

     components = pca.components_
     for i, col in enumerate(df_heart.columns.tolist()):
         ax.annotate('', xy=(2*components[0, i], -2*components[1, i]), xytext=(0,␣
      ↪0), arrowprops=dict(arrowstyle="->", ec="orange"))
         ax.text(2*components[0, i], -2*components[1, i], col, size=15,␣
      ↪color='orange')

     ax.set_ylim(ax.get_xlim());
```

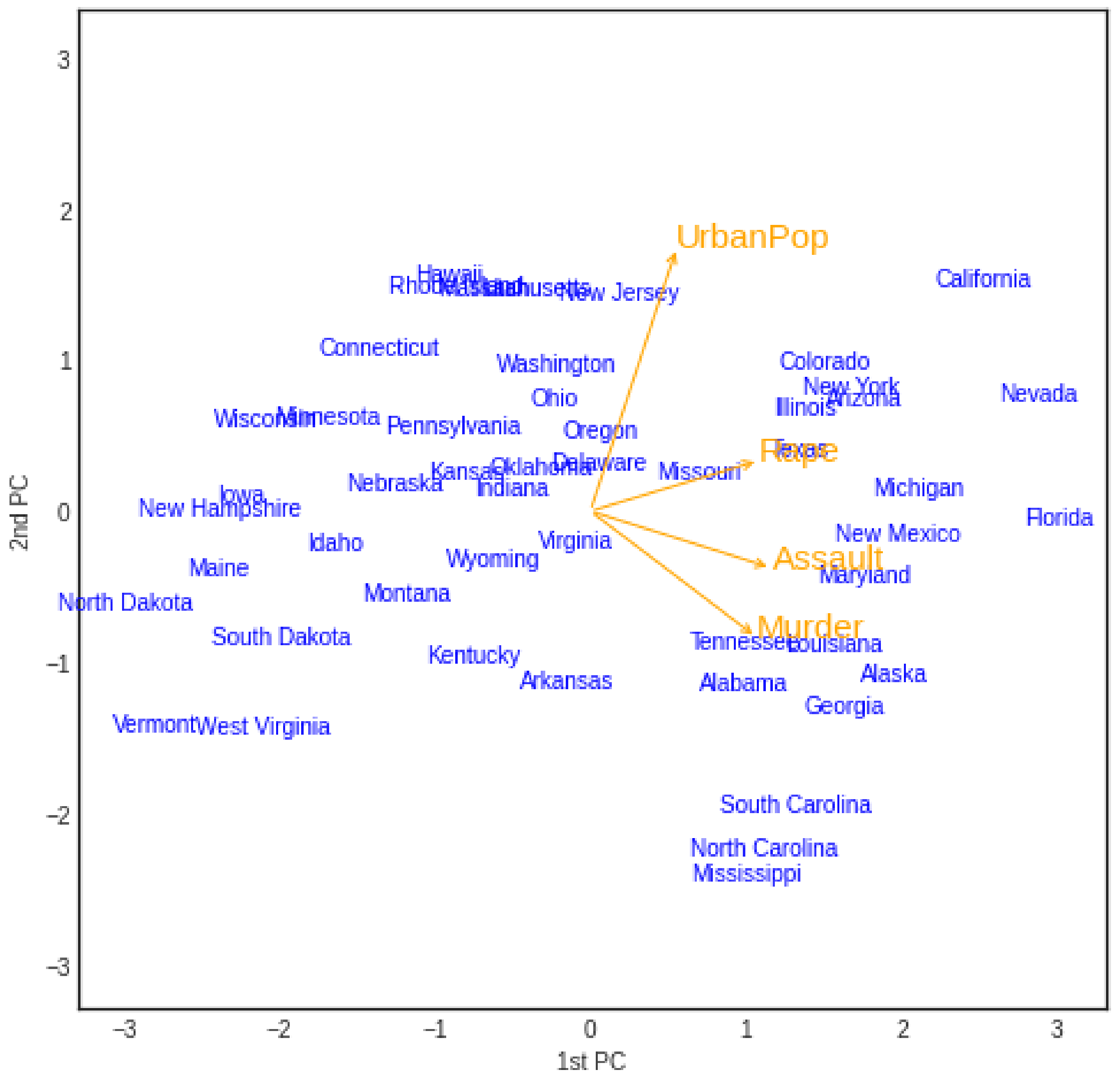


Figure details The first two principal components for the USArrests data. - The blue state names represent the scores for the first two principal components. - The orange arrows indicate the first two principal component loading vectors (with axes on the top and right). For example, the loading for Rape on the first component is 0:54, and its loading on the second principal component 0:17 [the word Rape is centered at the point (0:54; 0:17)]. - This figure is known as a biplot, because it displays both the principal component scores and the principal component loadings.

1.0.8 Proportion Variance Explained

- To understand the strength of each component, we are interested in knowing the proportion of variance explained (PVE) by each one.
- The total variance present in a data set (assuming that the variables have been centered to

have mean zero) is defined as

$$\sum_{j=1}^{p} \operatorname{Var}\left(X_j\right) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2$$

and the variance explained by the m th principal component is

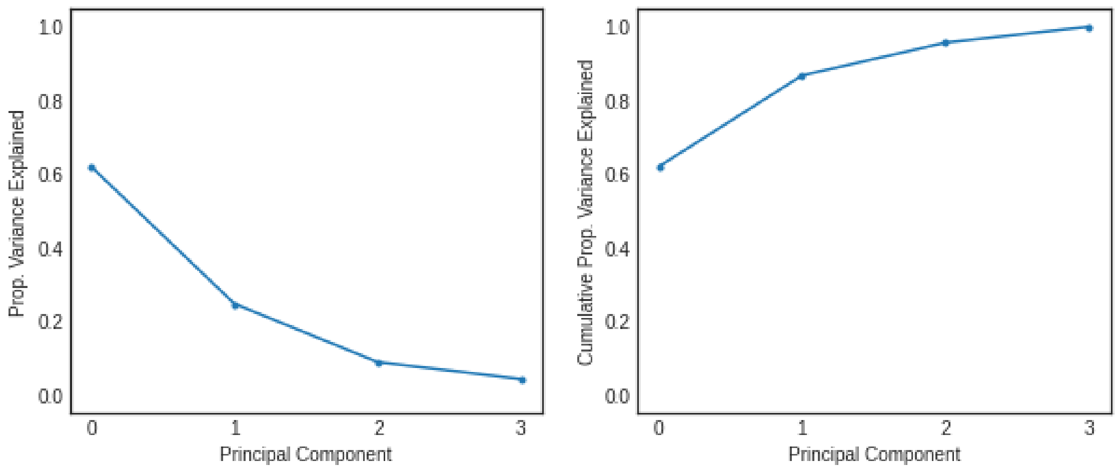
$$\operatorname{Var}\left(Z_m\right) = \frac{1}{n} \sum_{i=1}^{n} z_{im}^2$$

- It can be shown that $\sum_{j=1}^{p} \operatorname{Var}\left(X_j\right) = \sum_{m=1}^{M} \operatorname{Var}\left(Z_m\right)$ with $M = \min(n-1, p)$
- Therefore, the PVE of the m th principal component is given by the positive quantity between 0 and 1 $\frac{\sum_{i=1}^{n} z_{im}^2}{\sum_{j=1}^{p=1} \sum_{i=1} x_{ij}^2}$
- The PVEs sum to one. We sometimes display the cumulative PVEs.

[ ]:
```
scaled_pca = make_pipeline(StandardScaler(), PCA())
pca = scaled_pca.named_steps['pca']
scaled_pca.fit(X)
```

[ ]:
```
Pipeline(memory=None,
         steps=[('standardscaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('pca',
                 PCA(copy=True, iterated_power='auto', n_components=None,
                     random_state=None, svd_solver='auto', tol=0.0,
                     whiten=False))],
         verbose=False)
```

[ ]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,4))

ax1.plot(range(0, pca.n_components_), pca.explained_variance_ratio_, '.-')
ax1.set_ylabel('Prop. Variance Explained')

ax2.plot(range(0, pca.n_components_), np.cumsum(pca.explained_variance_ratio_),␣
 ↪'.-')
ax2.set_ylabel('Cumulative Prop. Variance Explained')

for ax in (ax1, ax2):
    ax.set_ylim(bottom=-0.05, top=1.05)
    ax.get_xaxis().set_major_locator(mpl.ticker.MaxNLocator(integer=True))
    ax.set_xlabel('Principal Component')
```



1.0.9 How many principal components should we use?

If we use principal components as a summary of our data, how many components are sufficient? - No simple answer to this question, as cross-validation is not available for this purpose. - Why not? - When could we use cross-validation to select the number of components? - the "screen plot" on the previous slide can be used as a guide: we look for an "elbow".

1.0.10 Clustering

- Clustering refers to a very broad set of techniques for finding subgroups, or clusters, in a data set.
- We seek a partition of the data into distinct groups so that the observations within each group are quite similar to each other,
- It make this concrete, we must define what it means for two or more observations to be similar or diffierent.
- Indeed, this is often a domain-specific consideration that must be made based on knowledge of the data being studied.

1.0.11 PCA vs Clustering

- PCA looks for a low-dimensional representation of the observations that explains a good fraction of the variance.
- Clustering looks for homogeneous subgroups among the observations.

1.0.12 Clustering for Market Segmentation

- Suppose we have access to a large number of measurements (e.g. median household income, occupation, distance from nearest urban area, and so forth) for a large number of people.
- Our goal is to perform market segmentation by identifying subgroups of people who might be more receptive to a particular form of advertising, or more likely to purchase a particular product.

- The task of performing market segmentation amounts to clustering the people in the data set.

1.0.13 Two clustering methods

- In K-means clustering, we seek to partition the observations into a pre-specified number of clusters.
- In hierarchical clustering, we do not know in advance how many clusters we want; in fact, we end up with a tree-like visual representation of the observations, called a dendrogram, that allows us to view at once the clusterings obtained for each possible number of clusters, from 1 to n.

1.0.14 Details of K-means clustering

Let $C_1, \ldots, C_K$ denote sets containing the indices of the observations in each cluster. These sets satisfy two properties: 1. $C_1 \cup C_2 \cup \ldots \cup C_K = \{1, \ldots, n\}$. In other words, each observation belongs to at least one of the K clusters. 2. $C_k \cap C_{k'} = \emptyset$ for all $k \neq k'$. In other words, the clusters are non-overlapping: no observation belongs to more than one cluster.

For instance, if the i th observation is in the k th cluster, then $i \in C_k$

- The idea behind K -means clustering is that a good clustering is one for which the within-cluster variation is as small as possible.
- The within-cluster variation for cluster C_k is a measure $\mathrm{WCV}\left(C_k\right)$ of the amount by which the observations within a cluster differ from each other.
- Hence we want to solve the problem

$$\underset{C_1, \ldots, C_K}{\operatorname{minimize}}\left\{\sum_{k=1}^K \mathrm{WCV}\left(C_k\right)\right\}$$

  In words, this formula says that we want to partition the observations into K clusters such that the total within-cluster variation, summed over all K clusters, is as small as possible.

1.0.15 K-Means Clustering Algorithm

1. Randomly assign a number, from 1 to K, to each of the observations. These serve as initial cluster assignments for the observations.
2. Iterate until the cluster assignments stop changing: 2.1 For each of the K clusters, compute the cluster centroid. The kth cluster centroid is the vector of the p feature means for the observations in the kth cluster. 2.2 Assign each observation to the cluster whose centroid is closest (where closest is de ned using Euclidean distance).

1.0.16 Hierarchical Clustering

- K-means clustering requires us to pre-specify the number of clusters K. This can be a disadvantage (later we discuss strategies for choosing K)
- Hierarchical clustering is an alternative approach which does not require that we commit to a particular choice of

K.

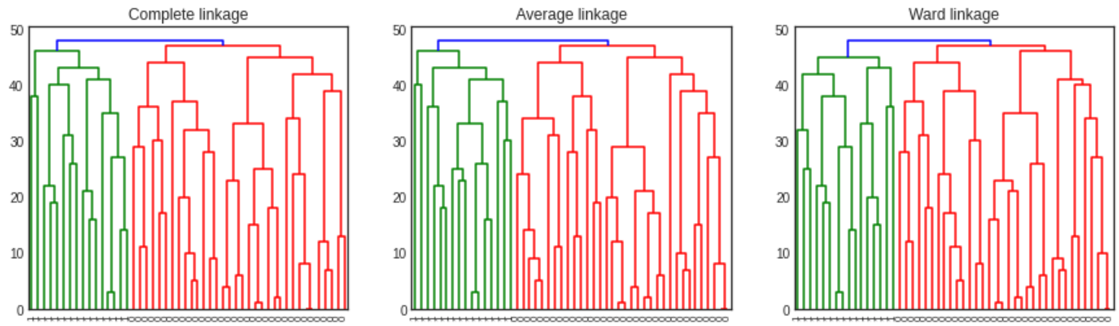
- In this section, we describe bottom-up or agglomerative clustering. This is the most common type of hierarchical clustering, and refers to the fact that a dendrogram is built starting from the leaves and combining clusters up to the trunk.

1.0.17 Hierarchical Clustering Algorithm

The approach in words: - Start with each point in its own cluster. - Identify the closest two clusters and merge them. - Repeat. - Ends when all points are in a single cluster.

[ ]:
```
agg_complete = AgglomerativeClustering(affinity='euclidean',␣
 ↪linkage='complete').fit(X)
agg_average = AgglomerativeClustering(affinity='euclidean', linkage='average').
 ↪fit(X)
agg_ward = AgglomerativeClustering(affinity='euclidean', linkage='ward').fit(X)
def plot_dendrogram(model, labels=None, **kwargs):
    # Children of hierarchical clustering
    children = model.children_

    # Distances between each pair of children
    # Since we don't have this information, we can use a uniform one for␣
 ↪plotting
    distance = np.arange(children.shape[0])

    # The number of observations contained in each cluster level
    no_of_observations = np.arange(2, children.shape[0]+2)

    # Create linkage matrix and then plot the dendrogram
    linkage_matrix = np.column_stack([children, distance, no_of_observations]).
 ↪astype(float)

    if labels is None:
        labels = model.labels_
    else:
        labels = [f'{lab1}_{lab2}' for lab1, lab2 in zip(model.labels_, labels)]

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, labels=labels, **kwargs)
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15,4))

color_threshold = 48

ax1.set_title('Complete linkage')
ax2.set_title('Average linkage')
ax3.set_title('Ward linkage')

plot_dendrogram(agg_complete, color_threshold=color_threshold, ax=ax1)
plot_dendrogram(agg_average, color_threshold=color_threshold, ax=ax2)
```

```
plot_dendrogram(agg_ward, color_threshold=color_threshold, ax=ax3)
```



1.0.18 Details of previous figure

- Left: Dendrogram obtained from hierarchically clustering the data from previous slide, with complete linkage and Euclidean distance.
- Center: The dendrogram from the left-hand panel, cut at a height of 9 (indicated by the dashed line). This cut results in two distinct clusters, shown in different colors.
- Right: The dendrogram from the left-hand panel, now cut at a height of 5. This cut results in three distinct clusters, shown in different colors. Note that the colors were not used in clustering, but are simply used for display purposes in this figure

1.0.19 Summary

- Unsupervised learning is important for understanding the variation and grouping structure of a set of unlabeled data, and can be a useful pre-processor for supervised learning
- It is intrinsically more difficult than supervised learning because there is no gold standard (like an outcome variable) and no single objective (like test set accuracy)
- It is an active field of research, with many recently developed tools such as self-organizing maps, independent components analysis and spectral clustering.

1.0.20 End of Chapter 10

NBB_Intro_Python_Data_Structures

December 14, 2023

0.1 Intro to Python Data Structures

0.1.1 Learning tips.

- Practice, practice, practice.
- Get used to making mistakes! It's OK.
- Don't memorize. There are thousands of packages in python. Learn to read the documentation.

0.1.2 Strings

[ ]:
```
from __future__ import print_function
x = 'Rick'
y = 'Morty'
print(x + ' & ' + y)
```

```
Rick & Morty
```

[ ]:
```
print(x*11)
```

```
RickRickRickRickRickRickRickRickRickRickRick
```

[ ]:
```
print(x[2:])
```

```
ck
```

[ ]:
```
z='bear'
print(z)
print(str.capitalize(z))
```

```
bear
Bear
```

[ ]:
```
z.isdigit()
```

[ ]:
```
False
```

[ ]:
```
z='5'
z.isdigit()
```

[ ]: True

[ ]:
```
print(y)
print(y.replace('o', 'OOOOO'))
print(y)
```

```
Morty
MOOOOOrty
Morty
```

0.2 Data Structures

0.3 Arrays

[ ]:
```
import array as arr
print(arr.typecodes)
```

```
bBuhHiIlLqQfd
```

Type code C Type Python Type Minimum size in bytes
'b' signed char int 1
'B' unsigned char int 1
'u' Py_UNICODE Unicode character 2 (1)
'h' signed short int 2
'H' unsigned short int 2
'i' signed int int 2
'I' unsigned int int 2
'l' signed long int 4
'L' unsigned long int 4
'q' signed long long int 8 (2)
'Q' unsigned long long int 8 (2)
'f' float float 4
'd' double float 8

[ ]:
```
a = arr.array("u",['3','5','7'])
print(type(a))
print(a)
print(a[0])
print(type(a[0]))
```

```
<class 'array.array'>
array('u', '357')
3
<class 'str'>
```

[ ]:
```
a = arr.array("B",[3,5,7])
print(type(a))
print(a)
print(a[0])
print(type(a[0]))
```

```
a.append(9)
print(a)
```

```
<class 'array.array'>
array('B', [3, 5, 7])
3
<class 'int'>
array('B', [3, 5, 7, 9])
```

The following data items and methods are also supported:

array.typecode The typecode character used to create the array.

array.itemsize The length in bytes of one array item in the internal representation.

array.append(x) Append a new item with value x to the end of the array.

array.buffer_info() Return a tuple (address, length) giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as array.buffer_info()[1] * array.itemsize. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain ioctl() operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

Note When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in Buffer Protocol. array.byteswap() "Byteswap" all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, RuntimeError is raised. It is useful when reading data from a file written on a machine with a different byte order.

array.count(x) Return the number of occurrences of x in the array.

array.extend(iterable) Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, TypeError will be raised. If iterable is not an array, it must be iterable and its elements must be the right type to be appended to the array.

array.frombytes(s) Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the fromfile() method).

New in version 3.2: fromstring() is renamed to frombytes() for clarity.

array.fromfile(f, n) Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won't do.

array.fromlist(list) Append items from the list. This is equivalent to for x in list: a.append(x) except that if there is a type error, the array is unchanged.

array.fromstring() Deprecated alias for frombytes().

array.fromunicode(s) Extends this array with data from the given unicode string. The array must be a type 'u' array; otherwise a ValueError is raised. Use array.frombytes(unicodestring.encode(enc)) to append Unicode data to an array of some other type.

array.index(x) Return the smallest i such that i is the index of the first occurrence of x in the array.

array.insert(i, x) Insert a new item with value x in the array before position i. Negative values are treated as being relative to the end of the array.

array.pop([i]) Removes the item with the index i from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.

array.remove(x) Remove the first occurrence of x from the array.

array.reverse() Reverse the order of the items in the array.

array.tobytes() Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the tofile() method.)

New in version 3.2: tostring() is renamed to tobytes() for clarity.

array.tofile(f) Write all items (as machine values) to the file object f.

array.tolist() Convert the array to an ordinary list with the same items.

array.tostring() Deprecated alias for tobytes().

array.tounicode() Convert the array to a unicode string. The array must be a type 'u' array; otherwise a ValueError is raised. Use array.tobytes().decode(enc) to obtain a unicode string from an array of some other type.

[ ]:
```
b=arr.array('l')
print(b)
c=arr.array('u', 'hello \u2641')
print(c)
d=arr.array('l', [1, 2, 3, 4, 5])
print(d)
e=arr.array('d', [1.0, 2.0, 3.14])
print(e)
```

```
array('l')
array('u', 'hello  ')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

0.3.1 Arrays versus Lists

Why do you need arrays at all. They are different in terms of the operations one can perform on them. With arrays, you can perform an operations on all its item individually, which may not be the case with lists.

[ ]:
```
a = arr.array("u",["c","a","t","s"])
print(a)
```

```
array('u', 'cats')
```

[ ]:
```python
a.tostring()
print(a)
```

```
array('u', 'cats')
```

[ ]:
```python
l = ["c","a","t","s"]
s=''.join(l)
print(s)
```

```
cats
```

0.4 NumPy Arrays

See https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

[ ]:
```python
import numpy as np

a = np.array([3, 5, 7])
print(a)
b = a/3.0 # Performing vectorized (element-wise) operations
print(b)
```

```
[3 5 7]
[ 1.          1.66666667  2.33333333]
```

[ ]:
```python
o = np.ones(5)
print(o)
```

```
[ 1.  1.  1.  1.  1.]
```

[ ]:
```python
o = np.zeros(5)
print(o)
```

```
[ 0.  0.  0.  0.  0.]
```

[ ]:
```python
a = np.arange(15).reshape(3, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

[ ]:
```python
print(a.shape)
print(a.ndim)
print(a.dtype.name)
print(a.itemsize)
print(a.size)
print(type(a))
```

```
(3, 5)
2
```

```
int64
8
15
<class 'numpy.ndarray'>
```

[ ]:
```python
a*=5
print(a)
```

```
[[ 0  5 10 15 20]
 [25 30 35 40 45]
 [50 55 60 65 70]]
```

[ ]:
```python
a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

[ ]:
```python
b = np.array([(1.5,2,3), (4,5,6)])
print(b)
```

```
[[ 1.5  2.   3. ]
 [ 4.   5.   6. ]]
```

[ ]:
```python
c = np.array( [ [1,2], [3,4] ], dtype=complex )
print(c)
```

```
[[ 1.+0.j  2.+0.j]
 [ 3.+0.j  4.+0.j]]
```

[ ]:
```python
c=np.arange(100)
print(c)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99]
```

[ ]:
```python
c=c.reshape(10,10)
print(c)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```

[ ]:
```
a = np.array( [20,30,40,50] )
print(a)
b = np.arange( 4 )
print(b)
c=a-b
print(c)
```

```
[20 30 40 50]
[0 1 2 3]
[20 29 38 47]
```

[ ]:
```
b=np.sqrt(c)
print(b)
```

```
[ 4.47213595  5.38516481  6.164414    6.8556546 ]
```

[ ]:
```
a = np.sqrt(np.arange(10)**3)
print(a)
```

```
[  0.           1.           2.82842712   5.19615242   8.          11.18033989
  14.69693846  18.52025918  22.627417    27.        ]
```

[ ]:
```
print(a[2:5])
```

```
[ 2.82842712  5.19615242  8.        ]
```

[ ]:
```
def f(x,y):
  return 10*x+y
b = np.fromfunction(f,(5,4),dtype=int)
print(b)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
```

[ ]:
```
print(b[2,3])
print(b[0:5, 1])
print(b[1:3, : ] )
```

```
23
[ 1 11 21 31 41]
[[10 11 12 13]
 [20 21 22 23]]
```

[ ]:
```
print(b)
print(b.T)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
```

```
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
[[ 0 10 20 30 40]
 [ 1 11 21 31 41]
 [ 2 12 22 32 42]
 [ 3 13 23 33 43]]
```

[ ]:
```
for element in b.flat:
    print (element)
```

```
0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

0.4.1 Sets

Sets are a collection of distinct (unique) objects.

[ ]:
```
x = set('Cake&Cookie')
print(x)
y = set('Cookie')
print(y)
```

```
{'C', 'o', '&', 'i', 'a', 'e', 'k'}
{'C', 'o', 'i', 'e', 'k'}
```

[ ]:
```
print(x-y)
```

```
{'a', '&'}
```

0.5 Lists

[ ]:
```python
months = [
1.0,
'January',
'February',
'March',
'April',
'May',
'June',
'July',
'August',
'September',
'October',
'November',
'December'
]
print (type(months))
print (type(months[0]))
print (len(months))
# print (class(months))
print(months[0])
print(months[-1])
```

```
<class 'list'>
<class 'float'>
13
1.0
December
```

[ ]:
```python
print(months[len(months)-1])
```

```
December
```

[ ]:
```python
print(months)
del months[0]
print(months)
```

```
[1.0, 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
```

[ ]:
```python
print(months[9:11])
```

```
['October', 'November']
```

[ ]:
```python
print(months[9:])
```

```
['October', 'November', 'December']
```

[ ]:
```
print(months[:]) # Print everything
print(months[::2]) # Print every other - i.e. skip by 2
```

```
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December']
['January', 'March', 'May', 'July', 'September', 'November']
```

[ ]:
```
print(months[-1])
print(months[-3])
print(months[-3:-1])  # Print second to last to third to last
```

```
December
October
['October', 'November']
```

[ ]:
```
print(months[10:1:-3]) # Print 10 to 1 - i.e. skip by 3
```

```
['November', 'August', 'May']
```

[ ]:
```
print(months[10:1:-1])  # Print 10 to 1 - i.e. no skip but descending
```

```
['November', 'October', 'September', 'August', 'July', 'June', 'May', 'April',
'March']
```

[ ]:
```
print([1, 2, 3] + [4, 5, 6]) # join two lists
```

```
[1, 2, 3, 4, 5, 6]
```

[ ]:
```
print('Hello, ' + 'world!')  # concatenate two strings
```

```
Hello, world!
```

[ ]:
```
name='Bear' # Create string 'Bear'
print(name*5) # concatenate 5 times
```

```
BearBearBearBearBear
```

[ ]:
```
print('B' in name) # Test if B in Bear
print('b' in name)  # Test if b in Bear
```

```
True
False
```

[ ]:
```
s=range(1,6) # Range doesn't create lists in python 3
print(s)
```

```
range(1, 6)
```

[ ]:
```
s=list(range(1,9)) # Using range to create lists in python 3
print(s)
s[2]=5
print(s)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 5, 4, 5, 6, 7, 8]
```

[ ]:
```
del s[2] # Delete an item
print(s)
del s[2:4] # Delete more than one item
print(s)
```

```
[1, 2, 4, 5, 6, 7, 8]
[1, 2, 6, 7, 8]
```

[ ]:
```
t=[1, 2, 3, 4, 5] #  Create a list
print(t)
```

```
[1, 2, 3, 4, 5]
```

[ ]:
```
t=(1, 2, 3, 4, 5) #  Create a tuple
print(t)
```

```
(1, 2, 3, 4, 5)
```

[ ]:
```
# del t[2]
print(t)
```

```
(1, 2, 3, 4, 5)
```

[ ]:
```
print(2 ** 5) # Power
```

```
32
```

[ ]:
```
print(pow(2, 5)) # Power using function
```

```
32
```

[ ]:
```
import math # Some common math functions
print(math.ceil(33.3))
```

```
34
```

[ ]:
```
print(math.sqrt(9))
```

```
3.0
```

[ ]:
```
from math import sqrt
print(sqrt(9))
```

```
3.0
```

0.6 List Methods

- list.append(elem) – adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) – inserts the element at the given index, shifting elements to the right.

- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) – searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) – searches for the first instance of the given element and removes it (throws ValueError if not present)
- list.sort() – sorts the list in place (does not return it). (The sorted() function shown below is preferred.)
- list.reverse() – reverses the list in place (does not return it)
- list.pop(index) – removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

[ ]:
```python
l = ['Rick Sanchez',1.0, 'Morty Smith', 'Mr. Meeseeks']
print (l)
l.append('Doofus Rick')         ## append item at end
print (l)
l.insert(0, 'Scary Terry')        ## insert item at index 0
print (l)
l.extend(['Squanchy', 'Mr. Poopybutthole'])  ## add list of items at end
print (l)
print (l.index('Morty Smith'))
print (l)
l.remove('Morty Smith') ## search and remove an item
print (l)
print(l.pop(1)) ## removes and returns 'Rick Sanchez' second item
print (l)  ## ['Scary Terry', 'Mr. Meeseeks', 'Doofus Rick', 'Squanchy', 'Mr.␣
↪Poopybutthole']
```

```
['Rick Sanchez', 1.0, 'Morty Smith', 'Mr. Meeseeks']
['Rick Sanchez', 1.0, 'Morty Smith', 'Mr. Meeseeks', 'Doofus Rick']
['Scary Terry', 'Rick Sanchez', 1.0, 'Morty Smith', 'Mr. Meeseeks', 'Doofus
Rick']
['Scary Terry', 'Rick Sanchez', 1.0, 'Morty Smith', 'Mr. Meeseeks', 'Doofus
Rick', 'Squanchy', 'Mr. Poopybutthole']
3
['Scary Terry', 'Rick Sanchez', 1.0, 'Morty Smith', 'Mr. Meeseeks', 'Doofus
Rick', 'Squanchy', 'Mr. Poopybutthole']
['Scary Terry', 'Rick Sanchez', 1.0, 'Mr. Meeseeks', 'Doofus Rick', 'Squanchy',
'Mr. Poopybutthole']
Rick Sanchez
['Scary Terry', 1.0, 'Mr. Meeseeks', 'Doofus Rick', 'Squanchy', 'Mr.
Poopybutthole']
```

0.7 Dictionaries

[ ]:
```
d={"python": 333, "R": 222, 33: 111, "C++": 111} # Create a dictionary
print(d.keys())
```

```
dict_keys(['python', 'R', 33, 'C++'])
```

[ ]:
```
print(d["python"]) # List value with key "python"
```

```
333
```

[ ]:
```
print("java" in d) # Check if "java" in dictionary
```

```
False
```

[ ]:
```
print("python" in d) # Check if "python" in dictionary
```

```
True
```

[ ]:
```
d2={"java": 33, "C#": 22, "Scala": 11} # Create a dictionary
print(d2)
```

```
{'java': 33, 'C#': 22, 'Scala': 11}
```

[ ]:
```
d.update(d2) # add dictionary to another dictionary
print(d)
```

```
{'python': 333, 'R': 222, 33: 111, 'C++': 111, 'java': 33, 'C#': 22, 'Scala':
11}
```

[ ]:
```
for key in d: # List keys in dictionary
  print(key)
print(d.keys())
```

```
python
R
33
C++
java
C#
Scala
dict_keys(['python', 'R', 33, 'C++', 'java', 'C#', 'Scala'])
```

[ ]:
```
for key in d: # List values in dictionary
  print(d[key])
print(d.values())
```

```
333
222
111
111
33
```

```
22
11
dict_values([333, 222, 111, 111, 33, 22, 11])
```

[ ]:
```
d_backup = d.copy() # Create  dictionary copy
print(d_backup)
```

```
{'python': 333, 'R': 222, 33: 111, 'C++': 111, 'java': 33, 'C#': 22, 'Scala':
11}
```

[ ]:
```
d['C++']=55
print(d)
print(d_backup)
```

```
{'python': 333, 'R': 222, 33: 111, 'C++': 55, 'java': 33, 'C#': 22, 'Scala': 11}
{'python': 333, 'R': 222, 33: 111, 'C++': 111, 'java': 33, 'C#': 22, 'Scala':
11}
```

0.7.1 Note:

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

[ ]:
```
print(len(d)-len(d_backup))
del d_backup["java"]
print(d_backup)
```

```
0
{'python': 333, 'R': 222, 33: 111, 'C++': 111, 'C#': 22, 'Scala': 11}
```

[ ]:
```
print(len(d)-len(d_backup))  # Note original isn't changed
```

```
1
```

0.8 Dictionary Methods

- d.fromkeys() - Create a new dictonary with keys from seq and values set to value.
- d.get(key, default=None) - For any key, returns value or default if key not in dictonary
- d.has_key(key) - Removed, use the in operation instead.
- d.items() - Returns a list of d.s (key, value) tuple pairs
- d.keys() - Returns list of dictonary d's keys
- d.setdefault(key, default = None) - Similar to get(), but will set d.key] = default if key is not already in dict
- d.update(d2) - Adds dictonary d2's key-values pairs to dict
- d.values() - Returns list of dictonary d's values
- d.clear() - Removes all elements of dictonary d

Note:

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

0.9 Trees

See [https://link.springer.com/chapter/10.1007/978-3-319-13072-9_6][https://link.springer.com/chapter/10.1007/3-319-13072-9_6]

0.10 help(), and dir()

here are a variety of ways to get help for Python.

Do a Google search, starting with the word "python", like "python list" or "python string lowercase". The first hit is often the answer. This technique seems to work better for Python than it does for other languages for some reason. The official Python docs site — docs.python.org — has high quality docs. Nonetheless, I often find a Google search of a couple words to be quicker. There is also an official Tutor mailing list specifically designed for those who are new to Python and/or programming! Many questions (and answers) can be found on StackOverflow and Quora. Use the help() and dir() functions (see below). Inside the Python interpreter, the help() function pulls up documentation strings for various modules, functions, and methods. These doc strings are similar to Java's javadoc. The dir() function tells you what the attributes of an object are. Below are some ways to call help() and dir() from the interpreter:

help(len) — help string for the built-in len() function; note that it's "len" not "len()", which is a call to the function, which we don't want help(sys) — help string for the sys module (must do an import sys first) dir(sys) — dir() is like help() but just gives a quick list of its defined symbols, or "attributes" help(sys.exit) — help string for the exit() function in the sys module help('xyz'.split) — help string for the split() method for string objects. You can call help() with that object itself or an example of that object, plus its attribute. For example, calling help('xyz'.split) is the same as calling help(str.split). help(list) — help string for list objects dir(list) — displays list object attributes, including its methods help(list.append) — help string for the append() method for list objects

0.11 Python Tutorials

- ["Dive into Python" (Chapters 2 to 4)] (http://diveintopython.org/)
- [Python 101 – Beginning Python] (http://www.rexx.com/~dkuhlman/python_101/python_101.html)
- [Nice free CS/python book] (https://www.cs.hmc.edu/csforall/index.html)

0.11.1 Things to refer to

- [The Official Python Tutorial] (http://www.python.org/doc/current/tut/tut.html)
- [The Python Quick Reference] (http://rgruet.free.fr/PQR2.3.html)

0.11.2 YouTube Python Tutorials

- [Python Fundamentals Training – Classes] (http://www.youtube.com/watch?v=rKzZEtxIX14)

- [Python 2.7 Tutorial Derek Banas] (http://www.youtube.com/watch?v=UQi-L-_chcc)
- [Python Programming Tutorial - thenewboston] (http://www.youtube.com/watch?v=4Mf0h3HphEA)
- Google Python Class

0.12 License